Answer-Quiz-3

Algorithm Design - Spring 2026

Instructor: Dr. Amin Eskandari

Head Teaching Assistants: Hamid Namjoo, Amir Hossein Hemati

Teaching Assistants: Ali Nikvan, Viyana Hasanbeigi, Nima Soltani, Reza Rezaie, Maryam Meidanshahi , Babak Janfaza , Elham Mosavi

---

سوال ۱ :

پاسخ الف)

این ایده یک روش حرصانه است، چون در هر مرحله بهترین انتخاب محلی را انجام می‌دهیم: یعنی فایلی را برمی‌داریم که در ازای هر واحد ظرفیت، بیشترین ارزش را ایجاد می‌کند. چنین انتخابی زمانی درست است که بتوان نشان داد انتخاب محلی را می‌توان به یک جواب بهینه‌ی سراسری تعمیم داد.

در مسئله‌ی کوله‌پشتی کسری، این کار درست است، چون:

- اگر بتوان بخشی از فایل را گرفت، دیگر مجبور نیستیم یک فایل را کامل برداریم یا رها کنیم.
- بنابراین همیشه منطقی است که ابتدا فایلی را انتخاب کنیم که بیشترین بازده ارزش به ازای هر واحد وزن را دارد.
- اگر ظرفیت باقی‌مانده کمتر از وزن فایل باشد، فقط همان بخش موردنیاز را برمی‌داریم.

انتخابی که در لحظه بهترین به نظر می‌رسد، می‌تواند با بهینگی نهایی سازگار باشد. پس در این حالت، انتخاب بر اساس نسبت $value/weight$ یک استراتژی مناسب و بهینه است.

پاسخ ب)

خیر، در حالت غیرکسری، همان استراتژی حریصانه همیشه به جواب بهینه نمی‌رسد.

دلیلش این است که وقتی فایل‌ها قابل تقسیم نیستند، انتخاب یک فایل بزرگ با نسبت خوب ممکن است باعث شود ظرفیت باقی‌مانده برای فایل‌های دیگر از بین برود و در نتیجه ترکیب نهایی از نظر ارزش کل، بهینه نباشد.

گاهی انتخاب محلیِ خوب، مانع رسیدن به جواب جهانیِ خوب می‌شود.

به طور مفهومی:

- در مسئله‌ی کسری، تصمیم‌ها انعطاف‌پذیرند.
- اما در مسئله‌ی غیرکسری، انتخاب‌ها گسسته هستند.
- بنابراین باید همه حالت‌های مهم بررسی شوند تا مشخص شود کدام ترکیب از فایل‌ها بیشترین ارزش را می‌دهد.

به همین دلیل، برای این نوع مسئله معمولاً از برنامه‌ریزی پویا استفاده می‌شود؛ چون برنامه‌ریزی پویا می‌تواند:

- زیرمسئله‌ها را بررسی کند،
- نتایج تکراری را ذخیره کند،
- و از بررسی دوباره‌ی حالت‌های مشابه جلوگیری کند.

پس در این حالت، روش حریصانه تضمین بهینگی ندارد و DP مناسب‌تر است.

پاسخ ج)

الگوریتم هافمن یک نمونه‌ی کلاسیک از روش حرصانه است.

منطق آن این است:

- کاراکترهایی که فراوانی کمتری دارند، اگر کدشان کمی بلندتر شود، تأثیر کمتری بر طول متوسط کل دارند.
- در مقابل، کاراکترهای پرتکرار باید کدهای کوتاه‌تری داشته باشند تا مجموع طول کد کاهش یابد.

پس در هر مرحله، ادغام دو کاراکتر کم‌فراوانی منطقی است، چون:

۱. آن‌ها در پایین‌ترین سطح درخت قرار می‌گیرند.

۲. هزینه‌ی بلند شدن کد آن‌ها کم‌اهمیت‌تر است.

۳. این کار باعث می‌شود گره‌های پرتکرار در لایه‌های بالاتر و با کدهای کوتاه‌تر قرار بگیرند.

بنابراین، انتخاب حریصانه‌ی دو کمینه در هر مرحله به ساخت یک درخت بهینه‌ی prefix-free منجر می‌شود و طول متوسط کدها را کمینه می‌کند.

پاسخ د)

۱. زیرمسئله‌های هم‌پوشان یعنی چه؟

یعنی در روند حل مسئله، یک زیرمسئله‌ی یکسان بارها و بارها به وجود می‌آید.

مثلاً در محاسبه‌ی فیبوناچی، مقدارهای $F(n-1)$ و $F(n-2)$ بارها از مسیرهای مختلف دوباره درخواست می‌شوند.

این همان چیزی است که در برنامه‌های بازگشتی ساده باعث تکرار زیاد و اتلاف زمان می‌شود.

۲. برنامه‌ریزی پویا چگونه مشکل را حل می‌کند؟

DP نتایج زیرمسئله‌ها را یک‌بار محاسبه و ذخیره می‌کند.

بعد هر وقت همان زیرمسئله دوباره لازم شد، به‌جای محاسبه‌ی مجدد، مقدار ذخیره‌شده را استفاده می‌کند.

این کار باعث می‌شود:

- محاسبات تکراری حذف شوند،
- زمان اجرا به‌طور چشمگیری کاهش یابد،
- و الگوریتم برای ورودی‌های بزرگ عملی شود.

سوال ۲ :

الف)

ظرفیت حافظه:

$$W = 7$$

جدول بسته‌ها:

	ارزش	وزن بسته
P_1	2	12
P_2	3	20
P_3	4	24
P_4	5	30

هدف: انتخاب زیرمجموعه‌ای از بسته‌ها به‌طوری‌که مجموع وزن از ۷ بیشتر نشود و مجموع ارزش بیشینه گردد.

بررسی حالت‌های ممکن

- فقط P_1:

وزن = ۲ ، ارزش = ۱۲

- فقط P_2:

وزن = ۳ ، ارزش = ۲۰

- فقط P_3:

وزن = ٤ ، ارزش = ۲٤

- فقط P_4:

وزن = ٥ ، ارزش = ۳۰

- $P_1 + P_2$:

وزن = $2 + 3 = 5$

ارزش = $12 + 20 = 32$

- $P_1 + P_3$:

وزن = $2 + 4 = 6$

ارزش = $12 + 24 = 36$

- $P_1 + P_4$:

وزن = $2 + 5 = 7$

ارزش = $12 + 30 = 42$

- $P_2 + P_3$:

وزن = $3 + 4 = 7$

ارزش = $20 + 24 = 44$

- $P_2 + P_4$:

وزن = $3 + 5 = 8 \leftarrow$ غیرمجاز

- $P_3 + P_4$:

وزن = $4 + 5 = 9 \leftarrow$ غیرمجاز

- $P_1 + P_2 + P_3$:

وزن = $2 + 3 + 4 = 9 \leftarrow$ غیرمجاز

- $P_1 + P_2 + P_4$:

وزن = $2 + 3 + 5 = 10 \leftarrow$ غیرمجاز

- $P_1 + P_3 + P_4$:

وزن = $2 + 4 + 5 = 11 \leftarrow$ غیرمجاز

- $P_2 + P_3 + P_4$:

وزن = $3 + 4 + 5 = 12 \leftarrow$ غیرمجاز

بهترین انتخاب

بیشترین ارزش مجاز برابر است با:

که از انتخاب

P_2 و P_3

به‌دست می‌آید.

۱. بسته‌های انتخاب‌شده:

$$P_2, P_3$$

۲. مجموع وزن:

$$3 + 4 = 7$$

۳. مجموع ارزش بیشینه:

$$20 + 24 = 44$$

ب)

دو رشته داده شده‌اند:

$$X = A\ C\ B\ D\ A\ B$$

$$Y = B\ C\ A\ D\ B$$

می‌خواهیم طول بلندترین زیردنباله‌ی مشترک را پیدا کنیم.

مرحله ۱: تعریف DP

اگر $L[i][j]$طول LCS بین پیشوندهای

$$X[1..i]\text{و}Y[1..j]$$

باشد، آنگاه:

- اگر $X_i = Y_j$، داریم:

$$L[i][j] = L[i-1][j-1] + 1$$

- اگر $X_i \neq Y_j$، داریم:

$$L[i][j] = \max\left(L[i-1][j], L[i][j-1]\right)$$

مرحله ۲: تشکیل جدول

رشته‌ها:

- $X = A, C, B, D, A, B$
- $Y = B, C, A, D, B$

جدول DP:

		0	B	C	A	D	B
	0	0	0	0	0	0	0
A	0	0	0	1	1	1	1
C	0	0	1	1	1	1	1
B	0	1	1	1	1	1	2
D	0	1	1	1	1	2	2
A	0	1	1	1	2	2	2
B	0	1	1	1	2	2	3

پس طول LCS برابر است با:

$$L[6][5] = 3$$

مرحله ۳: بازسازی یکی از جواب‌ها

از جدول می‌توان یکی از LCSهای ممکن را به‌دست آورد. مثلاً:

CDB

یا

CAB

هر دو طول ۳ دارند.

ج)

یال‌ها و وزن‌ها:

- $A - B = 2$
- $A - C = 3$
- $A - D = 6$
- $B - C = 1$
- $B - D = 4$
- $C - D = 5$

برای یافتن MST، یال‌ها را به ترتیب صعودی مرتب می‌کنیم:

۱. $B - C = 1$
۲. $A - B = 2$
۳. $A - C = 3$
٤. $B - D = 4$
٥. $C - D = 5$
٦. $A - D = 6$

اجرای انتخاب یال‌ها

- یال $B - C$با وزن ۱ را انتخاب می‌کنیم.
- یال $A - B$با وزن ۲ را انتخاب می‌کنیم.

- یال $A-C$با وزن ۳ را بررسی می‌کنیم، اما باعث تشکیل دور بین A, B, Cمی‌شود، پس رد می‌شود.
- یال $B-D$با وزن ٤ را انتخاب می‌کنیم.

اکنون همه‌ی ٤ رأس به هم متصل شده‌اند و ۳ یال داریم، پس MST کامل است.

درخت پوشای کمینه

$$\{B-C, A-B, B-D\}$$

هزینه کل

$$1+2+4=7$$

سوال ۳ :

۱ –

الف) مراحل الگوریتم جانسون:

۱ .یک رأس مجازی جدید مثل s به گراف اضافه می‌کنیم و از s به همه رأس‌های دیگر یال با وزن صفر اضافه می‌کنیم.

۲ .الگوریتم بلمن-فورد را از مبدأ s اجرا می‌کنیم تا کوتاه‌ترین فاصله از s به هر رأس v را به عنوان پتانسیل h(v) محاسبه کنیم.

۳ .برای هر یال (u,v) با وزن اصلی w(u,v) ، وزن جدید w'(u,v) را به صورت w(u,v) + h(u) - h(v) تعریف می‌کنیم. این وزن‌ها همگی نامنفی می‌شوند.

۴ .برای هر رأس u در گراف اصلی، یک بار الگوریتم دیجسترا را با وزن‌های جدید w' اجرا می‌کنیم تا کوتاه‌ترین فاصله از u به همه رأس‌ها را بدست آوریم.

۵ .فاصله واقعی از u به v برابر می‌شود با فاصله بدست آمده از دیجسترا منهای h(u) به اضافه h(v).

چرا اضافه کردن رأس مجازی و بلمن-فورد ضروری است؟
برای محاسبه پتانسیل‌هایی که تضمین کنند $w'(u,v) \geq 0$ باشد. بلمن-فورد می‌تواند وجود دور منفی را هم تشخیص دهد (اگر دور منفی وجود داشته باشد، الگوریتم جانسون قابل اجرا نیست).

چرا تبدیل وزن با پتانسیل، کوتاه‌ترین مسیرها را حفظ می‌کند؟
برای هر مسیر از u به v مانند $v_0,u_1,v_2,...,v_k$ ، مجموع وزن‌های جدید برابر است با مجموع وزن‌های اصلی به اضافه $h(u)-h(v)$. $h(u)-h(v)$ چون جمله $h(u)-h(v)$ برای همه مسیرهای بین یک جفت رأس مشخص ثابت است، ترتیب کوتاه‌ترین مسیرها تغییر نمی‌کند.

ب)

```
class FibonacciHeap:

    def _singleton(self, key, value):

        node = FibonacciNode(key, value)

        node.left = node.right = node

        return node


    def _merge(self, a, b):

        if a is None: return b

        if b is None: return a

        a_right = a.right

        b_left = b.left

        a.right = b

        b.left = a

        a_right.left = b_left

        b_left.right = a_right

        return a
```

```
def insert(self, key, value):
    node = self._singleton(key, value)
    if self.min_node is None:
        self.min_node = node
    else:
        self.min_node = self._merge(self.min_node, node)
        if node.key < self.min_node.key:
            self.min_node = node
    self.total_nodes += 1


def removeMinimum(self):
    if self.min_node is None:
        return None
    min_node = self.min_node
    # اضافه کردن فرزندان مینیمم به لیست ریشه
    if min_node.child is not None:
        child = min_node.child
        while True:
            child.parent = None
            child = child.right
            if child == min_node.child:
```

```
            break
        self.min_node = self._merge(self.min_node, min_node.child)


    if min_node.right == min_node:
        self.min_node = None
    else:
        self.min_node = min_node.right
        min_node.left.right = min_node.right
        min_node.right.left = min_node.left
        self._consolidate()
    self.total_nodes -= 1
    return min_node
```

توضیح یک خطی کار removeMinimum پس از برداشتن مینیمم:
درخت‌های موجود در لیست ریشه را بر اساس درجه مرتب کرده و درخت‌هایی با درجه یکسان را با هم ادغام می‌کند تا حداکثر درجه هر گره از O(log n) بیشتر نشود.

ج) حالت اول (دیجسترا با آرایه ساده):
زمان کل الگوریتم جانسون $O(n^3) = O(n^2 + n \times m)$ = در بدترین حالت

حالت دوم (دیجسترا با فیبوناچی هیپ):
زمان کل$O(n \times m + n^2 \log n)$ =

توضیح تفاوت:
فیبوناچی هیپ عملیات decreaseKey و extractMin را به ترتیب O(1) و O(log n) به طور سرشکن انجام می‌دهد. برای گراف‌های sparse ($m \approx n$) مرتبه می‌شود $O(n^2 \log n)$ که بهتر از $O(n^3)$ است.
برای گراف‌های dense ($m \approx n^2$) تفاوت چندانی ندارد.

د) دو حالتی که فیبوناچی هیپ نسبت به هیپ دودویی برتری عملی ندارد:

۱ .گراف‌های sparse با تعداد یال کم(m ≈ n) ، چون هزینه ثابت بالای فیبوناچی هیپ باعث می‌شود هیپ دودویی در عمل سریع‌تر کار کند.

۲ .زمانی که تعداد عملیات decreaseKey کم است، مثلاً در دیجسترا روی گراف‌هایی با وزن‌های صحیح کوچک که با آرایه ساده هم بهینه می‌شوند

۲_

الف)

```
def closest_pair(points):
    points = sorted(points, key=lambda p: p[0])

    def dist(p1, p2):
        return ((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)**0.5

    def recursive_closest(px):
        if len(px) <= 3:
            return min((dist(px[i], px[j]), (px[i], px[j]))
                       for i in range(len(px)) for j in range(i+1, len(px)))

        mid = len(px) // 2
        mid_x = px[mid][0]
        left = recursive_closest(px[:mid])
        right = recursive_closest(px[mid:])
        d, pair = min(left, right, key=lambda x: x[0])
```

```
    strip = [p for p in px if abs(p[0] - mid_x) < d]
    strip.sort(key=lambda p: p[1])

    for i in range(len(strip)):
        for j in range(i+1, min(i+7, len(strip))):
            if dist(strip[i], strip[j]) < d:
                d = dist(strip[i], strip[j])
                pair = (strip[i], strip[j])
    return (d, pair)

return recursive_closest(points)
```

ب)

شبه‌کد الگوریتم فلوید-وارشال:

1. dist = ماتریس n×n با مقدار ∞
2. for i in range(n): dist[i][i] = 0
3. for هر یال (u,v,w): dist[u][v] = w
4. for k in range(n):
5. for i in range(n):
6. for j in range(n):
7. dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

مرتبه زمانی$O(n^3)$:

دو سناریوی واقعی برای استفاده از فلوید-وارشال با وجود کند بودن آن:

سناریوی ۱: گراف چگال با $n \leq 500$ که نیاز به کوتاه‌ترین مسیر بین همه جفت‌ها داریم و کدنویسی ساده و عدم وجود حافظه اضافی مهم است.

دلیل: پیاده‌سازی فلوید-وارشال فقط ۱۰-۵ خط کد است، در حالی که جانسون و دیجستراهای مکرر کد پیچیده‌تری دارند.

سناریوی ۲: روی سخت‌افزارهای موازی یا GPU که ضرب ماتریس بهینه شده است.

دلیل: فلوید-وارشال شبیه ضرب ماتریس است و روی GPU می‌توان آن را بسیار سریعتر از $O(n^3)$ معمولی اجرا کرد.

ج)

```
def count_paths_with_limit(graph, s, t, L):
    n = len(graph)
    visited = [False] * n
    count = 0

    def dfs(current, length):
        nonlocal count
        if length > L:
            return
        if current == t and length <= L:
            count += 1
        for neighbor in graph[current]:
            if not visited[neighbor]:
```

```
            visited[neighbor] = True
            dfs(neighbor, length + 1)
            visited[neighbor] = False

    visited[s] = True
    dfs(s, 0)
    return count
```

د) تحلیل پیچیدگی زمانی (بدترین حالت):
O(n × (d-1)^(L-1))که d میانگین درجه گراف است. در گراف کامل
بدترین حالت O(n × (n-1)^(L-1)) می‌شود.

سوال ۴ :

بخش اول <<< KMP و Divide & Conquer

الگو:

P = "ABABACA"

رشته متن:

T = "ABABABACABABACA"

سوال ۱) محاسبه آرایه LPS و اجرای KMP

تعریف LPS

LPS[i] برابر است با طول بلندترین Prefix که همزمان Suffix نیز باشد برای زیررشته‌ای که تا اندیس i ادامه دارد

اندیس‌ها >>> ٠ ١ ٢ ٣ ٤ ٥ ٦

کاراکترها >>> A C A B A B A

محاسبه آرایه LPS:

LPS[0] = 0

در اندیس ١

B با A برابر نیست

LPS[1] = 0

در اندیس ٢

A با A برابر است

LPS[2] = 1

در اندیس ٣

B با B برابر است

LPS[3] = 2

در اندیس ٤

A با A برابر است

LPS[4] = 3

در اندیس ۵

C با B برابر نیست

بازگشت به مقدار LPS قبلی انجام می‌شود

در نهایت هیچ Prefix و Suffix مشترکی وجود ندارد

LPS[5] = 0

در اندیس ٦

A با A برابر است

LPS[6] = 1

آرایه نهایی LPS

[۰, ۰, ۱, ۲, ۳, ۰, ۱]

اجرای الگوریتم KMP

متن >>> ABABABACABABACA

الگو >>> ABABACA

مقایسه‌ها از ابتدای متن و الگو شروع می‌شود

پنج کاراکتر اول با موفقیت تطبیق پیدا می‌کنند

A=A

B=B

A=A

B=B

A=A

سپس عدم تطابق رخ می‌دهد

در متن >>> B

در الگو >>> C

در این حالت الگوریتم به جای شروع مجدد از آرایه LPS استفاده می‌کند

مقدار جدید j برابر است با >>> LPS[4] = 3

بنابراین جستجو از موقعیت مناسب ادامه پیدا می‌کند و مقایسه‌های تکراری حذف می‌شوند

در ادامه تمام کاراکترها تطبیق پیدا می‌کنند و الگو در متن پیدا می‌شود

محل شروع تطابق:

اندیس ۸ متن

نکته مهم(الگوریتم KMP با استفاده از اطلاعات قبلی، از انجام مقایسه‌های اضافی جلوگیری می‌کند)

پیچیدگی زمانی الگوریتم:

$$O(n + m)$$

سوال ۲) حل رابطه بازگشتی

رابطه داده شده

$$T(n) = 2T(n/2) + n \log n$$

پارامترهای رابطه

a = 2

b = 2

f(n) = n log n

محاسبه مقدار بحرانی:

n^(log_b a) = n^(log_2 2) = n

چون f(n) = n log n بزرگ‌تر از n است، حالت دوم Master Theorem برقرار می‌شود

نتیجه نهایی:

$$T(n) = \Theta(n \log^2 n)$$

نکته مهم:

اگرچه KMP دارای پیچیدگی خطی است، اما تقسیم داده‌ها و ادغام نتایج در روش Divide & Conquer می‌تواند سربار اضافی ایجاد کند و در بعضی شرایط باعث کاهش کارایی عملی سیستم شود

بخش دوم الگوریتم Strassen

سوال ۱) تحلیل الگوریتم Strassen

در ضرب معمولی ماتریس‌های ۲×۲ به ۸ ضرب نیاز داریم

الگوریتم Strassen با استفاده از ترکیب‌های هوشمندانه جمع و تفریق، تعداد ضرب‌ها را به ۷ کاهش می‌دهد

اهمیت این کاهش:

در الگوریتم‌های بازگشتی، کاهش تعداد ضرب‌ها در هر مرحله تأثیر بسیار زیادی روی پیچیدگی نهایی دارد

رابطه بازگشتی Strassen

$$T(n) = 7T(n/2) + O(n^2)$$

پارامترها >>> $a = 7$ و $b = 2$

پیچیدگی زمانی

$$T(n) = \Theta(n^{\log_2 7})$$

مقدار تقریبی

$$\Theta(n^{2.81})$$

مقایسه:

ضرب کلاسیک $O(n^3)$

Strassen سریع‌تر از ضرب معمولی عمل می‌کند

سوال ۲) بررسی عملکرد عملی Strassen

الگوریتم Strassen علاوه بر ضرب، تعداد زیادی عملیات جمع و تفریق انجام می‌دهد

در سخت‌افزارهایی که هزینه عملیات جمع بالا باشد

۱) حافظه بیشتری مصرف می‌شود
۲) هزینه بازگشت‌های بازگشتی افزایش پیدا می‌کند
۳) سربار اجرایی بیشتر می‌شود

در اندازه‌های کوچک ورودی، الگوریتم ضرب معمولی ممکن است سریع‌تر از Strassen باشد

در سیستم‌های واقعی عواملی مانند Cache و Memory Access و Parallelism هزینه سخت‌افزار نیز اهمیت زیادی دارند

بخش سوم Ford–Fulkerson

ظرفیت یال‌ها:

S → A = 12

S → B = 8

A → B = 5

A → T = 7

B → T = 10

سوال ۱) اجرای الگوریتم Ford–Fulkerson

مرحله اول:

مسیر افزایشی <<< S → A → T

ظرفیت مؤثر مسیر <<< min(12, 7) = 7

بنابراین ۷ واحد جریان ارسال می‌شود

ظرفیت‌های باقی‌مانده:

S → A = 5

A → T = 0

مرحله دوم:

مسیر افزایشی <<< S → B → T

ظرفیت مؤثر >>> min(8, 10) = 8

بنابراین ۸ واحد جریان دیگر ارسال می‌شود

جریان کل >>> (۷+۸)

پس حداکثر جریان شبکه = ۱۵

نکته مهم:

انتخاب نامناسب مسیرهای افزایشی می‌تواند تعداد مراحل اجرای الگوریتم را به شدت افزایش دهد

به همین دلیل در الگوریتم Edmonds-Karp از BFS برای انتخاب مسیر مناسب استفاده می‌شود

سوال ۲) تحلیل پیچیدگی کلی سیستم

پیچیدگی KMP

$$O(n)$$

پیچیدگی Strassen:

$$O(n^{2.81})$$

پیچیدگی Ford–Fulkerson:

در حالت کلی

$$O(E \times MaxFlow)$$

در Edmonds-Karp

$$O(VE^2)$$

گلوگاه اصلی سیستم معمولاً بخش Strassen یا Ford–Fulkerson ، زیرا KMP دارای پیچیدگی خطی است و نسبت به دو بخش دیگر بسیار سریع‌تر عمل می‌کند

Ford–Fulkerson زمانی پرهزینه‌تر می‌شود که

a. تعداد یال‌ها زیاد باشد
b. شبکه متراکم باشد
c. مقدار جریان بیشینه بزرگ باشد

در سیستم‌های امنیتی، مهاجم می‌تواند ورودی‌هایی تولید کند که الگوریتم را وارد بدترین حالت ممکن کنند

به همین دلیل تحلیل Worst Case اهمیت بسیار بیشتری نسبت به Average Case دارد