

توضیحات

سوال اول. با استفاده از روش پویا یک آرایه دو بعدی به که شامل n سطر و ۲ ستون می باشد تعریف می کنیم. (فرض کنید رشته ورودی شامل n حرف باشد). در این آرایه مقدار $dp[i][0]$ نشان دهنده بهترین جواب از ابتدای رشته تا حرف i -ام در رشته با فرض برداشته نشدن آن حرف می باشد، همچنین مقدار $dp[i][1]$ به طور مشابه تعریف می شود با این تفاوت که خود حرف i -ام انتخاب شده باشد. مشخصا در صورتی که ما این پویا سازی را با صحت پیش بیریم جواب مساله برابر $dp[n][1]$ می باشد.

```
string s;
cin >> s;

int n = s.size();
int dp[n][2];
```

در کد بالا رشته از ورودی گرفته شده و آرایه مورد نظر را تعریف کردیم.

حالا نوبت مقادیر اولیه است. مشخصا $dp[0][0]$ برابر صفر می باشد، اما $dp[0][1]$ بستگی به حرف اول رشته دارد؛ اگر حرف اول برابر ۰ بود آنگاه مقدارش ۱ و اگر حرف اول ۱ باشد مقدارش را صفر می گذاریم به این معنی که بهتر است آن را انتخاب نکنیم و به صفر برسیم تا اینکه مقدار منفی باشد. از این دیدگاه در ادامه نیز استفاده می کنیم.

```
dp[0][0] = 0;
// Agar s[0] == '0' bood pas dp[0][1] = 1 (true cast mishe be 1)
dp[0][1] = (s[0] == '0');
```

حالا برای پویا سازی راه حل باید آرایه را از اول پیمایش کرد و ساخت.

```
/*
 * Be soorate dynamic programming pish mirim ba int dp[n][2]
 * dp[i][0]: behtarin javab dar baaze 0 ta i age khode i ro hesab nakonim
 * dp[i][1]: behtarin javab dar baaze 0 ta i age khode i ro hesab bekonim
 */
for(int i = 1; i < n; i++) {
    // Agar ozve feli ro hesab nakonim ingar dp[i-1] hastim va ozve i-
    1 om ro hesab kardim
    dp[i][0] = dp[i-1][1];

    /**
     * Hala mikhahim halati ke ozve i om ra bar midarim ra hesab konim

```

```

        * age s[i] == 1 bashe az meghdare adadie dp[i-1][1] yek vahed kam mishe
    */
int v = dp[i-1][1] + (s[i] == '1' ? -1 : +1);

/**
 * va age in meghdar manfi beshe behtare kolan bikhiale meghdar haye ghab
l az i beshim
 * va az jeLo edame bedim
 */
dp[i][1] = (v < 0 ? 0 : v);
}

```

واضح است که $dp[0][0]$ برابر $dp[1][1]$ می باشد ولی برای $dp[i][1]$ باید به حرف i -ام رشته نیز توجه کرد. اگر . بود باید یک واحد به مقدار اضافه کند و اگر ۱ بود یک واحد کم کند. پس از کم شدن اگر مقدار منفی شد بهتر است کلا قبل آن را فراموش کنیم و مقدار . را ذخیره کنیم.

```

int ans = 0;
// Javab bara javab ham bayad rooye noghte payan ye Loop zad
for(int i = 0; i < n; i++) {
    if (dp[i][1] > ans) ans = dp[i][1];
}

cout << ans << endl;

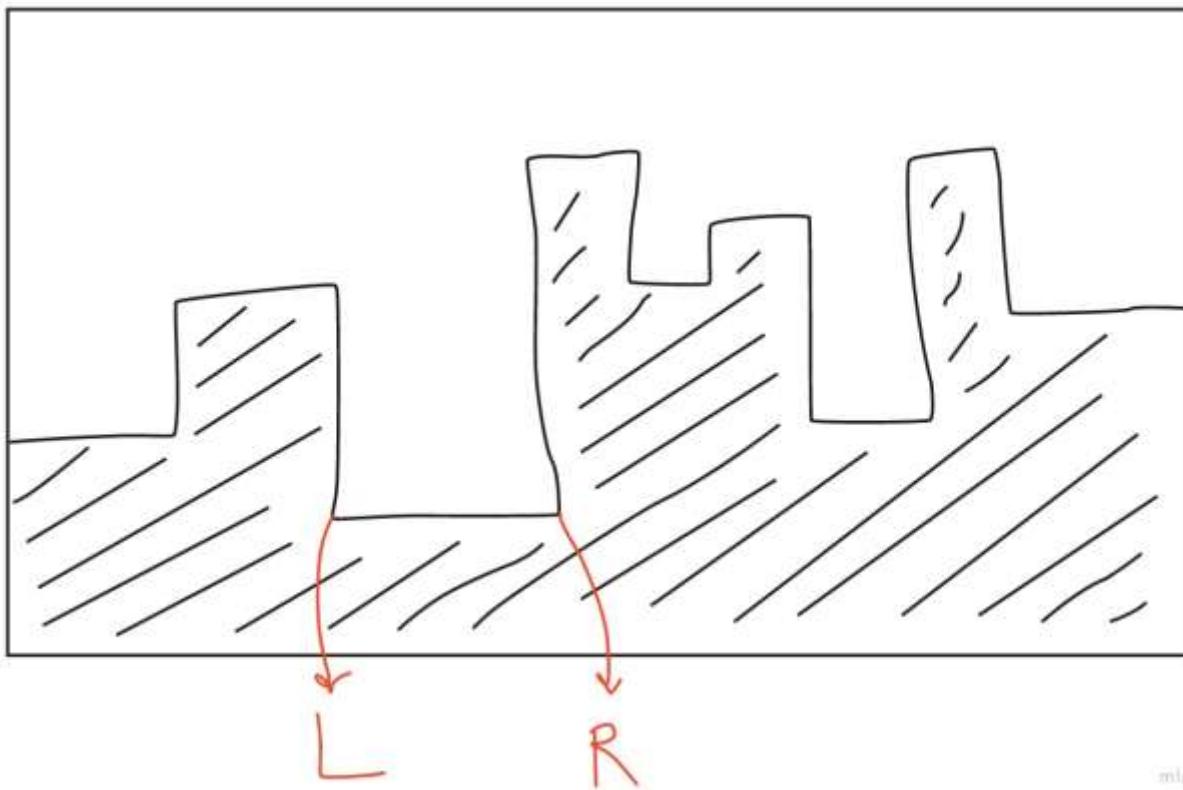
```

حالا برای مشخص کردن جواب باید تمام نقاط پایان را آزمود و بزرگ ترین جواب را نمایش داد.

تحلیل پیچیدگی زمان. مقدار دهی آرایه dp در یک پیمایش یک سطحی اتفاق می افتد که به اندازه n می باشد. برای بدست آورند مقدار ans هم یک پیمایش صورت میگیرد ولی باز هم به اندازه n پس در کل این الگوریتم در $O(n)$ اجرا می شود.

تحلیل پیچیدگی حافظه. در کل یک آرایه ۲ در n استفاده شده پس نیاز این الگوریتم به حافظه از مرتبه $O(n)$ می باشد.

سوال دوم. برای حل این سوال به روش پویا با این روند پیش می‌رویم که حالت اولیه را جدول خالی در نظر می‌گیریم و شروع به الگوریتم زیر می‌کنیم. البته قبل از آن ذکر می‌کنم که در این الگوریتم هیچ‌گاه در یک ستون هیچ خانه‌ای پر نمی‌شود مگر اینکه تمامی خانه‌های پایینش در همان ستون پر شده باشد. پرشدن به معنی آن است که مربعی وجود داشته باشد که آن خانه را پوشش داده باشد. حالت فرضی را در الگوریتم با h نشان می‌دهیم که آرایه‌ای از تعداد مربع‌های پر شده در هر ستون می‌باشد. با توجه به شرایطی که ذکر شده چون هر ستون به صورت یکپارچه از پایین تعدادی بزرگ‌تر از یا مساوی با صفر مربع پر شده ازین پس به هر ستون برج می‌گوییم و ارتفاع هر برج را با $[i]h$ نشان می‌دهیم.



اولین برج را که کمترین ارتفاع را از سمت چپ دارد مشخص می‌کنیم و اندیس آن را در L ذخیره می‌کنیم. حالات تمام برج‌های بعد از این که به صورت متواالی آمده اند و ارتفاع‌شان کمترین مقدار هست (برابر $[L]h$ می‌باشد) در نظر بگیرید، حال فرض کنید آخرین برجی که هست اندیسش R می‌باشد. حالات تمام مربع‌هایی که می‌توان جا داد دارای طولی بین $1 + R - L$ می‌باشد. پس تمام این حالات را چک می‌کنیم و مقدار h را بر اساس آن تغییر می‌دهیم. البته ما مقدار R را ذخیره نمی‌کنیم ولی در یک پیمایش مقادیر ممکنه را در ۲ نگه می‌داریم.

تنها یک چیز مانده آن هم چک کردن آن است که آیا حالت فعلی h را قبلاً حل کرده ایم یا نه؟ برای این باید ابزار پویا سازی مناسبی را انتخاب کرد. که آن ابزار *HashMap* می‌باشد. اما فقط در پایتون به راحتی می‌توان یک *list* را ابتدا به *tuple* تبدیل کرد و سپس آن را در *HashMap* قرار داد.

اثبات اینکه روش حریصانه‌ای که به کار بردیم بسیار ساده است. هر بار کمترین ارتفاع را پوشش داده و مقدار بالا می‌آوریم. البته مشخصاً باید ارتفاع عمودی هم سنجیده شود که مربعی که داریم آن را قرار می‌دهیم از جدول بیرون نزن. اما در کل هر بار تعداد برج‌هایی که کمترین ارتفاع را دارند حداقل یک واحد کم می‌شوند پس بعد از چندین بار اجرا مقدار عددی کمترین ارتفاع حداقل یک واحد افزایش پیدا می‌کند. اینجا به دلیل مشخص بودن سقف برای ارتفاع محدود بودن تعداد برج‌ها، ناوردا بی به وجود می‌آید که گام‌های ما مقدار عددی کمترین ارتفاع و تعداد برج‌ها با کمترین ارتفاع را به ترتیب افزایش و کاهش می‌دهد و به ترتیب حد های B برای مقدار عدد کمترین ارتفاع سقف می‌باشد و ۱ کفی برای تعداد برج‌هاست.

نقشه بازگشت حالتی است که تمام برج‌ها ارتفاعی برابر با B داشته باشند.

برای انتقال بهتر منظورم کد پایتونی که زدم و تست هایم را به درستی حل در این قسمت درج می‌کنم و توضیح می‌دهم:

```
def solve(h: tuple, n: int, m: int, d: dict):
    # dp check
    if h in d:
        return d[h]
```

این صورت تابع است و هر بار ارتفاع برج‌ها را در یک $tuple$ از آرگومان‌هایش می‌گیرد. همچنین یک رفرانس از دیکشنری می‌گیرد که نقش $HashMap$ را بازی می‌کند.

```
L = 0
min_height = h[0]

# Peyda kardane min_height va L
for i in range(len(h)):
    if (h[i] < min_height):
        min_height = h[i]
    L = i
```

سپس برای ادامه کار ما نیاز به اولین برجی داریم که کمترین ارتفاع دارد. با یک پیمایش همانند کد بالا آن‌ها را بدست می‌آوریم.

```
# Noghte nahayee
if min_height == n:
    return 0
```

نقشه بازگشت: اگر کوتاه‌ترین برج ارتفاع‌اش به سقف مستطیل خورده باشد پس کارمان تمام است و دیگر حالتی بعد از این وجود ندارد. پس نمی‌توان این را ادامه داد و به این دلیل . را برمی‌گرداند.

```
# Rah haye por kardan nmitune bishtr az n * m bashe
# chon age hame ro 1*1 bzarim mishe in
ans = n * m
```

لم. جواب مساله همیشه کوچک‌تر از ضرب ابعاد مستطیل است. این هم واضح است چرا که اگر همه خانه‌ها را با مربع‌های واحد پرکنیم این عدد بدست می‌آید.

```
# copy mikonim baraye check kardane hameye halat ha
dummy = list(h)
```

از روی حالت برج ها یک لیست موقعی می سازیم و قرار است برای قرار دادن تمام مربع های ممکن از آن استفاده کنیم.

```
# Gharar dadane tamame moraba hayee ke emkan darad
for R in range(L, m):
    if dummy[R] == min_height:
        width = R - L + 1

        if min_height + width <= n:
            dummy[L:R+1] = [min_height + width] * width
            ans = min(ans, solve(tuple(dummy), n, m, d) + 1)
    else:
        break
```

تمام حالت های ممکن برای نقطه پایانی مربعی که می خواهیم قرار دهیم را با R پیمایش می کنیم و اگر در پیمایش به برجی رسیدیم که ارتفاعش با کوتاه ترین برج متفاوت بود پیمایش را قطع کند.

حالا برای یک R مشخص که می خواهیم حالت را بررسی کنیم ابتدا طول مربع را بدست می اوریم و در نگه داری می کنیم. سپس به صورت عمودی این مربع را قبل اینکه قرار دهیم بررسی می کنیم اگر در مستطیل جا می شد، لیستی که به عنوان لیست کمکی از روی h کپی کرده بودیم را تغییر می دهیم به این صورت که از خانه L تا خود خانه R باید به مقدار $width$ بالا بیانند و سپس این حالت جدید را به تابع پاس می دهیم. همچنین جواب را به علاوه یک کرده و با ans مقایسه می کنیم که اگر کمتر شد مقدار بهتر را در خود نگه دارد.

```
# dp assignment
d[h] = ans

return d[h]
```

در انتهای هم باید کارهای مربوط به پویا سازی را انجام داد و مقدار نهایی را برگرداند.

تحلیل پیچیدگی زمان. چون پویا سازی انجام شده پس این الگوریتم در $O(f(n, m))$ کار می کند که $f(n, m)$ تعداد حالت های ممکن است. مشخصا حد بالای $f(n, m)$ برابر 2^{nm} می باشد زیرا هر خانه یا درونش پر شده یا خالی است. اما فقط حالت های خاصی در این الگوریتم به وجود می آیند. حالت هایی که از پایین ساخته شده باشند و در یک ستون یک پارچه باشند. پس به ازای هر ستون ما n حالت داریم و در کل الگوریتم در $O(m^n)$ کار می کند.

تحلیل پیچیدگی حافظه. از آنجایی که امکان دارد تمام حالات بررسی شوند و برای هر حالت یک بار تابع صدا زده می شود پس حداقل m^n رو داریم. ولی برای خود دیکشنری که اعمالش به طور میانگین در

$O(\log m^n) = O(n \log m)$ انجام می شود خواهیم داشت: که در برابر قسمتی که مربوط به *functional call* است. پس در کل نیاز الگوریتم به حافظه از $O(m^n)$ می باشد.

سوال سه. به صورت استقرایی فکر می کنیم و عضو n ام را عضو جدید در نظر میگیریم. این عضو یا تنها می ماند و یا به یک نفر دوست می شود. برای دوست شدن با یک نفر $1 - n$ حق انتخاب دارد.

$$f(n) = (n - 1) * f(n - 2) + f(n - 1)$$

اثبات درستی. به ازای یک n وقتی تابع صدا زده می شود. ابتدا چک می شود که آیا برای این مقدار قبلاً جواب حساب شده یا نه. اگر بله که نیازی به محاسبه نیست و مقدارش را از آرایه مورد نظر بر می گرداند. اگر هم محاسبه نشده باشد به صورت بازگشتی طبق فرمول بالا محاسبه می شود. هر کدام از توابعی که به کار رفته شده دوباره چک می کند که آیا قبلاً مقدار آرگومان خودشان در آرایه پویا وجود دارد یا نه؟

```
int solve(int n, vector<int> &dp) {
    if (n < 2) return 1;

    if (dp[n] == -1) {
        // Esteghrayee fekr mikonim

        // Ozve jadid ya ba yeki az (n-1) nafare dg doost mishe => (n-1) * f(n-2)
        // ya tanha mimoone => f(n)
        dp[n] = (n-1) * solve(n-2, dp) + solve(n-1, dp);
    }

    return dp[n];
}
```

تحلیل پیچیدگی زمان. از آنجایی که آرایه پویا شده اندازه اش n می باشد و هر مقدار یک بار محاسبه می شود پس: $O(n)$

تحلیل پیچیدگی حافظه. بهوضوح $O(n)$ که اندازه آرایه است.

سوال چهار. دو رشته ورودی را به ترتیب s و t در نظر بگیرید. همچنین برای قسمت پویا سازی یک آرایه دو بعدی $(n + 1) \times (m + 1)$ در نظر بگیرید. مقدار $dp[i][j]$ نشان دهنده جواب مساله برای زیر رشته $s[0:i+1]$ و $t[0:j+1]$ می باشد. ما سه نوع حرکت داریم که این حرکت ها را در آرایه dp شبیه سازی می کنیم:

- حذف: یک واحد از بعد اول کم کن. $i: i - 1$
- درج: یک واحد از بعد دوم کم کن. $j: j - 1$
- تغییر: یک واحد از هر دو بعد کم کن. $i: i - 1, j: j - 1$

تنها کافی است بین این سه حرکت کمترین مقدار را مشخص کنیم و در $[j][i] dp$ قرار دهیم.

نحوه کار. این الگوریتم از حرف اخر s با متغیر i و از حرف آخر t با متغیر j شروع به پیمایش می کند. در ادامه اگر s_i برابر t_j بود یک واحد پیمایش را پیش می برد بدون آنکه هزینه ای حساب شود ولی اگر متفاوت بود سه حالتی که در بالا مشخص شد را امتحان می کنیم.

اثبات درستی. نقاط بازگشت این الگوریتم زمانی هستند که حداقل یکی از i و j به صفر برسند. مقدار آن یکی که صفر نیست کمترین هزینه می شود. زیرا باید به آن تعداد عمل درج را انجام داد.

```
int solve(string s, string t, int i, int j, vector<vector<int>> &dp) {
    if (i == 0 || j == 0) return i + j;

    if (dp[i][j] != -1) return dp[i][j];

    // Akharin harf yeksan
    if (s[i-1] == t[j-1]) {
        return solve(s, t, i-1, j-1, dp);
    }

    // Akharin harf tafavot dare
    int remove = solve(s, t, i-1, j, dp) + 1;
    int insert = solve(s, t, i, j-1, dp) + 1;
    int replace = solve(s, t, i-1, j-1, dp) + 2;

    dp[i][j] = min( min(remove, insert), replace);

    return dp[i][j];
}
```

آرگومان های این تابع:

- s و t رشته های مساله
- i و j اندیس های پیمایش برای رشته ها
- آرایه دو بعدی برای پویا سازی dp

سپس در ابتدا بررسی می کنیم که آیا به نقطه بازگشت رسیدیم یا نه؟

سپس اگر حروف متناظر با i و j برابر بودند ادامه بده و هزینه ای رادر نظر نگیر. در غیر این صورت برای حذف و درج هزینه یک واحد و برای تغییر هزینه ۲ واحد را حساب کن و کمترین مقدار بین این سه مقدار را که به صورت بازگشته حساب می شوند را به عنوان خروجی باز می گرداند. بررسی اولیه نقطه بازگشت همانند همه راه حل های سوالات قبل و بعد از روش *memorization* استفاده می کند و باعث کاهش هزینه زمانی می شود.

تحلیل پیچیدگی زمان. از آنجایی که تمام خانه های جدول دقیقا یک بار محاسبه می شوند. (چون ما هم می توانیم از ز یک واحد کم کنیم و هم i) و به کمک پویا سازی این الگوریتم در $O(mn)$ اجرا می شود.

تحلیل پیچیدگی حافظه. یک آرایه دو بعدی به ابعاد $m \times n$ داریم پس: $O(nm)$ حافظه نیاز است.