Quiz-1

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 , Fatemeh Dehgan , Elham Mosavi

---

پاسخ سوال اول

(الف)

تعریف: در یک آرایه از اعداد حقیقی، زیردنباله‌ی متوالی (بازه‌ای پیوسته) با بیشترین مجموع اعضا را می‌یابیم.

الگوریتم :با یک متغیر current شروع می‌کنیم (مقدار اولیه ۰). برای هر عنصر x ، ابتدا اگر current منفی بود آن را ۰ می‌کنیم (زیرا شروع مجدد بهتر است)، سپس current += x و بیشینه‌ی دیده شده‌max را به‌روز می‌کنیم.

اجرا روی آرایه:

[-2, 3, -1, 2, -4, 5, -2, 3]

- بعد از -2: current=0, max=0
- بعد از 3: current=3, max=3
- بعد از -1: current=2, max=3
- بعد از 2: current=4, max=4
- بعد از -4: current=0, max=4
- بعد از 5: current=5, max=5
- بعد از -2: current=3, max=5
- بعد از 3: current=6, max=6

پاسخ نهایی: ۶ (زیردنباله‌ی [5, -2, 3] یا [3, -1, 2, -4, 5, -2, 3] مجموع 6 دارد)

(ب)

مرتب‌سازی بازه‌ها بر اساس زمان پایان:

(2,3), (1,4), (3,6), (5,7), (6,9), (8,10)

- انتخاب (2,3)

- (1,4) تداخل دارد (3>1) ← حذف

- (3,6) شروع=3 برابر پایان 3 ←مجاز (فرض بازه‌های نیمه‌باز) ←انتخاب

- (5,7) تداخل دارد (6>5) ←حذف

- (6,9) شروع=6 برابر پایان 6 ← انتخاب

- (8,10) تداخل دارد (9>8) ← حذف

تعداد انتخاب‌شده: ۳ کار (2,3), (3,6), (6,9)

دلیل بهینگی: با انتخاب زودترین پایان، فضای بیشتری برای بازه‌های بعدی باقی می‌ماند.

(ج)

چرا الگوریتم حریصانه برای سکه‌های ۱، ۵ و ۱۰ تومانی جواب بهینه می‌دهد؟

در سیستم پولی {۱, ۵, ۱۰}، اگر سکه‌ها را به‌ترتیب صعودی مرتب کنیم (۱, ۵, ۱۰)، هر سکه مضرب صحیحی از سکه ماقبل خود است:

- ۵ = ۵ × ۱
- ۱۰ = ۲ × ۵

این یک شرط کافی برای بهینگی الگوریتم حریصانه در مسئله «خرد کردن پول» (یا «کمترین تعداد سکه») است. به بیان دقیق‌تر، در چنین سیستم‌هایی، هر انتخاب حریصانه (برداشتن بزرگ‌ترین سکه ممکن) باعث نمی‌شود که در ادامه مجبور به استفاده از سکه‌های بیشتری نسبت به جواب بهینه شویم. به‌عبارت دیگر، اگر از یک سکه بزرگ‌تر صرف‌نظر کنیم، حداکثر می‌توانیم همان مقدار را با چند

سکه کوچک‌تر جایگزین کنیم که تعدادشان به‌اندازه ضریب تبدیل است (مثلاً یک سکه ۱۰ با دو سکه ۵)، و این تعداد هرگز از خود سکه بزرگ‌تر کمتر نیست. بنابراین حریصانه همواره بهینه عمل می‌کند.

مثال نقض: مجموعه سکه‌ای که الگوریتم حریصانه در آن جواب بهینه نمی‌دهد

مجموعه سکه‌های {۱, ۴, ۵} را در نظر بگیرید. مقدار هدف = ۸ تومان.

- روش حریصانه:
  بزرگ‌ترین سکه ≤ ۸، سکه ۵ است → برداشتن ۵.
  باقی‌مانده: ۳ تومان.
  بزرگ‌ترین سکه ≤ ۳، فقط ۱ است → سه سکه ۱ تومانی.
  مجموع سکه‌ها: ۵ + ۱ + ۱ + ۱ = ۴ سکه.
- جواب بهینه:
  دو سکه ۴ تومانی → ۴ + ۴ = ۸ تومان → ۲ سکه.

از آنجا که ۲ <۴، الگوریتم حریصانه در این مثال جواب بهینه (کمترین تعداد سکه) را ارائه نمی‌دهد. این نشان می‌دهد که شرط «هر سکه مضرب سکه قبلی باشد» یک شرط اساسی برای بهینگی حریصانه در این مسئله است و با شکستن آن (مثلاً وجود ۴ که مضرب ۱ است اما ارتباطش با ۵ خاصیت مضربی ندارد)، حریصانه ممکن است به خطا برود.

(د)

رشته‌ها: X="AGGTAB", Y="GXTXAYB"

رابطه: اگر X[i]==Y[j] → dp[i][j]=dp[i-1][j-1]+1، در غیر این صورت dp[i][j]=max(dp[i-1][j], dp[i][j-1]).

محاسبه: LCS برابر GTAB یا GXAB با طول ۴ است. (مثلاً G, T, A, B).

پاسخ سوال دوم

(الف)

گام‌های پریم از A:

۱. S={A}، یال‌های خروج: A-B(4) ,A-D(3) ,A-C(2) → کمترین A-C(2) اضافه.

۲. S={A,C}، یال‌های خروج: A-B(4), C-D(3), C-E(5) ,A-D(3) → کمترین وزن ۳ (بین A-D و C-D). فرض می‌کنیم یال A-D را انتخاب می‌کنیم (در صورت تساوی می‌توان هرکدام را گرفت). اضافه A-D(3).

۳. S={A,C,D}، یال‌های خروج: A-B(4), C-E(5), D-E(1), D-F(6) → کمترین D-E(1) اضافه.

۴. S={A,C,D,E}، یال‌های خروج از S به خارج (خارج = { → A-B(4), E-F(2), D-F(6)}): کمترین E-F(2) اضافه.

۵. S={A,C,D,E,F}، یال‌های خروج: A-B(4) ← اضافه (B تنها رأس باقیمانده).

MST: یال‌های (A-B:4) ,(E-F:2) ,(D-E:1) ,(A-D:3) ,A-C:2)) هزینه کل = 2+3+1+2+4 = ۱۲.

(ب)

مرتب‌سازی یال‌ها (وزن صعودی):

(1) D-E(1)

(2) A-C(2)، E-F(2) (هر دو وزن ۲)

(3) A-D(3)، C-D(3)

(4) A-B(4)، B-C(4)

(5) C-E(5)، B-E(5)

(6) D-F(6)

اجرای کروسکال:

- اضافه D-E(1)

- اضافه A-C(2)

- اضافه) E-F(2)هنوز بدون دور)

- بررسی A :A-D(3) در {A,C}، D در { → D,E,F}متفاوت، اضافه می‌شود.

- بررسی C :C-D(3) و D اکنون در یک مجموعه { A,C,D,E,F}هستند ← ایجاد دور، حذف.

- بررسی A :A-B(4) در مجموعه بزرگ، B تنها ← اضافه (اکنون ۵ یال برای ۶ رأس، MST کامل)

بقیه یال‌ها حذف می‌شوند.

MST: (D-E, A-C, E-F, A-D, A-B)با هزینه 1+2+2+3+4 = ۱۲.

درخت کروسکال با درخت پریم که در بخش (الف) به دست آمد یکسان است (هر دو شامل یال‌های A-C, A-D, D-E, E-F, A-B می‌باشند).

(ج)

اگر وزن‌ها متمایز باشند، MST یکتاست. در الگوریتم پریم، در هر گام کم‌وزن‌ترین یال خروجی به طور یکتا تعیین می‌شود (زیرا وزن‌ها تکراری نیستند). این انتخاب مستقل از رأس شروع، نتیجه نهایی یکسانی دارد (چون ویژگی برش برای هر افراز، یک یال منحصربه‌فرد با کمترین وزن معرفی می‌کند). در الگوریتم کروسکال نیز مرتب‌سازی یال‌ها ترتیب مشخصی دارد و تصمیم‌گیری برای اضافه کردن یال بدون ابهام است. بنابراین هر دو الگوریتم همان درخت یکتا را تولید می‌کنند.

(د)

الگوریتم پریم با انتخاب بزرگ‌ترین یال خروجی در هر مرحله، یک درخت پوشا تولید می‌کند (چون همیشه یال اضافه می‌کند و نهایتاً به همه رأس‌ها می‌رسد). آیا این درخت، درخت پوشای بیشینه (MST ماکزیمم) است؟ بله، با همان منطق ویژگی برش برای بیشینه‌سازی: بزرگ‌ترین یال هر برش در یک MST ماکزیمم قرار دارد.

مثال ساده: مثلث با رأس‌های A,B,C و یال‌های AB=1, BC=2, AC=3.

الگوریتم پریم بیشینه از A:

- S={A}، بزرگ‌ترین خروجی A-C(3)اضافه.

- S={A,C}، خروجی‌ها: C-B(2) و A-B(1) → بزرگ‌ترین 2 (IC-B) اضافه.

درخت حاصل: AC(3) و CB(2) با مجموع 5.

درخت پوشای بیشینه همین است (سایر درخت‌ها: AB+BC=3, AB+AC=4). بنابراین الگوریتم درخت پوشای بیشینه را می‌دهد.

جواب سوال سوم

بخش الف: دلیل بهینه بودن کد هافمن

کد هافمن تولید شده از درخت باینری است که کاراکترهای با فرکانس بالاتر در سطوح بالاتر (کوتاه‌تر) و کاراکترهای با فرکانس کمتر در سطوح پایین‌تر (بلندتر) قرار می‌گیرند.

- این خاصیت باعث می‌شود داده‌هایی که بیشتر تکرار می‌شوند کدهای کوتاه‌تری داشته باشند که میانگین طول کد را کمینه می‌کند.
- همچنین ساختار بدون پیشوند بودن یعنی هیچ کد کوتاه‌تر نمی‌تواند پیشوند کد بلندتر باشد و باعث ابهام نشود.
- قضیه اثبات می‌کند که هیچ الگوریتم دیگری برای کدگذاری بدون پیشوند نمی‌تواند میانگین طول کد بهتری بسازد.

بخش ب: ساخت درخت کد هافمن با Heap

مراحل الگوریتم:

1. همه کاراکترها را به صورت گره‌های مجزا با وزن برابر فرکانسشان در یک heap مین وارد کنید.
2. تا زمانی که فقط یک گره در heap نماند:
    - دو گره با کمترین وزن را از heap خارج کنید.
    - یک گره جدید بسازید که وزنش برابر جمع وزن این دو گره است و این دو گره فرزندش هستند.

- گره جدید را به heap اضافه کنید.

3. گره باقی‌مانده ریشه درخت هافمن است.

بخش ج: سودوکد(PseudoCode)

Input: freq[] = array of pairs (char, frequency)

// freq شامل فرکانس هر کاراکتر ورودی است

مرحله 1: قرار دادن همه کاراکترها به صورت گره‌های جداگانه در min-heap بر اساس فرکانس

Create a min-heap H containing all characters as separate nodes

توضیح heap: به ما اجازه می‌دهد در زمان log n کمترین فرکانس را سریع استخراج کنیم

مرحله 2: ساخت درخت هافمن با ترکیب گره‌ها

while size(H) > 1:

استخراج دو گره با کمترین فرکانس از heap

```
left = H.extract_min()
right = H.extract_min()
```

ساخت یک گره جدید که فرکانسش جمع دو گره استخراج شده است

```
merged = new Node()
merged.freq = left.freq + right.freq
merged.left = left شاخه چپ گره جدید
merged.right = rightشاخه راست گره جدید
```

بازگرداندن گره جدید به heap برای مراحل بعدی

```
H.insert(merged)
```

مرحله 3: گره باقیمانده درheap ، ریشه درخت هافمن است

```
return H.extract_min()
```

توضیح: پس از انجام ترکیب‌ها، یک درخت واحد با وزن کل فرکانس داریم که بهینه‌ترین کدگذاری را می‌سازد.

نکات کلیدی:

- heap اولویت می‌دهد به گره‌هایی با فرکانس کمتر تا ابتدا آنها را ترکیب کند.
- هر بار دو گره کوچک را می‌گیریم و با هم ادغام می‌کنیم تا نهایتاً درخت کامل تشکیل شود.
- نتیجه‌ی نهایی، درختی است که اجازه می‌دهد کدهای بدون پیشوند بسازیم که میانگین طول کد کمینه باشد.

بخش د: تحلیل زمانی

- ساخت اولیه‌ی heap شامل nگره، پیچیدگی زمانی $O(n)$دارد.
- در هر مرحله دو گره با کم‌ترین فرکانس استخراج می‌شوند و یک گره جدید جایگزین می‌شود که این عملیات به تعداد $n-1$بار تکرار می‌شود.
- هر عملیات استخراج و درج در heap هزینه‌ای برابر با $O(\log\ n)$دارد.
- بنابراین کل زمان اجرای الگوریتم تقریباً برابر است با:

$$O(n) + (n-1) \times O(\log\ n) = O(n\log\ n)$$

جواب سوال چهارم

1. چرا الگوریتم استاندارد sort-by-finish-time درست است؟ (با ایده‌ی Exchange Argument)

در مسئله‌ی Interval Scheduling هدف این است که بیشترین تعداد فعالیتی که با هم تداخل ندارند را انتخاب کنیم.

الگوریتم استاندارد می‌گوید:

- فعالیت‌ها را بر اساس کمترین زمان پایان مرتب کن.
- از ابتدا به انتها هر فعالیتی که با فعالیت‌های انتخاب‌شده تداخل ندارد را انتخاب کن.

ایده‌ی اصلی چرا این کار درست است را می‌توان با یک استدلال ساده توضیح داد که به آن exchange argument می‌گویند.

توضیح با زبان ساده:

فرض کنید یک جواب کاملاً بهینه وجود دارد که بیشترین تعداد فعالیت ممکن را دارد. حالا نگاه کنیم اولین فعالیتی که الگوریتم استاندارد انتخاب می‌کند کدام است؟

این فعالیت زودترین زمان پایان را دارد.

اگر در جواب بهینه، اولین فعالیت چیز دیگری باشد، می‌توانیم آن را با فعالیتی که الگوریتم انتخاب کرده جایگزین کنیم. چرا این جایگزینی همیشه شدنی است؟

- چون فعالیت انتخاب‌شده توسط الگوریتم زودتر تمام می‌شود.
- بنابراین فضای زمانی بیشتری برای فعالیت‌های بعدی باقی می‌گذارد.
- و هیچ تداخلی ایجاد نمی‌کند.

پس با جایگزینی آن، تعداد فعالیت‌های جواب بهینه تغییر نمی‌کند.

با تکرار همین منطق برای فعالیت‌های بعدی، می‌توان نشان داد که همیشه می‌توان یک جواب بهینه ساخت که دقیقاً همان انتخاب‌های الگوریتم sort-by-finish-time را دارد.

بنابراین این الگوریتم همیشه یک جواب بهینه تولید می‌کند.

2. مقایسه با مسئله Coin Change

تفاوت ساختاری Interval Scheduling و Coin Change

وضعیت در دو مسئله اصلاً مشابه نیست:

- در Interval Scheduling هر انتخاب فقط بر روی «زمان باقی‌مانده» اثر می‌گذارد. اگر فعالیتی انتخاب کنیم که زود تمام می‌شود، همیشه فرصت برای انتخاب‌های بعدی حفظ می‌شود.
- اما در Coin Change انتخاب یک سکهٔ بزرگ‌تر یا کوچک‌تر می‌تواند کاری کند که در ادامه امکان رسیدن به جواب بهتر از بین برود.

به زبان ساده:

- در Interval Scheduling انتخاب محلی (محور زمان پایان) با هدف کلی هماهنگ است.
- اما در Coin Change انتخاب محلی (بزرگ‌ترین سکه) همیشه با هدف کلی هماهنگ نیست.

چرا greedy استاندارد در Interval Scheduling همیشه درست است ولی در Coin Change ممکن است شکست بخورد؟

در Interval Scheduling:

الگوریتم sort-by-finish-time هر بار فعالیتی را انتخاب می‌کند که کمترین زمان پایان را دارد.

چرا درست است؟

- چون این انتخاب هیچ گزینه‌ی بهتری را در آینده خراب نمی‌کند.
- فعالیت زودپایان «بیشترین فضا» را برای انتخاب‌های بعدی باقی می‌گذارد.
- exchange argument نشان می‌دهد که همیشه می‌توان جواب بهینه را با انتخاب‌های greedy سازگار کرد.

بنابراین پاسخ همیشه بهینه است.

در Coin Change:

الگوریتم greedy معمولاً بزرگ‌ترین سکه‌ی ممکن را انتخاب می‌کند. اما در بسیاری از سیستم‌های سکه این انتخاب محلی اشتباه است و باعث می‌شود بهترین جواب از بین برود.

مثال معروف (سکه‌های 1، 3، 4 برای مقدار 6):

- greedy:4 + 1 + 1 = سه سکه
- بهینه:3 + 3 = دو سکه

در این مثال انتخاب محلی greedy (برداشتن ۴) فرصت ساختن جواب بهتر را می‌گیرد.

پس برخلاف Interval Scheduling، در Coin Change رفتار greedy همیشه با ساختار مسئله سازگار نیست.