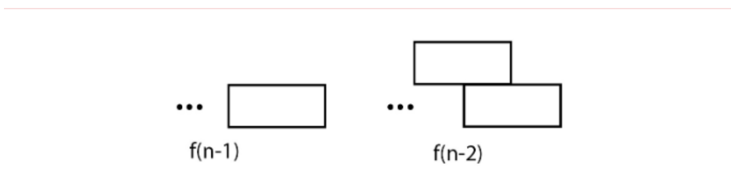
به نام خدا

دانشگاه تهران، دانشکده مهندسی برق و کامپیوتر
تحلیل و طراحی الگوریتم‌ها

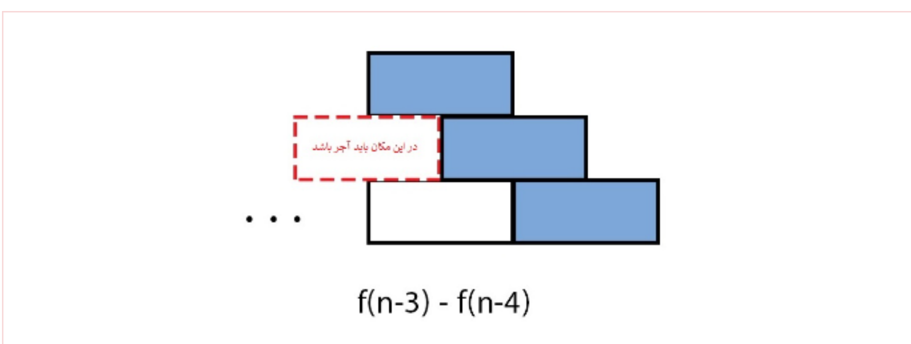
پاسخ تمرین کتبی چهارم
موعد تحویل: شنبه ۴ اسفند ۹۷، ساعت ۰۰:۹
طراح: ارشیا ابوالقاسمی arshiabolghasemi@gmail.com

---

۱. سعی می‌کنیم به صورت بازگشتی سوال را حل کنیم. بنابراین فرض می‌کنیم f_n را تعداد روش‌های چیدن n آجر باشد. حال راست‌ترین آجر لایه اول را در نظر بگیرید، اگر روی آن آجر دیگری قرار داشت آن‌را در نظر بگیرید و اگر باز هم روی آن آجر دیگری قرار داشت رویی آن را در نظر بگیرید. آن‌قدر ادامه دهید تا به آجری برسید که سمت راست و روی آن آجر دیگر نیست و با برداشتن سمت راست‌ترین آجر لایه‌ی اول آن آجر فرو بریزد. روی این آجر خاص حالت‌بندی می‌کنیم. توجه کنید در حل این‌گونه مسائل حالت‌بندی مناسب بسیار حائز اهمیت است. سه حالت داریم، یا این آجر روی زمین است، یا زیرش یک لایه‌است یا دولایه آجر. مطابق شکل زیر می‌توانید ببینید اگر زیرش زمین یا یک آجر دیگر باشد می‌توانیم آجرهای زیرین و سمت راستش را مطابق شکل برداریم و شکل باقی‌مانده یک آجرچینی معتبر با سایر آجر هاست.



و اما حالتی را در نظر بگیرید که دو آجر زیرش باشد. در این‌صورت اگر دو آجر پایین و راستش را مطابق شکل برداریم شکل باقیمانده یک آجرچینی با $n-3$ آجر خواهد بود اما در اینجا هر آجرچینی با $n-3$ آجر معتبر نخواهد بود به جز حالت‌هایی که مطابق شکل مکان دارای نقطه چین خالی از آجر باشد. تعداد این چینش‌های نامعتبر، f_{n-4} است، زیرا مطابق شکل کافیست زیر نقطه‌چین خالی از آجر باشد و بقیه شکل یک آجرچینی با $n-4$ آجر است. بنابراین با کم کردن این تعداد از f_{n-3} به آجرچینی‌های معتبر می‌رسیم.



پس داریم:

$$f_n = f_{n-1} + f_{n-2} + f_{n-3} - f_{n-4}$$

حال کافیست یک آرایه به اسم f به طول n در نظر بگیریم و برای حالت پایه نیز بایه داریم:

$$f_0 = f_1 = f_2 = 1$$

$$f_3 = 2$$

حال برای پر کردن آرایه f نیز به شکل زیر عمل می‌کنیم:

```
for i from 4 to n:
    f[n] = f[n-1] + f[n-2] + f[n-3] - f[n-4]
```

پیچیدگی زمانی الگوریتم $o(n)$ و پیچیدگی حافظه آن $o(n)$ می‌باشد

۲. (آ) برای هر این مساله از آرایه dp به طول s استفاده می‌کنیم. به‌طوری که $dp[i]$ برابر کمترین سکه مورد نیاز برای خرد کردن یک i گیلدری می‌باشد. اگر نمی‌توانستیم با پولدهای موجود یک i گیلدری را خرد کنیم $dp[i]$ را برابر صفر قرار می‌دهیم. در ابتدا برای همه i ها مقدار $dp[i]$ را بی‌نهایت می‌گذاریم. برای حالت پایه نیز $dp[0]$ را برابر صفر می‌گذاریم سپس برای پر کردن آن به این صورت عمل می‌کنیم: به ازای همه i اگر داشتیم $v_j \leq i$ $dp[i - v_j] + 1 < dp[i]$ آنگاه $dp[i]$ را برابر $dp[i - v_j]$ قرار می‌دهیم. شبه کد الگوریتم به صورت زیر می‌باشد.

```
for all of i:
    dp[i] = infinity
dp[0] = 0

for i = 1 to s:
    for j = 1 to n:
        if v[j] <= i and dp[i - v[j]] + 1 < dp[i] :
            dp[i] = dp[i - v[j]]
```

در انتها $dp[s]$ پاسخ مساله می‌باشد. پیچیدگی زمانی الگوریتم $o(sn)$ و پیچیدگی حافظه آن $o(n)$ می‌باشد

(ب) خیر. اعداد در کامپیوتر به صورت بیت‌های صفر و یکی ذخیره می‌شوند پس برای کامپیوتر **اندازه** اعداد مهم می‌باشد نه مقدار عددی آن‌ها. بنابراین در این مثال اندازه عدد s که همان $log_2 s = w$ می‌باشد، اهمیت دارد. در واقع پیچیدگی زمانی این الگوریتم $o(2^w n)$ می‌باشد که به وضوح چند جمله‌ای نیست. به این‌گونه پیچیدگی زمانی‌ها، اصطلاحا شبه چند جمله‌ای می‌گویند.

۳. (آ) ابتدا سعی می‌کنیم با یک راه‌حل ساده سوال را حل کنیم. راه‌حل بدیهی‌ای که احتمال ابتدا برای حل این سوال به ذهن‌مان می‌رسد این است که جمع مجموع اعداد همه زیربازه ها را حساب کرده، سپس از بین آن‌ها بزرگترین عدد ممکن را انتخاب کنیم. در کل n^2 زیر بازه دادیم.(n انتخاب برای امتخاب عدد اول زیربازه و n انتخاب برای عدد دوم زیربازه) حال برای هر کدام از این زیربازه ها نیز $o(n)$ زمان می‌برد تا جمع اعداد آن زیربازه را حساب کنیم. بنابراین این الگوریتم برای پیدا کردن جواب مساله ما $o(n^3)$ طول خواهد کشید که با پیچیدگی زمانی‌ای که مساله خواسته است فاصله دارد. حال سعی می‌کنیم که الگوریتم را بهینه‌تر کنیم. یکی از مشکلاتی که این الگوریتم دارد این است که به طور مثال فرض کنید می‌خواهیم جمع اعداد در زیربازه $[l, r)$ را جساب کنیم، اگر اعداد را یکی یکی از چپ به راست اضافه کنیم آنگاه جمع همه زیربازه‌های $[l, l + 1)$, $[l, l + 2)$, ..., $[l.r - 1)$ را نیز حساب کرده‌ایم و نیازی به دوباره حساب کردم مجموع اعداد این زیربازه‌ها نمی‌باشد. پس نیاز به روشی داریم که مجموع اعداد هر زیربازه را ذخیره کند. برای این‌کار از یک آرایه دوبعدی $n * n$ به نام dp کمک می‌گیریم به طوریکه $dp[l][r]$ برابر جمع همه اعداد در زیربازه $[l, r)$ می‌باشد. مقداردهی اولیه و نحوه پر کردن این آرایه در شبه کد زیر آمده است

```
for i from 1 to n:
    dp[i][i + 1] = a[i]

for sizeOfSubSegment from 2 to n:
    for l from 1 to n - sizeOfSubSegment:
        r = l + sizeOfSubSegment
        dp[l][r] = a[l] + dp[l + 1][r]
```

در کد بالا ابتدا همه زیربازه‌ها به طول یک، سپس همه زیربازه‌ها به طول ۲ و ... حساب می‌شوند پس هر گاه بخواهیم $dp[l][r]$ را حساب کنیم، مقادیر مورد نیاز از قبل حساب شده‌اند. بنابراین توانستیم این مساله را در $o(n^۲)$ حل کنیم. پیچیدگی حافظه مساله نیز $o(n^۲)$ می‌باشد

(ب) حال یک تعریف جدید برای آرایه dp ارائه می‌دهیم. این بار آرایه dp را یک آرایه یک بعدی به طول n در نظر می‌گیریم به طوری‌که $dp[i]$ برابر بزرگترین جمع زیربازه‌ای است که به عنصر i ام ختم می‌شود. حال پاسخ مساله ما در این حالت برابر بزرگترین مقدار آرایه dp می‌باشد. زیرا بزرگترین زیربازه در نهایت به یکی از عنصرهای n ,.., ۲ ,۱ اُم ختم خواهد شد. $dp[i]$ نیز یا فقط برابر عدد i ام می‌باشد یا اینکه شامل عنصر $i-۱$ اُم نیز هست که در این صورت باید آنقدر عقب رویم تا جمع زیربازه بیشینه شود. در واقع به دنبال j ای هستیم که $\sum_{k=j}^{i-۱} a_k$ بیشینه شود. که این همان $dp[i-۱]$ می‌باشد. برای حالت پایه نیز داریم: $dp[۱] = a_۱$. شبه کد پرکردن آرایه dp در زیر آمده است:

```
dp[1] = a[1]
for i from 1 to n:
    dp[i] = max(a[i] + a[i] + dp[i-1])
```

در انتها نیز بزرگترین مقدار آرایه dp پاسخ مساله خواهد بود. پیچیدگی زمانی و حافظه این الگوریتم هر دو $o(n)$ می‌باشد.

۴. راه‌حل بدیهی این مساله این است که برای هر زیر جدول جمع اعداد آن را حساب کرده و بزرگترین این اعداد را خزوجی دهیم. این کار $o(n^۶)$ طول خواهد کشید.($o(n^۴)$ انتخاب زیر مستطیل ها و $o(n^۲)$ برای حساب کردن جمع اعداد زیر مستطیل) حال کمی الگوریتم را بهبود می‌دهیم. فرض کنید آرایه a آرایه دوبعدی و شامل اعداد نوشته شده در مستطیل هستند. هر زیر مستطیل با چهار عدد مشخص می‌شود $r_۱, r_۲$ که ردیف های بالایی و پایینی زیر مستطیل را مشخص می‌کنند و $c_۱, c_۲$ که ستون های چپ و راست مستطیل را مشخص می‌کنند. حال فرض‌کنید $r_۱, r_۲$ شماره ستون‌های بالایی و پایینی این مستطیل باشند. حال اگر ستون j در این زیر مستطیل آمده باشد، آنگاه تمام اعداد $a[r_۱][j], a[r_۱+۱][j], ..., a[r_۲][j]$ در این زیر مستطی حضور دارند. حال آرایه b را به این صورت در نظر می‌گیریم که

$$b[j] = \sum_{i=r_۱}^{r_۲} a[i][j]$$

در اینصورت جمع بزرگترین زیر مستطیلی که ستون‌های بالایی و پایینی آن $r_۱, r_۲$ باشند برابر بزرگترین زیربازه در آرایه b است. با توجه به مساله قبلی بزرگترین زیر بازه را در $o(n)$ می‌توانستیم بدست آوردیم. پس اگر برای هر دو تا ردیف $r_۱, r_۲$ بزرگترین زیربازه را بدست آوریم، آنگاه بزرگترین این مقدارها برابر بزرگترین زیر مستطیل مورد نظر می‌باشد. پیچیدگی زمانی الگوریتم $o(n^۳)$ و پیچیدگی حافظه آن $o(n)$ می‌باشد.

۵. هر کدام از ضفرها را برابر منفی یک و هر کدام از یک‌ها را برابر مثبت یک در نظر می‌گیریم. حال در هر خانه جدول عدد معادلی که بر اساس تعداد صفرها و یک‌های آن بدست می‌آید را قرار می‌دهیم. حال باید مسیری در این جدول بیابیم به‌طوری که جمع عدد خانه‌های آن مسیر بیشترین مقدار شود. برای آن آرایه دو بعدی dp را به این صورت در نظر می‌گیریم که $dp[i][j]$ برابر مسیر با بیشترین جمع اعداد اط خانه چپ بالا($(۰, ۰)$) تا خانه (i, j) می‌باشد. پاسخ مساله برابر $dp[n][m]$ می‌باشد. حال سعی می‌کنیم آرایه dp را پر کنیم. به خانه (i, j) می‌توان از خانه‌های $(i, j-۱)$ یا $(i-۱, j)$ رسید. بنابراین باید مسیر های این دو میسیر بررسی شود. رابطه بازگشتی پر کردن آرایه dp به صورت زیر می‌باشد:

$$dp[i][j] = max(dp[i-۱][j], dp[i][j-۱]) + a[i][j]$$

حالت پایه نیز به صورت زیر است:

$$dp[۰][۰] = a[۰][۰]$$

پیچیدگی زمانی و پیچیدگی حافظه این الگوریتم هر دو برابر $o(n^۲)$ می‌باشد

۶. این مساله همان مساله کوله‌پشتی می‌باشد که کمی شرط‌های آن را تغییر داده‌ایم. کل پولی که افش دارد همان گنجایش کوله‌پشتی است. و هر درس هم یک شی است که میخواهیم در این کوله‌پشتی قرار دهیم. وزن هر کدام از این شی‌ها نیز معادل پولی است که باید برای برداشتن این درس هزینه کنیم. برای حل این سوال آرایه دو بعدی dp را اینگونه تعریف می‌کنیم: $dp[i][j]$ برابر بیشترین تعداد درسی است که می‌توانیم با j پوند تا سال i ام تحصیلی انتخاب کنیم. برای سال تحصیلی i اُم سه حالت داریم:

- در این سال هیچ کدام از این درس‌ها را انتخاب نمی‌کنیم دراین صورت داریم

$$dp[i][j] = dp[i-۱][j]$$

- در این سال فقط یک درس را انتخاب می‌کنیم و از آنجا که می‌خواهیم بیشترین تعداد درس را با پولمان برداریم آن یک درس بایک ارزان‌ترین درس باشد. پس داریم:

$$dp[i][j] = dp[i-۱][j - min(c[i])]$$

به طوری که $c[i]$ آرایه‌ای از قیمت درس‌ها در سال i اُم است.

- در این سال همه درس‌ها را انتخاب می‌کنیم در این صورت خواهیم داشت:

$$dp[i][j] = dp[i-۱][j - \sum_{j\in c[i]} c[i][j]]$$

پس برای $dp[i][j]$ باید ماکسیمم یکی از سه حالت بالا را قرار دهیم.
برای حالت پایه نیز داریم: $\forall j : dp[۰][j] = ۰$ شبکه کد الگوریتم نیز به صورت زیر است:

```
for i from 0 to n:
    for j from 0 to n:
        if i == 0:
            dp[i][j] = 0
        else:
            dp[i][j] = max(
                dp[i - 1][j],
                dp[i - 1][j -min(c[i])],
                dp[i - 1][j - sum(c[i])]
                )
```

۷. (آ) رشته اول را با s و رشته دوم را با p نشان می‌دهیم. حال آرایه دوبعدی dp را طوری درنظر می‌گیریم که $dp[i][j]$ برابر طول بزرگترین زیر رشته مشترک $s_{۱...i}$ و $p_{۱...j}$ می‌باشد. حال برای پر کردن آرایه dp روی آمدن یا نیامدن s_i و d_j حالت بندی می‌کنیم.
اگر $s_i \neq p_j$ آنگاه $dp[i][j] = max(dp[i-۱][j], dp[i][j-۱])$ می‌باشد زیرا در طولانی‌ترین زیررشته s_i و p_j مشترکا در بزرگترین زیر دنباله ظاهر نشده‌اند. اگر هم داشته باشم $s_i = p_j$ آنگاه خواهیم داشت: $dp[i][j] = max(dp[i-۱][j], dp[i][j-۱], dp[i-۱][j-۱]$ زیرا که ممکن است یک از s_i یا p_j ظاهر نشده باشند یا جفتشان ظاهر شده باشند. برای حالت پایه نیز داریم

$$\forall i : dp[i][۰] = d[۰][i] = ۰$$

کد این الگوریتم به صورت زیر است:

```
for all i:
    dp[i][0] = dp[0][i] = 0

for i in len(s):
    for j in len(p):
        dp[i][j] = max(
            dp[i][j - 1],
            dp[i - 1][j],
            dp[i - 1][j - 1]
            )
```

(ب) حال برای اینکه بزرگترین زیر رشته مشترک را نیز بتوانیم طولید کنیم از یک آرایه کمکی دیگر برای نگه داشتن روند بروز رسانی dp استفاده می‌کنیم.

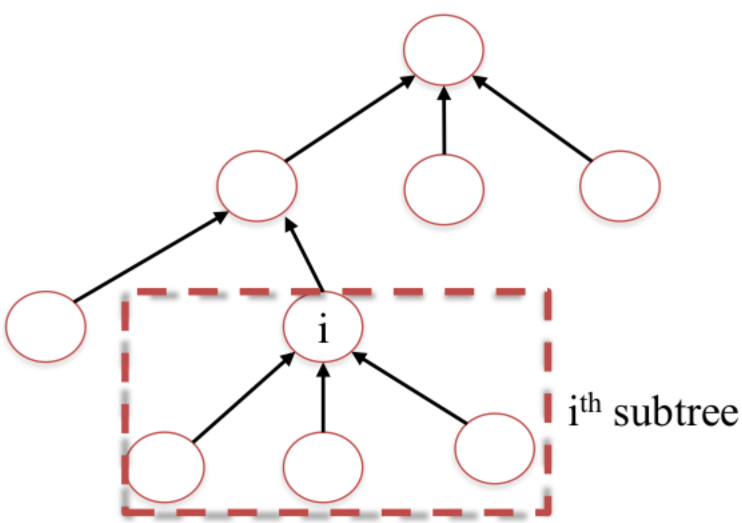
```
for all i:
    dp[i][0] = dp[0][i] = 0

for i in len(s):
    for j in len(p):
        if s[i] == p[j]:
            lsc.append(s[i]) // or p[j], they are same
            dp[i][j] = max(
                dp[i - 1][j - 1],
                dp[i][j - 1],
                dp[i - 1][j]
            )
        else:
            dp[i][j] = max(
                dp[i - 1][j],
                dp[i][j - 1]
            )

def printLSC(i):
    if i == 0:
        print(lsc[i])
        return

    printLSC(i - 1)
    print(lsc[i])
```

۸. آرایه یک بعدی A را طوری در نظر می‌گیریم که $A[i]$ بهترین روش مهمانی گرفتن است در صورتی که فرد i اُم دیگر رئیسی نداشته باشد. یعنی اگر رابطه رئیس و کارمندی را به صورت یک درخت تشبیه کنیم $A[i]$ بهترین روش مهمانی در زیر درختی است که فرد i ریشه آن است



حال برای $A[i]$ داریم:

$$A[i] = c_i + \sum_{j \in grandchild(i)} A[j] \quad if\ \ i\ \ is\ \ invited$$

$$A[i] = \sum_{j \in child(i)} \quad if\ \ i\ \ isn't\ \ invited$$

حال برای خوش تعریف‌تر کردن رابطه بالا از یک آرایه یک بعدی دیگر به اسم B کمک می‌گیریم به طوری $B[i]$ بهترین مهمانی است که می‌تواند برگزار شود در صورتی که i دیگر رئیسی نداشته باشد و همچنین i به مهمانی دعوت نشده باشد. در این صورت داریم:

$$B[i] = \sum_{j \in child(i)}$$

$$A[i] = max(c_i + \sum_{j \in child(j \in child(i)} B[j], B[i])$$

پس برای محاسبه $A[i], B[i]$ کافیست که برای تمام نوادگان i در درخت رئیسیُ کارمندی وزارت جادو مقدار A و B آن‌ها حساب شده باشد. شبه کد این الگوریتم به صورت زیر است:

```
lordVoldemortPartyPlanning(c, i):
    A[i] = c[i]
    B[i] = 0

    for j in ichildren: s'
        lordVoldemortPartyPlanning(c, j)
        A[i] += B[j]
        B[i] += B[j]

    A[i] = max(A[i], B[i])
```

پیچیدگی زمانی و حافظه این الگوریتم هر دو از $o(n)$ می‌باشد چون هر $node$ در درخت رئیسی کارمندی وزارت جادو را دقیقا یکبار ملاقات می‌کنیم.

۹. (آ) از ماتریس زیر کمک می‌گیریم:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

حال اگر این ماتریس را از چپ در ماتریس $\begin{bmatrix} f_۱ \\ f_۰ \end{bmatrix}$ ضرب کنیم.(f_i ها جملات فیبوناچی می‌باشد.) ماتریس $\begin{bmatrix} f_۲ \\ f_۱ \end{bmatrix}$ تولید خواهد شد. حال دوباره اگر ماتریس $\begin{bmatrix} ۱ & ۱ \\ ۱ & ۰ \end{bmatrix}$ از چپ در ماتریس حاصل ضرب کنیم ماتریس $\begin{bmatrix} f_۳ \\ f_۲ \end{bmatrix}$ بدست می‌آید. با طی کردن همین روند به رابطه زیر می‌رسیم:

$$\begin{bmatrix} ۱ & ۱ \\ ۱ & ۰ \end{bmatrix}^n . \begin{bmatrix} f_۱ \\ f_۰ \end{bmatrix} = \begin{bmatrix} f_{n+۱} \\ f_n \end{bmatrix}$$

حال با توجه به اینکه ضرب یک ماتریس ۲ در ۲ در خودش در $o(۱)$ امکان پذیر می‌باشد و بدست آوردن توان n اُم را هم به کمک برنامه‌سازی پویا می‌توانیم در $o(log(n))$ بدست می‌آوریم پس مساله فیبوناچی را در $o(log(n))$ می‌توانیم حل کنیم.

(ب) ایده این قسمت دقیقا مشابه قسمت قبل است فقط باید از یک ماتریس k در k به شکل زیر استفاده کرد

$$\begin{bmatrix} ۱ & ۱ & ۱ & \dots & ۱ \\ ۱ & ۰ & ۰ & \dots & ۰ \\ ۰ & ۱ & ۰ & \dots & ۰ \\ ۰ & ۰ & ۱ & \dots & ۰ \\ & & \dots & & \\ ۰ & ۰ & ۰ & \dots & ۱ \end{bmatrix}$$

رابطه نهایی این قسمت به شکل زیر می‌باشد:

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ & & \dots & & \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}^n . \begin{bmatrix} f_{k-1} \\ f_{k-2} \\ . \\ . \\ . \\ f_1 \end{bmatrix} = \begin{bmatrix} f_{n+1} \\ f_n \\ . \\ . \\ . \\ f_{n-k+1} \end{bmatrix}$$

بدست آوردن توان n اُم را دوباره در $o(log(n))$ انجام می‌دهیم ولی ضرب یک ماتریس k در k در خودش در $o(k^3)$ صورت می‌پذیرد، پس اردر زمانی الگوریتم $o(k^3 log(n))$ می‌باشد