طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل اول: مقدمه

حل مساله

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 📚

مقدمه 💡

در درس داده‌ساختارها و الگوریتم‌ها با تعریف کلی الگوریتم، محاسبه‌ی زمان اجرا و اثبات درستی الگوریتم آشنا شدیم و در این درس در ادامه‌ی همان مسیر به مسائل پیشرفته‌تر می‌پردازیم. راه حل این مسائل معمولا در چند دسته الگوریتم قرار می‌گیرند؛ یعنی روش‌هایی در طول چند دهه‌ی گذشته ابداع شده‌اند که برای حل این مسائل استفاده می‌شوند. این روش‌ها عبارتند از: برنامه‌نویسی پویا، حریصانه، تقسیم و حل و ...

البته بخش بزرگی از چالش‌های موجود در حل مسائل الگوریتمی، همین یافتن دسته‌بندی مناسب برای آن مساله و اعمال الگوریتمی است که بتواند آن را با صرف زمان و حافظه‌ی مناسب حل کند. شما در درس داده‌ساختارها با مفهوم تحلیل الگوریتم از نظر زمان و حافظه و مفاهیمی نظیر O و Θ آشنا شدید. در این بخش دو مساله را با هم مرور می‌کنیم که راه حل اولیه‌ای که در موردشان به ذهن می‌رسد از نظر زمانی بهینه نیست، و به مرور سعی می‌کنیم آن‌ها را گام‌به‌گام بهبود ببخشیم. این مسائل هم‌چنین در دسته‌بندی خاصی (از میان دسته‌های بالا) قرار نمی‌گیرند و بنابراین صرفا به کمک فکر کردن و ایده‌زنی حل می‌شوند.

مسئله‌ی اول: بزرگ‌ترین زیردنباله‌ی متوالی 📊

یک دنباله از اعداد حقیقی داده شده است. می‌خواهیم بین همه‌ی زیردنباله‌های متوالی آن، زیردنباله‌ای با بیشترین مجموع اعضا را بیابیم.

نکته: اکیدا توصیه می‌شود قبل از خواندن راه حل دقایقی خودتان به دنبال راه حل این مساله بگردید.

راه حل اول 💭

اولین راهی که برای حل این مسئله به ذهن می‌رسد، این است که همه‌ی زیردنباله‌های ممکن را درنظر بگیریم و اعضای آن را جمع بزنیم و آن زیردنباله‌ای که جمع آن بیشینه بود را به عنوان خروجی اعلام کنیم. می‌توانید کد این الگوریتم را در زیر ببینید.

In [4]:
```python
def max_subarray_sum(arr):
    n = len(arr)
    max_sum = float('-inf')  # مقدار اولیه برای بیشینه مجموع، منفی بی‌نهایت در نظر گرفته می‌شود

    # حلقه اول: انتخاب نقطه شروع زیردنباله
    for i in range(n):
        # حلقه دوم: انتخاب نقطه پایان زیردنباله
        for j in range(i, n):
            current_sum = 0
            # حلقه سوم: محاسبه مجموع عناصر بین i و j
            for k in range(i, j + 1):
                current_sum += arr[k]
            # بررسی و به‌روزرسانی بیشینه مجموع
            if current_sum > max_sum:
                max_sum = current_sum
    return max_sum

# مثال: یک آرایه نمونه
arr = [1, 2, 31, -3, 51, 12, -102, 1, 31, 1]
result = max_subarray_sum(arr)
print(result)
```

```
94
```

تحلیل زمانی این راه حل ⏱️

اگر این الگوریتم را تحلیل زمانی کنیم می‌بینیم که از $O(n^3)$ زمان می‌برد چرا که انتخاب دو اندیس برای ابتدا و انتهای این زیردنباله $\frac{n\times(n+1)}{2}$ عملیات نیاز دارد و پیمایش کردن عناصر بین این دو اندیس و جمع زدن آن‌ها هم به تعداد طول آن عملیات نیاز دارد که از $O(n)$ عملیات مصرف می‌کند. پس نهایتاً این الگوریتم از $O(n^3)$ زمان می‌برد. آیا راهی سریع‌تر برای حل این مسئله وجود دارد؟

راه حل دوم ⚙️

حال سعی می‌کنیم که راه حلی سریع‌تر ارائه دهیم. در راه حل اول، هر بار مجموع را مجدداً از اول محاسبه می‌کردیم و چنین کاری زمان زیادی از ما می‌گرفت. وقتی که می‌خواهیم جمع اعضای آرایه از اندیس iام تا jام را محاسبه کنیم، به جای جمع زدن همه‌ی این اعضا از اول، می‌توان به مجموع اعضای iام تا $j-1$ام، عضو jام را اضافه کرد.

پس برای حل مسئله، اندیس پایان را همیشه بعد از اندیس شروع در نظر می‌گیریم. حال هم‌زمان با پیمایش اندیس پایان روی اعضای آرایه، مجموع را نیز محاسبه کرده و با مقدار بیشینه‌ای که تا الآن به دست آمده مقایسه می‌کنیم و اگر مقدار حاصل بیشتر از مقدار قبل بود، مقدار جدید را جایگزین آن می‌کنیم.

می‌توانید کد این الگوریتم را در زیر ببینید.

In [5]:
```python
def max_subarray_sum(arr):
    n = len(arr)
    max_sum = float('-inf')  # مقدار اولیه برای بیشینه مجموع

    # حلقه اول: انتخاب نقطه شروع زیردنباله
    for i in range(n):
        current_sum = 0  # مجموع جاری را برای هر نقطه شروع صفر می‌کنیم
        # حلقه دوم: انتخاب نقطه پایان زیردنباله و محاسبه مجموع به صورت افزایشی
        for j in range(i, n):
            current_sum += arr[j]  # اضافه کردن عضو جدید به مجموع جاری
            # به‌روزرسانی بیشینه مجموع در صورت لزوم
            if current_sum > max_sum:
                max_sum = current_sum
    return max_sum

# مثال: یک آرایه نمونه
```

```
arr = [1, 2, 31, -3, 51, 12, -102, 1, 31, 1]
result = max_subarray_sum(arr)
print(result)
```

```
94
```

تحلیل زمانی این راه حل ⏱️

اگر این الگوریتم را تحلیل زمانی کنیم می‌بینیم که از $O(n^2)$ زمان می‌برد چرا که انتخاب دو اندیس برای ابتدا و انتهای این زیردنباله $\frac{n\times(n+1)}{2}$ عملیات نیاز دارد و از آن‌جایی که جمع زدن اعضا، الآن از $O(1)$ زمان می‌برد، پس نهایتاً این الگوریتم از $O(n^2)$ زمان می‌برد. آیا باز هم می‌توان الگوریتم سریع‌تری برای حل این مسئله ارائه کرد؟

راه حل سوم 🧠

از ابتدا شروع به پیمایش اعضای آرایه می‌کنیم. یک متغیر برای ذخیره‌سازی حاصل جمع در نظر می‌گیریم. به هر عضو جدید که می‌رسیم، مقدار این متغیر را با عضو جدید جمع زده و در همان متغیر قرار می‌دهیم. حال اگر مقدار متغیر مثبت بود، آن را با متغیر دیگری که حاصل جمع بیشینه تا الآن در آن نگهداری شده بود مقایسه می‌کنیم و اگر بیشتر از آن بود، مقدار آن متغیر را به‌روزرسانی می‌کنیم. اگر مقدار این متغیر منفی بود، آن‌گاه مقدار متغیر را ۰ کرده و به سراغ عضو بعدی آرایه می‌رویم. همین روند را تا پیمایش کامل اعضای آرایه طی می‌کنیم. نهایتاً مقدار عدد نوشته شده در متغیر دارای حاصل جمع بیشینه، جواب مسئله خواهد بود.

می‌توانید کد این الگوریتم را در زیر ببینید.

In [6]:
```
def max_subarray_sum(arr):
    max_sum = float('-inf')  # مقدار اولیه برای بیشینه مجموع
    current_sum = 0  # متغیر برای ذخیره مجموع جاری

    for num in arr:  # پیمایش اعضای آرایه
        if current_sum < 0:  # اگر مجموع جاری منفی شد، آن را صفر کن
            current_sum = 0
        current_sum += num  # اضافه کردن عضو فعلی به مجموع جاری
        if current_sum > max_sum:  # به‌روزرسانی بیشینه مجموع
            max_sum = current_sum
    return max_sum

# مثال: یک آرایه نمونه
arr = [1, 2, 31, -3, 51, 12, -102, 1, 31, 1]
result = max_subarray_sum(arr)
print(result)
```

```
94
```

تحلیل زمانی این راه حل ⏱️

اگر این الگوریتم را تحلیل زمانی کنیم می‌بینیم که از $O(n)$ زمان می‌برد چرا که کافی است فقط یک مرتبه تمام اعضای آرایه پیمایش شوند و در حین پیمایش تعدادی عملیات جمع و مقایسه و مقداردهی کردن که از $O(1)$ زمان می‌برند انجام می‌شود. پس نهایتاً الگوریتم از $O(n)$ خواهد بود.

بنابراین اکنون مساله‌ای را حل کردیم که در نگاه اول به نظر می‌رسید راه‌حلش از $O(n^3)$ است ولی با مقداری بررسی بیشتر موفق شدیم به راه‌حلی خطی برسیم که به مراتب بهتر از راه‌حل اولیه بود.

مساله‌ی دوم: سه-مجموع 🔢

یک مجموعه شامل n عدد حقیقی داده شده است. می‌خواهیم بدانیم که آیا سه عضو از این مجموعه وجود دارند که مجموع آن‌ها صفر شود؟

نکته: اکیدا توصیه می‌شود قبل از خواندن راه حل دقایقی خودتان به دنبال راه حل این مساله بگردید.

راه حل اول 💡

برای حل این مسئله اولین راه حلی که به ذهن می‌رسد این است که بر روی تمام اعضای مجموعه پیمایش کنیم و هر سه‌تایی از اعضا را درنظر گرفته و با یکدیگر جمع کنیم و اگر حاصل این جمع برابر با صفر شد آن‌گاه به یک جواب مطلوب رسیده‌ایم و اگر جمع هیچ سه‌تایی از اعضا برابر صفر نشد آن‌گاه این مسئله جواب ندارد.

می‌توانید کد این الگوریتم را در زیر ببینید.

In [7]:
```python
def has_three_sum_zero(arr):
    n = len(arr)
    # پیمایش روی تمام سه‌تایی‌های ممکن از اعضای مجموعه
    for i in range(n):
        for j in range(n):
            for k in range(n):
                # بررسی اینکه آیا سه عضو انتخابی متمایز هستند و مجموعشان صفر است
                if i != j and j != k and k != i and arr[i] + arr[j] + arr[k] == 0:
                    return "Yes"  # اگر پیدا شد، پاسخ "بله" برگردانده می‌شود
    return "No"  # اگر هیچ سه‌تایی پیدا نشد، پاسخ "خیر" برگردانده می‌شود

# مثال: یک آرایه نمونه
arr = [1, 2, 31, -3, 51, 12, -102, 1, 31, 1]
result = has_three_sum_zero(arr)
print(result)
```

```
Yes
```

تحلیل زمانی این راه حل ⏱️

اگر این الگوریتم را تحلیل زمانی کنیم، می‌بینیم که از $O(n^3)$ زمان می‌برد چرا که برای بررسی هر کدام از اعضا باید یک مرتبه آن‌ها را پیمایش کنیم پس مجموعاً نیاز به n^3 عملیات برای پیمایش اعضا داریم که از $O(n^3)$ زمان می‌برد.

راه حل دوم ⚡

اگر کمی بیشتر بررسی کنیم، متوجه می‌شویم که می‌توانیم همان راه حل اول را به گونه‌ای تغییر دهیم که به جای پیمایش همه‌ی اعضا، اعضا را به صورت تکراری پیمایش نکنیم. در واقع برای یافتن سه‌تایی‌ها، عضو اول سه‌تایی بر روی تمام اعضای مجموعه، عضو دوم سه‌تایی بر روی اعضای مجموعه از عضو اول به بعد و عضو سوم سه‌تایی هم روی اعضای مجموعه از عضو دوم به بعد پیمایش کنند.

می‌توانید کد این الگوریتم را در زیر ببینید.

In [8]:
```python
def has_three_sum_zero(arr):
    n = len(arr)
    # پیمایش روی تمام سه‌تایی‌های ممکن بدون تکرار
    for i in range(n):
        for j in range(i + 1, n):  # j شروع از عضو بعد از i
            for k in range(j + 1, n):  # k شروع از عضو بعد از j
                if arr[i] + arr[j] + arr[k] == 0:  # بررسی مجموع صفر
                    return "Yes"  # اگر پیدا شد، پاسخ "بله" برگردانده می‌شود
    return "No"  # اگر هیچ سه‌تایی پیدا نشد، پاسخ "خیر" برگردانده می‌شود

# مثال: یک آرایه نمونه
arr = [1, 2, 31, -3, 51, 12, -102, 1, 31, 1]
```

```
result = has_three_sum_zero(arr)
print(result)
```

```
Yes
```

تحلیل زمانی این راه حل ⏱️

اگر این الگوریتم را تحلیل زمانی کنیم، می‌بینیم که از $O(n^3)$ زمان می‌برد؛ چرا که تعداد عملیات‌های ما برابر با تعداد روش‌های انتخاب ۳ عدد بین ۱ تا n و قرار دادن آن‌ها با ترتیب ثابت است که دقیقاً $\frac{n\times(n-1)\times(n-2)}{6}$ عملیات برای پیمایش عناصر داریم که از $O(n^3)$ زمان می‌برد.

پس متوجه شدیم که با این تغییر هم مرتبه‌ی زمان بهتر نشد. آیا راه حلی با مرتبه‌ی زمانی کمتر وجود دارد؟

راه حل سوم 🧩

حال می‌خواهیم راه حلی ارائه دهیم که مرتبه‌ی زمانی کمتری داشته باشد. ابتدا اعضای مجموعه را مرتب می‌کنیم. برای یافتن عضو اول ۳تایی روی اعضای ۱ تا $n-1$ مجموعه پیمایش می‌کنیم. فرض می‌کنیم عضو دوم ۳تایی همیشه بعد از عضو اول و قبل از عضو سوم است. در ابتدا فرض می‌کنیم عضو دوم ۳تایی، عضو بعدی عضو اول در مجموعه و عضو سوم ۳تایی، آخرین عضو مجموعه است. حال در هر مرحله، این ۳ عضو را جمع می‌زنیم، اگر مقدار برابر ۰ شد این مسئله جواب دارد. اگر این جمع کمتر از ۰ بود، عضو دوم را عضو بعدی مقدار فعلی‌اش در مجموعه فرض می‌کنیم. اگر این جمع بزرگ‌تر از ۰ بود، عضو سوم را عضو قبلی مقدار فعلی‌اش در مجموعه فرض می‌کنیم. در نهایت اگر برای هیچ مقداری این جمع برابر با ۰ نشد پس این مسئله جواب نخواهد داشت.

می‌توانید کد این الگوریتم را در زیر ببینید.

In [9]:
```
def has_three_sum_zero(arr):
    n = len(arr)
    arr.sort()  # مرتب‌سازی آرایه

    for i in range(n - 1):  # پیمایش روی عضو اول سه‌تایی
        j = i + 1  # عضو دوم سه‌تایی
        k = n - 1  # عضو سوم سه‌تایی

        while j < k:  # هم‌پوشانی نداشته باشند k و j تا زمانی که
            current_sum = arr[i] + arr[j] + arr[k]  # محاسبه مجموع سه عضو
            if current_sum == 0:  # اگر مجموع صفر شد
                return "Yes"
            elif current_sum > 0:  # اگر مجموع بزرگتر از صفر بود، k را کاهش می‌دهیم
                k -= 1
            else:  # اگر مجموع کوچکتر از صفر بود، j را افزایش می‌دهیم
                j += 1
    return "No"  # اگر هیچ سه‌تایی پیدا نشد

# مثال: یک آرایه نمونه
arr = [1, 2, 31, -3, 51, 12, -102, 1, 31, 1]
result = has_three_sum_zero(arr)
print(result)
```

```
Yes
```

تحلیل زمانی این راه حل ⏱️

اگر این الگوریتم را تحلیل زمانی کنیم، می‌بینیم که از $O(n^2)$ زمان می‌برد؛ چرا که عضو اول بر روی $n-1$ مقدار متفاوت پیمایش می‌کند و عضو دوم و سوم هم هر دو با هم حداکثر بر روی $n-1$ مقدار متفاوت پیمایش می‌کنند. پس حداکثر $(n-1)^2$ عملیات برای پیمایش عناصر داریم که از $O(n^2)$ زمان می‌برد. بنابراین موفق شدیم بهبود قابل توجهی در زمان اجرای الگوریتم ایجاد کنیم.

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل دوم: حریصانه

بخش اول: زمان بندی، کوله پشتی کسری

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات

مقدمه

برای تعریف الگوریتم‌های حریصانه ابتدا با یک مثال شروع می‌کنیم که در آن حریصانه عمل کردن یا به عبارتی دیگر در هر مرحله بهترین کار را انجام دادن به ما جواب بهینه را در یک پروسه می‌دهد. سپس به معرفی اینکه چه دسته سوالاتی با استفاده از این نوع الگوریتم‌ها قابل حل هستند و چگونه می‌توان اثبات کرد حریصانه عمل کردن بهینه است، می‌پردازیم. در ادامه نیز ۳ دسته از مسائل معروف این بخش در طراحی الگوریتم را بحث می‌کنیم.

مسئله‌ی صفر: قرار دادن فایل‌های صوتی در یک نوار

مساله‌ای که با آن بحث را شروع می‌کنیم، مساله‌ی قرار دادن n فایل صوتی در یک نوار است. در این مساله، هر فایل صوتی با شماره‌ی i زمان $L[i]$ دارد و تضمین می‌شود که جمع زمان همه‌ی فایل‌ها برابر با ظرفیت نوار است. اگر $s(i)$ شماره‌ی فایلی باشد که در جایگاه i ام است، زمان دسترسی به فایلی در جایگاه i ام را این‌گونه تعریف می‌کنیم: $\sum_{k=1}^{i} L[s(k)]$. هدف پیدا کردن جایگشت s است به طوری که امید ریاضی دسترسی به یک فایل کمینه باشد. همچنین فرض بر این است که احتمال دسترسی به هر فایل $\frac{1}{n}$ است.

راه حل

ابتدا چیزی که سوال می‌خواهد، یعنی امید ریاضی دسترسی به یک فایل را تعریف می‌کنیم:

$$E[cost(i)] = \sum_{i=1}^{n} \frac{cost(i)}{n} = \sum_{i=1}^{n} \sum_{k=1}^{i} \frac{L[s(k)]}{n}$$

بنابراین هدف کمینه کردن مقدار $\sum_{i=1}^{n} \sum_{k=1}^{i} L[s(k)]$ است. حال در رویکرد حریصانه می‌گوییم فایل با کمترین هزینه یا زمان را در جایگاه اول، فایل با دومین کمترین هزینه را در جایگاه دوم و به همین ترتیب فایل با iامین کمترین هزینه را در جایگاه iام می‌گذاریم. یعنی در هر مرحله که قرار گرفتیم، فایل با کمترین حجم را انتخاب می‌کنیم.

حال باید نشان دهیم این رویکرد باعث می‌شود که مقدار اشاره شده کمینه شود. خوشبختانه در این مساله به لحاظ شهودی واضح است که رویکرد حریصانه به ما جواب بهینه را می‌دهد. اما اثبات آن به این شکل است که فرض خلف می‌کنیم، یعنی فرض می‌کنیم رویکرد حریصانه جواب بهینه را ندهد و الگوریتم بهینه‌ای وجود داشته باشد که جایگشت را به صورتی انجام می‌دهد که i ای وجود دارد که برای آن داریم:

$$L[s(i)] > L[s(i+1)]$$

حال باید نشان دهیم چنین الگوریتمی بهینه نیست و با جابه‌جا کردن مکان $s(i)$ و $s(i+1)$ جواب بهتری به دست می‌آید. اگر فایل‌های این دو مکان را با هم عوض کنیم اتفاقی که می‌افتد این است که هزینه برای جایگاه iام به اندازه‌ی $L[s(i)] - L[s(i+1)]$ کاهش می‌یابد و هزینه برای جایگاه $i+1$ام تغییری نمی‌کند. در نتیجه با جابه‌جا کردن این دو فایل به جواب بهتری رسیدیم، پس الگوریتمی که ابتدا فرض کرده بودیم بهینه است، بهینه نیست و از این نتیجه می‌گیریم که هیچ الگوریتم بهینه‌ای وجود ندارد که در آن i ای باشد که برای آن داشته باشیم $L[s(i)] > L[s(i+1)]$. پس الگوریتم حریصانه‌ی ما تنها الگوریتمی است که این ویژگی را ندارد و با تغییر دادن جایگشت در آن همیشه جواب بدتر می‌شود. در نتیجه حریصانه در این جا رویکردی است که به ما جواب بهینه را می‌دهد.

تحلیل زمانی این راه حل ⏱️

مرتبه زمانی این راه حل $O(n \log n)$ است زیرا در ابتدا باید فایل‌ها را بر اساس مدت زمانشان مرتب کنیم که این کار از مرتبه زمانی $O(n \log n)$ است و سپس هر فایل را در جایگاه خود قرار دهیم.

چگونه اثبات کنیم حریصانه بهینه است؟ ✅

حل مساله به روش حریصانه زمانی کامل می‌شود که پس از ارائه روش حریصانه، اثبات کنیم این روش بهینه است. به صورت کلی برای اثبات بهینه بودن یک روش حریصانه می‌توان به این صورت عمل کرد: هر روش یا الگوریتمی را در نظر گرفت، مثلاً در مساله‌ای که در بالا مطرح شده، هر جایگشتی را برای فایل‌ها در نظر گرفت. سپس این جایگشت را به نوعی عوض کرد تا هزینه کاهش یابد. این تغییرات را تا جایی اعمال کرد که دیگر تغییری وجود نداشته باشد که هزینه را کاهش دهد و نشان داد که جایگشت نهایی همان جایگشت حریصانه ماست.

شکل دیگری نیز می‌توان به مساله اثبات بهینه بودن روش حریصانه نگاه کرد: به این صورت که در همان مساله بالا، هر جایگشت دیگری به جز جایگشت حریصانه را در نظر گرفت و نشان داد هیچ‌کدام از آن‌ها بهینه نیست. بنابراین، بهینه‌ای وجود دارد که روش حریصانه است؛ زیرا هر کدام از جایگشت‌های دیگر می‌توانند بهبود پیدا کنند و یک جایگشت باید باشد که بهبود پیدا نکند.

این در حالی است که نشان دادن اینکه روش حریصانه بهینه نیست، کاری آسان‌تر بوده و فقط به مثال نقض نیاز دارد. در ادامه با مسائلی که حریصانه راه بهینه یا نابهینه برای آن‌ها ارائه می‌دهد، بیشتر آشنا می‌شویم.

مسئله‌ی اول: زمان‌بندی ⏰

فرض کنید n تسک داریم که هرکدام یک زمان شروع و یک زمان پایان دارند. می‌خواهیم از این n تسک بیشترین تعداد را انتخاب کنیم به طوری که هیچ دو کاری همپوشانی نداشته باشند.

در روش برنامه‌نویسی پویا مشاهده خواهید کرد که مساله با مرتبه زمانی $O(n^2)$ قابل حل است و نوع آن هم به صورت بازگشتی خواهد بود؛ یعنی ابتدا فرض می‌کنیم اگر تسک اول در زیرمجموعه بهینه وجود دارد، هر تسکی که با تسک اول تلاقی دارد وجود ندارد و به ۲ مجموعه می‌رسیم: مجموعه اول شامل تسک‌هایی که زمان پایانشان از زمان شروع تسک اول زودتر بوده و مجموعه دوم، تسک‌هایی که زمان شروعشان از زمان پایان تسک اول دیرتر بوده و مساله تبدیل می‌شود به این ۲ زیرمساله. اگر هم تسک اول در زیرمجموعه بهینه نیست، این روند را برای تسک‌های ۲ تا n تکرار می‌کنیم.

رویکرد حریصانه اول ⚡

با توجه به اینکه صحبت در ارتباط با جزئیات راه‌حل بالا در این بخش نمی‌گنجد، به سراغ حل مساله به روش حریصانه می‌رویم. پس باید معیاری مشخص کنیم که براساس آن تسک‌ها را مرتب کرده و انتخاب کنیم.

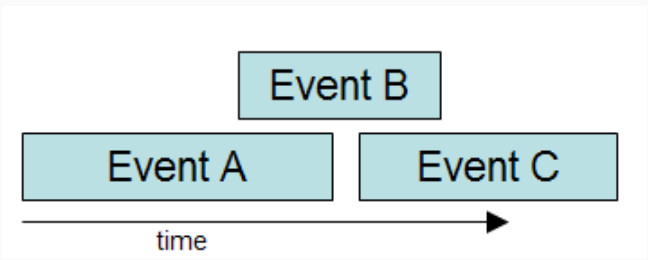
فرض کنید معیار را زمان شروع تسک‌ها قرار دهیم. یعنی تسک‌ها را براساس زمان شروعشان مرتب کرده و به این صورت عمل کنیم که کاری که زودتر از همه آغاز شده را ابتدا انتخاب کنیم و سپس همه کارهایی که با این کار تداخل دارند را حذف کرده و از بین کارهای باقی‌مانده همین پروسه را انجام می‌دهیم تا هیچ کاری باقی نماند.

آیا با این رویکرد بیشترین تعداد تسک را انتخاب خواهیم کرد؟ خیر. فرض کنید کاری که زودتر از همه شروع شده دیرتر از همه تمام شود؛ بنابراین تنها یک کار انتخاب خواهد شد در حالی که ممکن است بتوانیم چند کار با مدت زمان کمتر را همزمان انتخاب کنیم. بنابراین روش حریصانه بر اساس زمان شروع بهینه نیست و این مساله را با مثال نقض نشان دادیم.

رویکرد حریصانه دوم 🏹

کارها را بر اساس طول مرتب می‌کنیم و سپس از کوچک‌ترین شروع می‌کنیم و آن را انتخاب کرده و کارهایی که با آن تلاقی دارد را حذف می‌کنیم. همین کار را برای کارهای باقی‌مانده انجام می‌دهیم تا زمانی که کاری باقی نماند.

این روش هم روش بهینه نیست؛ زیرا ممکن است کار با کمترین طول با همه کارهای دیگر تلاقی داشته باشد در حالی که بتوان چند کار با طول بیشتر را همزمان انتخاب کرد.



shortest first greedy

در شکل بالا می‌بینیم که طبق الگوریتم ما event B انتخاب خواهد شد در حالی که جواب بهینه event A و event C می‌باشد.

رویکرد حریصانه سوم 🎯

براساس زمان پایان تسک‌ها را مرتب کرده و ابتدا تسکی را انتخاب می‌کنیم که زودتر از همه پایان یابد. سپس تسک‌هایی که با این تسک تداخل دارند را حذف کرده و این کار را برای تسک‌های باقی‌مانده انجام می‌دهیم. این روش بر خلاف ۲ روش پیشین بهینه بوده و باید آن را اثبات کنیم.

فرض می‌کنیم روش بهینه دیگری برای انتخاب تسک‌ها وجود داشته باشد که در آن روش تسک‌های $h_1, h_2, h_3, \dots, h_m$ به ترتیب به عنوان زیرمجموعه بهینه انتخاب شده است. همچنین زیرمجموعه‌ای که توسط الگوریتم حریصانه ما انتخاب شده است به ترتیب $g_1, g_2, g_3, \dots, g_k$ است. حال می‌خواهیم زیرمجموعه بهینه را به زیرمجموعه حریصانه خودمان تبدیل کنیم.

عضو اول هر دو مجموعه را در نظر می‌گیریم، یعنی h_1 و g_1. می‌گوییم با توجه به اینکه g_1 توسط الگوریتم حریصانه انتخاب شده است، زمان پایانش کوچک‌تر مساوی زمان پایان h_1 است و چون h_1 عضو زیرمجموعه بهینه است، پس با سایر اعضای زیرمجموعه هم‌پوشانی ندارد. از این ۲ گزاره نتیجه می‌شود که می‌توان h_1 را حذف و به جای آن g_1 گذاشت. زیرمجموعه ساخته شده همچنان بهینه است؛ زیرا هم‌پوشانی نداشته و سایز آن هم عوض نشده است.

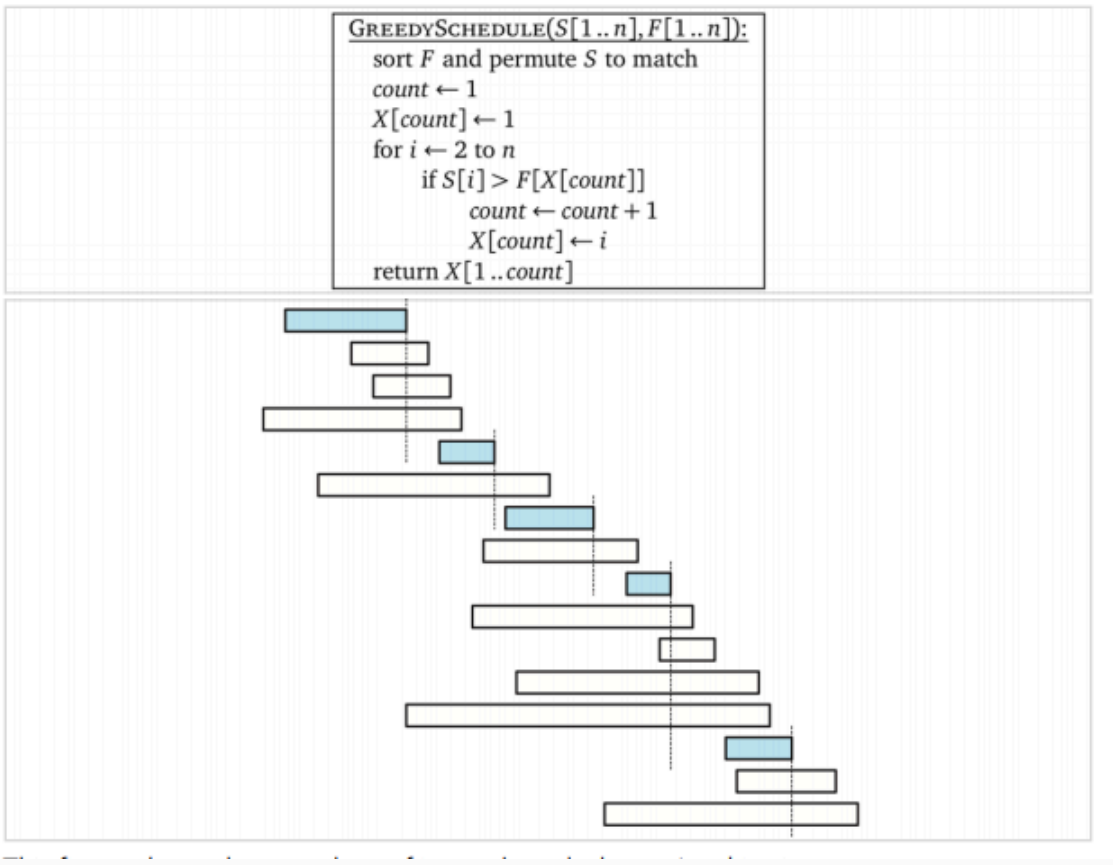
بنابراین نشان دادیم هر زیرمجموعه بهینه‌ای قابل تبدیل به زیرمجموعه‌ای است که عضو اولش همان عضوی باشد که اولین انتخاب الگوریتم حریصانه است. حال یک حالت کلی را برای این دو زیرمجموعه بررسی می‌کنیم که تا عضو اولشان دقیقاً یکی است:

$$g_1, g_2, \dots, g_i, g_{i+1}, \dots, g_k$$

$$g_1, g_2, \dots, g_i, h_{i+1}, \dots, h_k, \dots, h_m$$

چیزی که تا به اینجا اثبات کردیم برای $i = 1$ است. می‌خواهیم حال برای $i = 2$ تا $i = k$ اثبات کنیم. با توجه به اینکه g_{i+1} زودتر از h_{i+1} تمام می‌شود، پس با همه اعضای بعد از h_{i+1} تداخلی ندارد و چون توسط الگوریتم حریصانه ما انتخاب شده، با همه اعضای قبل از h_{i+1} نیز تداخل ندارد؛ پس می‌توان به جای h_{i+1}، g_{i+1} را در زیرمجموعه بهینه قرار داد تا زمانی که i برابر با k می‌شود.

حال ادعا می‌کنیم m باید حتماً برابر با k باشد؛ زیرا اگر هنوز مجموعه‌ای باشد که دیرتر از g_k تمام شود و با آن تداخل نداشته باشد، الگوریتم حریصانه آن را انتخاب می‌کند و چون حریصانه دیگر عضوی انتخاب نکرده است، پس چنین عضوی وجود ندارد. بنابراین انتخاب بهینه به انتخاب ما توسط الگوریتم حریصانه تبدیل شد.



finishes first greedy

تحلیل زمانی راه‌حل سوم ⏱️

مرتبه زمانی این راه‌حل $O(n \log n)$ است. ابتدا هر دو آرایه را مرتب می‌کنیم. در هر مرحله، عضوی که کمترین زمان پایان دارد را انتخاب می‌کنیم و هر عضوی که زمان شروع‌اش کمتر از زمان پایان عضو انتخاب شده است را خط می‌زنیم. سپس همین کار را تکرار می‌کنیم. چون هر عضو دقیقاً یکبار یا خط خورده است یا انتخاب شده است، روند انتخاب اعضا از $O(n)$ است. بنابراین، در نهایت مرتبه زمانی کل برابر با $O(n \log n)$ می‌شود.

در زیر کد این الگوریتم را مشاهده می‌کنید:

In [2]:
```python
class Task:
    def __init__(self, start, finish):
        self.start = start
        self.finish = finish

def print_max_tasks(tasks):
    # مرتب‌سازی تسک‌ها بر اساس زمان پایان
    tasks.sort(key=lambda x: x.finish)

    selected_tasks = []
    i = 0
    selected_tasks.append(tasks[i])  # انتخاب اولین تسک با کمترین زمان پایان

    for j in range(1, len(tasks)):
        # اگر زمان شروع تسک بعدی بعد از زمان پایان تسک انتخاب شده بود
        if tasks[j].start >= tasks[i].finish:
            selected_tasks.append(tasks[j])
            i = j  # به‌روزرسانی تسک انتخاب شده

    # چاپ تسک‌های انتخاب شده
    for task in selected_tasks:
        print(f"({task.start}, {task.finish}), ", end="")

# مثال: لیستی از تسک‌ها
tasks = [
    Task(5, 9),
    Task(1, 2),
    Task(3, 4),
    Task(0, 6),
    Task(5, 7),
    Task(8, 9)
]

print_max_tasks(tasks)
```

```
(1, 2), (3, 4), (5, 7), (8, 9),
```

مسئله‌ی دوم: خرد کردن پول 💰

فرض کنید تعداد نامتناهی از اسکناس‌های $c_1, c_2, \ldots, c_n$ تومانی داریم و می‌خواهیم با برداشتن تعدادی از هر اسکناس، به یک مجموع خاص داده شده **A** برسیم. مثلا فرض کنید به یک مغازه رفته‌اید و ۲۴ هزار تومان خرید کرده‌اید. حال اگر شما اسکناس ۵۰ هزار تومانی به فروشنده بدهید، فروشنده چگونه می‌تواند ۲۶ هزار تومان باقی مانده پول شما را بدهد؟ این مسئله از شما می‌خواهد تا با کمترین تعداد اسکناس ممکن این کار را انجام دهید. در بخش‌های بعد، الگوریتم پویا برای این مسئله ارائه خواهد شد اما در اینجا می‌خواهیم حل کردن آن به روش حریصانه را بررسی کنیم. در حالت حریصانه فرض می‌کنیم همواره $c_1 = 1$ است.

رویکرد حریصانه ⚙️

اولین اسکناسی را که نابیشتر از A است را برمی‌داریم و سپس آن را از A کم کرده و همین کار را مجدد انجام می‌دهیم. این کار را تا جایی ادامه می‌دهیم که مقدار A برابر صفر شود. به نظر شما این الگوریتم جواب بهینه را به ما نمی‌دهد؟
واضح است که این الگوریتم جواب بهینه را به ما می‌دهد. این مثال را درنظر بگیرید. فرض کنید با اسکناس‌های ۱، ۷ و ۱۰ هزار تومانی، مقدار ۱۴ هزار تومان را درست کنیم. با این الگوریتم ارائه شده در بالا، ابتدا ما یک اسکناس ۱۰ هزار تومانی برمی‌داریم. سپس ۴ اسکناس ۴ هزار تومانی برداشته و به مجموع دلخواه می‌رسیم. بدین ترتیب با برداشتن ۵ اسکناس به نتیجه رسیدیم درحالی که واضح است که با برداشتن دو اسکناس ۷ هزار تومانی، با تعداد کمتری اسکناس به مجموع مورد نظر می‌رسیدیم.
مثال بالا نشان داد که این الگوریتم با وجود آن‌که قطعا جواب دارد و احتمالا جواب آن به جواب بهینه نزدیک است، جواب بهینه را به ما نمی‌دهد.
می‌توان با بررسی مشاهده کرد که روش حریصانه دیگری برای این مسئله نمی‌توان ارائه داد. درواقع تا به امروز روش حریصانه‌ای برای حل این مسئله وجود ندارد.

حالت خاص 🧩

فرض کنید اسکناس‌های موجود ما برابر ۱، ۲ ،۵ و ۱۰ هزار تومانی باشد. ادعا می‌کنیم برای این مثال الگوریتم حریصانه جواب بهینه را به ما می‌دهد.
لم ۱: هر جواب بهینه‌ای کمتر مساوی ۱ اسکناس هزار تومانی، دو اسکناس دوهزارتومانی، یک اسکناس پنج‌هزارتومانی دارد. این تعداد را حد نصاب یک اسکناس می‌نامیم. درستی این لم واضح است چرا که درصورتی که بیشتر از این تعداد از هر اسکناس داشته‌باشیم، می‌توانیم با تعداد کمتری اسکناس و با استفاده از اسکناس بزرگتر، مجموع مورد نظر را بسازیم.
حال اثبات مسئله اصلی کار ساده‌ای شد. فرض کنید می‌خواهیم A هزار تومان را خرد کنیم. اولین اسکناس نابیشتر از A را x می‌نامیم. می‌دانیم قطعا در الگوریتم بهینه باید x برداشته شود. دلیل این امر آن است که اگر این کار را نکنیم مجبور می‌شویم تعداد بیشتر از حد نصاب اسکناس‌های با ارزش پایین‌تر برداریم. (دقت شود که مجموع <حدنصاب × ارزش اسکناس> از اسکناس اول تا هر اسکناسی، از اسکناس بعدی کمتر است.) بنابراین با توجه به لم ۱ مجبور می‌شویم اسکناس x را برداشته و سپس سراغ حل برای $A - x$ برویم. این مراحل دقیقا همان الگوریتم گریدی ارائه شده است! پس الگوریتم ما جواب بهینه را می‌دهد.
سوال: در چه حالاتی و با چه شرایطی الگوریتم حریصانه برای این مسئله جواب بهینه را می‌دهد؟
در زیر الگوریتم حریصانه را برای این مسئله مشاهده می‌کنید:

In [3]:
```python
def find_min_coins(V, denominations):
    # مرتب‌سازی اسکناس‌ها به ترتیب نزولی
    denominations.sort(reverse=True)

    ans = []  # لیست برای ذخیره اسکناس‌های انتخاب شده
    i = 0  # شروع از بزرگترین اسکناس

    while V > 0 and i < len(denominations):
        if V >= denominations[i]:  # اگر اسکناس فعلی کوچکتر یا مساوی مقدار باقیمانده است
            V -= denominations[i]  # کم کردن ارزش اسکناس از مقدار باقیمانده
```

```python
            ans.append(denominations[i])  # اضافه کردن اسکناس به لیست پاسخ
        else:
            i += 1  # رفتن به اسکناس کوچکتر بعدی

    return ans

# مثال: اسکناس‌های موجود و مقدار مورد نظر
denominations = [1, 2, 5, 10, 20, 50, 100, 500, 1000]
total_value = 93

result = find_min_coins(total_value, denominations)
print(f"Following is minimal number of change for {total_value}: {' '.join(map(str, result))}")
```

```
Following is minimal number of change for 93: 50 20 20 2 1
```

مسئله‌ی سوم: کوله‌پشتی کسری 🎒

فرض کنید **n** شی با ارزش‌های $v_1, v_2, \ldots, v_n$ و وزن‌های $w_1, w_2, \ldots, w_n$ داریم و یک کوله‌پشتی داریم که نهایتاً می‌تواند وزن W را تحمل کند. حال می‌خواهیم کوله‌پشتی را با اشیای موجود به نحوی پر کنیم که اگر در آن x_i ضریبی است که از شی **i**ام برداشته‌ایم،

$$\forall i : 0 \leqslant x_i \leqslant 1$$

$$\sum_{i=1}^{n} x_i w_i \leqslant W$$

و مقدار

$$\sum_{i=1}^{n} x_i v_i$$

بیشینه شود. این مسئله به **مسئله‌ی کوله‌پشتی کسری (Fractional Knapsack)** معروف است. همین مسئله را اگر تغییر دهیم به نحوی که نتوان کسری از هر شی را برداشت و مجبور باشیم هر شی را یا کامل برداریم یا نه، به مسئله‌ای بسیار سخت تبدیل می‌شود. در قسمت‌های بعدی درس خواهید دید که این مسئله که به **کوله‌پشتی صفر و یک (Knapsack 1-0)** معروف است، یک مسئله‌ی **NP-hard** می‌باشد.

رویکرد حریصانه اول ⚙️

در این روش شی دارای بیشترین ارزش را برمی‌داریم. سپس بین اشیای باقی‌مانده همین عمل را تکرار می‌کنیم تا گنجایش کوله‌پشتی تمام شود. شی آخر ممکن است کامل جا نشود و تکه‌ای از آن را در کوله‌پشتی بگذاریم. این روش به وضوح نابهینه است.

فرض کنید چهار شی با وزن‌های ۶۰، ۵۰، ۱۰ و ۴۰ و ارزش‌های ۳۰، ۸۰، ۲۰ و ۱۰ داریم. گنجایش کوله‌پشتی‌مان هم ۱۰۰ واحد است. با این روش، شی دوم و $\frac{۵}{۶}$ از شی اول را برمی‌داریم و کوله‌پشتی‌مان پر می‌شود. با این روش ۱۰۵ واحد ارزش در کوله‌پشتی گذاشته‌ایم، در حالی که با یک بررسی ساده و همچنین در رویکرد سوم خواهید دید که بیشتر از این مقدار نیز قابل برداشتن است.

رویکرد حریصانه دوم ⚙️

در این روش شی با کمترین وزن را برمی‌داریم. سپس بین اشیای باقی‌مانده همین عمل را تکرار می‌کنیم تا گنجایش کوله‌پشتی‌مان پر شود. در این حالت هم ممکن است شی آخر کامل در کوله‌پشتی جا نشده و بخشی از آن را در کوله‌پشتی قرار دهیم. این رویکرد نیز واضح است که جواب بهینه را به ما نمی‌دهد.

همان مثال رویکرد اول را در نظر بگیرید. اگر آن مثال را با این رویکرد حل کنیم، ابتدا شی دوم و سپس به ترتیب شی چهارم و دوم را برمی‌داریم و کوله‌پشتی‌مان پر می‌شود. در این رویکرد در مجموع ۱۱۰ واحد ارزش برداشته‌ایم در حالی که بیشتر از این مقدار نیز قابل برداشتن است.

رویکرد حریصانه سوم 🧩

در این روش چگالی یک شی را به‌صورت $\frac{value}{weight}$ تعریف می‌کنیم. حال در هر مرحله بین اشیای باقی‌مانده، شی با بیشترین چگالی را برمی‌داریم. این عمل را تا جایی ادامه می‌دهیم که کوله‌پشتی‌مان پر شود. واضح است که در این روش هم ممکن است شی آخر کامل جا نشده و کسری از آن را برداریم. این روش جواب بهینه را به ما می‌دهد. در ادامه به اثبات بهینه بودن آن می‌پردازیم و درنهایت آن را از نظر مدت زمان اجرا تحلیل می‌کنیم.

برای اثبات این رویکرد، فرض می‌کنیم اگر اشیا برحسب چگالی مرتب شده باشند، رویکرد ما x_i و رویکرد دیگری که بهینه است، x_i^* از شی **i**ام برمی‌دارد. شی با بیشترین چگالی را در نظر می‌گیریم و آن را **k** می‌نامیم. بدون این شی، کوله‌پشتی در هر دو حالت مقدار یکسانی گنجایش دارد.

دو حالت داریم:

۱. الگوریتم بهینه به همان اندازه ما از آن شی برداشته است. اگر شی دیگری باقی نمانده بود، نتیجه می‌شود محتوای دو کوله‌پشتی با هم برابر است و در نتیجه روش ما بهینه است. در صورتی که شی دیگری موجود بود، همین مراحل را برای شی بعدی با بیشترین چگالی تکرار می‌کنیم.

۲. الگوریتم بهینه مقدار کمتری از آن شی برداشته است. (واضح است که امکان ندارد مقدار بیشتری برداشته باشد، چرا که در الگوریتم ما تا جایی که جا داشتیم، شی با بیشترین چگالی را برمی‌داشتیم.) حال با توجه به اینکه بدون این شی کوله‌پشتی‌ها جای خالی یکسانی دارند، می‌توان مقدار $(x_k - x_k^*)$ از این شی به کوله‌پشتی بهینه اضافه کرد و از بقیه موارد (که چگالی کمتری دارند) برداشت.
واضح است که این کار مقدار ارزش کوله را کمتر نمی‌کند. در صورتی که چگالی جسمی که برداشته شده با گذاشته شده برابر باشد، ارزش کوله تغییری نمی‌کند و در غیر این صورت کمتر می‌شود. اگر ارزش کوله کمتر شود، با بهینه بودن کوله‌پشتی بهینه به تناقض می‌خوریم، و در صورتی که ارزش آن تغییری نکند، یک مرحله به کوله‌پشتی ما شبیه‌تر شده است و همین مراحل را برای شی بعدی با بیشترین چگالی تکرار می‌کنیم.

بدین ترتیب کوله‌پشتی بهینه بدون تغییر ارزش به کوله‌پشتی ما تبدیل شد. بنابراین کوله‌پشتی ما نیز بهینه بوده‌است و این رویکرد، جواب بهینه را به ما می‌دهد.

تحلیل زمانی رویکرد سوم ⏱️

مرتبه زمانی این راه‌حل $O(n \log n)$ است. ابتدا اشیا را برحسب چگالی‌شان مرتب می‌کنیم. در هر مرحله عضوی که بیشترین چگالی را دارد انتخاب می‌کنیم و سپس سراغ شی بعدی می‌رویم تا جایی که کوله‌پشتی‌مان پر شود. روند انتخاب اشیا از $O(n)$ است. در نتیجه مرتبه زمانی کل برابر با همان $O(n \log n)$ که زمان مرتب‌سازی اشیا بود، می‌باشد. در زیر کد این الگوریتم را مشاهده می‌کنید:

In [4]:
```python
class Item:
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight

def fractional_knapsack(W, items):
    # مرتب‌سازی اشیا بر اساس چگالی (ارزش به وزن) به صورت نزولی
    items.sort(key=lambda x: x.value / x.weight, reverse=True)

    current_weight = 0  # وزن فعلی کوله‌پشتی
    final_value = 0.0  # ارزش نهایی کوله‌پشتی

    for item in items:
        if current_weight + item.weight <= W:  # اگر شی کامل جا می‌شود
            current_weight += item.weight
            final_value += item.value
        else:  # اگر شی کامل جا نمی‌شود، بخشی از آن را برمی‌داریم
            remaining_weight = W - current_weight
            final_value += item.value * (remaining_weight / item.weight)
            break  # کوله‌پشتی پر شده است

    return final_value

# مثال: لیستی از اشیا و وزن حداکثر کوله‌پشتی
items = [Item(60, 10), Item(100, 20), Item(120, 30)]
W = 50

max_value = fractional_knapsack(W, items)
print(f"Maximum value we can obtain = {max_value}")
```

```
Maximum value we can obtain = 240.0
```

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

ترم دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل دوم: الگوریتمهای حریصانه

بخش دوم: فشرده سازی و کد هافمن

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 📑

مقدمه 🏁

کد کردن فایل‌ها 💾

می‌دانیم کامپیوترها به‌طور کلی از پردازش زبان‌های طبیعی عاجز هستند. هرچند در حال حاضر پیشرفت‌های چشمگیری در این زمینه صورت گرفته است، اما هنوز هم زبان اصلی مورد فهم کامپیوترها، زبان **صفر و یک** است. به همین علت، ذخیره‌سازی فایل‌های متنی به این صورت انجام می‌پذیرد؛ یعنی هر کاراکتر موجود در هر رشته به‌صورت یک **کد دودویی** (با طول دلخواه) از صفر و یک **انکد** می‌شود، پردازش روی آن انجام می‌گیرد و پس از آن با یک **دیکدینگ ساده** به حالت اولیه‌ی خود بازمی‌گردد.

کد کردن با طول ثابت 🔢

در الفبای انگلیسی ۲۶ حرف داریم؛ بنابراین با **۵ بیت** می‌توانیم همه‌ی حروف را کد کنیم. به این صورت که کاراکتر **'a'** به رشته‌ی باینری `00000` انکد می‌شود، کاراکتر **'b'** به رشته‌ی باینری `00001` ، ... و کاراکتر **'z'** به رشته‌ی باینری `11010` انکد می‌شود. می‌بینیم که همه‌ی کاراکترها به رشته‌های باینری با طول ثابت ۵ بیت انکد شده‌اند. به این نوع کد کردن فایل‌ها، **کد کردن با طول ثابت (Fixed-Length Encoding)** گفته می‌شود.

برای خواندن رشته‌ی دلخواه `0000000001` چه کاری انجام می‌دهیم؟
از آنجا که کدگذاری با طول ثابت است، هر ۵ بیت را جدا کرده و اقدام به دیکد کردن می‌کنیم. ۵ بیت اول کاراکتر **'a'** و ۵ بیت دوم کاراکتر **'b'** را تشکیل می‌دهند. در نتیجه رشته‌ی ما برابر با **'ab'** است.

کد کردن با طول متغیر ⚙️

پس از مدتی مشخص شد که در **کد کردن با طول ثابت**، حجم زیادی از فضای ذخیره‌سازی به هدر می‌رود. برای مثال، کاراکتر **'a'** می‌تواند تنها با یک `0` انکد شود و کاراکتر **'c'** با `10` . اما این کار موجب بروز **ابهام در فرایند دیکد کردن** می‌شود، چرا که ممکن است نتوان به‌راحتی تشخیص داد مرز بین کدها کجاست.

برای رفع این مشکل، **الگوریتم‌هایی برای کدگذاری با طول متغیر (Variable-Length Encoding)** طراحی شده‌اند که ضمن کاهش حجم داده، قابلیت دیکد کردن بدون ابهام را نیز فراهم می‌کنند. در ادامه به بررسی این الگوریتم‌ها خواهیم پرداخت.

کد هافمن 🔢

کد هافمن (Huffman Coding) یک **الگوریتم فشرده‌سازی داده** است که بر پایه‌ی **الگوریتم حریصانه** عمل می‌کند. ایده‌ی اصلی آن بر این اساس است که **کاراکترهایی که بیشتر تکرار می‌شوند، باید با کدهای کوتاه‌تر** و **کاراکترهایی که کمتر تکرار می‌شوند، با کدهای بلندتر** نمایش داده شوند.

می‌دانیم که محتوای هر فایل متنی در کامپیوتر به‌صورت **کدهای باینری** ذخیره می‌شود و معمولاً هر کاراکتر با رشته‌ای از صفر و یک نمایش داده می‌شود. در روش‌های ساده، این رشته‌ها طول ثابتی دارند.

برای مثال، می‌توان کاراکتر `"a"` را با `000` و کاراکتر `"b"` را با `001` کدگذاری کرد. در این حالت، هر دو کاراکتر دارای **کد باینری به طول ثابت** هستند:

```
a → 000
b → 001
```

در ادامه خواهیم دید که چگونه با استفاده از **الگوریتم هافمن** می‌توان با **کدهایی با طول متغیر**، حجم فایل را به‌طور قابل‌توجهی کاهش داد.

نمودار مربوط به ساختار درخت هافمن 📊

کاراکتر	a	b	c
کد	000	001	010
طول	3	3	3

برخلاف الگوریتم‌های کدگذاری متن معمولی، **الگوریتم هافمن** از کدگذاری با طول ثابت استفاده نمی‌کند. الگوریتم با استفاده از **فرکانس تکرار کاراکترها** به هر کاراکتر یک کد باینری اختصاص می‌دهد، به گونه‌ای که:

- کاراکتر با بیشترین تکرار، کوتاه‌ترین طول کد باینری را دارد.
- کاراکتر با کمترین فرکانس تکرار، بیشترین طول عدد کد شده باینری را داراست.

این روش باعث می‌شود حجم نهایی داده‌ها کاهش یابد و کدگذاری بهینه صورت گیرد.

📊 مثال دیکدینگ هافمن

کاراکتر	a	b	c
کد	000	101	1101
طول	1	3	4

از آنجا که کاراکترها با **فرکانس تکرار بیشتر** طول کمتری دارند، ذخیره‌ی آن‌ها **حافظه کمتری** نیاز دارد و فشرده‌سازی داده‌ها بهینه می‌شود.

مثال 💡

به مثال زیر توجه کنید:

📊 درخت هافمن برای مثال داده‌شده

کاراکتر	a	b	c	d	e	f
طول کد فیکس	000	001	010	011	100	101
فرکانس	51	20	2	3	9	15
طول کد متغیر	0	111	1100	1101	100	101

در این مثال، کاراکتر **"a"** پنجاه و یک بار در صد کاراکتر تکرار شده و بیشترین فرکانس را داراست، و کاراکتر **"c"** کمترین فرکانس تکرار را در بین کاراکترها دارد. همان‌طور که می‌بینیم، طول کد کاراکتر **"a"** که بیشترین تکرار را دارد برابر **1** و طول کد کاراکتر **"c"** که کمترین فرکانس را دارد برابر **4** می‌باشد.

حال با مقادیر فوق، مقدار حافظه را محاسبه می‌کنیم:

در **کد فیکس** به ازای هر کاراکتر ۳ بیت برای ذخیره‌سازی نیاز داریم، پس:

```
bits 300 = 3 * 100
```

می‌بینیم برای ذخیره‌ی یک متن صد کاراکتری به ۳۰۰ بیت نیاز داریم.

در الگوریتم **کد هافمن** برای هر کاراکتر به میزان **متغیر** بیت نیاز داریم، بنابراین داریم:

```
bits 203 = 3*15 + 3*9 + 4*3 + 4*2 + 3*20 + 1*51
```

می‌بینیم کد کردن همان متن صد کاراکتری با الگوریتم هافمن به جای کد طول ثابت به ۲۰۳ بیت نیاز دارد، پس استفاده از کد هافمن در این متن منجر به **کاهش ۳۲٪ فضای حافظه** می‌شود.

الگوریتم هافمن یک الگوریتم فشرده‌سازی **بدون از دست دادن اطلاعات** است؛ می‌بینیم در مثال فوق هیچ اطلاعاتی از دست نرفته و فقط از روش دیگری برای نمایش اطلاعات استفاده شده است.

سوالی که مطرح می‌شود این است که بیشینه طول کاراکتر کد شده در **کد طول ثابت** برابر ۳ است، اما این مقدار در الگوریتم هافمن برابر ۴ است! آیا نمی‌توان با ۳ بیت کد کنیم تا فضای کمتری اشغال شود؟

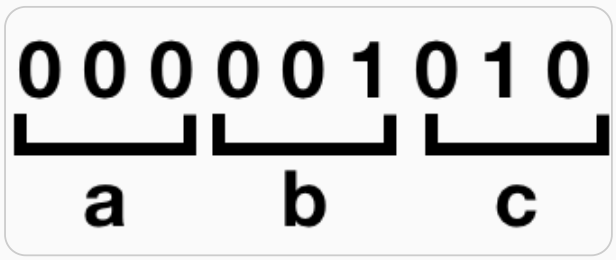
جواب **خیر** است؛ طول کدهای هر کاراکتر از **درخت پیشوندی** که در ادامه درباره‌ی آن صحبت می‌کنیم به دست می‌آید و این عمل برای حل مشکل دیکدینگ ضروری است.

دیکدینگ

در متن‌های کد شده، اطلاعات به صورت یک **توالی از بیت‌ها** دیده می‌شوند. برای مثال، رشته‌ی `"abc"` به صورت `00.001.010` کد می‌شود.

برای دیکد کردن، از ابتدا هر سه بیت را جدا می‌کنیم و با استفاده از **جدول کدگذاری**، هر کاراکتر را بازسازی می‌کنیم. به این ترتیب رشته‌ی اصلی به صورت کامل بازگردانده می‌شود.

📊 مثال دیکدینگ با درخت هافمن



اما این روش در الگوریتم **هافمن** به مشکل برمی‌خورد، زیرا کاراکترهای ما دارای **طول متغیر** هستند و روش جداسازی ثابت طول برای هافمن جواب نمی‌دهد. برای حل این مشکل، از **کد پیشوندی (Prefix Code)** استفاده می‌کنیم تا هر کاراکتر بتواند به درستی و بدون ابهام دیکد شود.

کد پیشوندی 🔑

در این روش، مقادیر هر کاراکتر **کد شده** به کاراکترهای دیگر بستگی دارد. به عبارت دیگر، هیچ کاراکتری نمی‌تواند پیشوند کد کاراکتر دیگری باشد. این ویژگی باعث می‌شود دیکد کردن رشته‌های باینری بدون ابهام و به‌صورت مرحله‌به‌مرحله امکان‌پذیر باشد.

مثال کد پیشوندی 💡

فرض کنید از `0` ، `1` ، `01` و `10` برای نمایش کاراکترهای **a**، **b**، **c** و **d** استفاده کنیم. (صفر پیشوند `01` و یک پیشوند `10` است) در این حالت، رشته‌ی `00101` می‌تواند به صورت‌های مختلفی دیکد شود: `aabc` ، `acab` ، `aadb` ، `acc` ، `aabab` و این موضوع موجب **ابهام در دیکدینگ** می‌شود.

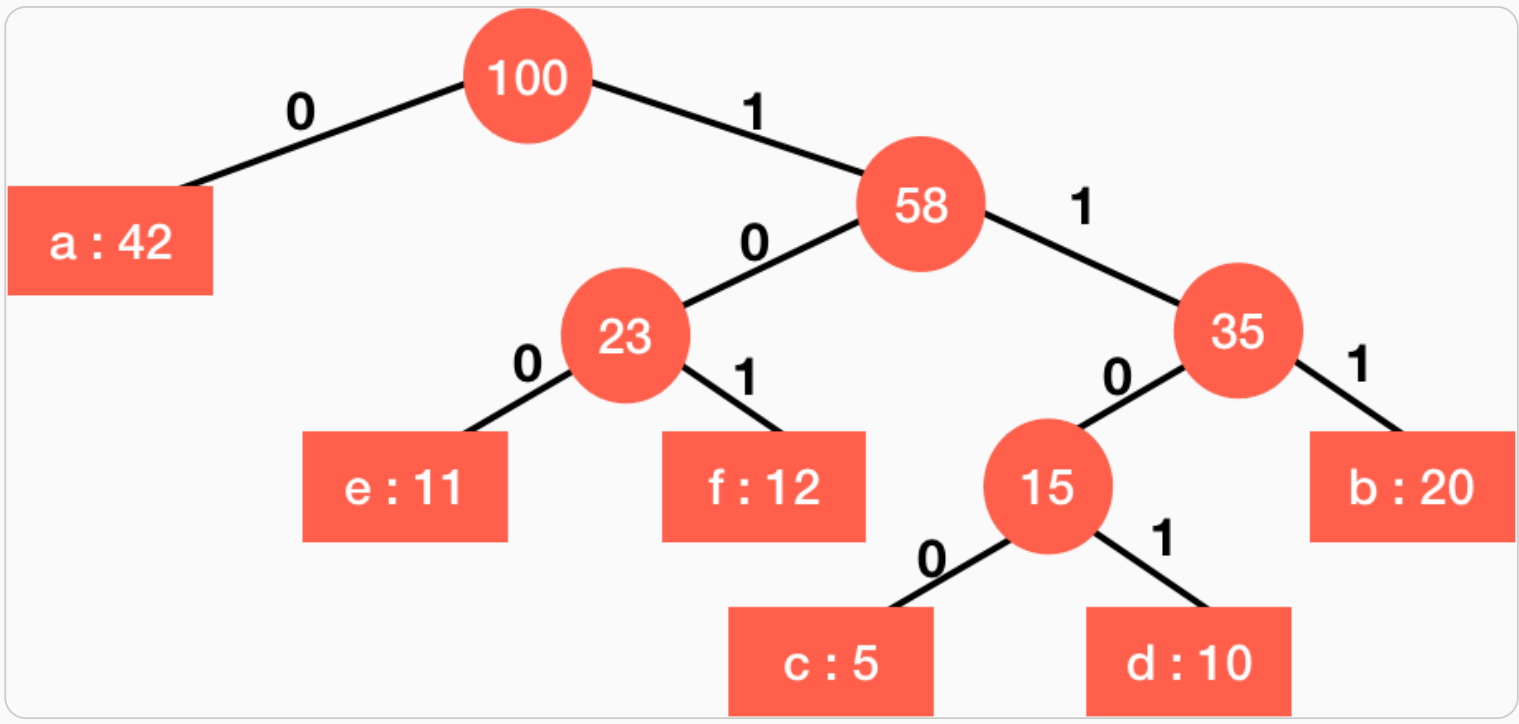
برای حل این مشکل از **کد پیشوندی** استفاده می‌کنیم. به طور مثال، اگر کاراکترهای **a**، **b** و **c** را به ترتیب با `0` ، `111` و `110` کد کنیم، می‌بینیم هیچ کدی پیشوند کد دیگر نیست. بنابراین در دیکد کردن، **هیچ ابهامی وجود ندارد**.

برای نمونه، رشته‌ی `0111110` به صورت یکتا به **abc** ترجمه می‌شود.

پیاده‌سازی کد هافمن 🛠️

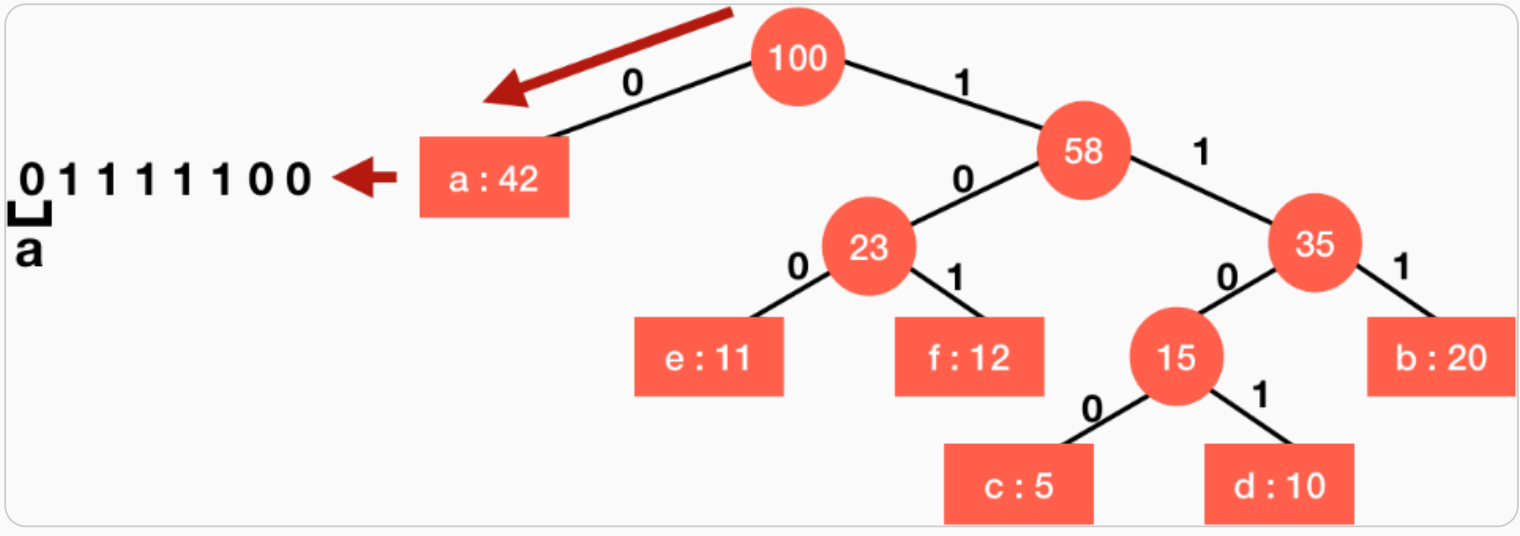
درخت زیر را در نظر بگیرید:

📊 درخت هافمن برای مثال



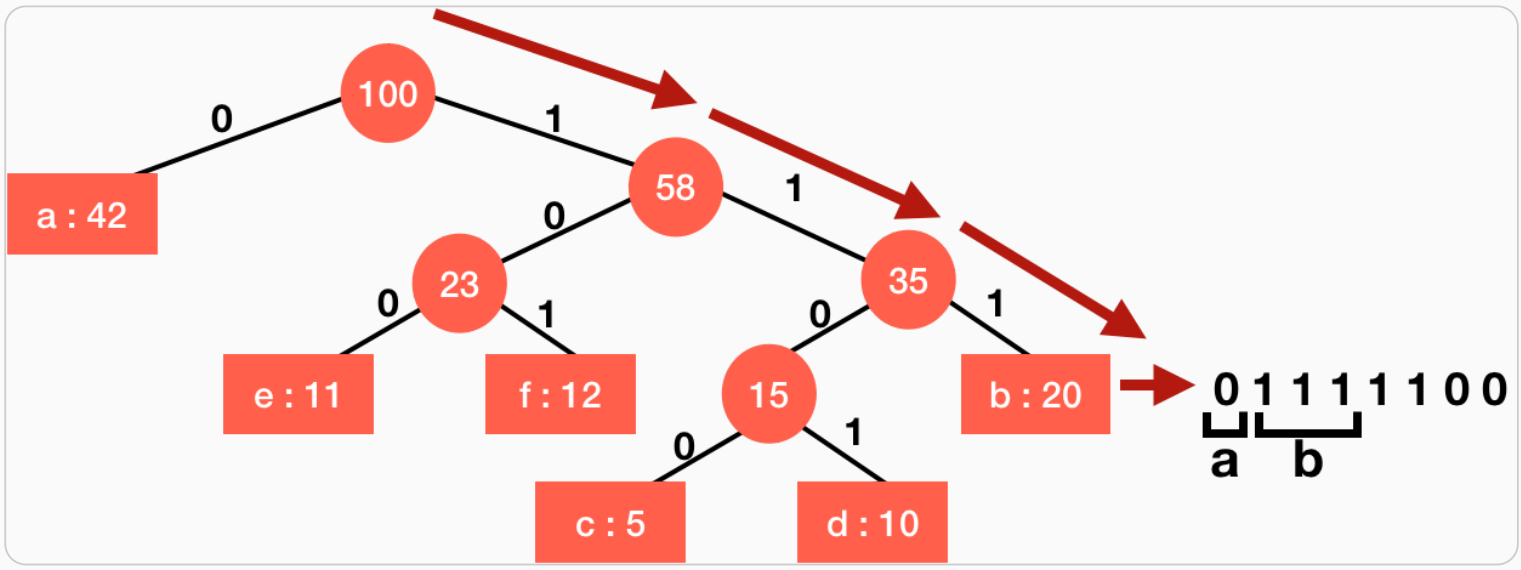
اولین برداشت از درخت این است که برگ‌های درخت، **کاراکترها و فرکانس‌های تکرار** هستند. با حرکت به سمت چپ هر نود، عدد **0** و با حرکت به سمت راست هر نود، عدد **1** تولید می‌شود. بنابراین، حرکت از سمت ریشه به هر برگ که یک کاراکتر است، **کد هافمن** آن کاراکتر را تولید می‌کند.

رشته‌ی `0111110` را در نظر بگیرید. برای دیکد کردن رشته، از ریشه شروع می‌کنیم:



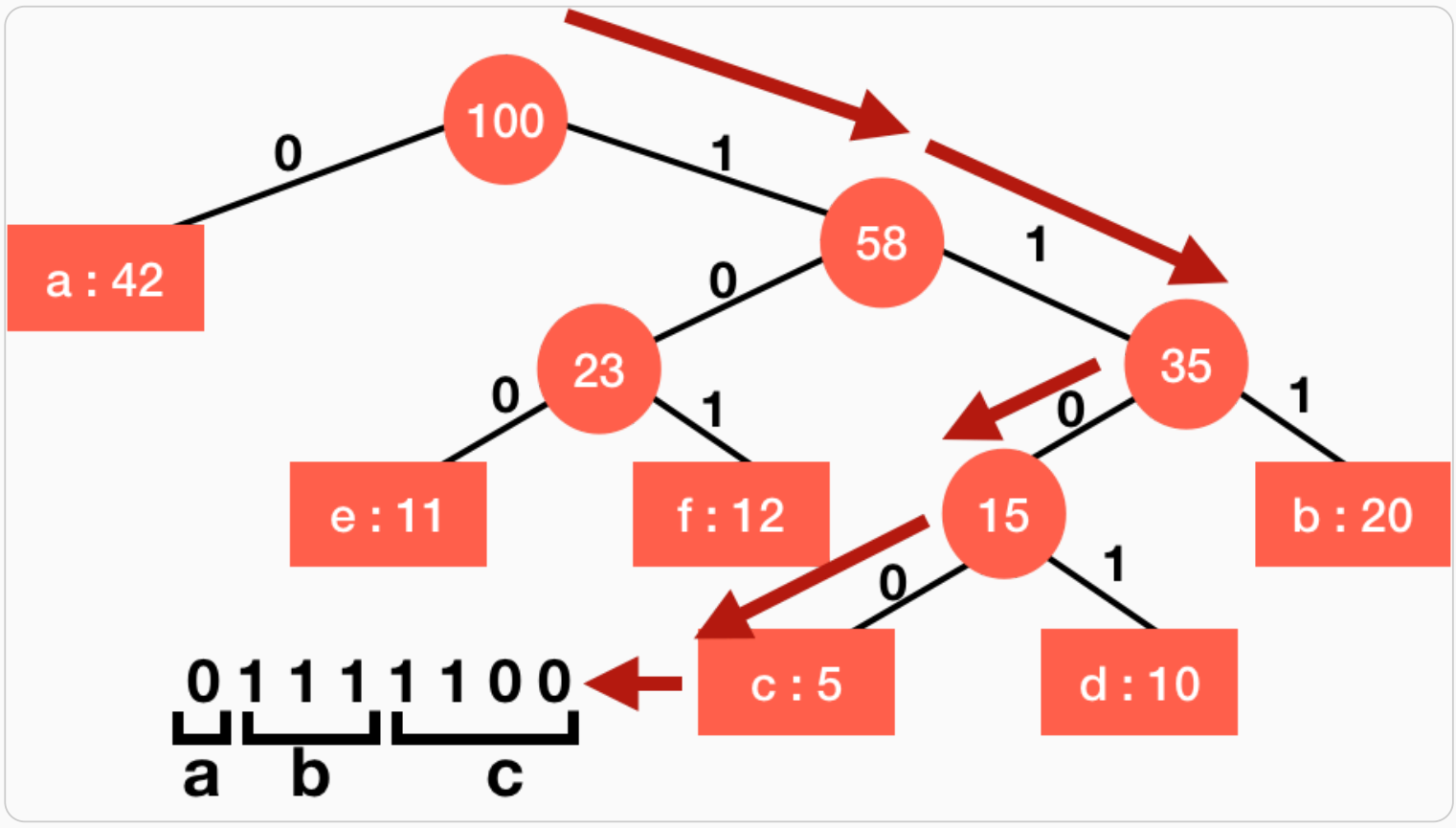
به برگی می‌رسیم که کاراکتر **a** در آن قرار دارد؛ بنابراین اولین کاراکتر رشته **a** است.

سپس دوباره از ریشه شروع می‌کنیم و ادامه‌ی رشته را دیکد می‌کنیم:



به همان صورت قبلی، کاراکتر دوم **b** است.

باز هم به ریشه می‌رویم و عملیات را روی ادامه‌ی رشته انجام می‌دهیم:



دیدیم رشته به این صورت دیکد می‌شود، اما برای ساخت **درخت پیشوندی** چه کاری باید انجام شود؟

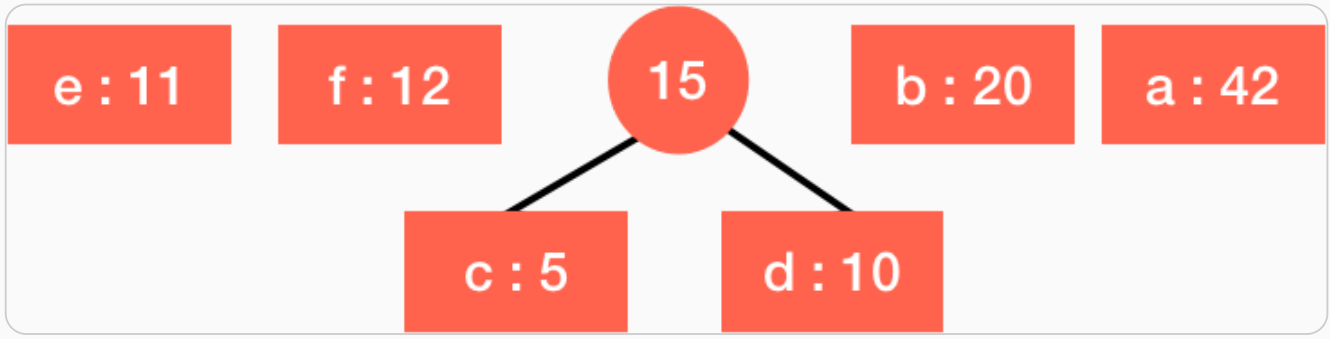
انکود با درخت 🌳

برای انکود، درخت به صورت زیر ساخته می‌شود:

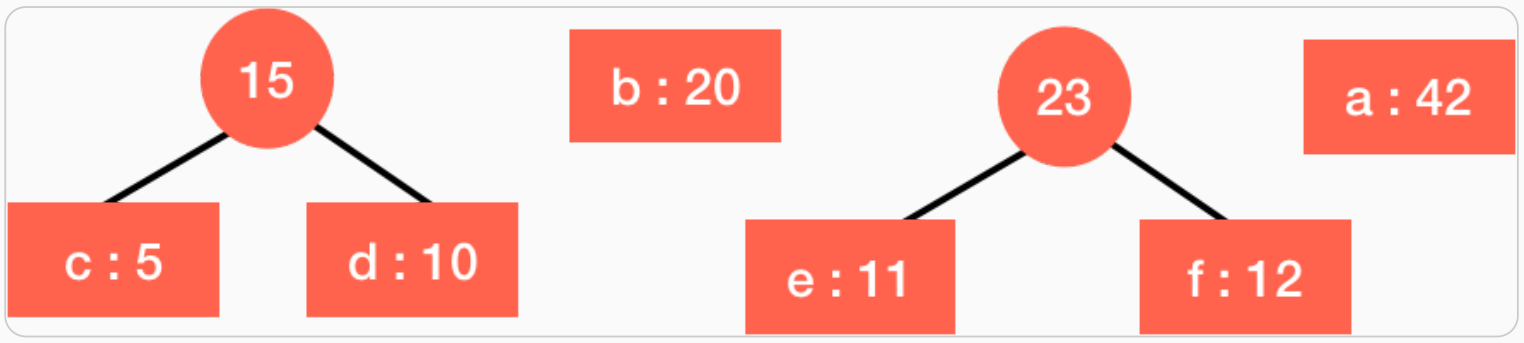
1. نودهایی شامل **کاراکتر و فرکانس** می‌سازیم و آن‌ها را به صورت صعودی مرتب می‌کنیم:

2. سپس به صورت **حریصانه** دو نود سمت چپ را با هم مرج می‌کنیم و در والد آن‌ها مجموع دو فرکانس برگ را می‌نویسیم و

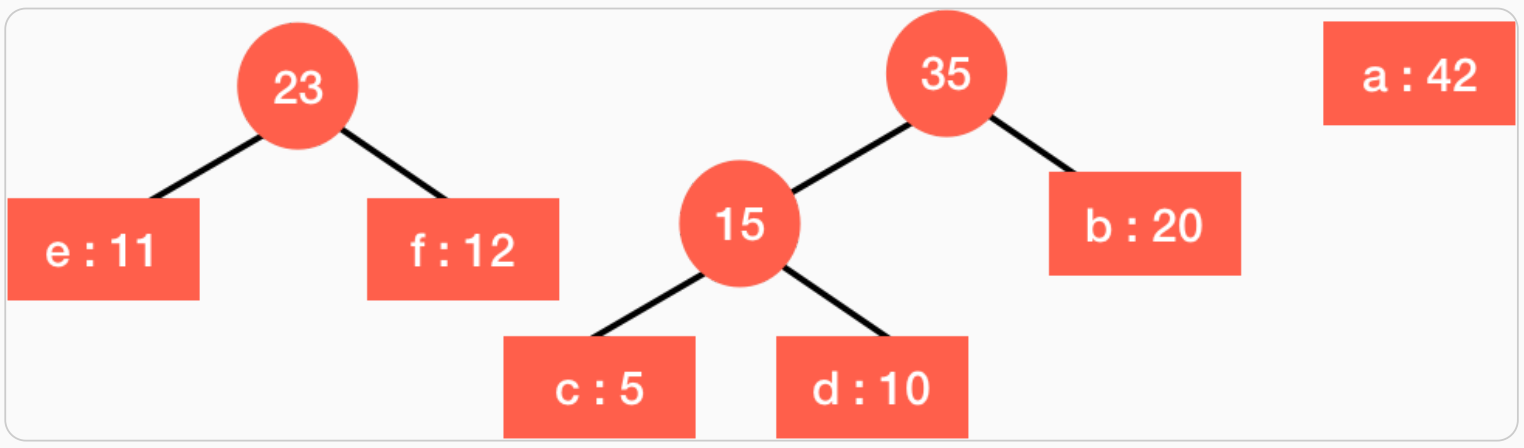
درخت حاصل را به مجموعه اول برمی‌گردانیم و دوباره مرتب می‌کنیم:

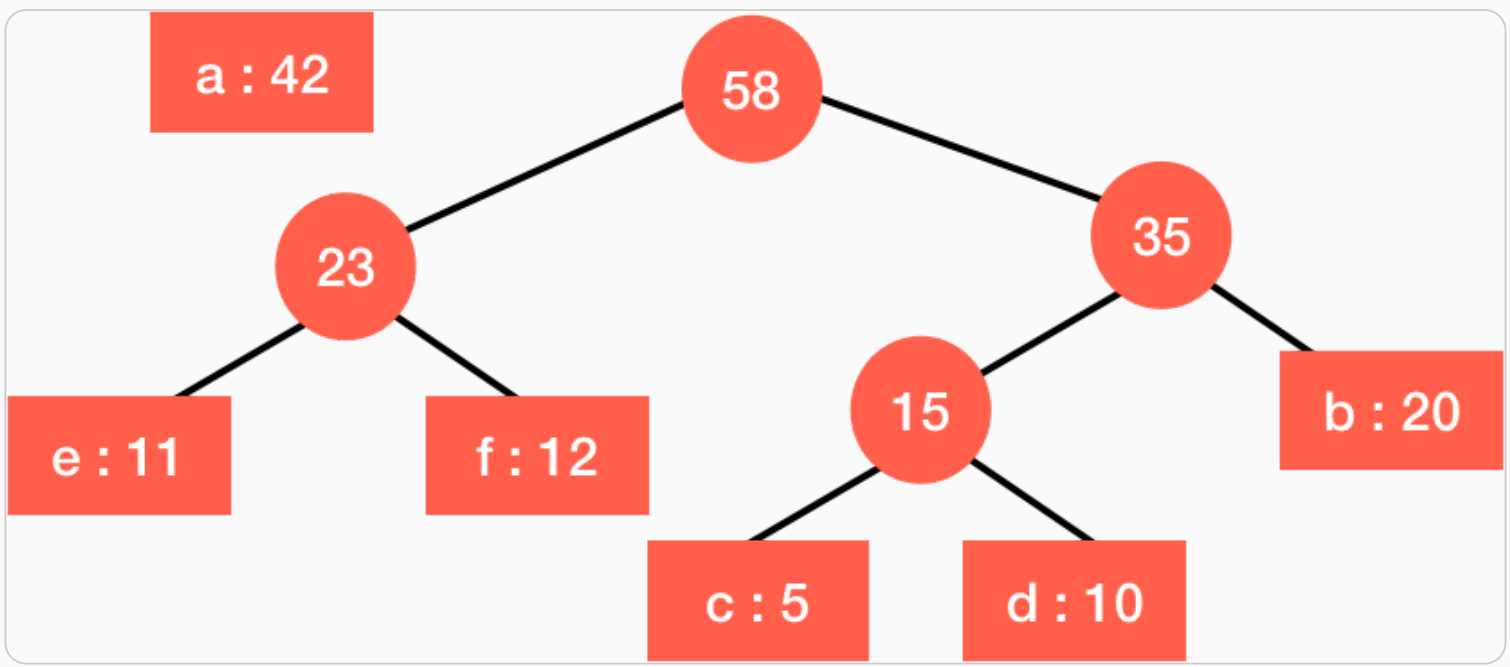


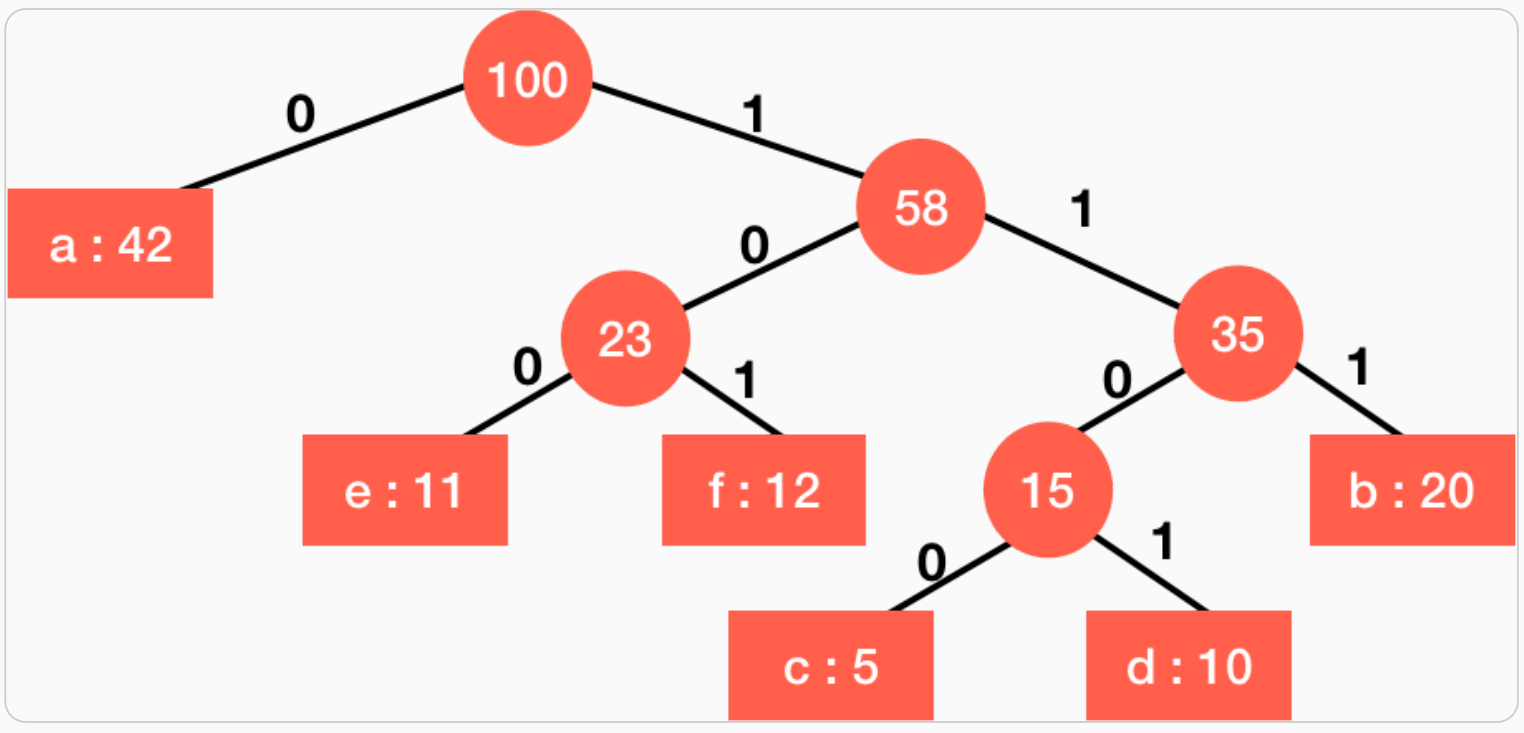
3. باز هم به صورت حریصانه دو نود سمت چپ را با هم مرج کرده و به مجموعه اول می‌بریم:



4. عملیات را ادامه می‌دهیم تا به یک درخت واحد برسیم:



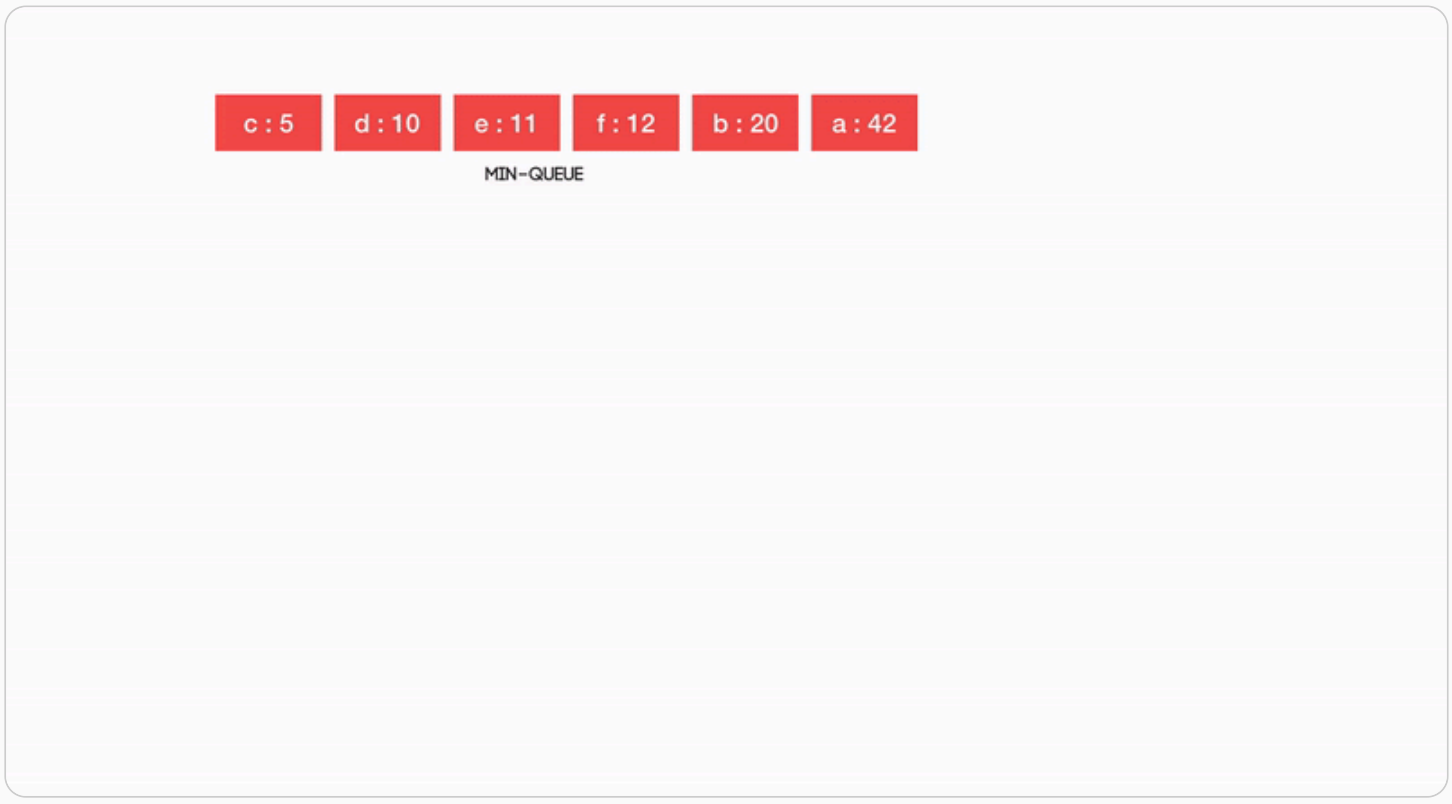




در نهایت به یک درخت **اپتیمال** می‌رسیم که عمق آن، حداکثر تعداد بیت هر کاراکتر است و تعداد بیت لازم برای کد کردن برابر است با:

$$Total\ bits = \sum d_i \times i.freq$$

مراحل ساخت را می‌توانید در **GIF** زیر مشاهده کنید:



کد ساخت درخت هافمن 💻

تعریف کلاس‌ها و توابع پایه برای درخت هافمن 1️⃣

In [1]:
```python
# کلاس برای گره‌های درخت هافمن
class MinHeapNode:
    def __init__(self, data, freq):
        self.data = data  # کاراکتر
        self.freq = freq  # فرکانس کاراکتر
        self.left = None  # اشاره‌گر به فرزند چپ
        self.right = None  # اشاره‌گر به فرزند راست

# کلاس برای هیپ حداقل
class MinHeap:
    def __init__(self, capacity):
        self.size = 0  # تعداد گره‌های فعلی در هیپ
        self.capacity = capacity  # ظرفیت هیپ
        self.array = []  # لیست برای ذخیره گره‌ها
```

توابع کمکی برای Min Heap 2️⃣

In [2]:
```python
# تابع برای ایجاد یک گره جدید
def new_node(data, freq):
    return MinHeapNode(data, freq)

# تابع برای ایجاد هیپ حداقل
def create_min_heap(capacity):
    return MinHeap(capacity)

# تابع برای تعویض دو گره
def swap_min_heap_node(a, b):
    a, b = b, a
```

```
    return a, b

# تابع برای تبدیل لیست به هیپ حداقل
def min_heapify(min_heap, idx):
    smallest = idx
    left = 2 * idx + 1
    right = 2 * idx + 2

    if left < min_heap.size and min_heap.array[left].freq < min_heap.array[smallest].freq:
        smallest = left

    if right < min_heap.size and min_heap.array[right].freq < min_heap.array[smallest].freq:
        smallest = right

    if smallest != idx:
        min_heap.array[idx], min_heap.array[smallest] = swap_min_heap_node(min_heap.array[idx], min_heap.array[smallest])
        min_heapify(min_heap, smallest)

# تابع برای بررسی اینکه آیا اندازه هیپ برابر ۱ است
def is_size_one(min_heap):
    return min_heap.size == 1

# تابع برای استخراج حداقل گره از هیپ
def extract_min(min_heap):
    temp = min_heap.array[0]
    min_heap.array[0] = min_heap.array[min_heap.size - 1]
    min_heap.size -= 1
    min_heapify(min_heap, 0)
    return temp

# تابع برای درج یک گره جدید در هیپ
def insert_min_heap(min_heap, min_heap_node):
    min_heap.size += 1
    i = min_heap.size - 1

    while i > 0 and min_heap_node.freq < min_heap.array[(i - 1) // 2].freq:
        min_heap.array[i] = min_heap.array[(i - 1) // 2]
        i = (i - 1) // 2

    min_heap.array[i] = min_heap_node

# تابع برای ساخت هیپ حداقل
def build_min_heap(min_heap):
    n = min_heap.size - 1
    for i in range((n - 1) // 2, -1, -1):
        min_heapify(min_heap, i)
```

3️⃣ توابع ساخت درخت هافمن

In [3]:
```
# تابع برای ساخت و پر کردن هیپ حداقل با داده‌ها
def create_and_build_min_heap(data, freq, size):
    min_heap = create_min_heap(size)
    for i in range(size):
        min_heap.array.append(new_node(data[i], freq[i]))
    min_heap.size = size
    build_min_heap(min_heap)
    return min_heap

# تابع برای ساخت درخت هافمن
def build_huffman_tree(data, freq, size):
    left = right = top = None
    min_heap = create_and_build_min_heap(data, freq, size)

    while not is_size_one(min_heap):
        left = extract_min(min_heap)
        right = extract_min(min_heap)

        top = new_node('$', left.freq + right.freq)
        top.left = left
        top.right = right

        insert_min_heap(min_heap, top)

    return extract_min(min_heap)
```

4️⃣ توابع چاپ کدهای هافمن

In [4]:
```
# تابع برای بررسی اینکه آیا گره برگ است
def is_leaf(root):
    return not root.left and not root.right

# تابع برای چاپ آرایه کدها
def print_arr(arr, n):
    for i in range(n):
        print(arr[i], end="")
    print()

# تابع برای چاپ کدهای هافمن
```

```python
def print_codes(root, arr, top):
    if root.left:
        arr[top] = 0
        print_codes(root.left, arr, top + 1)

    if root.right:
        arr[top] = 1
        print_codes(root.right, arr, top + 1)

    if is_leaf(root):
        print(f"{root.data}: ", end="")
        print_arr(arr, top)

# تابع اصلی برای ساخت درخت هافمن و چاپ کدها
def huffman_codes(data, freq, size):
    root = build_huffman_tree(data, freq, size)
    arr = [0] * 100
    print_codes(root, arr, 0)
```

اجرای مثال 👀

In [8]:
```python
# مثال: داده‌ها و فرکانس‌ها
arr = ['a', 'b', 'c', 'd', 'e', 'f']
freq = [42, 20, 5, 10, 11, 12]

# اجرای تابع هافمن
huffman_codes(arr, freq, len(arr))
```

```
a: 0
e: 100
f: 101
c: 1100
d: 1101
b: 111
```

اثبات بهینگی کد هافمن 📐

مسئله 📌

فرض کنید به شما حروف الفبای **A** و تابع فرکانس

$$f : A \longrightarrow (0, 1)$$

داده شده است به گونه‌ای که

$$\sum_{x} f(x) = 1$$

درخت باینری‌ای پیدا کنید (**T**) با تعداد |A| برگ که عبارت زیر را کمینه کند:

$$ABL(T) = \sum_{\text{leaves of } T} f(x) \cdot depth(x)$$

اگر چنین درختی پیدا شود، **اپتیمال** است.

لم اول 📏

فرض کنید درخت **T** با مقادیر **f** و **A** موجود است و **y** و **w** دو برگ آن هستند.

حال درخت **T'** را در نظر بگیرید که از روی درخت **T** ساخته شده و تنها تفاوت آن با **T** این است که جای دو برگ **y** و **w** با هم عوض شده‌اند. بنابراین:

$$ABL(T') - ABL(T) = f(y)\,depth(w,T) + f(w)\,depth(y,T) - f(w)\,depth(w,T) - f(y)\,depth(y,T)$$
$$= f(y)\,(depth(w,T) - depth(y,T)) + f(w)\,(depth(y,T) - depth(w,T))$$
$$= (f(y) - f(w))\,(depth(w,T) - depth(y,T))$$

لم دوم 📐

درخت **اپتیمالی** وجود دارد که در آن برگ‌ها با **کمترین فرکانس**، خویشاوند (Sibling) هستند.

اثبات 📝

فرض کنید درخت **T** یک درخت اپتیمال است و دو برگ **w** و **y** دو کاراکتر با کمترین فرکانس هستند. بنابراین بیشترین عمق را در درخت دارا هستند.

اگر دو برگ ما با هم خویشاوند باشند، کار ما به اتمام می‌رسد. در غیر این صورت، برای دو برگ با شرط **depth(w,T) ≥ depth(y,T)** حالات زیر داریم:

1. برگ **w** دارای خویشاوند **z** است. طبق فرض، درخت **T'** از روی درخت **T** ساخته شده و جای دو برگ **w** و **z** با هم عوض می‌شود. با استفاده از لم اول:

$$ABL(T') \leq ABL(T)$$

از آنجا که درخت **T** اپتیمال است، درختی با هزینه کمتر وجود ندارد و

$$ABL(T') = ABL(T)$$

بنابراین طبق شروط بالا، **T'** نیز اپتیمال است.

2. فرض کنید برگ **w** بدون خویشاوند است. در این حالت، درخت با حذف این برگ و متصل کردن به والد قبلی ساخته می‌شود. بنابراین درخت **T'** هزینه کمتری نسبت به **T** دارد که با شرط اولیه اپتیمال بودن درخت **T** تناقض دارد و امکان‌پذیر نیست.

3. فرض کنید نود **z** در عمق بیشتری از **w** قرار دارد و با تعویض این دو نود به شرط **f(w) < f(z)** می‌رسیم. با این شرط، هزینه درخت **T'** کمتر از درخت **T** می‌شود که با شرط اپتیمال بودن درخت **T** تضاد دارد و غیرممکن است.

با توجه به الگوریتم ساخت درخت هافمن، مشاهده می‌شود که این الگوریتم از تمامی شروط بالا پیروی می‌کند و طبق لم اول:

$$ABL(T) = ABL(T') + f(w) + f(y)$$

بنابراین درخت **T'** نیز اپتیمال است.

استفاده از کد هافمن در دنیای امروزی 🌐

کد هافمن پایه و اساس بسیاری از روش‌های فشرده‌سازی متن است. برای مثال، روش‌های **PKZIP (Winzip)**، **GZIP** و **BZIP2** برای فشرده‌سازی فایل‌ها در نرم‌افزار Winzip از کد هافمن استفاده می‌کنند. همچنین در فشرده‌سازی فرمت‌های **JPEG** و **PNG** نیز این الگوریتم کاربرد دارد.

مدل‌های فرانت‌اند، فایل‌های **MP3** و هزاران مورد دیگر، همگی برای فشرده‌سازی داده خود از این کد استفاده می‌کنند.

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل سوم: الگوریتم‌های برنامه‌نویسی پویا

بخش اول: اعداد فیبوناچی و خردکردن پول

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 📑

مقدمه 📘

برنامه‌نویسی پویا از روش‌های بسیار پرکاربرد در طراحی الگوریتم‌هاست. اولین بار در سال ۱۹۵۰ توسط ریچارد بلمن ابداع شد و در بسیاری از زمینه‌های علمی به خصوص ریاضیات و علوم کامپیوتر کاربرد دارد. ایده‌ی پشت این دسته از الگوریتم‌ها این است که یک مسئله‌ی سخت را به مسائل مشابه و کوچکتر که حل آن‌ها راحت‌تر است تقسیم کند و با حل آن‌ها مسئله‌ی اصلی را حل کند. در واقع شباهتی هم به استقرا دارد که شما برای اثبات یک حکم، ابتدا برای مثال‌های کوچک آن اثبات می‌کنید و سپس آن را تعمیم می‌دهید.

اعداد فیبوناچی 🔢

به عنوان اولین مثال فرض کنید می‌خواهیم برنامه‌ای داشته باشیم که با ورودی گرفتن **n**، در خروجی عدد فیبوناچی **n**ام را برگرداند. به یاد بیاورید که اعداد فیبوناچی به این شکل تعریف می‌شوند:

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

In [1]:
```python
# تابع بازگشتی ساده برای محاسبه اعداد فیبوناچی
def f_naive(n):
    if n < 2:  # اگر n برابر 0 یا 1 باشد، خود n را برگردان
        return n
    return f_naive(n - 1) + f_naive(n - 2)  # فراخوانی بازگشتی برای محاسبه فیبوناچی
```

```python
# مثال: دریافت ورودی و چاپ نتیجه
n = int(input("عدد n را وارد کنید: "))
print(f"عدد فیبوناچی {n}ام: {f_naive(n)}")
```

```
عدد فیبوناچی 10ام: 55
```

کد با تابع بازگشتی 💻

در بالا یک کد طبق همان تعریف اعداد فیبوناچی با استفاده از تابع بازگشتی پیاده‌سازی شده است. زمان اجرای این کد چقدر است؟ چگونه می‌توان آن را بهبود داد؟

کد بهینه‌تر ⚡

برای اینکه زمان اجرا بهتر شود، آرایه‌ای نگه می‌داریم که عنصر **i**ام آن همان **f_i** باشد. با یک حلقه می‌توان در **f_n** ،**O(n)** را محاسبه کرد. بعداً می‌بینید که می‌توان این مسئله را در زمان **O(log n)** هم حل کرد!

In [2]:
```python
# تابع بهینه برای محاسبه اعداد فیبوناچی با استفاده از برنامه‌نویسی پویا
def fibonacci_dp(n):
    # ایجاد یک لیست برای ذخیره مقادیر فیبوناچی
    f = [0] * (n + 1)
    f[0] = 0  # مقدار اولیه برای f(0)
    f[1] = 1  # مقدار اولیه برای f(1)

    # محاسبه مقادیر فیبوناچی از 2 تا n
    for i in range(2, n + 1):
        f[i] = f[i - 1] + f[i - 2]

    return f[n]

# مثال: دریافت ورودی و چاپ نتیجه
n = int(input("عدد n را وارد کنید: "))
print(f"عدد فیبوناچی {n}ام: {fibonacci_dp(n)}")
```

```
عدد فیبوناچی 10ام: 55
```

زمان‌بندی بازه‌ها ⏰

N بازه به صورت **(Start_i, End_i, Profit_i)** که i اعداد بین ۱ تا N هستند داده شده است، که به ترتیب نقاط شروع و پایان و سود حاصل از آن بازه را نشان می‌دهند. بیشترین مقدار سود را با انتخاب کردن تعدادی بازه که همپوشانی ندارند به‌دست آورید!

به طور مثال برای بازه‌های **I_1 = (1, 2, 50)** و **I_2 = (3, 5, 20)** و **I_3 = (6, 19, 100)** و **I_4 = (2, 100, 200)** بیشترین مقدار سود ممکن برابر ۲۵۰ است که با انتخاب بازه‌های ۱ و ۴ به‌دست می‌آید.

نکته: اکیداً توصیه می‌شود قبل از خواندن راه حل، دقایقی خودتان به دنبال راه حل این مساله بگردید.

نخستین ایده 💡

اولین راهی که برای حل این مسئله به ذهن می‌رسد، انتخاب بیشترین نسبت سود به طول بازه در بین بازه‌هایی است که می‌توان انتخاب کرد، اما با کمی تفکر متوجه می‌شویم که این روش گاهی جواب بهینه را نمی‌دهد. برای مثال، در مثال بالا طبق این روش ابتدا بازه‌ی **I_1** و سپس **I_3** انتخاب می‌شود که می‌دانیم جواب بهینه نیست!

راه حل اول 🛠️

ایده‌ی این راه‌حل مرتب کردن صعودی بازه‌ها بر اساس نقطه‌ی انتهایی آن‌ها و سپس حل سوال به صورت بازگشتی است. برای هر بازه دو حالت وجود دارد:

۱ - بازه‌ی مد نظر انتخاب شود و مساله را برای بقیه‌ی بازه‌های غیر مشترک با این بازه حل کنیم (چون بازه‌ها مرتب شده هستند می‌توان اولین بازه‌ی غیر مشترک را با باینری سرچ به‌دست آورد).
۲ - بازه‌ی مد نظر انتخاب نشود، که در این صورت مساله را با حذف این بازه برای بقیه‌ی بازه‌ها حل می‌کنیم.

نحوه‌ی پیاده‌سازی این الگوریتم را در زیر می‌بینید.

In [3]:
```python
class Job:
    def __init__(self, start, finish, profit):
        self.start = start
        self.finish = finish
        self.profit = profit

def binary_search(jobs, target_start, low, high):
    # جستجوی دودویی برای پیدا کردن آخرین شغل غیرمتداخل
    result = -1
    while low <= high:
        mid = (low + high) // 2
        if jobs[mid].finish <= target_start:
            result = mid
            low = mid + 1
        else:
            high = mid - 1
    return result

def find_last_non_conflicting_job(jobs, n):
    return binary_search(jobs, jobs[n].start, 0, n - 1)

def max_profit(jobs, n, memo):
    if n < 0:
        return 0
    if n == 0:
        return jobs[0].profit
    if memo[n] is not None:
        return memo[n]

    # پیدا کردن آخرین شغل غیرمتداخل با شغل فعلی
    index = find_last_non_conflicting_job(jobs, n)

    # محاسبه سود با شامل کردن شغل فعلی
    incl = jobs[n].profit
    if index != -1:
        incl += max_profit(jobs, index, memo)

    # محاسبه سود با خارج کردن شغل فعلی
    excl = max_profit(jobs, n - 1, memo)

    # ذخیره نتیجه در حافظه موقت
    memo[n] = max(incl, excl)
    return memo[n]

# مثال: لیستی از شغل‌ها
jobs = [
    Job(1, 2, 50),
    Job(3, 5, 20),
    Job(6, 19, 100),
    Job(2, 100, 200)
]

# مرتب‌سازی شغل‌ها بر اساس زمان پایان
jobs.sort(key=lambda x: x.finish)

# ایجاد لیست برای ذخیره نتایج موقت
memo = [None] * len(jobs)

# محاسبه بیشترین سود
result = max_profit(jobs, len(jobs) - 1, memo)
print(f"The maximum profit is {result}")
```

```
The maximum profit is 250
```

تحلیل زمانی این راه حل ⏱️

در این الگوریتم هر دو حالت گفته‌شده برای هر بازه چک می‌شود، بنابراین زمان اجرای این الگوریتم نمایی است و مقدار حافظه‌ی استفاده شده توسط آن برابر $\mathcal{O}(N)$ است.

راه حل دوم 💡

با استفاده از ایده‌ی برنامه‌نویسی پویا می‌توان مساله‌ی اصلی را با حل تعدادی مساله‌ی کوچکتر، حل کرد. با ایجاد یک جدول که مقادیر بیشترین سود را برای مساله‌های کوچکتر ذخیره می‌کند این مساله را حل می‌کنیم.

در این راه‌حل نیز با شروع از اولین بازه، استفاده کردن یا نکردن از این بازه را در نظر می‌گیریم، و مقدار به‌دست آمده را در $maxProfit[i]$ برای بازه‌ی i ام ذخیره می‌کنیم. هم‌چنین برای به‌دست آوردن بازه‌ی بعدی غیرمشترک از جست‌وجوی دودویی استفاده می‌کنیم.

نحوه‌ی پیاده‌سازی این الگوریتم را در زیر می‌بینید.

In [4]:
```python
# تعریف کلاس Job برای نگهداری اطلاعات هر شغل
class Job:
    def __init__(self, start, finish, profit):
        self.start = start
        self.finish = finish
        self.profit = profit

# تابع جستجوی دودویی برای یافتن آخرین شغل ناسازگار با شغل فعلی
def find_last_non_conflicting_job(jobs, i):
    low = 0
    high = i - 1
    result = -1
    while low <= high:
        mid = (low + high) // 2
        if jobs[mid].finish <= jobs[i].start:
            result = mid
            low = mid + 1
        else:
            high = mid - 1
    return result

# تابع اصلی برای محاسبه بیشترین سود با استفاده از برنامه‌نویسی پویا
def max_profit_dp(jobs):
    # مرتب‌سازی شغل‌ها بر اساس زمان پایان
    for i in range(len(jobs)):
        for j in range(i + 1, len(jobs)):
            if jobs[i].finish > jobs[j].finish:
                jobs[i], jobs[j] = jobs[j], jobs[i]

    n = len(jobs)
    # ایجاد جدول برای ذخیره بیشترین سود تا هر شغل
    max_profit = [0] * n
    max_profit[0] = jobs[0].profit  # مقدار اولیه برای اولین شغل

    # پر کردن جدول به صورت پایین به بالا
    for i in range(1, n):
        # محاسبه سود در صورت انتخاب شغل فعلی
        incl = jobs[i].profit
        index = find_last_non_conflicting_job(jobs, i)
        if index != -1:
            incl += max_profit[index]
        # انتخاب بیشترین سود بین انتخاب یا عدم انتخاب شغل فعلی
        max_profit[i] = incl if incl > max_profit[i - 1] else max_profit[i - 1]

    return max_profit[n - 1]

# تعریف لیست شغل‌ها
jobs = [
    Job(1, 2, 50),
    Job(3, 5, 20),
    Job(6, 19, 100),
    Job(2, 100, 200)
]

# محاسبه و چاپ بیشترین سود ممکن
result = max_profit_dp(jobs)
print("The maximum profit is", result)
```

```
The maximum profit is 250
```

تحلیل زمانی این راه حل ⏱️

در این الگوریتم زمان $\mathcal{O}(N \log N)$ برای مرتب سازی بازه‌ها بر اساس پایان هر بازه صرف می‌شود و زمان لازم برای به‌دست آوردن تمامی زیرمساله‌های موجود برابر $\mathcal{O}(N)$ است. بنابراین، زمان اجرای کل الگوریتم برابر جمع این دو، یعنی $\mathcal{O}(N \log N)$ می‌باشد.

خرد کردن پول 💰

یکی از مسئله‌های کلاسیک دیگر که در زندگی واقعی هم با آن مواجه بوده‌ایم مسئله خرد کردن پول است. فرض کنید بخواهیم مقدار مشخص c را با تعدادی دلخواه از سکه‌های a_1, a_2, \dots , a_n پرداخت کنیم.

اولین ایده 💡

ایده‌ی اول و طبیعی که به ذهن می‌رسد این است که حریصانه عمل کنیم. یعنی سعی کنیم تا جای ممکن سکه‌های با قیمت بالا را برداریم و از مقدار مورد نیاز بالاتر نرویم. برای اینکه حتما مبلغ قابل خرد کردن باشد، فرض ساده‌کننده $a_1 = 1$ را می‌گذاریم. این روش حتما بهترین جواب را پیدا نمی‌کند؛ برای مثال:

C = 10

coins = [1, 5, 6]

10 = 6 + 1 + 1 + 1 + 1 (Greedy = 5)

10 = 5 + 5 (Optimal = 2)

In [5]:
```python
# دریافت ورودی تعداد سکه‌ها
n = int(input())
a = [0] * 100

# دریافت مقدار هر سکه
for i in range(n):
    a[i] = int(input())

# مرتب‌سازی سکه‌ها به ترتیب صعودی
for i in range(n):
    for j in range(i + 1, n):
        if a[i] > a[j]:
            a[i], a[j] = a[j], a[i]

# دریافت مبلغ مورد نظر برای خرد کردن
C = int(input())
answer = 0

# پیمایش از سکه‌های با ارزش بالا به پایین
for i in range(n - 1, -1, -1):
    num = C // a[i]  # تعداد سکه‌هایی که می‌توانیم از نوع a[i] برداریم
    C -= a[i] * num  # کم کردن مقدار پرداخت‌شده از مبلغ باقی‌مانده
    answer += num    # اضافه کردن تعداد سکه‌ها به پاسخ نهایی

print("Greedy answer:", answer)
```

```
Greedy answer: 1
```

روش پیدا کردن جواب بهینه 🏆

قبل از خواندن این بخش سعی کنید ببینید چطور می‌توان روشی طراحی کرد که جواب بهینه را پیدا کند.

برای اینکار از برنامه‌نویسی پویا استفاده می‌کنیم. فرض کنیم **opt[i]** برابر کمترین تعداد سکه برای پرداخت مقدار i باشد.

به این ترتیب جواب مسئله می‌شود **opt[C]**.

In [9]:
```python
# دریافت تعداد انواع سکه
N = int(input())
w = [0] * 100

# دریافت مقدار هر سکه
for i in range(N):
    w[i] = int(input())

# دریافت مبلغ مورد نظر برای خرد کردن
c = int(input())

# ایجاد آرایه opt برای نگهداری کمترین تعداد سکه برای هر مقدار تا c
opt = [10**9] * 1001  # مقدار اولیه زیاد برای شبیه‌سازی بی‌نهایت
opt[0] = 0  # برای مبلغ صفر، نیاز به هیچ سکه‌ای نیست

# پر کردن آرایه opt با استفاده از برنامه‌نویسی پویا
for i in range(1, c + 1):
```

```python
    for j in range(N):
        if w[j] <= i:
            # بررسی اینکه آیا استفاده از سکه w[j] بهتر است یا نه
            candidate = opt[i - w[j]] + 1
            if candidate < opt[i]:
                opt[i] = candidate

# چاپ پاسخ نهایی با بررسی امکان‌پذیری
if opt[c] == 10**9:
    print("مقدار مورد نظر قابل پرداخت با سکه‌های داده‌شده نیست.")
else:
    print("Optimal answer:", opt[c])
```

```
Optimal answer: 2
```

تحلیل زمانی ⏱️

برای update کردن هر خانه آرایه **opt** هزینه $O(n)$ پرداخت می‌کنیم و در کل زمان اجرا می‌شود $O(NC)$. زمان اجرای الگوریتم حریصانه $O(n)$ است اما لزوماً جواب بهینه را پیدا نمی‌کند. البته که تا حدی منطقی است و ممکن است از آن استفاده شود.

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل سوم: برنامه‌نویسی پویا

بخش دوم: ضرب ماتریس‌ها، کوله‌پشتی، LIS، LCS

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 📑

مقدمه 📘

در این بخش، قصد داریم چند مثال دیگر از مسائلی که به کمک برنامه‌نویسی پویا (Dynamic Programming) حل می‌شوند را با هم بررسی کنیم.

مسئله‌ی اول: ضرب ماتریس‌ها 📘

سوال:

فرض کنید یک دنباله از ماتریس‌ها به صورت $\langle A_1, A_2, \ldots, A_n \rangle$ به ما داده شده است و می‌خواهیم مناسب‌ترین راه را برای ضرب این ماتریس‌ها پیدا کنیم.

در واقع مسئله ما این نیست که حاصل‌ضرب ماتریس‌ها را محاسبه کنیم بلکه این است که تصمیم بگیریم به چه ترتیبی آن‌ها را در یکدیگر ضرب کنیم.

کاری که قرار است ما برای حل این مسئله انجام دهیم در واقع یک نحوه پرانتزگذاری می‌باشد یعنی اهمیتی ندارد که کدام زوج ماتریس کنار هم اول در هم ضرب شوند و کدام یک آخر؛ در نهایت جواب یکسان خواهد بود.

$$\ldots = ABCD = A(BC)D = ((AB)C)D = A(B(CD))$$

نکته‌ای که باید بدان توجه شود این است که برابر بودن در جواب نهایی به معنای برابر بودن در تعداد عمل‌ها نیست. در واقع نحوه پرانتزگذاری ما روی تعداد عمل‌های ریاضی ساده‌ای که برای حاصل‌ضرب دو ماتریس انجام می‌شود تاثیر می‌گذارد.

می‌دانیم دو ماتریس A و B در هم قابل‌ضرب هستند اگر و تنها اگر تعداد ستون‌های ماتریس A با تعداد سطرهای ماتریس B برابر

باشد. اگر بخواهیم ماتریس $A_{p\times q}$ را در ماتریس $B_{q\times r}$ ضرب کنیم، تعداد عملیات‌های ضرب اسکالری که باید انجام دهیم، برابر $p\cdot q\cdot r$ است. این مقدار را هزینه‌ی ضرب ماتریس A در B تعریف می‌کنیم.

به طور خلاصه، می‌خواهیم یک پرانتزگذاری برای انجام عملیات ضرب انتخاب کنیم، به طوری که تعداد کل عمل‌های ضرب‌مان کمینه شود.

به عنوان مثال فرض کنید که ماتریس A یک ماتریس ۳۰×۱۰، ماتریس B یک ماتریس ۵×۳۰ و ماتریس C یک ماتریس ۶۰×۵ باشد.

محاسبه هزینه ضرب ماتریس‌ها با آرایه ابعاد 📘

$$(AB)C = (10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500\ operations$$

$$A(BC) = (30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000\ operations$$

مشخص است که پرانتزگذاری اول، هزینه‌ی کمتری دارد.

فرض کنید یک آرایه **P** به ما داده شده است که حاوی تعدادی عدد است و هر دو عدد متوالی بیانگر ابعاد یک ماتریس است.

به عنوان مثال **P = {10, 20, 30, 40, 30}**، تعریف کننده ۴ ماتریس به ابعاد ۱۰×۲۰، ۲۰×۳۰، ۳۰×۴۰ و ۴۰×۳۰ می‌باشد. حال می‌خواهیم برای ضرب این دنباله ماتریس‌ها، کمترین هزینه کل را محاسبه کنیم.

راه‌حل اول 📚

به عنوان یک روش ساده برای حل این مسئله، می‌توانیم تمام حالات مختلف پرانتزگذاری را اعمال کنیم؛ تعداد عمل‌ها را محاسبه کنیم و در نهایت کمترین مقدار را برگردانیم. برای این کار، از یک تابع بازگشتی استفاده می‌کنیم. این تابع، روی آخرین ضربِ انجام شده در این زنجیره، حالت‌بندی می‌کند. در یک زنجیره ماتریسی به سایز n، ما می‌توانیم اولین مجموعه پرانتز (که نشان‌دهنده آخرین ضربی است که انجام می‌شود) را در $n-1$ حالت مختلف قرار دهیم.

بعد از اینکه ما این‌کار را انجام دادیم، در واقع مساله را به دو زیرمسئله کوچکتر تقسیم کرده‌ایم. بنابراین می‌توان مسئله را به سادگی به روش بازگشتی حل کرد.

قطعه کد زیر پیاده‌سازی روش بالا را نشان می‌دهد.

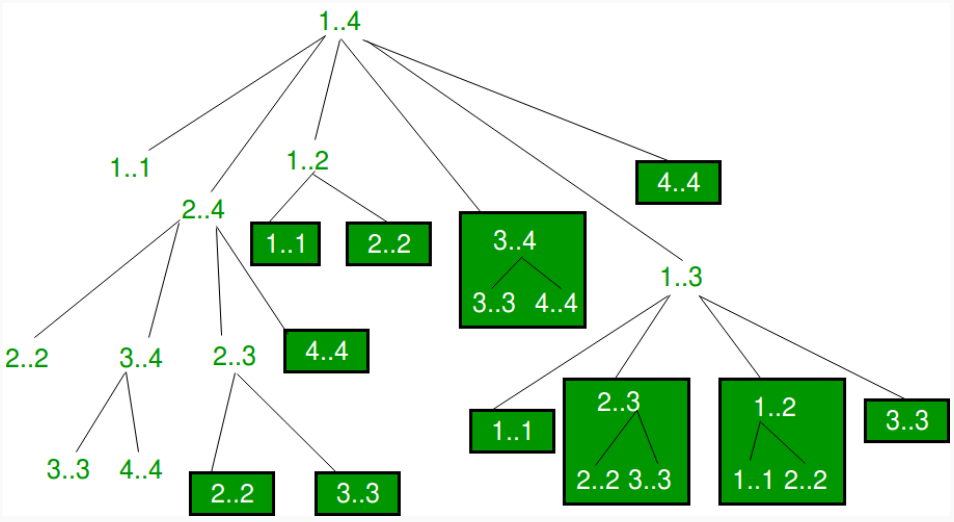
In [1]:
```python
# تابع بازگشتی برای محاسبه کمترین هزینه ضرب ماتریس‌ها
def recursive_matrix_chain(p, i, j):
    if i == j:
        return 0  # فقط یک ماتریس داریم، نیازی به ضرب نیست

    ans = float('inf')  # مقدار اولیه برای کمینه‌سازی
    for k in range(i, j):
        # تقسیم زنجیره به دو بخش و محاسبه هزینه ضرب آن‌ها
        operations = recursive_matrix_chain(p, i, k) + recursive_matrix_chain(p, k + 1, j) + p[i - 1] * p[k] * p[j]
        if operations < ans:
            ans = operations  # به‌روزرسانی کمترین هزینه

    return ans
```

In [2]:
```python
# تعریف آرایه ابعاد ماتریس‌ها
arr = [30, 35, 15, 5, 10, 20, 25]
n = 7

# محاسبه و چاپ کمترین هزینه ضرب ماتریس‌ها
print(recursive_matrix_chain(arr, 1, n - 1))
```

```
15125
```

تحلیل زمانی ⏱️

پیچیدگی زمانی روش بازگشتی بالا نمایی خواهد بود. باید توجه داشته باشیم که تعداد زیادی از زیرمسئله‌ها بارها و بارها در حال تکرار شدن هستند. به درخت زیر توجه کنید. این درخت برای یک زنجیره ماتریس با سایز ۴ است. در این مثال تابع `recursive_matrix_chain(p, 3, 4)` دوبار صدا زده شده‌است.



راه حل دوم 🛠️

از آنجایی که زیر مساله های مشابه بارها و بارها تکرار می‌شوند و همپوشانی دارند پس شرایط استفاده از برنامه‌نویسی پویا فراهم است.

می‌توانیم خیلی ساده تابع بازگشتی بالا را با استفاده از تکنیک memoization، به dp تبدیل کنیم. اما در این‌جا، قصد داریم با استفاده از روش جدولی bottom up، این سوال را حل کنیم.

در این سوال، `dp` را به این صورت تعریف می‌کنیم:

`dp[i][j]` : کمترین تعداد ضرب‌های مورد نیاز برای محاسبه ماتریس `A_{i..j}=A_iA_{i+1}...A_j`

حالت‌های پایه، حالت‌هایی هستند که `i=j` باشد. در این حالت، هزینه مورد نیاز برابر با صفر است. (لازم نیست ضربی انجام دهیم.)

حال فرض کنید پرانتزگذاری بهینه برای محاسبه `A_{i..j}`، دنباله‌ی `A_iA_{i+1}...A_j` را بین `A_k` و `A_{k+1}` قسمت می‌کند، که در آن `i ≤ k < j`.

در این صورت `dp[i][j]` برابر خواهد بود با کمترین هزینه محاسبه `A_{i..k}` و `A_{k+1..j}`، به علاوه هزینه ضرب این دو ماتریس در یک‌دیگر. از آن‌جایی که ابعاد ماتریس `A_i` برابر `p[i-1]×p[i]` است، می‌توان گفت که هزینه‌ی ضرب دو ماتریس `A_{i..k}` و `A_{k+1..j}` برابر است با `p[i-1].p[k].p[j]`. بنابراین می‌توان گفت:

$$dp[i][j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} \{dp[i][k] + dp[k+1][j] + p[i-1].p[k].p[j]\} & i < j \end{cases}$$

پاسخ مسئله‌ی اصلی در `dp[1][n]` قرار دارد. حال برای آپدیت کردن خانه‌های `dp`، با کمی دقت، متوجه میشویم که محاسبه `dp[i][j]` تنها به خانه‌هایی از `dp` مانند `dp[i'][j']` نیازمند است که `j'-i' < j-i`. پس اگر این آرایه را به گونه‌ای پر کنیم که `dp` بر حسب طول بازه‌ها به طور صعودی پر شود (یعنی ابتدا `dp` بازه‌هایی به طول ۱، سپس `dp` بازه‌هایی به طول ۲، و الی آخر) پر شوند، می‌توان مطمئن شد که خانه‌های `dp` به درستی محاسبه می‌شوند.

In [3]:
```python
# تابعی برای محاسبه کمترین هزینه ضرب ماتریس‌ها با روش bottom-up
def matrix_chain_order(p, n):
    MATRIX_COUNT = 201
    dp_matrix = [[0] * MATRIX_COUNT for _ in range(MATRIX_COUNT)]  # جدول هزینه‌ها

    # مقداردهی اولیه: هزینه ضرب یک ماتریس با خودش صفر است
    for i in range(1, n + 1):
        dp_matrix[i][i] = 0

    # طول زنجیره ماتریسی ا
    for l in range(2, n + 1):
        for i in range(1, n - l + 2):
            j = i + l - 1
            dp_matrix[i][j] = float('inf')  # مقدار اولیه برای کمینه‌سازی
            for k in range(i, j):
                # محاسبه هزینه ضرب دو بخش و جمع با هزینه ضرب نهایی
                cost = dp_matrix[i][k] + dp_matrix[k + 1][j] + p[i - 1] * p[k] * p[j]
                if cost < dp_matrix[i][j]:
                    dp_matrix[i][j] = cost

    return dp_matrix[1][n]
```

In [4]:
```python
# تعریف آرایه ابعاد ماتریس‌ها
p = [30, 35, 15, 5, 10, 20, 25]

# محاسبه و چاپ کمترین هزینه ضرب ماتریس‌ها
print(matrix_chain_order(p, 6))
```

```
15125
```

تحلیل زمانی این راه حل ⏱️

این الگوریتم، تمام خانه‌های `dp[i][j]` که `i≤j` باشد را محاسبه می‌کند (یعنی `O(n^2)` خانه). و محاسبه‌ی مقدار هر خانه، عملیاتی از `O(n)` است (یک حلقه که در آن مینیمم گرفته می‌شود)؛ پس در کل، مرتبه زمانی الگوریتم از `O(n^3)` است.

تمرین: آیا می‌توانید برنامهی بالا را به گونه‌ای تغییر دهید که روشِ ضرب کردن ماتریس‌ها (نحوه‌ی پرانتزگذاری) را نیز خروجی دهد؟ 🤔

مساله‌ی دوم: کوله‌پشتی 👜

یک کوله پشتی با گنجایش `V` واحد حجم داریم. همچنین تعدادی جنس داریم که هر کدام ارزش (value) و حجم (volume) مشخصی دارند. می‌خواهیم تعدادی از این اجناس را داخل کوله‌پشتی قرار دهیم، به طوری که مجموع ارزش اجناس داخل کوله‌پشتی، بیشینه شود. واضح است که مجموع حجم اجناس داخل کوله‌پشتی، نباید از گنجایش آن (`V`) بیشتر باشد.
نکته: اکیدا توصیه می‌شود قبل از خواندن راه حل دقایقی خودتان به دنبال راه حل این مساله بگردید.

این مسئله، نسخه‌های مختلفی دارد. برای مثال، در یک نسخه از این مسئله، می‌توان قسمتی از هر جنس را برداشت و داخل کوله‌پشتی گذاشت. که این نسخه به کمک برنامه‌نویسی حریصانه حل می‌شود. اما قصد داریم نسخه‌ی مطرح شده در بالا را به کمک برنامه‌نویسی پویا حل کنیم.

راه حل اول 🛠️

اولین راه‌حلی که به ذهن می‌رسد، این است که تمام حالت‌هایی که می‌توان اجناس را انتخاب کرد، به طوری که مجموع حجم آن‌ها از گنجایش کوله‌پشتی بیشتر نشود را پیدا کنیم، و میان آن‌ها، حالتی که مجموع ارزش‌شان بیشینه است را چاپ کنیم.

In [5]:
```python
# تابع بازگشتی برای حل مسئله کوله‌پشتی به روش ساده و نمایی
def knapsack_naive(n, V, volume, value, i=0, occupied=0, value_sum=0):
    if i == n:
        return value_sum  # رسیدن به انتهای لیست اجناس

    ans = -10**9  # مقدار اولیه برای بیشینه‌سازی

    # بررسی امکان انتخاب جنس فعلی
    if occupied + volume[i] <= V:
        # انتخاب جنس فعلی و ادامه حل برای بقیه اجناس
        taken = knapsack_naive(n, V, volume, value, i + 1, occupied + volume[i], value_sum + value[i])
        if taken > ans:
            ans = taken

    # عدم انتخاب جنس فعلی و ادامه حل برای بقیه اجناس
    not_taken = knapsack_naive(n, V, volume, value, i + 1, occupied, value_sum)
    if not_taken > ans:
        ans = not_taken

    return ans
```

In [6]:
```python
# تعریف داده‌های مسئله
n = 4
V = 13
volume = [5, 6, 7, 2]
value = [10, 20, 30, 40]

# محاسبه و چاپ بیشترین ارزش ممکن
print(knapsack_naive(n, V, volume, value))
```

```
70
```

تحلیل زمانی این راه حل ⏱️

واضح است که در این راه حل، در بدترین حالت، مجبور می‌شویم تمام زیرمجموعه‌های اجناس را پیدا کنیم (که تعداد آن‌ها از $O(2^n)$ و نمایی است). حال سعی داریم الگوریتمی با زمان اجرایی بهتر از نمایی برای این مسئله طراحی کنیم.

راه حل دوم 🛠️

شاید به ذهن‌تان برسد که الگوریتمی حریصانه برای این مسئله ارائه دهید. اما به احتمال خوبی، راه‌حل حریصانه‌ی شما مثال نقض دارد!

اگر بیشتر دقت کنیم، میتوانیم تابع بازگشتی‌مان را کمی بهینه‌تر کنیم. مثلاً جنس `n` ام را در نظر بگیرید؛ این جنس، یا در جواب بهینه، در کوله‌پشتی ظاهر می‌شود یا خیر.

اگر جنس `n` ام در جواب بهینه ظاهر شود، یعنی انگار باید بین `n-1` جنس دیگر، تعدادی جنس با بیشترین مجموع ارزش را در یک کوله پشتی با حجم خالی `V - volume[n]` قرار دهیم. و جواب محاسبه شده از این زیرمسئله را با ارزش جنس `n` ام (که آن را در کوله‌پشتی قرار دادیم) جمع کنیم.

و اگر هم جنس `n` ام در جواب بهینه نباشد، مانند این است که از بین `n-1` جنس دیگر، باید تعدادی را در کوله‌پشتی با حجم خالیِ `V` قرار دهیم.

بیایید `dp` مان را این‌گونه تعریف کنیم:

`dp[i][w]` : بیشترین ارزشی که می‌توان با استفاده از `i` جنس اول در کوله‌ای با حجم خالیِ `w` قرار داد.

همان‌طور که در بالا گفتیم، می‌توانیم این `dp` را به شکل زیر محاسبه کنیم:

$$dp[i][w] = \max(dp[i-1][w], dp[i-1][w - volume[i]] + value[i])$$

حالت پایه‌مان به شکل زیر است:

$$\forall w : dp[0][w] = 0$$

و پاسخ مسئله در `dp[n][V]` است.

در زیر، کدِ این برنامه با استفاده از تکنیک memoization آمده است. می‌توانید به عنوان تمرین، آرایه `dp` را به صورت bottom-up نیز محاسبه کنید.

In [7]:
```python
# تعریف ثابت‌ها برای حداکثر تعداد اجناس و حجم کوله‌پشتی
MAX_N = 1000
MAX_V = 2000

# ایجاد جدول dp برای ذخیره نتایج محاسبه‌شده
dp_knapsack = [[-1] * MAX_V for _ in range(MAX_N)]

# تابع بازگشتی با حافظه‌گذاری برای حل مسئله کوله‌پشتی
def knapsack_memoize(n, w, volume, value):
    if n < 0:
        return 0  # هیچ جنسی باقی نمانده، ارزش صفر است

    if dp_knapsack[n][w] != -1:
        return dp_knapsack[n][w]  # استفاده از مقدار محاسبه‌شده قبلی

    # حالت اول: جنس فعلی را انتخاب نمی‌کنیم
    ans = knapsack_memoize(n - 1, w, volume, value)

    # حالت دوم: اگر جا داریم، جنس فعلی را انتخاب می‌کنیم
    if w - volume[n] >= 0:
        candidate = value[n] + knapsack_memoize(n - 1, w - volume[n], volume, value)
        if candidate > ans:
            ans = candidate

    dp_knapsack[n][w] = ans  # ذخیره نتیجه برای استفاده‌های بعدی
    return ans
```

In [8]:
```python
# تعریف داده‌های مسئله
n = 4
V = 13
volume = [5, 6, 7, 2]
value = [10, 20, 30, 40]

# محاسبه و چاپ بیشترین ارزش ممکن
print(knapsack_memoize(n - 1, V, volume, value))
```

```
70
```

تحلیل زمانی این راه حل ⏱️

برای محاسبه پاسخ مسئله، در بدترین حالت باید تمام خانه‌های آرایه `dp` را پر کنیم. و هر خانه از این آرایه، حداکثر یک بار محاسبه می‌شود (و در دفعات بعد، از مقداری که قبلاً محاسبه شده استفاده می‌شود). پس می‌توان گفت مرتبه زمانی این الگوریتم از `O(nV)` است.

از آن‌جایی که مرتبه زمانی الگوریتم، تنها به تعداد اعداد ورودی بستگی ندارد (و به مقادیر آن‌ها یعنی `V`) بستگی دارد، می‌گوییم این الگوریتم در اصل **شبه چندجمله‌ای** است.

مساله‌ی سوم: بلندترین زیردنباله‌ی مشترک (Longest Common Subsequence یا LCS) 🧩

سوال: دو رشته $x=\langle x_1x_2...x_n\rangle$ و $y=\langle y_1y_2...y_m\rangle$ داده شده است.
می‌خواهیم بلندترین زیردنباله (غیرمتوالی) مشترک این دو رشته را پیدا کنیم.
به عنوان مثال، LCS دو رشته‌ی `ABCDGH` و `AEDFHR` برابر با `ADH` است.

این مسئله، در آنالیز رشته‌های DNA کاربرد زیادی دارد. به طور مثال، فرض کنید دو رشته‌ی DNA، متشکل از کاراکترهای A، C، G و T داشته باشیم. در بسیاری از مواقع، می‌خواهیم مشخص کنیم که دو رشته‌ی DNA، چقدر شبیه به هم هستند. می‌توان معیارهای مختلفی برای این میزان «شباهت» تعریف کرد. یکی از این معیارها، می‌تواند این باشد که رشته‌ای نسبتاً بلند مانند z یافت شود که زیردنباله‌ی هر دو رشته‌ی مورد آزمایش باشد. برای پیدا کردن این رشته‌ی z و یافتن شباهت میان دو رشته، نیاز داریم تا مسئله‌ی LCS را حل کنیم.

نکته: اکیدا توصیه می‌شود قبل از خواندن راه حل دقایقی خودتان به دنبال راه حل این مساله بگردید.

راه حل اول (غیرهوشمندانه) 🛠️

اولین راه حلی که به ذهن می‌رسد، این است که تمام زیردنباله‌های رشته‌ی اول را پیدا کنیم، و به ازای هر کدام از آن‌ها، چک کنیم که زیردنباله‌ی رشته‌ی دوم نیز باشد، و بین تمام زیردنباله‌های مشترک، زیردنباله با بیشترین طول را پیدا کنیم.
در این راه‌حل، باید تمام زیردنباله‌های رشته اول را پیدا کنیم، که تعداد آن‌ها از $O(2^n)$ است. و سپس به ازای هر زیردنباله، مشخص کنیم که زیردنباله‌ی رشته‌ی دوم هست یا نه. این عملیات نیز مرتبه زمانی از $O(m)$ دارد. پس مرتبه زمانی کلّی این الگوریتمِ غیرهوشمندانه، از $O(2^n\cdot m)$ و نمایی است. حال سعی می‌کنیم این زمان اجرای این الگوریتم را به چندجمله‌ای کاهش دهیم.

راه حل دوم 🛠️

هم‌چنان از تفکر بازگشتی بهره می‌گیریم! 😄

در این مسئله، می‌خواهیم طول LCS دو رشته‌ی $x=\langle x_1x_2...x_n\rangle$ و $y=\langle y_1y_2...y_m\rangle$ را پیدا کنیم. دو کاراکتر x_n و y_m را در نظر بگیرید. اگر $x_n = y_m$ باشد، می‌توان این کاراکتر را به عنوان آخرین عنصرِ LCS، در انتهای بزرگترین زیردنباله مشترک رشته‌های $x=\langle x_1x_2...x_{n-1}\rangle$ و $y=\langle y_1y_2...y_{m-1}\rangle$ قرار داد. $(*)$
پس $LCS(x, y)$ برابر است با:

$$LCS(\langle x_1x_2...x_{n-1}\rangle, \langle y_1y_2...y_{m-1}\rangle) + 1$$

اما اگر $x_n \neq y_m$ ، مشخص است که نمی‌توانیم عضوی را در LCS به نمایندگی از هر دویِ این کاراکترها قرار دهیم! پس باید یکی از آن‌ها را کنار بگذاریم و LCS را برای بقیه‌ی رشته محاسبه کنیم. یعنی $LCS(x, y)$ برابر است با:

$$\max(LCS(\langle x_1x_2...x_n\rangle, \langle y_1y_2...y_{m-1}\rangle), LCS(\langle x_1x_2...x_{n-1}\rangle, \langle y_1y_2...y_m\rangle)) \ (*)$$

تمرین: گزاره های مشخص شده با ستاره را به عنوان تمرین ثابت کنید.

حال این‌جاست که DP وارد می‌شود! فرض کنید `dp[i][j]` : طول بزرگترین زیردنباله مشترک `x[1:i]=\langle` `x_1x_2...x_i\rangle` و `y[1:j]=\langle y_1y_2...y_j\rangle` باشد.

طبق سخنان بالا، این DP به صورت زیر آپدیت می‌شود:

$$dp[i][j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ dp[i-1][j-1] + 1 & x_i = y_j \\ \max(dp[i][j-1], dp[i-1][j]) & x_i \neq y_j \end{cases}$$

در زیر، کدِ این سوال را می‌توانید مشاهده کنید.

In [9]:
```python
# تعریف رشته‌های ورودی
x = "ABCDGH"
y = "AEDFHR"

# ایجاد جدول dp طول برای هر زیررشته LCS برای نگهداری
dp_lcs = [[0] * (len(y) + 1) for _ in range(len(x) + 1)]

# پر کردن جدول dp با استفاده از برنامه‌نویسی پویا
for i in range(1, len(x) + 1):
    for j in range(1, len(y) + 1):
        if x[i - 1] == y[j - 1]:
            # اضافه می‌کنیم LCS اگر کاراکترها برابر باشند، به طول قبلی ۱
            dp_lcs[i][j] = dp_lcs[i - 1][j - 1] + 1
        else:
            # در غیر این صورت، بیشترین مقدار بین حذف یکی از کاراکترها را انتخاب می‌کنیم
            dp_lcs[i][j] = max(dp_lcs[i][j - 1], dp_lcs[i - 1][j])

# چاپ طول بلندترین زیردنباله مشترک
print(dp_lcs[len(x)][len(y)])
```

```
3
```

تحلیل زمانی این راه حل ⏱️

تعداد خانه‌هایی از آرایه `dp_lcs` که پر می‌شود، از `O(n×m)` است، و همچنین آپدیت کردن هر خانه از این آرایه، از `O(1)` طول می‌کشد. پس مرتبه زمانی کلّی برنامه، از `O(nm)` است (اگر طول رشته‌های ورودی برابر باشد، از `O(n^2)` است).

تمرین‌ها 📝

تمرین: آیا می‌توانیم حافظه مصرفی برنامه بالا را کاهش دهیم؟

تمرین: برنامه‌ی بالا را به گونه‌ای تغییر دهید که بتوانیم خودِ رشته‌ی LCS را نیز پیدا کنیم.

مساله چهارم: تراز دنباله‌ها (Sequence Alignment) 🧬

دو رشته‌ی `X=\langle x_1x_2\dots x_n\rangle` و `Y=\langle y_1y_2\dots y_m\rangle` متشکل از کاراکترهایی به ما داده شده‌است. می‌خواهیم تعدادی فاصله (gap) در رشته‌ها اضافه کنیم، تا به ازای هر کاراکتر از `x` مانند `x_i` ، تناظری یک‌به‌یک بین آن و کاراکتر `i`-ام از `y` (`y_i`) ایجاد شود. به ازای هر فاصله‌ی ایجاد شده در رشته‌ها، هزینه‌ی `δ` را متحمل می‌شویم، و به ازای هر تناظر بین کاراکترهای `p` و `q` ، هزینه‌ای برابر با `α_{pq}` دارد. (مشخصاً `α_{pp}` برابر با صفر است.) حال می‌خواهیم فاصله‌ها را به گونه‌ای انتخاب کنیم که هزینه‌ی کلی کمینه شود.

نکته: اکیدا توصیه می‌شود قبل از خواندن راه حل دقایقی خودتان به دنبال راه حل این مساله بگردید.

کاربرد این مسئله، مانند مسئله‌ی قبل، در پیدا کردن میزان مشابهت بین دو رشته است. مثلا فرض کنید می‌خواهید رشته‌ای را در یک نرم‌افزار فرهنگ لغت جستجو کنید. اگر کلمه‌ی مورد جستجو، در فرهنگ لغت موجود نباشد (به طول مثال `ocurrance`) این نرم‌افزار از شما می‌پرسد که آیا منظور شما `occurrence` بوده است؟ این نرم‌افزار چگونه شباهت میان این دو رشته را متوجه می‌شود؟

به طور کلی، مسئله به این تبدیل می‌شود که باید معیاری برای شباهت میان دو رشته پیدا کنیم. به طور مثال، می‌توان رشته‌ی `ocurrance` را با اضافه کردن یک کاراکتر `c` به آن و تغییر کاراکتر `a` به `e`، به رشته‌ی دوم تبدیل کرد.

به تعبیری دیگر، می‌توان این دو رشته را به این صورت، زیر هم ردیف کرد:

```
o-currance
occurrence
```

هر فاصله، به این معنی است که در این محل، باید کاراکتری اضافه شود. و همچنین، بعضی از کاراکترها با کاراکتر متناظر، یکسان نیستند. در این مثال، میزان تفاوت دو رشته از روی فاصله‌های اضافه شده و کاراکترهایی که با حرف متناظر خود متفاوت هستند محاسبه می‌شود. برای مثال، در این حالت، یک فاصله اضافه شده و یک عدم تطابق داریم. در روشی دیگر، می‌توان رشته‌ها را به این صورت با هم متناظر کرد:

```
o-curr-ance
occurre-nce
```

در این حالت، سه فاصله اضافه شده است و هیچ عدم تطابقی نداریم. حال بسته به هزینه‌ی هر کدام از این موارد، می‌توان مشخص کرد که کدام تطابق بهتر است. همچنین، مانند مثال قبل، پیدا کردن میزان مشابهت بین دو رشته می‌تواند در مسائل زیست‌شناسی نیز به کار گرفته شود.

برای حل این سوال، می‌توان راهکارهایی ساده با زمان نمایی پیشنهاد داد، که خیلی بهینه نیستند. برای حل این سوال با زمان چندجمله‌ای، از برنامه‌نویسی پویا کمک می‌گیریم.

هر تناظر بین کاراکترهای دو رشته را می‌توان به صورت مجموعه‌ای از زوج مرتب‌ها نمایش داد، که مؤلفه‌های زوج مرتب‌ها، نشان‌دهنده‌ی اندیسی از هر رشته هستند که با هم تناظر داده شده‌اند. برای مثال در تناظر:

```
stop-
-tops
```

می‌توان این تناظر را با مجموعه‌ی `M = {(2, 1), (3, 2), (4, 3)}` نیز نمایش داد. می‌توان نشان داد که در مجموعه‌ی زوج‌مرتب‌های مربوط به یک تناظر، هیچ دو زوج‌مرتبی با هم تقاطع ندارند؛ به ازای `M ∋('i', j) ,(i,j)`، اگر `i`، آن‌گاه `j`.

لم: اگر در یک تناظر `M` بین دو رشته‌ی `X` به طول `n` و `Y` به طول `m`، اگر `M ∌(n, m)`، آن‌گاه یا کاراکتر `n`-ام از `X` در تناظر، روبه‌روی فاصله قرار گرفته‌است، یا کاراکتر `m`-ام از `Y` روبه‌روی فاصله قرار گرفته‌است.

این لم را به عنوان تمرین اثبات کنید.

با استفاده از لم بالا، می‌توان نتیجه گرفت که در تناظر بهینه (تناظر با کمترین هزینه) `M`، حداقل یکی از گزاره‌های زیر اتفاق می‌افتد:

۱- کاراکتر `n`-ام از `X` با کاراکتر `m`-ام از `Y` تطابق می‌یابد.

۲- کاراکتر `n`-ام از `X` با فاصله تطابق می‌یابد.

۳- کاراکتر `m`-ام از `Y` با فاصله تطابق می‌یابد.

حال `dp[i][j]` را به صورت زیر تعریف می‌کنیم:

`dp[i][j]`: کمترین هزینه برای متناظر کردن دو رشته‌ی `X_i=\langle x_1x_2\dots x_i\rangle` و `Y_j=\langle y_1y_2\dots y_j\rangle`.

با استفاده از سه حالت تطابق بین دو کاراکتر آخر رشته‌ها، می‌توان رابطه‌ی زیر را برای محاسبه `dp[i][j]` نوشت:

$$dp[i][j] = \min(\alpha_{x_i,y_j} + dp[i-1][j-1], \delta + dp[i-1][j], \delta + dp[i, j-1])$$

حالت پایه: `dp[i][0]=dp[0][i]=iδ` ؛ زیرا تنها راه تناظر یک رشته به طول `i` و یک رشته به طول صفر، افزودن `i` کاراکتر فاصله به رشته‌ی به طول صفر است.

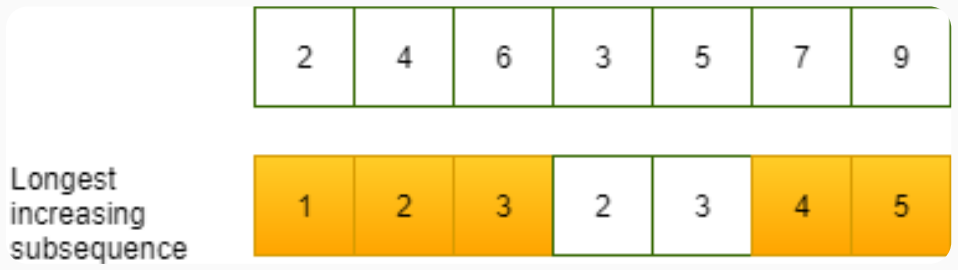
می‌توان کد این الگوریتم را به صورت top-down (با استفاده از memoization) یا bottom-up پیاده‌سازی کرد. در زیر، کد آن به روش bottom-up آمده است.

In [10]:
```python
# تعریف رشته‌های ورودی
x = "AGGGCT"
y = "AGGCA"

# تعریف هزینه فاصله و ماتریس هزینه عدم تطابق
MAX_CHAR = 255
delta = 3
alpha = [[0] * (MAX_CHAR + 1) for _ in range(MAX_CHAR + 1)]

# مقداردهی ماتریس alpha: هزینه عدم تطابق بین کاراکترها
for i in range(MAX_CHAR + 1):
    for j in range(MAX_CHAR + 1):
        alpha[i][j] = 0 if i == j else 2

# ایجاد جدول dp برای نگهداری کمترین هزینه تطابق
dp_seq = [[0] * (len(y) + 1) for _ in range(len(x) + 1)]

# مقداردهی اولیه: تطابق با رشته‌ی خالی فقط با فاصله ممکن است
for i in range(len(x) + 1):
    dp_seq[i][0] = i * delta
for j in range(len(y) + 1):
    dp_seq[0][j] = j * delta

# با استفاده از سه حالت ممکن dp پر کردن جدول
for i in range(1, len(x) + 1):
    for j in range(1, len(y) + 1):
        cost_match = alpha[ord(x[i - 1])][ord(y[j - 1])] + dp_seq[i - 1][j - 1]
        cost_gap_x = delta + dp_seq[i - 1][j]
        cost_gap_y = delta + dp_seq[i][j - 1]
        dp_seq[i][j] = min(cost_match, cost_gap_x, cost_gap_y)

# چاپ کمترین هزینه تطابق دو رشته
print(dp_seq[len(x)][len(y)])
```

```
5
```

مساله‌ی پنجم: بلندترین زیردنباله‌ی صعودی (Longest Increasing Subsequence یا LIS) 📈

- مثال تصویری از بلندترین زیردنباله‌ی صعودی (LIS)



arr = [1,3,2,3,5,6,7] LIS = [1,2,3,5,6,7]

روش بازگشتی برای LIS 🔹

فرض کنید به ما آرایه `arr[0..n-1]` داده شده‌است و می‌خواهیم بلندترین توالی صعودی را پیدا کنیم.

به عنوان یک روش کاربردی می‌توان مراحل زیر را طی کرد:

- فرض می‌کنیم `L(i)` طول LIS است که در آن `arr[i]` آخرین عضو آن باشد.
- حال می‌توانیم `L(i)` را به صورت بازگشتی به شکل زیر بدست آوریم.

L(i) = 1 + max(L(j)) where 0 < j < i and arr[j] < arr[i]; or L(i) = 1, if no such j exists.

جواب نهایی LIS 🎯

برای به‌دست آوردن طول بلندترین زیردنباله صعودی، کافی است `max(L(i))` را محاسبه کنیم.

In [11]:
```python
# تابع بازگشتی برای محاسبه طول بلندترین زیردنباله صعودی (LIS)
def _lis(arr, n, max_ref):
    # حالت پایه: فقط یک عنصر داریم
    if n == 1:
        return 1

    max_ending_here = 1  # طول LIS که با arr[n-1] تمام می‌شود

    for i in range(1, n):
        res = _lis(arr, i, max_ref)
        if arr[i - 1] < arr[n - 1] and res + 1 > max_ending_here:
            max_ending_here = res + 1

    # به‌روزرسانی مقدار بیشینه کلی
    if max_ref[0] < max_ending_here:
        max_ref[0] = max_ending_here

    return max_ending_here
```

In [12]:
```python
# تابع اصلی برای محاسبه طول LIS
def lis(arr):
    n = len(arr)
    max_ref = [1]  # استفاده از لیست برای ارجاع‌پذیری
    _lis(arr, n, max_ref)
    return max_ref[0]
```

In [13]:
```python
# تست با آرایه نمونه
arr = [10, 22, 9, 33, 21, 50, 41, 60]
print("Length of LIS is", lis(arr))
```

```
Length of LIS is 5
```

مثال عملی LIS 📊

فرض کنید می‌خواهیم الگوریتم بالا را روی یک آرایه با اندازه ۴ محاسبه کنیم.

lis(4) / | lis(3) lis(2) lis(1) / / lis(2) lis(1) lis(1) / lis(1)

بهینه‌سازی با DP 💡

با توجه به درخت محاسباتی، مشاهده می‌شود که بعضی از اعضای آرایه بارها و بارها محاسبه می‌شوند. بنابراین بهتر است که از رویکرد برنامه‌نویسی پویا (DP) استفاده کرد. به قطعه کد زیر توجه کنید.

In [14]:
```python
def binary_search(tail, target):
    # پیدا کردن اولین اندیسی که tail[idx] >= target
    left, right = 0, len(tail) - 1
    while left <= right:
        mid = (left + right) // 2
        if tail[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return left  # موقعیت مناسب برای جایگزینی یا افزودن
```

In [15]:
```python
def lis_nlogn_manual(arr):
    tail = []  # لیست کمکی برای نگهداری انتهای زیردنباله‌ها

    for num in arr:
        idx = binary_search(tail, num)
        if idx == len(tail):
            tail.append(num)  # عدد جدید بزرگ‌تر از همه است
        else:
            tail[idx] = num  # جایگزینی برای حفظ کمترین مقدار ممکن

    return len(tail)
```

In [16]:
```python
# تست با داده‌ی نمونه
arr = [10, 22, 9, 33, 21, 50, 41, 60]
print("Length of LIS is", lis_nlogn_manual(arr))
```

```
Length of LIS is 5
```

تحلیل زمانی و بهینه‌سازی LIS ⏱️

پیچیدگی زمانی الگوریتم بالا `O(n^2)` می‌باشد. حال می‌خواهیم الگوریتمی از مرتبه زمانی `O(n \log n)` برای پیدا کردن LIS ارائه دهیم.

ایده اصلی الگوریتم O(n log n) برای LIS 💡

ایده اصلی این الگوریتم این است که همه توالی‌های فعال را نگه داریم و براساس عدد جدید، این توالی‌ها را به‌روزرسانی کنیم. به مثال زیر توجه شود.

Á = [2,8,7] subseq1 = [2,7], subseq2 = [2,8]

مرحله بعدی الگوریتم LIS 🔄

حال اگر عدد `11` را به آرایه `A` اضافه کنیم، چه اتفاقی می‌افتد؟

A = [2,8,7,11] subseq1 = [2,7,11], subseq2 = [2,8,11]

مرحله بعدی الگوریتم LIS 🔄

حال اگر عدد `9` را به آرایه `A` اضافه کنیم، چه اتفاقی می‌افتد؟

A = [2,8,7,11,9] subseq1 = [2,7,9], subseq2 = [2,8,9]

تصمیم‌گیری در الگوریتم LIS 🤔

در واقع تصمیم‌گیری ما بر این اساس است که باید ببینیم با اضافه کردن عدد `9` آیا می‌توان توالی ۳تایی را حفظ کرد یا ارتقا بخشید، یا بهتر است که با `11` ادامه دهیم؟

علت انتخاب ما این بود که در صورت اضافه شدن عدد `10` ، توالی طولانی‌تری خواهیم داشت.

تصمیم‌گیری در افزودن عدد جدید به توالی 🔄

حال سوال اینجاست که تصمیم‌گیری ما پیرامون اینکه آیا عدد جدید را قرار دهیم یا جایگزین کنیم یا با همان اعداد قبلی ادامه دهیم چگونه خواهد بود؟

ما می‌توانیم عدد `A[i]` را به یک توالی موجود اضافه کنیم اگر یک عدد `A[j]` که (`i < j`) وجود داشته باشد به طوری که:

E < A[i] < A[j] or (E > A[i] < A[j] : for replace)

مثال عملی تصمیم‌گیری LIS 🔹

به طوری که `E` آخرین عضو توالی است.

در مثال بالا:

`A[j] = 10` و `E = 11`، `A[i] = 9`.

In [17]:
```python
def ceil_index(tail, left, right, key):
    while right - left > 1:
        mid = left + (right - left) // 2
        if tail[mid] >= key:
            right = mid
        else:
            left = mid
    return right
```

In [18]:
```python
def lis3(arr):
    if not arr:
        return 0

    tail = [0] * len(arr)
    length = 1  # طول LIS فعلی

    tail[0] = arr[0]

    for i in range(1, len(arr)):
        if arr[i] < tail[0]:
            # مقدار جدید کوچکتر از همه – جایگزین اولین عنصر
            tail[0] = arr[i]
        elif arr[i] > tail[length - 1]:
            # اضافه می‌شود LIS بزرگ‌تر از همه – به انتهای
            tail[length] = arr[i]
            length += 1
        else:
            # جایگزینی با کوچک‌ترین مقداری که >= arr[i] است
            idx = ceil_index(tail, -1, length - 1, arr[i])
            tail[idx] = arr[i]

    return length
```

In [19]:
```python
v = [10, 22, 9, 33, 21, 50, 41, 60]
print("Length of LIS is", lis3(v))
```

```
Length of LIS is 5
```

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل سوم: برنامه‌نویسی پویا

بخش سوم: مجموعه مستقل در درخت

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 📑

In [1]:
```
def max(x, y):
    return x if x > y else y
```

مجموعه مستقل در درخت 🌳

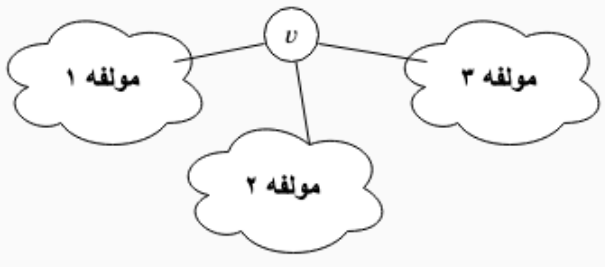
در این بخش می‌خواهیم به بررسی مجموعه‌های مستقل در گراف‌ها بپردازیم، و مسئله‌ای درباره پیدا کردن بزرگ‌ترین مجموعه مستقل در یک درخت را حل کنیم. برای شروع اول نیاز به تعریف یک مفهوم مهم داریم:

- **مجموعه مستقل:** به مجموعه‌ای رئوس گراف می‌گوییم که هیچ یالی بین اعضای این مجموعه وجود نداشته باشد.

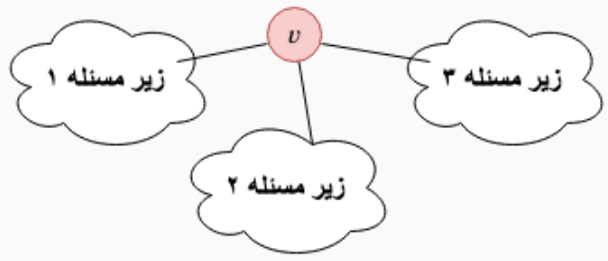
حال با دانستن این تعریف، مسئله روشن است: اندازه بزرگ‌ترین مجموعه مستقل برای یک درخت مانند **T** چند است؟

کلیت راه‌حل 🔹

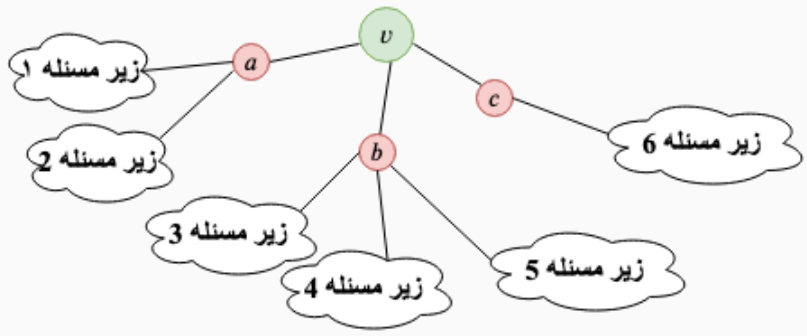
بیاید بررسی کنیم ببینیم جواب این مسئله چه شرایطی ممکن است داشته باشد. یک راس مانند **v** را در نظر بگیرید. این راس از طریق یال‌ها به تعدادی راس دیگر وصل است که می‌دانیم به دلیل درخت بودن گراف، با در نظر نگرفتن این راس و یال‌های متصل به آن، این گراف به چندین مولفه همبندی تقسیم می‌شود. مثلا در شکل زیر، راس **v** با سه یال به سه مولفه همبندی متصل است.



این راس یا در جواب نهایی قرار دارد یا ندارد. اگر بدانیم که این راس در جواب نهایی قرار ندارد، مطابق شکل زیر، مسئله به ۳ زیر مسئله کوچک‌تر تقسیم می‌شود که در هر یک از این مسائل باید جواب بهینه را به صورت بازگشتی پیدا کنیم.



اما اگر در جواب نهایی قرار داشته باشد چطور؟ در این صورت می‌دانیم که همسایگان این راس در جواب قرار ندارند. همانند حالت قبل، همسایگان همسایگان راس **v** به چندین مولفه همبندی تقسیم می‌شوند و نیاز داریم که مسئله را در این زیر مسئله‌ها به صورت بازگشتی حل کنیم.



تا به اینجا دیدیم که اگر بدانیم یک راس مانند **v** قرار دارد یا ندارد، مسئله را می‌توانیم به زیر مسائلی تقسیم کنیم. اما سوال اصلی همچنان مانده: **v** در جواب نهایی قرار دارد یا ندارد؟ از آنجایی که نمی‌توانیم از قبل جواب این سوال را بفهمیم، مسئله را در هر دو حالت حل می‌کنیم، و بینشان بیشینه را انتخاب می‌کنیم.
پس به عبارتی، جواب این مسئله به این ترتیب خواهد بود:

$$
\begin{aligned}
&\max\{ \\
&\text{1 + مجموع جواب زیرمسائل با } \mathbf{v} \text{ ,} \\
&\text{مجموع جواب زیرمسائل بدون } \mathbf{v} \\
&\} = \text{جواب}
\end{aligned}
$$

در ادامه برای آنکه راحت‌تر بتوانیم مسئله را حل کنیم، نمایش ریشه‌دار درخت را با راس **r** در نظر می‌گیریم. با این نمایش همه رئوس بچه یا نوادگان راس **r** خواهند بود. برای هر راس مانند **u** این گراف در این نمایش، زیر مسئله **LIS(u)** را برابر اندازه بزرگ‌ترین زیرمجموعه مستقل در زیرگرافی که **u** ریشه آن است می‌نامیم. جواب مسئله اصلی هم برابر خواهد بود با **LIS(r)**. همچنین طبق آنچه در بالا گفتیم، برای هر **v** رابطه زیر برقرار خواهد بود:

$$LIS(v) = \max\{\Sigma_{u \in children(v)} LIS(u), 1 + \Sigma_{u \in grandchildren(v)} LIS(u)\}$$

تلاش اول: بازگشتی 🌳

طبق فرم ولی که در ریزبخش قبل به دست آوردیم، به نظر می‌آید به صورت بازگشتی می‌توان مسئله را حل کرد. برای سادگی، فرض می‌کنیم درخت مورد نظر، یک درخت ریشه‌دار دودویی است (هر راس حداکثر دو فرزند دارد). حل مسئله به صورت مشابه برای حالت کلی به خواننده واگذار می‌شود. برای این منظور کلاسی برای نگه‌داری رئوس گراف تعریف می‌کنیم:

In [2]:
```python
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

پیاده‌سازی تابع بازگشتی 🔁

حال کافیست تابع بازگشتی برای محاسبه جواب را به شکل زیر پیاده‌سازی کنیم:

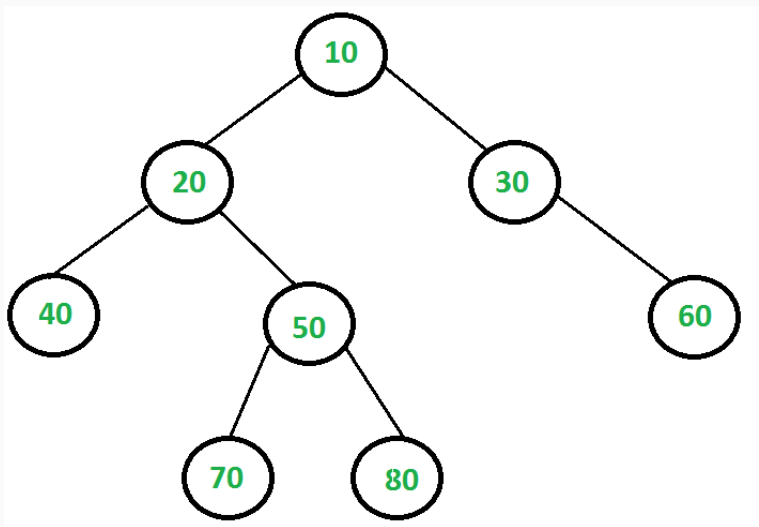
In [3]:
```python
def LIS_recursive(root):
    if root is None:
        return 0

    # حالت ۱: ریشه در جواب نیست
    size_excl = LIS_recursive(root.left) + LIS_recursive(root.right)

    # حالت ۲: ریشه در جواب هست، پس فرزندانش نباید باشند
    size_incl = 1
    if root.left:
        size_incl += LIS_recursive(root.left.left)
        size_incl += LIS_recursive(root.left.right)
    if root.right:
        size_incl += LIS_recursive(root.right.left)
        size_incl += LIS_recursive(root.right.right)

    return max(size_incl, size_excl)
```

In [4]:
```python
# ساخت درخت نمونه
root = Node(10)
root.left = Node(20)
root.right = Node(30)
root.left.left = Node(40)
root.left.right = Node(50)
root.right.right = Node(60)

# محاسبه و چاپ جواب
print("Size of Largest Independent Set is", LIS_recursive(root))
```

```
Size of Largest Independent Set is 4
```

مثال: حل مسئله برای یک درخت دودویی 🌳

حال می‌توانیم جواب را برای یک درخت دودویی دلخواه به دست آوریم. به عنوان مثال برای درخت زیر مسئله را حل می‌کنیم:



In [6]:
```python
# ساخت درخت مطابق مثال
root = Node(10)
root.left = Node(20)
root.left.left = Node(40)
root.left.right = Node(50)
root.left.right.left = Node(70)
root.left.right.right = Node(80)
root.right = Node(30)
root.right.right = Node(60)

print("Size of the Largest Independent Set is", LIS_recursive(root))
```

```
Size of the Largest Independent Set is 5
```

تمرین ⏱

تمرین: پیچیدگی زمانی این روش را محاسبه کنید.

جواب کوتاه: نمایی

تلاش دوم: برنامه‌نویسی پویا 💻

برای برطرف کردن مشکلات راه‌حل قبلی، نگه‌داری جواب زیرمسائلی هست که به صورت بازگشتی حل می‌شود، به عبارتی همان برنامه‌نویسی پویا. برای این منظور کد قسمت قبل را به شکل زیر تغییر می‌دهیم تا جواب‌های زیرمسئله‌ها را ذخیره کنیم:

In [7]:
```python
class Node:
    def __init__(self, number):
        self.number = number
        self.children = []
        self.D = 0   # شامل این گره
        self.Dp = 0  # بدون این گره
```

محاسبه جواب برای یک راس خاص 🎯

جواب مسئله را هم برای یک راس خاص به این ترتیب می‌توانیم به دست آوردیم (به خط هفتم توجه کنید):

In [8]:
```python
def DP(v):
    v.D = 1  # اگر این گره در جواب باشد، خودش را حساب می‌کنیم

    for child in v.children:
        DP(child)
        v.D += child.Dp  # اگر این گره در جواب باشد، فرزندانش نباید باشند
        v.Dp += max(child.D, child.Dp)  # اگر این گره نباشد، فرزندان آزادند

    print(f"Node {v.number}: D = {v.D}, Dp = {v.Dp}")
```

محاسبه جواب مثال قبل 🧮

برای همان مثال قبل، جواب را به صورت زیر محاسبه می‌کنیم:

In [10]:
```python
# ساخت گره‌ها
nodes = {i: Node(i) for i in range(1, 16)}
root = Node(0)

# ساخت ساختار درخت
nodes[5].children.append(nodes[10])
nodes[1].children.extend([nodes[4], nodes[5], nodes[6]])
nodes[8].children.extend([nodes[11], nodes[12]])
nodes[9].children.extend([nodes[13], nodes[14], nodes[15]])
nodes[3].children.extend([nodes[7], nodes[8], nodes[9]])
root.children.extend([nodes[1], nodes[2], nodes[3]])

DP(root)
print("Size of the Largest Independent Set is", max(root.D, root.Dp))
```

```
Node 4: D = 1, Dp = 0
Node 10: D = 1, Dp = 0
Node 5: D = 1, Dp = 1
Node 6: D = 1, Dp = 0
Node 1: D = 2, Dp = 3
Node 2: D = 1, Dp = 0
Node 7: D = 1, Dp = 0
Node 11: D = 1, Dp = 0
Node 12: D = 1, Dp = 0
Node 8: D = 1, Dp = 2
Node 13: D = 1, Dp = 0
Node 14: D = 1, Dp = 0
Node 15: D = 1, Dp = 0
Node 9: D = 1, Dp = 3
Node 3: D = 6, Dp = 6
Node 0: D = 10, Dp = 10
Size of the Largest Independent Set is 10
```

پیچیدگی زمانی الگوریتم ⏱️

اما پیچیدگی زمانی این الگوریتم چگونه است؟

می‌دانیم که جواب مسئله به ازای هر راس یک بار محاسبه می‌شود و از آنجایی که هر راس حداکثر ۲ فرزند و ۴ نوه دارد، با دانستن جواب زیرمسئله برای فرزندان و نوادگان، جواب برای هر راس با تعداد ثابتی عمل به دست می‌آید. پس تعداد عملیات مورد استفاده ضریبی از تعداد رئوس است. بنابراین پیچیدگی زمانی الگوریتم از **O(n)** است.

تمرین ⏱️

تمرین: مسئله را در حالتی که درخت دودویی نباشد حل کنید و پیچیدگی زمانی‌اش را محاسبه کنید.

راهنمایی: تعداد فرزندان و نودگان هر راس دیگر ثابت نیست، پس تحلیل قبلی درست نیست.

منابع 📚

کدهای استفاده شده در این بخش با تغییر از این سایت برداشته شده است.

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل چهارم: گراف

بخش اول: درخت پوشای کمینه

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 📑

مسئله درخت پوشای کمینه 🌳

شبکه‌ای از کامپیوترها را در نظر بگیرید که با تعدادی لینک دوطرفه به هم متصل شده باشند. فرض کنید هر لینک هزینه‌ای برای انتقال پیام داشته باشد. حال ما می‌خواهیم پیامی را از یک کامپیوتر دلخواه با استفاده از تعدادی از لینک‌ها به بقیه‌ی کامپیوترها انتقال دهیم. اگر فرض کنیم هزینه‌ی این انتقال برابر با جمع هزینه‌ی لینک‌هایی باشد که استفاده می‌کنیم و بخواهیم این هزینه را کمینه کنیم، این مسئله به گرافی با یال‌های وزن‌دار با وزن‌های مثبت مدل می‌شود که می‌خواهیم تعدادی از یال‌های آن را انتخاب کنیم به‌طوری‌که جمع وزن آن‌ها کمینه شود و گرافی همبند تشکیل دهند.

می‌توان دید که چنین زیرگرافی باید حتماً یک درخت را تشکیل دهد؛ زیرا اگر دوری داشته باشد، با حذف هر یال دور همچنان همبند می‌ماند و چون یال‌ها وزن مثبت دارند، با حذف آن جمع وزن‌ها کمتر می‌شود. بنابراین، این زیرگراف درختی شامل تمام رأس‌های گراف می‌شود که کمترین جمع وزن را دارد و به آن **Minimum Spanning Tree (MST)** می‌گویند.

ساختن جاده‌هایی برای وصل کردن تعدادی شهر با کمترین هزینه، طراحی مدارها و ... مثال‌های دیگری از استفاده‌های درخت پوشای کمینه هستند. در مسئله‌ی درخت پوشای کمینه می‌خواهیم برای یک گراف، درخت پوشای کمینه‌ی آن را پیدا کنیم.

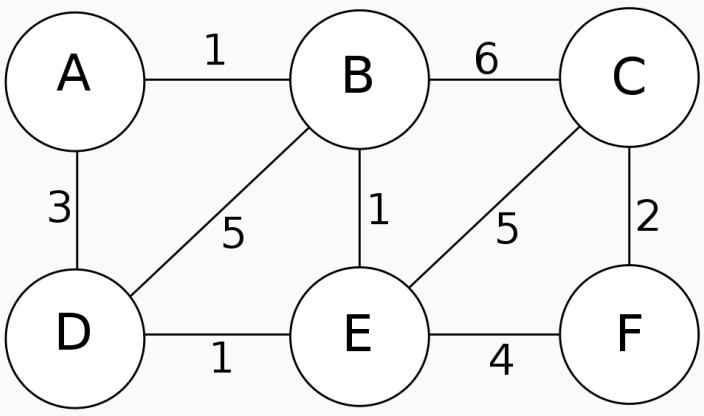
الگوریتم Prim ⚡

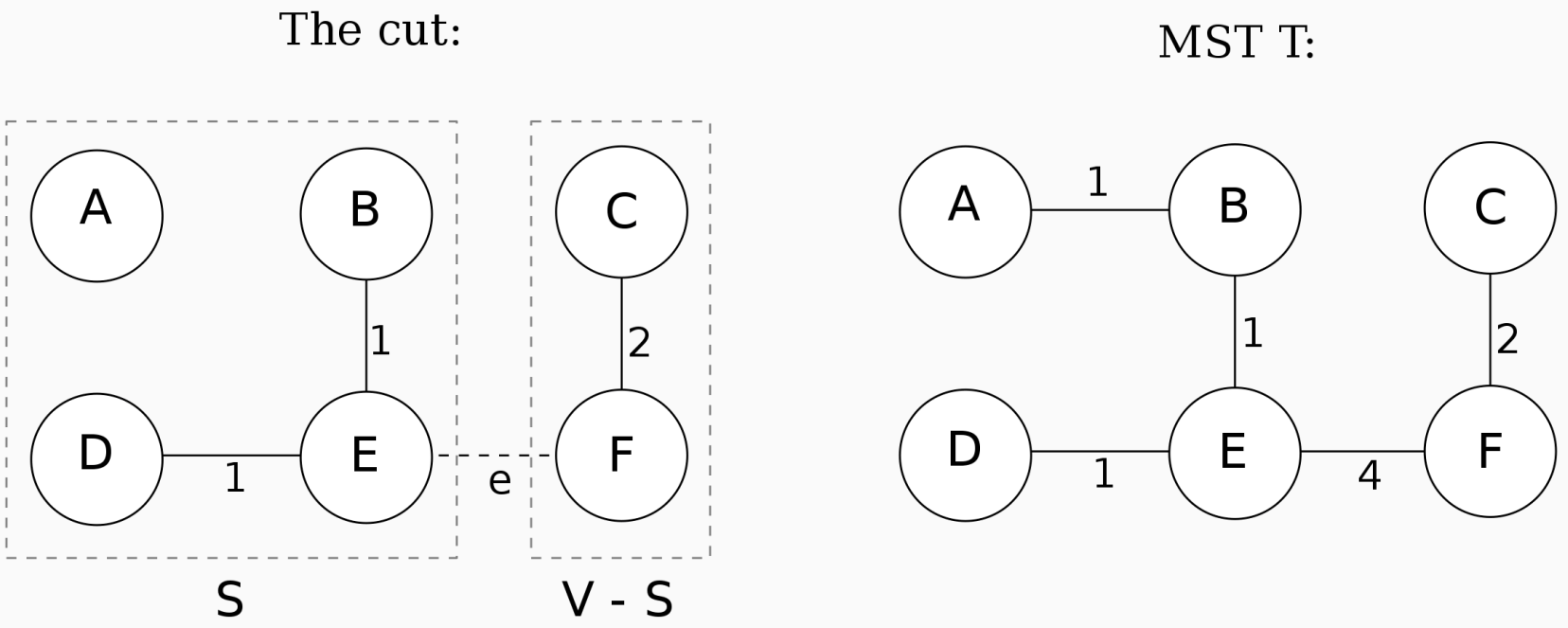
می‌خواهیم الگوریتمی برای یافتن MST در یک گراف همبند پیدا کنیم.
فرض کنید می‌خواهیم یال به یال درخت پوشا را بیابیم. آیا یالی وجود دارد که بتوان گفت حتماً در جواب وجود دارد یا ندارد؟
می‌توان دید که درخت پوشای کمینه حتماً شامل کم‌وزن‌ترین یال گراف می‌شود؛ اگر این یال در درخت پوشای کمینه نباشد، با اضافه کردن آن دوری تشکیل می‌شود که با حذف هر یال دیگر آن باز یک درخت پوشا خواهیم داشت که وزنش کمتر یا مساوی وزن درخت پوشای قبلی می‌شود. چون یک یال با کمترین وزن جایگزین شده است، پس این درخت حتماً یک درخت پوشای کمینه

خواهد بود که شامل کم‌وزن‌ترین یال است.

می‌توان با همین استدلال دید که کم‌وزن‌ترین یال متصل به هر راس حتماً در یک MST ظاهر می‌شود و با تعمیم آن، **ویژگی برش** را داریم:

برای هر افراز راس‌های گراف به دو دسته، یال‌های بین این دو دسته را در نظر می‌گیریم. هر یالی که بین این یال‌ها کمترین وزن را داشته باشد حتماً در یک MST ظاهر شده‌است.





اثبات این ویژگی نیز مانند اثبات آمدن یال کمینه در MST است؛ اگر این یال در یک MST نباشد، با اضافه کردن آن دوری تشکیل می‌شود که در این دور یک یال باید از یال‌های بین دو دسته افراز باشد، چون مسیر بین دو سر یال اضافه شده در درخت، یک سرش در یک دسته و سر دیگرش در دسته دیگر است. با حذف یال درون درخت و اضافه کردن یال مورد نظر، زیرگراف درخت باقی می‌ماند چون یکی از یال‌های دور حذف شده است و وزن درخت نیز بیشتر نمی‌شود، پس این نیز حتماً یک MST است.
اکثر الگوریتم‌های MST بر پایه ویژگی برش هستند.

با استفاده از همین ویژگی می‌توانیم الگوریتمی برای یافتن MST پیدا کنیم: در هر مرحله یک درخت داریم که یال‌های آن حتماً زیرمجموعه یک MST هستند. حال با افراز راس‌های گراف به راس‌های این درخت و بقیه راس‌های گراف، کمترین یال بین این دو دسته را به درختمان اضافه می‌کنیم و فرضمان طبق ویژگی برش درست می‌ماند و درخت جدید نیز زیرمجموعه یک MST خواهد بود چون یال اضافه شده عضو MST است.
برای شروع الگوریتم نیز می‌توانیم از یک راس تنها شروع کنیم که شرط را دارد. با این کار الگوریتم **Prim** را به صورت زیر داریم:
در این الگوریتم یک مجموعه رئوس **S** داریم که در ابتدا شامل یک راس **v** است و در هر مرحله بین یال‌هایی که یک سرشان در **S** و یک سرشان خارج از **S** است، یالی که کمترین وزن را دارد را می‌گیریم و آن را به MST و سر خارج از **S** آن را به **S** اضافه می‌کنیم. این مراحل را ادامه می‌دهیم تا راسی خارج از **S** نباشد، که در این صورت MST را یافته‌ایم.

In [1]:
```
# تعریف ثابت‌ها
N = 5
```

```
INF = 10**9 + 100
```

In [2]:
```
def prim(parent, adj):
    # آرایه‌ای برای مشخص کردن اینکه آیا رأس در مجموعه MST قرار دارد یا نه
    in_set = [False] * N

    # بهترین یال برای اتصال هر رأس به مجموعه فعلی
    best_edge = [INF] * N
    best_edge[0] = 0  # شروع از رأس ۰
    parent[0] = -1    # رأس ۰ ریشه درخت خواهد بود

    min_cost = 0  # هزینه کل درخت پوشای کمینه

    for _ in range(N):
        v = -1
        # پیدا کردن رأسی که کمترین یال به مجموعه دارد
        for j in range(N):
            if not in_set[j] and (v == -1 or best_edge[v] > best_edge[j]):
                v = j

        min_cost += best_edge[v]  # اضافه کردن وزن یال انتخاب‌شده
        in_set[v] = True          # اضافه کردن رأس به مجموعه MST

        # به‌روزرسانی بهترین یال‌ها برای رأس‌های خارج از مجموعه
        for u in range(N):
            if adj[v][u] != 0 and not in_set[u] and adj[v][u] < best_edge[u]:
                best_edge[u] = adj[v][u]
                parent[u] = v

    return min_cost
```

In [3]:
```
# تعریف ماتریس مجاورت گراف وزن‌دار
g = [
    [0, 3, 4, 0, 0],
    [3, 0, 2, 0, 1],
    [4, 2, 0, 9, 0],
    [0, 0, 9, 0, 5],
    [0, 1, 0, 5, 0],
]

p = [0] * N  # آرایه پدرها برای بازسازی درخت

# اجرای الگوریتم و چاپ نتیجه
print(prim(p, g))
for i in range(N):
    print(p[i], end=' ')
print()
```

```
11
-1 0 1 4 1
```

تحلیل زمانی ⏳

این پیاده‌سازی از $O(n^2)$ زمان می‌برد زیرا حلقه بیرونی n بار اجرا می‌شود و در هر اجرا کمینه یال با یک سر در مجموعه در $O(n)$ پیدا می‌شود و کمترین یال بقیه راس‌ها نیز در $O(n)$ آپدیت می‌شود.

در هر مرحله در این پیاده‌سازی $inSet$ نشان می‌دهد کدام راس‌ها درون S اند و $bestEdge$ برای راس‌های بیرون از S وزن سبک‌تریم یال آن‌ها که سر دیگرش در S است را دارد و برای یافتن سبک ترین یال از S به بیرون آن کمترین $bestEdge$ برای راس‌های خارج از S را می‌گیریم و عملیات گفته شده را انجام می‌دهیم. اگر به جای ماتریس مجاورت از لیست مجاورت استفاده کنیم و برای یافتن راس با یال کمینه خارج از S از هرم کمینه استفاده کنیم زمان اجرای الگوریتم برابر $O(mlogn)$ می‌شود که این پیاده‌سازی نیز در زیر آمده‌است. همچنین اگر در این پیاده‌سازی از هرم فیبوناچی استفاده کنیم، زمان اجرا برابر $O(nlogn + m)$ می‌شود.

In [4]:
```
# پیاده‌سازی الگوریتم Prim با استفاده از لیست مجاورت و صف اولویت دستی
def prim(parent, adj):
    # صف اولویت برای نگهداری یال‌های کاندید با کمترین وزن
    candidates = []

    # مشخص‌کننده اینکه آیا رأس در مجموعه MST قرار دارد یا نه
    in_set = [False] * N

    # بهترین یال برای اتصال هر رأس به مجموعه فعلی
    best_edge = [INF] * N
    best_edge[0] = 0
    parent[0] = -1

    # اضافه کردن رأس شروع به صف اولویت
    candidates.append((0, 0))  # (وزن، شماره رأس)

    min_cost = 0  # هزینه کل درخت پوشای کمینه

    while candidates:
        # پیدا کردن یال با کمترین وزن از صف اولویت
        min_index = 0
```

```
        for i in range(1, len(candidates)):
            if candidates[i][0] < candidates[min_index][0]:
                min_index = i
        v = candidates[min_index][1]
        candidates.pop(min_index)

        if in_set[v]:
            continue

        min_cost += best_edge[v]
        in_set[v] = True

        # بررسی یال‌های متصل به رأس v
        for u, w in adj[v]:
            if not in_set[u] and w < best_edge[u]:
                best_edge[u] = w
                parent[u] = v
                candidates.append((best_edge[u], u))

    return min_cost
```

In [5]:
```
# تعریف گراف به صورت لیست مجاورت
g = [
    [(1, 3), (2, 4)],
    [(0, 3), (2, 2), (4, 1)],
    [(0, 4), (1, 2), (3, 9)],
    [(2, 9), (4, 5)],
    [(1, 1), (3, 5)]
]

p = [0] * N  # آرایه پدرها برای بازسازی درخت

# اجرای الگوریتم و چاپ نتیجه
print(prim(p, g))
for i in range(N):
    print(p[i], end=' ')
print()
```

```
11
-1 0 1 4 1
```

تحلیل این پیاده‌سازی 📊

در این حالت در هرم کمینه **candidates** در هر لحظه برای راس‌های بیرون از **S** کم‌وزن‌ترین یال متصل به آن‌ها را داریم و هر بار سر این هرم را می‌گیریم و آن را به **S** اضافه می‌کنیم و سپس کم‌وزن‌ترین یال متصل به همسایه‌های خارج از **S** آن را در صورت نیاز تغییر می‌دهیم و آن‌ها را دوباره به **candidates** اضافه می‌کنیم.

در این پیاده‌سازی برای هر راس، محتویات حلقه یک بار اجرا می‌شود و حداکثر هر همسایه‌اش در آن یک بار به هرم اضافه می‌شود که هر اضافه کردن در **O(log n)** انجام می‌شود و در مجموع الگوریتم در زمان **O(m log n)** اجرا می‌شود.

🔹 الگوریتم Kruskal

ایده حل

راه حل حریصانه‌ای که به ذهن می‌رسد این است که این درخت از سبک‌ترین یال‌های گراف تشکیل می‌شود، پس یال‌های سبک گراف را نگه می‌داریم و یال‌های سنگین‌تر را در صورتی که گراف بدون آن‌ها هم همبند می‌ماند حذف می‌کنیم. در الگوریتم کروسکال، اول همه‌ی یال‌های گراف را برمی‌داریم و بعد یکی یکی سبک‌ترین یال‌ها را اضافه می‌کنیم و این کار را تا جایی ادامه می‌دهیم که گراف همبند شود.

توضیح الگوریتم

الگوریتم کروسکال به صورت زیر عمل می‌کند:

1. همه‌ی یال‌ها را از گراف برمی‌داریم و در یک مجموعه قرار می‌دهیم. یال‌ها را در این مجموعه به ترتیب صعودی وزن مرتب می‌کنیم. پس الان یک گراف تهی داریم و یک مجموعه یال.
2. حالا سبک‌ترین یال را از مجموعه برمی‌داریم. چک می‌کنیم که اگر این یال را سرجایش بگذاریم در گرافمان دور ایجاد می‌شود یا نه. اگر دور ایجاد نمی‌شود، آن را به گراف اضافه می‌کنیم وگرنه آن را دور می‌اندازیم.
3. اینقدر مرحله‌ی 2 را تکرار می‌کنیم تا $V-1$ یال در گراف داشته باشیم (که V تعداد رئوس گراف است)

الگوریتم Kruskal

راه حل حریصانه‌ای که به ذهن می‌رسد این است که این درخت از سبک‌ترین یال‌های گراف تشکیل می‌شود، پس یال‌های سبک گراف را نگه می‌داریم و یال‌های سنگین‌تر را در صورتی که گراف بدون آن‌ها هم همبند می‌ماند حذف می‌کنیم. در الگوریتم کروسکال، ابتدا همه‌ی یال‌های گراف را برمی‌داریم و بعد یکی یکی سبک‌ترین یال‌ها را اضافه می‌کنیم و این کار را تا جایی ادامه می‌دهیم که گراف همبند شود.

توضیح الگوریتم 🛠️

الگوریتم کروسکال به صورت زیر عمل می‌کند:

1. همه‌ی یال‌ها را از گراف برمی‌داریم و در یک مجموعه قرار می‌دهیم. یال‌ها را در این مجموعه به ترتیب صعودی وزن مرتب می‌کنیم. پس الان یک گراف تهی داریم و یک مجموعه یال.
2. حالا سبک‌ترین یال را از مجموعه برمی‌داریم. چک می‌کنیم که اگر این یال را سرجایش بگذاریم در گرافمان دور ایجاد می‌شود یا نه. اگر دور ایجاد نمی‌شود، آن را به گراف اضافه می‌کنیم وگرنه آن را حذف می‌کنیم.
3. این مرحله را تکرار می‌کنیم تا **V – 1** یال در گراف داشته باشیم (که **V** تعداد رئوس گراف است).

پیاده‌سازی گراف 🗂️

برای پیاده‌سازی گراف، راه‌های مختلفی وجود دارد. یکی این است که ماتریس مجاورت گراف را ذخیره کنیم (یعنی برای هر راس، یال‌های آن راس با راس‌های دیگر را نگه داریم حتی اگر نداشته باشد). راه دیگر این است که گراف را با تعداد راس‌ها و تعداد یال‌ها مشخص کنیم و سپس هر یال را با دو راس آن و وزن آن نگه داریم. اینجا راه دوم را انتخاب می‌کنیم، زیرا یال‌ها را یک بار در اول الگوریتم به ترتیب وزن مرتب می‌کنیم و بعد یکی یکی از مجموعه‌ی مرتب شده برمی‌داریم. بنابراین نیازی به ماتریس مجاورت نیست و به اندازه‌ی **O(n²)** حافظه مصرف نمی‌شود.

In [7]:
```python
# تعریف کلاس یال
class Edge:
    def __init__(self, first, second, weight):
        self.first = first      # رأس اول یال
        self.second = second    # رأس دوم یال
        self.weight = weight    # وزن یال

# تعریف کلاس گراف
class Graph:
    def __init__(self, vertice_num, edge_num):
        self.vertice_num = vertice_num  # تعداد رأس‌ها
        self.edge_num = edge_num        # تعداد یال‌ها
        self.edge = [None] * edge_num   # لیست یال‌ها
```

In [8]:
```python
# تابع ساخت گراف با تعداد رأس و یال مشخص
def make_graph(vertice_num, edge_num):
    graph = Graph(vertice_num, edge_num)  # ایجاد شیء گراف با تعداد رأس و یال
    graph.edge = [Edge(0, 0, 0) for _ in range(edge_num)]  # مقداردهی اولیه لیست یال‌ها
    return graph  # بازگرداندن گراف ساخته‌شده
```

چک کردن دور در مرحله دوم 🔍

یک راه این است که مولفه‌های همبندی درختی که داریم می‌سازیم را با مجموعه‌ی راس‌های موجود در آن‌ها نگه داریم. ابتدا فقط راس‌های گراف را قرار می‌دهیم و بین هیچ دو راسی هیچ مسیری وجود ندارد، بنابراین هر راس یک مولفه‌ی همبندی است. حال هر وقت یک یال را از مجموعه‌ی مرتب‌شده‌ی یال‌ها برمی‌داریم تا آن را به درخت اضافه کنیم، چک می‌کنیم که راس‌های دو سر این یال، در یک مولفه‌ی همبندی هستند یا نه (فرض کنیم **a** و **b** راس‌های دو سر این یال باشند).

اگر بودند، یعنی طبق تعریف مولفه‌ی همبندی، بین **a** و **b** مسیر وجود دارد. پس اگر آن یال را اضافه کنیم بینشان دو مسیر وجود خواهد داشت، یعنی دور ایجاد می‌شود و بنابراین یال را اضافه نمی‌کنیم.
اگر **a** و **b** در یک مولفه‌ی همبندی نباشند، یعنی بینشان مسیر وجود ندارد. در این صورت اضافه کردن یال فعلی دور ایجاد نمی‌کند، پس یال را اضافه می‌کنیم. پس از اضافه کردن یال، بین هر راس موجود در مولفه‌ی همبندی **a** و هر راس موجود در مولفه‌ی همبندی **b** یک مسیر وجود دارد و دو مولفه‌ی همبندی اکنون یکی شده‌اند. بنابراین این دو مولفه را از لیست حذف کرده و اجتماعشان را جایگزین می‌کنیم.

برای پیاده‌سازی، هر کدام از مولفه‌های همبندی را یک **مجموعه‌ی مجزا (disjoint set)** می‌گیریم. برای اینکه بررسی کنیم دو راس در یک مجموعه‌ی مجزا هستند یا خیر، از عملیات **find** استفاده می‌کنیم. این عملیات برای هر عضو سرگروه مجموعه‌ای که آن عضو را شامل می‌شود برمی‌گرداند. بنابراین اگر **a** و **b** سرگروه یکسان داشتند، یعنی در یک مجموعه‌ی مجزا هستند؛ در غیر این صورت باید دو مجموعه را با عملیات **union** ترکیب کنیم.

راه دیگر برای پیدا کردن دور در گراف، استفاده از روش‌های پیمایش درخت مثل **DFS** است.

In [9]:
```python
# ساختار مجموعه‌های مجزا برای تشخیص دور
class DisjointSet:
    def __init__(self, parent):
        self.parent = parent  # پدر مجموعه
        self.count = 0        # عمق درخت برای بهینه‌سازی union
```

In [10]:
```python
# پیدا کردن نماینده (سرگروه) مجموعه‌ای که عضو i در آن است
def find(disjoint_sets, i):
    if disjoint_sets[i].parent != i:
        disjoint_sets[i].parent = find(disjoint_sets, disjoint_sets[i].parent)
    return disjoint_sets[i].parent
```

In [11]:
```python
# ترکیب دو مجموعه مجزا
def union(disjoint_sets, x, y):
    xroot = find(disjoint_sets, x)
    yroot = find(disjoint_sets, y)

    if disjoint_sets[xroot].count < disjoint_sets[yroot].count:
        disjoint_sets[xroot].parent = yroot
    elif disjoint_sets[xroot].count > disjoint_sets[yroot].count:
        disjoint_sets[yroot].parent = xroot
    else:
        disjoint_sets[yroot].parent = xroot
        disjoint_sets[xroot].count += 1
```

In [12]:
```python
# الگوریتم Kruskal برای یافتن درخت پوشای کمینه
def kruskal(graph):
    vertice_num = graph.vertice_num
    result = []  # لیست یال‌های انتخاب‌شده برای MST

    # مرتب‌سازی یال‌ها بر اساس وزن
    graph.edge.sort(key=lambda e: e.weight)

    # مقداردهی اولیه مجموعه‌های مجزا
    disjoint_sets = [DisjointSet(i) for i in range(vertice_num)]

    e = 0  # شمارنده یال‌های انتخاب‌شده
    i = 0  # اندیس یال فعلی

    while e < vertice_num - 1 and i < graph.edge_num:
        next_edge = graph.edge[i]
        i += 1

        x = find(disjoint_sets, next_edge.first)
        y = find(disjoint_sets, next_edge.second)

        # اگر اضافه کردن یال دور ایجاد نکند، آن را اضافه می‌کنیم
        if x != y:
            result.append(next_edge)
            e += 1
            union(disjoint_sets, x, y)

    # چاپ یال‌های درخت پوشای کمینه
```

```
    for edge in result:
        print(f"{edge.first} to {edge.second} with weight {edge.weight}")
```

In [13]:
```
# تعریف گراف نمونه و اجرای الگوریتم Kruskal
V = 4
E = 5
graph = make_graph(V, E)

# اضافه کردن یال‌ها به گراف
graph.edge[0].first = 0
graph.edge[0].second = 1
graph.edge[0].weight = 10

graph.edge[1].first = 0
graph.edge[1].second = 2
graph.edge[1].weight = 6

graph.edge[2].first = 0
graph.edge[2].second = 3
graph.edge[2].weight = 5

graph.edge[3].first = 1
graph.edge[3].second = 3
graph.edge[3].weight = 15

graph.edge[4].first = 2
graph.edge[4].second = 3
graph.edge[4].weight = 4

kruskal(graph)
```

```
2 to 3 with weight 4
0 to 3 with weight 5
0 to 1 with weight 10
```

پیچیدگی زمانی ⏱️

ابتدا باید یال‌ها را مرتب کنیم، که چون پیچیدگی مرتب کردن n شیء $O(n \log n)$ است، این بخش پیچیدگی $O(E \log E)$ خواهد داشت که E تعداد یال‌هاست.

سپس به ازای هر یال در مجموعه، دو بار عملیات **find** و صفر یا یک بار عملیات **union** روی مولفه‌های همبندی راس‌های دو سر یال انجام می‌دهیم. بنابراین حداکثر E تا عملیات union-find داریم که هرکدام حداکثر پیچیدگی $O(\log V)$ دارند که V تعداد رئوس است.

در مجموع پیچیدگی این بخش $O(E \log V)$ است و جمع کل پیچیدگی الگوریتم می‌شود $O(E \log E + E \log V)$. از آنجایی که E حداکثر $O(V^2)$ است، $\log E$ و $\log V$ از یک مرتبه‌ی پیچیدگی هستند.

اثبات درستی الگوریتم Kruskal ☑️

درخت پوشا بودن:

فرض کنیم گراف اصلیمان G باشد که همبند و وزن‌دار است. اگر Y زیرگرافی باشد که الگوریتم کروسکال تولید می‌کند، این زیرگراف نمی‌تواند دور داشته باشد، زیرا همیشه قبل از اضافه کردن یال چک می‌کنیم که با اضافه شدنش دور ایجاد نشود. Y ناهمبند هم نمی‌تواند باشد، زیرا اگر دو مولفه‌ی همبندی جدا داشت، وقتی الگوریتم به سبک‌ترین یالی که دو مؤلفه را وصل می‌کند برسد، آن یال را به گراف اضافه می‌کند. بنابراین دو مولفه یکی شده و چون همه‌ی رئوس در یک مولفه هستند و دوری هم ندارد، نتیجه یک درخت پوشا خواهد بود.

کمینه بودن:

اثبات با استقرا انجام می‌شود. نشان می‌دهیم که اگر F مجموعه‌ای از یال‌های انتخاب‌شده در هر مرحله از الگوریتم باشد، آنگاه یک درخت پوشای کمینه وجود دارد که F زیرگراف آن است (گزاره **P**).

در ابتدا وقتی F تهی است، گزاره P درست است؛ زیرا همه درخت‌های پوشای گراف، این گراف تهی را شامل می‌شوند و حداقل یک درخت پوشای کمینه وجود دارد.

فرض کنیم الگوریتم تا این مرحله F را ساخته و F زیرگراف یک درخت پوشای کمینه مانند T است. فرض کنیم یال بعدی e باشد.

- اگر e در T باشد، گزاره P برای F + e هم درست است.

- اگر e در T نباشد، T + e یک دور دارد. این دور شامل یالی e' است که در F + e نیست. e' در T است و وزن آن ≤ وزن e است.

بنابراین T - e' + e یک درخت پوشای کمینه است و شامل F + e می‌شود. گزاره P برقرار است.

با استقرا، وقتی الگوریتم به مرحله‌ای برسد که F یک درخت پوشا شود، P درست است؛ بنابراین F یک درخت پوشای کمینه خواهد بود.

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل چهارم: گراف

بخش دوم: کوتاه‌ترین مسیر بین هر دو رأس (All Pairs Shortest Paths)

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 🗂️

مقدمه 📘

همان‌طور که می‌دانید، پیدا کردن کوتاه‌ترین مسیر بین رأس‌ها در گراف‌ها، کاربرد بسیاری دارد. از جمله در پیدا کردن بهترین مسیر برای رانندگی، طراحی شبکه جاده‌ای، مسیریابی شبکه (Network Routing) و ... . در درس ساختمان داده، با الگوریتم‌های پیدا کردن کوتاه‌ترین مسیر از یک مبدأ (Single-Source Shortest Paths) از جمله بلمن-فورد و دایکسترا آشنا شدیم. این الگوریتم‌ها، به ازای هر رأس از گراف، کوتاه‌ترین مسیر برای رسیدن از رأسی مشخص به آن رأس را پیدا می‌کنند. حال اگر بخواهیم بین هر دو رأس در گراف، کوتاه‌ترین مسیر را پیدا کنیم، چه باید کرد؟

سوال؛ کوتاه‌ترین مسیر بین هر دو راس 📌

در یک گراف جهت‌دار و وزن‌دار بدون دور منفی، به ازای هر دو رأس $u,v \in V$، کوتاه‌ترین مسیر (مسیر با کمترین وزن) بین u و v را بیابید.

راه حل اولیه 💡

اولین راهی که به ذهن می‌رسد، این است که به ازای هر رأس، یک بار الگوریتم دایکسترا (Dijkstra) را به مبدأ آن رأس اجرا کنیم تا برای هر رأس، کوتاه‌ترین مسیرها به مبدأ این راس را پیدا کنیم، و به این ترتیب، کوتاه‌ترین مسیر بین هر دو رأس را پیدا کنیم. این راه‌حل به خوبی جواب می‌دهد و مرتبه زمانی آن برابر با $|V|$ بار اجرای دایکسترا است، یعنی $O(V(V \log V + E)) = O(V^2 \log V + VE)$ است.

اما یک نکته‌ی مهم را باید در نظر گرفت؛ اگر گراف یال منفی داشته باشد، الگوریتم دایکسترا جواب درست نمی‌دهد! در این حالت چه کنیم؟

راه حل با بلمن-فورد

شاید بگویید می‌توانیم به‌جای دایکسترا، از الگوریتم بلمن-فورد (Bellman-Ford) استفاده کنیم، که در گراف‌های با یال منفی نیز پاسخ درست می‌دهد. (فقط کافی است گراف، دور منفی نداشته‌باشد.) بله درست است. این الگوریتم به درستی کار می‌کند.

تحلیل زمانی

مرتبه زمانی الگوریتم بلمن-فورد، از $O(VE)$ است. حال اگر به ازای هر رأس به عنوان مبدأ، این الگوریتم را اجرا کنیم، مرتبه زمانی کلی از $O(V(VE)) = O(V^2E)$ می‌شود. در بدترین حالت، تعداد یال‌های گراف از $O(V^2)$ است، پس مرتبه‌ی زمانی این روش در بدترین حالت از $O(V^4)$ است.

آیا می‌توان الگوریتمی با مرتبه زمانی بهتر ارائه داد؟

In [1]:
```python
# الگوریتم بلمن-فورد برای یافتن کوتاه‌ترین مسیر از رأس s به سایر رأس‌ها

MAX = 100  # حداکثر تعداد رأس‌ها
MAXINT = 10**9  # مقدار بسیار بزرگ برای مقداردهی اولیه فاصله‌ها

# ورودی‌ها:
# n: تعداد رأس‌ها
# m: تعداد یال‌ها
# u[i]: امنا مبدأ یال
# v[i]: امنا مقصد یال
# w[i]: امنا وزن یال
# s: رأس مبدأ

n = 5
m = 7
u = [0, 0, 1, 1, 1, 3, 4]
v = [1, 2, 2, 3, 4, 4, 3]
w = [6, 7, 8, 5, -4, 9, 7]
s = 0  # رأس مبدأ

d = [MAXINT] * n  # فاصله‌ها از مبدأ
parent = [-1] * n  # پدر هر رأس در مسیر کوتاه

d[s] = 0
parent[s] = s

for k in range(n):
    relaxed = False
    for i in range(m):
        if d[u[i]] + w[i] < d[v[i]]:
            d[v[i]] = d[u[i]] + w[i]
            parent[v[i]] = u[i]
            relaxed = True
    if not relaxed:
        break
    if k == n - 1:
        print("Graph has negative cycle!")

# چاپ فاصله‌ها و مسیرها
for i in range(n):
    print(f"Distance to vertex {i} = {d[i]}, parent = {parent[i]}")
```

```
Distance to vertex 0 = 0, parent = 0
Distance to vertex 1 = 6, parent = 0
Distance to vertex 2 = 7, parent = 0
Distance to vertex 3 = 9, parent = 4
Distance to vertex 4 = 2, parent = 1
```

الگوریتم جانسون 🚀

اگر بتوانیم وزن یال‌ها را به گونه‌ای تغییر دهیم که وزن همه‌ی یال‌ها مثبت شود، و در عین حال کوتاه‌ترین مسیر بین هیچ دو رأسی عوض نشود، می‌توانیم از همان راه حل V بار اجرای دایکسترا استفاده کنیم. حال سوال این است که چگونه این کار را انجام دهیم؟

ممکن است بگویید که همه‌ی یال‌ها را با مقداری ثابت (مثلاً اندازه‌ی منفی‌ترین یال گراف) جمع می‌کنیم تا وزن همه‌ی یال‌ها مثبت شود. با کمی بررسی می‌توان متوجه شد این حرکت، ممکن است کوتاه‌ترین مسیر بین بعضی رئوس را عوض کند. پس چه باید کرد؟ 🤔

راهی که الگوریتم جانسون پیشنهاد می‌دهد، این است:
به ازای هر رأس مانند $v \in V$، یک عدد مانند $h(v)$ نسبت می‌دهیم. حال به ازای هر یال $(u,v) \in E$ با وزن $w(u,v)$، وزن جدید $\hat{w}(u,v)$ را مطابق زیر تعریف می‌کنیم:
$$\hat{w}(u,v) = w(u,v) + h(u) - h(v)$$

ابتدا باید ثابت کنیم که این تغییر در گراف، کوتاه‌ترین مسیرها را تغییر نمی‌دهد. یک مسیر دلخواه مانند $p = \langle v_0, v_1, ..., v_k \rangle$ در نظر بگیرید. وزن این مسیر در گراف با وزن‌های جدید برابر است با:
$$\hat{w}(p) = \hat{w}(v_0,v_1) + \hat{w}(v_1,v_2) + ... + \hat{w}(v_{k-1},v_k) = \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i) = \sum_{i=1}^{k} (w(v_{i-1},v_i) + h(v_{i-1}) - h(v_i)) = \sum_{i=1}^{k} w(v_{i-1},v_i) + h(v_0) - h(v_k) = w(p) + h(v_0) - h(v_k)$$

بنابراین مشاهده می‌شود که مسیرهایی با ابتدا و انتهای مشخص، وزن‌شان به مقدار ثابتی ($h(v_0)-h(v_k)$) تغییر می‌کند. یعنی اگر مسیری در گراف اصلی، کوتاه‌ترین مسیر بین دو رأس v و u باشد، در گراف با وزن‌های جدید نیز کوتاه‌ترین مسیر بین این دو رأس است و بالعکس.
همچنین بر اثر این تغییر وزن یال‌ها، وزن دورهای گراف تغییری نمی‌کند. پس گراف با وزن یال‌های جدید، دور منفی ندارد اگر و تنها اگر گراف اصلی دور منفی نداشته باشد.

سپس، باید یک روش مناسب برای محاسبه تابع h برای رأس‌ها پیشنهاد دهیم، به طوری که وزن جدید تمام یال‌ها مثبت شود.
روش الگوریتم جانسون این است: گراف G' را از روی گراف اصلی G می‌سازیم: یک راس جدید مانند $s \notin V$ به گراف اضافه می‌کنیم، سپس از s به رأس‌های دیگر، یالی با وزن صفر قرار می‌دهیم. مشخص است که این گراف G' دور منفی ندارد اگر و تنها اگر G دور منفی نداشته باشد.
حال روی گراف G' الگوریتم بلمن-فورد را با مبدأ s اجرا می‌کنیم و به ازای هر رأس $h(v)$، $v \neq s$ را برابر با $d(v) = \delta(s,v)$ قرار می‌دهیم.
ادعا می‌کنیم که گراف G بر اثر تغییر وزن یال‌ها با استفاده از تابع h مذکور، دیگر یالی با وزن منفی نخواهد داشت.
در الگوریتم بلمن-فورد، پس از اتمام اجرا ($|V|-1$ بار Relax کردن تمام یال‌ها)، به ازای هر یال مانند $(u,v) \in E$ داریم:
$$d_u + w(u,v) \geq d_v \Rightarrow w(u,v) + d_u - d_v \geq 0 \Rightarrow \hat{w}(u,v) \geq 0$$
مشخص است که وزن جدید تمام یال‌ها بر اثر این تغییر، مثبت می‌شود. بنابراین این روش برای محاسبه تابع h روش مناسبی است.

In [2]:
```python
# الگوریتم جانسون برای یافتن کوتاه‌ترین مسیر بین همه جفت رأس‌ها در گراف وزن‌دار

MAXINT = 10**9

# ساختار یال
class Edge:
    def __init__(self, u, v, w):
        self.u = u  # مبدأ یال
        self.v = v  # مقصد یال
        self.w = w  # وزن یال

# الگوریتم Bellman-Ford برای محاسبه h(v) برای همه رأس‌ها
def bellman_ford(n, edges, s):
    d = [MAXINT] * n
    d[s] = 0
    for _ in range(n - 1):
        for edge in edges:
            if d[edge.u] + edge.w < d[edge.v]:
                d[edge.v] = d[edge.u] + edge.w
    # بررسی وجود دور منفی
    for edge in edges:
        if d[edge.u] + edge.w < d[edge.v]:
            print("Graph has negative cycle!")
            return None
```

```
    return d  # همان h(v)ها

# برای یافتن مسیرهای کوتاه از یک رأس Dijkstra الگوریتم
def dijkstra(n, adj, src):
    visited = [False] * n
    dist = [MAXINT] * n
    dist[src] = 0

    for _ in range(n):
        u = -1
        for i in range(n):
            if not visited[i] and (u == -1 or dist[i] < dist[u]):
                u = i
        if dist[u] == MAXINT:
            break
        visited[u] = True
        for v, w in adj[u]:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
    return dist

# الگوریتم جانسون
def johnson(n, edges):
    # به همه رأس‌ها s و یال‌های صفر وزن از s اضافه کردن رأس جدید
    new_edges = edges[:]
    s = n
    for v in range(n):
        new_edges.append(Edge(s, v, 0))

    # اجرای Bellman-Ford برای محاسبه h(v)
    h = bellman_ford(n + 1, new_edges, s)
    if h is None:
        return

    # ساخت گراف با وزن‌های جدید ŵ(u,v) = w(u,v) + h(u) - h(v)
    adj = [[] for _ in range(n)]
    for edge in edges:
        new_weight = edge.w + h[edge.u] - h[edge.v]
        adj[edge.u].append((edge.v, new_weight))

    # اجرای دایکسترا برای هر رأس و چاپ فاصله‌ها
    for u in range(n):
        dist = dijkstra(n, adj, u)
        print(f"Distances from vertex {u}:")
        for v in range(n):
            # تبدیل فاصله‌ها به وزن اصلی با فرمول معکوس
            if dist[v] < MAXINT:
                real_dist = dist[v] - h[u] + h[v]
                print(f"  to {v}: {real_dist}")
            else:
                print(f"  to {v}: unreachable")

# تعریف گراف نمونه
n = 5
edges = [
    Edge(0, 1, 3),
    Edge(0, 2, 8),
    Edge(0, 4, -4),
    Edge(1, 3, 1),
    Edge(1, 4, 7),
    Edge(2, 1, 4),
    Edge(3, 0, 2),
    Edge(3, 2, -5),
    Edge(4, 3, 6)
]

johnson(n, edges)
```

```
Distances from vertex 0:
  to 0: 0
  to 1: 1
  to 2: -3
  to 3: 2
  to 4: -4
Distances from vertex 1:
  to 0: 3
  to 1: 0
  to 2: -4
  to 3: 1
  to 4: -1
Distances from vertex 2:
  to 0: 7
  to 1: 4
  to 2: 0
  to 3: 5
  to 4: 3
Distances from vertex 3:
  to 0: 2
  to 1: -1
  to 2: -5
  to 3: 0
  to 4: -2
Distances from vertex 4:
  to 0: 8
  to 1: 5
  to 2: 1
  to 3: 6
  to 4: 0
```

تحلیل زمانی الگوریتم جانسون ⏱️

در ابتدا، برای محاسبه‌ی مقادیر h(v) به ازای رئوس، یک بار الگوریتم بلمن-فورد را روی گراف G' با |V|+1 راس و |E|+|V| یال اجرا می‌کنیم.
مرتبه‌ی زمانی این مرحله برابر با O((|V|+1)(|V|+|E|)) = O(VE) است.
(اگر |V| = O(|E|) باشد. در غیر این صورت، یعنی گراف ناهمبند است، الگوریتم بلمن-فورد را برای هر مؤلفه همبندی به صورت جدا اجرا می‌کنیم.)

سپس |V| بار الگوریتم دایکسترا را اجرا می‌کنیم، که اگر با استفاده از هرم فیبوناچی پیاده‌سازی شود، زمان اجرای آن از مرتبه‌ی O(V(V log V + E)) = O(V² log V + VE) است.
پس زمان اجرای کلّی الگوریتم از مرتبه‌ی O(VE + V² log V + VE) = O(V² log V + VE) است.

این مرتبه زمانی در بدترین حالت (زمانی که |E| = |V|² باشد) برابر با O(V³) است، که برابر با مرتبه زمانی الگوریتم فلوید-وارشال است.
اما در گراف‌های خلوت (Sparse)، زمان اجرای آن به صورت حدی از الگوریتم فلوید-وارشال کم‌تر است.

می‌توانید کد این الگوریتم را در زیر ببینید.

روشی دیگر برای یافتن کوتاه‌ترین مسیرها 🔄

خب بیایید با ابزار دیگری به سراغ حل مسئله بیان شده برویم. آیا اگر کمترین فاصله دو راس دلخواه را داشته باشیم، برای پیدا کردن کمترین فاصله دو راس دیگر از این اطلاعات قبلی می‌توانیم استفاده کنیم؟
به عبارتی دیگر، مسئله را می‌توانیم به یک سری مسائل کوچکتر تقسیم کنیم که با حل آن‌ها، مسئله اصلی نیز حل شود؟

همان‌گونه که احتمالا حدس زده‌اید، برای حل این مسئله از ایده‌ی **برنامه‌ریزی پویا** استفاده می‌کنیم و می‌خواهیم فاصله میان رئوس را به یک سری مسائل کوچک‌تر تبدیل کنیم که جواب آن‌ها قبلاً محاسبه شده باشد.

فرض کنید کم‌ترین فاصله میان دو راس i و j را با d[i,j] نشان بدهیم.
حال می‌دانیم که نامساوی زیر برای هر سه راس دلخواه i,j,k برقرار است:

$$d[i,j] \leq d[i,k] + d[k,j]$$

بنابراین برای محاسبه d[i,j] می‌توان تمام رئوس k ممکن را در نظر گرفت و مقدار سمت راست نامساوی بالا را برای آن‌ها محاسبه کرد و خواهیم داشت:

$$d[i,j] = \min_k \left\{ d[i,k] + d[k,j] \right\}$$

اگر در رابطه به دست آمده دقت کنید می‌بینید که این رابطه یک مشکل اساسی دارد. این رابطه وابستگی دوری دارد و به درستی عمل نخواهد کرد.
به عبارت دیگر، با استفاده از آن هیچ‌گاه نمی‌توان کمترین فاصله میان رئوس را پیدا کرد. اما با ایجاد ترتیب مناسب در حل زیرمسئله‌ها و اندکی تغییر در رابطه بالا، این مشکل قابل حل است.

الگوریتم فلوید وارشال 📊

در الگوریتم فلوید وارشال به جای d[i,j] که کوتاه‌ترین مسیر میان i و j می‌باشد، d[i,j,k] را داریم که کوتاه‌ترین مسیر میان رئوس i و j با شرط عبور از رئوس 1 تا k است.
به بیان دیگر، رئوس میانی مسیر صرفاً رئوس 1 تا k هستند (فرض کنید رئوس گراف از 1 تا n شماره‌گذاری شده‌اند).

با این فرض در هر مرحله می‌توان نوشت:

$$d[i,j,k] = \min(d[i,j,k-1], d[i,k,k-1] + d[k,j,k-1]), \quad k \geq 1$$

یعنی در کوتاه‌ترین مسیر میان i و j که رئوس میانی آن‌ها از 1 تا k باشند، یا از راس kام استفاده شده است یا نه. اگر استفاده نشده باشد که پاسخ همان d[i,j,k-1] می‌باشد، وگرنه کمترین مسیر از i به k و از k به j که صرفاً از رئوس 1 تا k-1 استفاده شده است را با هم جمع می‌کنیم و مینیمم این دو حالت برابر با d[i,j,k] خواهد بود.

پایه الگوریتم برای k = 0 است، یعنی هیچ راس میانی‌ای نداریم. مقدار d[i,j,0] اگر دو راس i و j مجاور باشند، برابر وزن یال بین i و j است (که با w[i,j] نشان می‌دهیم) و در غیر این صورت بی‌نهایت خواهد بود:

$$d[i,j,0] = \begin{cases} w[i,j] & \text{اگر یالی بین i و j وجود دارد} \\ \infty & \text{در غیر این صورت} \end{cases}$$

حال پاسخ سؤال اولیه ما، یعنی طول کوتاه‌ترین مسیر میان دو راس دلخواه، چیست؟
فاصله دو راس دلخواه i و j برابر با d[i,j,n] خواهد بود (یعنی تمامی رئوس 1 تا n می‌توانند در طول مسیر حضور داشته باشند).

تحلیل زمانی الگوریتم فلوید وارشال ⏱️

برای تحلیل زمانی الگوریتم تعداد زیرمسئله‌هایی که داریم و مقدار محاسباتی که باید برای حل هر کدام انجام بدهیم را بررسی می‌کنیم.
در این الگوریتم ما مقدار d[i,j,k] را به ازای 1 ≤ i,j,k ≤ n باید محاسبه کنیم. بنابراین تعداد مسائلی که باید حل کنیم از O(n^3) خواهد بود و برای حل هر کدام از این مسائل به اندازه O(1) هزینه صرف می‌کنیم.
در نتیجه پیچیدگی زمانی الگوریتم فلوید وارشال از O(n^3) خواهد بود.

سوال: اگر با الگوریتم فلوید وارشال بخواهیم کمترین فاصله میان تمام رئوس را به دست آوریم، حافظه مورد نیاز در چه پیچیدگی‌ای خواهد بود؟
پاسخ: با O(n^2) حافظه می‌توان این الگوریتم را پیاده‌سازی کرد (چرا؟)

مسئله:
یک گراف دلخواه با n راس و m یال داریم. در هر مرحله با دو نوع درخواست مواجه می‌شویم:

- درخواست نوع اول: یک راس از گراف حذف شود.
- درخواست نوع دوم: فاصله میان دو راس دلخواه در گراف باقیمانده تا آن لحظه چه قدر است.

اگر تعداد درخواست‌ها q باشد، الگوریتمی از پیچیدگی O(n^3) برای حل مسئله ارائه دهید (q << n^3).

راهنمایی: حذف شدن یک راس را می‌توان اضافه شدن راس در نظر گرفت اگر درخواست‌ها به صورت offline و از آخر به اول بررسی شوند.

In [3]:
```python
# الگوریتم فلوید-وارشال برای یافتن کوتاه‌ترین مسیر بین تمام جفت رأس‌ها

MAX = 1000
MAXINT = 10**9

def all_pairs_shortest_paths(G, n):
    # کپی اولیه از ماتریس وزن‌ها برای شروع الگوریتم
    d = [[G[i][j] for j in range(n)] for i in range(n)]

    # سه حلقه تو در تو برای بررسی تمام مسیرهای ممکن با رأس‌های میانی
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if d[i][k] + d[k][j] < d[i][j]:
                    d[i][j] = d[i][k] + d[k][j]

    # چاپ ماتریس نهایی فاصله‌ها
    for i in range(n):
        for j in range(n):
            print(d[i][j], end="\t")
        print()

# خواندن ورودی و ساخت گراف
n, m = map(int, input().split())
G = [[MAXINT if i != j else 0 for j in range(n)] for i in range(n)]

for _ in range(m):
    u, v, w = map(int, input().split())
    G[u - 1][v - 1] = w  # شماره رأس‌ها از ۱ هستند، در پایتون از ۰ شروع می‌شود

# اجرای الگوریتم
all_pairs_shortest_paths(G, n)
```

```
0       5       8       9
1000000000      0       3       4
1000000000      1000000000      0       1
1000000000      1000000000      1000000000      0
```

(مقدار 1000000000 چاپ می‌شود INF در پایتون به جای)

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل چهارم: گراف

بخش سوم: هرم فیبوناچی

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 📑

مقدمه 📘

تعریف: **Fibonacci Heap** یا هرم فیبوناچی یک داده ساختار برای پیاده سازی صف اولویت است. پیچیدگی زمانی هرم فیبوناچی از سایر داده ساختارهای صف اولویت، به طور سرشکن کم‌تر است.

Operation	Linked list	Binary heap	Binomial heap	Fibon. heap
MAKE-HEAP	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
INSERT	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
MINIMUM	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
EXTRACT-MIN	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
MERGE	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DECREASE-KEY	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
DELETE	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

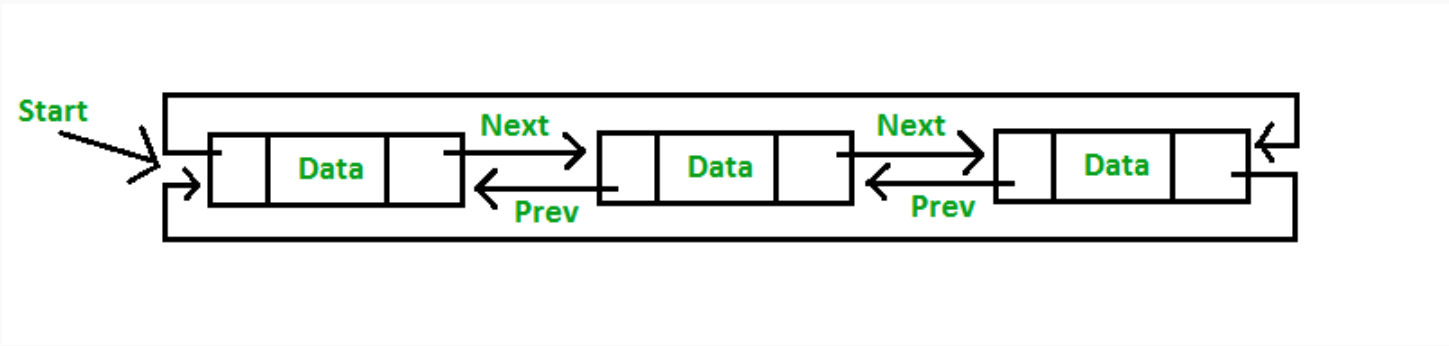
نکته:

> مقادیر این جدول برای هرم فیبوناچی به طور سرشکن تحلیل شده‌اند. 🧮

ساختار درونی هرم فیبوناچی به شکل اجتماع تعدادی درخت است 🌳 . تمامی این درخت‌ها هرم کمینه هستند؛ یعنی مقدار هر گره، از مقدار فرزندانش کم‌تر است. لازم به ذکر است که این درخت‌ها می‌توانند هر شکلی داشته باشند؛ حتی تمامی این درخت‌ها می‌توانند تنها متشکل از یک راس باشند.

ریشه‌های این درخت‌ها، به وسیله‌ی یک لیست پیوندی دوطرفه و دوری (doubly circular linked list) به هم متّصل شده‌اند.

هم‌چنین همواره اشاره‌گری به گره‌ی حاوی عنصر کمینه نگه داشته می‌شود. بدیهی است که چنین گره‌ای، ریشه‌ی یکی از درخت‌هاست.



استفاده از لیست پیوندی دوطرفه و مدور دو مزیت دارد؛ اوّل آن که اضافه کردن یک عضو در زمان O(1) انجام می‌شود و دوم آن که اگر دو لیست به ما داده شود می‌توان آنها را در زمان O(1) ترکیب کرد.

ایده اصلی در هرم فیبوناچی این است که تمامی عملیات را به شکل «lazy» انجام دهیم 💤. به طور مثال در عملیات ادغام دو هرم فیبوناچی(merge) تنها کاری که انجام می‌دهیم این است که لیست ریشه‌های دو هرم را به هم متصل کنیم. برای اضافه کردن یک راس به هرم فیبوناچی(insert) نیز به سادگی یک درخت تک راس را به مجموعه درخت‌ها اضافه می‌کنیم.

یادآوری: الگوریتم دایسترا (Dijkstra) 🚀

فرض کنید یک گراف وزن‌دار G داریم. وزن یال‌های این گراف مثبت است و هدف الگوریتم دایسترا یافتن کوتاه‌ترین مسیر از راسی مانند s به سایر رئوس است.

در هر مرحله:

1. راس v با کمترین فاصله از S را پیدا کنید. 🏃
2. فاصله‌های کاندید شده برای همه راس‌های متصل به V (همسایه‌ها) را بروزرسانی کنید. 🖍️

مرحله اول مشابه extract-min در صف اولویت است اما برای مرحله دوم مفهوم decrease-key اهمیت دارد.

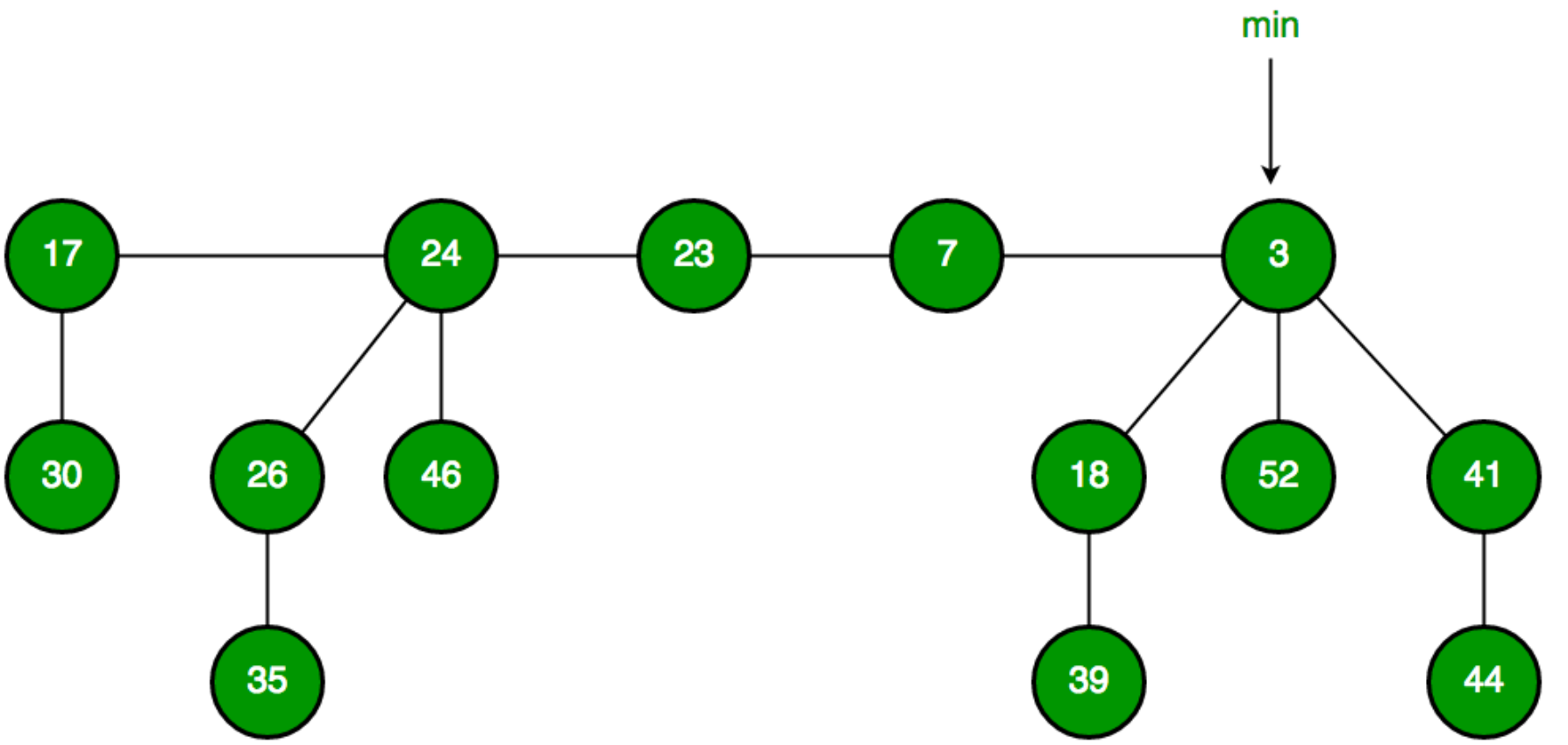
یادآوری: الگوریتم پریم (Prim) 🌲

این الگوریتم درخت پوشای کمینه در یک گراف بدون جهت را پیدا می‌کند.

در هر مرحله:

1. راس v را پیدا کنید که خارج از درخت فعلی باشد و کمترین هزینه اضافه شدن را داشته باشد. 📐
2. فاصله کاندید شده برای V به راس‌های خارج از ست فعلی را بروزرسانی کنید (مشابه دایسترا). 🖍️

مرحله اول همان extract-min و مرحله دوم decrease-key است.



پیاده‌سازی 💻

برای پیاده‌سازی این داده‌ساختار، از دو کلاس با interfaceهای زیر استفاده خواهیم کرد. پیاده‌سازی متدهای مربوط به هر بخش، در همان بخش آمده است. 🛠️

در ابتدا کلاس ساده‌ی **Node** را داریم که متدهای نسبتا ساده‌ای دارد و نیازی به توضیح اضافه ندارد. 📦

In [1]:
```python
# کلاس گره برای استفاده در هرم فیبوناچی
class Node:
    def __init__(self, value):
        self.value = value          # مقدار گره
        self.degree = 0             # تعداد فرزندان
        self.marked = False         # آیا گره علامت‌گذاری شده؟
        self.parent = None          # اشاره‌گر به پدر
        self.child = None           # اشاره‌گر به یکی از فرزندان

        # لیست پیوندی دوطرفه و مدور برای ریشه‌ها و فرزندان
        self.prev = self
        self.next = self

    def getPrev(self):
        return self.prev

    def getNext(self):
        return self.next

    def getChild(self):
        return self.child

    def getParent(self):
        return self.parent

    def getValue(self):
        return self.value

    def isMarked(self):
        return self.marked

    def hasChildren(self):
        return self.child is not None

    def hasParent(self):
        return self.parent is not None
```

کلاس FibonacciHeap 🏗️

در ادامه نیز کلاس **FibonacciHeap** آمده است. اکثر پیچیدگی‌های الگوریتم در متدهای این کلاس بیان شده‌اند. ⚡

In [2]:
```python
# کلاس هرم فیبوناچی برای پیاده‌سازی صف اولویت با عملیات سرشکن سریع
class FibonacciHeap:
    def __init__(self):
        self.heap = None  # اشاره‌گر به ریشه‌ای با کمترین مقدار

    # بررسی تهی بودن هرم
    def is_empty(self):
        return self.heap is None

    # ساخت یک گره تنها برای درج مقدار جدید
    def _singleton(self, value):
        return Node(value)

    # ادغام دو لیست پیوندی و بازگرداندن گره با مقدار کمینه
    def _merge(self, a, b):
        if a is None:
            return b
        if b is None:
            return a
        # اتصال دو لیست پیوندی مدور
        a_next = a.next
        b_prev = b.prev
        a.next = b
        b.prev = a
        a_next.prev = b_prev
        b_prev.next = a_next
        # بازگرداندن گره با مقدار کمینه
        return a if a.value < b.value else b

    # درج مقدار جدید در هرم
    def insert(self, value):
        node = self._singleton(value)
        self.heap = self._merge(self.heap, node)
        return node

    # گرفتن مقدار کمینه از هرم
    def get_minimum(self):
        if self.heap is None:
            return None
        return self.heap.value

    # ادغام دو هرم فیبوناچی
    def merge(self, other):
        self.heap = self._merge(self.heap, other.heap)
        other.heap = None

    # پیدا کردن گره با مقدار مشخص (جستجوی خطی در لیست پیوندی)
    def _find(self, heap, value):
        if heap is None:
            return None
        current = heap
        while True:
            if current.value == value:
                return current
            current = current.next
            if current == heap:
                break
        return None

    def find(self, value):
        return self._find(self.heap, value)

    # حذف همه گره‌ها (برای آزادسازی حافظه)
    def _delete_all(self, node):
        if node is None:
            return
        start = node
        while True:
            child = node.child
            self._delete_all(child)
            next_node = node.next
            node.prev = node.next = node.child = node.parent = None
            if next_node == start:
                break
            node = next_node

    def __del__(self):
        self._delete_all(self.heap)
        self.heap = None

    # تابع کمکی برای ساخت هرم تهی
    def _empty(self):
        return None
```

⚙️ Constructor و Destructor

پیش از شرح الگوریتم‌ها، کدهای دو تابع **Constructor** و **Destructor** این کلاس را آورده‌ایم. این دو متد مسئول مقداردهی اوّلیه به متغیّرهای داخلی این کلاس و پاکسازی این متغیّرهای **allocate** شده پس از اتمام کار هستند. 🧹

In [3]:
```python
# Constructor و Destructor برای کلاس FibonacciHeap
```

```python
class FibonacciHeap:
    def __init__(self):
        # مقداردهی اولیه: هرم خالی است
        self.heap = None

    def __del__(self):
        # پاکسازی حافظه در پایان کار
        if self.heap is not None:
            self._delete_all(self.heap)
            self.heap = None

    # حذف بازگشتی تمام گره‌ها از حافظه
    def _delete_all(self, node):
        if node is not None:
            current = node
            while True:
                next_node = current.next
                self._delete_all(current.child)  # حذف فرزندان گره فعلی
                current.prev = current.next = current.child = current.parent = None
                if next_node == node:
                    break
                current = next_node
```

همان طور که می‌بینید در پیاده‌سازی هر یک از این دو متد، از یک متد private استفاده شده است. کد این ۲ متد در ادامه آمده‌اند.

متد isEmpty ✅

هم‌چنین متد ساده‌ای به نام **isEmpty** داریم که همان‌طور که می‌توان حدس زد، کدش شامل همین یک خط است! 💡

In [4]:
```python
    # بررسی تهی بودن هرم فیبوناچی
    def is_empty(self):
        return self.heap is None
```

اجتماع و اضافه کردن 🌳➕

اضافه کردن (Insertion):

برای اضافه کردن یک راس مراحل زیر را به ترتیب انجام می‌دهیم:

1. راس V را می‌سازیم
2. چک می‌کنیم که آیا هرم ما خالی است یا خیر
3. اگر خالی بود V را تنها راس در مجموعه راس‌های root list قرار می‌دهیم و V را برابر مینیموم هرم می‌گذاریم (root list همان لینک لیست دوطرفه برای نگه داشتن ریشه درخت‌ها است که پیش‌تر معرفی شد)
4. در غیر این صورت V را به root list اضافه می‌کنیم و در صورت نیاز مینیموم را آپدیت می‌کنیم

مثالی از این مرحله:

اجتماع (Union یا Merge):

برای اجتماع دو فیبوناچی هیپ مراحل زیر را اجرا می‌کنیم:

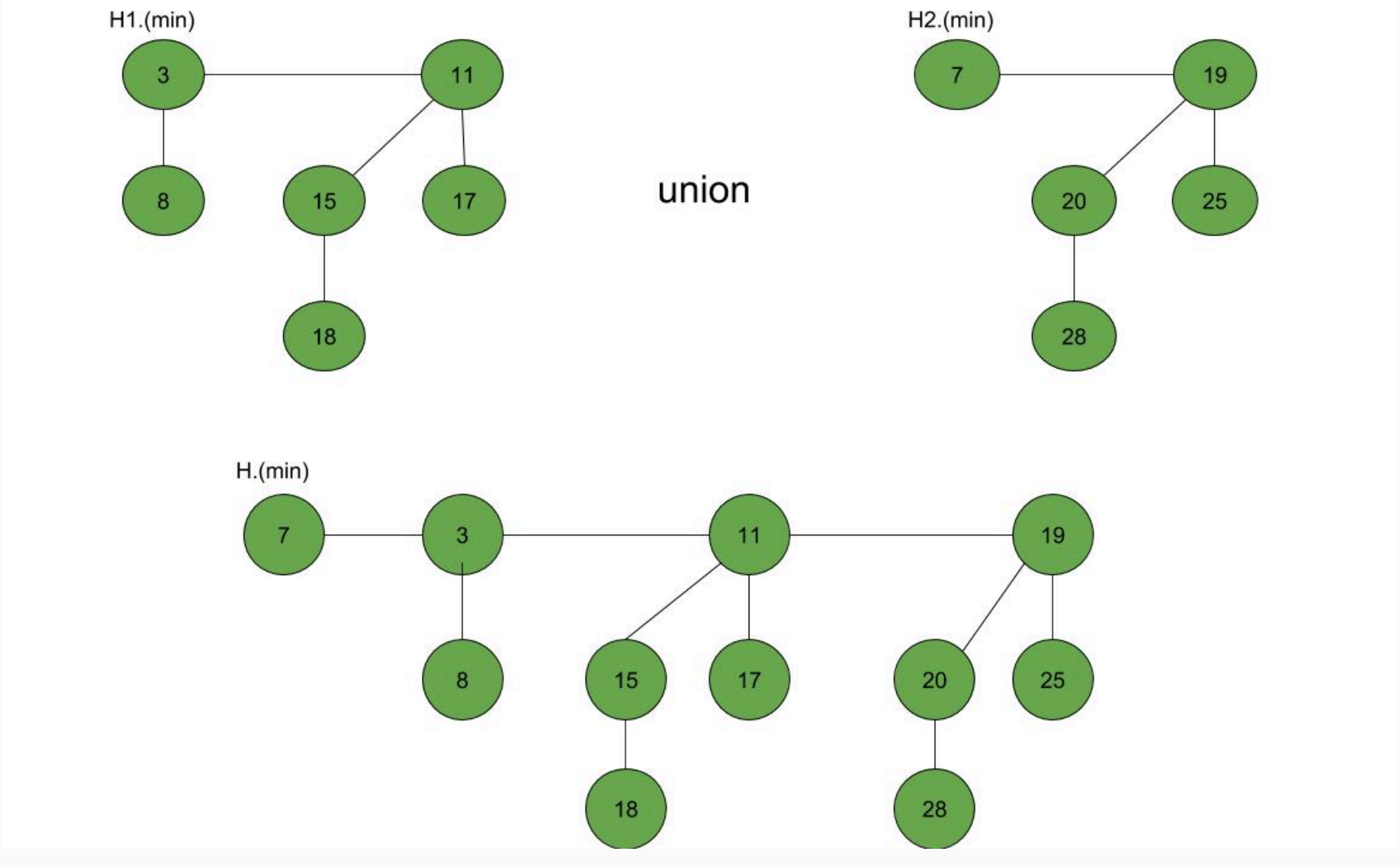
1. ریشه‌های دو هرم را در کنار هم در یک root list قرار می‌دهیم
2. if H1(min) < H2(min) then H(min) = H1(min)
3. else: H(min) = H2(min)

H(min) = مینیموم هرم حاصل

H1(min) = مینیموم هرم اول

H2(min) = مینیموم هرم دوم

📌 نکته: این مرحله فقط لیست ریشه‌ها را به هم وصل می‌کند و عملیاتی lazy محسوب می‌شود.



پیاده‌سازی 💻⚙️

اضافه کردن (Insertion) 🟢

برای این قسمت ابتدا دو متد به نام‌های *singleton_* و *merge_* ایجاد می‌کنیم. سپس با استفاده از این دو متد که **private** هستند، یک متد به نام *insert* پیاده‌سازی می‌کنیم که **public** خواهد بود.

متد singleton_ 🌱

متد **singleton_** یک مقدار دریافت می‌کند و یک هرم تک‌گره‌ای می‌سازد که تنها گره آن شامل مقدار ورودی این تابع است.

In [6]:
```python
# ساخت یک گره تنها با مقدار داده‌شده
def _singleton(self, value):
    n = Node(value)
    n.prev = n.next = n  # لیست پیوندی مدور با یک عضو
    n.degree = 0
    n.marked = False
    n.child = None
    n.parent = None
    return n
```

متد merge_ 🔗

متد **merge_** دو اشاره‌گر به دو گره دریافت می‌کند. با توجه به اینکه این دو گره جزئی از یک لیست پیوندی هستند و هر کدام شامل مقدار کمینه خود هستند، این دو لیست پیوندی را با یکدیگر ادغام می‌کند و در نهایت اشاره‌گری به گره‌ی کمینه‌ی لیست جدید بازمی‌گرداند. بدیهی است که این مقدار بازگشتی، یکی از دو ورودی تابع خواهد بود.

In [7]:
```python
# ادغام دو لیست ریشه و بازگرداندن گره با مقدار کمینه
def _merge(self, a, b):
    if a is None:
        return b
    if b is None:
        return a
    if a.value > b.value:
        a, b = b, a  # اطمینان از اینکه a کمینه است

    # اتصال لیست‌های پیوندی مدور
    a_next = a.next
    b_prev = b.prev
    a.next = b
    b.prev = a
    a_next.prev = b_prev
    b_prev.next = a_next

    return a  # بازگرداندن گره با مقدار کمینه
```

اضافه کردن مقدار جدید ➕

در نهایت، برای اضافه کردن یک مقدار جدید به کل داده ساختار، ابتدا با استفاده از متد **singleton_** یک هرم شامل عنصر جدید ایجاد می‌کنیم. سپس تنها گره هرم جدید را با استفاده از متد **merge_** به لیست ریشه‌های هرم‌های کنونی داده ساختارمان اضافه می‌کنیم.

In [21]:
```python
# اضافه کردن مقدار جدید به هرم
def insert(self, value):
    node = self._singleton(value)
    self.heap = self._merge(self.heap, node)
    return node
```

اجتماع هرم‌ها

برای این قسمت، به وسیله‌ی متد **merge_** که قبلا پیاده‌سازی کرده‌ایم، لیست پیوندی ریشه‌ی هرم‌های دو **FibonacciHeap** را با هم ادغام می‌کنیم و در نهایت متغیر **heap** را به ازای هرم دوم برابر با **NULL** قرار می‌دهیم.

این کار به این منظور است که در این ادغام از روی داده‌های درخت دوم کپی نمی‌گیریم؛ بلکه خود آن‌ها را مستقیما به هرم فیبوناچی کنونی منتقل می‌کنیم. بنابراین دسترسی به این مقادیر تنها به هرم فیبوناچی کنونی محدود می‌شود و هرم دوم نمی‌تواند تغییری روی آن‌ها اعمال کند.

In [9]:
```python
# ادغام دو هرم فیبوناچی
def merge(self, other):
    self.heap = self._merge(self.heap, other.heap)
    other.heap = self._empty()

# ساخت هرم تهی (برابر با None)
def _empty(self):
    return None
```

حذف، کاهش کلید و استخراج کمینه

بخش اول: استخراج کمینه *O(log n)*

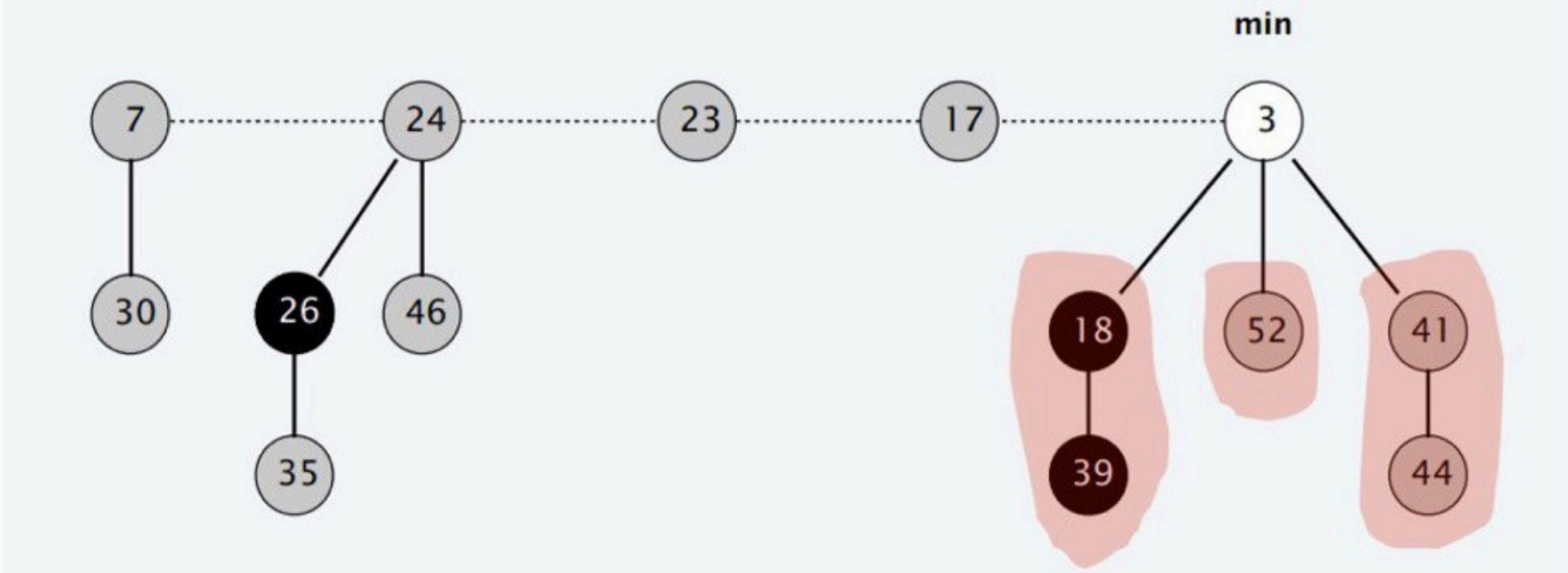
برای این کار، عملگر ترکیب دو درخت **Merge** معرفی می‌شود:

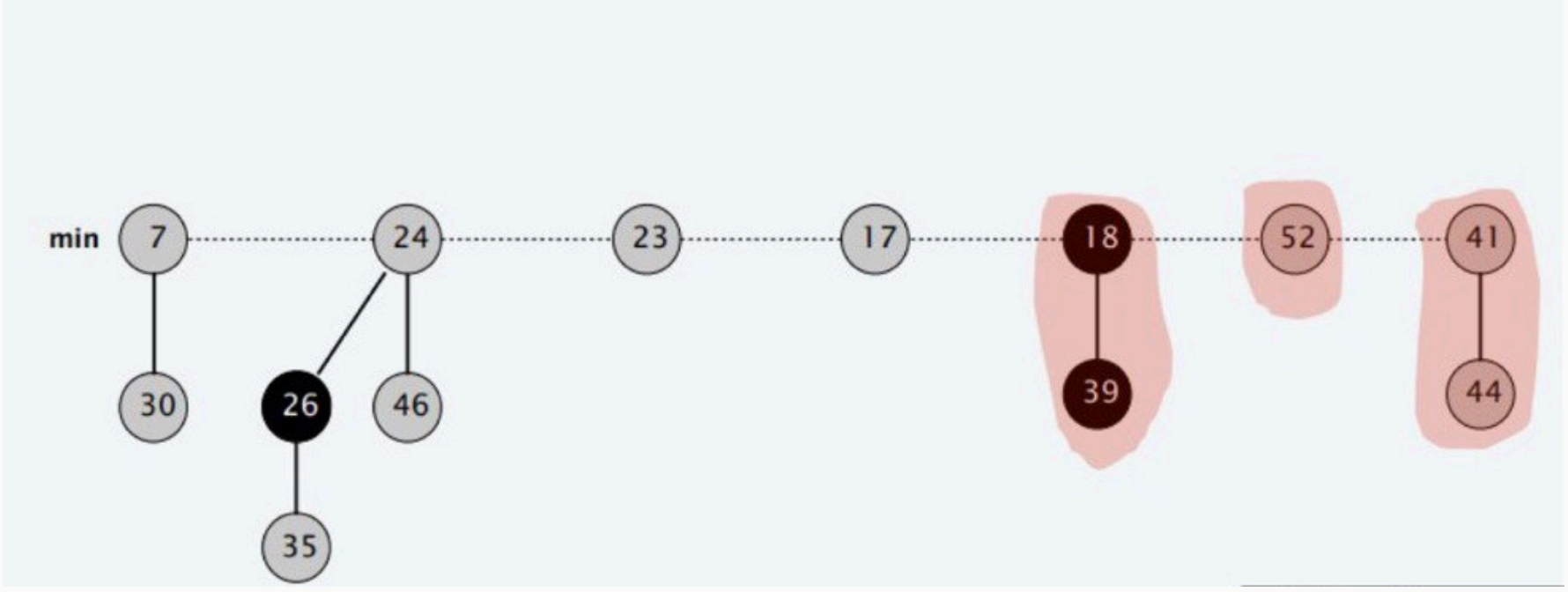
1. درختی که ریشه بزرگتر دارد را فرزند دیگری قرار می‌دهیم.
2. درخت نهایی دارای رنک k+1 خواهد بود.

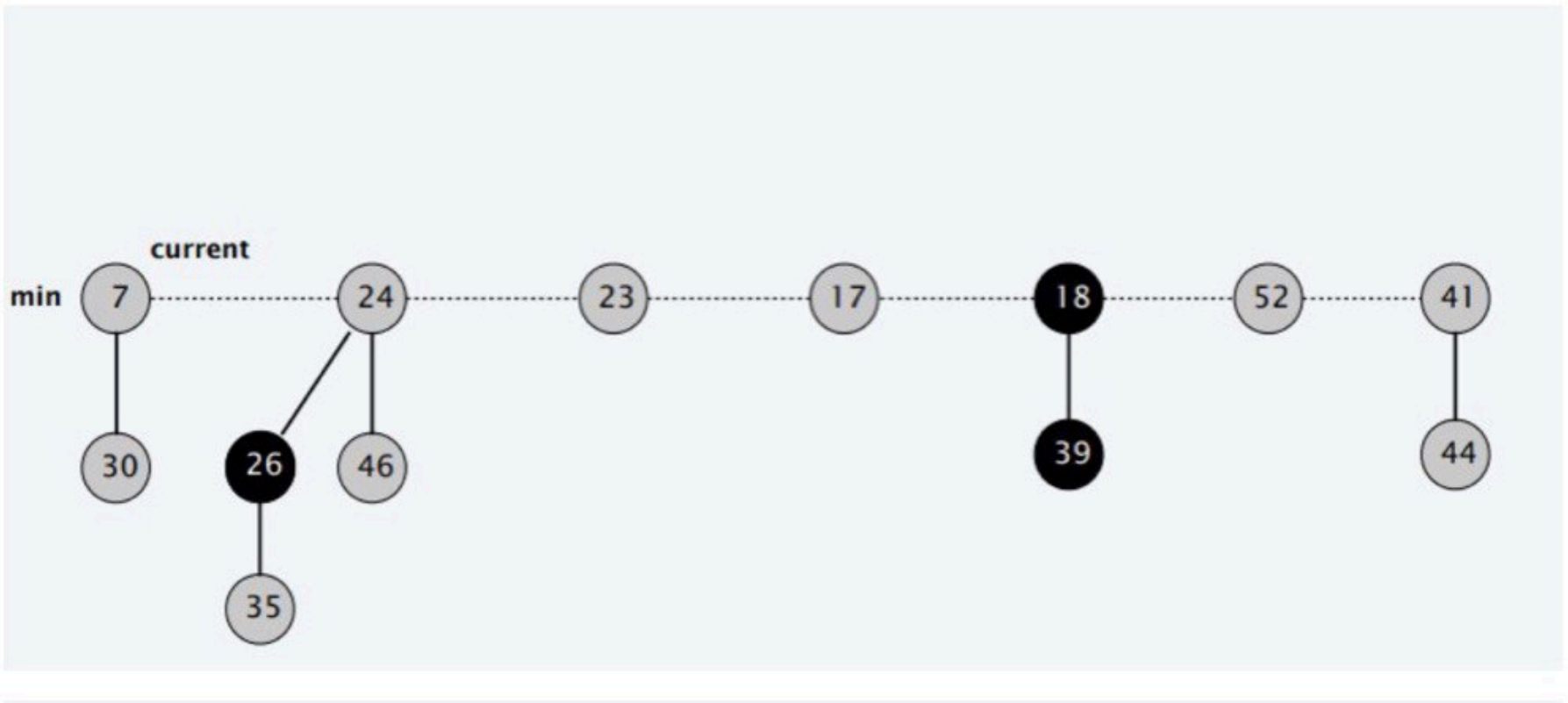
نکته: Merge همان Union مرحله پیشین است، اما تنها برای دو درخت با رنک برابر انجام می‌شود.

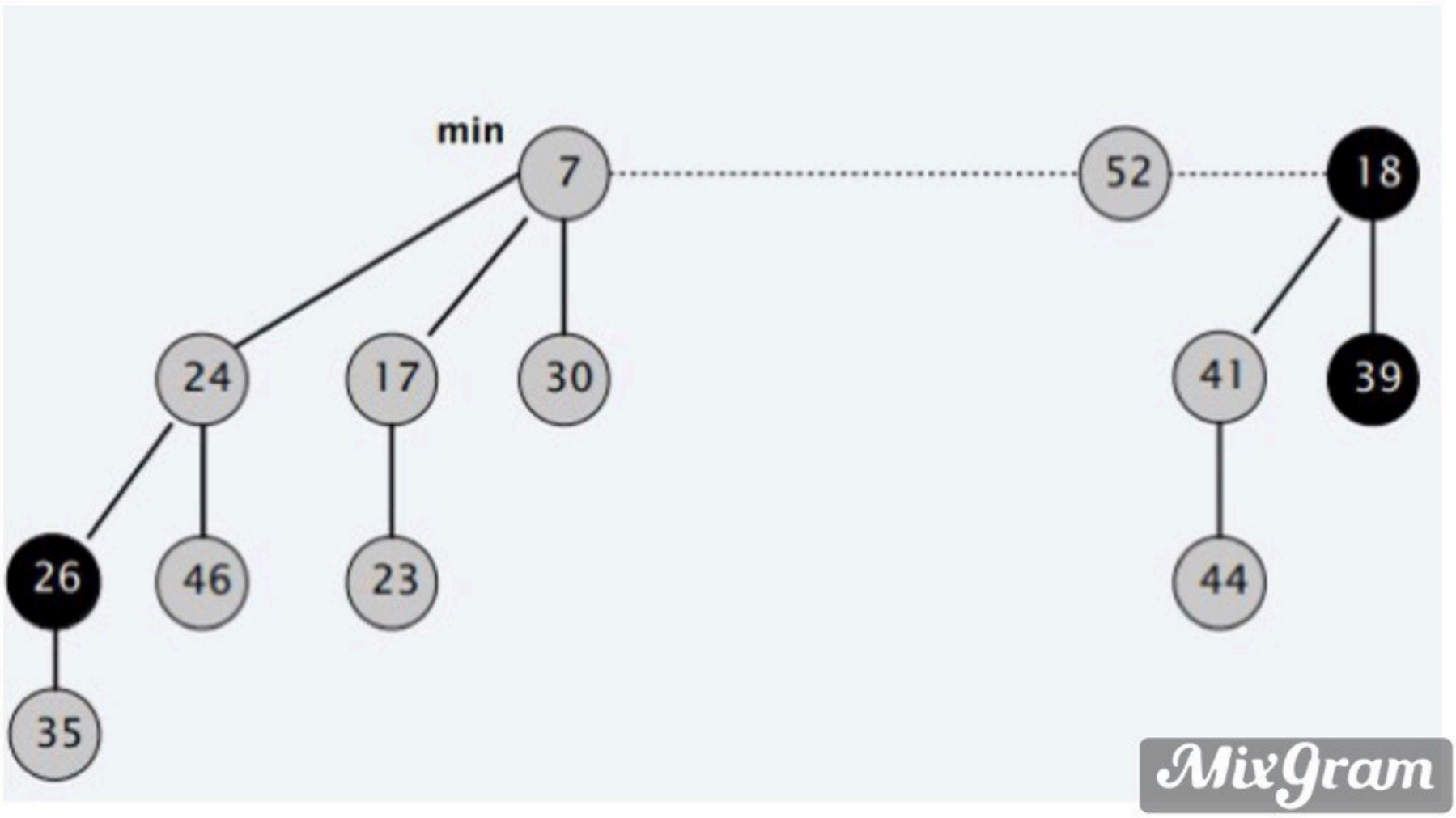
مراحل استخراج کمینه:

1. گره کمینه حذف می‌شود.
2. کمینه بروزرسانی و زیر درخت‌های گره حذف شده به root list اضافه می‌شوند.
3. یکی کردن درخت‌ها تا هیچ دو درختی دارای رنک یکسان نباشند.


min
7
24
23
17
3
30
26
46
18
52
41
35
39
44


min
7
24
23
17
18
52
41
30
26
46
39
44
35

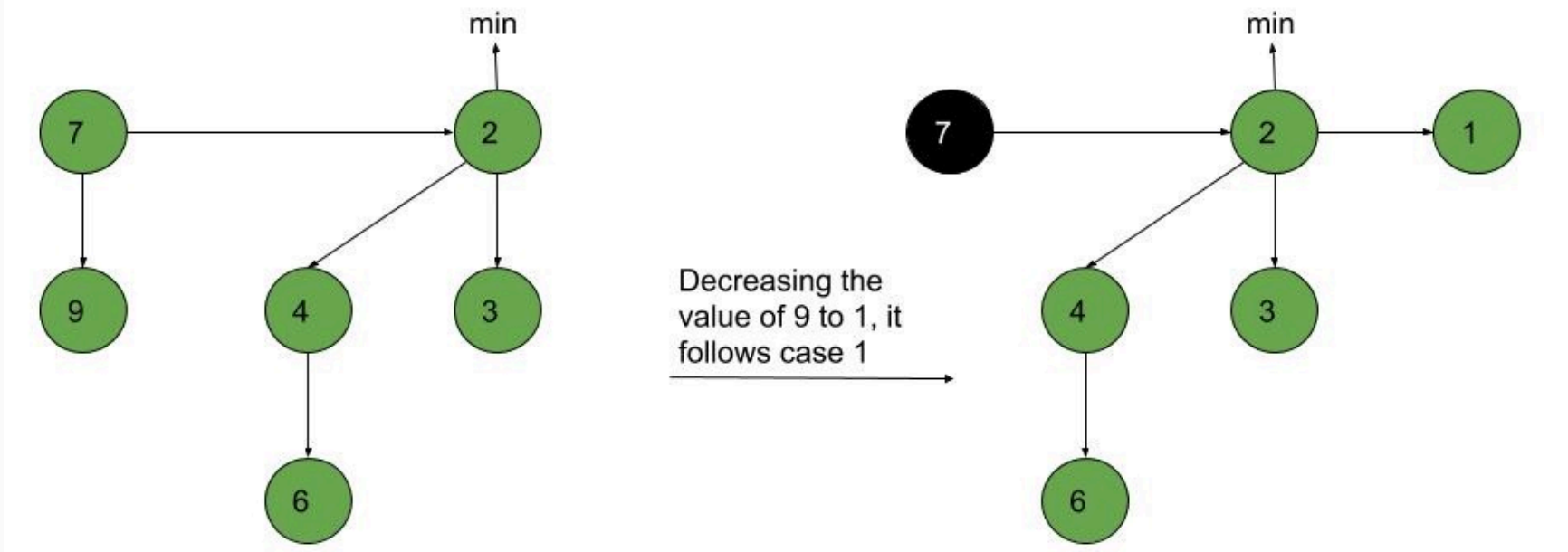




بخش دوم: کاهش کلید *O(1)* ✨

برای کاهش کلید، ابتدا یک فیلد علامت در هر گره اضافه می‌کنیم و سپس:

- مقدار گره V کاهش می‌یابد و سه حالت ممکن است رخ دهد:
  - حالت اول: خواص هرم نقض نشده → فقط مینیموم بروزرسانی می‌شود.
  - حالت دوم: خواص نقض شده و پدر علامت نخورده → ارتباط با پدر قطع، پدر علامت زده، درخت ریشه V به root list اضافه می‌شود.
  - حالت سوم: خواص نقض شده و پدر علامت خورده → زنجیره‌ای از قطع کردن و اضافه کردن به root list مطابق الگوریتم انجام می‌شود.



بخش سوم: حذف *O(log n)* ❌

مراحل حذف:

- مقدار گره با Decrease_Key به کمینه کاهش داده می‌شود.
- درخت ریشه V به root list اضافه می‌شود.
- الگوریتم Extract_Min اجرا می‌شود.

نکته: استفاده از Extract-min باعث می‌شود پیچیدگی O(log n) باشد.

پیاده‌سازی 🛠️

یافتن عنصر کمینه 🔍

در این قسمت کافی است مقدار آن **Node** که اشاره‌گر **heap** به آن اشاره می‌کند را چاپ کنیم.

In [10]:
```python
# مقدار کمینه را برمی‌گرداند
def get_minimum(self):
    if self.heap is None:
        return None
    return self.heap.value
```

حذف عنصر کمینه ❌

همان‌طور که می‌دانید، این قسمت یکی از پیچیده‌ترین بخش‌های این داده‌ساختار است! 🔧

به این منظور، تعدادی تابع کمکی تعریف کرده‌ایم. تابع کمکی اول **unMarkAndUnParentAll_** است که روی تمام برادران یک گره به نام **n** حرکت می‌کند، ابتدا **parent** آن‌ها را حذف می‌کند و سپس **marked** را برای همه آن‌ها به **false** تغییر می‌دهد.

In [11]:
```python
# حذف عنصر کمینه از هرم
def remove_minimum(self):
    z = self.heap
    if z is not None:
        # اضافه کردن فرزندان گره کمینه به root list
        if z.child is not None:
            self._unmark_and_unparent_all(z.child)
            self.heap = self._merge(self.heap, z.child)

        # حذف گره کمینه از root list
        if z.next == z:
            self.heap = None
        else:
            z.prev.next = z.next
            z.next.prev = z.prev
            self.heap = z.next
            self._consolidate()

        z.prev = z.next = z.child = z.parent = None
    return z.value if z else None
```

```
# حذف parent و علامت‌گذاری false برای همه فرزندان
def _unmark_and_unparent_all(self, node):
    if node is None:
        return
    current = node
    while True:
        current.marked = False
        current.parent = None
        current = current.next
        if current == node:
            break
```

اضافه کردن فرزند 👶

تابع کمکی دیگر، **addChild_** است که دو اشاره‌گر به دو گره دریافت می‌کند و یکی از گره‌ها را به لیست بچه‌های یک گره دیگر اضافه می‌کند.

طرز کار این متد به این شکل است که ابتدا **next** و **prev** و **parent** را برای گره فرزند مشخص می‌کند و پس از افزایش درجه‌ی گره پدر، با متد **merge_** که قبلاً پیاده‌سازی کرده‌ایم، آن گره را به جمع فرزندان گره پدر اضافه می‌کند.

In [12]:
```
# اضافه کردن یک فرزند به گره پدر
def _add_child(self, parent, child):
    child.prev = child.next = child
    child.parent = parent
    parent.degree += 1
    parent.child = self._merge(parent.child, child)
```

حذف کمینه 🗑️

سپس متد **removeMinimum_** را داریم که یک متد **private** برای حذف گرهِ داده شده است.

In [13]:
```
# حذف گره کمینه و بازگرداندن گره جدید با مقدار کمینه
def _remove_minimum(self, n):
    self._unmark_and_unparent_all(n.child)

    # اگر فقط یک گره در root list باشد
    if n.next == n:
        n = n.child
    else:
        # حذف n از لیست و ادغام فرزندان با لیست باقی‌مانده
        n.next.prev = n.prev
        n.prev.next = n.next
        n = self._merge(n.next, n.child)

    if n is None:
        return None

    trees = [None] * 64  # لیست درخت‌ها با درجات مختلف

    while True:
        d = n.degree
        if trees[d] is not None:
            t = trees[d]
            if t == n:
                break
            trees[d] = None
            if n.value < t.value:
                # حذف t از لیست و افزودن به فرزندان n
                t.prev.next = t.next
                t.next.prev = t.prev
                self._add_child(n, t)
            else:
                # حذف n از لیست و افزودن به فرزندان t
                t.prev.next = t.next
                t.next.prev = t.prev
                if n.next == n:
                    t.next = t.prev = t
                    self._add_child(t, n)
                    n = t
                else:
                    n.prev.next = t
                    n.next.prev = t
                    t.next = n.next
                    t.prev = n.prev
                    self._add_child(t, n)
                    n = t
            continue
        else:
            trees[d] = n
        n = n.next
```

```
    # پیدا کردن گره با مقدار کمینه در لیست نهایی
    min_node = n
    start = n
    while True:
        if n.value < min_node.value:
            min_node = n
        n = n.next
        if n == start:
            break
    return min_node
```

پیاده‌سازی removeMinimum

در نهایت با استفاده از این متدهای کمکی، متد نهایی **removeMinimum** که یک متد **public** است را پیاده‌سازی می‌کنیم.

در این متد ابتدا عنصر کمینه را ذخیره می‌کنیم. سپس متد **removeMinimum_** را به ازای آن گره صدا می‌زنیم. در نهایت حافظه‌ی مربوط به آن گره را آزاد می‌کنیم و مقدار آن گره را برمی‌گردانیم.

In [14]:
```
# حذف عنصر کمینه و بازگرداندن مقدار آن
def remove_minimum(self):
    old = self.heap  # ذخیره گره کمینه فعلی
    self.heap = self._remove_minimum(self.heap)  # بروزرسانی heap با حذف کمینه
    ret = old.value if old else None  # مقدار گره حذف‌شده
    old.prev = old.next = old.child = old.parent = None  # آزادسازی حافظه
    return ret
```

کاهش کلید

ابتدا متدی کمکی به نام **cut_** تعریف می‌کنیم. وظیفه‌ی این متد این است که یک گره به نام **n** دریافت کند، آن گره را از پدرش جدا کند و سپس آن گره را به یک لیست پیوندی (که در ورودی تابع داده شده) اضافه کند.

In [15]:
```
# جدا کردن گره n از پدرش و افزودن آن به لیست ریشه‌ها
def _cut(self, heap, n):
    if n.next == n:
        # اگر n تنها فرزند باشد، لیست فرزندان پدر را خالی می‌کنیم
        n.parent.child = None
    else:
        # حذف n از لیست فرزندان پدر
        n.next.prev = n.prev
        n.prev.next = n.next
        n.parent.child = n.next

    # تنظیمات اولیه برای افزودن به root list
    n.next = n.prev = n
    n.marked = False
    return self._merge(heap, n)
```

کاهش داخلی کلید

سپس متد داخلی **decreaseKey_** را پیاده‌سازی می‌کنیم. وظیفه‌ی این متد این است که مقدار یک گره را به مقدار داده شده «کاهش» دهد. گره مذکور، مقدار مذکور و یک لیست پیوندی دو طرفه، ورودی‌های این متد هستند.

In [16]:
```
# کاهش مقدار گره n به مقدار جدید value
def _decrease_key(self, heap, n, value):
    if n.value < value:
        return heap  # مقدار جدید باید کمتر باشد

    n.value = value  # به‌روزرسانی مقدار گره

    if n.parent:
        if n.value < n.parent.value:
            heap = self._cut(heap, n)
            parent = n.parent
            n.parent = None

            # زنجیره‌ای از قطع کردن در صورت علامت‌خورده بودن پدر
            while parent and parent.marked:
                heap = self._cut(heap, parent)
                n = parent
                parent = n.parent
                n.parent = None
```

```
            # علامت‌گذاری پدر در صورت وجود پدر بزرگ
            if parent and parent.parent:
                parent.marked = True
    else:
        # اگر گره در root list باشد، مینیموم را به‌روزرسانی کن
        if n.value < heap.value:
            heap = n

    return heap
```

کاهش کلید نهایی ✨

در نهایت متد **decreaseKey** را پیاده‌سازی می‌کنیم. در این متد تنها کافی است که عضو کمینه‌ی لیست پیوندی ریشه‌های هرم‌ها را به همراه ۲ ورودی دیگر خود این متد، به متد قبلی پاس دهیم.

In [17]:
```
# کاهش مقدار گره n به مقدار جدید value (متد عمومی)
def decrease_key(self, n, value):
    self.heap = self._decrease_key(self.heap, n, value)
```

پیدا کردن عنصر 🔍

در نهایت، به پیاده‌سازی متدی برای پیدا کردن یک مقدار در بین تمام گره‌ها می‌پردازیم. البته می‌دانیم انجام دادن این کار، لزوما روشی بهینه ندارد و پیچیدگی زمانی‌ش مانند سایر عملیات‌هایی که تا کنون انجام دادیم، خوب نیست!

برای انجام این کار، به نوعی یک الگوریتم **DFS** روی تمام هرم‌ها اجرا می‌کنیم تا مقدار مذکور را بیابیم.

لذا در ابتدا یک متد به نام **find** می‌سازیم که متد درونی **DFS** را صدا می‌زند. اسم این متد درونی را **_find** می‌گذاریم.

In [18]:
```
# جستجوی مقدار در کل هرم (متد عمومی)
def find(self, value):
    return self._find(self.heap, value)

# جستجوی DFS در لیست پیوندی و فرزندان برای یافتن مقدار
def _find(self, node, value):
    if node is None:
        return None
    visited = set()
    stack = [node]
    while stack:
        current = stack.pop()
        if current in visited:
            continue
        visited.add(current)
        if current.value == value:
            return current
        # افزودن فرزندان به پشته
        if current.child:
            child = current.child
            child_nodes = []
            while True:
                child_nodes.append(child)
                child = child.next
                if child == current.child:
                    break
            stack.extend(child_nodes)
        # افزودن برادران به پشته
        if current.next != node:
            stack.append(current.next)
    return None
```

متد DFS داخلی 🧭

متد **_find** به عنوان ورودی یک لیست پیوندی می‌گیرد و یک مقدار. قاعدتا این لیست پیوندی توسّط یکی از اعضایش مشخّص می‌شود.

در این متد ابتدا مطمئن می‌شویم که لیست کنونی خالی نباشد. سپس روی تمام اعضای این لیست حرکت می‌کنیم. به هر گره که رسیدیم، ابتدا چک می‌کنیم خود این گره، گره مطلوب ماست یا خیر؛ اگر گره مطلوب ما بود، آدرس آن را برمی‌گردانیم. در غیر این

صورت همین تابع را روی لیست بچّه‌هایش اجرا می‌کنیم.

بنابراین تمام گره‌های درخت با الگوریتم **DFS** جست‌وجو می‌شوند تا به گره‌ای با مقدار مطلوب برسیم.

In [19]:
```python
# برای یافتن گره با مقدار مشخص DFS جستجوی
def _find(self, heap, value):
    n = heap
    if n is None:
        return None
    while True:
        if n.value == value:
            return n
        ret = self._find(n.child, value)
        if ret:
            return ret
        n = n.next
        if n == heap:
            break
    return None
```

مثال 🔢

مرتّب‌سازی

برای نشان دادن طرز استفاده از این داده ساختار، مثالی برای مرتب‌سازی ۱۰ عدد صحیح آورده‌ایم.

In [28]:
```python
class Node:
    def __init__(self, value):
        self.value = value
        self.degree = 0
        self.marked = False
        self.parent = None
        self.child = None
        self.prev = self
        self.next = self


class FibonacciHeap:
    def __init__(self):
        self.heap = None

    def is_empty(self):
        return self.heap is None

    def _singleton(self, value):
        return Node(value)

    def _merge(self, a, b):
        if a is None:
            return b
        if b is None:
            return a
        if a.value > b.value:
            a, b = b, a
        a_next = a.next
        b_prev = b.prev
        a.next = b
        b.prev = a
        a_next.prev = b_prev
        b_prev.next = a_next
        return a

    def insert(self, value):
        node = self._singleton(value)
        self.heap = self._merge(self.heap, node)
        return node

    def get_minimum(self):
        return self.heap.value if self.heap else None

    def _unmark_and_unparent_all(self, node):
        if node is None:
            return
        current = node
        while True:
            current.marked = False
            current.parent = None
            current = current.next
            if current == node:
                break

    def _add_child(self, parent, child):
        child.prev = child.next = child
        child.parent = parent
        parent.degree += 1
```

```
        parent.child = self._merge(parent.child, child)

    def _remove_minimum(self, n):
        self._unmark_and_unparent_all(n.child)
        if n.next == n:
            n = n.child
        else:
            n.next.prev = n.prev
            n.prev.next = n.next
            n = self._merge(n.next, n.child)

        if n is None:
            return None

        trees = [None] * 64
        while True:
            d = n.degree
            if trees[d] is not None:
                t = trees[d]
                if t == n:
                    break
                trees[d] = None
                if n.value < t.value:
                    t.prev.next = t.next
                    t.next.prev = t.prev
                    self._add_child(n, t)
                else:
                    t.prev.next = t.next
                    t.next.prev = t.prev
                    if n.next == n:
                        t.next = t.prev = t
                        self._add_child(t, n)
                        n = t
                    else:
                        n.prev.next = t
                        n.next.prev = t
                        t.next = n.next
                        t.prev = n.prev
                        self._add_child(t, n)
                        n = t
                continue
            else:
                trees[d] = n
            n = n.next

        min_node = n
        start = n
        while True:
            if n.value < min_node.value:
                min_node = n
            n = n.next
            if n == start:
                break
        return min_node

    def remove_minimum(self):
        old = self.heap
        self.heap = self._remove_minimum(self.heap)
        ret = old.value if old else None
        old.prev = old.next = old.child = old.parent = None
        return ret
```

In [29]:
```
# فرض بر این است که کلاس‌های FibonacciHeap و Node قبلاً تعریف شده‌اند

def fibonacci_heap_sort(numbers):
    fh = FibonacciHeap()
    for num in numbers:
        fh.insert(num)

    sorted_list = []
    while not fh.is_empty():
        sorted_list.append(fh.remove_minimum())
    return sorted_list

# تست با ۱۰ عدد
nums = [42, 17, 8, 99, 23, 4, 73, 15, 61, 30]
sorted_nums = fibonacci_heap_sort(nums)
print("The sorted order of these 10 numbers:")
print(sorted_nums)
```

```
The sorted order of these 10 numbers:
[4, 8, 15, 17, 23, 30, 42, 61, 73, 99]
```

دایسترا 🚀

در این مثال نیز ابتدا یک کلاس برای ذخیره‌سازی یک گراف جهت‌دار ایجاد می‌کنیم و سپس روی این گراف با استفاده از الگوریتم دایسترا کوتاه‌ترین مسیر از راسی دلخواه به سایر رئوس را به دست می‌آوریم.

In [32]:
```
class Node:
    def __init__(self, value):
        self.value = value
        self.degree = 0
```

```
        self.marked = False
        self.parent = None
        self.child = None
        self.prev = self
        self.next = self


class FibonacciHeap:
    def __init__(self):
        self.heap = None

    def is_empty(self):
        return self.heap is None

    def _singleton(self, value):
        return Node(value)

    def _merge(self, a, b):
        if a is None:
            return b
        if b is None:
            return a
        if a.value > b.value:
            a, b = b, a
        a_next = a.next
        b_prev = b.prev
        a.next = b
        b.prev = a
        a_next.prev = b_prev
        b_prev.next = a_next
        return a

    def insert(self, value):
        node = self._singleton(value)
        self.heap = self._merge(self.heap, node)
        return node

    def get_minimum(self):
        return self.heap.value if self.heap else None

    def _unmark_and_unparent_all(self, node):
        if node is None:
            return
        current = node
        while True:
            current.marked = False
            current.parent = None
            current = current.next
            if current == node:
                break

    def _add_child(self, parent, child):
        child.prev = child.next = child
        child.parent = parent
        parent.degree += 1
        parent.child = self._merge(parent.child, child)

    def _remove_minimum(self, n):
        self._unmark_and_unparent_all(n.child)
        if n.next == n:
            n = n.child
        else:
            n.next.prev = n.prev
            n.prev.next = n.next
            n = self._merge(n.next, n.child)

        if n is None:
            return None

        trees = [None] * 64
        while True:
            d = n.degree
            if trees[d] is not None:
                t = trees[d]
                if t == n:
                    break
                trees[d] = None
                if n.value < t.value:
                    t.prev.next = t.next
                    t.next.prev = t.prev
                    self._add_child(n, t)
                else:
                    t.prev.next = t.next
                    t.next.prev = t.prev
                    if n.next == n:
                        t.next = t.prev = t
                        self._add_child(t, n)
                        n = t
                    else:
                        n.prev.next = t
                        n.next.prev = t
                        t.next = n.next
                        t.prev = n.prev
                        self._add_child(t, n)
                        n = t
                continue
            else:
                trees[d] = n
            n = n.next

        min_node = n
```

```
        start = n
        while True:
            if n.value < min_node.value:
                min_node = n
            n = n.next
            if n == start:
                break
        return min_node

    def remove_minimum(self):
        old = self.heap
        self.heap = self._remove_minimum(self.heap)
        ret = old.value if old else None
        old.prev = old.next = old.child = old.parent = None
        return ret

    def _cut(self, heap, n):
        if n.next == n:
            n.parent.child = None
        else:
            n.next.prev = n.prev
            n.prev.next = n.next
            n.parent.child = n.next
        n.next = n.prev = n
        n.marked = False
        return self._merge(heap, n)

    def _decrease_key(self, heap, n, value):
        if n.value < value:
            return heap
        n.value = value
        if n.parent:
            if n.value < n.parent.value:
                heap = self._cut(heap, n)
                parent = n.parent
                n.parent = None
                while parent and parent.marked:
                    heap = self._cut(heap, parent)
                    n = parent
                    parent = n.parent
                    n.parent = None
                if parent and parent.parent:
                    parent.marked = True
        else:
            if n.value < heap.value:
                heap = n
        return heap

    def decrease_key(self, n, value):
        self.heap = self._decrease_key(self.heap, n, value)
```

In [33]:

```
class Graph:
    def __init__(self, n):
        self.n = n
        self.adj = [[] for _ in range(n)]  # لیست مجاورت برای هر رأس

    def add_edge(self, u, v, w):
        self.adj[u].append((v, w))  # یال جهت‌دار از u به v با وزن w

    def dijkstra(self, source):
        INF = 10**9
        fh = FibonacciHeap()
        pos = [None] * self.n
        distance = [INF] * self.n
        distance[source] = 0

        # درج اولیه همه رأس‌ها در صف اولویت
        for u in range(self.n):
            pos[u] = fh.insert((distance[u], u))

        while not fh.is_empty():
            dist_u, u = fh.remove_minimum()
            for v, w in self.adj[u]:
                if distance[u] + w < distance[v]:
                    distance[v] = distance[u] + w
                    fh.decrease_key(pos[v], (distance[v], v))

        # چاپ فاصله‌ها از رأس مبدأ
        for i in range(self.n):
            print(distance[i], end="\n" if i == self.n - 1 else " ")
```

In [34]:

```
g = Graph(5)
g.add_edge(0, 1, 5)
g.add_edge(0, 2, 10)
g.add_edge(1, 2, 2)
g.add_edge(2, 1, 5)
g.add_edge(2, 4, 3)
g.add_edge(1, 4, 10)
g.add_edge(4, 3, 5)

g.dijkstra(0)
```

```
0 5 7 15 10
```

📘 چرا هرم فیبوناچی؟

سوال اصلی این است چرا این داده‌ساختار در اسم خود کلمه فیبوناچی را دارد؟

برای پاسخ دادن به این سوال باید تمامی چیزهایی که تا به‌حال آموخته‌ایم را در کنار هم بگذاریم تا به جواب و چرایی آن دست پیدا کنیم.

همانطور که تا به‌حال متوجه شده‌اید هرم فیبوناچی از مجموعه درخت‌های کوچکتر به شکل (heap tree) ساخته شده است که در مجموعه از محدودیت‌ها قرار دارند. در اصل این درخت‌ها به گونه‌ای قرار دارند که درخت با اردر n از حداقل F(n+2) راس تشکیل شده باشد که این تابع F همان تابع فیبوناچی است.

برای اینکه ببینیم چرا این قضیه همواره برقرار است، بیایید بار دیگر به مراحل ساخت این درخت‌ها بازگردیم. در ابتدا هرگاه یک راس به هرم فیبوناچی اضافه می‌شود یک درخت جدید به آن اختصاص داده می‌شود که فقط همان راس را دارد و از اردر صفر است. هرگاه یک مقدار نیز حذف می‌شود، در نتیجه مراحلی که پیشتر کامل توضیح داده شد، برخی از این درخت‌ها با هم ترکیب می‌شوند که در نتیجه تعداد درخت‌ها بی‌رویه زیاد نشود.

وقتی این درخت‌ها با هم ترکیب می‌شوند، همانطور که می‌دانید، فقط آن‌هایی که از یک اردر هستند با هم ترکیب می‌شوند.

مراحل ترکیب شدن نیز اینگونه بود که از میان دو درختی که در حال ترکیب شدن با هم هستند، آن درختی که ریشه بیشتری دارد فرزند مستقیم دیگری می‌شود. یکی از نتایج این مرحله این است که در نهایت درخت‌های از اردر n همواره n فرزند خواهند داشت.

یکی از مواردی که توضیح داده شد این است که در این داده‌ساختار عمل کاهش‌کلید در زمان سرشکن O(1) انجام می‌شود. یکی از مشکلات انجام این کار این است که در عملیات کاهش کلید در عمل فرزندان از پدر جدا می‌شدند، پس اگر این اتفاق تعداد زیادی رخ دهد، ما در نهایت ممکن است یک درخت با اردر بالا اما تعداد راس بسیار اندک داشته باشیم. زمان خوب کارکرد این داده‌ساختار دقیقا وقتی گارانتی می‌شود که ما درخت‌های با اردر زیاد و راس‌های زیاد داشته باشیم.

حال چه‌کار کنیم که این مشکل رخ ندهد؟ برای این کار یک قانون قرار داده می‌شود که می‌گوید: اگر شما دو فرزند را از یک درخت بریدید، باید در عوض کل آن درخت را از پدرش جدا کنید (اینجا همان مفهوم مارک کردن که در پیاده‌سازی استفاده کردیم نمود پیدا می‌کند).

حال به پاسخ سوال اصلی و دلیل آن نزدیک‌تر می‌شویم. تا به‌حال موارد زیر را فهمیده‌ایم:

- درختی از اردر n دقیقا n فرزند دارد
- درخت اردر n از ترکیب کردن دو درخت از اردر n-1 و گذاشتن یکی فرزند دیگری بدست می‌آید
- اگر یک درخت دو فرزندش جدا شوند، خودش نیز از پدرش جدا می‌شود

حال بیایید با مثال و استقرا جلو برویم:

کوچکترین درخت از رتبه 0 تنها یک راس دارد.

کوچکترین درخت از رتبه 1 نیز یک درخت با یک ریشه و یک فرزند است.

اما در مورد رتبه 2 چه می‌توان گفت؟ با توجه به قوانینی که بالا گفته شد، این درخت حتماً دو فرزند دارد و از ترکیب دو درخت با رتبه 1 تشکیل شده‌است. در نتیجه درخت در ابتدا دو فرزند دارد، یکی از رتبه صفر و دیگری از رتبه یک.

اگر فرزند درخت با رتبه یک را از آن جدا کنیم، ما با یک درخت از رتبه یک باقی می‌مانیم که دو فرزند دارد و هر دو از رتبه صفر هستند. پس کوچکترین یک درخت دودویی با سه راس می‌شود.

حال وارد مرحله استقرا می‌شویم: کوچکترین درخت از رتبه n+2 به اینگونه خواهد بود:

اول یک درخت نرمال از رتبه n+2 می‌سازیم که فرزندانی از رتبه n+1, n, n-1, ..., 2, 1, 0 دارد. سپس این درخت‌ها را به اندازه ممکن کوچک می‌کنیم با بریدن زیر درخت‌ها از آنها به‌گونه‌ای که هیچگاه دو فرزند از یک راس جدا نشوند. اینکار درختی با فرزندان از رتبه: n, n-1, ..., 1, 0, 0 تشکیل می‌دهد.

حال ما می‌توانیم رابطه بازگشتی را بنویسیم:

pseudo code:

ST(0) = 1
ST(1) = 2
ST(n+2) = ST(n) + ST(n-1) + .. + ST(1) + ST(0) + ST(0)

چرا هرم فیبوناچی؟ - دنباله فیبوناچی 📘

حال اگر مورد بالا محاسبه کنیم می‌بینید دقیقا دنباله فیبوناچی رخ داده‌است.

حال اگر از شما بپرسند چرا هرم فیبوناچی می‌گویید، چون هرکدام از درخت‌ها از حداقل F(n+2) راس تشکیل شده است.

برای دیدن نحوه گرافیکی مراحل می‌توانید از لینک زیر نیز بهره ببرید:

Fibonacci Heap

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل پنجم: روش تقسیم و حل

بخش اول: «کاربرد روابط بازگشتی»، مسئله‌ی «توان» و مسئله‌ی «نزدیک‌ترین زوج نقاط»

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 📖

معرفی روش تقسیم و حل ⚙️

فرایند به دست آوردن پاسخ مسئله به روش **تقسیم و حل (Divide and Conquer)** شامل سه مرحله‌ی اصلی است:

- **تقسیم/شکستن:** در این مرحله، مسئله را به چند «زیرمسئله»ی کوچک‌تر با نوع یکسان یا مشابه، تقسیم می‌کنیم. معمولاً این تقسیم، به صورت بازگشتی و تا جایی که دیگر تقسیم کردن ممکن نباشد، ادامه می‌یابد.
- **حل:** در این مرحله، زیرمسئله‌های به دست آمده در پایین‌ترین سطح، حل می‌شوند.
- **ادغام/ترکیب:** برای این که بتوان زیرمسئله‌های سطح بالاتر را حل کرد، لازم است زیرمسئله‌های سطح پایین با هم ترکیب شوند، یعنی پاسخ مسئله‌ی بزرگ‌تر با توجه به پاسخ آن‌ها ساخته شود. این مرحله را تا جایی که به پاسخ مسئله‌ی اصلی برسیم، ادامه می‌دهیم.

مسئله‌های زیر، نمونه‌هایی از مسائلی هستند که با روش تقسیم و حل، قابل حل هستند:

- مرتب‌سازی ادغامی (Merge Sort)

- مرتب‌سازی سطلی (Bucket Sort)
- ضرب چندجمله‌ای‌ها (Polynomial Multiplication)
- مسئله‌ی کاشی‌کاری یا فرش کردن (Tiling Problem)
- مسئله‌ی آسمان‌خراش‌ها (Skyline Problem)
- مسئله‌ی زمان‌بندی تورنمنت (Tournament Scheduling Problem)
- مسئله‌ی آدم مشهور (The Celebrity Problem)

تعدادی مسئله‌ی دیگر را می‌توانید از لینک زیر مشاهده کنید:

LeetCode - Divide and Conquer

کاربرد روابط بازگشتی در تقسیم و حل 🔁

در دروس **ساختمان‌های گسسته** و **ساختمان داده‌ها و الگوریتم‌ها**، با رابطه‌های بازگشتی و روش‌های حل آن‌ها آشنا شدید. برای تحلیل زمانی الگوریتم‌های تقسیم و حل از این روابط استفاده می‌شود. در ادامه، با کاربرد این روابط آشنا می‌شویم.

مسئله‌ای با اندازه‌ی ورودی n داده شده است. با روش تقسیم و حل، آن را به a زیرمسئله تقسیم می‌کنیم که اندازه‌ی هرکدام، b/1 اندازه‌ی ورودی است. هم‌چنین هزینه‌ی مرحله‌ی ادغام، f(n) است.

پیچیدگی زمانی این مسئله به کمک رابطه‌ی بازگشتی زیر به دست می‌آید:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

روش‌های مختلفی برای حل این رابطه وجود دارد. یکی از آن‌ها استفاده از **قضیه‌ی اساسی تحلیل الگوریتم‌ها (Master Theorem)** است.

قضیه‌ی اساسی تحلیل الگوریتم‌ها 📘

تعریف می‌کنیم:

$$c_{crit} = \log_b a$$

طبق قضیه‌ی اساسی:

- **حالت 1:** اگر $f(n) = O(n^c)$ و $c < c_{crit}$ باشد، خواهیم داشت: $T(n) = \Theta(n^{c_{crit}})$
- **حالت 2:** اگر $f(n) = \Theta(n^{c_{crit}})$ باشد، خواهیم داشت: $T(n) = \Theta(n^{c_{crit}} \log n)$
- **حالت 3:** اگر $f(n) = \Omega(n^c)$ و $c > c_{crit}$ باشد، خواهیم داشت: $T(n) = \Theta(f(n))$ (البته به شرطی که مقدار $c < 1$ موجود باشد، به طوری که برای تمام مقادیر n که به اندازه‌ی کافی بزرگ هستند، داشته باشیم: $af(\frac{n}{b}) \leq cf(n)$)

حال که با ایده‌ی کلی روش تقسیم و حل و کاربرد رابطه‌های بازگشتی در آن آشنا شدید، به معرفی و حل مسئله‌هایی که با این روش حل می‌شوند، می‌پردازیم.

مسئله‌ی توان ⚡

دو عدد n و x داده شده‌است؛ مقدار x به توان n را به‌دست آورید.

نکته: اکیداً توصیه می‌شود قبل از خواندن راه‌حل، خودتان برای دقایقی به دنبال راه‌حل این مسئله بگردید.

روش اولیه 🧮

اولین راهی که برای حل این مسئله به ذهن می‌رسد، n بار ضرب کردن عدد x در خودش است. می‌توان کد این الگوریتم را در ادامه دید.

In [1]:
```python
# الگوریتم ساده برای محاسبه توان x به n
def naive_algorithm(x, n):
    answer = 1  # مقدار اولیه پاسخ را برابر ۱ قرار می‌دهیم
    for i in range(n):
        answer = answer * x  # در هر مرحله x را در پاسخ ضرب می‌کنیم
    return answer

# تست تابع با ورودی ۵ به توان ۴ → خروجی باید ۶۲۵ باشد
print(naive_algorithm(5, 4))
```

```
625
```

پیچیدگی زمانی ⏱️

در این الگوریتم لازم است عدد x را n بار در خودش ضرب کنیم. با در نظر گرفتن زمان ثابت برای عمل ضرب، زمان اجرا در $O(n)$ است.

روش تقسیم و حل ⚡

حال سعی می‌کنیم الگوریتمی با پیچیدگی زمانی بهتر بسازیم. در راه اول n بار عدد x را در خودش ضرب می‌کردیم و این کار زمان بسیار زیادی از ما می‌گرفت. حال سعی می‌کنیم با استفاده از تقسیم و حل راه‌حلی جدید ارائه دهیم.

در این راه‌حل به‌جای آن که x^{n-1} را محاسبه کنیم و در x ضرب کنیم، مقدار $x^{\frac{n}{2}}$ را محاسبه می‌کنیم و در خودش ضرب می‌کنیم تا x^n به دست آید.

کد مربوط به این الگوریتم را در ادامه می‌بینید.

In [2]:
```python
# الگوریتم تقسیم و حل برای محاسبه توان x به n با زمان بهینه
def divide_and_conquer_algorithm(x, n):
    if n == 0:
        return 1  # هر عددی به توان صفر برابر ۱ است

    answer = divide_and_conquer_algorithm(x, n // 2)  # محاسبه توان نصف به صورت بازگشتی

    if n % 2 == 0:
        return answer * answer  # اگر n زوج باشد، کافیست توان نصف را در خودش ضرب کنیم
    else:
        return answer * answer * x  # اگر n فرد باشد، یک بار دیگر در x ضرب می‌کنیم

# تست تابع با ورودی ۵ به توان ۴ → خروجی باید ۶۲۵ باشد
print(divide_and_conquer_algorithm(5, 4))
```

```
625
```

پیچیدگی زمانی ⏱️

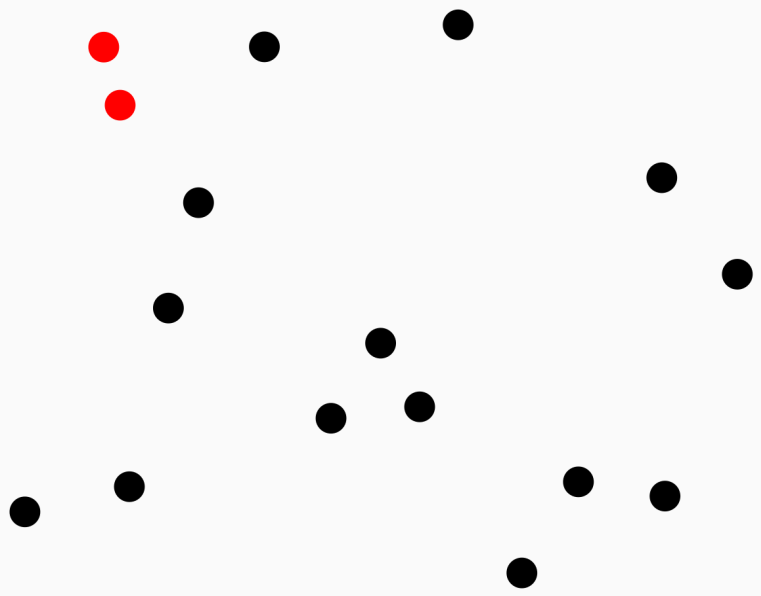
در تحلیل زمانی این راه‌حل می‌دانیم برای محاسبه‌ی x^n می‌توان x^{n-1} را محاسبه کرد و با تعدادی عمل ضرب و یک دستور شرطی مقدار x^n را محاسبه کرد.
پس اگر زمان این الگوریتم را برابر $\mathcal{T}(n)$ در نظر بگیریم، می‌دانیم که $\mathcal{T}(n) = \mathcal{T}(\frac{n}{2}) + \mathcal{O}(1)$ که با حل معادله‌ی بازگشتی، زمان الگوریتم در $O(\log n)$ است.

مسئله‌ی نزدیک‌ترین زوج نقاط 🌌

زمانی را در نظر بگیرید که به همراه دوستان خود، مشغول حل کردن تمرین‌های درس طراحی الگوریتم بوده‌اید و کارتان تا پاسی از شب طول کشیده است. هنگام خروج از دانشکده، توجهتان به آسمان صاف بالای سرتان جلب می‌شود و سعی می‌کنید چند ستاره را که می‌شناسید، پیدا کنید. دوستتان که هنوز در فضای تمرین‌ها قرار دارد، از شما می‌پرسد که «کدام جفت ستاره، دارای کمترین فاصله با هم هستند؟». کار شما نسبتاً آسان است، چون تعداد ستاره‌های قابل مشاهده در آسمان تهران اندک است. اما اگر با دوستتان به منطقه‌ی «سه‌قلعه» سفر کنید و هنگامی که در حال لذت بردن از تماشای آسمان پرستاره هستید، دوستتان این سؤال را از شما بپرسد، چه کار می‌کنید؟!

آسمان رصدگاه سه‌قلعه

برای رفع نیازهای روزمره‌ی ما، رایانه‌ها نیاز دارند تا به سؤالاتی از جنس سؤالات دوستتان پاسخ دهند. البته، تعداد نقاط ورودی داده شده به آن‌ها بسیار بیشتر از ستارگان قابل مشاهده با چشم در یک آسمان تاریک (2000 تا 3000 ستاره) است. هم‌چنین، گاهی لازم است جست‌وجو در فضایی با تعداد ابعاد زیاد انجام شود. بهینه‌سازی زمان اجرای الگوریتم در این فضاها، امری مهم و حیاتی است.
در مسئله‌ی نزدیک‌ترین زوج نقاط، n نقطه داریم، و می‌خواهیم دو نقطه‌ای را که کمترین فاصله را از هم دارند، پیدا کنیم.



این مسئله در حالت کلی و برای فضای k بعدی هم مطرح می‌شود. در اینجا تمرکزمان روی صفحه $(k = 2)$ است. فاصله را هم همان فاصله‌ی اقلیدسی در نظر می‌گیریم. یعنی:

$$(d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2})$$

In [3]:
```python
# تعریف کلاس Point برای نمایش مختصات یک نقطه در صفحه
class Point:
    def __init__(self, x, y):
        self.x = x  # مختصات x نقطه
        self.y = y  # مختصات y نقطه
```

In [4]:
```python
# تابع محاسبه فاصله بین دو نقطه به روش اقلیدسی (بدون جذر برای سادگی و مقایسه)
def distance(p1, p2):
    dx = p1.x - p2.x  # اختلاف مختصات x
    dy = p1.y - p2.y  # اختلاف مختصات y
    return dx * dx + dy * dy  # مربع فاصله اقلیدسی
```

روش اولیه 💡

راه حل ساده این است که فاصله‌ی تمام زوج‌های نقاط را حساب کنیم و سپس بین آن‌ها کمینه را پیدا کنیم.

In [5]:
```python
# تابع bruteForce برای پیدا کردن کمترین فاصله بین زوج نقاط
def brute_force(points):
    min_dist = float('inf')  # مقدار اولیه فاصله کمینه را بی‌نهایت در نظر می‌گیریم
    n = len(points)
    for i in range(n):
        for j in range(i + 1, n):  # بررسی تمام زوج‌های ممکن
            d = distance(points[i], points[j])  # محاسبه فاصله بین دو نقطه
            if d < min_dist:
                min_dist = d  # به‌روزرسانی فاصله کمینه در صورت یافتن مقدار بهتر
    return min_dist
```

In [6]:
```python
# تست تابع با سه نقطه اول از لیست
p = [Point(1, 2), Point(3, 1), Point(4, 5), Point(3, 4)]
print(brute_force(p[:3]))  # خروجی باید فاصله کمینه بین سه نقطه اول باشد
```

```
5
```

پیچیدگی زمانی ⏱️

باید $\binom{n}{2}$ تا فاصله حساب کنیم. در نتیجه پیچیدگی زمانی در $O(n^2)$ است.

روش تقسیم و حل 🔀

در روش تقسیم و حل لازم است:

- نقطه‌ها را به دسته‌هایی تقسیم کنیم
- بین نقاط هر دسته فاصله‌ها را دوبه‌دو حساب کنیم تا مقدار کمینه پیدا شود.
- بررسی کنیم که آیا با مقایسه‌ی نقاط هر دسته با دسته‌های دیگر، فاصله‌ی کوچک‌تری به دست می‌آید یا خیر.

روش تقسیم و حل به شرطی می‌تواند مؤثر باشد که در مرحله‌ی سوم، ناچار به محاسبه‌ی تمامی فاصله‌ها نباشیم. بلکه باید از ابتدا دسته‌بندی به گونه‌ای انجام شود که هر دسته دارای تعدادی نقطه‌ی مرزی باشد و بتوان مرحله‌ی سوم را با محاسبه‌ی فاصله‌ی نقاط مرزی هر دسته با نقاط مرزی دسته‌های دیگر انجام داد. این صرفه‌جویی در محاسبه، باعث بهبود مرتبه‌ی زمانی می‌شود.

پیش‌پردازش 🛠️

ابتدا نقاط را بر اساس مؤلفه‌ی x آن‌ها مرتب می‌کنیم. مرتب کردنشان $O(n \log n)$ هزینه می‌برد. (در واقع داریم نقطه‌هایی که x آن‌ها به هم نزدیک‌تر است را در یک دسته قرار می‌دهیم.)

In [7]:
```python
# تابع مقایسه برای مرتب‌سازی نقاط بر اساس مؤلفه x
def compare_by_x(p):
    return p.x  # فقط مؤلفه x را برای مرتب‌سازی در نظر می‌گیریم
```

In [8]:
```python
# تابع پیش‌پردازش برای مرتب‌سازی نقاط بر اساس x
def preprocess(points):
    points.sort(key=compare_by_x)  # مرتب‌سازی لیست نقاط با استفاده از تابع مقایسه
```

مراحل حل: تلاش اول 📝

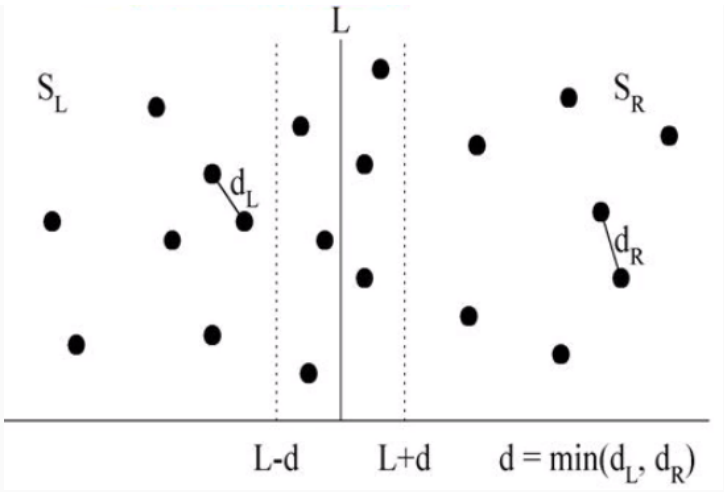
1. یک مقدار مرزی انتخاب می‌کنیم و نقطه‌هایی که x آن‌ها بیشتر از آن بود در یک ناحیه و سایر نقاط را در ناحیه‌ای دیگر قرار می‌دهیم. مثلا می‌توانیم x نقطه‌ی اُم $\frac{n}{2}$ را به عنوان مقدار مرزی انتخاب کنیم. معادله‌ی خط جداکننده‌ی ناحیه‌ها به شکل زیر می‌شود:

$$x = P[\frac{n}{2}] \to x$$

2. بر اساس نقطه‌ی مرزی، آرایه‌ی نقاط را به دو نیمه تقسیم می‌کنیم. طبیعتاً نصف نقاط در نیمه اول و نصف دیگر در نیمه‌ی دوم قرار می‌گیرند.

3. به طور بازگشتی مرحله‌ی 1 و 2 را روی هر کدام از ناحیه‌ها انجام می‌دهیم. اگر به مرحله‌ای رسیدیم که در یک ناحیه فقط دو یا سه نقطه داشتیم، فواصل دوبه‌دوی آن‌ها را حساب می‌کنیم و فاصله‌ی کمینه را به دست می‌آوریم.

4.



5. فرض کنید در یک مرحله، فاصله‌ی کمینه را برای نواحی سمت چپ و سمت راست یک نقطه‌ی مرزی (L) پیدا کرده‌ایم (d_L و d_R). بین این دو مقدار کمینه، مقدار کوچک‌تر را d می‌نامیم. این مقدار، حد بالایی برای پاسخ مسئله است.

6. حال، دو ناحیه را ادغام می‌کنیم. باید مطمئن شویم که هیچ دو نقطه‌ای پیدا نمی‌شوند که در نواحی متفاوت بوده و فاصله‌ی آن‌ها کمتر از d باشد. نقاطی که فاصله‌ی‌شان از خط وسط، بیشتر از d است ممکن نیست چنین شرایطی داشته باشند. تمام نقاطی که این ویژگی را ندارند، در آرایه‌ای به نام **border** می‌ریزیم.

7. با محاسبه‌ی فواصل دوبه‌دو بین نقاط موجود در **border**، کمترین مقدار را حساب می‌کنیم (d'). کمترین فاصله‌ی ممکن بین نقاط دو دسته $min(d, d')$ است.

In [9]:
```python
# تابع اصلی برای پیدا کردن کمترین فاصله بین نقاط با روش تقسیم و حل
def closest_pair(points):
    def recursive(P):
        n = len(P)
        if n <= 3:
            return brute_force(P)  # اگر تعداد نقاط کم باشد، از روش ساده استفاده می‌کنیم

        mid = n // 2
        mid_x = P[mid].x  # نقطه‌ی میانی برای تقسیم ناحیه‌ها x مقدار

        left = P[:mid]    # نقاط ناحیه‌ی چپ
        right = P[mid:]   # نقاط ناحیه‌ی راست

        # محاسبه فاصله کمینه در هر ناحیه به صورت بازگشتی
        d1 = recursive(left)
        d2 = recursive(right)
        d = min(d1, d2)  # کمترین فاصله بین دو ناحیه

        # است d ساخت لیست نقاط مرزی که فاصله‌شان از خط وسط کمتر از #
        border = []
        for p in P:
            if abs(p.x - mid_x) ** 2 < d:
                border.append(p)

        # برای بررسی مؤثرتر y مرتب‌سازی نقاط مرزی بر اساس #
        border.sort(key=lambda p: p.y)

        # بررسی فاصله بین نقاط مرزی برای یافتن فاصله‌ی بهتر #
        for i in range(len(border)):
            for j in range(i + 1, len(border)):
                if (border[j].y - border[i].y) ** 2 >= d:
                    break
                d = min(d, distance(border[i], border[j]))

        return d

    # مرتب می‌کنیم x ابتدا نقاط را بر اساس #
    preprocess(points)
    return recursive(points)
```

پیچیدگی زمانی ⏱️

ممکن است در هر مرحله تعداد زیادی از نقاط عضو آرایه‌ی **border** شوند. در بدترین حالت، همه‌ی نقاط عضو **border** می‌شوند و پیچیدگی زمانی الگوریتم در $O(n^2)$ خواهد شد.

مراحل حل: تلاش دوم 🔄

نقاط موجود در آرایه‌ی **border** را به ترتیب y مرتب می‌کنیم. با استفاده از هندسه اثبات می‌شود که در این صورت برای هر نقطه، محاسبه‌ی فاصله‌ی آن با 15 نقطه بعد از آن کافیست.

در کد زیر شرط گذاشته‌ایم که تنها در صورتی اختلاف دو نقطه‌ی مرزی را حساب کنیم که اختلاف yهایشان کمتر از d باشد. این شرط معادل این است که برای هر نقطه حداکثر فاصله‌اش با 15 نقطه بعد از آن را پیدا کنیم. در این صورت پیچیدگی زمانی الگوریتم از $O(n^2)$ به $O(n \log n) + O(15n)$ یا همان $O(n \log n)$ کاهش می‌یابد. برای اثبات می‌توانید اینجا را ببینید.

این قضیه را برای ۶ نقطه (از مجموعه‌ی دیگر) هم می‌توان اثبات کرد؛ یعنی با مرتب کردن نقاط به ترتیب y، به جای اینکه فاصله‌ی هر دو نقطه را پیدا کنیم، به ازای هر نقطه فاصله‌اش با ۶ نقطه‌ی دیگر از مجموعه‌ی دیگر را حساب می‌کنیم.

In [10]:
```python
# تعریف کلاس Point برای نمایش مختصات یک نقطه در صفحه
class Point:
    def __init__(self, x, y):
        self.x = x  # مختصات x نقطه
        self.y = y  # مختصات y نقطه
```

In [11]:
```python
# تابع محاسبه فاصله بین دو نقطه به روش اقلیدسی (بدون جذر برای مقایسه)
def distance(p1, p2):
    dx = p1.x - p2.x
    dy = p1.y - p2.y
    return dx * dx + dy * dy
```

In [12]:
```python
# تابع bruteForce برای پیدا کردن کمترین فاصله بین زوج نقاط
def brute_force(points):
    min_dist = float('inf')
    n = len(points)
    for i in range(n):
        for j in range(i + 1, n):
            d = distance(points[i], points[j])
            if d < min_dist:
                min_dist = d
    return min_dist
```

In [13]:
```python
# مرتب‌سازی نقاط بر اساس مؤلفه x
def preprocess(points):
    points.sort(key=lambda p: p.x)
```

In [14]:
```python
# بررسی نقاط مرزی برای یافتن فاصله‌ی بهتر با محدودیت هندسی
def check_border(border, d):
    min_dist = d
    border.sort(key=lambda p: p.y)  # مرتب‌سازی بر اساس y

    for i in range(len(border)):
        for j in range(i + 1, len(border)):
            if (border[j].y - border[i].y) ** 2 >= min_dist:
                break  # اگر اختلاف y بیشتر از min بیشتر نده، ادامه باشد
            dist = distance(border[i], border[j])
            if dist < min_dist:
                min_dist = dist
    return min_dist
```

In [15]:
```python
# تابع کمکی برای گرفتن مقدار کمتر بین دو عدد
def min_val(d1, d2):
    return d1 if d1 <= d2 else d2
```

In [16]:
```python
# الگوریتم تقسیم و حل برای پیدا کردن نزدیک‌ترین زوج نقاط
def divide_and_conquer(P):
    n = len(P)
    if n <= 3:
        return brute_force(P)

    middle = n // 2
    mid_point = P[middle]

    dl = divide_and_conquer(P[:middle])
    dr = divide_and_conquer(P[middle:])
    d = min_val(dl, dr)

    border = []
    for i in range(n):
        if abs(P[i].x - mid_point.x) ** 2 < d:
            border.append(P[i])

    return min_val(d, check_border(border, d))
```

In [17]:
```python
# تابع نهایی برای پیدا کردن فاصله‌ی نزدیک‌ترین زوج نقاط
def find_closest_pair(P):
    preprocess(P)
    return divide_and_conquer(P) ** 0.5  # جذر فاصله برای مقدار واقعی
```

In [18]:
```python
# تست الگوریتم با مجموعه‌ای از نقاط
p = [Point(1, 2), Point(3, 1), Point(4, 5), Point(3, 4)]
print(find_closest_pair(p[:3]))  # خروجی باید فاصله‌ی کمینه بین سه نقطه اول باشد
```

```
2.23606797749979
```

پیچیدگی زمانی ⏱️

فرض کنیم که پیچیدگی زمانی الگوریتم برای n داده، $T(n)$ باشد. و فرض کنیم که از یک الگوریتم مرتب‌سازی با پیچیدگی $O(n \log n)$ استفاده می‌کنیم (اینجا از Quick Sort استفاده کردیم که برای حجم داده‌های کمتر، عملکرد بهتری دارد. اما در تعداد داده‌های زیاد، این الگوریتم پیچیدگی زمانی $O(n^2)$ دارد و در آن شرایط بهتر است از Merge Sort استفاده کنیم). الگوریتم نقطه‌ها را به دو مجموعه تقسیم می‌کند و بعد، در زمان $O(n)$ آرایه‌ی **border** را درست می‌کند، آن را در زمان $O(n \log n)$ مرتب می‌کند و سپس نزدیک‌ترین زوج نقاط موجود در آرایه‌ی **border** را در زمان $O(n)$ پیدا می‌کند. بنابراین رابطه بازگشتی T به این صورت می‌شود:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n)$$

$$T(n) = O(n \times \log(n) \times \log(n))$$

حال اگر از اول یک کپی از آرایه بگیریم و نقاط آن را بر حسب y مرتب کنیم و در هر بار که نقاط را به دو مجموعه تقسیم می‌کنیم، اعضای دو مجموعه را از روی این مجموعه‌ی مرتب شده بر حسب y مرتب کنیم، زمان مرتب‌سازی از $O(n \log n)$ به $O(n)$ کاهش پیدا می‌کند و بنابراین رابطه بازگشتی بالا به این شکل تغییر می‌کند:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$T(n) = O(n \log n)$$

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل پنجم: روش تقسیم و حل

بخش دوم: «الگوریتم استراسن برای ضرب ماتریس‌ها» و «تبدیل سریع فوریه (FFT)»

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 📑

مروری بر روش تقسیم و حل 🔀

همان‌گونه که در جلسه‌ی گذشته مشاهده کردیم، گاهی اوقات تقسیم کردن یک مسئله و تبدیل آن به چند زیر مسئله‌ی کوچکتر مشابه می‌تواند بسیار راهگشا باشد. در این نوع از حل مسئله، تا وقتی که بتوانیم مسئله را به صورت مستقیم حل کنیم، آن را به قسمت‌های کوچکتر تقسیم می‌کنیم، سپس با ترکیب پاسخ‌ها، پاسخ مسئله‌ی اصلی را بازسازی می‌کنیم.

پس در الگوریتم‌هایی که با این روش ساخته می‌شوند سه مرحله وجود دارد:

- تقسیم کردن مسئله به چند مسئله‌ی کوچک‌تر
- حل کردن زیر مسئله‌ها
- ادغام پاسخ‌ها

در این دفترچه تعدادی دیگر از این الگوریتم‌ها که کاربردهای بسیاری نیز دارند، معرفی شداند.

بخش اول: الگوریتم استراسن برای ضرب دو ماتریس 🟩

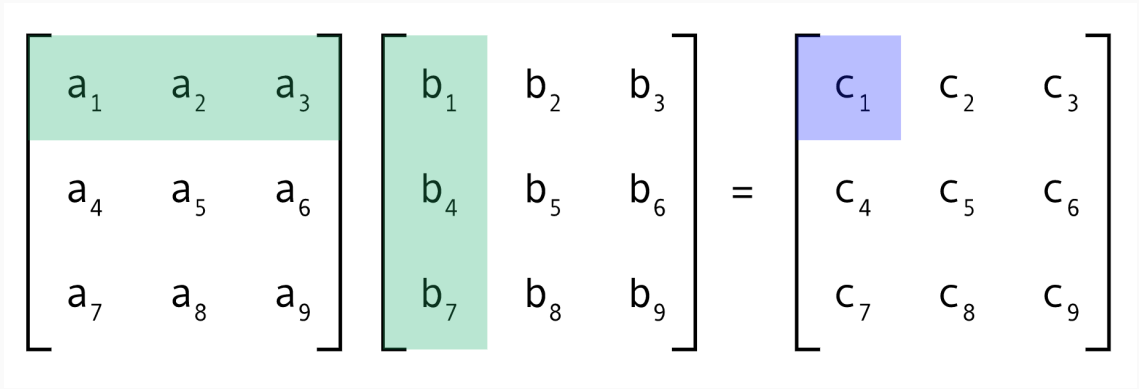
در بخش‌های قبلی درس بررسی کردیم که اگر بخواهیم چند ماتریس را در هم ضرب کنیم، چگونه بهترین توالی را برای این ضرب‌ها بدست بیاوریم تا سریعتر به جواب برسیم و به روش ضرب دو ماتریس کاری نداشتیم.

در این بخش چگونگی ضرب دو ماتریس را مورد بررسی قرار می‌دهیم و به دنبال راه‌حلی سریع و بهینه برای این کار هستیم.

فرض کنید دو ماتریس A_{n*n} و B_{n*n} را داریم و می‌خواهیم $C_{n*n} = AB$ را محاسبه کنیم.

راه‌حل اول (راه حل بدیهی)

طبق تعریف ضرب ماتریس‌ها عمل کرده و هر سطر از ماتریس A را در هر ستون از ماتریس B ضرب می‌کنیم.



پیچیدگی زمانی: برای محاسبه‌ی هر درایه از ماتریس C باید ضرب یک سطر در یک ستون انجام شود که از مرتبه زمانی $O(n)$ زمان می‌برد. و چون ماتریس C دارای n^2 درایه است، انجام ضرب با این روش در مرتبه‌ی زمانی $O(n^3)$ انجام می‌شود.

In [1]:
```python
# تابع ضرب معمولی دو ماتریس مربعی با اندازه مشخص
def regular_multiple(a, b, size):
    # ایجاد ماتریس خروجی با مقدار اولیه صفر
    c = [[0 for _ in range(size)] for _ in range(size)]

    for i in range(size):
        for j in range(size):
            for k in range(size):
                c[i][j] += a[i][k] * b[k][j]  # ضرب سطر i از a در ستون j از b
    return c
```

In [2]:
```python
# تابع چاپ ماتریس به صورت مرتب
def print_matrix(A, size):
    for i in range(size):
        for j in range(size):
            print(A[i][j], end=" ")  # چاپ درایه‌های هر سطر
        print()  # رفتن به خط بعدی پس از پایان سطر
```

In [3]:
```python
# تعریف ماتریس اول
A = [[1, 2], [4, 5]]
print("first matrix is:")
print_matrix(A, 2)
print()

# تعریف ماتریس دوم
B = [[1, 2], [1, 2]]
print("second matrix is:")
print_matrix(B, 2)
print()

# محاسبه حاصل ضرب دو ماتریس
C = regular_multiple(A, B, 2)

# چاپ نتیجه
print("the answer is:")
print_matrix(C, 2)
```

```
first matrix is:
1 2
4 5

second matrix is:
1 2
1 2

the answer is:
3 6
9 18
```

راه‌حل دوم (الگوریتم اشتراسن)

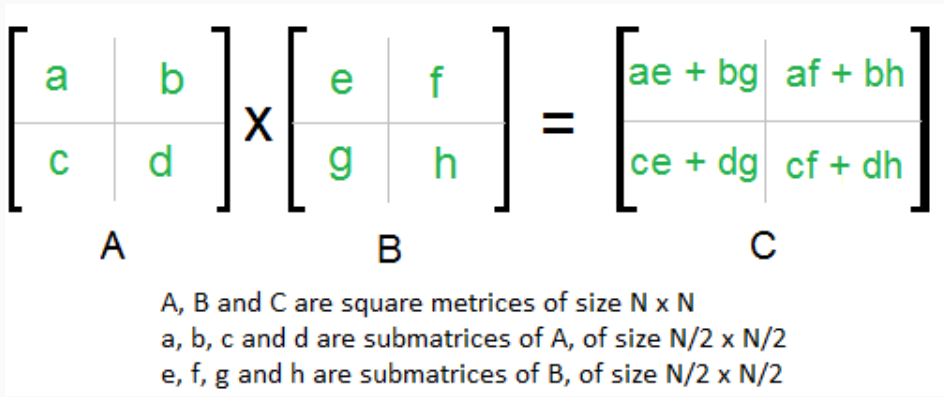
ایده‌ی اصلی این الگوریتم تقسیم و حل است. دو ماتریس با ابعاد $n \times n$ و A و B داده شده است. با فرض اینکه عدد n به توانی از عدد 2 است $(n = 2^k, k \in \mathbb{N})$. هر کدام از دو ماتریس A و B را به 4 زیرماتریس تقسیم می‌کنیم که هر کدام در ابعاد $\frac{n}{2} \times \frac{n}{2}$ هستند. (هر کدام از A_{ij} ها و B_{ij} ها ماتریس هستند.)

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

الگوریتم استراسن: محاسبه زیرماتریس‌ها ⚙️

به راحتی می‌توان اثبات کرد:

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$



در اینجا در هر مرحله باید 8 ضرب ماتریس‌های $\frac{n}{2} \times \frac{n}{2}$ را حساب کنیم، یعنی:

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$$

که با محاسبه با قضیه اصلی بدست می‌آید: $T(n) \in O(n^3)$

می‌بینیم که زمان اجرای ضرب هنوز بهبود پیدا نکرده است. ایده‌ی اشتراسن برای محاسبه‌ی ضرب ماتریس‌ها این بود که به‌جای محاسبه‌ی این 8 عبارت، 7 عبارت جدید را محاسبه کند. این 7 عبارت باید به صورتی باشند که بتوان به راحتی با استفاده از آن‌ها حاصل زیرماتریس‌های C را محاسبه کرد.

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = M_1 + M_4 + M_7 - M_5$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 + M_3 + M_6 - M_2$$

نکته: در ابتدای این روش فرض کردیم n به صورت توانی از 2 است. اگر در هر مرحله‌ی تقسیم ماتریس، ابعاد ماتریس توانی از 2 نباشد، با افزودن سطرها و ستون‌هایی که همه‌ی درایه‌های آن‌ها صفر است به ماتریس‌ها، آن‌ها را بصورت توانی از 2 در می‌آوریم.

In [4]:
```python
# جمع یا تفریق دو ماتریس با اندازه مشخص
def add_and_sub_matrix(A, B, flag):
    size = len(A)
    C = [[0 for _ in range(size)] for _ in range(size)]
    for i in range(size):
        for j in range(size):
            C[i][j] = A[i][j] + flag * B[i][j]  # اگر flag برابر ۱ باشد جمع، اگر -۱ باشد تفریق
    return C
```

In [5]:
```python
# الگوریتم استراسن برای ضرب دو ماتریس مربعی با اندازه توانی از ۲
def strassen(A, B):
    size = len(A)
    if size == 2:
        return regular_multiple(A, B, size)  # اگر اندازه ۲ باشد، ضرب معمولی انجام می‌دهیم

    new_size = size // 2

    # تقسیم ماتریس‌ها به چهار زیرماتریس
    A11 = [[A[i][j] for j in range(new_size)] for i in range(new_size)]
    A12 = [[A[i][j + new_size] for j in range(new_size)] for i in range(new_size)]
    A21 = [[A[i + new_size][j] for j in range(new_size)] for i in range(new_size)]
    A22 = [[A[i + new_size][j + new_size] for j in range(new_size)] for i in range(new_size)]

    B11 = [[B[i][j] for j in range(new_size)] for i in range(new_size)]
    B12 = [[B[i][j + new_size] for j in range(new_size)] for i in range(new_size)]
    B21 = [[B[i + new_size][j] for j in range(new_size)] for i in range(new_size)]
    B22 = [[B[i + new_size][j + new_size] for j in range(new_size)] for i in range(new_size)]

    # محاسبه هفت ضرب اصلی الگوریتم استراسن
    M1 = strassen(add_and_sub_matrix(A11, A22, 1), add_and_sub_matrix(B11, B22, 1))
    M2 = strassen(add_and_sub_matrix(A21, A22, 1), B11)
    M3 = strassen(A11, add_and_sub_matrix(B12, B22, -1))
    M4 = strassen(A22, add_and_sub_matrix(B21, B11, -1))
    M5 = strassen(add_and_sub_matrix(A11, A12, 1), B22)
    M6 = strassen(add_and_sub_matrix(A21, A11, -1), add_and_sub_matrix(B11, B12, 1))
    M7 = strassen(add_and_sub_matrix(A12, A22, -1), add_and_sub_matrix(B21, B22, 1))

    # ترکیب نتایج برای ساخت ماتریس نهایی
    C11 = add_and_sub_matrix(add_and_sub_matrix(M1, M4, 1), add_and_sub_matrix(M7, M5, -1), 1)
    C12 = add_and_sub_matrix(M3, M5, 1)
    C21 = add_and_sub_matrix(M2, M4, 1)
    C22 = add_and_sub_matrix(add_and_sub_matrix(M1, M3, 1), add_and_sub_matrix(M6, M2, -1), 1)

    # ساخت ماتریس خروجی
    C = [[0 for _ in range(size)] for _ in range(size)]
    for i in range(new_size):
        for j in range(new_size):
            C[i][j] = C11[i][j]
            C[i][j + new_size] = C12[i][j]
            C[i + new_size][j] = C21[i][j]
            C[i + new_size][j + new_size] = C22[i][j]
    return C
```

In [6]:
```python
# تعریف ماتریس اول
size = 8
A = [[i + 2 * j for j in range(size)] for i in range(size)]
print("first matrix is:")
print_matrix(A, size)
print()

# تعریف ماتریس دوم
B = [[i + j for j in range(size)] for i in range(size)]
print("second matrix is:")
print_matrix(B, size)
print()

# محاسبه حاصل ضرب با الگوریتم استراسن
C = strassen(A, B)

# چاپ نتیجه
print("the answer is:")
print_matrix(C, size)
print()
```

```
first matrix is:
0 2 4 6 8 10 12 14
1 3 5 7 9 11 13 15
2 4 6 8 10 12 14 16
3 5 7 9 11 13 15 17
4 6 8 10 12 14 16 18
5 7 9 11 13 15 17 19
6 8 10 12 14 16 18 20
7 9 11 13 15 17 19 21

second matrix is:
0 1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
2 3 4 5 6 7 8 9
3 4 5 6 7 8 9 10
4 5 6 7 8 9 10 11
5 6 7 8 9 10 11 12
6 7 8 9 10 11 12 13
7 8 9 10 11 12 13 14

the answer is:
280 336 392 448 504 560 616 672
308 372 436 500 564 628 692 756
336 408 480 552 624 696 768 840
364 444 524 604 684 764 844 924
392 480 568 656 744 832 920 1008
420 516 612 708 804 900 996 1092
448 552 656 760 864 968 1072 1176
476 588 700 812 924 1036 1148 1260
```

⏱️ تحلیل زمانی الگوریتم استراسن

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

که طبق قضیه اصلی:

$$T(n) \in O(n^{\lg 7}) \approx O(n^{2.81})$$

می‌بینیم که زمان بهتر شد.

پرسش: آیا اگر بجای تقسیم ماتریس‌ها به 4 زیرماتریس، آنها را به 9 زیرماتریس تقسیم کنیم زمان بهبود پیدا می‌کند؟ 16 زیرماتریس چطور؟ آیا همیشه با افزایش تعداد زیرماتریس‌ها زمان بهبود می‌یابد؟

بخش دوم: FFT (تبدیل سریع فوریه)

به عنوان مثالی دیگر از روش تقسیم و حل، می‌خواهیم به سراغ مسئله معروف و پرکاربردی برویم که شاید قبلا اسم آن را شنیده باشید: تبدیل فوریه! 📊

البته که کل گستره مبحث تبدیل فوریه خارج از حیطه این درس است، اما می‌خواهیم به نمونه خاصی از حالت گسسته این مسئله بپردازیم و برای ملموس بودن این مثال از یک مسئله آشنا استفاده می‌کنیم: ضرب چندجمله‌ای‌ها.

تعریف مسئله 📝

فرض کنید $p(x)$ و $q(x)$ دو چندجمله‌ای با درجه n باشند. می‌خواهیم چندجمله‌ای $r(x) = p(x) \cdot q(x)$ را محاسبه کنیم.

تلاش اول

ساده‌ترین روش شاید همان روشی باشد که ضرب دو چندجمله‌ای تعریف می‌شود. اگر دو چندجمله‌ای را به شکل

$$p(x) = p_n x^n + p_{n-1} x^{n-1} + \ldots + p_1 x + p_0$$

و

$$q(x) = q_n x^n + q_{n-1} x^{n-1} + \ldots + q_1 x + q_0$$

داشته باشیم، در این صورت ضرب این دو با n^2 عمل ضرب و حدودا n^2 عمل جمع قابل انجام است.

تمرین:

ضرب دو چند جمله‌ای را در قسمت زیر پیاده‌سازی کنید.

In [9]:
```python
# answered

# به روش ساده n ضرب دو چندجمله‌ای با درجه
def poly_mult(p, q, n):
    r = [0] * (2 * n - 1)  # ۲ لیست ضرایب حاصل ضرب با اندازه n-۱

    for i in range(n):
        for j in range(n):
            r[i + j] += p[i] * q[j]  # ضرب ضریب‌های متناظر و جمع آن‌ها در درجه مناسب
    return r
```

In [10]:
```python
# تعریف چندجمله‌ای‌ها و تست تابع ضرب
n = 3
poly_p = [1, 2, 3]  # p(x) = 1 + 2x + 3x^2
poly_q = [4, 5, 6]  # q(x) = 4 + 5x + 6x^2

poly_r = poly_mult(poly_p, poly_q, n)

# چاپ ضرایب حاصل ضرب
print("Resulting polynomial coefficients:")
print(poly_r)
```

```
Resulting polynomial coefficients:
[4, 13, 28, 27, 18]
```

تمرین

تعداد دقیق ضرب و جمع‌های روش اول برای محاسبه ضرب دو چندجمله‌ای را به دست آورید و نشان دهید که پیچیدگی زمانی ضرب دو چندجمله‌ای به این روش از $O(n^2)$ است.

حال می‌خواهیم ببینیم چطور می‌توانیم بهتر از این عمل کنیم. برای این کار باید اول نمایشمان از چندجمله‌ای را بازنگری کنیم.

تلاش دوم

نمایش یک چندجمله‌ای با ضرایبش را می‌شناسیم، طوری که چندجمله‌ای

$$p(x) = p_n x^n + p_{n-1} x^{n-1} + \ldots + p_1 x + p_0$$

با ضرایبش به صورت یکتا مشخص می‌شود. اما راه دیگری هم برای نمایش یک چندجمله‌ای وجود دارد. به عنوان مثال، یک خط با هر دو نقطه روی آن به صورت یکتا تعیین می‌شود. یک چندجمله‌ای درجه دو با سه نقطه روی آن به صورت یکتا مشخص می‌شود. به طور کلی هم قضیه زیر برقرار است:

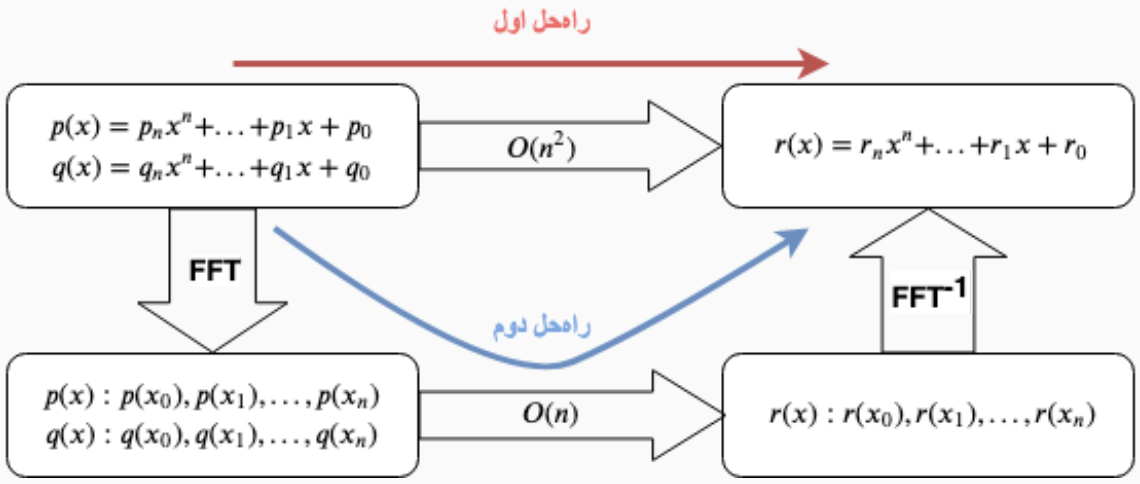
قضیه:

یک چندجمله‌ای درجه n (با ضرایب مختلط) با $n+1$ نقطه روی آن به صورت یکتا مشخص می‌شود. برقرار بودن قضیه بالا برای اعداد مختلط نه تنها از نظر ریاضی درست است، بلکه در ادامه برای حل مسئله مورد نظرمان لازم هم خواهد بود.

با در دست داشتن قضیه بالا، می‌توان نمایش دیگری برای یک چندجمله‌ای پیدا کرد به این صورت که به جای آنکه چندجمله‌ای را با ضرایبش مشخص کنیم، با تعدادی نقطه روی آن (یعنی تعدادی $p(x_i)$ برای x_i های مختلف) مشخص می‌کنیم. انجام ضرب با این نمایش چندجمله‌ای بسیار ساده است، چون با داشتن $p(x_i)$ و $q(x_i)$ می‌توان مقدار $p(x_i)q(x_i)$ را که متعلق به حاصل‌ضرب دو چندجمله‌ای است، محاسبه کرد.

نکته:

چندجمله‌ای حاصلضرب یک چندجمله‌ای از درجه $2n$ است و باید حاصلضرب $2n$ نقطه را به دست آورد. خب طبق آنچه در بالا گفته شد، با داشتن تعداد کافی نقطه از دو چندجمله‌ای، محاسبه حاصل‌ضرب‌شان با $2n$ ضرب قابل انجام است. اما ما معمولا نمایش عادی یک چندجمله‌ای را داریم، پس سوال این است که هزینه تبدیل چندجمله‌ای به این نمایش چقدر است؟



شکل نشان می‌دهد که روش جدیدی برای محاسبه حاصل‌ضرب پیدا کرده‌ایم اما اگر تبدیل را به صورت بهینه‌ای انجام ندهیم، بهبودی حاصل نشده و پیچیدگی ضرب همچنان $O(n^2)$ است. اگر بتوانیم راهی سریع‌تر برای درونیابی و برونیابی چندجمله‌ای پیدا کنیم، می‌توانیم ضرب را سریع‌تر از حالت معمول انجام دهیم. این تبدیل نمایش ضرایب به نقاط همان **تبدیل فوریه** نام دارد.

تبدیل فوریه

طبق آنچه در بالا گفتیم، صورت مسئله ساده شده به این ترتیب است: با داشتن نمایش یک چندجمله‌ای درجه n با ضرایب به شکل

$$p(x) = p_{n-1}x^{n-1} + p_{n-2}x^{n-2} + \ldots + p_1 x + p_0$$

مقدار این چندجمله‌ای را در n نقطه به دست آورید. شاید به نظر بیاید جایی برای بهبود سرعت این محاسبات نداشته باشیم، اما مقادیر x_i را می‌توانیم تعیین کنیم. پس سعی می‌کنیم مقادیری انتخاب کنیم که بتوان محاسبات را سریع‌تر انجام داد.

از آنجایی که نمی‌توانیم x_i یکسان انتخاب کنیم، ساده‌ترین جایگزین $x_i = -x_j$ به نظر می‌رسد که شاید بخشی از محاسبات را کم کند. (فرض کنید بدون کاهش کلیت مسئله n فرد باشد.) چندجمله‌ای را به صورت زیر نشان می‌دهیم:

$$p(x) = p_n x^n + p_{n-1}x^{n-1} + \ldots + p_1 x + p_0$$
$$= (p_n x^{2k+1} + p_{n-2}x^{2k-1} + \ldots + p_1 x) + (p_{n-1}x^{2k} + p_{n-3}x^{2k-2} + \ldots + p_0)$$
$$= x \cdot P_0(x^2) + P_1(x^2)$$

که در فرمول بالا:

$$P_0(x) = p_n x^k + p_{n-2}x^{k-1} + \ldots + p_1, \quad P_1(x) = p_{n-1}x^k + \ldots + p_0$$

تمرین:

درستی روابط بالا را اثبات کنید. آنچه از روابط بالا باید دریافت این است که برای x_i و $x_j = -x_i$ داریم:

$$P_0(x_i^2) = P_0(x_j^2), \quad P_1(x_i^2) = P_1(x_j^2)$$

اگر بخواهیم دقیق‌تر بگوییم، فرض کنید $x_0, x_1, \ldots, x_{n-1}$ طوری باشند که نیمه اول قرینه نیمه دوم باشد ($x_{i+n/2} = -x_i$). پس محاسبه $p(x)$ در این n نقطه برابر است با محاسبه $p_0(x)$ و $p_1(x)$ در $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ به انضمام چند عملیات اضافه که تعدادشان از $O(n)$ است. اما نیمه اول قرینه نیمه دوم است! پس کافیست $p_0(x)$ و $p_1(x)$ را فقط در نصف نقاط حساب کنیم و اینگونه به دو زیر مسئله با اندازه کوچک‌تر دست پیدا می‌کنیم. (چون $p_0(x)$ و $p_1(x)$ از درجه $n/2$ هستند و می‌خواهیم در $n/2$ نقطه مقدارشان را پیدا کنیم.)

به نظر می‌آید مسئله به طور کامل به دو زیرمسئله شکسته شده، تنها چیزی که مانده، این است که مقادیر x_i ها را تعیین کنیم. در مرحله اول n عدد داشتیم که نیمه اول مثبت بود و نیمه دوم منفی. اما در مرحله دوم چه؟ مگر همه اعداد به شکل x_i^2

نیستند؟ پس چطور می‌توان نصف را مثبت و نصف دیگر را منفی انتخاب کرد؟ جواب این سوال را بالاتر به صورت ضمنی داده‌ایم: استفاده از اعداد مختلط! اعدادی که شرایط مورد نظر برای حل مسئله را دارند، ریشه‌های n ام واحد هستند.

به صورت خلاصه ریشه‌های n ام واحد اعدادی هستند که در معادله

$$\omega^n - 1 = 0$$

صدق می‌کنند. توضیحات درباره خواص اعداد مختلط و اینکه چرا این اعداد ویژگی‌های مورد نظر ما را دارند این دفترچه را طولانی خواهد کرد. برای همین فقط الگوریتم بالا را برای x_i های مناسب توضیح می‌دهیم. اگر ω برابر ریشه n ام واحد باشد، آنوقت $x_i = \omega^i$ قرار می‌دهیم. آنوقت می‌خواهیم مقدار $p(x)$ را در $\omega^0, \omega^1, \dots, \omega^{n-1}$ حساب کنیم. برای این کار با استفاده از معادله

$$p(x) = p_0(x^2) + xp_1(x^2)$$

ابتدا مقدار $p_0(x)$ و $p_1(x)$ را در $(\omega^0)^2, (\omega^1)^2, \dots, (\omega^{n-1})^2$ حساب می‌کنیم. اما با کمی محاسبه می‌توان دید که این‌ها درواقع فقط $n/2$ عدد متمایز یا همان ریشه‌های $n/2$ ام واحد هستند.

نکته:

ممکن است فکر کنید که پیاده‌سازی اعداد مختلط چگونه خواهد بود. اما هر عدد مختلط به صورت یک زوج مرتب قابل نمایش است و ضرب و جمع اعداد مختلط هم به راحتی با همان نمایش با زوج مرتب قابل انجام است. در زبان C این نوع داده پیاده‌سازی شده است و به صورت زیر قابل استفاده است.

پیاده‌سازی FFT

با کمک این نوع داده می‌توان تبدیل فوریه را پیاده‌سازی کرد.

In [11]:
```python
import math

# تعریف کلاس عدد مختلط با عملیات جمع و ضرب
class Complex:
    def __init__(self, re, im):
        self.re = re  # بخش حقیقی
        self.im = im  # بخش موهومی

    def __add__(self, other):
        return Complex(self.re + other.re, self.im + other.im)  # جمع دو عدد مختلط

    def __sub__(self, other):
        return Complex(self.re - other.re, self.im - other.im)  # تفریق دو عدد مختلط

    def __mul__(self, other):
        # ضرب دو عدد مختلط با استفاده از فرمول (a+bi)(c+di)
        re = self.re * other.re - self.im * other.im
        im = self.re * other.im + self.im * other.re
        return Complex(re, im)

    def __repr__(self):
        return f"{self.re:.2f} + {self.im:.2f}i"  # نمایش عدد مختلط به صورت خوانا
```

In [12]:
```python
# پیاده‌سازی بازگشتی FFT برای چندجمله‌ای با ضرایب مختلط
def fft(p):
    n = len(p)
    if n == 1:
        return [p[0]]  # پایه بازگشت: فقط یک ضریب داریم

    # محاسبه ریشه‌های n ام واحد
    w = []
    for i in range(n):
        alpha = 2 * math.pi * i / n
        w.append(Complex(math.cos(alpha), math.sin(alpha)))

    # تقسیم ضرایب به ضرایب با اندیس زوج و فرد
    p0 = [p[i * 2] for i in range(n // 2)]  # زیرمسئله اول
    p1 = [p[i * 2 + 1] for i in range(n // 2)]  # زیرمسئله دوم

    # فراخوانی بازگشتی برای هر زیرمسئله
    y0 = fft(p0)
```

```
    y1 = fft(p1)

    # ترکیب نتایج برای ساخت خروجی نهایی
    y = [None] * n
    for k in range(n // 2):
        y[k] = y0[k] + w[k] * y1[k]
        y[k + n // 2] = y0[k] - w[k] * y1[k]
    return y
```

In [13]:
```
# تعریف چندجمله‌ای p(x) = 1 + 2x + 3x^2
poly_p = [Complex(1, 0), Complex(2, 0), Complex(3, 0), Complex(0, 0)]  # صفر اضافه برای رسیدن به توان ۲

# اجرای FFT روی چندجمله‌ای
fft_p = fft(poly_p)

# چاپ خروجی تبدیل فوریه
for i in range(len(fft_p)):
    print(f"y[{i}] =", fft_p[i])
```

```
y[0] = 6.00 + 0.00i
y[1] = -2.00 + 2.00i
y[2] = 2.00 + 0.00i
y[3] = -2.00 + -2.00i
```

نکته تمرینی 📊

لازم به ذکر است که عکس تبدیل فوریه نیز به صورت مشابه و با یک تئوری مشابه قابل انجام است که پیاده‌سازی آن به عنوان تمرین به خواننده واگذار می‌شود.

تحلیل زمانی ⏱️

ابتدا پیچیدگی زمانی تبدیل فوریه را به شرح زیر می‌توانیم حساب کنیم:

$$T(n) = 2T(n/2) + O(n)$$

که طبق قضیه اصلی می‌دانیم که جواب برابر خواهد بود با:

$$T(n) = O(n \log n)$$

به عنوان تمرین هم می‌توانید پیچیدگی زمانی ضرب دو چند جمله‌ای را که متشکل از یک تبدیل فوریه و یک تبدیل عکس فوریه است، حساب کنید.

منابع 📚

کدهای استفاده شده در این بخش با تغییر از این سایت برداشته شده است.

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل ششم: تطابق رشته

بخش اول: اثر انگشت و الگوریتم رابین-کارپ

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

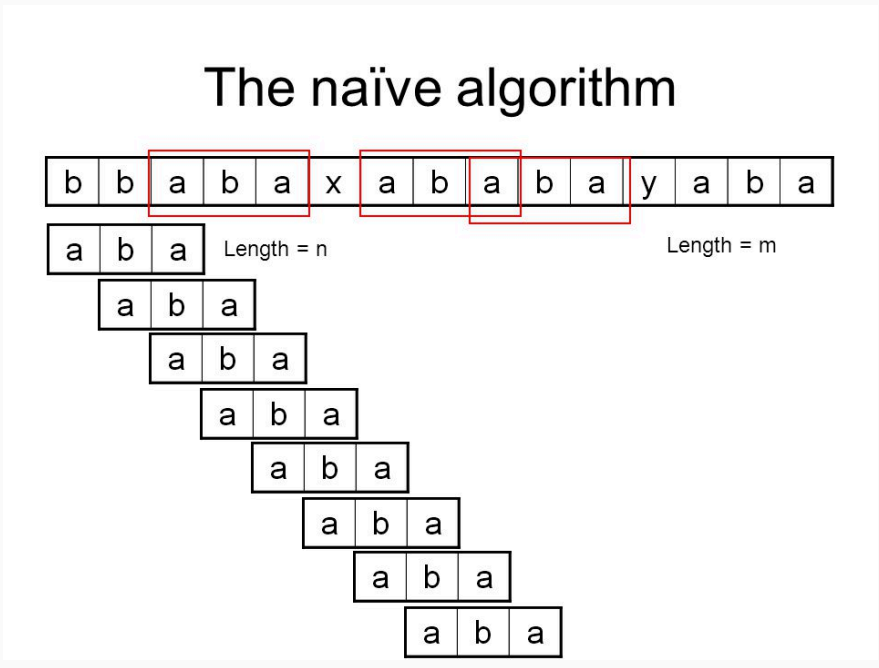
🔍 الگوریتم رابین-کارپ

مسالهٔ اصلی در این درس‌نامه پیدا کردن یک الگو در یک رشته است؛ این که در ورودی دو رشتهٔ `s` و `t` داریم که $len(t) >> len(s)$. و می‌خواهیم بدانیم که آیا رشتهٔ `s` در `t` تکرار شده است یا نه. روش‌های مختلفی برای این کار ارائه شده و در ادامه به روش رابین-کارپ که روشی مبتنی بر هش است خواهیم پرداخت.

راه حل اولیه

ابتدا برای متمرکز شدن بر روی هدف اصلی الگوریتم سعی می‌کنیم تا مساله را با راحت‌ترین روشی که به ذهن می‌رسد حل کنیم.
سپس مشکلات این روش را بررسی کرده و به روش رابین-کارپ برسیم.
روشی که به صورت خیلی ساده مساله را حل می‌کند چک کردن هر زیررشتهٔ `t` به صورت حرف به حرف با حروف رشتهٔ `s` است.
در صورتی که بخواهیم شبه‌کد این الگوریتم را بنویسیم به کد زیر می‌رسیم. هم‌چنین تصویر زیر این الگوریتم را شرح می‌دهد.

🖼️ الگوریتم brute force



In [1]:
```python
# الگوریتم ساده برای بررسی وجود رشته s در رشته t
def simple_algorithm(t, s):
    for i in range(len(t) - len(s) + 1):
        # بررسی زیررشته‌ای از t با طول برابر s
```

```
    if t[i:i + len(s)] == s:
        return True  # اگر برابر بود، رشته پیدا شده
return False  # در غیر این صورت، رشته وجود ندارد
```

تحلیل زمانی الگوریتم ساده ⏱️

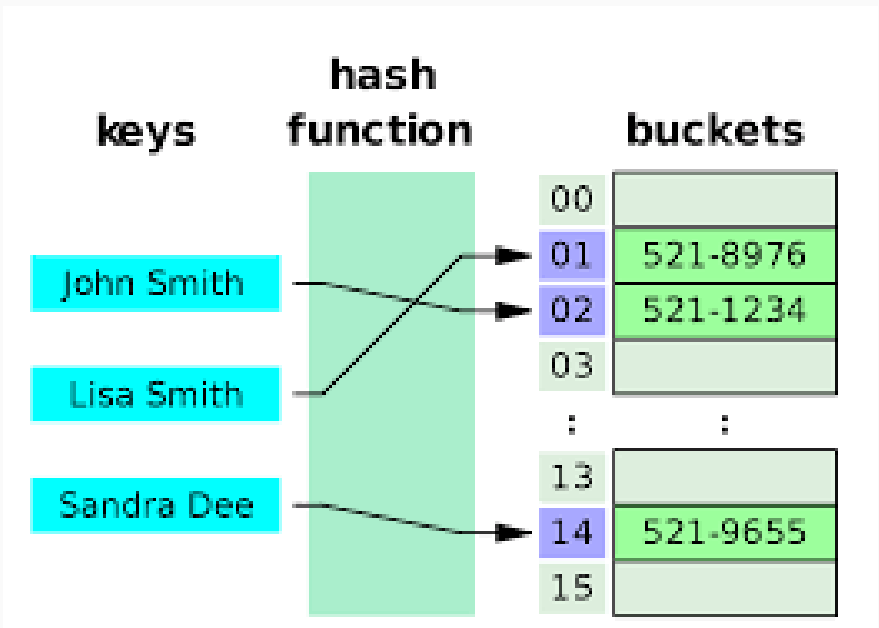
برای تحلیل زمانی این الگوریتم باید توجه کنیم که به اندازهٔ $len(t-s)$ بار دو زیر رشته با هم مقایسه می‌شوند و هر مقایسه حرف به حرف $len(s)$ هزینه دارد. پس داریم:

$$T(simple\_algorithm) = \theta(\mid s \mid * (\mid t \mid - \mid s \mid))$$

بدیهتا در صورتی که طول رشتهٔ الگو زیاد باشد این الگوریتم بسیار نابهینه عمل می‌کند و می‌تواند از $\theta(n^2)$ باشد. همچنین می‌توانیم حد پایین $\theta(\mid s \mid + \mid t \mid)$ را برای حل این مساله متصور شویم چون حداقل یک‌بار باید دو رشته را از ورودی بخوانیم. پس بهترین الگوریتمی که می‌تواند داشته باشیم در زمان خطی نسبت به s و t مساله را حل می‌کند. روش‌های مختلفی برای حل این مساله در این زمان وجود دارند و روش رابین-کارپ نیز در average-case یکی از آن‌هاست. در ادامه این روش را بررسی خواهیم کرد.

رابین-کارپ 🧩

می‌دانیم که هر رشته متنی را می‌توان با الگوریتم‌های مختلف درهم‌سازی (هش) به یک عدد نظیر کرد. یکی از راه‌حل‌هایی که می‌توانیم برای تطابق رشته داشته باشیم این است که به جای این که دو رشته را به صورت حرف به حرف با هم مقایسه کنیم، مقدار هش آن‌ها را با هم مقایسه کنیم. در صورتی که به هر شکلی (مثلا پیش‌پردازش) در زمان ثابت به مقدار هش هر زیررشته دسترسی داشته باشیم، می‌توانیم در زمان ثابت زیررشتهٔ انتخابی از `t` را با رشتهٔ `s` مقایسه کنیم و در این صورت مساله را می‌توانیم در زمان خطی که مطلوب ماست حل کنیم.



عملکرد تابع‌های درهم‌سازی به صورت انتزاعی

داده‌ساختار فرضی را در نظر بگیرید که بتواند در O(1) مقدار هش یک رشته را بدهد، حرفی به انتهای آن رشته اضافه کند و هش رشتهٔ جدید را محاسبه کند یا این که حرف اول رشته را حذف کرده و مقدار هش جدید را نیز محاسبه کند. در صورتی که چنین داده‌ساختاری داشته باشیم، می‌توانیم مساله تطابق رشته را با شبه‌کد زیر حل کنیم.

In [2]:
```
# تابع هش برای رشته با استفاده از پایه ۳۱
def hash_function(s):
    hash_val = 0
    P = 1
    for ch in s:
        hash_val += ord(ch) * P  # تبدیل کاراکتر به عدد و ضرب در ضریب پایه
        P *= 31  # به‌روزرسانی ضریب پایه برای موقعیت بعدی
    return hash_val
```

In [3]:
```
# کلاس RollingHash برای مدیریت هش متحرک رشته‌ها
class RollingHash:
    def __init__(self, length):
        self.hash_val = 0  # مقدار هش فعلی
        self.base = 31  # پایه درهم‌سازی
        self.length = length  # طول الگو
        self.P = [1] * (length + 1)  # پیش‌محاسبه توان‌های پایه
```

```
        for i in range(1, length + 1):
            self.P[i] = self.P[i - 1] * self.base
        self.window = []  # لیست کاراکترهای فعلی در پنجره

    def append(self, ch):
        self.window.append(ch)
        self.hash_val += ord(ch) * self.P[len(self.window) - 1]  # افزودن کاراکتر جدید به هش

    def skip(self, ch):
        self.hash_val -= ord(ch)  # حذف اثر کاراکتر اول از هش
        self.window.pop(0)  # حذف کاراکتر از پنجره
        # به‌روزرسانی هش با تقسیم بر پایه برای جابجایی موقعیت‌ها
        self.hash_val //= self.base

    def get_hash(self):
        return self.hash_val
```

In [4]:
```
# الگوریتم رابین-کارپ برای بررسی وجود رشته s در رشته t
def rabin_karp(t, s):
    if len(s) > len(t):
        return False

    rs = RollingHash(len(s))
    rt = RollingHash(len(s))

    for ch in s:
        rs.append(ch)  # محاسبه هش الگو

    for ch in t[:len(s)]:
        rt.append(ch)  # محاسبه هش اولین زیررشته از t

    if rs.get_hash() == rt.get_hash():
        return True  # تطابق در ابتدای رشته

    for i in range(len(s), len(t)):
        rt.skip(t[i - len(s)])  # حذف کاراکتر قدیمی
        rt.append(t[i])         # افزودن کاراکتر جدید
        if rs.get_hash() == rt.get_hash():
            return True  # تطابق پیدا شد

    return False  # تطابقی وجود ندارد
```

توضیح تکمیلی الگوریتم رابین-کارپ ✅

الگوریتم فوق در صورتی که هیچ‌گاه مقدار هش دو رشتهٔ متفاوت هیچ‌گاه دو مقدار یکسان نشوند درست عمل می‌کند، اما می‌توانیم برای مطمئن شدن از درستی الگوریتم، وقتی مقدار هش `rs()` و `rt()` برابر شدند، حرف به حرف دو رشته را چک کنیم. این الگوریتم در زمان خطی نسبت به ورودی‌ها مسالهٔ ما را حل می‌کند. کلیت الگوریتم رابین-کارپ نیز به همین شکل کار می‌کند، ولی مساله نهایی این است که چطور می‌توان داده‌ساختار فرض شده را پیاده‌سازی کرد.

پیاده‌سازی 💻

اولین سوال در این قسمت این است که چطور یک رشتهٔ متنی را به یک عدد تبدیل کنیم تا پس از آن با یک تابع درهم‌سازی آن را هش کنیم. شاید مجموع اعداد اسکی حروف رشته به ذهن خطور کند اما باید توجه داشت که این نمایش رشته به صورت عدد ترتیب کاراکترها را در نظر نمی گیرد و غلط است. در نمایش دیگری می‌توانیم هر کاراکتر را یک عدد در مبنای کل حروف الفبا در نظر بگیریم. این نظیرسازی یک‌به‌یک و برگشت‌پذیر است و انتخابی ایده‌آل برای تبدیل رشته به عدد محسوب می شود، پس از این پس هر رشته را یک عدد در مبنای تعداد حروف الفبا می‌بینیم.

حال باید این عدد متناظر با رشته را هش کنیم؛ یکی از ساده‌ترین تابع‌های درهم‌سازی تابع گرفتن باقی‌مانده بر یک عدد اول بزرگ است؛ برای یادآوری از مبحث هش، هر تابع درهم سازی تابعی‌ست که یک عدد ورودی که در فضای بزرگ N قرار دارد را ورودی می‌گیرد و آن را به یک عدد در فضای کوچک‌تر M نظیر می‌کند، N.

می‌دانیم که می‌توانیم باقی‌ماندهٔ یک عدد را بر دیگری در زمان ثابت به دست بیاوریم، پس اضافه کردن دو قابلیت اصلی حذف حرف c از اول رشته و اضافه کردن حرف c به انتهای رشته برای پیاده‌سازی الگوریتم بالا می‌ماند. اگر دوباره به تعبیر عددی خود از یک رشتهٔ متنی برگردیم، می‌بینیم که اضافه کردن یک عدد به انتهای یک رشته معادل با یک بار شیفت چپ تمامی رقم‌های قبلی و جمع کردن عدد جدید با عدد جدید است، که معادل با ضرب عدد فعلی در مبنا و جمع با عدد جدید است. اگر مقدار قبلی هش را

u و عدد مبنا را a و عدد حرف جدید در مبنای a را c در نظر بگیریم داریم.

هش و عملیات روی رشته

append(c):

$$u = u.\,a + c$$

همچنین می‌دانیم که مقدار باقی‌ماندهٔ عدد هش اولیه نسبت به m با مقدار این باقی‌مانده نسبت به m تفاوتی ندارد.

برای حذف یک حرف از اول رشته می‌توانیم تصور کنیم که msb را باید حذف کنیم، پس باید عدد فعلی را از $c * a^{|u|-1}$ کم کنیم.

skip(c):

$$u = u - c \cdot a^{|u|-1}$$

کد زیر این الگوریتم را با توجه به نکات گفته‌شده پیاده‌سازی کرده است.

In [5]:
```python
# الگوریتم رابین-کارپ برای بررسی وجود رشته s در رشته t و عدد اول ۲۶ و هش پایه q با استفاده از
def rabin_karp(t, s, q):
    t_hash = 0  # مقدار هش رشته t
    s_hash = 0  # مقدار هش رشته s
    h = 1       # ضریب بیشینه برای حذف حرف اول

    # محاسبه ضریب h برای حذف حرف اول از پنجره
    for i in range(len(s) - 1):
        h = (h * 26) % q

    # محاسبه هش اولیه برای رشته s و اولین زیررشته از t
    for i in range(len(s)):
        s_hash = (26 * s_hash + ord(s[i])) % q
        t_hash = (26 * t_hash + ord(t[i])) % q

    # بررسی تمام زیررشته‌های t با طول برابر s
    for i in range(len(t) - len(s) + 1):
        if s_hash == t_hash:
            if t[i:i + len(s)] == s:  # بررسی دقیق در صورت برابر بودن هش‌ها
                return True

        if i < len(t) - len(s):
            # حذف حرف اول و افزودن حرف جدید به پنجره هش
            t_hash = (26 * (t_hash - ord(t[i]) * h) + ord(t[i + len(s)])) % q
            if t_hash < 0:
                t_hash += q  # اصلاح مقدار منفی هش

    return False
```

In [6]:
```python
# تست الگوریتم با رشته‌های نمونه
t = "salam bar aqa daqiq"
s = "daqiq"
print(rabin_karp(t, s, 157))  # خروجی باید True باشد چون "daqiq" در t وجود دارد
```

```
True
```

تحلیل زمانی و جمع‌بندی ⏱️

تحلیل زمانی و جمع‌بندی به بررسی مدت زمان اجرای الگوریتم‌ها و جمع‌بندی نتایج آن‌ها می‌پردازد تا عملکرد کلی الگوریتم‌ها ارزیابی شود. این بخش کمک می‌کند تا نقاط قوت و ضعف الگوریتم‌ها در شرایط مختلف شناسایی شود.

مرور مراحل الگوریتم 📌

یک‌بار دیگر اتفاقات افتاده در الگوریتم را خلاصتا مرور می‌کنیم:

1. دو رشته s , t را ورودی می‌گیریم، عدد q را به عنوان یک عدد اول بزرگ برای هش کردن نیز در نظر می‌گیریم.
2. مقدار هش رشته s را محاسبه می‌کنیم.
3. مقدار هش s کاراکتر اول t را محاسبه می‌کنیم.
4. در صورتی که مقدار هش دو رشته یکی بودند، حرف به حرف چک می‌کنیم و در صورت برابر بودن جواب را برمی‌گردانیم.
5. در غیر این صورت حرف اول را از مقدار هش rt اسکیپ کرده و حرف جدید بعدی را به آن اضافه می‌کنیم.
6. تا به انتهای رشته t نرسیده‌ایم، به مرحلهٔ ۴ برمی‌گردیم.

هم‌چنین برای اضافه کردن حرف به انتهای رشته و حذف حرف از ابتدای آن از نکات گفته‌شده در قسمت پیاده‌سازی استفاده می‌کنیم.

در تحلیل زمانی اگر از double check رشته استفاده نکنیم و اگر مقدار هش دو رشته یکی بود مقدار True را برگردانیم، الگوریتم از سری الگوریتم‌های مونت‌کارلو خواهد بود و ممکن است در شرایطی پاسخ صحیح را برنگرداند، اما در اردر زمانی **o(|s| + |t|)** پاسخ را پیدا می‌کند. اما اگر double check کنیم، در شرایطی می‌توان نشان داد که در average-case الگوریتم در اردر **o(|s| + |t|)** جواب می‌دهد. پیشنهاد می‌کنم برای آشنایی بیشتر با این الگوریتم این لینک را نیز ببینید.

In [7]:
```python
# الگوریتم Rabin-Karp با استفاده از پیش‌محاسبه توان‌ها و هش کل رشته
def rabin_carp_19(t, s):
    print("Salam")
    if len(s) > len(t):
        return

    mod = int(1e9 + 7)  # عدد اول بزرگ برای جلوگیری از برخورد هش
    p = 2               # پایه درهم‌سازی
    powers = [1]        # لیست توان‌های p

    # پیش‌محاسبه توان‌های p تا طول رشته t
    for i in range(1, len(t) + 1):
        powers.append((powers[-1] * p) % mod)

    t_hash = []         # لیست هش تجمعی برای رشته t
    s_hash = 0          # مقدار هش رشته s

    for i in range(len(t)):
        last = t_hash[-1] if i > 0 else 0
        val = (ord(t[i]) - ord('a') + 1) * powers[len(t) - i] % mod
        t_hash.append((last + val) % mod)

        if i < len(s):
            s_val = (ord(s[i]) - ord('a') + 1) * powers[len(t) - i] % mod
            s_hash = (s_hash + s_val) % mod

    # چاپ هش‌های تجمعی رشته t
    for h in t_hash:
        print(h, end=' ')
    print()
    print(s_hash)

    # بررسی تطابق زیررشته‌ها با هش s
    for i in range(len(t) - len(s) + 1):
        left = t_hash[i - 1] if i > 0 else 0
        substr_hash = (t_hash[i + len(s) - 1] - left + mod) % mod
        substr_hash = (substr_hash * powers[i]) % mod
        print(substr_hash, end=' ')
        if substr_hash == s_hash:
            print(f"Found an occurrence at index {i}")
```

In [8]:
```python
# تست الگوریتم با رشته‌های نمونه
rabin_carp_19("aaaaaaaaaa", "a")
```

```
Salam
1024 1536 1792 1920 1984 2016 2032 2040 2044 2046
1024
1024 Found an occurrence at index 0
1024 Found an occurrence at index 1
1024 Found an occurrence at index 2
1024 Found an occurrence at index 3
1024 Found an occurrence at index 4
1024 Found an occurrence at index 5
1024 Found an occurrence at index 6
1024 Found an occurrence at index 7
1024 Found an occurrence at index 8
1024 Found an occurrence at index 9
```

In [10]:
```python
# تبدیل کاراکتر به عدد بر اساس موقعیت در الفبای انگلیسی
def char_to_number(c):
    return ord(c) - ord('a') + 1

# تست تابع با کاراکتر 's'
print(char_to_number('s'))
```

```
19
```

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل ششم: تطابق رشته

تطابق رشته‌ها:الگوریتم kmp

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 🗂️

مقدمه 📖

تا کنون الگوریتم‌های مختلفی را برای تطابق رشته دیده‌اید. الگوریتم KMP هم یک الگوریتم تطابق رشته است که مبنای آن محاسبه و استفاده از یک تابع پیشوندی π روی الگویی است که به دنبال آن هستیم (π کاری به رشته اصلی ندارد).

فرض کنید می‌خواهیم رشته **Pattern[1..m]** را در رشته **S[1..n]** پیدا کنیم. به صورت شهودی می‌خواهیم از چپ به راست حرکت کنیم و هر بار بدانیم که چه‌مقدار از Pattern را دیدیم. الگوریتم بدیهی را در نظر بگیرید: برای هر جای رشته تا جایی که با الگو مطابقت دارد جلو می‌رویم و اگر کل رشته الگو تطابق داشت، یک match رخ داده است. از نظر زمانی این روش **O(n × m)** هزینه دارد که برای رشته‌های طولانی نابهینه است.

تابع پیشوند برای یک الگو 🔗

فرض کنیم دنبال یافتن **aab** هستیم. وقتی روی کاراکترهای S حرکت می‌کنیم تا زمانی که **a** نبینیم، اتفاق خاصی نیفتاده. بعد از اولین **a** اگر یک **a** دیگر هم باشد به یافتن الگو نزدیک می‌شویم و اگر نباشد مثل حالت اولیه است که هیچ تطابق نسبی‌ای پیدا نکردیم. اگر الان در وضعیت **aa** باشیم، بر حسب حرف بعدی در چه حالتی قرار می‌گیریم؟ آیا می‌توانید برای رشته‌های طولانی‌تر این تغییر حالت را بسط دهید؟

تابع پیشوند برای یک الگو، اطلاعاتی را در مورد اینکه الگو چگونه با جابجایی‌هایی از خود مطابقت می‌یابد ذخیره می‌کند.
برای همین مفید است جواب سوال زیر را بدانیم: با دانستن اینکه کارکترهای **P[1..q]** در الگو با کارکترهای **S[s+1..s+q]** در متن مطابقت دارند، کوچک‌ترین جابجایی **s' > s** به طوری که **P[1..k] = S[s'+1..s'+k]** که در آن **s' + k = s + q** باشد چیست؟

به طور مثال، فرض کنید یک زیررشته به طول 5 از Pattern با یک زیررشته به طول 5 از رشته اصلی مطابقت دارد. اگر کارکتر ششم با حرف بعدی S مطابقت داشت، جلو می‌رویم، در غیر این صورت به کمک تابع **π** می‌فهمیم که به چه پیشوند کوتاه‌تری از Pattern بازگردیم که بلندترین باشد و همچنان آخر متنمان با آن مساوی است!

مراحل الگوریتم

فرض کنید می‌خواهیم رخدادهای **t** در **s** را پیدا کنیم. زیررشته‌ی متوالی حروف i تا j یک رشته مانند **a** را با **a[i..j]** نشان می‌دهیم.
فرض کنید **F(i)** بلندترین پیشوند از **t** باشد که پسوند **t[0..i-1]** نیز هست اما با آن برابر نیست. برای **i=0**، فرض می‌کنیم **F(0)=-1** و **F(1)=0**. برای مثال:

i 0 1 2 3 4 5 6 7 W[i] A B C D A B D - F[i] -1 0 0 0 0 1 2 0

توضیح الگوریتم KMP

فرض کنید بتوانیم این تابع را در **O(m)** برای **t** محاسبه کنیم. الگوریتم KMP با استفاده از این تابع می‌تواند در **O(n+m)** مسئله‌ی یافتن رخدادها را حل کند. برای این‌کار الگوریتم روی حروف **s** حرکت می‌کند و با اضافه شدن هر حرف، عدد **len** را به‌روزرسانی می‌کند طوری که عدد **len** بعد از اضافه شدن حرف **i** برابر با طول بلندترین پیشوند **t** باشد که پسوند **s[0..i]** است.

برای این‌کار کافی است به این نکته توجه شود که در هر لحظه **s[i-len+1..i]=t[0..len-1]** و به ازای هر **y>len**، **s[i-y+1..i]≠t[0..y-1]**. بنابراین اگر **s[i-z+1..i+1]=t[0..z-1]** داریم **z≤len+1**. بنابراین در محاسبه‌ی **len** پس از اضافه شدن حرف **i+1** تنها زیررشته‌ی **s[i-x+1..i+1]** موثر است. براساس همین استدلال می‌توان به صورت استقرایی نتیجه گرفت که برای محاسبه‌ی **len** بعدی می‌توان از تکه‌کد زیر استفاده کرد.

در واقع محاسبه خود تابع پیشوندی هم از روشی مشابه استفاده می‌کند؛ انگار دنبال پیدا کردن pattern در خودش هستیم، صرفا نمی‌خواهیم کامل آن را پیدا کنیم. با دقت به نوع تغییرات در متغیرهای **cur** و **len** می‌توانید ببینید چرا الگوریتم خطی است!

while(len && s[i] != pat[len]) len = f[len]; if(s[i] == pat[len]) len++;

In [1]:
```python
# برای الگو (Pi) محاسبه تابع پیشوندی
def compute_prefix_function(pat):
    f = [0] * (len(pat) + 1)  # f[0] استفاده نمی‌شود، از f[1] تا f[m]
    f[1] = 0
    cur = 0  # طول بلندترین پیشوند که پسوند نیز هست

    for i in range(1, len(pat)):
        while cur and pat[i] != pat[cur]:
            cur = f[cur]  # بازگشت به پیشوند کوتاه‌تر
        if pat[i] == pat[cur]:
            cur += 1
        f[i + 1] = cur  # مقدار تابع پیشوندی برای موقعیت i+1
    return f
```

In [2]:
```python
# برای یافتن تطابق‌های الگو در متن KMP اجرای الگوریتم
def kmp_search(s, pat):
    f = compute_prefix_function(pat)

    print("test is:", s)
    print("pat is:", pat)
    print("Pi function\n")
    for i in range(1, len(pat) + 1):
        print(f"F[{i}] = {f[i]}")

    print("\nGoing over S to find pat")
    len_match = 0  # طول تطابق فعلی با الگو
    for i in range(len(s)):
        while len_match and s[i] != pat[len_match]:
            len_match = f[len_match]  # بازگشت به پیشوند کوتاه‌تر
        if s[i] == pat[len_match]:
```

```
            len_match += 1
        print(len_match, end=' ')
        if len_match == len(pat):
            print(f"[a match found at {i}]")
            len_match = f[len_match]  # ادامه جستجو برای تطابق‌های بعدی
```

In [3]:
```
# تست الگوریتم با رشته نمونه
s = "ABABABAC"
pat = "ABABAC"
kmp_search(s, pat)
```

```
test is: ABABABAC
pat is: ABABAC
Pi function

F[1] = 0
F[2] = 0
F[3] = 1
F[4] = 2
F[5] = 3
F[6] = 0

Going over S to find pat
1 2 3 4 5 4 5 6 [a match found at 7]
```

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل هفتم: بیشینه شار، برش کمینه، مروری بر بهبودهای فورد-فالکرسن و ادموندز-کارپ

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 🗂️

مقدمه 📘

در نظریه گراف، شبکه‌ی شار یک گراف جهت‌دار است که شامل دو راس مبدا (**source**) و مقصد (**sink**) است. این گراف معمولا شامل راس‌های دیگری است که با یال‌های جهت‌دار به هم متصل شده‌اند. هر یال در یک شبکه‌ی شار دارای ظرفیت مشخصی است. یک شار تنها زمانی می‌تواند از یک یال عبور کند که از میزان ظرفیت آن یال بیشتر نباشد.

قوانین شبکه شار ✅

یک شبکه‌ی شار باید در شرایط زیر صدق کند:

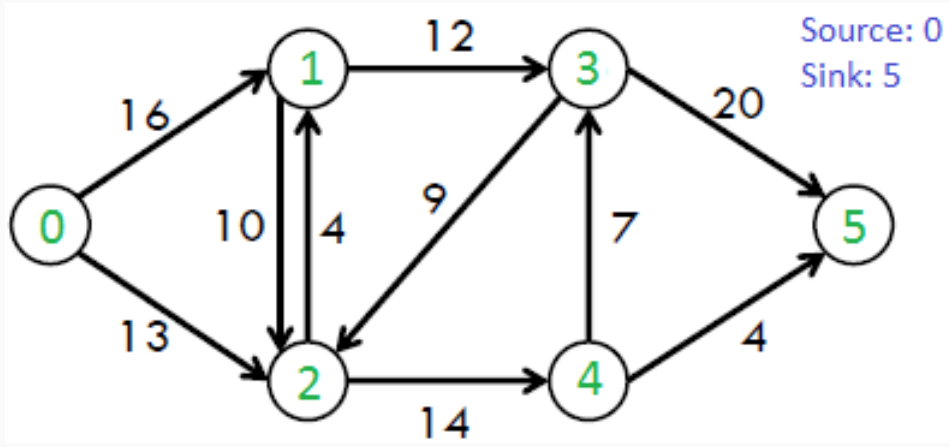
1. شار خروجی از راس مبدا باید با شار ورودی به راس مقصد برابر باشد.
2. برای سایر راس‌ها، میزان شار ورودی باید برابر با میزان شار خروجی باشد.
3. شار عبوری از هر یال عددی نامنفی است و حداکثر برابر با ظرفیت آن یال است.

شار بیشینه 📈

بیشترین شار خروجی از راس مبدا به صورتی که شرایط ذکر شده برای شبکه شار را نقض نکند.

شبکه شار مقابل را در نظر بگیرید:

شبکه شار مثال 🖼️



نتیجه شار بیشینه 🎯

بیشینه شار عبوری از شبکه‌ی بالا برابر **23** است. برای حل مسئله‌ی شار بیشینه، الگوریتم‌های متعددی موجود هستند که در این درس به دو روش **فورد-فالکرسن** و **ادموندز-کارپ** اشاره می‌گردد.

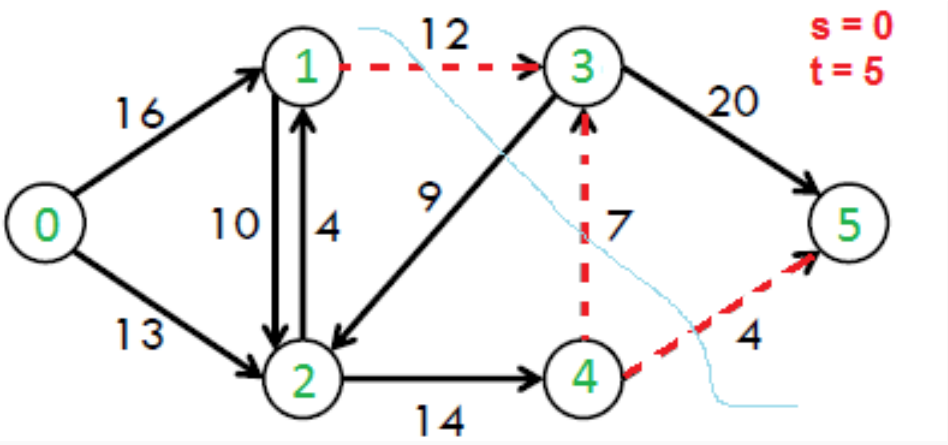
برش کمینه ✂️

برش **s-t** در گراف همبند به صورت مجموعه‌ای از یال‌ها تعریف می‌شود که حذف آن‌ها از گراف، گراف را به دو مؤلفه همبندی تقسیم می‌کند، به گونه‌ای که راس **s** در یک مؤلفه و راس **t** در مؤلفه دیگر قرار می‌گیرد.

برش کمینه‌ی **s-t** در گراف همبند **G**، برشی است که مجموع ظرفیت یال‌های آن کمینه باشد. می‌توان نشان داد که مجموع ظرفیت یال‌های برش کمینه برابر با شار بیشینه عبوری از راس **s** به راس **t** است.

در شکل زیر، برش کمینه را برای گراف **G** مشاهده می‌کنید:

برش کمینه در گراف 🖼️



نکته مهم ⚠️

توجه: بیشینه شار خروجی از راس **s** به راس **t** عددی یکتاست اما برش کمینه لزوما یکتا نیست و ممکن است چندین برش کمینه‌ی **s-t** داشته باشیم.

تمرین 📝

نشان دهید مجموع ظرفیت یال‌های برش کمینه برابر با بیشینه شار عبوری از مبدا به مقصد است. (راهنمایی: از برنامه‌ریزی خطی استفاده کنید و نشان دهید مسئله‌ی برش کمینه، دوگان مسئله‌ی شار بیشینه است)

گراف باقی‌مانده 🔄

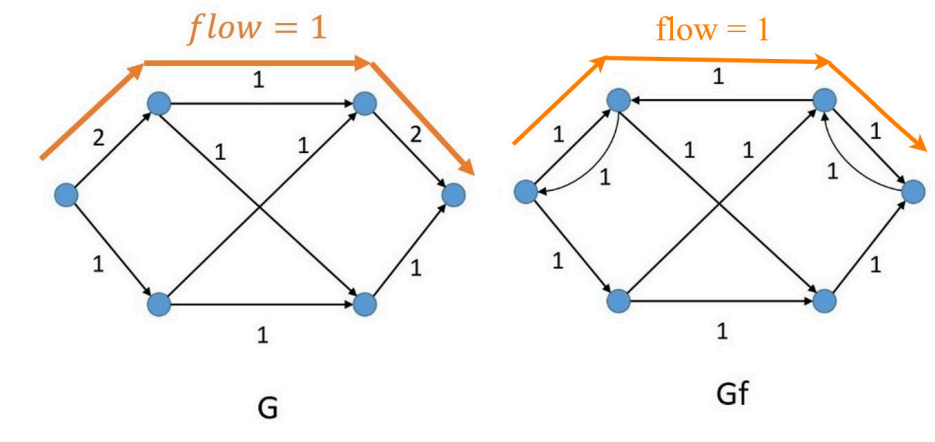
برای یک شار **f**، گراف باقی‌مانده به صورت زیر تعریف می‌شود:

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{اگر } e \in E \\ f(e) & \text{اگر } e^r \in E \end{cases}$$

که در معادله‌ی بالا e^r یالی در خلاف جهت **e** است.

در شکل زیر می‌توانید یک گراف و گراف باقی‌مانده‌ی آن را مشاهده کنید:

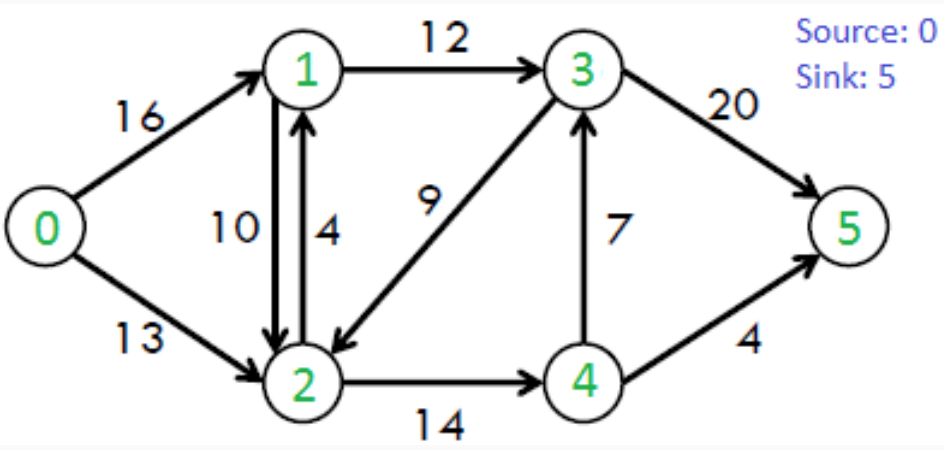
گراف باقی‌مانده مثال 🖼️



روش فورد-فالکرسن ⚙️

شبکه‌ی شار مثال 📊

شبکه‌ی شار مقابل را در نظر بگیرید:



مراحل الگوریتم فورد-فالکرسن 🔄

ابتدا شار عبوری از هر یال را برابر با 0 در نظر می‌گیریم. در روش فورد-فالکرسن تا زمانی که مسیری از راس مبدا به راس مقصد وجود داشته باشد، به گونه‌ای که شار عبوری از همه‌ی یال‌های آن مسیر کمتر از ظرفیت آن یال‌ها باشد، مراحل زیر را طی می‌کنیم:

مراحل الگوریتم فورد-فالکرسن ➕

1. کمینه‌ی ظرفیت یال‌های مسیر را می‌یابیم. این کمینه را با متغیر **m** نشان می‌دهیم.
2. برای همه‌ی یال‌های مسیر تغییرات زیر را اعمال می‌کنیم: (فرض بر آن است که یال‌های جهت‌دار مسیر از راس **u** به راس **v** بوده‌اند)

- $f(u, v) = f(u, v) + m$
- $f(v, u) = f(v, u) - m$

(*f(v, u)* نشان‌دهنده‌ی شار عبوری از یال‌ها در جهت عکس مسیر ذکر شده است)

- $\text{maxFlow} = \text{maxFlow} + m$

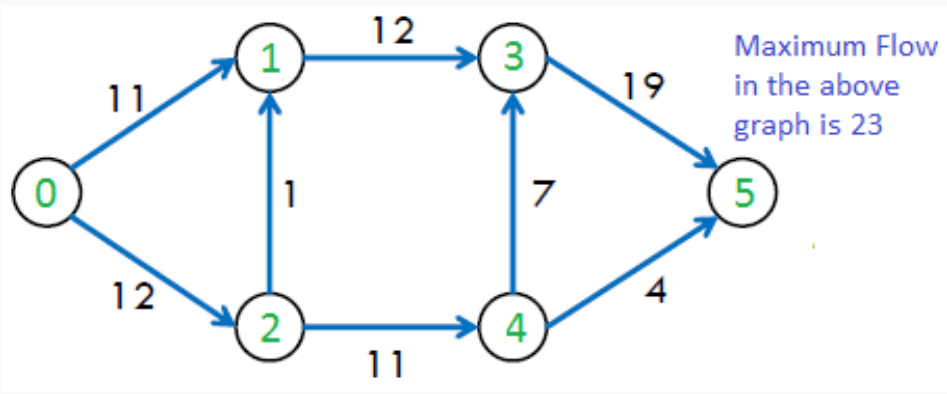
مثال اجرای فورد-فالکرسن 🔢

حال الگوریتم فورد-فالکرسن را بر روی گراف فوق اجرا می‌کنیم. ابتدا دنباله‌ی راس‌های {0,1,3,5} را به عنوان مسیر در نظر می‌گیریم و به اندازه‌ی کمینه‌ی وزن یال یعنی 12 به مقدار **maxFlow** اضافه می‌کنیم. حال دنباله‌ی راس‌های {0,1,2,4,5} را انتخاب می‌کنیم و کمینه‌ی مقدار ظرفیت این مسیر، یعنی 4 را به مقدار **maxFlow** اضافه می‌کنیم و در پایان این مرحله مقدار 16 در **maxFlow** ذخیره می‌گردد. در نهایت مسیر شامل دنباله راس‌های {0,2,4,3,5} را در نظر می‌گیریم. مقدار کمینه‌ی ظرفیت یال‌های این مسیر برابر با 7 است و در نتیجه مقدار 23 در **maxFlow** ذخیره می‌شود.

با توجه به آنکه دیگر مسیری از راس 0 به راس 5 وجود ندارد، به گونه‌ای که شار عبوری از تمام یال‌های مسیر کمتر از مقدار ظرفیت آن یال‌ها باشد، الگوریتم خاتمه می‌یابد و مقدار 23 به عنوان بیشینه شار خروجی داده می‌شود.

نمایش شار بیشینه در گراف 📊

در شکل زیر، می‌توانید چگونگی تقسیم شار بیشینه بین یال‌های گراف فوق را مشاهده کنید:



شبه‌کد الگوریتم فورد-فالکرسن 💻

برای درک بهتر الگوریتم فورد-فالکرسن، شبه‌کد زیر را مشاهده کنید:

```
Ford_Fulkerson(G):
Initialize F = 0, Gf = G
while p = exists_augmenting_path(G):
        augment F along p
        update Gf
return F
```

In [1]:
```python
# Original code

from collections import deque

# پیاده‌سازی الگوریتم فورد-فالکرسون برای محاسبه شار بیشینه در یک گراف جهت‌دار با ظرفیت‌ها
class Graph:
    def __init__(self, vertices):
        self.V = vertices  # تعداد رئوس گراف
        self.graph = [[0] * vertices for _ in range(vertices)]  # ماتریس ظرفیت یال‌ها
```

```
    # تابع BFS برای یافتن مسیر افزایشی از مبدا به مقصد
    def bfs(self, s, t, parent):
        visited = [False] * self.V
        queue = deque()
        queue.append(s)
        visited[s] = True

        while queue:
            u = queue.popleft()
            for v in range(self.V):
                if not visited[v] and self.graph[u][v] > 0:
                    queue.append(v)
                    visited[v] = True
                    parent[v] = u
                    if v == t:
                        return True
        return False

    # الگوریتم اصلی فورد-فالکرسون
    def ford_fulkerson(self, source, sink):
        parent = [-1] * self.V  # برای ذخیره مسیر افزایشی
        max_flow = 0  # مقدار نهایی شار بیشینه

        # تا زمانی که مسیر افزایشی وجود دارد
        while self.bfs(source, sink, parent):
            path_flow = float('inf')
            v = sink
            # یافتن کمترین ظرفیت در مسیر افزایشی
            while v != source:
                u = parent[v]
                path_flow = min(path_flow, self.graph[u][v])
                v = u

            # به‌روزرسانی ظرفیت‌ها در گراف باقیمانده
            v = sink
            while v != source:
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = u

            max_flow += path_flow  # افزودن شار مسیر به شار کل

        return max_flow
```

تحلیل زمانی الگوریتم فورد-فالکرسن ⏱️

اگر گراف G دارای n راس و m یال باشد، مسیر افزایشی (augmenting path) را می‌توان در زمان O(m) یافت. حال با فرض این که ظرفیت هر یال C است، شار بیشینه حداکثر برابر nC خواهد بود. پس در بدترین حالت، مرتبه زمانی الگوریتم O(nmC) است. توجه کنید که این الگوریتم از مرتبه‌ی زمانی چندجمله‌ای نیست.

بهبود ادموندز-کارپ ⚡

این الگوریتم مشابه الگوریتم فورد-فالکرسن است، با این تفاوت که مسیر افزاینده کوتاه‌ترین مسیر افزایشی بین مبدا و مقصد است. برای یافتن کوتاه‌ترین مسیر در این الگوریتم می‌توان از الگوریتم BFS استفاده کرد. این الگوریتم در زیر پیاده‌سازی شده است. شما می‌توانید با وارد کردن گراف خود و ظرفیت هر یال، شار بیشینه را مشاهده کنید:

In [2]:
```
from collections import deque

INF = 987654321

# تعریف کلاس الگوریتم ادموندز-کارپ
class EdmondsKarp:
    def __init__(self, node_count):
        self.node_count = node_count
        self.adjMat = [[0] * node_count for _ in range(node_count)]  # ماتریس ظرفیت یال‌ها
        self.flowMat = [[0] * node_count for _ in range(node_count)]  # ماتریس جریان فعلی
        self.path = [-1] * node_count  # مسیر افزایشی
        self.pathC = [0] * node_count  # ظرفیت مسیر افزایشی
```

In [3]:
```
# تابع BFS برای یافتن کوتاه‌ترین مسیر افزایشی از مبدا به مقصد
def bfs(self, s, t):
    self.path = [-1] * self.node_count
    self.pathC = [0] * self.node_count
    queue = deque()
```

```
    queue.append(s)
    self.pathC[s] = INF

    while queue:
        currentNode = queue.popleft()
        for to in range(self.node_count):
            if self.adjMat[currentNode][to] > 0:
                residual = self.adjMat[currentNode][to] - self.flowMat[currentNode][to]
                if self.path[to] == -1 and residual > 0:
                    self.path[to] = currentNode
                    self.pathC[to] = min(residual, self.pathC[currentNode])
                    if to == t:
                        return self.pathC[t]
                    queue.append(to)
    return 0
```

In [4]:

```
    # تابع اصلی الگوریتم ادموندز-کارپ برای محاسبه شار بیشینه
    def edmonds_karp(self, s, t):
        self.flowMat = [[0] * self.node_count for _ in range(self.node_count)]
        max_flow = 0

        while True:
            flow = self.bfs(s, t)
            if flow == 0:
                break
            max_flow += flow
            currentNode = t
            while currentNode != s:
                previous = self.path[currentNode]
                self.flowMat[previous][currentNode] += flow
                self.flowMat[currentNode][previous] -= flow
                currentNode = previous

        return max_flow
```

In [14]:

```
# Complete Class (to make sure by the following dependencies)

from collections import deque

INF = 987654321

class EdmondsKarp:
    def __init__(self, node_count):
        self.node_count = node_count
        self.adjMat = [[0] * node_count for _ in range(node_count)]  # ظرفیت یال‌ها
        self.flowMat = [[0] * node_count for _ in range(node_count)]  # جریان فعلی
        self.path = [-1] * node_count  # مسیر افزایشی
        self.pathC = [0] * node_count  # ظرفیت مسیر افزایشی

    def bfs(self, s, t):
        self.path = [-1] * self.node_count
        self.pathC = [0] * self.node_count
        queue = deque()
        queue.append(s)
        self.pathC[s] = INF

        while queue:
            currentNode = queue.popleft()
            for to in range(self.node_count):
                if self.adjMat[currentNode][to] > 0:
                    residual = self.adjMat[currentNode][to] - self.flowMat[currentNode][to]
                    if self.path[to] == -1 and residual > 0:
                        self.path[to] = currentNode
                        self.pathC[to] = min(residual, self.pathC[currentNode])
                        if to == t:
                            return self.pathC[t]
                        queue.append(to)
        return 0

    def run(self, s, t):
        self.flowMat = [[0] * self.node_count for _ in range(self.node_count)]
        max_flow = 0

        while True:
            flow = self.bfs(s, t)
            if flow == 0:
                break
            max_flow += flow
            currentNode = t
            while currentNode != s:
                previous = self.path[currentNode]
                self.flowMat[previous][currentNode] += flow
                self.flowMat[currentNode][previous] -= flow
                currentNode = previous

        return max_flow
```

ورود داده برای گراف 📝

تعداد راس‌ها و یال‌ها را وارد کنید:

In [15]:
```
# مرحله ۱: دریافت تعداد رئوس و یال‌ها
node_count = int(input("تعداد رئوس گراف را وارد کنید: "))
edge_count = int(input("تعداد یال‌ها را وارد کنید: "))
```

تعیین راس مبدا و مقصد 🎯

در این قسمت راس‌های مبدا و مقصد را مشخص کنید:

In [16]:
```
# مرحله ۲: دریافت راس مبدا و مقصد
s = int(input("راس مبدا را وارد کنید: "))
t = int(input("راس مقصد را وارد کنید: "))
```

In [17]:
```
# مرحله ۳: تعریف ماتریس ظرفیت یال‌ها
adjMat = [[0] * node_count for _ in range(node_count)]
```

In [18]:
```
# مرحله ۴: دریافت یال‌ها و ظرفیت‌ها
print("یال‌ها را وارد کنید (from to capacity):")
for _ in range(edge_count):
    from_node, to_node, cap = map(int, input().split())
    adjMat[from_node][to_node] = cap
```

```
یال‌ها را وارد کنید (from to capacity):
```

In [19]:
```
# مرحله ۵: تعریف کلاس الگوریتم ادموندز-کارپ
ek = EdmondsKarp(node_count)
ek.adjMat = adjMat
```

In [20]:
```
# مرحله ۶: اجرای الگوریتم و محاسبه شار بیشینه
max_flow = ek.run(s, t)
```

In [21]:
```
# مرحله ۷: چاپ نتیجه
print("شار بیشینه برابر است با:", max_flow)
```

```
شار بیشینه برابر است با: 12
```

تحلیل زمانی الگوریتم ادموندز-کارپ ⏱️

هربار اجرای الگوریتم BFS از مرتبه‌ی $O(m)$ است. ولی در این الگوریتم، در هربار عبور شار از یک مسیر افزایشی می‌توانیم اطمینان داشته باشیم که ظرفیت یک یال پر شده و دیگر استفاده نخواهد شد. پس الگوریتم BFS حداکثر mn بار اجرا می‌شود. لذا مرتبه‌ی زمانی این الگوریتم $O(nm^2)$ است.

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل هفتم: شار

بخش دوم :تطابق در گراف 2 بخشی ،مسیر مجزا ، گردکردن ماتریس

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

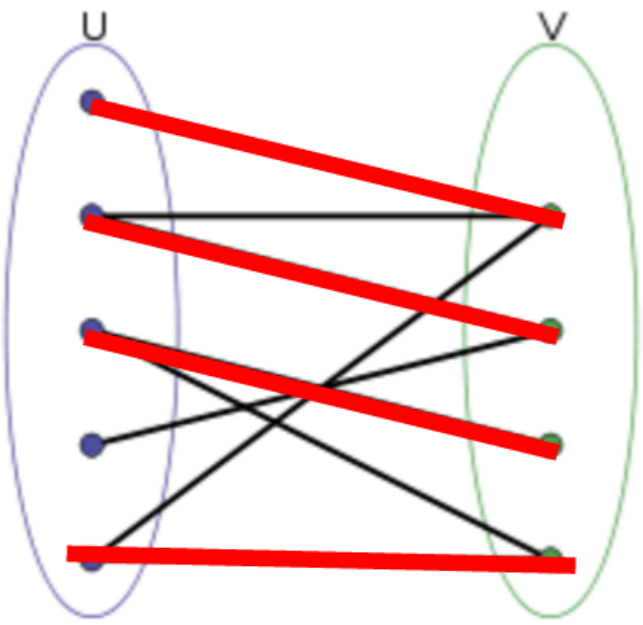
فهرست محتویات 📑

مقدمه 📘

در این بخش می‌خواهیم در ارتباط با کاربردهایی که از مفهوم شار بیشینه در طراحی الگوریتم‌ها وجود دارد صحبت کنیم و ببینیم چه مسائلی که به ظاهر مسائل شار بیشینه نیستند را می‌توان با استفاده از آنچه در بخش گذشته آموختیم حل کرد. سه دسته از این مسائل را در این بخش مرور می‌کنیم، اما مسائل متنوع دیگری نیز می‌توان به این لیست اضافه کرد.

مسئله‌ی اول: تطابق در گراف 2 بخشی 🔗

مسئله اول پیدا کردن بیشترین تطابق در یک گراف دو بخشی داده شده است. گراف دو بخشی گرافی است که بتوان راس‌های آن را به دو بخش تقسیم کرد به گونه‌ای که هر راس با راس‌های بخش خود هیچ یالی نداشته باشد. حال می‌خواهیم در چنین گرافی بیشترین تطابق را پیدا کنیم. تطابق، زیرگرافی است که در آن هر راس حداکثر درجه ۱ داشته باشد یا به عبارتی دیگر هیچ دو یالی وجود نداشته باشد که راسی را به اشتراک بگذارند.

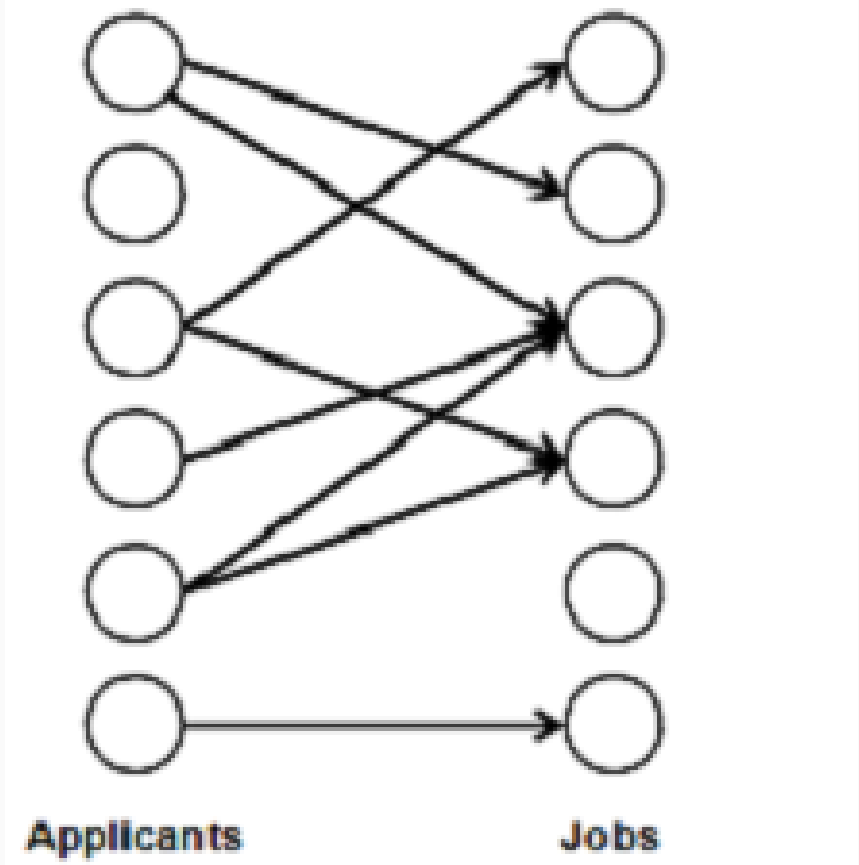


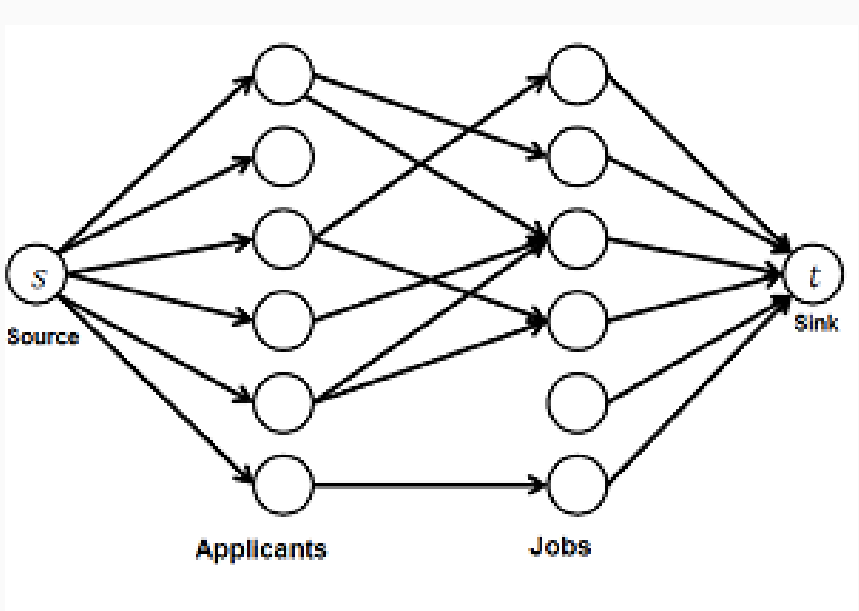
تطابق در گراف دو بخشی (bipartite matching)

در شکل بالا، یال‌هایی که قرمز شده‌اند، تطابقی را در گراف دو بخشی نشان می‌دهند.

راه حل 🛠️

برای حل مسئله با استفاده از مسئله بیشینه شار، گراف G داده شده را به گراف H تبدیل می‌کنیم. برای این کار، هر یالی که بین دو مجموعه سمت چپ و راست وجود دارد را در نظر می‌گیریم و یال را از راس سمت چپ به راس سمت راست جهت‌دار می‌کنیم. دو راس s و t در نظر می‌گیریم. از راس s به همه رئوس مجموعه چپ یال جهت‌دار می‌کشیم و از همه رئوس مجموعه راست به راس t نیز یال جهت‌دار می‌کنیم. سپس ظرفیت همه یال‌ها را برابر 1 قرار می‌دهیم. حال برای گراف H می‌توانیم مسئله بیشینه شار را از مبدا s به مقصد t حل کنیم.



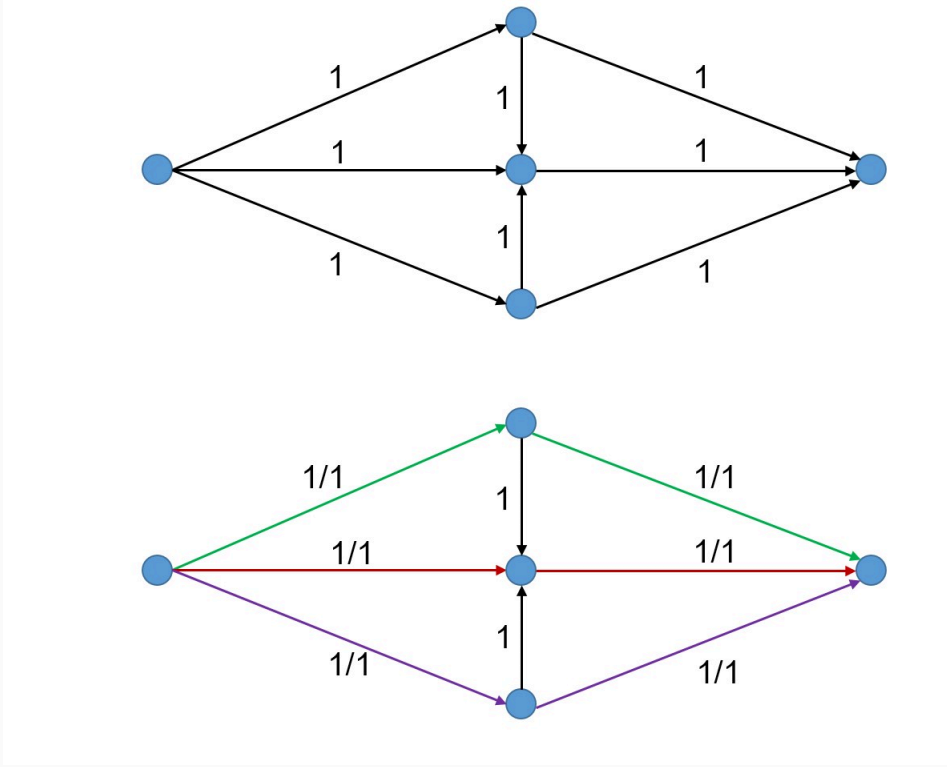


حال ادعا می‌کنیم پیدا کردن ماکسیمم شار در گراف ساخته شده برابر با ماکسیمم تطابق در گراف اولیه است. هر واحد شار مسیری از s به یکی از رئوس سمت چپ و از آن راس به یکی از رئوس سمت راست و سپس به راس t طی می‌کند. در نتیجه، ظرفیت همه یال‌هایی که طی شده پس از عبور یک واحد شار برابر با صفر خواهد شد. با توجه به اینکه جهت همه یال‌ها از سمت چپ به سمت راست است، پس بعد از این هیچ گاه ظرفیت یال‌ها تغییر نمی‌کند. این یک واحد شار عبور داده شده برابر با یک عضو از تطابق است، زیرا باعث می‌شود پس از عبور شار، دو راس وسطی که از آن‌ها شار عبور کرده حذف شوند و تمامی یال‌های متصل به آن‌ها نیز حذف شوند. بنابراین، چون هر واحد شار یک عضو تطابق است، ماکسیمم شار برابر با ماکسیمم تطابق است.

اگر بتوانیم ماکسیمم شار را در گراف ساخته شده پیدا کنیم، ماکسیمم تطابق در گراف اولیه برابر خواهد بود با یال‌های میانی که ظرفیتشان صفر شده است.

مسئله‌ی دوم: مسیرهای مجزا 🏞️

در این مسئله هدف این است که بیشترین تعداد مسیر با یال‌های مجزا را بین دو راس s و t از یک گراف پیدا کنیم. فرض کنید گراف جهت‌دار است. با قرار دادن ظرفیت 1 برای هر یال و یافتن شار بیشینه، بیشترین تعداد مسیرهای مجزا نیز به دست می‌آید. زیرا شاری که به راس t می‌رسد، در واقع تعداد مسیرهایی را نشان می‌دهد که از s به t می‌رسند و چون ظرفیت هر یال 1 است، هیچ دو مسیری نمی‌توانند یال مشترک داشته باشند.



نمایش مسیرهای مجزا در گراف

تمرین ✏️

برای گراف بدون جهت، چطور می‌توان مسیرهای مجزا را به دست آورد؟

تمرین ✏️

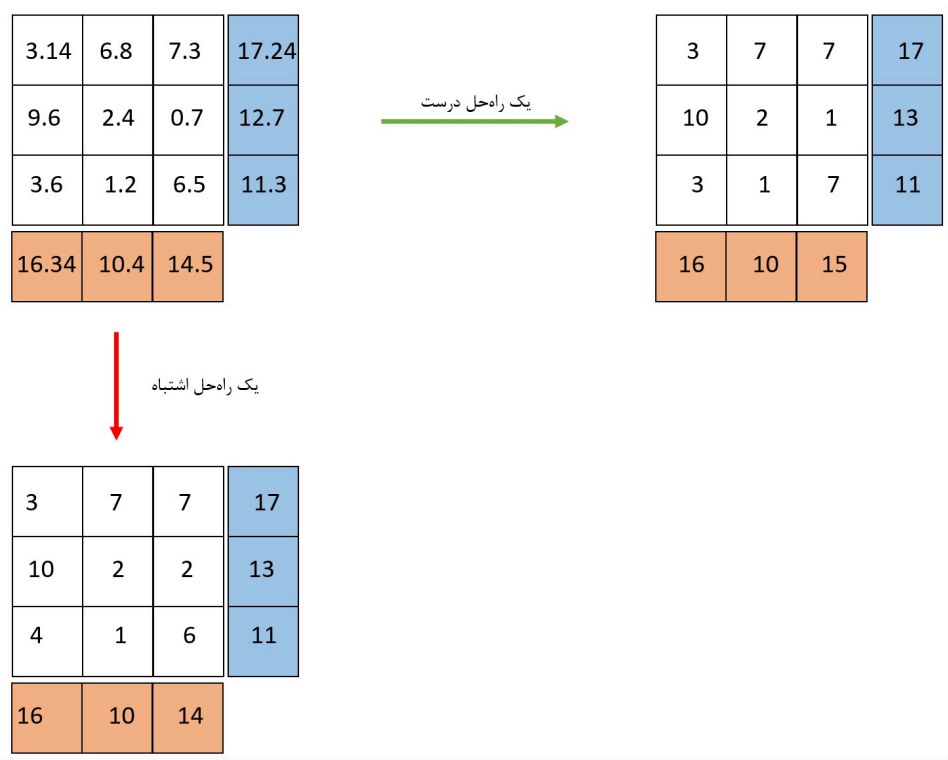
چگونه می‌توان بیشترین تعداد مسیر راس‌مجزا را به دست آورد؟

راهنمایی: باید برای راس‌ها ظرفیت تعریف کرد و راهی برای حل الگوریتم شار بیشینه پیدا نمود که در آن راس‌ها نیز دارای ظرفیت باشند.

مسئله‌ی سوم: گرد کردن ماتریس 🧮

فرض کنید یک ماتریس $m \times n$ به نام D داریم. در این ماتریس جمع اعضای سطر i را با r_i و جمع اعضای ستون j را با c_j نمایش می‌دهیم. هدف این مسئله این است که همه‌ی اعضای ماتریس (d_{ij}) و همچنین r_i و c_j ها را طوری گرد کنیم که جمع‌ها پایدار باشند. به این معنا که جمع اعضای گرد شده‌ی ماتریس در سطر i با r_i گرد شده برابر باشد و این موضوع برای ستون‌ها نیز صدق کند.

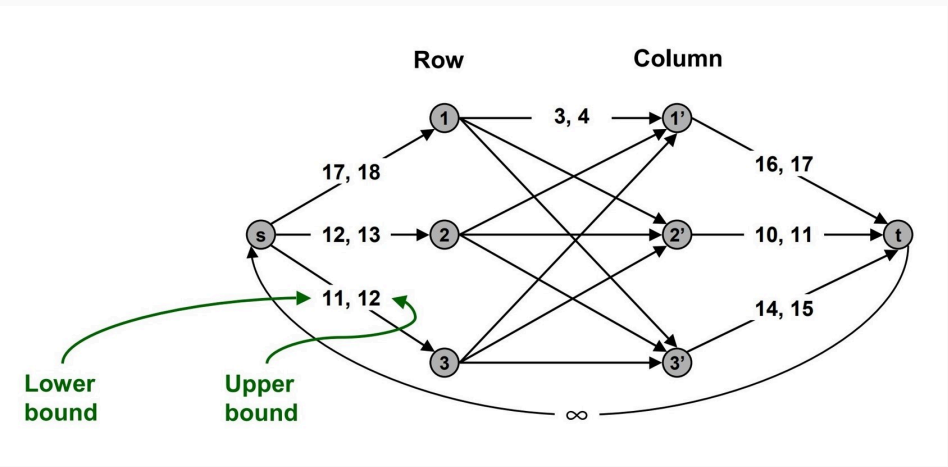
شاید اولین راه که به ذهن برسد، قرار دادن یک حد آستانه مانند 0.5 برای گرد کردن باشد، اما همانطور که در مثال زیر مشاهده می‌کنید این روش اشتباه است.



نمونه مشکل گرد کردن ماتریس با روش ساده

برای حل این مسئله می‌توان از الگوریتم شار بیشینه استفاده کرد، اما ابتدا باید مسئله را به صورت یک گراف مدل کنیم. گراف دوبخشی می‌سازیم که یک بخش مربوط به سطرها و یک بخش مربوط به ستون‌ها است. بین هر راس مربوط به سطر i و ستون j یک یال وجود دارد که مربوط به عضو d_{ij} ماتریس است. همچنین یک راس سورس (S) به همه راس‌های بخش سطر متصل است و یک راس سینک (t) به همه راس‌های بخش ستون.

ظرفیت کمینه‌ی هر یال برابر گرد شده‌ی رو به پایین آن عضو و ظرفیت بیشینه برابر گرد شده‌ی رو به بالای آن عضو است. برای یال‌های بین سورس و سینک و راس‌های سطر و ستون نیز کمینه‌ی ظرفیت برابر گرد شده‌ی رو به پایین مجموع آن سطر یا ستون و بیشینه‌ی ظرفیت برابر گرد شده‌ی رو به بالای آن سطر یا ستون است. اگر معادله‌ی شار ورودی و خروجی برای هر راس را بنویسید، واضح است که این معادلات همان محدودیت‌هایی هستند که با حل آن‌ها مسئله گرد کردن ماتریس حل می‌شود.



گراف ساخته شده برای مسئله گرد کردن ماتریس

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل هشتم: برنامه‌ریزی خطی

بخش اول: برنامه ریزی خطی، فرم استاندارد، مدل‌سازی، الگوریتم سیمپلکس

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 📖

مقدمه 📘

بهینه‌سازی یک راه مهم زندگی کردن است، ما همیشه در کارهایی که می‌کنیم منابع محدودی داریم و در استفاده از آن‌ها مجبور به انتخاب بهینه هستیم. بهینه‌سازی به تنهایی یک موضوع بسیار گسترده است و یکی از مفاهیم مهم آن **برنامه‌ریزی خطی (Linear Programming)** می‌باشد.

برنامه‌ریزی خطی

برنامه‌ریزی خطی چیست؟ برنامه‌ریزی خطی یک تکنیک ساده است که با استفاده از آن روابط پیچیده را از طریق تابع‌های خطی نمایش می‌دهیم و در نهایت نقطه بهینه این دستگاه را پیدا می‌کنیم. نکته مهم این است که کلیت روابط ممکن است بسیار پیچیده باشد، ولی ما با برنامه‌ریزی خطی آن‌ها را بسیار ساده و جامع مثال می‌کنیم.

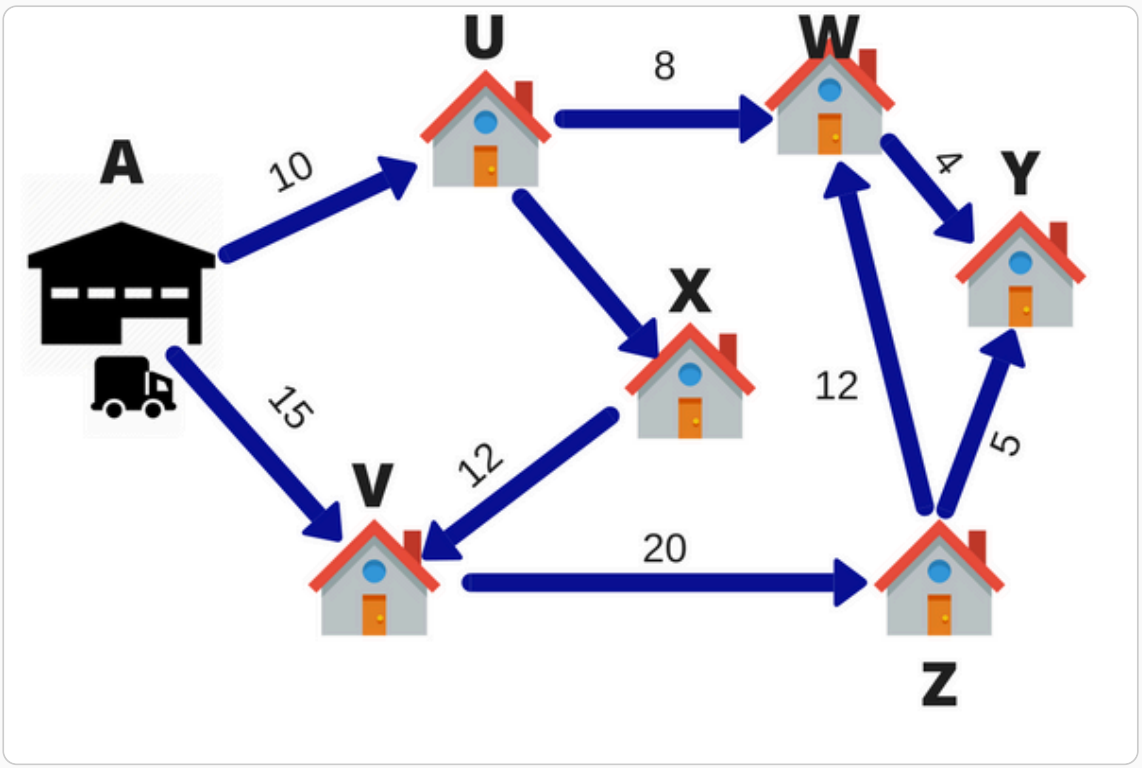
استفاده و کاربرد برنامه‌ریزی خطی در همه‌جا دیده می‌شود. برای مثال، وقتی می‌خواهید تمرینات خود را به گونه‌ای و به ترتیبی انجام دهید که به موقع تمامی آن‌ها را پیش از ددلاین انجام دهید، از برنامه‌ریزی خطی به صورت ناخودآگاه بهره می‌برید. حتی هنگامی که برای بازگشت از دانشگاه به خانه مسیریابی می‌کنید، در عمل از این تکنیک بهره می‌برید.

در این یادداشت ابتدا دو مسئله بهینه‌سازی که یکی خطی و دیگری غیرخطی است معرفی می‌کنیم، سپس به بیان مفاهیم LP به کمک مثال‌های متنوع و کاربردی می‌پردازیم و در نهایت یک روش ساده و بسیار کاربردی در حل مسائل LP را معرفی می‌کنیم.

📋 طرح چند مسئله

مسئله اول: پستچی

پستچی داریم که می‌خواهد ۶ بسته را در روز به مقصد برساند. پست‌خانه در نقطه A قرار دارد و ۶ مقصد با نقاط (U, V, W, X, Y, Z) در شکل نمایش داده شده است. اعداد روی خطوط فاصله بین خانه‌ها را نشان می‌دهد. برای ذخیره زمان و بنزین، پستچی می‌خواهد کوتاه‌ترین مسیر ممکن را طی کند.



برای این مسئله پستچی تمامی مسیرها را بررسی می‌کند تا بهترین آن‌ها را انتخاب کند. این تکنیک انتخاب همان مفهوم بهینه‌سازی است. در این مثال هدف پستچی رساندن همه نامه‌ها به موقع است. برنامه‌ریزی خطی برای پیدا کردن بهترین جواب با وجود شروط خطی روی ورودی‌ها استفاده می‌شود. آیا می‌توانید شروط خطی برای این مسئله بیان کنید؟

مسئله دوم: برنامه‌ای برای زندگی احسن!!

می‌دانیم هر هفته از ۱۶۸ ساعت تشکیل شده‌است. فرض کنید شما برای یک زندگی سالم و مفید می‌خواهید زمان خود را به سه بخش درس و دانشگاه (S)، تفریح (P) و دیگر کارها مانند خوابیدن و غذا خوردن (E) تقسیم کنید. برای اینکه سلامت روان خود را نگه داریم، نیاز داریم:

$$P + E \geq 70$$

همچنین برای گرفتن نمرات خوب از دروس خود:

$$S \geq 60$$

و همچنین شرط اول برای شاداب بودن در طول روز ۸ ساعت خوابیدن است:

$$2S + E - 3P \geq 150$$

پرسش اول:

آیا ما می‌توانیم این کار را انجام دهیم؟ آیا جوابی وجود دارد؟

بله، برای مثال: S=80, P=20, E=68

پرسش دوم:

فرض کنید مقداری خوشحال بودن ما با عبارت 2P + E بیان شود. جوابی را پیدا کنید که در روابط صدق می‌کند. به فرمول بالا که می‌خواهیم آن را بیشینه (گاهی کمینه) کنیم **تابع هدف** می‌گویند.

رابطه بالا نیز برنامه‌ریزی خطی است زیرا تمام شروط ما خطی و همچنین تابع هدف نیز خطی است.

در بخش بعد به بیان چند تعریف اصلی در برنامه‌ریزی خطی می‌پردازیم.

واژه‌شناسی در برنامه‌ریزی خطی

- **متغیرهای تصمیم‌گیری (Decision Variables):** متغیرهایی هستند که خروجی ما را مشخص می‌کنند. برای حل هر مسئله نیاز داریم این متغیرها را مشخص کنیم.
- **تابع هدف (Objective Function):** به عنوان هدف تصمیم‌گیری‌های ما شناخته می‌شود.
- **شروط (Constraints):** محدودیت‌ها روی متغیرهای تصمیم‌گیری.
- **محدودیت‌های نامنفی (Non-negative Restriction):** برای همه برنامه‌ریزی‌های خطی، همه متغیرها باید نامنفی باشند.

در بخش بعدی به بیان فرم استاندارد برنامه‌ریزی خطی می‌پردازیم.

فرم استاندارد 📝

در تعریف برنامه‌ریزی خطی (LP) پیش از این بیان شد که LP روشی برای رسیدن به خروجی بهینه با رعایت دستگاهی از شروط خطی است. اگر بخواهیم دقیق‌تر بگوییم، برنامه‌ریزی خطی می‌تواند مسئله موجود که بیشینه یا کمینه کردن یک تابع هدف است را تحت محدودیت‌های خطی حل کند.

فرم استاندارد متشکل است از:

متغیرها:

$$x = (x_1, x_2, . . , x_d)^T$$

تابع هدف:

$$c^T x = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$$

$$c = (c_1, c_2, \ldots, c_d)^T$$

شروط:

$$a_{11} x_1 + a_{12} x_2 + \cdots + a_{1n} x_n \leq b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2$$

$$\cdots$$

$$a_{d1}x_1 + a_{d2}x_2 + \cdots + a_{dn}x_n \leq b_d$$

$Ax \leq b$ که A ماتریسی $d \times n$ است

هدف نهایی ما بیشینه کردن تابع هدف است. LP کاربردهای بسیاری دارد، مانند حل مسئله شار یا یافتن کوتاه‌ترین مسیر بین یک راس مشخص و دیگر رئوس در گراف (shortest path). یعنی می‌توان این مسائل را به LP تبدیل و سپس حل کرد.

مسأله تقسیم بهینه کار ⚙️

فرض کنید شما مدیریت منابع‌مالی و انسانی شرکت بزرگی را به عهده دارید و شما در کارتان بسیار خبره هستید اما دیگر علاقه‌ای به ادامه کار ندارید. از قضا مدیر شرکت صمیمی‌ترین دوست شما است و استعفا شما بزرگترین لطمه روحی را به او میزند و شما نیز بسیار انسانی خجالتی هستید. ناگهان فکری به سر شما می‌آید: «آنقدر بد کار کن تا اخراج شوی». حالا شما با نگاه به پروژه‌های امروز خود می‌بینید: l تا کار دارید و i نفر برای انجام دادن آنها درخواست داده‌اند. قیمت فرد iام برای یک روز کارکردن در کار jام برابر

$$a_{ij}$$

است. مساله این است که به‌گونه‌ای این i فرد را در این l کار قرار دهید که مجموع میزان پولی که صرف می‌شود بیشترین شود.

پاسخ 📝

پاسخ باید به‌گونه‌ای باشد که

$$x_{ij}$$

مقدار بخش زمانی باشد که فرد i برای کار jام صرف می‌کند، یعنی این فرد چه کسری از زمانش را برای این کار صرف می‌کند. پس داریم:

$$\sum_{j=1}^{J} x_{ij} \leq 1$$

و

$$\sum_{i=1}^{I} x_{ij} \leq 1$$

$$x_{ij} \geq 0 \quad \text{for } i = 1..I \text{ and } j = 1..J$$

معادله اول به این معنا است که هیچ‌کس نمی‌تواند بیش از صددرصد زمانش را کار کند و معادله دوم بیان می‌کند که فقط یک نفر در هر زمان می‌تواند در هر کار کار کند (مفهوم موازی نداریم). معادله سوم نیز شرط بدیهی نبود کار منفی را بیان می‌دارد. با توجه به شروط بالا، ما به‌دنبال بیشینه کردن تابع هدف زیر خواهیم بود:

$$\sum_{i=1}^{I}\sum_{j=1}^{J} a_{ij} \cdot x_{ij}$$

در ادامه، این مساله همان حالت استاندارد با

$$n = I + J$$

و

$$d = I \cdot J$$

می‌باشد.

سوال ❓

اگر این مساله را به‌گونه‌ای می‌خواستیم حل کنیم که اخراج نشویم، چه‌طور؟

قضیه دوگان 🔄

این قضیه بیان می‌کند برای هر مسئله که به شکل استاندارد (LP) بیان می‌شود:

Maximize:

$$c \cdot x$$

Subject to:

$$Ax \leq b,\ x \geq 0$$

یک دوگان وجود دارد که:

Minimize:

$$b \cdot y$$

Subject to:

$$A^T y \geq c,\ y \geq 0$$

این خاصیت اهمیت بسیاری دارد و می‌تواند قضایا زیادی را اثبات کند، مانند اینکه اگر قضیه Max Flow را به صورت برنامه‌ریزی خطی بیان کنیم، از روی این خاصیت ثابت می‌شود جواب آن، جواب دوگانش است که دوگانش هم قضیه Min Cut خواهد شد.

تبدیل به فرم استاندارد 🔄

ممکن است پس از نوشتن مسئله خود به صورت برنامه‌ریزی خطی، شکل آن به شکل استاندارد نباشد، ولی راه‌های بسیار ساده‌ای برای تبدیل هر دستگاه معادله‌ای به فرم استاندارد وجود دارد. در زیر به بیان این موارد می‌پردازیم:

- کمینه کردن تابع هدف: برای این کار ضرایب را در -۱ ضرب کنید و تابع هدف بدست آمده را بیشینه کنید 📈

- متغیری وجود دارد که شرط نامنفی بودن ندارد:

$$x_i \rightarrow x_j - x_k$$

and

$$x_j \geq 0, x_k \geq 0$$

- شروطی که تساوی دارند:

$$x = b \rightarrow x \leq b, x \geq b$$

یعنی در حقیقت به جای معادله تساوی، دو نابرابری بالا را به دستگاه معادلات اضافه می‌کنیم ⚖️

- شروطی که به شکل بزرگتر یا مساوی هستند:

ضرایب را در -۱ ضرب می‌کنیم و تبدیل به کوچکتر مساوی می‌کنیم ↕️

نکته مهم این است که تعاریف بالا برای فرم استاندارد گفته شده و نه حالت دوگان، البته حالت دوگان نیز به سادگی قابل تفهیم است.

مدل سازی 🛠️

در این بخش می‌خواهیم به مبحث مدل کردن پرسش‌های خود به برنامه‌ریزی خطی و در نهایت استفاده از تکنیک **primal-dual** برای حل مساله خود بپردازیم. برای یادگیری ساده‌تر این کار به بیان چند مساله معروف و طرز مدل‌کردن آنها به LP خواهیم پرداخت.

کوتاه‌ترین مسیر: 🛣️

باتوجه به سوال که کوتاه‌ترین مسیر از هر راس به راس مبدا ما می‌باشد به‌نظر می‌آید که می‌توانیم مساله کوتاهترین مسیر را به صورت یک مساله مینیموم‌سازی مدل کنیم. این مساله را به‌خاطر بیاورید، هدف ما پیدا کردن یک راه برای مدل کردن این سوال با یک سری شرط خطی و در نهایت یک تابع هدف به صورت مینیموم یا ماکسیموم است.

اولین شروطی که به ذهن می‌آیند چه هستند؟ در اکثر روش‌هایی که تا اینجا برای حل این سوال به‌کار می‌بردیم چند قانون به‌وضوح قابل لمس بود. برای مثال در الگوریتم **dijkstra** ما ابتدا مقدار فاصله هر راس از راس مبدا ما (S) را برابر بی‌نهایت مقدارگذاری می‌کردیم و با انجام تکراری بررسی یک سری شرط به نتیجه مد نظر که کوتاه‌ترین مسیر بود می‌رسیدیم.

فاصله راس V از راس S را به تابعی نمایش دهیم:

$$d[v]$$

یک شرط مهم که در اکثر راه‌حل‌های ما نیز کاربرد اساسی داشت، نامساوی مثلث بود:

$$d[v] - d[u] \leq w(u, v), \forall (u, v) \in E$$

به‌نظر می‌رسد مساله ما به درستی مدل شد:

$$\text{Obj Function: } \min \sum_{v \in V} d[v]$$

$$s.t: \ d[v] - d[u] \leq w(u,v), \forall (u,v) \in E$$

$$d[s] = 0$$

$$d[v] \geq 0, \forall v \in V$$

مشکل: یک جواب قابل قبول:

$$d[v] = 0, \forall v \in V$$

با این مدل‌سازی یک شرط مهم و اساسی را زیر پا می‌گذاریم: یافتن مسیر کمینه. برای درست شدن مدل‌سازی کافی است تابع هدف خود را به ماکسیموم تبدیل کنیم:

$$\text{Obj Function: } \max \sum_{v \in V} d[v]$$

شار بیشینه: 💧

برای این مساله وارد جزئیات نمی‌شویم، اما برای مدل کردن این مسئله به یک LP باید طبق روال محدودیت‌ها و شروط دستگاه را پیدا کنیم. تابع هدف: بیشینه کردن مجموع شار عبوری از راس مبدا (s)

$$\max \sum_{v \in V} f(s,v)$$

شروط موجود باید به‌صورت خطی بیان شوند: - شار عبوری از هر یال از ظرفیت ماکسیموم آن کمتر است:

$$f(u,v) \leq c(u,v), \forall v \in V$$

- شار عبوری از راس v به راس u برابر قرینه شار عبوری از راس u به راس v است:

$$f(u,v) = f(v,u), \forall u,v \in V$$

- برای هر راس جز راس مبدا و نهایی مجموع شار ورودی برابر مجموع شار خروجی است:

$$\sum_{v \in V} f(u,v) = 0, \forall u \in V - \{s,t\}$$

- و نهایی، همه شارها غیر منفی هستند:

$$f(u,v) \geq 0, \forall (u,v) \in E$$

مدل بالا به فرم استاندارد نیست، اما تبدیل هر دستگاهی به فرم استاندارد قبلاً توضیح داده شد. هدف این بخش نشان دادن قدرت ابزار LP و تکنیک‌های primal-dual برای حل مسائل معروف است.

تئوری‌های کاربردی مفهوم دوگان 🔑

شاید تا کنون سوال برایتان پیش آمده که کاربرد اصلی تعریف دوگان یک عبارت چیست؟ اساساً می‌توان دو کاربرد اصلی برای این تکنیک بیان کرد:

1. خیلی از اوقات ممکن است تبدیل یک مسأله از فرم استاندارد به دوگان و حل حالت دوگان آن بسیار ساده‌تر از حل مسأله اولیه باشد.
2. بیان تساوی دو مسأله معروف به کمک اثبات اینکه دوگان یکدیگرند (مثلاً **Max-flow min-cut theorem**).

قضیه اول: اگر x برای مسأله استاندارد اولیه دارای جواب باشد و y نیز برای حالت دوگان دارای جواب (feasible) باشد آنگاه:

$$c^T x \leq y^T b$$

اثبات:

$$c^T x \leq y^T A x \leq y^T b$$

طرف اول نامعادله از این می‌آید:

$$x \geq 0, \quad c^T \leq y^T A$$

و نیمه دوم نیز:

$$y \geq 0, \quad Ax \leq b$$

نتیجه اول: اگر مسأله استاندارد و دوگان آن هر دو امکان‌پذیر (feasible) باشند، آنگاه هر دو کراندار (bounded) نیز هستند.

نتیجه دوم: اگر یک جواب مانند

$$x^*$$

برای مسأله اصلی و

$$y^*$$

برای مسأله دوگان وجود داشته باشد که

$$c^T x^* = y^* T b$$

آنگاه هر دو جواب بهینه مسأله خود خواهند بود.

اثبات: اگر x یک بردار شدنی برای مسأله استاندارد باشد، آنگاه داریم:

$$c^T x \leq y^* T b = c^T x^*$$

نشان می‌دهد که

$$x^*$$

پاسخ بهینه می‌باشد. (به‌تقارن برای

$$y^*$$

نیز همین است.)

قضیه دوم (The Duality Theorem): اگر یک برنامه‌ریزی خطی استاندارد کراندار و شدنی باشد، آنگاه دوگان آن نیز کراندار و شدنی است و مقادیر بهینه آنها حتماً وجود خواهد داشت و با هم برابرند.

با کمک این دو قضیه و نتایج آنها، به یک نتیجه بسیار مهم می‌رسیم: برای هر مسأله برنامه‌ریزی خطی سه حالت وجود دارد:

1. کراندار و شدنی
2. بی‌کران و شدنی
3. ناشدنی

حال، با کمک دو قضیه پیشین ثابت می‌شود که از ۹ حالت ممکن میان یک مسأله و دوگان آن تنها ۴ فرم امکان‌پذیر خواهد بود:

- هر دو کراندار و شدنی باشند
- اگر حالت استاندارد بی‌کران اما شدنی باشد، حالت دوگان آن لزوماً نشدنی خواهد بود
- اگر حالت استاندارد شدنی نباشد، دوگان آن می‌تواند جواب داشته باشد اما آن جواب حتماً بی‌کران است
- اگر حالت استاندارد شدنی نباشد، دوگان آن نیز ممکن است نشدنی باشد

جدا از قضایا بیان شده، قضایا و نتایج بیشتری نیز در این موضوع وجود دارد، اما ما به بیان همین‌ها اکتفا می‌کنیم. با دانستن این موارد، می‌توانید حدس بزنید اثبات برابری دو مسأله **max-flow** و **min-cut** چگونه خواهد بود؟ احتمالا درست حدس زدید:

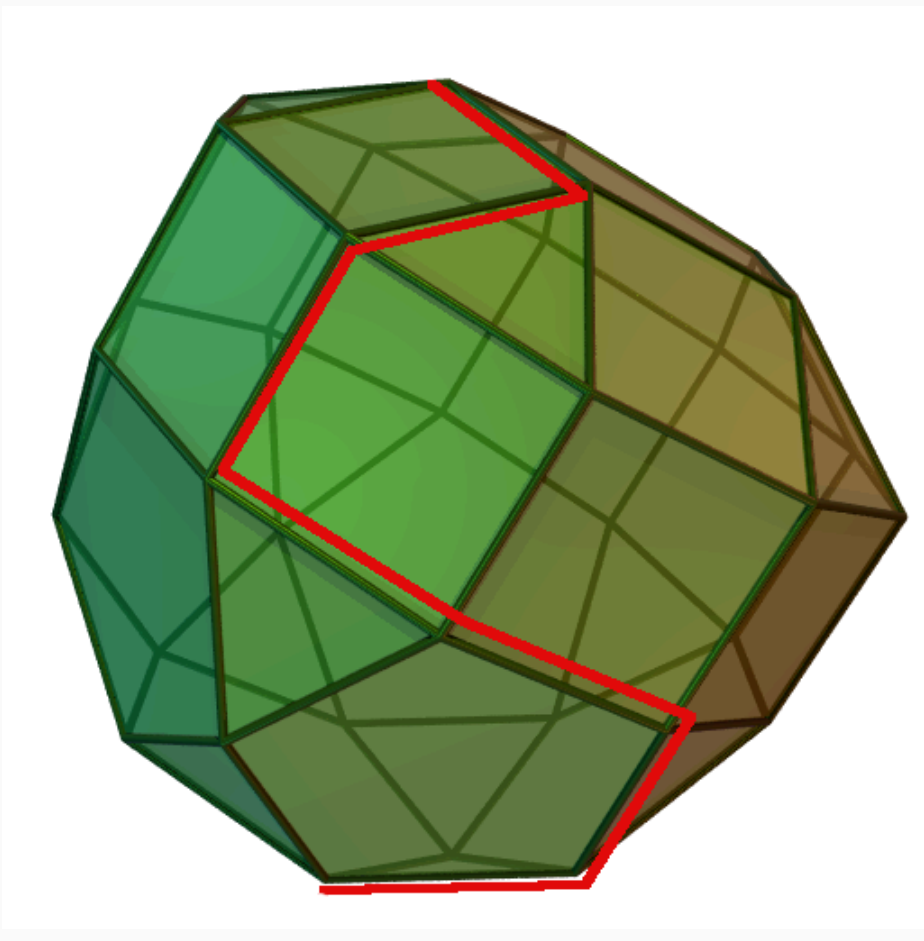
1. قدم اول: باید هر دو را به یک مسأله برنامه‌ریزی خطی مدل کنیم
2. قدم دوم: باید بگوییم این دو مدل بدست آمده دوگان یکدیگر هستند. این مرحله بسیار ساده است چون ضرایب را داریم.
3. قدم سوم: باید از اصل نتیجه‌گیری بالا و قضیه **The Duality Theorem** استفاده کنیم، یعنی ابتدا ثابت می‌کنیم یکی از این دو مسئله (به دلخواه) کراندار و شدنی است و سپس به کمک قضیه دوم ثابت می‌شود دوگان آن نیز کراندار و شدنی است و مقادیر بهینه آنها برابرند.

الگوریتم سیمپلکس

پس از مطرح کردن کلیت برنامه‌ریزی به الگوریتم‌های حل آن می‌رسیم. می‌توان اثبات کرد که یک مسألهٔ برنامه‌ریزی خطی را می‌توان در زمان چندجمله‌ای حل کرد؛ اما در این قسمت ما به معرفی الگوریتم سیمپلکس می‌پردازیم که در worst-case مسأله را در زمان اجرای نمایی حل می‌کند اما در عمل بسیار سریع‌تر عمل می‌کند.

مقدمه‌ای بر کلیت کار سیمپلکس

می‌دانیم که هر مسألهٔ برنامه‌ریزی خطی را می‌توان به صورت پیدا کردن نقطهٔ مینیمم یا ماکسیمم در یک فضا مدل کرد. همچنین می‌دانیم که نقطه بهینه مساله در یکی از نقاط تلاقی ابرصفحه‌ها رخ می‌دهد. در الگوریتم سیمپلکس، نقطهٔ تلاقی یک ابرصفحه را به صورت رندم پیدا می‌کنیم و از آن به سمت راسی که ما را به نقطهٔ بهینه نزدیک‌تر می‌کند حرکت می‌کنیم. شکل زیر کلیت کار این الگوریتم را نشان می‌دهد:



نقطهٔ بهینه در شکل در مینیمم محور y مختصات قرار دارد و در نتیجه هر بار الگوریتم به سمت نقطهٔ پایین‌تر از نقطهٔ فعلی حرکت می‌کند تا زمانی که در نقطهٔ مینیمم محلی گیر کند؛ از آن‌جایی که با تابعی محدب سر و کار داریم، مینیمم محلی و کلی یک مقدار دارند و در این حالت الگوریتم به پایان می‌رسد.

شبه‌کد الگوریتم سیمپلکس

تا این‌جای کار با تصور هندسی الگوریتم سیمپلکس و کلیت آن آشنا شدیم. در ادامه باید روش عملی برای پیدا کردن نقطهٔ بهینه ارائه دهیم. برای راحتی کار، برای هر مسألهٔ برنامه‌ریزی خطی ابتدا آن را به فرم استاندارد تبدیل می‌کنیم و سپس توسط الگوریتم سیمپلکس حل می‌کنیم. با توجه به این فرض، همیشه به دنبال یافتن نقطهٔ مینیمم هستیم و از هر نقطهٔ انتخابی به سمت نقطهٔ پایین‌تر حرکت می‌کنیم و همیشه متغیرها شرط نامنفی بودن را دارند.

```
primal_simplex(H):
if intersect(H) == None
        return "infeasible"

   x = random_feasible_vertex()
   while x is not locally optimal:
       if every feasible neighbor of x is higher than x:
           return "unbounded"
       x = any feasible neighbor of x that is lower than x
   return x
```

In [1]:

```python
# Real code (Optional!)

# پیاده‌سازی ساده الگوریتم سیمپلکس اولیه برای حل مسائل برنامه‌ریزی خطی
# به فرم استاندارد داده شده است H این نسخه فرض می‌کند که ماتریس اولیه

def primal_simplex(H, b, c):
    n = len(c)  # تعداد متغیرها
    m = len(b)  # تعداد محدودیت‌ها

    # ساخت جدول اولیه سیمپلکس
    tableau = []
    for i in range(m):
        row = H[i][:] + [0] * m + [b[i]]
        row[n + i] = 1  # افزودن متغیرهای کمکی (اسلک)
        tableau.append(row)

    # افزودن تابع هدف به انتهای جدول
    objective = [-ci for ci in c] + [0] * (m + 1)
    tableau.append(objective)

    while True:
        # پیدا کردن ستون ورودی (ستونی با بیشترین مقدار منفی در تابع هدف)
        pivot_col = -1
        min_value = 0
        for j in range(n + m):
            if tableau[-1][j] < min_value:
                min_value = tableau[-1][j]
                pivot_col = j

        if pivot_col == -1:
            # اگر هیچ مقدار منفی وجود ندارد، بهینه شده‌ایم
            break

        # پیدا کردن سطر محوری با استفاده از نسبت RHS به مقدار ستون محوری
        pivot_row = -1
        min_ratio = float('inf')
        for i in range(m):
            if tableau[i][pivot_col] > 0:
                ratio = tableau[i][-1] / tableau[i][pivot_col]
                if ratio < min_ratio:
                    min_ratio = ratio
                    pivot_row = i

        if pivot_row == -1:
            # اگر هیچ سطر مناسبی پیدا نشد، مسأله نامحدود است
            return "unbounded"

        # عملیات محور کردن: نرمال‌سازی سطر محوری
        pivot_val = tableau[pivot_row][pivot_col]
        for j in range(n + m + 1):
            tableau[pivot_row][j] /= pivot_val

        # صفر کردن سایر سطرها در ستون محوری
        for i in range(m + 1):
            if i != pivot_row:
                factor = tableau[i][pivot_col]
                for j in range(n + m + 1):
                    tableau[i][j] -= factor * tableau[pivot_row][j]

    # استخراج مقدار بهینه از ستون آخر تابع هدف
    return tableau[-1][-1]
```

شبه‌کد الگوریتم سیمپلکس (جزئیات) ⚙️

این شبه‌کد در ابتدا چک می‌کند که فضای محدودیت‌های داده‌شده ناتهی باشد و اشتراک محدودیت‌ها فضای ناتهی بسازد. در غیر این صورت، این دستگاه برنامه‌ریزی خطی پاسخ **feasible** ندارد. در صورتی که اشتراک ناتهی بود، راسی را به صورت رندم انتخاب می‌کند و مادامی که راسی پایین‌تر از راس فعلی وجود داشته باشد به آن راس می‌رود. در صورتی که چنین راسی وجود نداشته باشد، دو حالت کلی داریم: یا راس feasible دیگری وجود دارد یا خیر. اگر چنین راسی وجود نداشته باشد، الگوریتم به نقطهٔ کمینه محلی که همان نقطهٔ بهینه است رسیده و پاسخ را برمی‌گرداند. اما در غیر این صورت، دو راس را می‌توان بر روی یک صفحهٔ بهینه تصور کرد که تمام نقاط روی این صفحه پاسخ‌اند و پاسخ **unbounded** است.

محاسبهٔ سیمپلکس به صورت تئوری 🧮

اما وقتی که سیمپلکس را روی کاغذ بیاوریم اوضاع کمی فرق می‌کند. در این حالت دیگر نمی‌شود هر بار فضایی متصور شد و نقطهٔ بهینه را به دست آورد، بلکه باید با استفاده از محاسبات جبری هربار از راسی به راس دیگری برویم تا به نقطهٔ بهینه برسیم. ابتدا تعریف‌های اولیه مورد نیاز برای حل یک مسأله سیمپلکس به صورت تئوری را انجام می‌دهیم و سپس با حل یک مثال، مبحث سیمپلکس را جمع‌بندی خواهیم کرد.

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل نهم: کاهش چندجمله ای

بخش اول: کاهش چندجمله ای

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 🧭

مقدمه 📖

در نظریهٔ محاسبات و نظریهٔ پیچیدگی محاسباتی، کاهش، الگوریتمی برای انتقال یک مسئله به مسئلهٔ دیگری می‌باشد. کاهش از یک مسئله به مسئلهٔ دیگر ممکن است برای نشان دادن اینکه مسئلهٔ دوم حداقل به اندازهٔ مسئلهٔ اول، سخت است استفاده شود. ساختار ریاضی ایجاد شده در مجموعه‌ای از مسائل با کاهش یک نوع خاص، عموماً یک مرتبهٔ پایین‌تر را تشکیل می‌دهد که کلاس‌های هم‌ارزی آن ممکن است برای تعریف درجهٔ غیرقابل حل بودن و کلاس‌های پیچیدگی استفاده شود. به‌طور شهودی، مسئلهٔ A قابل کاهش به مسئلهٔ B است اگر الگوریتم کارای حل مسئلهٔ B بتواند به عنوان یک زیرروال برای حل کارای مسئلهٔ A نیز استفاده شود. اگر این اتفاق بیفتد، حل A نمی‌تواند از حل B مشکل‌تر باشد.

کاهش چندجمله‌ای 🔽

الگوریتم‌هایی که تا کنون دیده‌اید راه‌هایی کارآمد برای مسائل ارائه می‌دادند؛ اما همیشه این‌طور نیست و بعضی مسائل از این دست الگوریتم‌ها ندارند. منظور ما در اینجا از واژه‌ی **کارآمد**، مفهوم عام آن نیست. شاید از نظر شما الگوریتمی که اردر زمانی `$n^{10}$` دارد، الگوریتم کارآمدی نباشد. اما در دسته‌بندی که در ادامه درس آن را به‌طور دقیق‌تر یاد خواهید گرفت، این الگوریتم نیز کارآمد خواهد بود.

در واقع در این دسته‌بندی، اگر الگوریتمی از **O(P(n))** باشد که **P(n)** به معنای عبارتی چندجمله‌ای بر حسب n است، الگوریتمی کارآمد حساب می‌شود و به مسائلی که با این دست الگوریتم‌ها حل می‌شوند، **مسائل نرم (tractable)** می‌گویند. این نوع دسته‌بندی در نهایت منتهی خواهد شد به دو دسته: **P** (دسته‌ای که با الگوریتم‌های چندجمله‌ای حل می‌شوند) و **NP** که تعریف دقیق‌تر آن‌ها را در ادامه درس خواهید دید.

قسمتی از علوم کامپیوتر به این می‌پردازد که چه مسائلی در دسته‌ی P قرار می‌گیرند. راهی که می‌توان برای پیدا کردن این دست مسائل ارائه داد، این است که یک مسئله با چنین الگوریتمی پیدا کنیم و به طریقی یک مسئله‌ی جدید را به مسئله‌ی قبلی و الگوریتم چندجمله‌ای آن مربوط کنیم.

دلیل اینکه این بحث در این قسمت انجام شد، این است که با انگیزه‌ی مطرح شدن مبحث این جلسه آشنا شده باشید؛ که آن هم ارائه‌ی راهی است برای ایجاد ارتباط بین مسائلی که در پاراگراف قبل به آن‌ها اشاره کردیم.

مسائل تصمیمی ❓

در این مبحث مسائلی را در نظر می‌گیریم که از نوع **تصمیمی (decision problem)** هستند. این دسته از مسائل به مسائلی گفته می‌شوند که پاسخ به آن‌ها با **آری** یا **نه** صورت می‌گیرد. همچنین می‌توان مسائلی را که از نوع تصمیمی نیستند، به این نوع مسائل تبدیل کرد.

برای مثال، مسئله‌ی **بیشترین درجه‌ی یک گراف** را می‌توان به‌صورت تصمیمی به این شکل بیان کرد که:

آیا رأسی با درجه بیشتر از k وجود دارد؟

مسائل تشخیص زبان 🧩

مسائل تصمیمی را می‌توان به شکل **مسائل تشخیص زبان (language-recognition-problem)** مطرح کرد. فرض کنید **U** مجموعه‌ی تمامی ورودی‌های ممکن برای مسئله باشد و $L \subset U$ مجموعه‌ی تمامی ورودی‌هایی باشد که جواب مسئله به آن‌ها "بله" است. حال مسئله به این تبدیل می‌شود که بفهمیم ورودی عضو **L** هست یا نه. به **L زبان مسئله** می‌گوییم.

حال فرض کنید **L_1** و **L_2** زبان دو فضای **U_1** و **U_2** باشند. در این صورت می‌گوییم **L_1** به **L_2** در زمان چندجمله‌ای کاهش‌پذیر است وقتی که الگوریتمی با زمان چندجمله‌ای موجود باشد که هر ورودی $u_1 \subset U_1$ را به ورودی $u_2 \subset U_2$ تبدیل کند به گونه‌ای که $u_1 \subset L_1$ باشد اگر و تنها اگر $u_2 \subset L_2$ باشد. الگوریتم بر اساس اندازه‌ی **u_1** چندجمله‌ای است. فرض می‌کنیم که تعریف اندازه در **U_1** و **U_2** خوش‌تعریف است. پس **u_2** نیز بر اساس اندازه‌ی **u_1** چندجمله‌ای است.

اگر الگوریتمی در زمان چندجمله‌ای برای **L_2** داشته باشیم و **L_1** نیز به **L_2** در زمان چندجمله‌ای کاهش‌پذیر باشد، آنگاه می‌توانیم هر ورودی **L_1** را در زمان چندجمله‌ای به ورودی **L_2** تبدیل کنیم و چون الگوریتم چندجمله‌ای برای حل **L_2** داریم، آن را در زمان چندجمله‌ای حل کنیم. پس با استفاده از دو الگوریتم زمان چندجمله‌ای می‌توانیم **L_1** را حل کنیم که در مجموع الگوریتمی چندجمله‌ای است.

خواص کاهش چندجمله‌ای ⚙️

مفهوم کاهش چندجمله‌ای دوطرفه نیست. یعنی ممکن است L_1 به L_2 در زمان چندجمله‌ای کاهش یابد، اما L_2 به L_1 کاهش نیابد. اگر هم L_1 به L_2 در زمان چندجمله‌ای کاهش یابد و هم L_2 به L_1، آنگاه می‌گوییم L_1 و L_2 به‌صورت چندجمله‌ای، **معادل** هستند.

خاصیت **تعدی** در مفهوم کاهش چندجمله‌ای وجود دارد. به این معنی که اگر L_1 به L_2 در زمان چندجمله‌ای کاهش‌پذیر باشد و L_2 نیز در زمان چندجمله‌ای به L_3 کاهش‌پذیر باشد، آنگاه L_1 به L_3 نیز در زمان چندجمله‌ای کاهش‌پذیر است؛ زیرا ورودی‌های L_1 در زمان چندجمله‌ای به ورودی‌های L_2 تبدیل می‌شوند و ورودی‌های L_2 نیز در زمان چندجمله‌ای به ورودی‌های L_3 تبدیل می‌شوند. پس با اجرای دو الگوریتم چندجمله‌ای می‌توان ورودی‌های L_1 را به ورودی‌های L_3 تبدیل کرد و جمع این دو الگوریتم چندجمله‌ای نیز الگوریتمی چندجمله‌ای به‌دست می‌دهد.

اصولاً به این مفهوم در این قسمت از درس به این منظور پرداخته شده است که وقتی نمی‌توانیم مسئله‌ای را با الگوریتمی کارآمد حل کنیم، به دنبال مسئله‌ای می‌گردیم که به‌صورت چندجمله‌ای **معادل** باشد و آن مسئله، مسئله‌ای **سخت** است. آنگاه می‌توان مسئله‌ی جدید را در دسته‌بندی **مسائل سخت** قرار داد.

روش کار 🧠

ایده‌ی نشان دادن اینکه یک مسئله سخت‌تر یا آسان‌تر از مسئله‌ی دیگری نیست، حتی زمانی که هر دو مسئله، مسئله‌ی تصمیم‌گیری باشند، نیز به کار می‌رود. ما از این ایده تقریباً برای اثبات تمامی مسائل **NPC** استفاده می‌کنیم.

اجازه دهید یک مسئله‌ی تصمیم‌گیری مانند **A** را در نظر بگیریم که می‌خواهیم آن را در پیچیدگی زمانی چندجمله‌ای $O(n^k)$ حل کنیم. ورودی را به یک مسئله‌ی خاص به‌عنوان «نمونه‌ای» از این مسئله فراخوانی می‌کنیم. به‌عنوان مثال، در مسئله‌ی مسیر، یک نمونه می‌تواند یک گراف خاص **G** باشد با دو رأس خاص **u** و **v** از **G** و مقداری مشخص برای پارامتر **k**.

حال فرض می‌کنیم که یک مسئله‌ی تصمیم‌گیری دیگر داریم با نام **B** و می‌دانیم که این مسئله راه‌حلی در پیچیدگی زمانی چندجمله‌ای دارد. در نهایت فرض می‌کنیم رویه‌ای داریم که هر یک از نمونه‌های **a** از مسئله‌ی **A** را به نمونه‌های **b** از مسئله‌ی **B** با خصوصیات زیر تبدیل می‌کند یا «کاهش می‌دهد»:

1. عملیات کاهش دارای پیچیدگی زمانی چندجمله‌ای است.
2. جواب‌های هر دو مسئله یکسان است؛ یعنی جواب مسئله‌ی **a** «بلی» است اگر و فقط اگر جواب مسئله‌ی **b** «بلی» باشد.

پس کلاً سه مرحله داریم:

1. نمونه‌ی **a** از مسئله‌ی **A** را با استفاده از «الگوریتم کاهشی» با پیچیدگی زمانی چندجمله‌ای به نمونه‌ی **b** از مسئله‌ی **B** تغییر شکل می‌دهیم.
2. الگوریتم کاهشی را بر روی نمونه‌ی **b** از مسئله‌ی **B** اجرا می‌کنیم.
3. از جوابی که برای **b** به دست آورده‌ایم به‌عنوان جوابی برای مسئله‌ی **a** استفاده می‌کنیم.

مثال 📘

فرض کنید که **H(M, w)** مسئلهٔ تشخیص این باشد که آیا یک ماشین تورینگ **M** در رشتهٔ ورودی **w** توقف می‌کند یا نه. این زبان به عنوان یک **زبان تصمیم‌ناپذیر** شناخته می‌شود.

همچنین فرض کنید **E(M)** مسئلهٔ تشخیص این باشد که آیا زبان پذیرش‌های یک ماشین تورینگ داده شده **M** تهی است یا نه. نشان می‌دهیم که **E** توسط کاهش از **H** تصمیم‌ناپذیر است.

برای ایجاد تناقض، فرض کنید که **R** یک تصمیم‌گیرنده برای **E** باشد. ما از این تصمیم‌گیرنده برای تولید تصمیم‌گیرندهٔ **S** برای **H** استفاده می‌کنیم.

اگر ورودی‌های **M** و **w** داده شده باشند، **S(M, w)** را به این صورت تعریف می‌کنیم: **S** ماشین تورینگ **N** را می‌سازد که فقط در حالتی قبول می‌کند که رشتهٔ ورودی به **N**، همان **w** باشد و **M** در ورودی **w** متوقف شود؛ و در غیر این صورت متوقف نشود.

اکنون تصمیم‌گیرندهٔ **S** می‌تواند **R(N)** را ارزیابی کند تا ببیند آیا زبان پذیرفته‌شده توسط **N** تهی است یا نه.

- اگر **R**، **N** را **بپذیرد**، آنگاه زبان پذیرفته‌شده توسط **N** تهی است؛ بنابراین **M** در ورودی **w** توقف نمی‌کند، و **S** می‌تواند **رد کند**.
- اگر **R**، **N** را **رد کند**، آنگاه زبان پذیرفته‌شده توسط **N** ناتهی است؛ بنابراین **M** در ورودی **w** توقف می‌کند، و **S** می‌تواند **بپذیرد**.

پس اگر ما تصمیم‌گیرندهٔ **R** را برای **E** داشته باشیم، قادر خواهیم بود تصمیم‌گیرندهٔ **S** را برای مسئلهٔ توقف **H(M, w)** برای هر ماشین **M** و ورودی **w** تولید کنیم. از آنجا که می‌دانیم چنین **S**ی وجود ندارد، پس زبان **E** نیز **تصمیم‌ناپذیر** است. ❌

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل نهم: NP و NP-Complete

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 📑

مقدمه 🎯

در بخش قبل با **مسئله‌ی تصمیم** آشنا شدیم و دانستیم که مجموعه‌ی مسائل چندجمله‌ای (**P**) به دسته‌ای از مسائل تصمیم گفته می‌شود که الگوریتمی چندجمله‌ای برای حل آن‌ها وجود دارد. مثلاً تشخیص **اول بودن یک عدد** یا **وجود یک مسیر بین دو رأس مشخص از گراف** از این دسته مسائل محسوب می‌شوند.

در این قسمت با مجموعه‌ی بزرگ‌تری از مسائل موسوم به **NP** آشنا می‌شویم و مجموعه مسائل **NP-Complete** را معرفی می‌کنیم.

NP 🧩

مجموعه‌ای از **مسائل تصمیم** هستند که برای هر نمونه‌ای (**instance**) از مسئله‌ی تصمیم که خروجی آن **"بله"** باشد، یک **تصدیق‌کننده** از مرتبه‌ی زمانی چندجمله‌ای وجود داشته باشد.

تعریف فوق کمی گنگ به نظر می‌رسد! در نتیجه با مثالی آن را روشن می‌کنیم.

مسئله‌ی **یافتن مسیر همیلتونی** در یک گراف **n** راسی را در نظر بگیرید. این مسئله وجود مسیری را در گراف بررسی می‌کند که از تمامی رأس‌ها دقیقاً یک‌بار عبور کرده باشد. حال گراف **n** راسی داده‌شده به عنوان **نمونه (instance)** در این مسئله مطرح است.

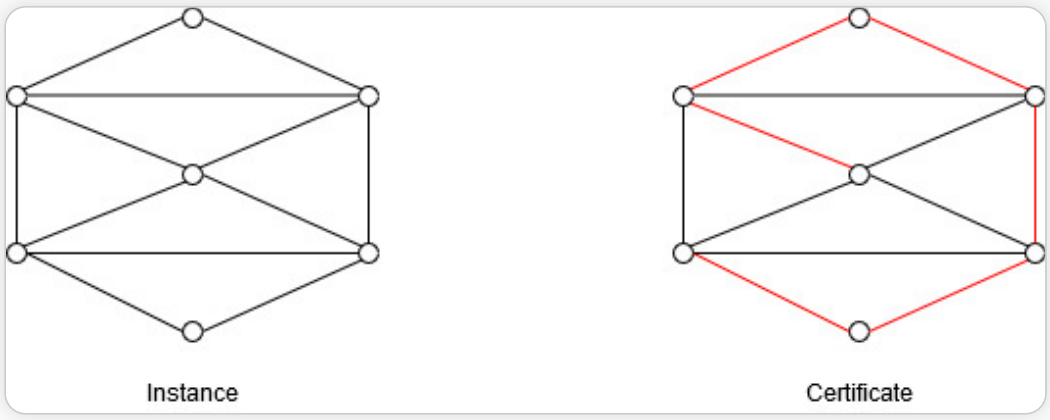
یک **گواهی (certificate)** برای این نمونه، دنباله‌ای متشکل از **n** رأس است که ادعا می‌شود یک مسیر همیلتونی را تشکیل می‌دهند. و **تصدیق‌کننده (verifier)** در این مسئله می‌تواند الگوریتمی باشد که بررسی کند:

- هر رأس فقط یک‌بار در دنباله آمده باشد،
- و بین هر دو رأس متوالی از دنباله، یال موجود باشد.

در شکل زیر یک نمونه و گواهی ارائه‌شده برای آن نمونه را می‌توانید مشاهده کنید. (یال‌های قرمز مسیر ارائه‌شده به عنوان گواهی را نشان می‌دهند.)

نمایش مسیر همیلتونی در گراف 🖼️

در تصویر زیر، مسیر قرمز نشان‌دهنده‌ی گواهی (certificate) برای نمونه‌ی مسئله‌ی همیلتونی است.



P در مقابل NP ⚖️

با تعریف دسته‌ی مسائل **P** و دسته‌ی مسائل **NP** آشنا شدیم. حال می‌خواهیم بررسی کنیم که بین این دو مجموعه چه ارتباطی برقرار است. آیا این دو مجموعه با هم برابرند؟

می‌توان نشان داد که $P \subseteq NP$، زیرا برای هر مسئله‌ای مانند X که $X \in P$، می‌توان **الگوریتم حل مسئله** را که از مرتبه‌ی زمانی چندجمله‌ای است، به عنوان **تصدیق‌کننده** ارائه داد. پس نتیجه می‌گیریم $X \in NP$.

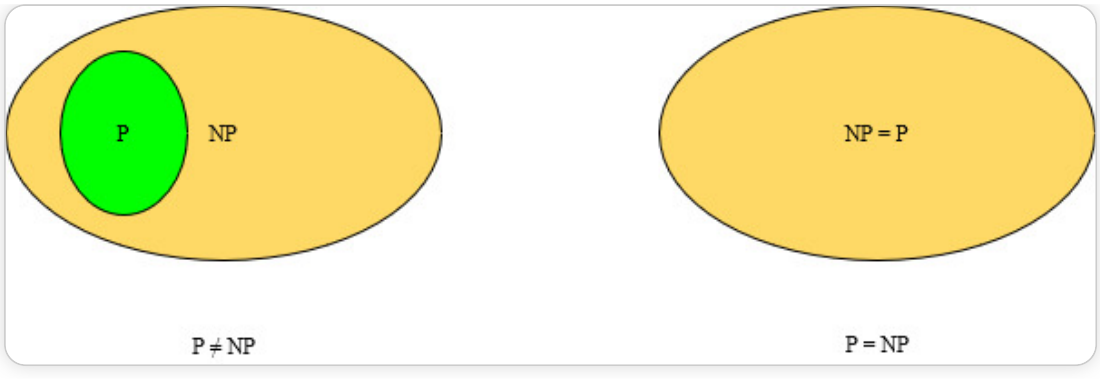
اما آیا برعکس این رابطه هم صادق است؟ یعنی می‌توان نتیجه گرفت که $NP \subseteq P$ و در نتیجه $P = NP$؟

تا کنون هیچ اثبات قابل قبولی برای درست بودن یا نبودن این حکم ارائه نشده است. یعنی هنوز معلوم نیست که اگر مسئله‌ای جزو مسائل **NP** باشد، الزاماً الگوریتمی از مرتبه‌ی زمانی چندجمله‌ای برای حل آن وجود دارد یا نه.

برای اثبات درست بودن یا نبودن این رابطه، **جایزه‌ی میلیون دلاری** در نظر گرفته شده است! 🏆 برای اطلاعات بیشتر می‌توانید **اینجا** را ببینید.

بررسی رابطه‌ی بین مجموعه‌های P و NP 🔍

در شکل زیر، دو وضعیت گفته‌شده برای دو مجموعه‌ی **P** و **NP** را مشاهده می‌کنید.



⚙️ NP-Complete

مسئله **Y** ∈ **NP** یک مسئله‌ی **NP-complete** است هرگاه به ازای هر مسئله‌ی **X** ∈ **NP**، مسئله‌ی **X** قابل کاهش به مسئله‌ی **Y** باشد. (یعنی: $X \leq_p Y$)

تعریف فوق نشان می‌دهد که برای اثبات اینکه یک مسئله **NP-complete** است باید دو گام طی شود:

1. نشان دهیم که مسئله، عضوی از **NP** است.
2. نشان دهیم که مسئله، **NP-hard** نیز هست.

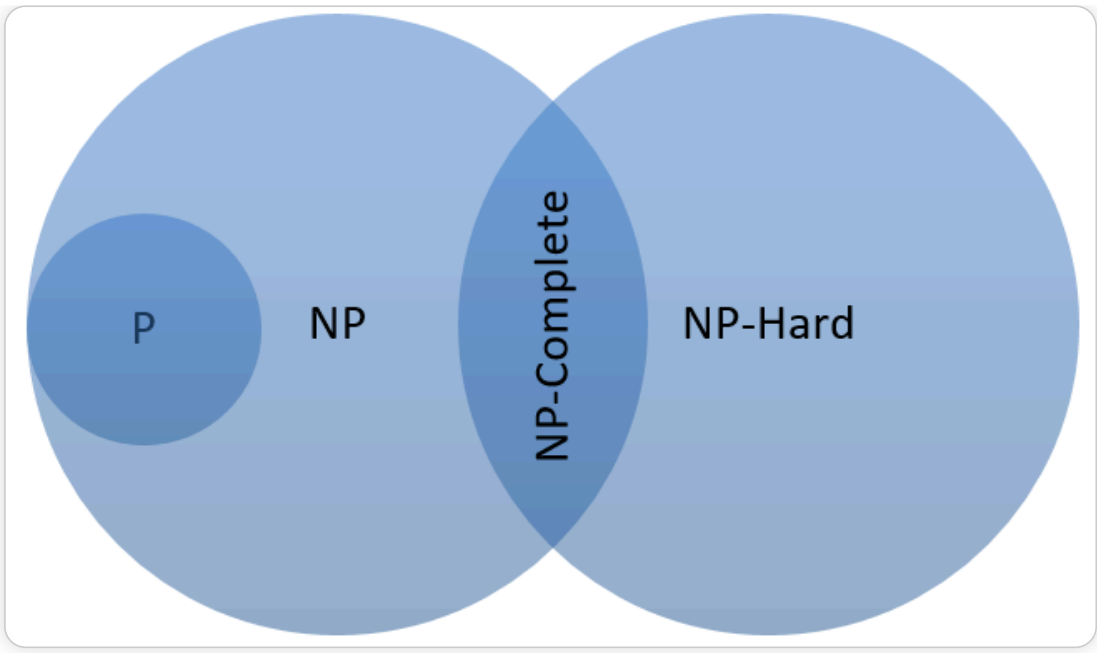
💡 **تعریف:** مسئله‌ی **Y** یک مسئله‌ی **NP-hard** است هرگاه برای هر مسئله‌ی **X** ∈ **NP**، **X** در زمان چندجمله‌ای قابل کاهش به **Y** باشد.

توجه کنید که یک مسئله می‌تواند **NP-hard** باشد ولی در **NP** نباشد، زیرا ممکن است تصدیق‌کننده‌ای از مرتبه‌ی زمانی چندجمله‌ای برای آن وجود نداشته باشد. برعکس، ممکن است مسئله‌ای در **NP** باشد ولی **NP-hard** نباشد.

در واقع هنوز مشخص نیست که مجموعه‌ی مسائل **NP** زیرمجموعه‌ی مسائل **NP-hard** هستند یا نه. اما می‌دانیم که این دو مجموعه اشتراک دارند و آن اشتراک همان مجموعه‌ی **NP-complete** است. اگر روزی نشان دهیم **NP-hard** ⊆ **NP**، آنگاه نتیجه خواهیم گرفت که **NP = NP-complete** — هرچند هنوز اثباتی قطعی برای آن وجود ندارد 🤔

🔹 نمایش حالت مفروض مجموعه‌های NP، NP-hard و NP-complete

در شکل زیر، یکی از حالت‌های مفروض برای این مجموعه مسائل نمایش داده شده است. همان‌طور که ذکر شد، حالت‌های دیگری نیز برای این مجموعه‌ها می‌تواند وجود داشته باشد.



🏁 اولین مسئله NP-complete

قضیه: اگر **X** یک مسئله **NP-complete** و **Y** یک مسئله **NP** باشند، به گونه‌ای که $Y \leq_p X$ برقرار باشد، آنگاه **Y** نیز یک مسئله **NP-complete** است.

اثبات: فرض کنید **L** یک مسئله **NP** دلخواه باشد. از آنجا که **X** یک مسئله **NP-complete** است، باید **NP-hard** هم باشد، و بنابراین $L \leq_p X$. حال با توجه به رابطه‌ی $X \leq_p Y$ و بنا به خاصیت تعدی در کاهش‌پذیری، نتیجه می‌گیریم $L \leq_p Y$. بنابراین، مسئله **Y** باید **NP-hard** باشد و با توجه به **NP** بودن **Y** در فرض، حکم اثبات می‌شود.

بنا بر قضیه فوق، برای اثبات **NP-complete** بودن یک مسئله **Y** کافی است نشان دهیم که **Y ∈ NP** و برای یک مسئله‌ی دلخواه **X** که می‌دانیم **NP-complete** است، **$X \leq_P Y$** برقرار باشد. بنابراین کافی است یک مسئله را به عنوان اولین مسئله **NP-complete** اثبات کنیم و باقی مسائل با همین روش قابل اثبات خواهند بود.

از مسئله **SAT** به عنوان اولین مسئله **NP-complete** یاد می‌شود. این مسئله بررسی می‌کند که آیا در یک عبارت متشکل از عملگرهای **and**، **or** و **not** و متغیرهای **x_1** تا **x_n**، می‌توان این متغیرها را به گونه‌ای مقداردهی کرد که خروجی عبارت **true** شود یا نه. اثبات **NP-complete** بودن این مسئله نخستین بار در سال 1971 توسط استفن کوک ارائه شد.

برای اثبات **NP-complete** بودن سایر مسائل، می‌توان از قضیه ابتدای این بخش و این نکته که **SAT** یک مسئله **NP-complete** است استفاده کرد. توجه کنید که هر مسئله‌ای که **NP-complete** بودن آن اثبات شود، می‌تواند برای اثبات مسائل دیگر مورد استفاده قرار گیرد.

(اثبات **NP-complete** بودن مسئله **SAT** را به عنوان مطالعه‌ی بیشتر می‌توانید در اینجا مشاهده کنید.)

🔹 مثال: NP-Complete بودن 3-SAT

برای اثبات این مسئله از قضیه بیان شده در بخش قبل استفاده می‌کنیم. فرض کنید که فقط می‌دانیم مسئله **SAT** یک مسئله **NP-complete** است و همچنین می‌دانیم که **SAT ∈ NP-3** (چرا؟ 🤔). بنابراین کافی است بررسی کنیم که آیا می‌توان **SAT** را به **SAT-3** کاهش داد یا نه.

اگر بتوانیم **SAT** را به **SAT-3** کاهش دهیم، طبق قضیه، NP-complete بودن **SAT-3** اثبات می‌شود ✅.

مسئله **SAT** را به شکل **AND** یک سری عبارات که فقط شامل **OR** و **NOT** هستند در نظر بگیرید. یک عبارت دلخواه را به شکل $l_1 \vee l_2 \vee \cdots \vee l_m$ فرض کنید. برای تبدیل آن به **SAT-3**، عبارت بالا را به $m-2$ جمله به شکل زیر تقسیم می‌کنیم:

$$(l_1 \vee l_2 \vee x_1) \wedge (\neg x_1 \vee l_2 \vee x_2) \wedge (\neg x_2 \vee l_3 \vee x_3) \wedge \cdots \wedge (\neg x_{m-4} \vee l_{m-2} \vee x_{m-3}) \wedge (\neg x_{m-3} \vee l_{m-1} \vee l_m)$$

در اینجا، $x_1, \ldots, x_{m-3}$ متغیرهای بولی جدید هستند که در جای دیگری استفاده نشده‌اند. مشاهده می‌کنیم که اگر عبارت فوق قابل حل باشد، عبارت $l_1 \vee \cdots \vee l_m$ نیز قابل حل است؛ یعنی این دو مسئله **معادل** هستند 🔗.

علاوه بر این، تولید مسئله **SAT-3** در **زمان چندجمله‌ای** نسبت به تعداد متغیرهای بولی قابل انجام است. بنابراین، با اعمال همین روش برای تمامی عبارات شامل صرفاً **OR** و **NOT** میان **AND**ها، می‌توانیم هر مسئله **SAT** را در زمان چندجمله‌ای به **SAT-3** کاهش دهیم 🛠️.

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل نهم: پیچیدگی محاسبات

بخش سوم: قضیه ی کوک، دور همیلتونی رنگ امیزی گراف

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 🗂️

مقدمه 🧩

بعضی وقت‌ها برای حل کردن یک مسئله راه‌های زیادی وجود دارد.

اول از همه برای خود تعریف کنیم معنی حل مسئله چیست؟ رسیدن به جواب نهایی با حالت اپتیمال حل مسئله نامیده می‌شود.

اکنون که تعریف درستی از حل مسئله داریم، سوال دیگری مطرح می‌شود: اگر چند راه حل داشته باشیم که همه‌ی آن‌ها مسئله‌ی ما را حل کنند، کدام راه حل از همه بهتر است؟

جواب مشخص است: راه حلی که زودتر از همه ما را به جواب برساند.

دانشمندان علوم کامپیوتر و ریاضی برای محاسبه و مقایسه سرعت راه حل‌ها از تعریفی به نام **پیچیدگی زمانی** استفاده می‌کنند.

پیچیدگی زمانی حداقل زمان مورد نیاز برای حل مسئله با توجه به اندازه‌ی ورودی‌های مسئله است.

پیچیدگی زمانی ⏱️

فرض کنید به خانه‌ی آخر یک آرایه‌ی خالی می‌خواهید عددی را اضافه کنید. شما تنها لازم است درون خانه‌ی آخر آرایه یک عدد بریزید. بنابراین این کار را با **پیچیدگی زمانی ثابت** 1 انجام می‌دهید که آن را به صورت:

$$O(1)$$

نشان می‌دهند.

حال فرض کنید که می‌خواهید تعداد n عدد را با هم جمع بزنید. رویکرد شما این است که از ابتدا شروع کرده و اعداد را با هم جمع بزنید. باید برای رسیدن به هدف خود تعداد n جمع انجام دهیم، پس پیچیدگی زمانی ما n است:

$$O(n)$$

دیدیم که هردوی این مسائل در زمان چندجمله‌ای حل می‌شوند که به آن‌ها مسائل **polynomial** گفته می‌شود.

اما مسائلی موجود هستند که در زمان چندجمله‌ای حل نمی‌شوند. برای مثال:
فرض کنید تعدادی شهر داریم و هزینه رفتن مستقیم از یکی به دیگری را می‌دانیم. مطلوب است کم‌هزینه‌ترین مسیری که از یک شهر شروع شود و از تمامی شهرها دقیقاً یک‌بار عبور کند و به شهر شروع بازگردد.
برای محاسبه این مقدار باید تمام n! جایگشت شهرها را بدست آوریم.
محاسبه این مقدار یک مسئله **غیر چندجمله‌ای** است و پیچیدگی زمانی این مسئله با تعداد شهر n برابر است:

$$O(n!)$$

به این سری مسائل که در زمان چندجمله‌ای حل نمی‌شوند، مسائل **non-polynomial** گفته می‌شود.

برخی از مسائل **NP** هستند که اگر جواب آن‌ها را داشته باشیم می‌توانیم درستی جواب را در زمان چندجمله‌ای پیدا کنیم. به این مسائل، مسائل **NP-complete** گفته می‌شود.

قضیه‌ی دودویی 🔢

مسئله‌ی صدق‌پذیری دودویی یک سوال می‌پرسد: آیا می‌توان ارزش متغیرهای یک فرمول را به گونه‌ای تعریف کرد که کل گزاره درست باشد؟

سوال را برای مثال زیر حل کنید:

$$A \wedge B$$

جواب سوال بله است. اگر ارزش هر دو متغیر درست باشد، کل گزاره‌ی ما درست است. اما پیچیدگی زمانی این مسئله چیست؟

مسئله SAT و NP-Complete

استفن کوک در سال 1971 اثبات کرد که مسئله‌ی **SAT** اولین مسئله‌ی **NP-Complete** شناخته شده است و تا آن زمان مفهومی برای **NP-Complete** ساخته نشده بود. تئوری کوک این مسئله را مطرح کرد که:

هر مسئله‌ی **NP** می‌تواند در زمان چندجمله‌ای به یک مسئله‌ی **SAT** تبدیل شود.

به عبارت دیگر، برای هر مسئله‌ی تصمیم‌گیری یک ماشین تورینگ غیرقطعی می‌سازیم که بتواند مسئله را در زمان چندجمله‌ای حل کند. سپس ورودی ماشین را با یک عبارت دودویی مدل می‌کنیم که به ما بگوید آیا با این ورودی‌ها مقدار خروجی نهایی بله (T) است یا خیر (F).

پس این ماشین کاهش داده شده به مسئله‌ی **SAT** است که آیا با این ورودی‌ها گزاره‌ی نهایی T است یا F.

اثبات تئوری کوک 🔎

دو راه موجود است که ثابت کنیم مسئله **SAT** غیر چندجمله‌ای است. راه اول، اثبات مستقیم **غیر چندجمله‌ای** بودن است و راه دوم این است که نشان دهیم هر مسئله‌ی **NP** قابل تبدیل به **SAT** است.

راه حل 🛠️

طبق فرض، مسئله‌ی ما می‌تواند با ماشین تورینگ غیرقطعی حل شود:

$$M = (Q, \Sigma, s, F, \delta)$$

$$Q = States\ set$$

$$s \in Q, Starting\ state$$

$$F \subseteq Q, Accepted\ States$$

$$\delta : Q \times \Sigma \times -1, +1$$

$$\Sigma = tape\ cell$$

فرض کنید ماشین ما یک حالت را در زمان P(n) پذیرش یا رد می‌کند که منظور از P یک تابع چندجمله‌ای است. برای هر ورودی I یک عبارت دودویی موجود است که شرط مسئله را ارضا می‌کند اگر و تنها اگر ماشین ما ورودی I را بپذیرد.

عبارت دودویی ما به صورت زیر از متغیرها استفاده می‌کند:

$$q \in Q, -p(n) \leq i \leq p(n), j \in \Sigma, 0 \leq k \leq p(n)$$

مثال تصویری حل مسئله 🖼️

$T_{i,j,k}$	True if tape cell *i* contains symbol *j* at step *k* of the computation.	$O(p(n)^2)$
$H_{i,k}$	True if the *M*'s read/write head is at tape cell *i* at step *k* of the computation.	$O(p(n)^2)$
$Q_{q,k}$	True if *M* is in state *q* at step *k* of the computation.	$O(p(n))$

توضیح: این تصویر روند حل مسئله را نشان می‌دهد و می‌توانید از آن برای درک بهتر مراحل استفاده کنید.

نتیجه‌گیری و اهمیت NP-Completeness 📌

اگر در I مقدار صحیحی موجود باشد برای ماشین ما، پس شرط مسئله را ارضا می‌کند با سه شرط بالا.

بنابر این در زمان چندجمله‌ای تبدیل انجام می‌پذیرد.

نتیجه‌ی اثبات این است که تمام مسائل غیر چندجمله‌ای می‌توانند به یک نمونه مسئله‌ی **SAT** کاهش پیدا کنند و اگر این مسئله در زمان چندجمله‌ای توسط ماشین تورینگ غیرقطعی حل شود، P = NP می‌شود.

اهمیت **NP-completeness** توسط کارپ با مسائل نظریه گراف نشان داده شد.

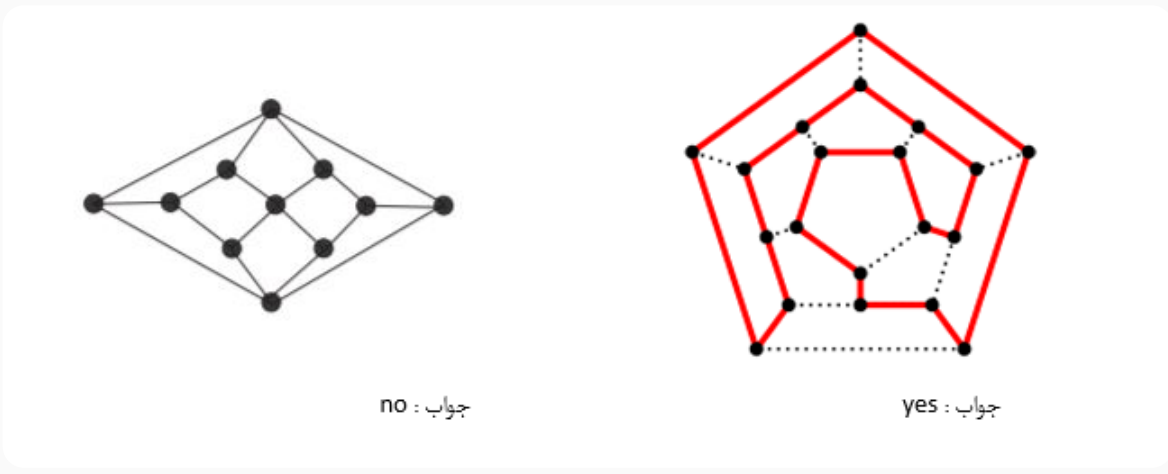
دور همیلتونی 🔄

دور همیلتونی، دوری است که هر راس را دقیقاً یک‌بار مشاهده می‌کند (به جز راسی که هم به عنوان آغاز و هم پایان می‌باشد؛ در نتیجه این راس دو بار دیده می‌شود).

به‌طور قراردادی، گرافی کوچک شامل یک راس، دارای دور همیلتونی است، ولی گراف متصلی دارای دو راس، شامل دور همیلتونی نیست.

مسئله دور همیلتونی 🏁

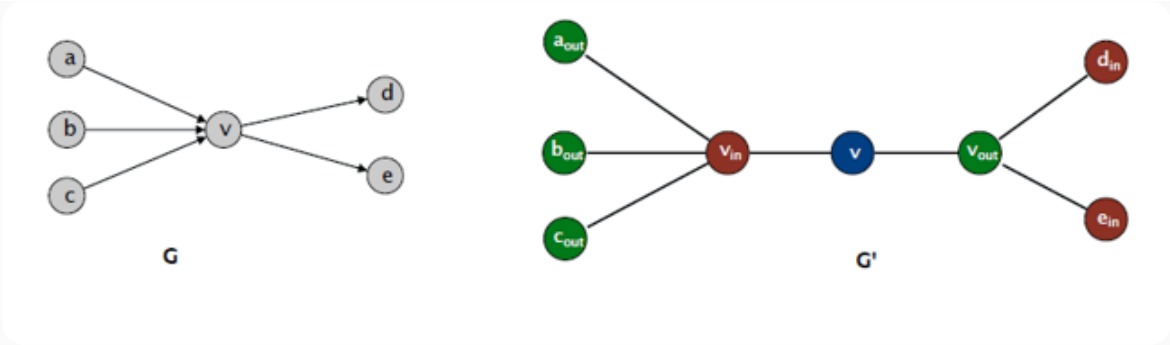
مسئله‌ی دور همیلتونی: گراف غیر جهت‌دار G = (V, E) داده شده است، آیا یک دور ساده‌ی Γ وجود دارد که شامل همه‌ی راس‌های V باشد؟



قضیه: مسئله‌ی دور همیلتونی جهت‌دار به مسئله‌ی دور همیلتونی کاهش می‌یابد. ($dir\_ham\_cycle \leq_p ham\_cycle$)

اثبات:

گراف G با n راس داده شده است. ما گراف G' را به 3n راس مانند تصویر می‌سازیم.



G دور همیلتونی جهت‌دار دارد اگر و فقط اگر G' دور همیلتونی داشته باشد.

⇐ فرض کنید G دور همیلتونی جهت‌دار دارد، آنگاه G' هم دور همیلتونی‌ای با همان ترتیب دارد.

⇒ فرض کنید G' دور همیلتونی غیر جهت‌دار Γ' دارد. آنگاه Γ' باید راس‌های G' را با یکی از دو ترتیب زیر مشاهده کند:

...B, G, R, B, G, R...

...B, R, G, B, R, G...

ترتیب راس‌های آبی یا برعکس این ترتیب در Γ' یک دور همیلتونی جهت‌دار Γ در G می‌سازد.

قضیه: مسئله‌ی 3SAT به مسئله‌ی دور همیلتونی جهت‌دار کاهش می‌یابد. ($3sat \leq_p dir\_ham\_cycle$)

اثبات: فرض کنید ما یک I 3SAT با متغیرهای $x_1, x_2, ..., x_n$ و کلازهای $C_1, C_2, ..., C_k$ داشته باشیم.

حال گراف G را به صورت زیر می‌سازیم:

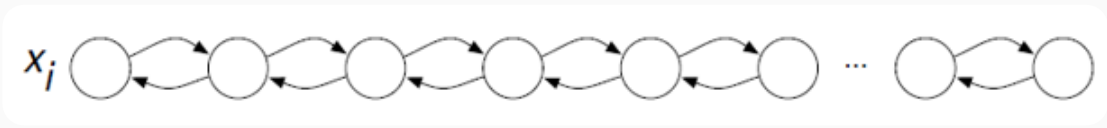
- گراف‌هایی می‌سازیم (که gadget نامیده می‌شوند) که بیانگر متغیرها باشند.

- گراف‌هایی می‌سازیم که بیانگر کلازها باشند.

این گراف‌ها را طوری به هم متصل می‌کنیم که گراف مورد نظر را بسازد.

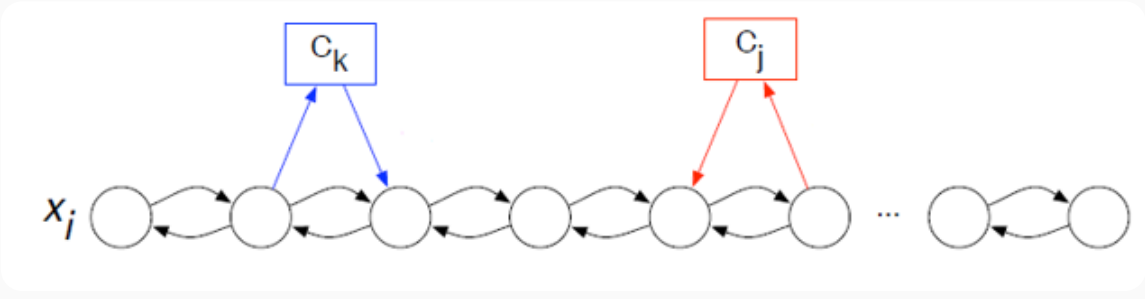
سپس نشان می‌دهیم این گراف دور همیلتونی جهت‌دار دارد اگر و فقط اگر I را satisfy کند.

برای هر متغیر gadgetها را به صورت زیر می‌سازیم:

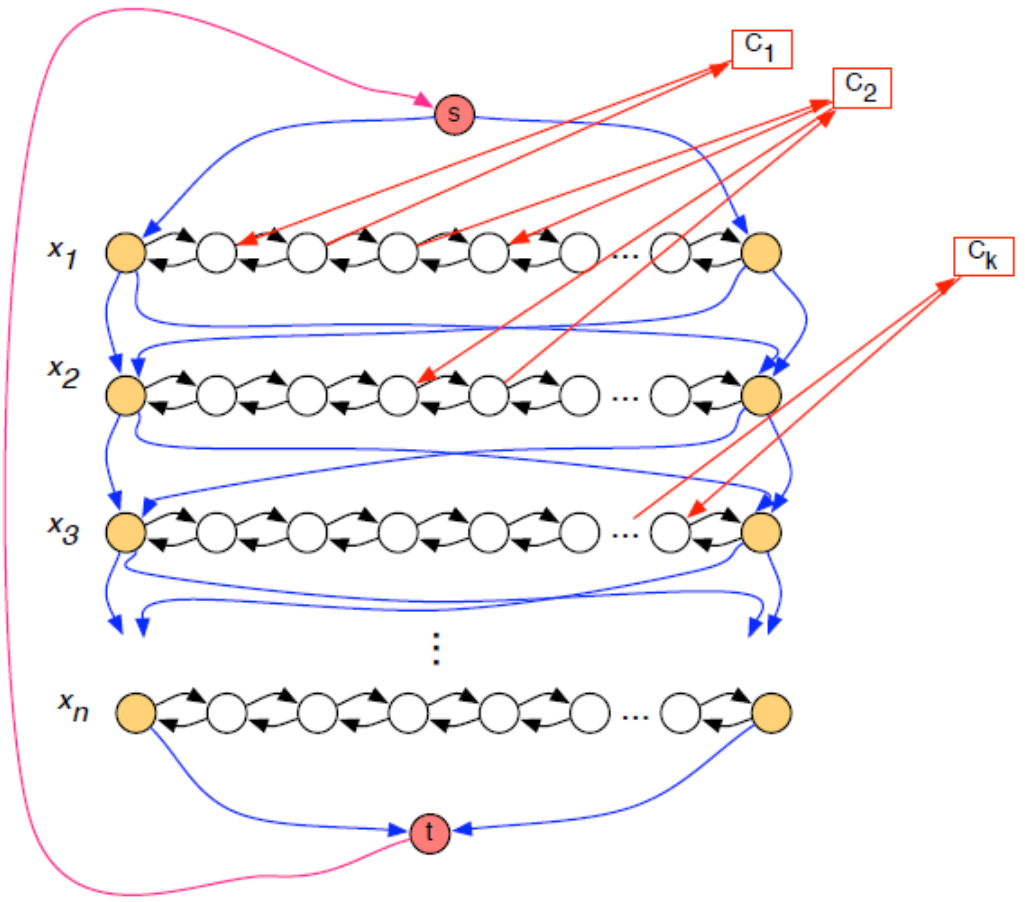


نوع حرکت دور همیلتونی در این gadget بیانگر صحیح یا غلط بودن متغیر x_i است. اگر از سمت راست به چپ باشد، x_i درست است؛ اگر از سمت چپ به راست باشد، x_i غلط است.

حال به ازای هر کلاز مانند زیر عمل می‌کنیم. اگر x_i در کلاز C_j بود، آن را به صورت زیر و اگر x_i! در کلاز C_k بود، آن را مانند زیر متصل می‌کنیم.



حال تمام gadgetها را مانند زیر به یکدیگر متصل می‌کنیم و راس s و t را اضافه می‌کنیم تا گراف G ساخته شود.



با توجه به جهت دور همیلتونی در هر gadget، یک دور همیلتونی برای هر متغیر یک مقدار true یا false در نظر می‌گیرد.
برای آنکه دور همیلتونی باشد، باید تمامی کلازها ملاقات شوند.
تنها در صورتی می‌توانیم یک کلاز را ملاقات کنیم که l را satisfy کنیم.
بنابراین اگر یک دور همیلتونی وجود داشته باشد، یک جواب satisfy برای l وجود دارد.

حال به راحتی می‌توان اثبات کرد که dir_ham_cycle در NP است. (چرا؟)

رنگ‌آمیزی گراف 🎨

در **تئوری گراف**، مسئله‌ای به نام **رنگ‌آمیزی گراف** وجود دارد که حالت خاصی از **لیبل‌گذاری گراف** است.

مسئله‌ی اصلی، **رنگ‌آمیزی رأس‌های گراف** با **حداقل تعداد رنگ** به گونه‌ای است که هر رأس با رأس‌های مجاورش رنگ متفاوت داشته باشد.
این مسئله در این حالت **NP-Complete** است؛ یعنی در زمان چندجمله‌ای پاسخی برای آن موجود نیست.

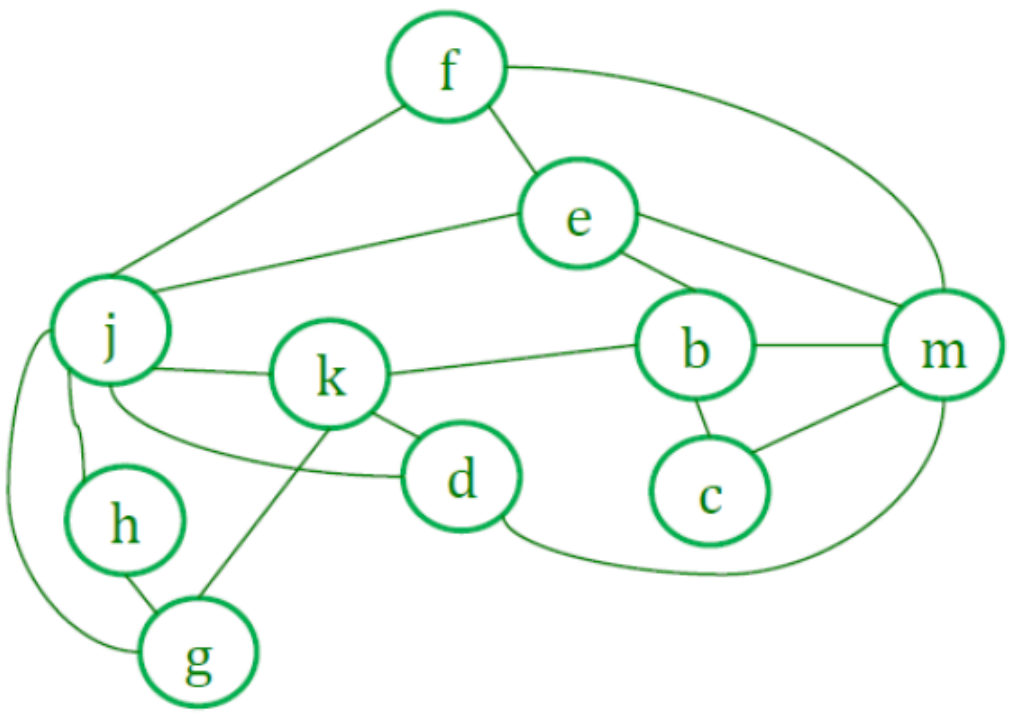
نوع دیگری از این مسئله وجود دارد که در آن با **تعداد محدودی رنگ** باید رأس‌های گراف را به گونه‌ای رنگ‌آمیزی کنیم که هیچ دو رأس مجاوری همرنگ نباشند.
این مسئله نیز از **۳ رنگ به بالا** در دسته‌ی مسائل **NP-Complete** قرار می‌گیرد.

کاهش رنگ‌آمیزی گراف به SAT 🧩

می‌توان این مسئله را به مسئله‌ی **SAT** کاهش داد و کاهش آن متناظر است. برای مثال، 3-GraphColoring با مسئله‌ی 3-SAT متناظر است.

برای حل این مسئله با اندازه‌ی محدود می‌توان از الگوریتم‌های متنوعی استفاده کرد؛ پس سعی می‌کنیم راه‌حلی برای آن بیابیم:

گراف زیر را در نظر بگیرید:



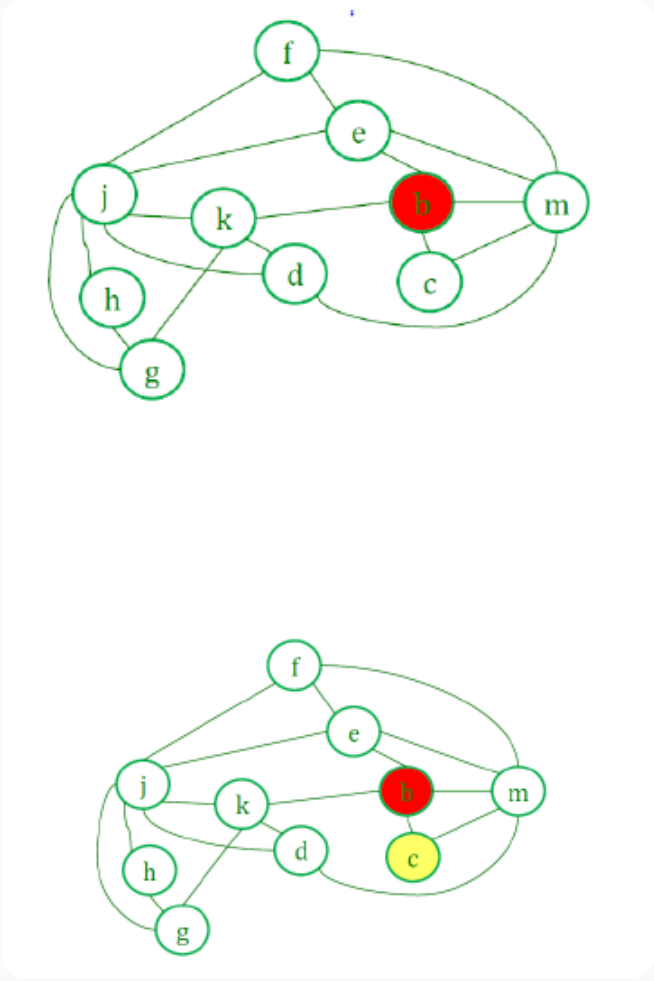
فرض کنید گراف فوق به شما داده شده است و از شما خواسته شده آن را با **سه رنگ** رنگ‌آمیزی کنید.

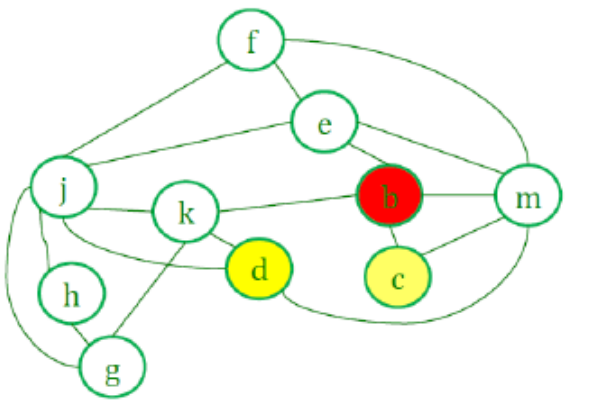
آیا انتخاب تصادفی برای رنگ‌آمیزی انتخاب خوبی است؟ 🎲

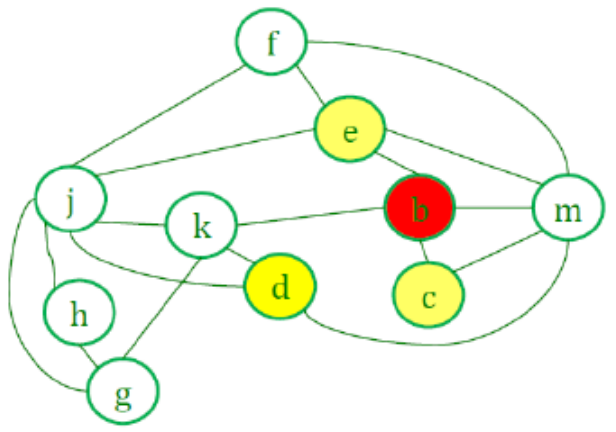
به‌صورت یک الگوی تصادفی رأس‌ها را وارد پشته می‌کنیم:

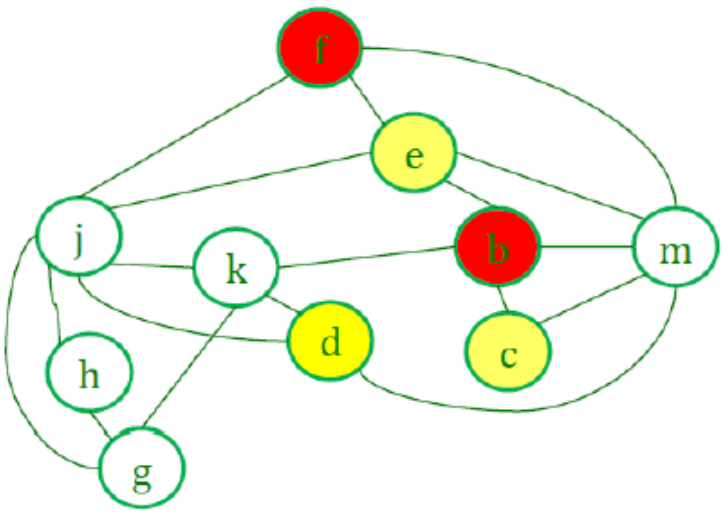
```
Coloring order: b c d e f g h j k m
```

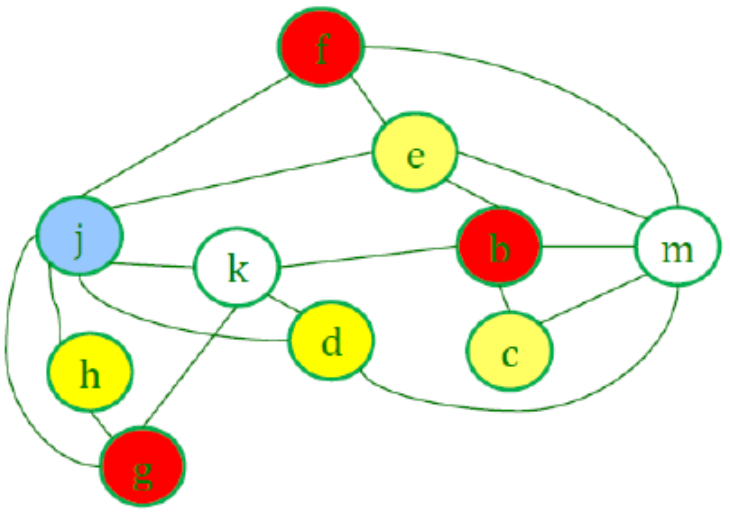
سپس به‌ترتیب رأس‌ها را از پشته خارج کرده و اقدام به رنگ‌آمیزی آن‌ها می‌کنیم:











مشاهده می‌کنیم که با رسیدن به رأس k، حالتی برای رنگ زدن باقی نمی‌ماند؛ بنابراین این الگوریتم نیز به پاسخ نهایی نمی‌رسد.

❌

پس در ادامه، **راه‌حل دیگری** برای این مسئله ارائه خواهیم داد.

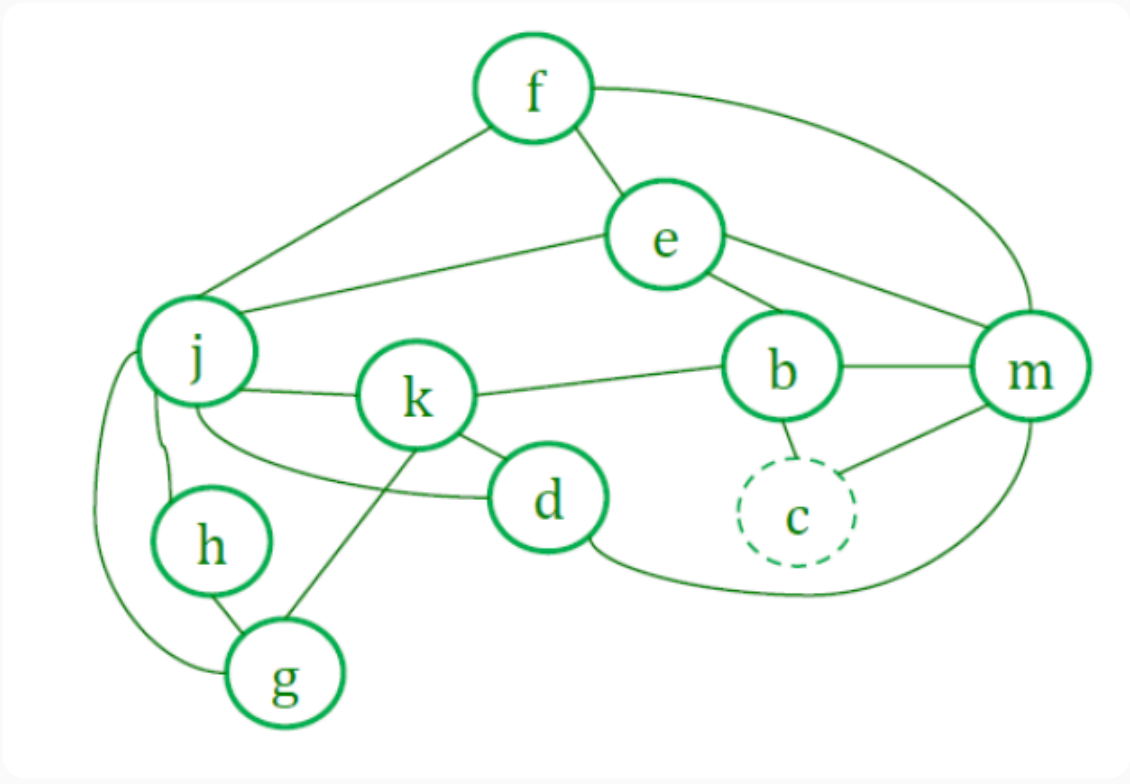
راه‌حل 🎨

در گام نخست برای حل این مسئله، یک **پشته (Stack)** در نظر بگیرید. 🧱

گام دوم: به دنبال رأسی با کوچک‌ترین درجه بگردید.

گام سوم: شرط زیر را بررسی کنید:
آیا درجه‌ی رأس انتخابی کوچک‌تر از ۳ است؟ 🤔

در صورت مثبت بودن جواب، رأس را به همراه یال‌هایش از گراف حذف کرده و شماره‌ی آن را در پشته قرار دهید:

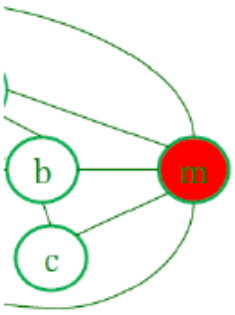


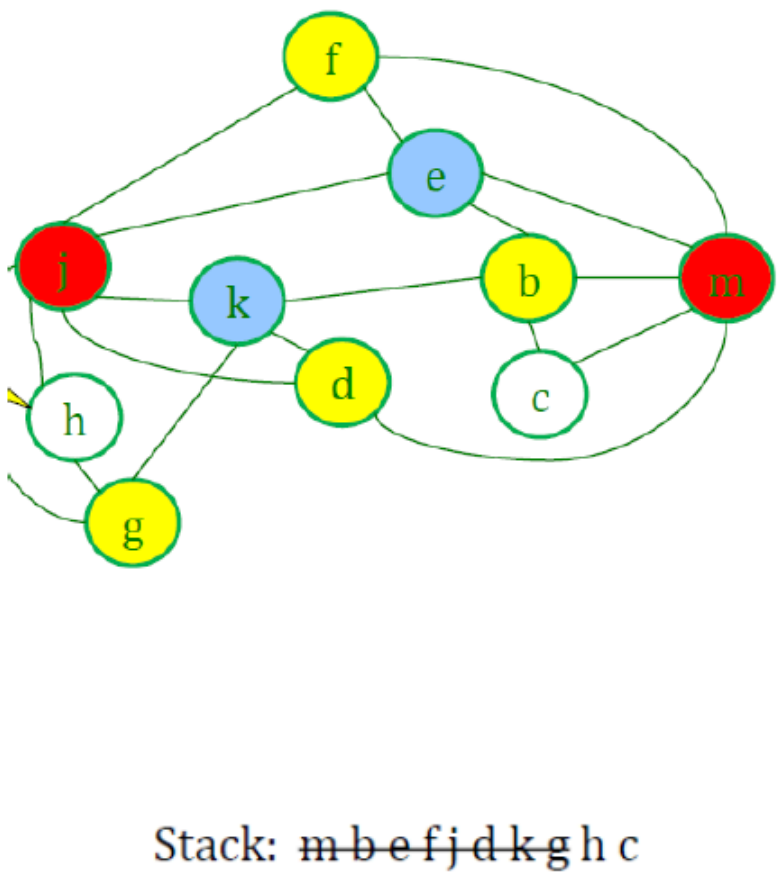
این عمل را تا جایی ادامه دهید که هیچ رأسی در گراف باقی نماند.

پس از پایان حذف‌ها، پشته به صورت زیر خواهد بود:

Stack: m b e f j d k g h c

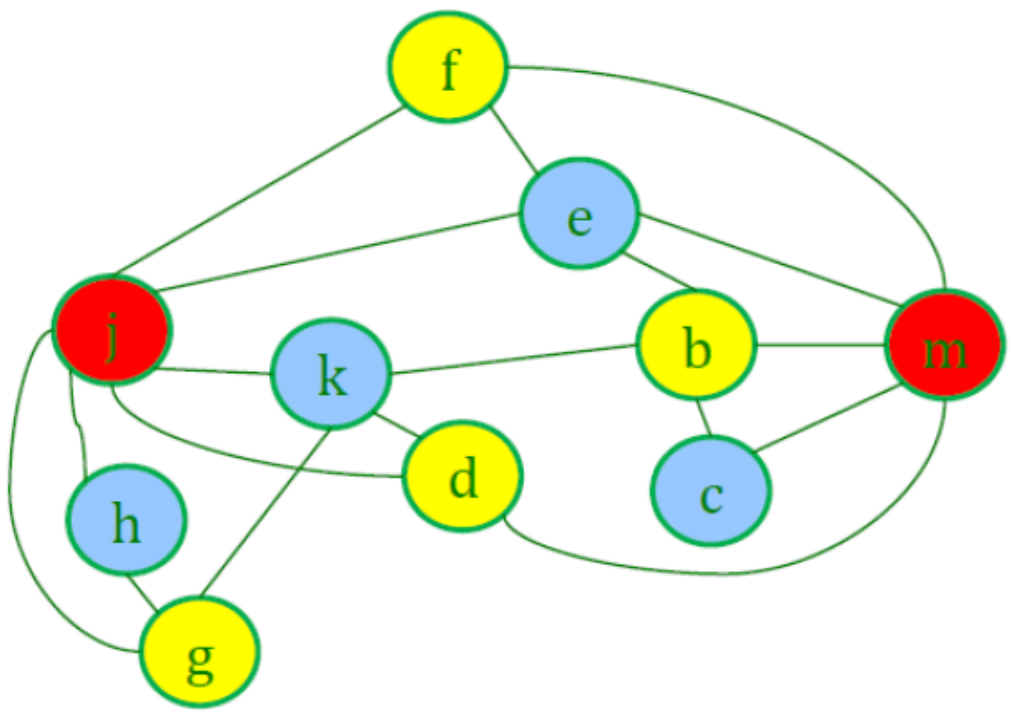
حال، تا خالی شدن پشته، به‌ترتیب خروج از آن، اقدام به رنگ‌کردن رأس‌ها می‌کنیم. 🖍️





Stack: ~~m b e f j d k g~~ h c

در نهایت، گراف به شکل زیر رنگ‌آمیزی می‌شود:



با خالی شدن پشته، یک گراف رنگ‌شده در اختیار داریم. ✅

اما سؤال مهمی پیش می‌آید:

۱. چرا الگوریتم ما کار کرد؟ 🤓

پاسخ: معمولاً رأسی که روی پشته قرار می‌گیرد، کمتر از سه یال ورودی دارد. بنابراین هنگامی‌که آن را از پشته به گراف

بازمی‌گردانیم، حداکثر به دو رأس رنگ‌شده متصل است، و انتخاب رنگ برای آن ساده می‌شود. 🎯

خود مسئله‌ی رنگ‌آمیزی گراف‌ها شاید مستقیماً کاربرد زیادی نداشته باشد، اما در **مدل‌سازی شرایط با منابع محدود** یا **قید و بندها** از آن استفاده می‌شود. به عنوان مثال در:

- طراحی کامپایلرها 🧩
- تخصیص منابع سخت‌افزاری ⚙️
- زمان‌بندی و مدیریت حافظه 💾

کد رنگ آمیزی گراف 👩‍💻

In [4]:
```python
# کلاس گراف بدون جهت برای رنگ‌آمیزی حریصانه
class Graph:
    def __init__(self, V):
        self.V = V  # تعداد رأس‌ها
        self.adj = [[] for _ in range(V)]  # لیست مجاورت برای هر رأس

    # افزودن یال بین دو رأس
    def add_edge(self, v, w):
        self.adj[v].append(w)
        self.adj[w].append(v)  # چون گراف بدون جهت است، هر دو جهت را اضافه می‌کنیم

    # الگوریتم رنگ‌آمیزی حریصانه گراف
    def greedy_coloring(self):
        result = [-1] * self.V  # رنگ هر رأس، در ابتدا بدون رنگ

        result[0] = 0  # رأس اول را با رنگ صفر رنگ می‌کنیم

        available = [False] * self.V  # لیست رنگ‌های مجاز برای هر رأس

        for u in range(1, self.V):
            # بررسی رنگ رأس‌های مجاور و علامت‌گذاری رنگ‌های استفاده‌شده
            for neighbor in self.adj[u]:
                if result[neighbor] != -1:
                    available[result[neighbor]] = True

            # پیدا کردن اولین رنگ آزاد
            cr = 0
            while cr < self.V and available[cr]:
                cr += 1

            result[u] = cr  # اختصاص رنگ به رأس فعلی

            # ریست کردن رنگ‌های مجاز برای تکرار بعدی
            for neighbor in self.adj[u]:
                if result[neighbor] != -1:
                    available[result[neighbor]] = False

        # چاپ رنگ اختصاص داده‌شده به هر رأس
        for u in range(self.V):
            print("Vertex", u, "---> Color", result[u])
```

In [5]:
```python
# ساخت گراف اول و افزودن یال‌ها
g1 = Graph(5)
g1.add_edge(0, 1)
g1.add_edge(0, 2)
g1.add_edge(1, 2)
g1.add_edge(1, 3)
g1.add_edge(2, 3)
g1.add_edge(3, 4)
print("Coloring of graph 1")
g1.greedy_coloring()
```

```
Coloring of graph 1
Vertex 0 ---> Color 0
Vertex 1 ---> Color 1
Vertex 2 ---> Color 2
Vertex 3 ---> Color 0
Vertex 4 ---> Color 1
```

In [6]:
```python
# ساخت گراف دوم و افزودن یال‌ها
g2 = Graph(5)
g2.add_edge(0, 1)
g2.add_edge(0, 2)
g2.add_edge(1, 2)
g2.add_edge(1, 4)
g2.add_edge(2, 4)
g2.add_edge(4, 3)
```

```
print("\nColoring of graph 2")
g2.greedy_coloring()
```

```
Coloring of graph 2
Vertex 0 ---> Color 0
Vertex 1 ---> Color 1
Vertex 2 ---> Color 2
Vertex 3 ---> Color 0
Vertex 4 ---> Color 3
```

جمع زیرمجموعه ➕

مسئله‌ی جمع زیرمجموعه (Subset Sum Problem):

یک مجموعه از اعداد طبیعی $w_{1}, w_{2}, w_{3}, \ldots, w_{n}$ داده شده است. آیا زیرمجموعه‌ای از این مجموعه وجود دارد که جمع عناصر آن برابر با **W** شود؟ 🤔

مثال:

$\{1, 3, 4, 17, 22, 34, 96, 115, 202\}$, **W = 241**

✅ پاسخ: $\{1, 4, 34, 202\}$

این مسئله یکی از مسائل کلاسیک در نظریه‌ی پیچیدگی محاسباتی است و در دسته‌ی مسائل **NP-Complete** قرار می‌گیرد. زیرا بررسی درستی یک پاسخ در زمان چندجمله‌ای ممکن است، اما یافتن پاسخ بهینه به‌صورت کلی در زمان چندجمله‌ای ممکن نیست. ⏱️

کاهش 3SAT به جمع زیرمجموعه 🔗

قضیه: مسئله‌ی 3SAT به مسئله‌ی جمع زیرمجموعه کاهش می‌یابد. (sat $\leqslant_p$ subset sum) ✅

اثبات:

یک نمونه 3SAT با n متغییر و k کلاز به ما داده شده است. برای تبدیل به مسئله جمع زیرمجموعه، 2n + 2k عدد می‌سازیم که دارای n + k رقم هستند: یک رقم برای هر x_i و هر c_j وجود دارد.

- دو عدد برای هر متغییر x_i وجود دارد.
- دو عدد برای هر کلاز c_j وجود دارد.
- جمع هر رقم x_i یک و جمع هر رقم c_j چهار است.

برای درک بهتر، مثال زیر را در نظر بگیرید:

$$
\begin{aligned}
C_1 &= \neg x_1 \vee x_2 \vee x_3 \\
C_2 &= x_1 \vee \neg x_2 \vee x_3 \\
C_3 &= \neg x_1 \vee \neg x_2 \vee \neg x_3
\end{aligned}
$$

برای این مثال، جدول زیر ساخته می‌شود:

	x_1	x_2	x_3	C_1	C_2	C_3	
x_1	1	0	0	0	1	0	100,010
$\neg x_1$	1	0	0	1	0	1	100,101
x_2	0	1	0	1	0	0	10,100
$\neg x_2$	0	1	0	0	1	1	10,011
x_3	0	0	1	1	1	0	1,110
$\neg x_3$	0	0	1	0	0	1	1,001
	0	0	0	1	0	0	100
	0	0	0	2	0	0	200
	0	0	0	0	1	0	10
	0	0	0	0	2	0	20
	0	0	0	0	0	1	1
	0	0	0	0	0	2	2
W	1	1	1	4	4	4	111,444

I satisfiable است اگر و فقط اگر یک زیرمجموعه وجود داشته باشد که جمع آن برابر W شود.

فرض کنید I satisfiable باشد. آن اعدادی را انتخاب می‌کنیم که باعث true شدن I می‌شوند.

از آنجا که I satisfiable است، مجموع ارقام هر ستون c_j حداقل ۱ و حداکثر ۳ است. بدین ترتیب مجموعی پیدا می‌کنیم که جمع رقم c_j را ۴ کند.

حال فرض می‌کنیم یک مجموعه از اعداد موجود است که مجموعشان W است:

- رقم x_i مجبورمان می‌کند تنها یکی از ردیف‌های x_i یا x_i! را انتخاب کنیم.
- رقم c_j مجبورمان می‌کند حداقل یک literal در هر کلاز انتخاب شود.
- x_i را true در نظر می‌گیریم اگر و فقط اگر ردیف x_i انتخاب شده باشد.

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل دهم: الگوریتم‌های تقریبی

بخش اول: پوشش رأسی، فروشنده دوره‌گرد، سختی تقریب

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 📑

مقدمه 📘

ضرب‌المثلی قدیمی در میان مهندسان وجود دارد که می‌گوید: "ارزان، سریع، مطمئن، دو تا را انتخاب کن!"

در دنیای مسائل علوم کامپیوتر هم وضعیت مشابهی برقرار است. همانطور که در فصل‌های پیش آموختیم، بسیاری از مسئله‌های بهینه‌سازی معروف **NP-Hard** هستند. بنابراین اگر $P \neq NP$ باشد، نمی‌توانیم الگوریتم‌هایی داشته باشیم که در زمان چندجمله‌ای جواب بهینه یک مسئله را برای هر نمونه از آن بدهند. وقتی با این مسائل در کاربردهای واقعی سر و کار داریم، حداقل یکی از شرط‌ها باید کنار گذاشته شود.

یکی از راه‌ها این است که شرط چندجمله‌ای بودن الگوریتم را کنار بگذاریم و مانند الگوریتم‌های $A*$ یا CSP فضای جواب‌های مسئله را جستجو کنیم و بر اساس یک آستانه‌ی مناسب جواب نیمه‌اپتیمال به دست آوریم. اما ممکن است مجبور شویم برای رسیدن به جواب قابل قبول، زمان نامشخصی صرف کنیم که در بسیاری از کاربردها مناسب نیست.

راه دیگر این است که شرط "برای هر نمونه" را در نظر نگیریم و با در نظر گرفتن شروط دیگر، حالت خاصی از مسئله را حل کنیم.

این روش تنها زمانی جواب می‌دهد که کاربرد ما در این حالت‌های خاص قرار داشته باشد، که اغلب رخ نمی‌دهد.

راه سوم و مرسوم‌ترین راه، این است که به جای یافتن جواب اپتیمال، جوابی را پیدا کنیم که "به اندازه‌ی کافی خوب" باشد. الگوریتم‌های تقریبی از همینجا به وجود آمده‌اند و به دنبال جوابی هستند که با نسبت خوبی، برای هر نمونه از مسئله، جواب اصلی را تقریب بزنند.

کاربرد الگوریتم‌های تقریبی 🎯

الگوریتم‌های تقریبی وقتی به کار می‌روند که یافتن جواب بهینه‌ی مسئله غیرعملی باشد. بسیاری از مسائل دنیای واقعی شروط بسیار پیچیده‌ای دارند که طراحی یک الگوریتم تقریبی را برایشان دشوار می‌کند. در اینجا مجبور می‌شویم مسئله را ساده‌تر کنیم و برخی از شروط را سست کنیم تا به الگوریتمی مناسب برسیم. سپس از روی این الگوریتم برای حالت ایده‌آل، الگوریتمی برای حالت کلی با تقریب مناسب طراحی می‌کنیم.

تعاریف و ضریب تقریب 📏

اکنون که متوجه شدیم چرا و در کجا از الگوریتم‌های تقریبی استفاده می‌کنیم، در این بخش به بررسی تعاریفی که در حل مسائل با این الگوریتم‌ها به کار می‌آید می‌پردازیم.

الگوریتم‌های تقریبی برای حل مسائل بهینه‌سازی به کار می‌روند که **NP-Hard** هستند. این مسائل می‌توانند مسائل **مینیمم‌سازی** یا **ماکسیمم‌سازی** باشند.

تعریف:

یک الگوریتم `$alpha\$-approx\$` برای یک مسئله‌ی بهینه‌سازی، الگوریتمی با زمان چندجمله‌ای است که به ازای هر نمونه از این مسئله جوابی تولید می‌کند که مقدار آن در نسبت `$alpha\$` برابر جواب بهینه مسئله است.

مقدار α را **ضریب تقریب** یا **ضمانت عملکرد الگوریتم** می‌نامیم. از تعریف مشخص است که:

- برای مسائل مینیمم‌سازی: $\alpha > 1$
- برای مسائل ماکسیمم‌سازی: $\alpha < 1$ (چرا؟)

بنابراین یک الگوریتم `$frac{1}{2}$-approx\$` برای یک مسئله ماکسیمم‌سازی، الگوریتمی از زمان چندجمله‌ای است که همواره جوابش حداقل نصف جواب بهینه مسئله است.

مشابه آن، یک الگوریتم `approx-3` برای یک مسئله مینیمم‌سازی، الگوریتمی از زمان چندجمله‌ای است که جوابش به ازای هر نمونه مسئله حداکثر ۳ برابر جواب بهینه است.

در ادامه‌ی این دفترچه، به مثال‌هایی جالب از طراحی این الگوریتم‌ها برای مسائل **NP-Hard** معروف می‌پردازیم.

پوشش رأسی 🛡️

معرفی مسئله 📌

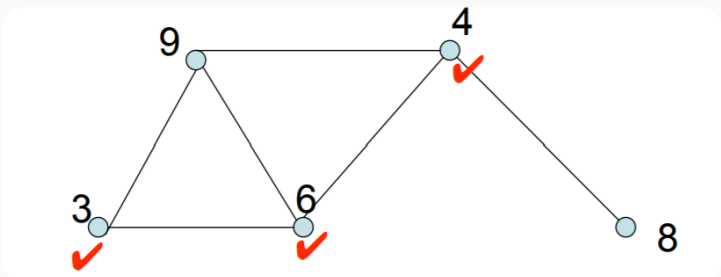
مسئله پوشش راسی (**Vertex Cover**) که در اینجا حالت وزن‌دار آن را بررسی می‌کنیم، یک مسئله کلاسیک **NP-Complete** است و الگوریتم‌های تقریبی متعددی برای آن ارائه شده‌اند.

تعریف مسئله: گراف بدون جهت $G = (V, E)$ داده شده است، به طوری که به هر راس $i \in V$ وزن $w_i \ge 0$ نسبت داده شده است. می‌خواهیم مجموعه‌ای از رئوس $C \subseteq V$ انتخاب کنیم به گونه‌ای که:

- مجموع وزن رئوس C کمینه شود.
- برای هر یال $(i,j) \in E$، حداقل یکی از دو سر آن در C باشد، یعنی $i \in C$ یا $j \in C$.

به عبارت ساده‌تر، یکی از دو سر هر یال باید در مجموعه پوشش راسی باشد.

برای مثال، شکل زیر یک پوشش راسی مینیمال را نشان می‌دهد:



راه‌حل با برنامه‌ریزی خطی 📊

در فصل‌های پیش آموختیم که برنامه‌ریزی خطی یک تکنیک کاربردی برای نمایش و حل مسائل بهینه‌سازی است. ابتدا مسئله پوشش راسی را با برنامه‌ریزی خطی فرموله می‌کنیم.

برای هر رأس $i \in V$، متغیر x_i را تعریف می‌کنیم:

- **$x_i = 1$** اگر رأس i در مجموعه انتخاب شود.
- **$x_i = 0$** در غیر این صورت.

مسئله پوشش راسی به شکل برنامه‌ریزی خطی صحیح (ILP) به صورت زیر است:

$$\min \sum_i x_i w_i$$

$$s.t. \quad \forall (i,j) \in E : x_i + x_j \ge 1$$

$$\forall i \in V : x_i \in \{0,1\}$$

از آنجا که ILPها **NP-Hard** هستند، برای یافتن جواب تقریبی زمان چندجمله‌ای، شرط صحیح بودن x_iها را برمی‌داریم و به جای آن شرط $x_i \in [0,1]$ قرار می‌دهیم. برنامه خطی بدست آمده:

$$\min \sum_i x_i w_i$$

$$s.t. \quad \forall (i,j) \in E : x_i + x_j \ge 1$$

$$\forall i \in V : x_i \ge 0$$

حل این LP در زمان چندجمله‌ای امکان‌پذیر است. فرض کنید جواب بهینه LP برابر `x^*` باشد. مجموعه پوشش راسی را به صورت زیر تعریف می‌کنیم:

$$C = \{i \mid x_i^* \geq \frac{1}{2}\}$$

طبق شرط LP، برای هر یال $(i,j) \in E$ داریم:

$$x_i^* + x_j^* \geq 1 \implies x_i^* \geq \frac{1}{2} \text{ ای } x_j^* \geq \frac{1}{2}$$

بنابراین حداقل یکی از دو رأس در مجموعه C قرار دارد و C یک پوشش راسی معتبر است. ✅

همچنین اگر C^* پوشش راسی مینیمال باشد، داریم:

$$w(C^*) = \sum_{i \in C^*} w_i = OPT_{ILP} \geq OPT_{LP} = \sum_i x_i^* w_i \geq \frac{1}{2} \sum_{i \in C} w_i = \frac{1}{2} w(C)$$

بنابراین:

$$w(C) \leq 2w(C^*)$$

سؤال: چرا داریم $OPT_{ILP} \geq OPT_{LP}$؟

پاسخ: هر جواب صحیح ILP، یک جواب معتبر برای LP نیز هست. بنابراین بهینه ILP حداقل برابر بهینه LP است.

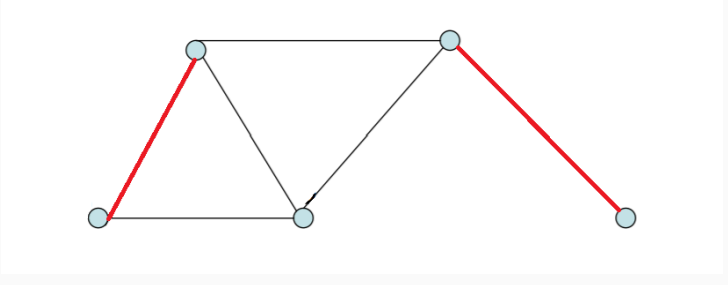
نتیجه: توانستیم الگوریتمی **approx-2** برای مسئله پوشش راسی ارائه کنیم.

راه‌حل با بزرگترین تطابق ⚙️

در این بخش می خواهیم راه حل جالب دیگری برای تقریب زدن مسئله پوشش راسی ارائه دهیم. برای سادگی در این بخش مسئله پوشش راسی بدون وزن را در نظر می گیریم. در واقع $\forall i : w_i = 1$. بنابراین مسئله معادل با یافتن کم ترین تعداد راس می شود که یک پوشش راسی تشکیل دهند.

تطابق بیشینه M را به این صورت می سازیم: ابتدا یکی از یال های گراف را در نظر می گیریم و دو سر آن را در مجموعه ی M قرار می دهیم و این دو راس و یال های متصل بهشان از گراف اصلی حذف می کنیم. سپس از گراف باقی مانده یال دیگری در نظر می گیریم و همین فرآیند را برای تمام یال ها تکرار می کنیم تا در نهایت هیچ یالی در گراف باقی نماند. مجموعه ی به دست آمده مجموعه ی مجزا از یال های گراف است که به آن تطابق بیشینه می گویند.

برای مثال در شکل ابتدای این بخش یک تطابق بیشینه را به دست آورده ایم.



اکنون ثابت می کنیم M یک پوشش راسی است. فرض کنیم یال $(i,j) \in E$ وجود داشته باشد که هیچ یک از دو سر آن در M

نباشد. در این صورت $M \cup e_{ij}$ یک تطابق است. (چرا؟) اما این با فرض این که M یک تطابق بیشینه بود در تناقض است.
بنابراین هر یالی حداقل یکی از دو سرش در M است.
همچنین به ازای هر $(i,j) \in M$ هر پوشش راسی C باید حداقل یکی از i یا j یا شامل شود. بنابراین اگر C^* پوشش راسی اپتیمال باشد داریم

$$|M| \leq 2 \times |C^*|$$

که عبارت بالا از این نتیجه می شود که یال های M مجزا هستند.
بنابراین با استفاده از تطابق ماکسیمال نیز می توان یک الگوریتم $2 - approx$ برای مسئله پوشش راسی به دست آورد.

فروشنده دوره‌گرد 🧳

معرفی مسئله

مسئله فروشنده دوره‌گرد (Travelling Salesman Problem) یا به اختصار TSP از مسائل معروف NP است که بسیار مطالعه شده‌است. در این مسئله تعدادی شهر داریم که نمایانگر رئوس و بین هر دو شهر جاده‌ای داریم که نمایانگر یال‌ها می‌باشند طی کردن هر جاده مدت زمانی طول می‌کشد. حال یک فروشنده دوره‌گرد سعی می‌کند از شهری شروع کند، کل شهرها را یکبار ببیند و سر جای اولیه‌اش برگردد. دغدغه او پیدا کردن چنین دوری است به طوریکه مدت زمانی که در جاده‌ها سپری می‌کند کمینه شود.
به بیان ساده‌تر مسئله فروشنده دوره‌گرد پیدا کردن کوتاهترین دور همیلتونی در یک گراف کامل بدون جهت وزن‌دار با وزن‌های مثبت است.
این مسئله یک مسئله NP-Hard است و نسخه تصمیم‌گیری‌اش به این صورت است:
گرافی وزن‌دار و یک عدد k داده‌شده‌است. حال می‌خواهیم ببینیم آیا دور همیلتونی‌ای وجود دارد که وزنش کمتر یا مساوی k باشد.

راه‌حل حریصانه 🏃

حال که با مسئله NP-Hard طرفیم سعی می‌کنیم به سراغ راه‌حل‌های تقریبی برای حل این مسئله برویم. برای راحتی کار فعلاً فرض کنید گراف ما اقلیدسی است. یعنی هر رأس معادل نقطه‌ای روی صفحه مختصات است و یال بین دو رأس بیانگر فاصله بین آن دو است.
حال به سراغ حل این حالت می‌رویم. فرض کنید از رأس ۱ شروع می‌کنیم و روشی حریصانه انتخاب می‌کنیم و هر سری بین رئوسی که هنوز ندیدیم رأسی را انتخاب می‌کنیم که فاصله‌اش با یکی از رئوس مجموعه رئوس دیده‌شده کمینه باشد. یعنی یک مجموعه S داریم که در ابتدا در آن ۱ قرار دارد سپس بین رئوس خارج از S رأسی را انتخاب می‌کنیم که به S از همه رئوس دیگر نزدیکتر است. یعنی وزن مینیمم یالش به این مجموعه کمینه‌است. سپس این رأس را به S اضافه می‌کنیم. فرض کنید در تمامی مراحل می‌خواهیم دوری را نگه داریم و بزرگ کنیم. فرض کنید پس از سه مرحله رئوس a و b و c به مجموعه اضافه شده‌اند. دور C را برابر این سه‌تایی تعریف می‌کنیم.
سپس در هر مرحله که رأسی مثل j قرار است به مجموعه اضافه شود، فرض کنید کوتاهترین یالش به S یالش به رأس i باشد. اگر k رأس مجاور i در دور C باشد، یال ik را حذف کرده و دو یال ij و jk را به C اضافه می‌کنیم.
حال سراغ اثبات این الگوریتم می‌رویم.
الگوریتم در انتخاب کردن رئوس بسیار شبیه به الگوریتم Prim در پیدا کردن درخت کمینه فراگیر عمل می‌کند. بنابراین مقایسه وزن کوچکترین درخت فراگیر و کوچکترین دور همیلتونی امری طبیعی است:

لم: وزن دور همیلتونی کمینه بزرگتر مساوی کوچکترین درخت فراگیر است.

اثبات چنین چیزی خیلی راحت است چرا که با حذف یک یال از دور همیلتونی یک مسیر داریم که فراگیر است و چون هر مسیر یک درخت است وزن این مسیر قاعدتاً بیشتر مساوی درخت فراگیر کمینه است در نتیجه وزن این دور بیشتر از درخت فراگیر کمینه است.

حالا اگر ثابت کنیم الگوریتم داده‌شده وزنش حداکثر ۲ برابر کوچکترین درخت فراگیر است می‌توانیم به سادگی ثابت کنیم الگوریتم $2 - approx$ می‌باشد:

$$ans \leq 2 \times MST \leq 2 \times OPT \ \rightarrow \ ans \leq 2 \times OPT$$

برای اثبات این موضوع از درستی الگوریتم Prim استفاده می‌کنیم. می‌دانیم رأس ij رأسی از درخت فراگیر کمینه برای مجموعه S است. اگر وزن C را قبل از تغییرات c بنامیم اکنون این وزن اینگونه تغییر کرده‌است:

$$c' = c - w_{ik} + w_{ij} + w_{jk}$$

اگر وزن درخت پوشای کمینه در هر مرحله t باشد اکنون به این‌صورت تغییر کرده‌است:

$$t' = t + w_{ij}$$

نامساوی مثلثی برای یال‌ها برقرار است یعنی داریم:

$$w_{jk} \leq w_{ij} + w_{ik} \rightarrow w_{jk} - w_{ik} \leq w_{ij}$$

بنابراین:

$$c' = c - w_{ik} + w_{jk} + w_{ij} \leq c + 2w_{ij} \leq 2t + 2w_{ij} \leq 2t' \rightarrow c' \leq 2t'$$

در نتیجه الگوریتمی چند جمله‌ای داریم که می‌تواند با ضریب تقریب ۲ مسئله TSP را روی گراف اقلیدسی حل کند! به طور دقیق‌تر هر گرافی که نامساوی مثلثی به‌ازای یال‌هایش برقرار باشد این الگوریتم برایش جواب‌گو خواهد بود! به این گراف‌ها گراف‌های **متریک** می‌گویند.

روش راحت‌تر برای پیاده‌سازی الگوریتم:
فرض کنید به هر روشی یک درخت پوشای کمینه را پیداکردیم. حالا روی این درخت DFS می‌زنیم و رئوس را به ترتیب دیدنشان کنار هم می‌گذاریم. می‌توان ثابت کرد اگر این لیست را به ترتیب بپیماییم و در نهایت به رأس شروعمان برگردیم دور همیلتونی حاصل نیز حداکثر ۲ برابر وزن دور همیلتونی کمینه را دارد.

In [2]:
```python
# الگوریتم تقریبی برای حل TSP و DFS روی گراف‌های متریک با استفاده از درخت پوشای کمینه

INF = 10**9 + 10
MAX_N = 220

# تعریف متغیرهای اصلی
n = 0  # تعداد رئوس گراف
cost = [[0] * MAX_N for _ in range(MAX_N)]  # ماتریس وزن یال‌ها
mst = [[] for _ in range(MAX_N)]  # لیست مجاورت درخت پوشای کمینه
d_prim = [INF] * MAX_N  # فاصله‌ها برای الگوریتم Prim
seen = [False] * MAX_N  # بررسی بازدید رأس‌ها
TSP = []  # مسیر تقریبی TSP

# اجرای الگوریتم Prim برای ساخت درخت پوشای کمینه
def run_prim_algorithm():
    d_prim[0] = 0
    for _ in range(n):
        mn = -1
        for j in range(n):
            if not seen[j] and (mn == -1 or d_prim[j] < d_prim[mn]):
                mn = j
        seen[mn] = True
        for j in range(n):
            if seen[j] and cost[j][mn] == d_prim[mn] and j != mn:
                mst[j].append(mn)
                mst[mn].append(j)
        for j in range(n):
            if not seen[j]:
```

```
                d_prim[j] = min(d_prim[j], cost[mn][j])

# پیمایش DFS روی درخت پوشای کمینه برای ساخت مسیر تقریبی TSP
def dfs(s):
    seen[s] = True
    TSP.append(s + 1)  # شماره‌گذاری از ۱ برای خروجی
    for v in mst[s]:
        if not seen[v]:
            dfs(v)

# دریافت ورودی و بررسی متریک بودن گراف
n = int(input())
for i in range(n):
    for j in range(i):
        cost[i][j] = int(input())
        cost[j][i] = cost[i][j]

# بررسی نامساوی مثلثی برای اطمینان از متریک بودن گراف
for i in range(n):
    for j in range(i + 1, n):
        for k in range(n):
            if cost[i][j] > cost[i][k] + cost[k][j]:
                print("Graph is not metric!")
                exit()

run_prim_algorithm()
seen = [False] * MAX_N  # ریست بازدیدها برای DFS
dfs(0)

# چاپ مسیر تقریبی TSP
for v in TSP:
    print(v, end=' ')
print()
```

1 3 2 4

راه‌حلی بهتر 🧩

در ادامه به بررسی روشی می‌پردازیم که کمی تقریب بهتری روی گراف‌های متریک به ما می‌دهد. کلید این راه‌حل این است که سعی کنیم یک تور اویلری روی گراف پیداکنیم که وزنش کم است. به صورت دقیق‌تر حداکثر $\frac{3}{2}$ وزن کوچکترین درخت فراگیر است.
فرض کنید باز هم کوچکترین درخت فراگیر کمینه را پیدا کردیم، حالا می‌خواهیم تعدادی یال به این درخت اضافه کنیم تا درجه همه رئوس زوج شود، در این‌صورت می‌دانیم تور اویلری‌ای وجود دارد که شامل کل رئوس بشود. برای اینکار رئوس درجه فرد را در مجموعه‌ای مثل O در نظر می‌گیریم. حال اگر تطابقی از این رئوس پیدا کنیم و آن‌را به مجموعه یال‌ها اضافه کنیم هر رأسی که قبلاً درجه‌اش فرد بوده زوج می‌شود. توجه کنید ممکن است یک یال دوبار تکرار شود اما مشکلی بوجود نخواهد آمد.
حالا نکته‌ای که وجود دارد این است که این $\frac{|O|}{2}$ یال اضافه را طوری انتخاب کنیم که مجموع وزن یال‌ها کمینه شود. برای این‌کار می‌توانیم از الگوریتم کوچکترین تطابق وزن‌دار استفاده نماییم.

لم: اگر کوچکترین دور همیلتونی کمینه در یک گراف متریک OPT باشد آن وقت کوچکترین تطابق برای مجموعه O اندازه‌اش حداکثر $\frac{OPT}{2}$ است.
دور همیلتونی کمینه را در نظر بگیرید و رئوس O را روی آن در نظر بگیرید. اگر این رئوس را به ترتیب شماره گذاری کنیم یکی از دو تطابق

$$\{o_1, o_2\}, \{o_3, o_4\}, \ldots, \{o_{n-1}, o_n\}$$

یا

$$\{o_2, o_3\}, \{o_4, o_5\}, \ldots, \{o_n, o_1\}$$

وزنش حداکثر $\frac{OPT}{2}$ است (چرا؟)

بنابراین حالا که لم ثابت شد می‌توانیم تور اویلری‌ای بیابیم که مجموع وزن یال‌هایش حداکثر $\frac{3}{2}OPT$ است. حالا کافیست در تور اویلری حرکت کنیم و اگر بین رأس u و v تعدادی رأس تکراری مثل $x_1, x_2, \ldots, x_k$ وجود داشت آن‌ها را نبینیم و به جای آن مستقیم از u به v برویم. به‌دلیل متریک بودن گراف داریم:

$$w_{uv} \leq w_{u,x_1} + w_{x_1,x_2} + \ldots + w_{x_k,v}$$

بنابراین دوری که با انجام این‌کار می‌بینیم وزنی کمتر مساوی تور اویلری دارد که خود کمتر مساوی

$$\frac{3}{2}\,OPT$$

است.

سختی تقریب 🔒

برای تحلیل سختی یک الگوریتم تقریبی با ضریب تقریب α راه‌های مختلفی وجود دارد. یکی از مرسوم‌ترین راه‌ها کاهش دادن مسئله به مسئله $NP-Complete$ است. اینطور که می‌گوییم اگر فرض کنیم $P \neq NP$ در این‌صورت الگوریتمی $\alpha - approx$ برای مسئله X نداریم.

برای مثال بر می‌گردیم به مسئله فروشنده دوره‌گرد! در حالت متریک الگوریتم‌های تقریبی خوبی به‌دست آوردیم. اما حالا می‌خواهیم ببینیم آیا برای حالت کلی می‌توان برای مثال الگوریتم تقریبی $2 - approx$ ارائه کرد؟

قضیه: ⚖️ اگر $P \neq NP$ در این‌صورت الگوریتمی چندجمله‌ای $2 - approx$ برای مسئله TSP وجود ندارد.
اثبات: فرض می‌کنیم الگوریتم چندجمله‌ای و $2 - approx$ برای این‌کار داریم. با استفاده از این الگوریتم راه‌حلی چند جمله‌ای برای مسئله $NP - Complete$ وجود داشتن دور همیلتونی ارائه می‌کنیم. به این‌صورت که یال‌های داخل گراف را با وزن ۱ وزن‌دار می‌کنیم و یال‌هایی که ظاهر نشده‌اند با $n+2$ وزن‌دار می‌کنیم. اگر الگوریتم تقریبی را اجرا کنیم و عددی کمتر از $2n+1$ حاصل شد یعنی این گراف دور همیلتونی دارد. و در غیر اینصورت فاقد دور همیلتونی است. (توجه کنید این وزن‌گذاری متریک نیست!)

می‌توانیم از این هم فراتر برویم فرض کنید الگوریتمی $\alpha - approx$ برای این مسئله داشته‌باشیم. در این‌صورت اگر وزن یال‌های گراف را برابر ۱ و وزن‌های دیگر را برابر $(\alpha-1)\times n+1$ بگذاریم می‌توانیم وجود یا عدم وجود دور همیلتونی را تشخیص دهیم. بنابراین به ازای **هر عدد ثابت** $\alpha \geq 1$ **هیچ الگوریتم تقریبی چند جمله‌ای** $\alpha - approx$ برای مسئله TSP نداریم!

تمرین: 📝 مسئله $k - centre$ مسئله‌ای است که در آن تعدادی نقطه در صفحه مختصات داریم. می‌خواهیم این نقاط را در k گروه تقسیم کنیم و برای هر گروه یک نقطه تحت عنوان مرکز بکشیم. وزن هر دسته را برابر ماکسیمم فاصله نقاط آن دسته از مرکزش می‌گذاریم. می‌خواهیم تقسیم‌بندی و مرکزگذاری‌ای را بیابیم به‌طوریکه ماکسیمم وزن مینیمم شود.
الف) سعی کنید الگوریتمی چندجمله‌ای $2 - approx$ برای این‌کار ارائه دهید.
ب) ثابت کنید هیچ الگوریتم چند جمله‌ای $\alpha - approx, \alpha < 2$ برای این‌کار وجود ندارد مگر این‌که $P = NP$

راهنمایی: 💡 برای بخش ب) سعی کنید با مسئله کوچکترین مجموعه غالب تناظر دهید. مجموعه غالب در یک گراف برابر مجموعه رأسی است که هر رأس در گراف یا داخل مجموعه غالب است یا اینکه به یکی از اعضای مجموعه غالب یال دارد.

منابع 📚

- کتاب The design of Approximation Algorithms نوشته Williamson و Shmoys
- جزوه درس الگوریتم‌های تقریبی دانشگاه Illinois

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل دهم: الگوریتم‌های‌تقریبی

بخش دوم: طرح‌های تقریبی چندجمله‌ای، کوله‌پشتی

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 🗂️

مقدمه 📝

در بخش قبلی دیدید یک مسئله ممکن است تقریب‌های متفاوتی داشته‌باشد؛ برای مثال برای فروشنده دوره‌گرد ابتدا یک ۲-تقریب و سپس یک ۱.۵ تقریب ارائه کردیم.

سؤال منطقی‌ای که پیش می‌آید این است که این روند تا کجا ادامه پیدا می‌کند؟ آیا ممکن است الگوریتمی با تقریب بهتر ارائه کنیم.

گاهی ممکن است تن به محاسبات بیشتری بدهیم تا دقت الگوریتم را بیشتر کنیم؛ برای مثال ممکن است یک ۱۰-تقریب داشته باشیم که با هزینه زمانی کمتری حل شود اما از طرفی حاضر باشیم هزینه زمانی بیشتری بدهیم که این تقریب ۲-تقریب بشود.

در این بخش ابتدا تعریفی را ارائه می‌دهیم و سپس یک مسئله معروف «کوله‌پشتی» با چند روش برای حل کردن آن به‌دست می‌آوریم. سپس یک طرح تقریبی چندجمله‌ای ارائه می‌کنیم که بتوان با هر دقتی مسئله را حل کرد.

طرح‌های تقریبی چندجمله‌ای ⚙️

در بخش قبلی سعی می‌کردیم الگوریتم‌ها با ضرایب تقریب متفاوت ارائه کنیم. اما بعضاً ممکن است بخواهیم دقت الگوریتم را خودمان تعیین کنیم. برای مثال فرض کنید سیستمی قوی داریم و می‌توانیم هزینه بیشتری از نظر زمانی بدهیم و در عوض دقت بیشتری بگیریم. و یا بالعکس.

بنابراین می‌خواهیم پارامتری تحت عنوان دقت را به صورت یک **متغیر** ارائه دهیم. در این بخش با چنین الگوریتم‌هایی سر و کار داریم.

تعریف: (PTAS) طرح‌های تقریبی چندجمله‌ای خانواده‌ای از الگوریتم‌ها مثل A_ϵ است. به طوریکه به‌ازای هر ضریب $\epsilon > 0$ الگوریتم چندجمله‌ای $(1+\epsilon) - approx$ است (برای مسائل کمینه‌سازی) و $(1-\epsilon) - approx$ است (برای مسائل بیشینه‌سازی).

توجه کنید ممکن است زمان الگوریتم ما وابسته به این ϵ باشد. برای مثال ممکن است الگوریتمی با $O(n^{\frac{1}{\epsilon}})$ داشته‌باشیم. در اینصورت اگر مقدار ϵ خیلی کم باشد (یعنی الگوریتمی با دقت زیاد داشته‌باشیم) آن وقت زمان الگوریتم بیشتر شود. این الگوریتم بر حسب n چند جمله‌ای است ولی برحسب $\frac{1}{\epsilon}$ چندجمله‌ای نیست!

تعریف: (FPTAS) طرح‌های تقریبی کاملاً چندجمله‌ای یا FPTAS دسته‌ای از الگوریتم‌های PTAS هستند که هم بر اساس اندازه ورودی و هم بر اساس پارامتر دقت یعنی ϵ چندجمله‌ای است.

معمولاً پیدا کردن FPTAS در مسائل بسیار خوشحال کننده‌است. زیرا می‌توانیم با توجه به سیستمی که در اختیار داریم یک TradeOff بین دقت و performance داشته‌باشیم.

مسئله کوله‌پشتی 🎒

معرفی مسئله مسئله کوله‌پشتی مسئله معروفی است که تا الآن با آن آشنایی دارید اما صرفاً جهت یادآوری صورت مسئله را در زیر بیان می‌کنیم:

به عنوان ورودی n شی داریم که هر کدام ارزش s_i و اندازه p_i دارند. هدف، قرار دادن این اشیا در کوله‌پشتی با ظرفیت B به صورتی است که مقدار ارزش بیشینه حاصل شود. (برای سادگی فرض می‌کنیم برای تمامی اشیاء $p_i < B$ و باید زیرمجموعه‌ای از این اشیاء پیدا کنیم $I \subset [n]$ تا مقدار $\sum_{i \in I} s_i$ را ماکسیمم کند و شرط مسئله $\sum_{i \in I} p_i \leqslant B$ است.)

سؤال منطقی‌ای که اول بوجود می‌آید این‌است که: «آیا واقعا لازم است سراغ راه‌حل‌های تقریبی برویم؟ آیا راه‌چند جمله‌ای برای حل این مسئله هست یا نه؟»

از آنجا که مسئله‌ی کوله‌پشتی از مسئله‌ی تقسیم‌بندی (partition problem) کاهش می‌یابد، بنابراین جز NP-Hard است.

روش برنامه‌ریزی پویا 💻

در ابتدا راه‌حلی ارائه می‌دهیم که با استفاده از برنامه‌نویسی پویا انجام‌پذیر است. فرض کنید به ازای هر i, j که $0 \leq i \leq n$ و $0 \leq j \leq p_{max}$ که در آن p_{max} اندازه بیشترین مقدار ممکن برای اندازه جواب است، یعنی $p_{max} = \max_{I \subset [n]} : (\sum_{i \in I} p_i) \leq B$ فرض کنید یک «دانای کل» داریم به فرم $O(i, j)$ که می‌گوید اگر ظرفیت کوله‌پشتی ما j باشد و بخواهیم صرفاً از اشیای ۱ تا i استفاده کنیم، ماکسیمم ارزشی که می‌توانیم برداریم چیست.

حال می‌توانیم به‌شکل زیر $O(i, j)$ را از روی یکسری $O(i', j')$ های دیگر به‌دست بیاوریم:

- اگر $p_i \leq j$ باشد آنگاه: $O(i, j) = \max(O(i-1, j), O(i-1, j-p_i) + s_i)$ یعنی می‌توانیم انتخاب کنیم که شئ i ام را برداریم یا برنداریم.
- در غیر اینصورت: $O(i, j) = O(i-1, j)$ زیرا نمی‌توانیم شئ i ام را برداریم.

برای پایه هم می‌توانیم فرض کنیم: $\forall_{k \in [0, p_{max}]} : O(0, k) = 0$

کد زیر چیزی که ما گفتیم را پیاده‌سازی می‌کند:

In [1]:
```python
# الگوریتم برنامه‌ریزی پویا برای حل مسئله کوله‌پشتی با حذف اشیاء بزرگ‌تر از ظرفیت

MAX_B = 2020
MAX_N = 2020

# تعریف جدول dp برای نگهداری جواب‌های زیرمسئله
dp = [[0] * MAX_B for _ in range(MAX_N)]

# کلاس شیء شامل ارزش و اندازه
class Obj:
    def __init__(self, s, p):
        self.s = s  # ارزش
        self.p = p  # اندازه

# حل مسئله با استفاده از برنامه‌نویسی پویا
def solve_using_dp(n, B, obj):
    for j in range(B + 1):
        dp[0][j] = 0  # مقدار پایه برای هیچ شیء

    for i in range(n):
        for j in range(B, -1, -1):  # پیمایش ظرفیت از بالا به پایین
            if j >= obj[i].p:
                dp[i + 1][j] = max(dp[i][j], dp[i][j - obj[i].p] + obj[i].s)
            else:
                dp[i + 1][j] = dp[i][j]
    return dp[n][B]

# دریافت ورودی
B, n = map(int, input().split())
obj = []
for _ in range(n):
    p, s = map(int, input().split())
    obj.append(Obj(s, p))

# حذف اشیایی که اندازه‌شان بیشتر از ظرفیت است
filtered = []
for i in range(n):
    if obj[i].p <= B:
        filtered.append(obj[i])
n = len(filtered)

# چاپ جواب نهایی
print(solve_using_dp(n, B, filtered))
```

```
90
```

می‌توانیم به‌راحتی ببینیم که این الگوریتم $O(np_{max})$ است. ممکن است گول ظاهر این تحلیل را بخورید و نتیجه بگیرید که مسئله NP_Hard نیست! (یا شاید هم باید جایزه میلیون دلاری **P = NP** را دریافت کنید!) ⏱️

در واقع ما الگوریتمی از $O(poly(n, p_{max}))$ به دست آوردیم، اما واقعیت این است که تحلیل زمانی الگوریتم‌ها بر اساس اندازه ورودی در نظر گرفته می‌شود.

به‌عبارت بهتر، شما برای ذخیره‌سازی هر p_i به اندازه $log(p_i)$ بیت ورودی می‌گیرید و زمان الگوریتم در واقع $O(n \times radix^{log(p_i)})$ است که براساس اندازه ورودی نمایی است!

در واقع به اینگونه مسائل، مسائل **Psuedo Polynomial** نیز می‌گویند.

راه‌حل با الگوریتم حریصانه 🎯

توجه کنید اساس طراحی الگوریتم حریصانه پیداکردن پارامتر خوب است. ممکن است ترتیبمان این‌باشد که اشیا با ارزش‌تر را زودتر برداریم اما می‌توان به‌سادگی دید چنین روشی برای الگوریتم حریصانه مفید نیست! ممکن است چیزی که برمی‌داریم خیلی وزین باشد و بهتر باشد اشیا کم‌ارزش ولی کوچک برداریم تا بتوانیم مقدار زیادی از آن‌ها داشته‌باشیم.

از طرفی توجه کردن تنها به اندازه هم به ما الگوریتم‌های غلطی می‌دهد لذا بهترین کار چیزی مابین این‌دو است، یعنی چیزی را برداریم که «چگالی» ارزشش بیشینه باشد و به ازای هر واحد اندازه بیشترین ارزش را بتوانیم برداریم.

استفاده از الگوریتم حریصانه 1:

- اشیا را به صورت نزولی برحسب $\frac{s_i}{p_i}$ مرتب کن.
- به صورت حریصانه اشیاء را به ترتیب بردار.

اما متأسفانه حتی این‌روش هم روش خوبی نیست؛ در واقع مثالی ارائه می‌دهیم که به‌ازای Bهای بزرگ بد عمل می‌کند.

$$p =< 1, B >$$
$$s =< 2, B >$$

طبق الگوریتم عنصر اول را برمی‌داریم و دیگر چیزی برنمی‌داریم اما جواب بهینه B است و جواب ما 2 است، در نتیجه هیچ ضریب اطمینانی نداریم!

استفاده از الگوریتم حریصانه 2 (redux):

- اشیا را به صورت نزولی برحسب $\frac{s_i}{p_i}$ مرتب کن.
- به صورت حریصانه شیءها را اضافه کن تا زمانی که دیگر نتوانیم شیئی اضافه کنیم $(\sum_{k=1}^{i} p_i > B)$.
- در این‌صورت یا همان اشیا را انتخاب می‌کنیم یا آخرین عضوی را که دیگر جا نمی‌شد به تنهایی انتخاب می‌کنیم.

در نگاه اول صرفاً به نظر می‌رسد راهی برای پیشگیری از مثال نقض سری پیش است، اما در کمال تعجب این الگوریتم واقعاً یک تقریب خوب به ما می‌دهد.

قضیه: الگوریتم حریصانه redux دو-تقریب برای مسئله کوله‌پشتی است.

اثبات: ما برای این مسئله، یک الگوریتم حریصانه پیشنهاد دادیم. بنابراین می‌توانیم بگوییم اگر الگوریتم ما زیربهینه (suboptimal) باشد، در آخر یک فضای B-P باقی می‌ماند. اگر الگوریتم ما می‌توانست قسمتی از اشیاء را درون کوله‌پشتی قرار دهد، برای مثال یک چهارم شیئی را قرار دهد، آنگاه با اضافه کردن $\frac{B-P}{p_i} s_i$ به کوله‌پشتی ما یا به OPT می‌رسیدیم، یا به کوله‌پشتی پرشده. (دقت

کنید در این مسئله نمی‌توانیم از بخشی از اشیاء استفاده کنیم.)

بنابراین یا $s_i \geqslant \frac{1}{2}OPT$ یا $\sum_{k=1}^{i-1} s_i \geqslant \frac{1}{2}OPT$ یا $s_i \geqslant \frac{B-P}{p_i}s_i \geqslant \frac{1}{2}OPT$ اتفاق می‌افتد.

از دیگر فواید این روش، کد کوتاه آن است!

In [2]:
```python
# الگوریتم حریصانه redux ۲ برای مسئله کوله‌پشتی با تقریب

# کلاس شیء شامل ارزش و اندازه
class Obj:
    def __init__(self, s, p):
        self.s = s  # ارزش
        self.p = p  # اندازه

# تابع مقایسه برای مرتب‌سازی بر اساس چگالی ارزش (s/p)
def cmp(a):
    return -a.s / a.p  # منفی برای مرتب‌سازی نزولی

# الگوریتم تقریبی redux برای حل مسئله
def redux(objs, B):
    return_value = 0
    sum_of_s = 0
    remaining_space = B
    for obj in objs:
        remaining_space -= obj.p
        if remaining_space < 0:
            return_value = max(obj.s, sum_of_s)
            break
        sum_of_s += obj.s
    return max(return_value, sum_of_s)

# دریافت ورودی
B, n = map(int, input().split())
objs = []
for _ in range(n):
    p, s = map(int, input().split())
    if p <= B:  # حذف اشیاء بزرگ‌تر از ظرفیت
        objs.append(Obj(s, p))

# مرتب‌سازی اشیاء بر اساس چگالی ارزش
objs.sort(key=cmp)

# چاپ جواب الگوریتم تقریبی
print(redux(objs, B))
```

```
90
```

راه‌حل با برنامه‌ریزی خطی 📊

در این مدل سعی می‌کنیم الگوریتمی تقریبی ارائه دهیم که به ما تنها یک عدد می‌دهد و به ما چینش مناسبی ارائه نمی‌کند، اما این عدد به ما حدودی از جواب بهینه را می‌دهد.

ابتدا یک مدل IP برای این مسئله ارائه می‌دهیم که به صورت زیر است:

$$x_i \in \{0, p_i\}$$

$$Constraint: \quad \sum_{1 \leq i \leq n} x_i \leq B$$

$$Maximize: \quad \sum_{1 \leq i \leq n} x_i \times s_i$$

یکی از ایده‌های روتین برای به‌دست آوردن الگوریتم تقریبی به اصطلاح relax کردن IP است که به فرم LP می‌رسیم:

$$Constraint: \quad \sum_{1 \leq i \leq n} x_i \leq B, \ \forall_{1 \leq i \leq n} 0 \leq x_i \leq p_i$$

$$Maximize: \quad \sum_{1 \leq i \leq n} x_i \times s_i$$

می‌دانیم این مسئله راه‌حل چندجمله‌ای، مثلاً با استفاده از الگوریتم Simplex دارد، لذا جوابی به ما می‌دهد. آیا این جواب خوب است؟

ابتدا برای سادگی فرض کنید s_i ها متمایزند. البته می‌توان کاری کرد که در حالت کلی هم این فرض برقرار نباشد، اما آن را به خواننده واگذار می‌کنیم.

حال ادعا می‌کنیم در جواب این مسئله LP همه x_i ها یا صفرند یا p_i بجز حداکثر یکی از آن‌ها!
برای اثبات این ادعا فرض کنید i, j داریم که $0 < x_i < p_i, 0 < x_j < p_j$. بدون کاستن از کلیت فرض کنید $s_i > s_j$. در اینصورت کافیست مقدار x_i را به اندازه ϵ کم کنیم و x_j را به همان اندازه اضافه کنیم و این اپسیلون به حد کافی کوچک فرض شده‌است. با انجام این‌کار عبارت $\sum_{1 \leq i \leq n} x_i \times s_i$ بیشتر می‌شود که با فرض بیشینه بودن آن در تناقض است.
این عنصر خاص که $x_k \neq p_k, 0$ است را عنصر k ام در نظر بگیرید.
فرض اشیا با اندازه بزرگ را از ابتدا دور ریخته باشیم، یعنی i هایی که $p_i > B$ است را ابتدای کار دور انداخته‌ایم. حالا ادعا می‌کنیم اگر i هایی که $x_i = p_i$ هستند را برداریم، یک جواب تقریبی خوبی ارائه کرده‌ایم.

لم: اگر جواب به‌دست آمده برابر V باشد و جواب بهینه برابر OPT باشد داریم: $OPT \leq V \leq 2 \times OPT$
اثبات: می‌دانیم در حالت relaxation جواب محاسبه شده حتماً بهتر از جواب بهینه است، لذا $OPT \leq V$.
حال داریم:

$$V = \sum_{i:x_i=p_i} x_i \times s_i + x_k \times s_k \leq (\sum_{i:x_i=p_i} x_i \times s_i) + (p_k \times s_k)$$

می‌دانیم $p_k \times s_k \leq OPT$ زیرا خود عنصر k ام به تنهایی یک کاندید برای جواب است و $p_k \leq B$. همچنین بخش $\sum_{i:x_i=p_i} x_i \times s_i$ هم کاندیدی برای جواب است، پس:

$$V \leq \sum_{i:x_i=p_i} x_i \times s_i + p_k \times s_k \leq 2 \times OPT$$

بنابراین اگر $\frac{V}{2}$ را ارائه کنیم، یک ۲-تقریب به ما می‌دهد:

$$\frac{1}{2} OPT \leq \frac{V}{2} \leq OPT$$

تقریب ε-1 🎯

اکنون برمی‌گردیم به راه‌حل اولی که داشتیم و با استفاده از برنامه‌ریزی پویا مسئله را حل می‌کردیم و سعی می‌کنیم تا به‌وسیله آن ایده راه‌حلی برای یک تقریب بهتر ارائه کنیم. الگوریتم ارائه شده $O(n \times p_{max})$ بود. توجه کنید که ما پارامتری را با این الگوریتم می‌توانیم relax کنیم؛ پارامتر p یعنی اندازه اشیا است، در حالیکه می‌خواهیم تقریبی روی بیشینه ارزش انجام دهیم. به همین خاطر دنبال این هستیم که آیا می‌توانیم الگوریتمی ارائه دهیم که به‌جای اینکه $Poly(n, p_i)$ باشد، $Poly(n, s_i)$ باشد. در ادامه چنین چیزی نشان می‌دهیم.

روشی دیگر برای حل کوله‌پشتی با برنامه‌ریزی پویا 🔄

مسئله کوله‌پشتی‌مان را به مدل دیگری از کوله‌پشتی کاهش می‌دهیم. فرض کنید $S = \sum_{i \in [n]} s_i$ و حالا مسئله جدیدمان یک مسئله کوله‌پشتی است با این تفاوت که شئ iام اندازه‌اش برابر s_i و ارزشش برابر $-p_i$ است.

حال فرض کنید در مسئله جدید $B = S$ است و ما الگوریتم پویا که قبلاً به آن اشاره کردیم را روی مسئله اجرا می‌کنیم، با این تفاوت که $O(i, j)$ به معنای پر کردن کوله‌پشتی با بیشترین ارزش از اشیا ۱ تا i است به طوریکه جمع اندازه آن‌ها برابر «دقیقاً» j بشود. تنها کاری که کافیست بکنیم این است که پایه را تغییر دهیم:

$$O(0, 0) = 0, \forall_{i>0} O(0, i) = \infty$$

حال فرض کنید الگوریتم را اجرا کردیم و به ازای هر $0 \leq i \leq S$ مقدار $O(n, i)$ را بررسی می‌کنیم. این مقدار جمع یکسری $-p_i$ها را نشان می‌دهد:

$$O(n, i) = \sum_{k \in objects(O(n,i))} -p_k$$

اگر مقدار بالا بیشتر مساوی $-B$ بود، یعنی حداقل اندازه‌ای که بتواند ارزش i را بسازد کمتر مساوی B است.
بنابراین هر i که این شرط را داشت یعنی $O(n, i) \geq -B$، یعنی می‌توانیم ارزش i را استخراج کنیم با حفظ کردن شرایط مسئله.
کافیست با یک پیمایش از آخر به اول روی i بزرگترین مقداری را پیدا کنیم که $O(n, i) \geq -B$. در اینصورت جواب مسئله «اصلی» ما برابر i می‌شود!

حال می‌دانیم در روش قبلی به اندازه $O(np_{max})$ هزینه می‌کردیم. در این روش باید همه $O(i, j), 0 \leq i \leq n, 0 \leq j \leq S$ را بسازیم، یعنی $O(nS) < O(n \times ns_{max})$، بنابراین الگوریتمی $poly(n, s_{max})$ به دست آورده‌ایم. $s_{max} = \max_{i \in [n]} s_i$

ادامه کار و الگوریتم FPTAS ⚡

حال که الگوریتمی از $poly(n, s_{max})$ داریم، کافیست مقدار ϵ را به عنوان دقت وارد کار کنیم. یک پارامتر $k = \lfloor \frac{n}{\epsilon} \rfloor$ تعریف می‌کنیم و سپس ارزش شئ iام را با مقدار زیر جایگذاری می‌کنیم:

$$\lfloor \frac{s_i}{s_{max}} \times k \rfloor$$

و این مسئله را با روشی که ارائه کردیم حل می‌کنیم.

الان ارزش اشیا در بازه صفر تا $\frac{n}{\epsilon}$ است که نتیجه می‌دهد الگوریتم قبلی ما در $O(n^2 \times \frac{n}{\epsilon})$ عمل می‌کند و براساس پارامتر دقت و ورودی چند جمله‌ای است. اگر بخواهیم دقت را افزایش دهیم باید ϵ را کم کنیم و در نتیجه $\frac{n^3}{\epsilon}$ زیاد می‌شود.
دقت بیشتر یعنی هزینه زمانی بیشتر، اما نکته‌ای که وجود دارد این است که می‌توانیم به صورت دلخواه آن‌را تنظیم کنیم.
الگوریتم FPTAS for KnapSack (fully polynomial time approximation scheme) به این ترتیب عمل می‌کند که توسط Kim و Ibarra ارائه شد.

اثبات این الگوریتم:

فرض کنید $\hat{s}_i$ وزن‌های جدید ما هستند و OPT جواب بهینه است و $\hat{S}$ زیرمجموعه‌ای از عناصر است که ما به‌دست می‌آوریم.

$$\sum_{i \in \hat{S}} s_i \geq \sum_{i \in \hat{S}} (\lfloor \frac{s_i}{s_{max}} k \rfloor \frac{s_{max}}{k})$$

$$\sum_{i \in \hat{S}} s_i \geq (\frac{s_{max}}{k}) \sum_{i \in \hat{S}} \hat{s}_i$$

مقدار بالا ارزشی است که از الگوریتم FPTAS به دست می‌آید.

در ادامه ارزش الگوریتم OPT را بیان می‌کنیم:

$$(\frac{s_{max}}{k}) \sum_{i \in OPT} \lfloor \frac{s_i}{s_{max}} k \rfloor \geq (\frac{s_{max}}{k} (\sum_{i \in OPT} \frac{s_i}{s_{max}} k - 1))$$

$$(\frac{s_{max}}{k}) \sum_{i \in OPT} \lfloor \frac{s_i}{s_{max}} k \rfloor \geq \sum_{i \in OPT} s_i - \sum_{i \in OPT} \frac{s_{max}}{k}$$

$$(\frac{s_{max}}{k}) \sum_{i \in OPT} \lfloor \frac{s_i}{s_{max}} k \rfloor \geq \sum_{i \in OPT} s_i - \frac{n s_{max}}{k}$$

$$(\frac{s_{max}}{k}) \sum_{i \in OPT} \lfloor \frac{s_i}{s_{max}} k \rfloor \geq \sum_{i \in OPT} s_i - \epsilon s_{max}$$

می‌دانیم که $\epsilon s_{max} \leq \epsilon OPT$، که در نهایت داریم:

$$(\frac{s_{max}}{k}) \sum_{i \in OPT} \lfloor \frac{s_i}{s_{max}} k \rfloor \geq (1 - \epsilon) OPT$$

کاربرد روش‌های تقریبی در بهینه‌سازی گسسته (مطالعه بیشتر)

فرض کنید می‌خواهیم جواب دقیق مسئله کوله‌پشتی را با استفاده از روش backtracking به دست بیاوریم، به این‌صورت که در مرحله iام به ازای دو حالت انتخاب کردن و نکردن شئ متناظر، دو تا انشعاب به درخت backtracking می‌دهیم. چنین کاری به صورت نمایی زمان می‌برد و اصلا کارآمد نیست، اما با استفاده از روش‌های تقریبی‌ای که در این بخش آموختیم، نشان می‌دهیم روند را خیلی بهتر می‌توان کرد.

در این حوزه از "Heuristic" استفاده می‌کنیم. فرض کنید می‌خواهیم پیش‌بینی‌ای انجام بدهیم که با اشیائی که تا الآن انتخاب کرده‌ایم، بهینه‌ترین جواب ممکن با انتخاب بعدی‌ها چه «کرانی» دارد. توجه کنید به دست آوردن جواب دقیق به معنای پیمایش کل زیردرخت متناظر است، اما پیدا کردن کران هزینه کمتری می‌برد.

فرض کنید در رأس v درخت پسگرد (یا همان backtracking) هستیم و می‌خواهیم ببینیم جواب‌هایی که با گسترش این جواب به دست می‌آیند، حداکثر چند است. با استفاده از الگوریتمی نسبتا سریع و با انتخاب یک approximation مناسب، می‌توانیم این حدود را به‌دست آوریم و اگر مقدار به‌دست آمده که مقداری «خوش‌بینانه» است بدتر از جوابی بود که تا الآن در درخت پسگرد

دیده‌ایم، می‌توانیم به‌طور کلی زیر درخت v را پیمایش نکنیم!

این حرکت، حرکت مناسبی در روش‌های بهینه‌سازی است که به آن branch and bound نیز می‌گویند. برای مثال، اگر از یک α-approximation استفاده کنیم، آنوقت رأس‌هایی از درخت که مقدارشان بهتر مساوی αOPT است را می‌بینیم.

در ادامه کدی پیاده‌سازی شده که از الگوریتم redux برای بهینه‌سازی درخت پسگرد در آن استفاده شده‌است:

In [3]:
```python
# برای هرس کردن درخت redux حریصانه heuristic الگوریتم دقیق مسئله کوله‌پشتی با استفاده از backtracking و

INF = 10**18

# کلاس شیء شامل ارزش و اندازه
class Obj:
    def __init__(self, s, p):
        self.s = s  # ارزش
        self.p = p  # اندازه

# تابع مقایسه برای مرتب‌سازی بر اساس چگالی ارزش (s/p)
def cmp(obj):
    return -obj.s / obj.p  # منفی برای مرتب‌سازی نزولی

# تقریب حریصانه redux برای بازه [l, r)
def redux_heuristic(objs, l, r, B):
    return_value = 0
    sum_of_s = 0
    remaining_space = B
    for i in range(l, r):
        remaining_space -= objs[i].p
        if remaining_space < 0:
            return_value = max(objs[i].s, sum_of_s)
            break
        sum_of_s += objs[i].s
    return max(return_value, sum_of_s)

# حل دقیق مسئله با backtracking و هرس کردن با heuristic
def find_solution(objs, l, r, accumulated_s, remaining_space, best_solution=0):
    if l == r:
        return accumulated_s
    # اگر حتی با خوش‌بینانه‌ترین تقریب هم بهتر از جواب فعلی نمی‌شویم، این شاخه را حذف کن
    if redux_heuristic(objs, l, r, remaining_space) + accumulated_s < best_solution:
        return accumulated_s
    # با حالت بدون انتخاب شئ
    best_solution = max(best_solution, find_solution(objs, l + 1, r, accumulated_s, remaining_space, best_solution))
    # با حالت با انتخاب شئ (اگر جا داشته باشیم)
    if objs[l].p <= remaining_space:
        best_solution = max(best_solution, find_solution(objs, l + 1, r, accumulated_s + objs[l].s, remaining_space - objs[l].p, best_solution))
    return best_solution

# دریافت ورودی
n, B = map(int, input().split())
objs = []
for _ in range(n):
    p, s = map(int, input().split())
    if p <= B:  # حذف اشیاء بزرگ‌تر از ظرفیت
        objs.append(Obj(s, p))
n = len(objs)

# مرتب‌سازی اشیاء بر اساس چگالی ارزش
objs.sort(key=cmp)

# اجرای الگوریتم و چاپ جواب
print(find_solution(objs, 0, n, 0, B))
```

```
90
```

منابع 📖

- https://www.cs.cmu.edu/afs/cs/academic/class/15854-f05/www/scribe/lec10.pdf

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل یازدهم: مسائل سخت

بخش اول: روش پس‌گرد، تکنیک انشعاب و حد، آشنایی با درخت بازی

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

مقدمه 🚪

فرض کنید درون یک ساختمان و در اتاقی هستید که تعدادی «در» دارد. با گشودن هر در، یا از ساختمان خارج می‌شوید و یا وارد اتاق دیگری می‌شوید که تعدادی در دارد. هدف شما خروج از ساختمان است و می‌دانید که با گشودن حداکثر 10 در، موفق به این کار خواهید شد.

استراتژی پیشنهادی شما برای خروج از ساختمان چیست؟

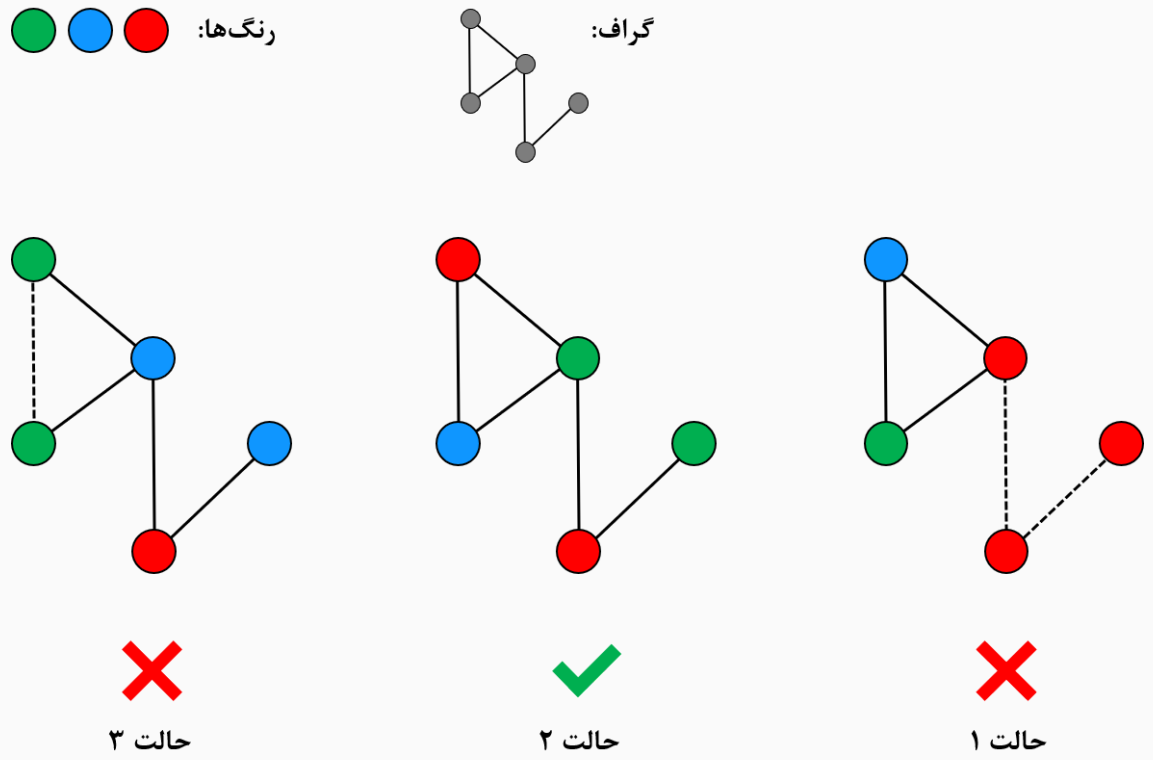
فهرست محتویات 🗂️

توصیف حالت‌های مسئله 🧩

هدف از روش پس‌گرد، پیمایش حالت‌های مختلف برای یافتن پاسخ مسئله است. در ادامه با ذکر چند مثال، مفهوم حالت‌های مسئله را مرور می‌کنیم.

مثال 1 – مسئله‌ی رنگ‌آمیزی گراف با k رنگ:

در این مسئله گراف G(V,E) و عدد k داده شده است. می‌خواهیم با استفاده از k رنگ، رأس‌های گراف را رنگ‌آمیزی کنیم؛ به شرطی که دو رأس مجاور، هم‌رنگ نباشند. در صورت ممکن بودن رنگ‌آمیزی، خروجی مسئله «بله» و در غیر این صورت «خیر» است.
«حالت»های این مسئله، رنگ‌آمیزی‌های مختلف رئوس گراف هستند. به شکل 1 توجه کنید.



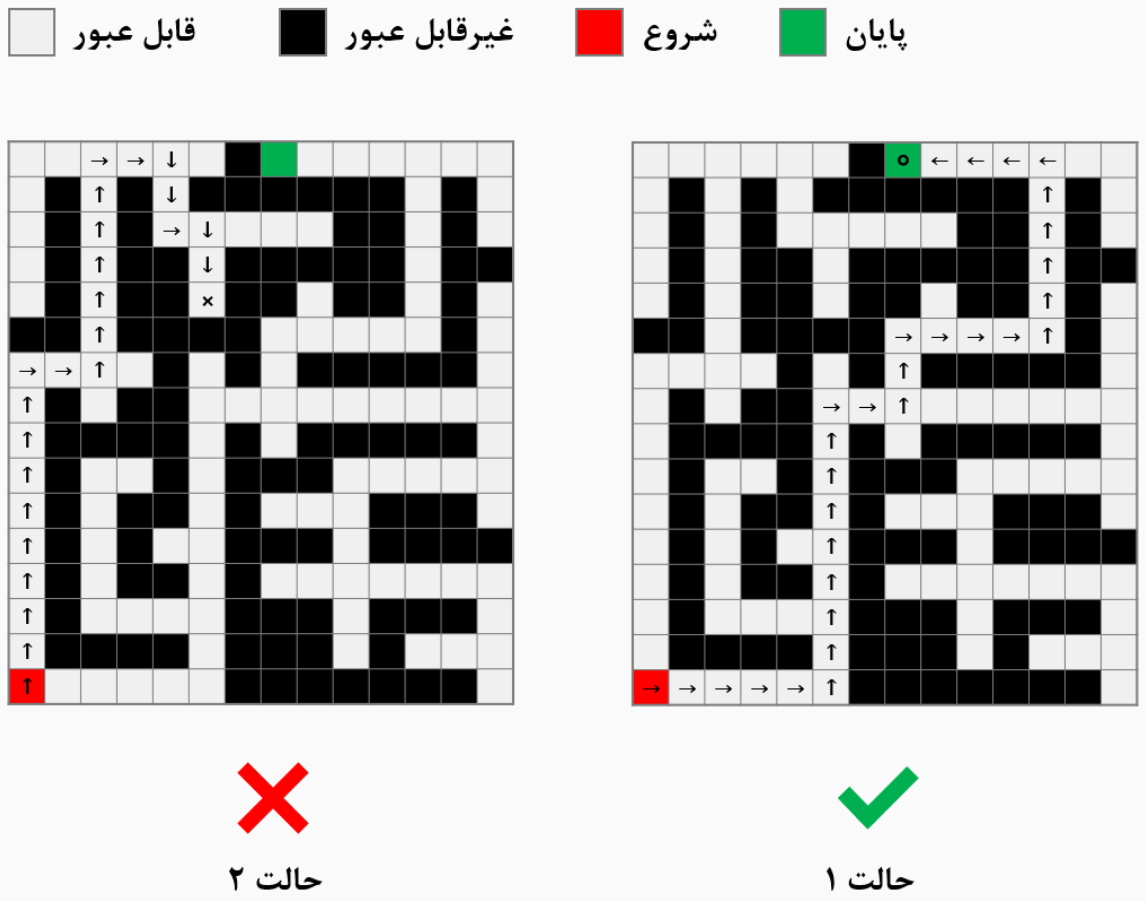
شکل 1 – سه حالت مختلف از مسئله‌ی رنگ‌آمیزی گراف

حالت‌های مختلف، ممکن است در شرط مسئله صدق کنند یا نکنند. در این مثال، اگر حالتی پیدا شود که شرط مسئله را برآورده کند، خروجی مسئله «بله» خواهد بود.

مثال 2 – مسئله‌ی عبور از ماز:

در این مسئله یک صفحه‌ی شطرنجی با خانه‌های قابل عبور و غیرقابل عبور به ما داده شده است. هم‌چنین دو خانه از صفحه به عنوان نقاط شروع و پایان مشخص شده‌اند. می‌خواهیم از نقطه‌ی شروع به نقطه‌ی پایان برسیم، به شرطی که فقط از خانه‌های قابل عبور رد شویم. در صورت وجود مسیر از نقطه‌ی شروع به نقطه‌ی پایان، خروجی مسئله «بله» و در غیر این صورت «خیر» است.

برای تعریف یک «حالت» در این مسئله، کافی است در هر خانه جهت حرکت بعدی را مشخص کنیم. به شکل 2 توجه کنید.



شکل 2 – دو حالت مختلف از مسئله‌ی عبور از ماز

فکر کنید:

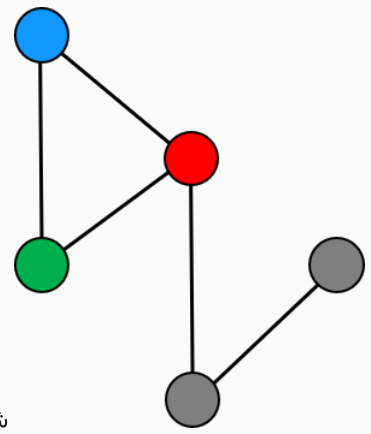
برای مسائل زیر، حالت‌های مسئله را شناسایی کنید:

N-Queens Problem, Sudoku, Kirkman Schoolgirls Problem

شبه‌حالت، متغیر و مقدار مجاز

برای توصیف هر حالت از تعدادی «متغیر» استفاده می‌کنیم. به عنوان مثال، در مسئله‌ی رنگ‌آمیزی گراف، به ازای هر رأس گراف یک متغیر داریم که می‌تواند مقادیر «قرمز»، «سبز» یا «آبی» را بپذیرد.

به ترکیبی که در آن زیرمجموعه‌ای از متغیرها مقداردهی شده باشند، «شبه‌حالت» گفته می‌شود. شکل 3 یک شبه‌حالت از مسئله‌ی رنگ‌آمیزی گراف را نشان می‌دهد.



شکل 3 – یک شبه‌حالت مجاز از مسئله‌ی رنگ‌آمیزی گراف

به شبه‌حالتی که شرط مسئله در آن نقض نشده باشد، «شبه‌حالت مجاز» گفته می‌شود. اگر با داشتن یک شبه‌حالت مجاز، یکی از متغیرهای باقی مانده را طوری مقداردهی کنیم که شرط مسئله نقض نشود، یک «شبه‌حالت مجاز» دیگر ایجاد می‌شود. به مقدار تعیین شده برای متغیر، «مقدار مجاز» می‌گوییم.

نکته:

انواع مسائل قابل حل با روش پس‌گرد عبارت‌اند از:

- **مسئله‌ی تصمیم‌گیری (Decision Problem):** بررسی وجود حالت مجاز. پاسخ این مسئله «بله» یا «خیر» است.
- **مسئله‌ی شمارش (Enumeration Problem):** شمارش تعداد حالت‌های مجاز. پاسخ این مسئله یک عدد است.
- **مسئله‌ی بهینه‌سازی (Optimization Problem):** انتخاب مناسب‌ترین حالت از میان مجموعه‌ی حالت‌ها. پاسخ این مسئله نیز یک عدد است.

البته در هر کدام از این مسائل، می‌توانیم حالت(های) متناظر با جواب را نیز به دست آوریم.

نکته:

روش پس‌گرد برای حل مسائلی مناسب است که دارای تعداد حالت‌های شمارا هستند.

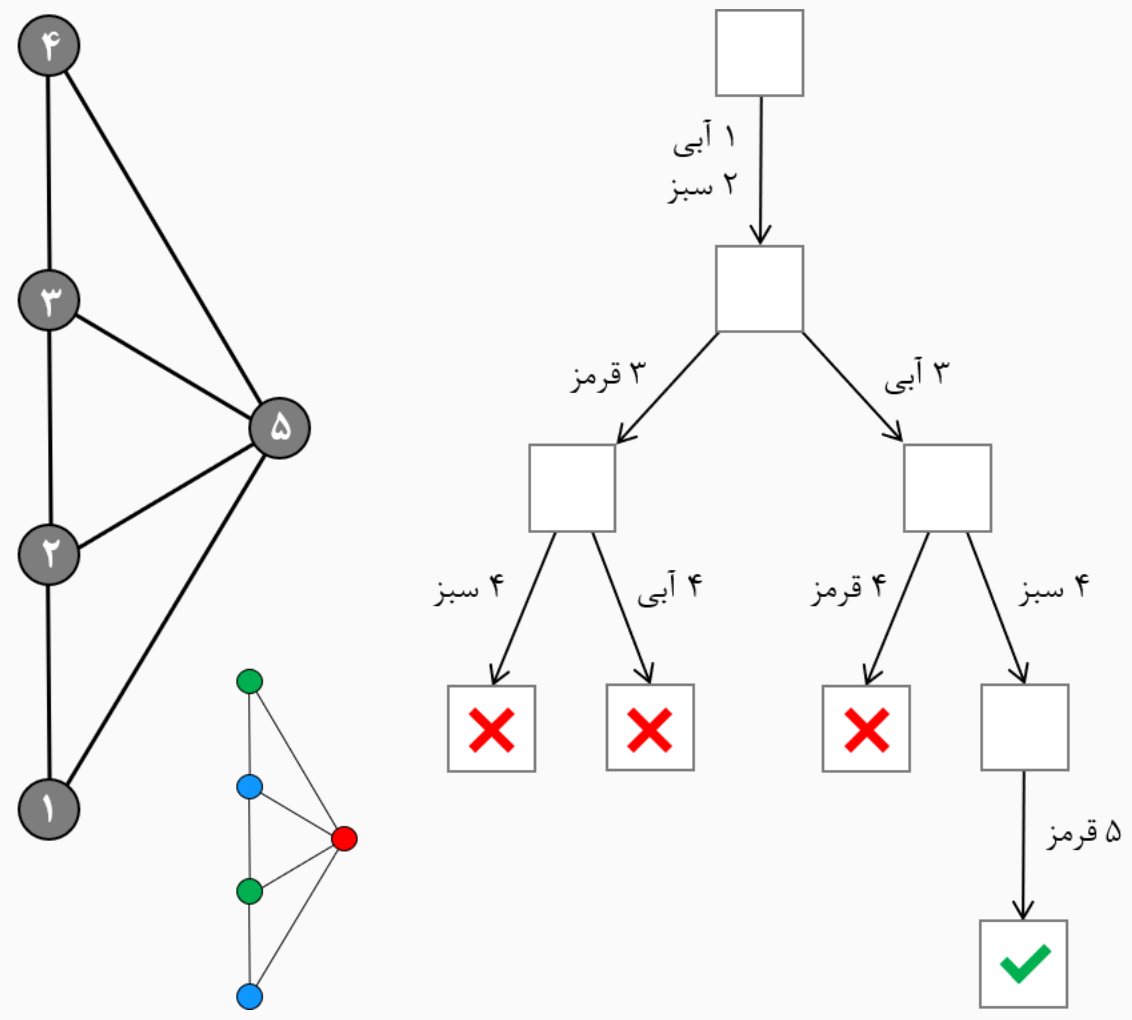
روش پس‌گرد 🔄

درخت پس‌گرد (Backtrack Tree) 🌳 : برای پیمایش حالت‌های مسئله از یک درخت استفاده می‌کنیم. هر «گره» از این درخت نشان دهنده‌ی یک شبه‌حالت مجاز برای مسئله است. هم‌چنین هر «شاخه» از درخت، مشخص شدن مقدار یک (یا چند) متغیر را نشان می‌دهد. ریشه‌ی درخت، شبه‌حالتی است که در آن مقدار هیچ متغیری مشخص نشده باشد.

در روش پس‌گرد، درخت پس‌گرد هم‌زمان با پیمایش ساخته می‌شود. از ریشه‌ی درخت شروع می‌کنیم و در هر مرحله با داشتن یک شبه‌حالت:

1. یکی از متغیرهای باقی مانده را انتخاب می‌کنیم.
2. به ازای تمامی مقادیر مجاز متغیر، یک شاخه به درخت اضافه می‌کنیم (انشعاب یا Branch) 🌿 .
3. یکی از شاخه‌ها را انتخاب کرده و با داشتن گرهِ فرزند، وارد مرحله‌ی بعد می‌شویم.

اگر به مرحله‌ای برسیم که برای یکی از متغیرهای باقی مانده هیچ مقدار مجازی موجود نباشد، روی درخت به عقب برمی‌گردیم (پس‌گرد یا Backtracking 🔙) تا به اولین مجموعه از شاخه‌های انتخاب نشده برسیم. سپس یکی از آن‌ها را انتخاب می‌کنیم. این کار را تا جایی انجام می‌دهیم که یک حالت مجاز تولید شود. شکل 4، درخت پس‌گرد را برای مسئله‌ی رنگ‌آمیزی گراف نشان می‌دهد.



شکل 4 – درخت پس‌گرد برای حل مسئله‌ی رنگ‌آمیزی گراف با 3 رنگ 🎨 (پیمایش اولیه به سمت چپ انجام شده است.)

نکته 📌 : برای تفکیک «گره»های درخت از «رأس»های گراف، آن‌ها را با مربع نشان می‌دهیم. هم‌چنین به تفاوت کاربرد کلمات «شاخه» و «یال» توجه کنید.

نکته 📌 : تا این لحظه در مورد «نحوه‌ی انتخاب متغیر باقی مانده» و «نحوه‌ی انتخاب مقدار مجاز» صحبتی نکرده‌ایم. اما به نظر می‌رسد ترتیب انتخاب متغیرها و مقادیر، بر زمان رسیدن به حالت مجاز، مؤثر باشد.

قطعه کد زیر، حل مسئله‌ی رنگ‌آمیزی گراف با 4 رنگ را نشان می‌دهد 💻 .

تکنیک انشعاب و حد 🔍

تکنیک انشعاب و حد (Branch and Bound) برای حل مسائل بهینه‌سازی به روش پس‌گرد استفاده می‌شود. در این مسائل، یافتن یک حالت مجاز کافی نیست، بلکه می‌خواهیم حالت مجازی را پیدا کنیم که یک مقدار خاص در آن بهینه باشد.

مثال 1 – مسئله‌ی رنگ‌آمیزی گراف با کمترین رنگ 🎨 **:** در این مسئله گراف G(V,E) داده شده است. می‌خواهیم با استفاده از کمترین تعداد رنگ، رأس‌های گراف را رنگ‌آمیزی کنیم؛ به شرطی که دو رأس مجاور، هم‌رنگ نباشند. خروجی مسئله کمترین تعداد رنگ‌های ممکن است.

برای تشکیل و پیمایش درخت پس‌گرد مسئله، به ازای هر گره درخت پس‌گرد، یک متغیر در نظر می‌گیریم. هم‌چنین یک متغیر سراسری (Global) نیز تعریف می‌کنیم. مقدار اولیه‌ی متغیر سراسری را بی‌نهایت در نظر می‌گیریم. متغیر هر گره نشان دهنده‌ی تعداد رنگ‌های استفاده شده تا آن مرحله (حداقل تعداد رنگ‌های مورد نیاز) و متغیر سراسری نشان دهنده‌ی حداکثر تعداد رنگ‌های مورد نیاز است. حال، الگوریتم پس‌گرد را به صورت زیر تغییر می‌دهیم. فرض کنید مقدار متغیر سراسری را با k نشان دهیم.

در هر مرحله از پیمایش، اگر در گرهی قرار داشتیم که مقدار متغیر کمتر از k بود، پیمایش را ادامه می‌دهیم تا به یک برگ برسیم. حال، مقدار متغیر برگ را به جای k قرار می‌دهیم (کران‌گذاری یا Bound). سپس، عملیات پس‌گرد را انجام می‌دهیم.

اگر در پیمایش درخت پس‌گرد، وارد گرهی شدیم که مقدار متغیر آن بیشتر از k بود، از ادامه‌ی پیمایش صرف نظر می‌کنیم و به روش پس‌گرد، مسیر دیگری را برای پیمایش انتخاب می‌کنیم.

می‌توانید حل این مسئله را در قطعه کد زیر ببینید 💻 .

مثال 2 – مسئله‌ی یافتن گروهک بیشین 🧩

تعریف گروهک: در یک گراف، به زیرگرافی که تمام رئوس آن دوبه‌دو مجاور باشند، گروهک گفته می‌شود.

در این مسئله گراف G(V,E) داده شده است و قصد داریم بیشترین اندازه‌ی ممکن برای یک گروهک را بیابیم. این مسئله را به **Integer Linear Programming** تبدیل می‌کنیم.

به ازای هر رأس i، یک متغیر دودویی (x_i) در نظر می‌گیریم که حضور یا عدم حضور آن در گروهک را نشان می‌دهد:

$$x_i \le 1 \ge 0$$

هم‌چنین به‌ازای هر دو رأس غیرمجاور i و j داریم:

$$x_i + x_j \le 1$$

هدف ما، بیشینه کردن حاصل جمع متغیرها است:

$$\max f = x_1 + x_2 + \cdots + x_n$$

n تعداد رأس‌های گراف است.

این مسئله را نیز می‌توانیم به کمک روش پس‌گرد 🔄 و تکنیک انشعاب و حد 🔍 حل کنیم. قطعه کد زیر، حل مسئله را نشان می‌دهد 💻.

In [1]:
```python
# الگوریتم پس‌گرد برای یافتن گروهک بیشینه در گراف بدون جهت

def is_clique(graph, subset):
    # بررسی اینکه آیا زیرمجموعه داده‌شده یک گروهک است یا نه
    for i in range(len(subset)):
        for j in range(i + 1, len(subset)):
            if subset[j] not in graph[subset[i]]:
                return False
    return True

def max_clique(graph, n):
    max_size = 0
    max_clique_set = []

    def backtrack(current, start):
        nonlocal max_size, max_clique_set
        if is_clique(graph, current):
            if len(current) > max_size:
                max_size = len(current)
                max_clique_set = current[:]
        for v in range(start, n):
            current.append(v)
            backtrack(current, v + 1)
            current.pop()

    backtrack([], 0)
    return max_size, max_clique_set
```

In [2]:
```python
# مثال

# تعریف گراف به صورت لیست مجاورت
# گراف زیر شامل یک گروهک از ۳ رأس است: 0-1-2
graph = {
    0: [1, 2],
    1: [0, 2],
    2: [0, 1, 3],
    3: [2, 4],
    4: [3]
}

n = 5  # تعداد رأس‌ها

size, clique = max_clique(graph, n)
print("Max clique size:", size)
print("Clique vertices:", clique)
```

```
Max clique size: 3
Clique vertices: [0, 1, 2]
```

آشنایی با بازی‌ها و دسته‌بندی آن‌ها 🎮

در ادامه‌ی این جزوه، قرار است با **درخت بازی** آشنا شوید. اطلاعات بیشتر و الگوریتم‌های مربوط آن در درس هوش مصنوعی ارائه می‌شوند. در ابتدا کمی راجع به بازی‌ها صحبت می‌کنیم و سپس، با درخت بازی آشنا می‌شوید.

روش‌های مختلفی برای دسته‌بندی بازی‌ها وجود دارد. مثلاً از این لحاظ که «آیا فرایندهای تصادفی نقشی در بازی ایفا می‌کنند یا خیر؟» یا «آیا هر بازیکن از کل حالت‌ها و اعمال گذشته و فعلی بازی مطلع است یا فقط بخشی از آن را می‌داند؟» می‌توان بازی‌ها را طبقه‌بندی کرد. هم‌چنین تعداد بازیکنان، مسئله‌ای تعیین‌کننده است. شکل 5، مثال‌هایی از بازی‌های مختلف را نشان می‌دهد.

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon, monopoly
imperfect information	stratego	bridge, poker, scrabble, nuclear war

Number of Players? 1, 2, …?

شکل 5 – دسته‌بندی بازی‌ها 🎲

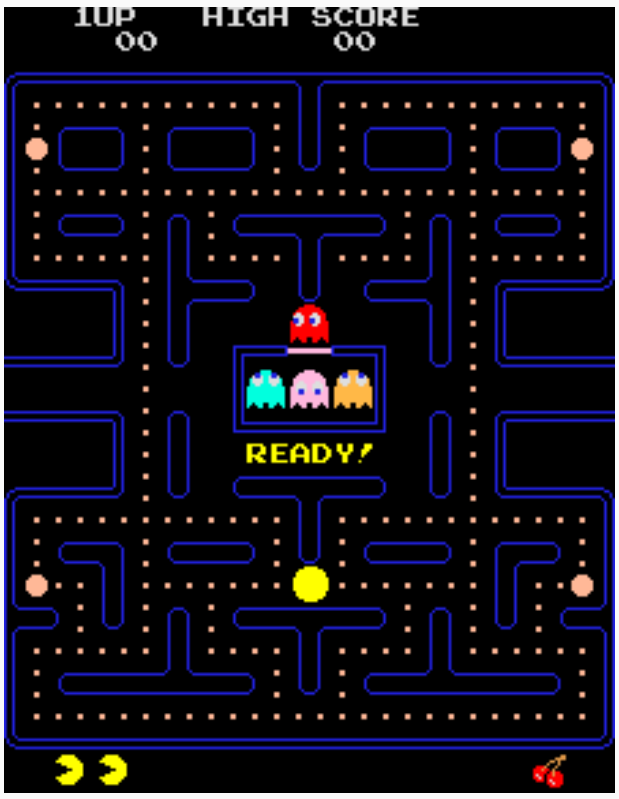
در یکی از روش‌هایی که برای توصیف بازی‌های قطعی (Deterministic) وجود دارد، بازی به کمک شش‌تایی زیر توصیف می‌شود:

- مجموعه‌ی حالت‌ها (**S**) (شروع با s_0)
- مجموعه‌ی بازیکنان (**P**) (معمولاً به صورت نوبتی بازی می‌کنند.)
- مجموعه‌ی کنش‌ها (**A**) (می‌تواند به حالت یا بازیکن بستگی داشته باشد.)
- تابع گذار، که به صورت $S \times A \to S$ تعریف می‌شود و نشان می‌دهد از هر حالت و با هر کنش، چگونه حالت بازی تغییر می‌کند.
- مجموعه‌ی حالت‌های پایانی (**F**)
- تابع پاداش به ازای هر حالت پایانی که به صورت $F \times P \to R$ تعریف می‌شود. **R** مجموعه‌ی پاداش‌های ممکن است.

هدف از انجام بازی، کسب بیشترین پاداش ممکن است 🏆 . «راه حل»، «استراتژی» یا «سیاست» یک بازی، تابعی به صورت $S \to A$ است که کنش انتخابی در هر حالت را مشخص می‌کند؛ به طوری که پاداش بازیکن، بیشینه شود.

مثال – بازی پَک‌مَن (Pac-Man) 👾

در این بازی پَک‌مَن و 4 لولوی مختلف در یک ماز قرار دارند. تعدادی نقطه نیز در صفحه وجود دارد. هدف بازی این است که پک‌من بدون برخورد با هیچ‌کدام از لولوها، تمام نقطه‌های موجود در صفحه را بخورد 🍒 .



شکل 6 – یکی از مراحل بازی پک‌من 🎮

- مجموعه‌ی حالت‌ها، برابر با ضرب دکارتی تمام مکان‌های مختلف برای قرارگیری پک‌من و لولوها و تمام حالت‌های مختلف نقطه‌ها (خورده شده یا خورده نشده) است.
- مجموعه‌ی بازیکنان، شامل پک‌من و لولوها است:

  $P = \{PacMan, Blinky, Pinky, Inky, Clyde\}$ هر کدام از لولوها دارای هوش مصنوعی هستند 🤖.
- مجموعه‌ی کنش‌ها، مستقل از حالت‌ها و بازیکنان، یک واحد حرکت به سمت چپ، راست، پایین یا بالا است ⬅️ ➡️ ⬆️ ⬇️.

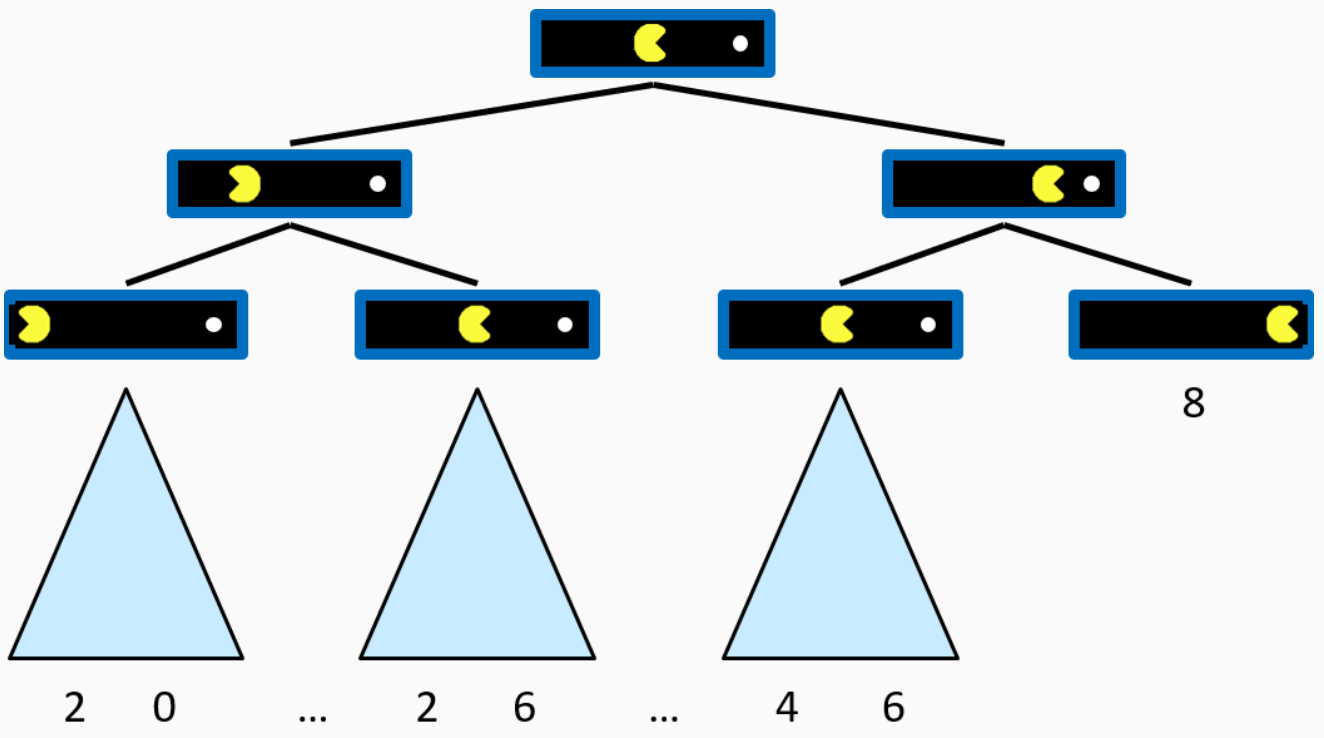
تمرین 1 📝**:** سعی کنید برای 3 جزء باقی مانده از بازی، تعریف مناسبی ارائه کنید.

تمرین 2 📝**:** فرض کنید پک‌من دارای تعدادی **جان** ❤️ است. به طوری که با برخورد با یک لولو، یکی از جان‌هایش کم می‌شود و به شرطی می‌میرد که همه‌ی جان‌هایش تمام شود. هم‌چنین هر چه قدر برای خوردن نقطه‌ها حرکات کمتری انجام شود، کاربر امتیاز بیشتری می‌گیرد. حال در تعاریف شش‌تایی، تغییرات لازم را اعمال کنید.

آشنایی با درخت بازی 🎮🌳

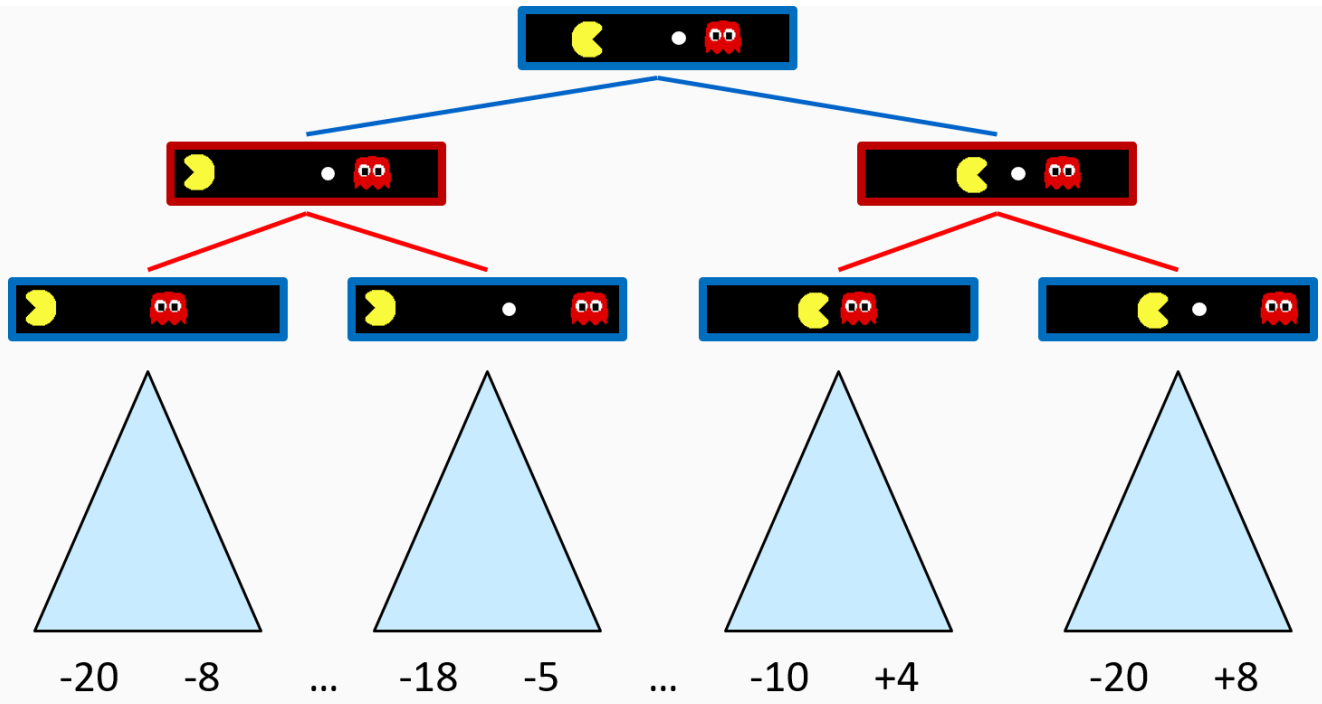
برای یافتن راه حل یک بازی، لازم است برای هر حالت، کنشی که منجر به بیشترین پاداش برای بازیکن می‌شود را پیدا کنیم. درخت بازی به ما کمک می‌کند این کار را انجام دهیم. گره‌های این درخت، حالت‌های بازی هستند و شاخه‌های خارج شده از هر گره، کنش‌های ممکن را نشان می‌دهند و به حالت حاصل ختم می‌شوند. برگ‌های این درخت، حالت‌های پایانی هستند. با تحلیل این درخت، می‌توان استراتژی مربوط به هر حالت را پیدا کرد. ادامه‌ی این مبحث در درس هوش مصنوعی ارائه می‌شود.

مثال 1 – پک‌من یک بُعدی بدون لولو 👾**:** شکل 7، نمونه‌ای از درخت بازی را برای نوع ساده‌ای از بازی پک‌من نشان می‌دهد. مقدار پاداشِ هر حالت پایانی، زیر آن نوشته شده است.



شکل 7 🏆

مثال 2 – پک‌من یک بُعدی با یک لولو 👾**:** در بازی‌هایی که چند بازیکن دارند، کنترل بازی معمولاً به صورت نوبتی بین بازیکنان جابه‌جا می‌شود. همان طور که در شکل 8 می‌بینید، در هر سطح از درخت بازی، یکی از بازیکنان حرکت می‌کند. به این نوع درخت بازی، **درخت بازی تقابلی** (Adversarial Game Tree) گفته می‌شود.



شکل 8 ⚔️

پیشنهاد 1 💡 **:** به این فکر کنید که «جست‌وجو برای استراتژی بهینه، چگونه با استفاده از درخت بازی انجام می‌شود؟». هم‌چنین، سعی کنید ارتباط میان جست‌وجوی پس‌گرد 🔄 و درخت بازی را بیابید.

پیشنهاد 2 💡 **:** در مورد انواع دیگر درخت بازی جست‌وجو کنید. آیا درخت بازی، فقط برای حل بازی‌های قطعی کاربرد دارد؟

مطالعه‌ی بیشتر

1. Introduction to Algorithms A Creative Approach (Udi Manber)
2. Artificial Intelligence A Modern Approach, Third Edition (Stuart Russell & Peter Norwig)

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل یازدهم: مسائل سخت

بخش دوم: بهینه سازی محدب یا کوژ

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 📑

مقدمه 📝

در بسیاری از مسائل برنامه‌نویسی پویا، به‌ویژه زمانی که تابع انتقال وابسته به مقادیر قبلی باشد، چالش اصلی در سرعت اجرای الگوریتم است. اگر برای محاسبه هر حالت مجبور باشیم تمام حالت‌های قبلی را بررسی کنیم، پیچیدگی زمانی الگوریتم به‌راحتی به $O(n^2)$ می‌رسد که برای ورودی‌های بزرگ قابل اجرا نیست.

یکی از تکنیک‌های قدرتمند برای کاهش این پیچیدگی، استفاده از «پوش محدب» است. این تکنیک زمانی کاربرد دارد که تابع انتقال به‌صورت خطی یا محدب باشد؛ یعنی اگر بتوانیم رابطه‌ی بازگشتی را به فرم خطی بنویسیم، می‌توانیم از هندسه پوش محدب برای بهینه‌سازی استفاده کنیم.

ایده‌ی اصلی این است که به‌جای نگه‌داشتن تمام حالت‌های قبلی، فقط آن‌هایی را نگه داریم که در آینده ممکن است بهترین باشند. این کار باعث می‌شود که در هر مرحله فقط با تعداد کمی از خطوط یا حالت‌ها سروکار داشته باشیم و بتوانیم در زمان لگاریتمی یا خطی، مقدار بهینه را پیدا کنیم.

این تکنیک در مسائل کلاسیکی مانند تقسیم‌بندی بهینه، کوله‌پشتی با وابستگی خطی، یا حتی برخی مسائل گرافی که در آن‌ها هزینه‌ی انتقال بین حالت‌ها به‌صورت خطی تعریف شده، کاربرد دارد. در واقع، پوش محدب یکی از ابزارهای کلیدی در طراحی الگوریتم‌های سریع برای مسائل سخت محسوب می‌شود.

در این فصل، ابتدا با مفهوم هندسی پوش محدب آشنا می‌شویم، سپس آن را به زبان برنامه‌نویسی پویا ترجمه می‌کنیم و در نهایت یک پیاده‌سازی ساده و قابل فهم از آن ارائه می‌دهیم. هدف این است که دانشجو بتواند این تکنیک را در مسائل مشابه تشخیص دهد و به‌درستی به‌کار گیرد.

بهینه‌سازی پوش محدب (Convex Hull Optimization) ⚡

در بسیاری از مسائل برنامه‌نویسی پویا، به‌ویژه زمانی که تابع انتقال به صورت خطی یا محدب باشد، می‌توان با استفاده از تکنیکی به نام «پوش محدب» سرعت اجرای الگوریتم را به‌طور چشم‌گیر کاهش داد. این تکنیک به‌جای بررسی تمام حالت‌های قبلی، فقط بهترین حالت‌ها را نگه می‌دارد و با استفاده از جستجوی دودویی یا صف دو سر، مقدار بهینه را سریع‌تر پیدا می‌کند.

فرض کنید در یک مسئله DP داریم:

$$dp[i] = \min(dp[j] + C[j] * A[i]) \quad \text{برای } j < i$$

این رابطه نشان می‌دهد که برای محاسبه dp[i] باید از بین تمام jهای قبلی، آن j را پیدا کنیم که مقدار dp[j] + C[j] * A[i] کمینه شود. اگر تعداد i زیاد باشد، این کار به‌صورت ساده زمان $O(n^2)$ خواهد داشت.

اما اگر تابع انتقال به‌گونه‌ای باشد که C[j] و A[i] به صورت خطی یا تابعی محدب باشند، می‌توان از هندسه پوش محدب استفاده کرد. در این حالت، هر عبارت dp[j] + C[j] * A[i] را می‌توان به‌صورت خطی در نظر گرفت: $y = m * x + b$ که در آن $m = C[j]$ و $b = dp[j]$.

بنابراین، مسئله به پیدا کردن کمترین مقدار از بین مجموعه‌ای از خطوط در نقطه $x = A[i]$ تبدیل می‌شود. این دقیقاً همان کاری است که پوش محدب انجام می‌دهد: نگهداری مجموعه‌ای از خطوط به‌گونه‌ای که بتوان در زمان لگاریتمی بهترین خط را در یک نقطه مشخص پیدا کرد.

برای این کار، خطوط را به ترتیب شیب اضافه می‌کنیم و هر خطی که توسط خطوط جدید «غیرمفید» شود را حذف می‌کنیم. این حذف با استفاده از بررسی تقاطع خطوط انجام می‌شود. اگر تقاطع خط جدید با خط قبلی، قبل از تقاطع خط قبلی با خط قبل‌تر باشد، خط وسط دیگر در هیچ نقطه‌ای بهینه نخواهد بود و باید حذف شود.

این تکنیک به دو روش قابل پیاده‌سازی است: با استفاده از جستجوی دودویی (برای حالت‌های آفلاین) یا با استفاده از صف دو سر (برای حالت‌های آنلاین). در این فصل، ما روش جستجوی دودویی را بررسی می‌کنیم.

کاربردهای پوش محدب بسیار گسترده‌اند. از جمله در مسائل زیر:

- بهینه‌سازی هزینه در زنجیره‌های تولید
- تقسیم‌بندی بهینه در مسائل تقسیم فایل یا حافظه
- حل سریع‌تر مسائل کلاسیک مانند کوله‌پشتی، مسیرهای بهینه، و حتی برخی مسائل گراف

در ادامه، پیاده‌سازی ساده‌ای از این تکنیک را مشاهده می‌کنید که برای حل یک DP خطی استفاده شده‌است. هدف این است که شما بتوانید با درک هندسی و الگوریتمی، این تکنیک را در مسائل مشابه به‌کار بگیرید و زمان اجرای الگوریتم را از $O(n^2)$ به $O(n \log n)$ کاهش دهید.

In [1]:
```python
# کلاس خط برای نمایش معادله به فرم y = m * x + b
class Line:
    def __init__(self, m, b):
        self.m = m  # شیب خط
        self.b = b  # عرض از مبدا

    def value(self, x):
        return self.m * x + self.b  # مقدار y در نقطه x

# بررسی اینکه آیا خط وسط غیرمفید شده و باید حذف شود
def is_bad(l1, l2, l3):
    # غیرمفید است l2 باشد، یعنی l2 و l1 قبل از تقاطع l3 و l1 اگر تقاطع
    return (l3.b - l1.b) * (l1.m - l2.m) < (l2.b - l1.b) * (l1.m - l3.m)

# کلاس پوش محدب برای نگهداری خطوط بهینه
class ConvexHullTrick:
    def __init__(self):
        self.lines = []

    def add_line(self, m, b):
        new_line = Line(m, b)
        # حذف خطوط غیرمفید از انتهای لیست
```

```
        while len(self.lines) >= 2 and is_bad(self.lines[-2], self.lines[-1], new_line):
            self.lines.pop()
        self.lines.append(new_line)

    def query(self, x):
        # جستجوی دودویی برای پیدا کردن بهترین خط در نقطه x
        low, high = 0, len(self.lines) - 1
        while low < high:
            mid = (low + high) // 2
            if self.lines[mid].value(x) < self.lines[mid + 1].value(x):
                high = mid
            else:
                low = mid + 1
        return self.lines[low].value(x)

# مثال تست برای حل dp[i] = min(dp[j] + C[j] * A[i])
A = [1, 2, 3, 4, 5]  # ورودی‌های A[i]
C = [5, 4, 3, 2, 1]  # ضرایب C[j]
dp = [0] * len(A)    # مقدارهای dp[i]

cht = ConvexHullTrick()
cht.add_line(C[0], dp[0])  # افزودن خط اولیه

for i in range(1, len(A)):
    dp[i] = cht.query(A[i])  # پیدا کردن مقدار بهینه با پوش محدب
    cht.add_line(C[i], dp[i])  # افزودن خط جدید به پوش محدب

print("dp =", dp)
```

```
dp = [0, 10, 15, 20, 25]
```

طراحی الگوریتم‌ها

دانشگاه آزاد شیراز - دانشکده مهندسی کامپیوتر

نیم‌سال دوم سال تحصیلی ۱۴۰۵-۱۴۰۴

فصل یازدهم: مسائل سخت

بخش سوم: DQ & Knuth Optimization

استاد: دکتر امین اسکندری

تیم طراحی محتوای آموزشی و ویراستاری (علمی و ادبی): حمید نامجو و امیرحسین همتی

فهرست محتویات 📑

بهینه‌سازی تقسیم و حل (Divide and Conquer Optimization) ⚡

در بعضی مسائل DP به رابطه بازگشتی‌ای مثل:

$$dp(i,j) = \min_{k \le j}\{dp(i-1,k) + C(k,j)\}$$

بر می‌خوریم که در آن **C(k, j)** یک تابع هزینه است. اگر i بتواند از ۱ تا n و j از ۱ تا m باشد و بتوانیم مقدار C را در $O(1)$ محاسبه کنیم، می‌توانیم جواب مسئله را به سادگی در $O(nm^2)$ محاسبه کنیم که برای آن $O(nm)$ استیت برای DP داریم و هر استیت در $O(m)$ محاسبه می‌شود. می‌خواهیم ببینیم آیا می‌توان این DP را سریع‌تر محاسبه کرد 🔍.

فرض کنید **opt(i,j)** برابر با مقداری از k باشد که dp(i,j) به ازای آن از dp(i-1,k) محاسبه می‌شود. اگر برای هر i و j، داشته باشیم که **opt(i,j) ≤ opt(i,j+1)**، می‌توانیم از بهینه‌سازی تقسیم و حل استفاده کنیم. به این شرط، شرط **یکنوایی** گفته می‌شود.

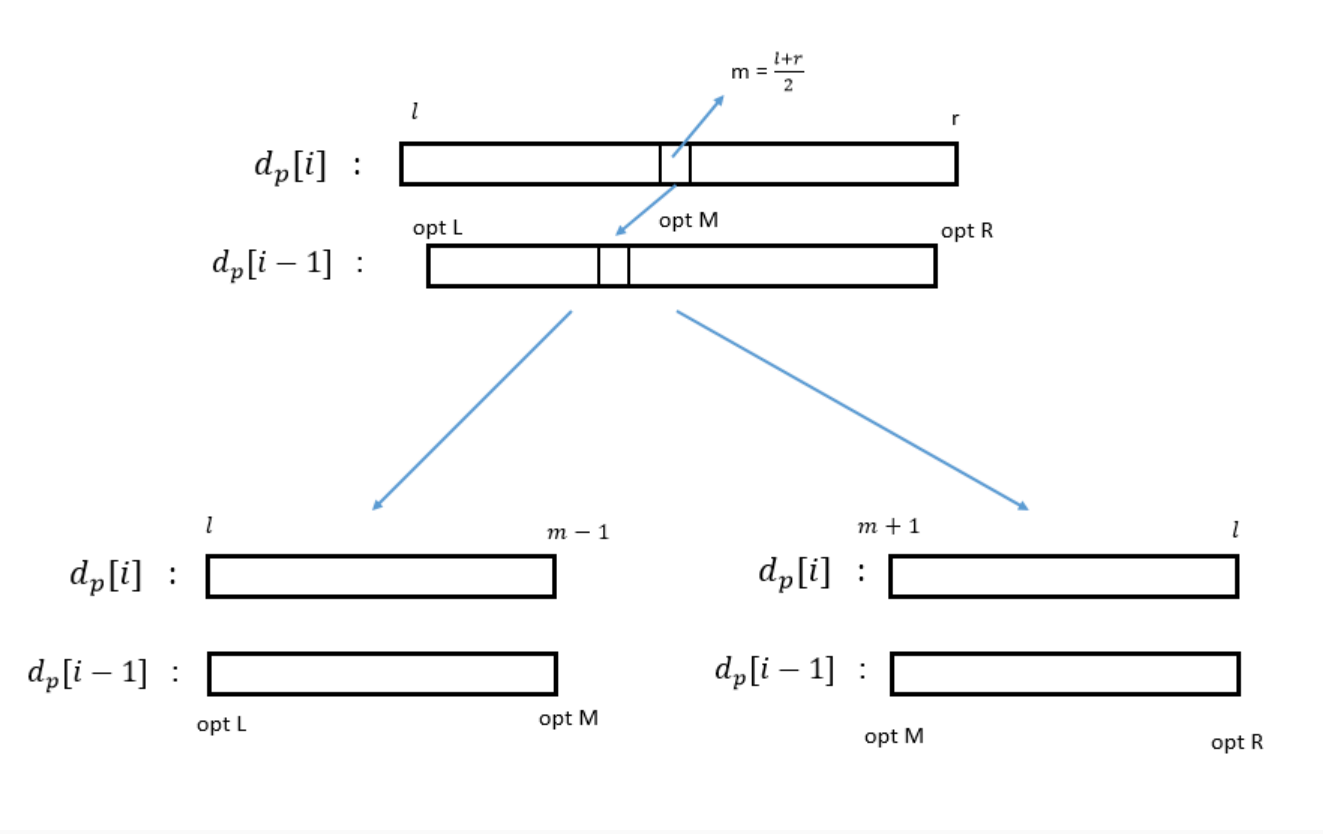
فرض کنید مقادیر DP را برای i-1 می‌دانیم و می‌خواهیم آن را برای i به دست آوریم. حال اگر مقدار opt(i,j) را برای یک j داشته باشیم، برای هر j' > j کافی است رابطه بازگشتی را برای k ≤ opt(i,j) چک کنیم.

برای استفاده از این خاصیت و کاهش زمان، می‌توانیم از ایده تقسیم و حل استفاده کنیم. ابتدا dp(i, m/2) را محاسبه و opt(i, m/2) را به دست می‌آوریم. اگر این مقدار برابر optM باشد، جواب برای j > m/2 با استفاده از k ≥ optM و برای j < m/2 با استفاده از k ≤ optM محاسبه می‌شود. سپس مسئله را بازگشتی برای دو نیمه حل می‌کنیم و مقادیر DP را در دو طرف به دست می‌آوریم. این کار را تا رسیدن به تمامی مقادیر ادامه می‌دهیم.

در این روش، در هر طبقه محاسبه DPها از O(m) زمان می‌برد، چون بازه‌ها تنها در سر و ته اشتراک دارند. همچنین با نصف شدن اندازه بازه‌ها، تعداد طبقات O(log m) می‌شود و مقادیر DP در O(n m log m) محاسبه می‌شوند ⚡.

نمونه‌ای از مسئله: n نفر در یک شهربازی در صف چرخ‌وفلک هستند. می‌خواهیم آن‌ها را به k دسته تقسیم کنیم تا در k کابین متفاوت سوار شوند. هر دسته شامل بازه‌ای متوالی از افراد است. هر فرد دوست ندارد با افراد ناآشنا در یک کابین باشد و برای هر

دو نفر یک سطح ناآشنایی داریم. هدف، کمینه کردن جمع ناآشنایی‌ها است. می‌توان برای این مسئله dp(i,j) را به عنوان کمینه جمع ناآشنایی‌های ممکن اگر j نفر اول صف به i دسته تقسیم شوند، در نظر گرفت. در این صورت اگر C(i,j) برابر جمع ناآشنایی افراد nم تا jم صف باشد، شرط یکنوایی برقرار است و جواب در O(n k log n) محاسبه می‌شود ✅ .



در پایین یک پیاده‌سازی کلی آمده است که با استفاده از مقادیر dp(i-1,x)، مقادیر dp(i,y) را به دست می‌آورد 💻 .

In [2]:
```python
import sys

INF = sys.maxsize

# تابع هزینه C(i, j) – در این مثال فرضی، مجموع مربعات فاصله بین i و j است
def C(i, j):
    # این تابع باید متناسب با مسئله واقعی تعریف شود
    # برای مثال تستی، فرض می‌کنیم هزینه برابر با مربع فاصله بین i و j باشد
    return (j - i) ** 2

# تابع اصلی تقسیم و حل برای محاسبه new_dp از old_dp
def compute(l, r, optl, optr, old_dp, new_dp):
    if l > r:
        return

    mid = (l + r) // 2
    best_cost = INF
    best_k = -1

    # فقط بازه [optl, optr] را بررسی می‌کنیم به‌جای کل kها
    for k in range(optl, min(mid, optr) + 1):
        cost = old_dp[k] + C(k, mid)
        if cost < best_cost:
            best_cost = cost
            best_k = k

    new_dp[mid] = best_cost
    optm = best_k

    # حل بازگشتی برای نیمه چپ و راست
    compute(l, mid - 1, optl, optm, old_dp, new_dp)
    compute(mid + 1, r, optm, optr, old_dp, new_dp)

# مثال تستی برای اجرای الگوریتم
n = 10  # تعداد موقعیت‌ها
old_dp = [i * i for i in range(n)]  # فرضی: مقدارهای قبلی DP
new_dp = [0] * n  # مقدارهای جدید DP که باید محاسبه شوند

compute(0, n - 1, 0, n - 1, old_dp, new_dp)

print("مقادیر جدید DP:")
print(new_dp)
```

```
مقادیر جدید DP:
[0, 1, 2, 5, 8, 13, 18, 25, 32, 41]
```

بهینه‌سازی نوث (Knuth Optimization) ⚡

در برخی مسائل برنامه‌نویسی پویا، رابطه بازگشتی به شکل زیر است:

$$dp[i][j] = \min_{k \in [i,j)} \{dp[i][k] + dp[k+1][j] + C(i,j)\}$$

که در آن $C(i,j)$ تابع هزینه‌ای است که برای ترکیب بازه $[i,j]$ پرداخت می‌شود. این نوع رابطه در مسائل کلاسیکی مانند ضرب زنجیره‌ای ماتریس‌ها، فشرده‌سازی فایل‌ها، یا تقسیم‌بندی بهینه دیده می‌شود.

اگر بتوانیم نشان دهیم که برای هر $i < j < k < l$ داریم:

$$opt[i][j] \leq opt[i][k] \leq opt[j][l]$$

آنگاه می‌توانیم از تکنیک Knuth Optimization استفاده کنیم. این شرط به نام **چهارضلعی (Quadrangle Inequality)** یا **Monotonicity of Decision** شناخته می‌شود.

در این تکنیک، به‌جای بررسی تمام kهای ممکن برای هر dp[i][j]، فقط بازه‌ای از kها را بررسی می‌کنیم که از opt[i][j-1] تا opt[i+1][j] هستند. این باعث می‌شود که زمان اجرای الگوریتم از $O(n^3)$ به $O(n^2)$ کاهش یابد.

این روش به‌ویژه زمانی مفید است که تابع هزینه $C(i,j)$ دارای خاصیت محدب یا زیرمجموعه‌ای از آن باشد. در ادامه، پیاده‌سازی این تکنیک را برای یک مسئله کلاسیک مشاهده می‌کنید 💻.

In [1]:
```python
# پیاده‌سازی بهینه‌سازی نوث (Knuth Optimization) برای حل مسائل DP با ساختار خاص

INF = float('inf')

# تابع هزینه C(i, j) – در این مثال فرض می‌کنیم مجموع عناصر بین i و j است
def cost(prefix_sum, i, j):
    return prefix_sum[j + 1] - prefix_sum[i]

# پیاده‌سازی Knuth Optimization
def knuth_optimization(arr):
    n = len(arr)
    dp = [[0] * n for _ in range(n)]
    opt = [[0] * n for _ in range(n)]

    # پیشوند مجموع برای محاسبه سریع هزینه‌ها
    prefix_sum = [0] * (n + 1)
    for i in range(n):
        prefix_sum[i + 1] = prefix_sum[i] + arr[i]

    # مقداردهی اولیه برای بازه‌های طول 1
    for i in range(n):
        dp[i][i] = 0
        opt[i][i] = i

    # حل بازه‌های طول بیشتر
    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            dp[i][j] = INF
            # فقط بازه بین opt[i][j-1] تا opt[i+1][j] را بررسی می‌کنیم
            left = opt[i][j - 1]
            right = opt[i + 1][j] if i + 1 <= j else j
            for k in range(left, right + 1):
                if k < j:
                    val = dp[i][k] + dp[k + 1][j] + cost(prefix_sum, i, j)
                    if val < dp[i][j]:
                        dp[i][j] = val
                        opt[i][j] = k

    return dp[0][n - 1]

# مثال تست: آرایه‌ای از اعداد که می‌خواهیم آن را بهینه تقسیم کنیم
arr = [4, 2, 7, 6, 9]
print("کمینه هزینه:", knuth_optimization(arr))
```

```
کمینه هزینه: 62
```