طراحی الگوریتم

جزوه دوم - برنامه‌ریزی پویا

برنامه‌ریزی پویا یا داینامیک، روشی کارآمد برای حل مسائل بهینه‌سازی با استفاده از دو ویژگی زیرمسئله‌های هم‌پوشان و زیرساخت‌های بهینه است. در برنامه ریزی پویا ما معمولا دنبال ترسیم جدولی و پر کردن هر یک از خانه‌های خالی جدول بر اساس مقدار دیگر خانه‌های جدول هستیم. ما معمولا زمانی از برنامه ریزی پویا استفاده می‌کنیم که مسئله اصلی ما را بتوان با حل زیر مسئله‌هایی حل کرد. باید بتوان نتیجه حل این زیر مسئله‌ها را ذخیره کرد و بعداً بازیابی کرد و همچنین ممکن است چند بار به جواب این زیر مسئله‌ها نیاز داشته باشیم که ذخیره آن کمک می‌کند تا فقط یک بار هر یک را محاسبه کنیم. در برنامه ریزی پویا نگاه ما پایین به بالا است و در واقع از حل مسائل کوچکتر به پاسخ مسائل بزرگتر می‌رسیم.

۱. فیبوناچی

همه ما مسئله فیبوناچی را می‌دانیم که دنباله ای از اعداد است که غیر از دو عدد اول، اعداد بعدی از جمع دو عدد قبلی خود به‌دست می‌آیند.

$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0, F(1) = 1$$

حال به دنبال بدست آوردن جمله n ام این دنباله هستیم.

پاسخ :

تعریف DP:

جدول $n \times 1$ را در نظر می‌گیریم که $A[i]$ مقدار جمله i ام دنباله را دارد.

پایه DP:

همان پایه‌ی دنباله فیبوناچی هستند یعنی $A[0] = 0, A[1] = 1$.

رابطه DP:

این قسمت هم مثل رابطه فیبوناچی است و در واقع $A[i] = A[i-1] + A[i-2]$.

پاسخ DP:

از پایه یعنی $A[0]$ و $A[1]$ شروع می‌کنیم و طبق رابطه ای که بدست آوردیم مقدار $A[i]$ها را حساب می‌کنیم تا به n برسیم و در نهایت جواب ما در $A[n]$ قرار دارد و از آنجا که فقط یک بار از 1 تا n را پیمایش کردیم پیچیدگی زمانی آن از مرتبه $O(n)$ خواهد بود.

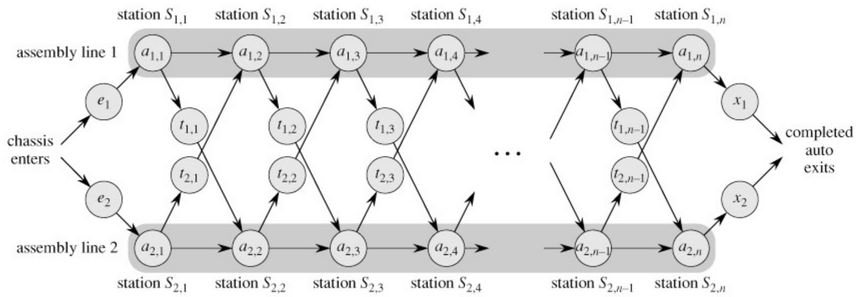
```
Fib(n){
    A[0] = 0
    A[1] = 1
    for i = 2 to n
        A[i] = A[i-1] + A[i-2]
    return A[n]
}
```

به ساختار کلی کد که رویکرد پایین به بالا دارد، توجه کنید چون اساس حل تمام مسئله‌های DP همین است.

۲. زمان‌بندی خط تولید

کارخانه ای 2 خط تولید ماشین دارد به طوریکه قطعات اولیه از سمت چپ می‌توانند وارد هر یک از خطوط تولید شوند و در هر خط تولید n ایستگاه قرار دارد که در هر ایستگاه قطعه‌ای به ماشین ورودی اضافه می‌شود و در نهایت ماشین اسمبل شده از سمت راست و ایستگاه nام که می‌تواند در خط تولید 1 یا 2 باشد، خارج شود. شماره اندیس ایستگاه‌ها را با j و خط تولیدها را با i نشان می‌دهیم به طوریکه i می‌تواند 1 یا 2 باشد و j می‌تواند از 1 تا n باشد حال هر ایستگاه را با $S[i][j]$ نشان می‌دهیم که نشان دهنده ایستگاه jام در خط تولید iام است. در هر ایستگاه که باشیم به اندازه $a[i][j]$ زمان صرف می‌شود تا قطعه مورد نظر به ماشین اضافه شود. همچنین ممکن است به هر دلیل از یک خط تولید به خط تولید دیگری سوییچ کنیم که هزینه زمانی آن را با $t[i][j]$ نمایش می‌دهیم که یعنی از ایستگاه jام در خط تولید i به ایستگاه بعدی یعنی $j+1$ام در خط تولید دیگر سوییچ کنیم. همچنین هزینه ورودی اولیه به هر یک از خط تولیدها را با e_1 و e_2 نشان می‌دهیم و هزینه خروجی از خط تولیدها را با x_1 و x_2.

حال می‌خواهیم کمترین زمان ممکن برای تولید یک ماشین و مسیر پیموده شده توسط آن را پیدا کنیم.



پاسخ :

ممکن است ابتدا این ایده به ذهنتان برسد که تمام حالت‌ها را امتحان کنیم و سپس کمترین آن‌ها را پیدا کنیم که از آنجا که در هر ایستگاهی که باشیم، دو حق انتخاب داریم (یا در خط تولید فعلی بمانیم یا سوییچ کنیم)، مرتبه زمانی آن از مرتبه $O(2^n)$ خواهد بود که اصلا مطلوب ما نیست!

تعریف DP:

دو جدول $2 \times n$ به نام‌های f و l در نظر می‌گیریم.

ما $f[i][j]$ را کمترین زمان ممکن صرف شده تا رسیدن به ایستگاه j ام در خط تولید i ام در نظر می‌گیریم یعنی کمترین زمان صرف شده تا رسیدن به $S[i][j]$. f^* را سریع‌ترین زمان ممکن برای خروج از خط تولید ماشین به صورت کلی در نظر می‌گیریم.

همچنین $l[i][j]$ مقدار 1 یا 2 را میگیرد که نشان می‌دهد برای رسیدن به ایستگاه $S[i][j]$ از کدام خط تولید آمدیم. (مرحله قبل یعنی $j-1$، در کدام خط تولید بودیم که سپس به ایستگاه $S[i][j]$ رسیدیم.)
l^* یعنی از کدام لاین در مرحله آخر یعنی nام خارج شدیم. **پایه DP:**

طبق تعریف‌هایمان داریم:

$$f[1][1] = e_1 + a[1][1]$$

$$f[2][1] = e_2 + a[2][1]$$

و همچنین حالت نهایی به صورت زیر می‌شود:

$$f^* = \min(f[1][n] + x_1, f[2][n] + x_2)$$

رابطه DP:

$$f[1][j] = \min(f[1][j-1] + a[1][j], f[2][j] + t[2][j-1] + a[1][j])$$

$$f[2][j] = \min(f[2][j-1] + a[2][j], f[1][j] + t[1][j-1] + a[2][j])$$

پاسخ DP:

فرض کنیم سریع‌ترین مسیر رسیدن به ایستگاه $S[1][j]$ در مرحله قبل از ایستگاه $S[1][j-1]$ عبور می‌کند در آن صورت مسیری هم که از ابتدا تا ایستگاه $S[1][j-1]$ طی شده است هم باید کمترین هزینه زمانی را داشته باشد.

✓ برای اثبات آن میتوان از برهان خلف استفاده کرد. اگر در نظر بگیریم که مسیر از ابتدا تا رسیدن به $S[1][j-1]$کمترین هزینه زمانی را ندارد در آن صورت مسیر سریع‌ترین از این ایستگاه عبور نمی‌کرد.

و همینطور این استدلال را برای زمانی که مسیر بهینه تا $S[1][j]$ از $S[2][j-1]$ هم بگذرد می‌توان در نظر گرفت. در نتیجه برای رسیدن به پاسخ مسئلی در ایستگاه $S[i][j]$ باید جواب بهینه برای زیر مسئله‌های آن یعنی $S[1][j-1]$ و $S[2][j-1]$ پیدا کرد که به آن زیر ساختار بهینه می‌گویند.

جدول $2 \times n$ در نظر می‌گیریم که سطر اول نشان دهنده‌ی مقادیر $f[1][j]$ هست و سطر دوم مقادیر $f[2][j]$. حال طبق روابط بدست آورده از سمت جدول شروع می‌کنیم و هم سطر اول و هم سطر دوم را با توجه به مقادیر ستون قبلی آن پر می‌کنیم

همچنین جدول $2 \times n$ دیگری هم در نظر می‌گیریم که بر اساس روابط گفته شده مقدار $l[i][j]$ها را آپدیت می‌کنیم.

```
FastestWay(a, t, e, x, n){
    f₁[1] = e₁ + a₁,₁
    f₂[1] = e₂ + a₂,₁
    for j = 2 to n {
        if f₁[j - 1] + a₁,ⱼ ≤ f₂[j - 1] + t₂,ⱼ₋₁ + a₁,ⱼ {
            f₁[j] = f₁[j - 1] + a₁,ⱼ
            l₁[j] = 1
        } else {
            f₁[j] = f₂[j - 1] + t₂,ⱼ₋₁ + a₁,ⱼ
            l₁[j] = 2
        }
        if f₂[j - 1] + a₂,ⱼ ≤ f₁[j - 1] + t₁,ⱼ₋₁ + a₂,ⱼ {
            f₂[j] = f₂[j - 1] + a₂,ⱼ
            l₂[j] = 2
        } else {
            f₂[j] = f₁[j - 1] + t₁,ⱼ₋₁ + a₂,ⱼ
            l₂[j] = 1
        }
    }
    if f₁[n] + x₁ ≤ f₂[n] + x₂ {
        f* = f₁[n] + x₁
        l* = 1
    } else {
        f* = f₂[n] + x₂
        l* = 2
    }
}
```

۳. بلند ترین زیردنباله مشترک

دو دنباله به نام‌های X و Y داریم که به ترتیب از حروف $\langle x_1, x_2, ..., x_m\rangle$ و $\langle y_1, y_2, ..., y_n\rangle$ تشکیل شده‌اند. بلندترین زیر دنبالهای که در هر دو دنباله مشترک است را پیدا کنید.

در زیر دنباله لازم نیست که حروف لزوما متوالی باشند ولی ترتیبشان اهمیت دارد.

پاسخ :

✓ اگر دنباله Z با حروف $< z_1, ..., z_k >$ بلندترین زیر دنباله مشترک دو دنباله X و Y باشد آنگاه داریم:

اگر داشته باشیم $x_m = y_n$ آنگاه این حرف در Z هم خواهد بود و آخرین حرف آن یعنی z_k خواهد بود. حال می‌توانیم این حرف مشترک را از هر سه دنباله حذف کنیم و آنگاه خواهیم داشت که $< z_1, ..., z_{k-1} >$ بلندترین زیر دنباله مشترک $< x_1, ..., x_{m-1} >$ و $< y_1, ..., y_{n-1} >$ خواهد بود.

اگر $x_m \neq y_n$ آنگاه اگر z_k با x_m برابر نباشد، می‌توان نتیجه گرفت که حرف آخر دنباله X تاثیری نخواهد داشت و می‌توان آن را حذف کرد و آنگاه $< z_1, ..., z_k >$ بلندترین زیر دنباله $< x_1, ..., x_{m-1}$ و $< y_1, ..., y_n$ خواهد بود و یا اگر z_k با y_n برابر نباشد، می‌توان نتیجه گرفت که حرف آخر دنباله Y تاثیری نخواهد داشت و می‌توان آن را حذف کرد و آنگاه $< z_1, ..., z_k >$ بلندترین زیر دنباله $< x_1, ..., x_m >$ و $< y_1, ..., y_{n-1} >$ خواهد بود.

☞ عبارت X_i را معادل حروف اول تا i ام دنباله X ، عبارت Y_i را معادل حروف اول تا i ام دنباله Y و عبارت Z_i را معادل حروف اول تا i ام دنباله Z در نظر می‌گیریم.

تعریف DP:

جدول $m \times n$ای به نام c را در نظر می‌گیریم.

$c[i][j]$ را معادل طول بلندترین زیر دنباله مشترک X_i و Y_j در نظر می‌گیریم. و برای ما مطلوب است که $c[m][n]$ را بدست آوریم.

پایه DP:

اگر i یا j صفر باشند آنگاه $c[i][j] = 0$

$$c[i][0] = c[0][j] = 0$$

که بدیهی است چون صفر بودن i یا j یعنی یکی از دنباله‌ها خالی است پس طول بلندترین زیر دنباله مشترک هم 0 می‌شود.

رابطه DP:

اگر $i, j > 0$ و $x_i != y_j$ آنگاه:

$$c[i][j] = \max(c[i-1][j], c[i][j-1])$$

اگر $i, j > 0$ و $x_i = y_j$ آنگاه:

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

پاسخ DP:

در واقع رابطه را طبق لمی که ابتدا معرفی کردیم بدست آوردیم و حال جدول m در n ای را برای پر کردن مقادیر $c[i][j]$ در نظر می‌گیریم و طبق پایه، سطر و ستون اول آن را مقدار 0 قرار می‌دهیم و سپس از آنجا که طبق رابطه، برای پر کردن خانه جدول به خانه بالایی و سمت چپ و به طور اریب به سمت چپ بالا نیاز داریم، باید جدول را سطر به سطر پر کنیم تا در نهایت به $c[m][n]$ برسیم و همچنین برای بدست آوردن دنباله مد نظر می‌توانیم از جدول $m \times n$ دیگری کمک بگیریم که مسیر حرکتمان را مشخص کند و هر جا که اریب حرکت کردیم (یعنی حرفی مشترک مشاهده کردیم و آن در بلندترین زیر دنباله مشترک خواهد بود) را به صورت بازگشتی چاپ می‌کنیم. همچنین بدلیل پر کردن جدول $m \times n$ مرتبه زمانی آن $O(m \times n)$ خواهد بود.

```
LCS-Length(X, Y){
    m = length[X]
    n = length[Y]
    for i = 1 to m
        c[i, 0] = 0
    for j = 0 to n
        c[0, j ] = 0
    for i = 1 to m {
        for j = 1 to n {
            if xi = yj {
                c[i, j] = c[i-1, j-1] + 1
                b[i, j] = "↖"
            } else if c[i-1, j ] ≥ c[i, j-1] {
                c[i, j] = c[i- 1, j ]
                b[i, j] = "↑"
            } else {
                c[i, j] = c[i, j-1]
                b[i, j] = "←"
            }
        }
    }
    return c and b
}
```

۴. کوله پشتی

یک دزد برای سرقت به خانه‌ای وارد شده است و با خود کوله پشتی دارد. می‌خواهد کوله پشتی خود را با اجناس خانه پر کند به طوری که در مجموع ارزش کالاهایی که دزدیده است بیشینه شود و وزن آن‌ها از مقدار مشخصی بیشتر نشود. بیشینه ارزشی که می‌تواند بدزد با توجه به وزن محدود شده، چقدر است؟

پاسخ :

تعریف DP:

در نظر می‌گیریم که بیشترین وزنی که می‌توان با کوله پشتی حمل کرد، W است. n جنس با وزن‌های $w_1, w_2, ..., w_n$ و با ارزش $v_1, v_2, ..., v_n$ داریم. حال به دنبال زیر مجموعه‌ای مانند S از مجموعه $\{1, 2, ..., n\}$ هستیم به طوریکه:

$$Maximize \sum_{i \in S} v_i$$

$$\sum_{i \in S} w_i \leq W$$

حال جدول $n \times W$ ای به نام A در نظر می‌گیریم به طوریکه $A[i][j]$ بیشترین ارزشی که می‌توانیم از جنس i اول با محدودیت وزن j بدست آوریم را نشان می‌دهد.

✓ زیر ساختار بهینه این مسئله بدین صورت است که جنس i ام را درنظر می‌گیریم. اگر تصمیم بگیریم که آن را اضافه کنیم، محدودیت جدید وزن برابر با $W - w_i$ می‌شود در غیر اینصورت محدودیت وزن همان باقی می‌ماند و سراغ بقیه اجناس می‌رویم.

پایه DP:

بدیهی است که اگر هیچ جنسی نداشته باشیم با هر محدودیت وزنی ارزشی که می‌توانیم بدست آوریم صفر است.

$$A[0][j] = 0$$

رابطه DP:

طبق زیر ساختار بهینه گفته شده می‌توان رابطه زیر را بدست آورد:

اگر نتوان جنسی را اضافه کرد یعنی $w_i > j$:

$$A[i][j] = A[i-1][j]$$

و اگر بتوان اضافه کرد یعنی $w_i \leq j$:

$$A[i][j] = \max\left\{A[i-1][j], A[i-1][j-w_i] + v_i\right\}$$

پاسخ DP:

همانطور که گفته شد ما جنس i ام را بررسی می‌کنیم و سپس با توجه به محدودیت وزن و ارزش اضافه نکردن شی i ام تصمیم می‌گیریم که آن را اضافه کنیم یا خیر و بدین ترتیب خانه‌های جدول A را به صورت سطری پر می‌کنیم چون برای هر خانه تنها به مقدار خانه قبلی آن نیاز داریم و پیچیدگی زمانی آن از مرتبه $O(W \times n)$ خواهد بود.

```
Knapsack(v, w, n, W) {
    for j=0 to W {
        A[0, j] = 0
    }
    for i = 1 to n {
        for j=0 to W {
            A[i, j] = A[i-1, j]
            if wᵢ ≤ j and A[i, j]<vᵢ + A[i-1, j-wᵢ] {
                A[i, j] = vᵢ + A[i-1, j-wᵢ]
            }
        }
    }
    return A[n, W]
}
```

Runtime complexity: $O(nW)$
Space complexity: $O(nW)$

توجه کنید که برای پر کردن هر خانه جدول به مقادیر موجود در سطر قبلی آن وابسته است. در نتیجه فقط میتوانیم از دو آرایه Wتایی استفاده کنیم که باعث میشود حجم حافظه مصرفی به جای $O(W \times n)$ به $O(W)$ کاهش یابد.

```
Knapsack(v, w, n, W) {
    for j=0 to W
    prevA[j] = 0

    for i = 1 to n {
        for j=0 to W {
            currA[j] = prevA[j]
            if wi ≤ j and currA[j] < vi + prevA[j-wi]
                currA[j] = vi + prevA[j-wi]
        }
        prevA = currA
    }
    return currA[W]
}
```

Replace A[i,j] with currA[j] and A[i-1,j] with prevA[j]

Implementation with $O(W)$ space

۵. ضرب ماتریس‌ها

فرض می‌کنیم ماتریس‌های $A_1, A_2, \ldots, A_n$ را داریم. می‌خواهیم کمترین تعداد ضرب ممکن را برای محاسبه ضرب آن‌ها بدست آوریم. در واقع می‌خواهیم پرانتزگذاری انجام دهیم، که ابتدا کدام ماتریس‌ها را در هم ضرب کنیم تا در نهایت کمترین تعداد ضرب را انجام داده باشیم. همچنین برای اینکه ضرب ممکن باشد، در نظر می‌گیریم که ابعاد ماتریس A_i به صورت $p_{i-1} \times p_i$ است تا بتوان حاصل ضرب ماتریس‌ها را به صورت $A_1 * A_2 * \ldots * A_n$ محاسبه کرد.

پاسخ :

تعریف DP:

✓ زیر ساختار بهینه این مسئله بدین صورت است که برای مثال در نظر می‌گیریم $A_i * \ldots * A_k * \ldots A_n$ را در نقطه k شکستیم تا کمترین تعداد ضرب را داشته باشند. خود هر تکه از آنها هم یعنی A_1 تا A_k و A_{k+1} تا A_n هم می‌توان به عنوان زیر مسئله‌های کوچکتر در نظر گرفت و در آن‌ها هم از نقطه‌ای می‌شکنیم که کمترین تعداد ضرب را به ما بدهند و به همین ترتیب ادامه پیدا می‌کند.

جدول $n \times n$ای به نام m در نظر می‌گیریم به طوریکه $m[i][j]$ یعنی کمترین تعداد ضرب ممکن برای محاسبه ضرب ماتریس A_i تا A_j. همچنین می‌توانیم جدول n در n دیگری هم در نظر بگیریم که مشخص کند از ماتریس A_i تا A_j را در کدام نقطه شکسته‌ایم.

پایه DP:

اگر $i = j$ باشد آنگاه یعنی تنها یک ماتریس داریم که بدیهی است که نیاز به محاسبه ضربی نداریم.

رابطه DP:

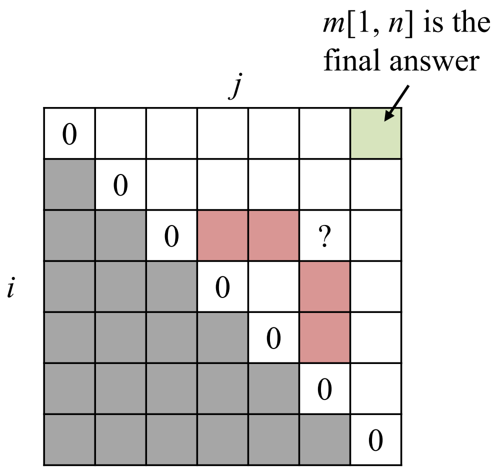
از آنجا که $m[i][j]$ را برای ماتریس iام تا jام تعریف کردیم پس داریم اگر $i < j$ آنگاه:

$$m[i][j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i][k] + m[k+1][j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

در واقع عبارت $m[i][k] + m[k+1][j] + p_{i-1}p_kp_j$ بیان می‌کند که اگر ضرب از i تا j را در نقطه k بشکنیم در بهترین حالت چه تعداد ضرب نیاز است و سپس برای بدست آوردن $m[i][j]$ مینیمم آن‌ها را محاسبه می‌کنیم.

پاسخ DP:

در واقع برای حل این مسئله آن را اینگونه در نظر می‌گیریم که برای بدست آوردن مقدار $m[i][j]$ باید آن را به دو قسمت از نقطه‌ای مانند k بشکنیم و بهترین حالتی که می‌توانیم این یک شکست را انجام دهیم پیدا کنیم و از آنجا که پاسخ برای زیر مسئله‌ها پیدا شده و ذخیره شده است می‌توانیم جواب $m[i][j]$ را محاسبه کنیم. همانطور که گفته شد با توجه به تعریف و رابطه‌مان قطر اصلی جدول m چون $i = j$ است مقدار صفر را می‌گیرد. حال برای پر کردن بقیه خانه‌های جدول برای مینیمم گرفتن باید از خانه‌های چپ و پایین خانه فعلی را بررسی کنیم پس به صورت اریب باید جدولمان را پر کنیم تا در نهایت خانه $m[1][n]$ پر شود که جواب مسئله‌مان است. در نظر بگیرید که برای اینگونه پیمایش کردن و مینیمم گرفتن روی آن‌ها نیاز داریم تا از مرتبه $O(n^3)$ هزینه کنیم.



m table

```
MatrixChainOrder(p) {
    for i =1 to n
        m[i, i] = 0

    for l = 2 to n {     // l is the chain length
        for i = 1 to n - l + 1 {
            j = i + l - 1
            m[i, j] = ∞
            for k = i to  j - 1 {
                q = m[i, k] + m[k + 1, j] + pi-1pkpj
                if q < m[i, j] {
                    m[i, j] = q
                    s[i, j] = k
                }
            }
        }
    }
    return m[1, n], s
}
```

Runtime complexity: $O(n^3)$
Space complexity: $O(n^2)$

بدین ترتیب کمترین تعداد ضرب مورد نیاز را به دست آوردیم. حال می‌توان به کمک یک تابع بازگشتی،

پرانتزگذاری موردنظر که منجر به کمترین تعداد ضرب می‌شود را ارائه داد.

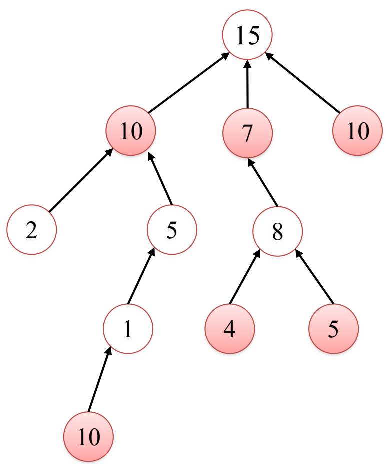
$$
\begin{array}{l}
PrintOptimalParents(s,i,j) \\
\quad if\ \ i=j \\
\qquad then\ \ print\ \ A_i \\
\quad else\ \ print\ \ "(" \\
\qquad PrintOptimalParents(s,i,s[i,j]) \\
\qquad PrintOptimalParents(s,s[i,j]+1,j) \\
\qquad print\ \ ")"
\end{array}
$$

۶. برنامه‌ریزی مهمانی شرکت

فرض کنید می‌خواهید کارمندان یک شرکت را به مهمانی دعوت کنید. هر کارمند (مثل i) یک امتیاز اجتماعی بودن به میزان v_i دارد. همچنین به علت معذب بودن کارمندان، نمی‌خواهیم هیچ کارمندی هم‌زمان با مدیر مستقیمش در مهمانی باشد. چگونه کارمندان را به مهمانی دعوت کنیم، به طوری که میزان امتیاز روابط اجتماعی مهمانی بیشینه باشد؟

به طور مثال، بیشینه مجموع امتیاز روابط اجتماعی حالت زیر، در صورتی که راس‌های رنگی را به مهمانی دعوت کنیم، برابر است با:

$$10+7+10+4+5+10=46$$



پاسخ :

برای حل این مسئله، ابتدا باید دقت شود، اگر یک مدیر به مهمانی دعوت شود، هیچ‌یک از کارمندانی که مستقیم از او دستور می‌گیرند، دیگر امکان حضور در مهمانی را ندارند؛ و این تصمیم‌گیری برای حضور یا عدم حضور مدیر باید به گونه‌ای باشد که مجموع امتیازات بیشینه شود. برای حل این مسئله به روش برنامه‌ریزی پویا پیش می‌رویم.

تعریف DP:

$A[i]$ را بیشینه مجموع امتیاز روابط اجتماعی برای شرکتی که i مدیرعامل شرکت است، و خود او نیز به مهمانی

دعوت می‌شود، تعریف می‌کنیم (به زبان گرافی، پاسخ مسئله برای زیردرخت i را $A[i]$ می‌گوییم).

$B[i]$ را به طور مشابه، بیشینه مجموع امتیاز روابط اجتماعی برای شرکتی که i مدیرعامل شرکت است، و خود او نیز به مهمانی دعوت نمی‌شود، تعریف می‌کنیم.

پایه DP:

در این مسئله، پایه‌ها برگ‌های گراف هستند. به ازای تمامی برگ‌ها $A[i] = v_i, B[i] = 0$، چراکه هیچ کارمندی ندارند و بهترین حالت دعوت خود این افراد به مهمانی است.

رابطه DP:

$$B[i] = \sum_{j \in \text{child}(i)} A[j]$$
$$A[i] = \max\left\{v_i + \sum_{j \in \text{child}(i)} B[j], B[i]\right\}$$

پاسخ DP:

طبق تعاریف گفته شده، پاسخ مسئله، حالت بیشینه در ریشه درخت داده شده است. به عبارتی، پاسخ برابر $max(A[root], B[root])$ است.

```
PartyPlanning(v, i) {
    // A and B are defined outside the function
    A[i] = v[i]
    B[i] = 0
    for j ∈ i’s children {
        PartyPlanning(v, j)  // DFS
        // A[j] and B[j] are already calculated
        A[i] += B[j]
        B[i] += A[j]
    }
    A[i] = max(A[i], B[i])
}
```

Runtime complexity: $O(n)$
Space complexity: $O(n)$

۷. فاصله ویرایش

دو کلمه به شما داده می‌شود و از شما خواسته شده فاصله‌ی ویرایشی بین این دو کلمه را به دست آورید. فاصله‌ی ویرایشی بین دو کلمه، کمترین تعداد حذف، اضافه، یا تغییر حروف است، که نیاز داریم تا یکی از این کلمات را به دیگری تبدیل کنیم. برای مثال فرض کنید می‌خواهیم فاصله‌ی کلمه‌ی $Intention$ و $Execution$ را محاسبه کنیم. این مقدار حداقل برابر پنج است:

$$Intention \longrightarrow Inteution \longrightarrow Intecution \longrightarrow Inecution \longrightarrow Ixecution \longrightarrow Execution$$

✓ یک خاصیت تعریف ارائه شده این است که فاصله‌ی دو کلمه، به ترتیب آن‌ها بستگی ندارد. یعنی فاصله‌ی کلمه‌ی الف با کلمه‌ی ب، برابر است با فاصله‌ی کلمه‌ی ب و کلمه‌ی الف، چون می‌توان تمام مراحل را به صورت وارونه اجرا کرد.

پاسخ :

تعریف DP:

$d_{i,j}$ را فاصله‌ی بین i خانه‌ی اول از کلمه‌ی اول و j خانه‌ی اول از کلمه‌ی دوم تعریف می‌کنیم.

پایه DP:

اگر i برابر صفر باشد، جواب برابر j است، چون باید تمام j حرف را به کلمه‌ی اول اضافه کنیم. همچنین اگر j برابر صفر باشد، جواب برابر i است، چون باید تمام i خانه‌ی کلمه‌ی اول را حذف کنیم تا به کلمه دوم برسیم.

رابطه DP:

در هر گام می‌توانیم یکی از سه کار زیر را انجام دهیم:

- حرف i ام کلمه‌ی اول را **حذف** کنیم، در نتیجه باید علاوه بر این عمل حذف، باید $i-1$ حرف اول کلمه‌ی اول را نیز به j حرف اول کلمه‌ی دوم تبدیل کنیم که این مقدار خودش یکی از خانه‌های جدول ما است، یعنی $d_{i-1,j}$ پس تعداد تغییرات برابر $d_{i-1,j}+1$ می‌شود.
- حرف j ام کلمه‌ی دوم را به کلمه اول **اضافه** کنیم، در نتیجه باید علاوه بر این باید i حرف اول کلمه‌ی اول را به $j-1$ حرف اول کلمه‌ی دوم تبدیل کنیم، یعنی $d_{i,j-1}$ پس تعداد تغییرات برابر $d_{i,j-1}+1$ می‌شود.
- حرف iام کلمه اول را به حرف jام کلمه‌ی دوم **تغییر** می‌دهیم. پس باید بعد از این $i-1$ حرف اول کلمه‌ی اول را به $j-1$ حرف اول کلمه‌ی دوم تبدیل کنیم. در نتیجه تعداد تغییرات برابر $d_{i-1,j-1}+1$ می‌شود. البته اگر این دو حرف یکسان باشند، نیازی به تغییر نیست و تعداد تغییرات در این حالت برابر $d_{i-1,j-1}$ می‌شود.

پاسخ DP:

با توجه به تعریف DP، جواب مسئله $d_{n,m}$ است؛ به طوری‌که n و m به ترتیب تعداد حروف کلمه‌ی اول و دوم هستند.

```
EditDistance(X, Y) {
    m, n = X.length, Y.length
    for i = 1 to m {
        d[i, 0] = i
    }
    for j = 1 to n {
        d[0, j] = j
    }
    for i = 1 to m {
        for j = 1 to n {
            if X[i-1] = Y[j-1] {
                d[i,j] = d[i-1, j-1]
                ptr[i, j] = 2    // Assume that array indices are zero based.
            } else {
                d[i,j] = min(d[i-1,j], d[i,j-1], d[i-1, j-1]) + 1
                ptr[i, j] = argmin(d[i-1, j], d[i, j-1], d[i-1, j-1])
            }
        }
    }
    return d[m,n], ptr
}
```

پیچیدگی الگوریتم:

از آنجایی که هر ستون فقط از روی خودش و ستون قبلش به روز رسانی می‌شود، می‌توان به جای نگه داشتن یک آرایه $m \times n$ از یک آرایه‌ی $n \times 2$ که از $O(n)$ است استفاده کنیم. اما اگر برای ما به جز تعداد تغییرات در بهترین حالت، خود این تغییرات نیز مهم باشد، در هر صورت باید یک آرایه‌ی $m \times n$ رای نگه‌داری مسیر برگشت استفاده کنیم که اندازه‌ی حافظه از $O(n \times m)$ می‌شود. اما در هر دو صورت باید $n \times m$ مقدار را محاسبه کنیم و هر محاسبه هم از $O(1)$ است. پس پیچیدگی زمانی $O(n \times m)$ است.

۸. مسائل بیشتر

برش میله:

یک میله به طول n و یک آرایه شامل قیمت تمام قطعات با اندازه‌های متفاوت داریم. می‌خواهیم حداکثر ارزش قابل دستیابی با بریدن میله و فروش قطعات حاصل از بریدن آن‌ها را به دست آوریم.

برش طناب:

یک طناب به طول n داریم. می‌خواهیم این طناب را حداقل یک‌بار به اجزائی با طول طبیعی برش بزنیم، به طوری که حاصل‌ضرب طول این قطعات، بیشینه شود. شما باید یک روش برش برای این طناب ارائه دهید.