1- این سوال با افراز آرایه قیمت‌ها به تعدادی زیر آرایه صعودی و خرید اتریوم در اول بازه و فروش آن در انتهای بازه در زمان $\mathcal{O}(n)$ قابل حل است. اما برای حل آن به روش برنامه‌ریزی پویا، به روش زیر عمل می‌کنیم:

در ابتدا واضح است که در i روز حداکثر i فروش خواهیم داشت. آرایه دو بعدی dp را با اندازه $(n+1)\times(n+1)$ تعریف می‌کنیم به صورتی که dp[i][j] برابر با بیشترین سودی باشد که از طریق خرید و فروش اتریوم در i روز اول و با حداکثر تعداد j بار فروش بدست می‌آید.

با توجه به اینکه مطلوب مسئله، بیشترین سود ناشی از خرید و فروش اتریوم در n روز است و محدودیتی برای تعداد خرید و فروش وجود ندارد (بیشتر از n فروش تفاوتی در جواب مسئله ایجاد نمی‌کند و حداکثر n فروش خواهیم داشت)، پاسخ نهایی مسئله dp[n][n] خواهد بود.

در هر روز 2 انتخاب داریم، یک انتخاب این است که فروشی انجام ندهیم و انتخاب دیگر این است که اتریومی را که از روزهای گذشته خریداری کردیم را به فروش برسانیم. اگر فروشی انجام ندهیم، در این روز سودی نصیبمان نمی‌شود و بیشترین سودی که تا این روز (روز i-ام) کسب کرده‌ایم برابر است با بیشترین سودی که تا روز قبل (روز (i-1)-ام) کسب کردیم. اما اگر بخواهیم اتریوم خریداری شده از روزهای قبل را به فروش برسانیم، باید بررسی کنیم که فروش اتریوم ناشی از خرید در کدام یک از روزهای قبلی سود بیشتری خواهد داشت. در نهایت بین 2 انتخاب ذکر شده، حالتی را انتخاب می‌کنیم که بیشترین سود را داشته باشد.

$firstOption = dp[i-1][j]$ عدم فروش اتریوم در روز مورد نظر

$secondOption = max(p_i - p_k + dp[k][j-1])\ (1 \le k \le i-1)$ فروش اتریوم خریداری شده

$dp[i][j] = max(firstOption, secondOption)$ انتخاب حالتی که با بیشترین سود همراه است

محاسبه secondOption برای هر خانه سبب می‌شود که محاسبه مقدار هر خانه آرایه در زمان $\mathcal{O}(i)$ انجام شود که مطلوب ما نیست. با کمی دقت مشاهده می‌کنیم که محاسبه secondOption به صورت زیر قابل بازنویسی است:

$$secondOption = p_i + max(-p_k + dp[k][j-1])\ (1 \le k \le i-1)$$

$$= p_i + max(prevMax, -p_{i-1} + dp[i-1][j-1]),$$

$$prevMax = max(-p_k + dp[k][j-1])\ (1 \le k \le i-2)$$

همانطور که مشاهده می‌شود، prevMax در مرحله قبل محاسبه شده و این امکان را به ما می‌دهد که مقدار dp[i][j] را در زمان $\mathcal{O}(1)$ محاسبه کنیم.

برای محاسبه حالت پایه، واضح است که اگر در روز 0 باشیم، هیچ سودی نمی‌توانیم کسب کنیم و همچنین اگر حداکثر تعداد فروش تا روز i-ام برابر با 0 باشد، باز هم تا آن روز نمی‌توانیم سودی کسب کنیم:

$$dp[i][j] = 0\ if\ i == 0\ or\ j == 0$$

نحوه پر کردن این جدول به صورت Row-Major است، ابتدا یک سطر را از ابتدا تا انتها پر کرده و سپس به سراغ سطر بعدی می‌رویم.

در این الگوریتم هر خانه جدول یک بار پیمایش می‌شود و زمان انجام این پیمایش $O(1)$ است و در نتیجه زمان انجام کل الگوریتم $O(n^2)$ خواهد بود.

```
function MaxProfit(p, n) do
    // p is the array of prices and is one-base indexed
    declare dp[n + 1][n + 1]
    for (i = 0; i ⩽ n; ++i) do // base case
        dp[i][0] = 0 // profit is 0 if no transaction can be made
        dp[0][i] = 0 // profit is 0 on day 0
    end

    set prevMax = -infinity
    for (i = 1; i ⩽ n; ++i) do
        for (j = 1; j ⩽ n; ++j) do
            prevMax = max(prevMax, dp[i - 1][j - 1] - p[i - 1])
            dp[i][j] = max(dp[i - 1][j], prevMax + p[i])
        end
    end

    return dp[n][n]
end
```

برای حل این سوال به کمک الگوریتم بازگشتی حافظه‌دار، از همان رابطه بازگشتی و حالات پایه ذکر شده استفاده می‌کنیم که در این صورت، شبه کد الگوریتم به شکل زیر خواهد بود:

```
function MaxProfitMemoization(p, n, table, day, transaction) do
    if (table[day][transaction] ≠ -1) do
        return table[day][transaction]
    end
    if (day == 0 or transaction == 0) do // base case
        table[day][transaction] = 0
        return table[day][transaction]
    end

    set maxOption = MaxProfitMemoization(p, n, table, day - 1, transaction)
    for (i = 1; i < day; ++i) do
        maxOption = max(maxOption, p[day] - p[i] + MaxProfitMemoization(p, n, table, i, transaction - 1))
    end
    table[day][transaction] = maxOption
    return table[day][transaction]
end

function MaxProfit(p, n) do
    set table[n + 1][n + 1] = { -1 }
    return MaxProfitMemoization(p, n, table, n, n)
end
```

2- آرایه 3 بعدی dp را در اندازه‌های $(m+1)\times 31\times (k+1)$ تعریف می‌کنیم به طوری که dp[i][j][k] نشان‌دهنده بیشترین امید ریاضی باشد زمانی که i ماه باقی مانده باشد و از ماه فعلی j روز باقی مانده باشد و همچنین تعداد مرخصی‌های باقی‌مانده نیز k عدد باشد. با توجه به اینکه برنامه‌ریزی برای m ماه است، پاسخ نهایی مسئله برابر با dp[m][30][k] خواهد بود.

برای هر روز 2 حالت وجود دارد، یا مرخصی می‌گیرد و یا مرخصی نمی‌گیرد. در هر 2 حالت تعداد روزهای باقی‌مانده یک واحد کم می‌شود. در حالتی که مرخصی می‌گیرد، یک روز از تعداد مرخصی‌ها هم کسر می‌شود. همچنین در این حالت، احتمال کوه رفتن در این روز برابر است با $\frac{1}{Days\ Left}$ که این مقدار با توجه به خطی بودن امید ریاضی، با امید ریاضی کوه رفتن در کل ماه‌های بعدی جمع می‌شود. در نهایت بین این 2 حالت، حالتی انتخاب می‌شود که بیشترین امید ریاضی را دارد:

$$dp[i][j][k] = max\left(dp[i-1][30][k-1] + \frac{1}{j}, dp[i][j-1][k]\right)$$

برای حالت پایه، مورد زیر را در نظر می‌گیریم:

$$dp[i][j][k] = 0\ if\ i == 0\ or\ j == 0\ or\ k == 0$$

نحوه پر کردن آرایه هم به این صورت است که در یک ماه به ازای هر تعداد روز باقی‌مانده، تمام مقادیر برای مرخصی‌ها را محاسبه کرده و سپس ماه بعدی را محاسبه می‌کنیم.

همانطور که واضح است، حافظه مصرفی الگوریتم $O(mk)$ است و هر خانه یک بار و در زمان $O(1)$ محاسبه می‌شود و در نتیجه زمان انجام الگوریتم برابر با $O(mk)$ خواهد بود.

```
function MaximumExpectedValue(m, k) do
    declare dp[m + 1][31][k + 1]
    for (i = 0; i ⩽ m; ++i) do // base case
        for (j = 0; j ⩽ 30; ++j) do
            dp[i][j][0] = 0
        end
        for (l = 0; l ⩽ k; ++l) do
            dp[i][0][l] = 0
        end
    end
    for (j = 0; j ⩽ 30; ++j) do // base case
        for (l = 0; l ⩽ k; ++l) do
            dp[0][j][l] = 0
        end
    end
    for (i = 1; i ⩽ m; ++i) do
        for (j = 1; j ⩽ 30; ++j) do
            for (l = 1; l ⩽ k; ++l) do
                dp[i][j][l] = max(dp[i - 1][30][l - 1] + (1 / j), dp[i][j - 1][l])
            end
        end
    end
    return dp[m][30][k]
end
```

3- پیش از حل مسئله باید این نکته را اثبات کنیم که دورهایی که در گراف تشکیل می‌شوند نمی‌توانند با هم در هیچ یالی مشترک باشند (اشتراک در راس موردی ندارد). دلیل این مورد این است که اگر دو گراف با طول فرد داشته باشیم که در k یال با هم مشترک باشند، با حذف این k یال، دوری به طول زوج

تشکیل خواهد شد که مغایر با فرض مسئله است. اگر طول دور اول را برابر با 2n+1 و طول دور دوم را برابر با 2m+1 بگیریم، با حذف k یال، دوری به طول 2n+1-k+2m+1-k=2q تشکیل خواهد شد.
جدول تک بعدی dp به طول n را به طوری تعریف می‌کنیم که dp[i] نشان‌دهنده بیشترین هزینه یال‌های اضافه شده به زیردرخت با ریشه i باشد. واضح است که dp[root] جواب است (اگر ریشه درخت مشخص نشده بود، درخت را از یک راس ریشه‌دار می‌کنیم).
می‌دانیم که با جمع مقادیر dp فرزندان i، بیشترین هزینه یال‌ها به زیردرخت‌های فرزندان i اضافه شده است. اما همچنان می‌توان بدون تشکیل دور زوج، تعدادی یال uv را اضافه کرد. این یال‌ها، یال‌هایی هستند که اولین جد مشترکشان i است. در نتیجه ابتدا مقدار اولیه‌ای برای dp[i] تعریف می‌کنیم که همان جمع مقادیر dp برای فرزندانش است و سپس یال‌های uv را اضافه کرده و ماکسیمم می‌گیریم.
برای این کار، ابتدا مجموعه C که مجموعه فرزندان i است را تعریف می‌کنیم. سپس مقدار اولیه dp[i] را به صورت زیر تعریف می‌کنیم:

$$dp[i] = \sum_{j \in C} dp[j]$$

حالا یال‌های uv که i اولین جد مشترک u و v باشد را اضافه می‌کنیم. می‌دانیم که طول دور به وجود آمده با اضافه کردن یال uv را می‌توانیم از اضافه کردن یک واحد به طول مسیر uv قبل از اضافه کردن این یال بدست آوریم. در نتیجه اگر طول این مسیر فرد باشد، با اضافه کردن یال مورد نظر، طول دور ایجاد شده زوج خواهد بود که مطلوب ما نیست. همچنین طول مسیر uv نیز از طریق حرکت از راس i به سمت u و v قابل محاسبه است. مجموعه E شامل رئوس دور حاصل از اضافه کردن یال uv را تعریف می‌کنیم.
برای محاسبه تمام حالات بدون اشتراک یال در دورها، آرایه 2 بعدی helper را تعریف می‌کنیم به صورتی که helper[i][j] نشان‌دهنده بیشترین هزینه یال‌های اضافه شده به زیر درخت با ریشه i بدون اضافه کردن یال به زیردرخت فرزند j-اش باشد. محاسبه این مقدار با کمک رابطه dp[i]-dp[j] قابل انجام است.
متغیر s را تعریف می‌کنیم و مقدار اولیه آن را برابر با جمع مقادیر dp برای فرزندان i که در E نیستند قرار می‌دهیم:

$$s = \sum_{j \in C, j \notin E} dp[j]$$

حالا از راس i به سمت u و v حرکت کرده و به ازای هر راس مانند m که فرزند n-اش در E قرار دارد، مقدار helper[m][n] را به s اضافه می‌کنیم. چون در این حالت به helper[i][j] نیازی نداریم و تمام mها فرزندان i هستند و مقادیر dp آن‌ها قبلا محاسبه شده، در این حالت مشکلی به وجود نمی‌آید. با انجام این کار، یال uv به همراه بیشترین وزن یال‌ها بدون ایجاد دور با یال مشترک به گراف اضافه شده‌اند. در نهایت با استفاده از dp[i]=max(dp[i], s)، مقدار dp[i] را محاسبه می‌کنیم.
برای حالت پایه، مقدار dp برای برگ‌ها را برابر با 0 قرار می‌دهیم. جهت آپدیت هم از طرف برگ به سمت ریشه است (در واقع از معکوس DFS استفاده می‌کنیم). هر راس یک بار پیمایش می‌شود ($\mathcal{O}(n)$) و به ازای هر راس تمام یال‌ها جهت اضافه کردن یال uv پیمایش می‌شوند ($\mathcal{O}(m)$) و به ازای هر یال حرکت به سمت u و v، در زمان $\mathcal{O}(n)$ انجام می‌شود که در نهایت مرتبه زمانی الگوریتم $\mathcal{O}(n^2m)$ خواهد بود.

4- در ابتدا برای کاهش هزینه بررسی اجبار پرانتز باز در یک خانه، یک آرایه به صورت 0 و 1 به طول 2n ایجاد کرده و اندیس‌هایی که در آرایه داده شده به طول m وجود دارند را در آرایه جدید مقداردهی می‌کنیم (آرایه را به طول 2n+1 ایجاد کرده و از اندیس 1 شروع می‌کنیم).
برای حل مسئله از راه برنامه‌ریزی پویا، به روش زیر عمل می‌کنیم:
آرایه دو بعدی dp را در اندازه $(2n+1)\times(2n+1)$ تعریف می‌کنیم به صورتی که dp[i][j] نشان دهنده تعداد حالاتی باشد که i خانه اول را به صورت صحیح پرانترگذاری کرده باشیم به طوری که تعداد پرانتزهای باز شده، j واحد بیشتر از تعداد پرانتزهای بسته شده باشد. منظور از پرانتزگذاری صحیح این است که عبارت بدست آمده، پیشوندی از یک عبارت صحیح پرانتزی باشد و همچنین در خانه‌هایی که به اجبار باید پرانتز باز قرار گیرد، پرانتز بسته قرار نگرفته باشد.
با توجه به اینکه برای محاسبه عبارت خواسته شده در صورت سوال باید کل 2n خانه پرانتزگذاری شده باشد و همچنین عبارت کاملا صحیح باشد و تعداد پرانتزهای باز و بسته با هم برابر باشند، پاسخ مسئله dp[2n][0] خواهد بود.
در رابطه بازگشتی، برای محاسبه مقدار یک خانه در آرایه دو حالت داریم، یا در position مورد نظر طبق آرایه داده شده، به اجبار پرانتز باز قرار می‌گیرد و یا اینکه اجباری وجود ندارد و با توجه به positionهای قبلی می‌توانیم بین پرانتز باز و یا بسته انتخاب کنیم. در حالت اول که به اجبار در position مورد نظر پرانتز باز قرار می‌گیرد، چون در این حالت j باید مقداری بزرگ‌تر از صفر داشته باشد (با باز شدن یک پرانتز جدید، قطعا تعداد پرانتزهای باز بیشتر از تعداد پرانتزهای بسته خواهد بود)، اگر j برابر با 0 باشد، هیچ حالتی برای ساخت این عبارت به صورت صحیح وجود ندارد و مقدار 0 در خانه مورد نظر dp قرار می‌گیرد. اما اگر j بزرگ‌تر از 0 باشد، تعداد حالات ساخت این عبارت دقیقا برابر با تعداد حالات ساخت عبارت پرانتزی با i-1 پرانتز و یک پرانتز باز کمتر است (j-1)، زیرا با اضافه کردن فقط یک پرانتز باز به انتهای این عبارت، به عبارت مورد نظر می‌رسیم:

$if\ j == 0:$
  $dp[i][j] = 0$
$else:$
  $dp[i][j] = dp[i-1][j-1]$

اما اگر اجباری به قرار دادن پرانتز باز در یک position نباشد، اگر j برابر با 0 باشد، نمی‌توانیم در این position پرانتز باز قرار دهیم زیرا در این صورت حتما تعداد پرانتزهای باز بیشتر از تعداد پرانتزهای بسته خواهد بود که با فرض j=0 تناقض دارد. در نتیجه در این حالت فقط می‌توانیم پرانتز بسته قرار دهیم که این کار را با اضافه کردن یک پرانتز بسته به انتهای عبارت با i-1 پرانتز و j=1 انجام می‌دهیم. دلیل j=1 این است که دقیقا باید یک پرانتز باز بیشتر از پرانتزهای بسته داشته باشیم تا بتوانیم پرانتز مورد نظر را ببندیم. اگر j برابر با 0 نباشد، در position مورد نظر می‌توان پرانتز باز و یا پرانتز بسته قرار داد. این کار را با اضافه کردن پرانتز به انتهای عبارت با i-1 پرانتز و j-1 (اضافه کردن پرانتز باز) و یا j+1 (اضافه کردن پرانتز بسته و بستن یکی از پرانتزهای باز) انجام دهیم.

$if\ j == 0:$
  $dp[i][j] = dp[i-1][j+1]$
$else:$
  $dp[i][j] = dp[i-1][j-1] + dp[i-1][j+1]$

برای حالت پایه کافیست dp[0][0] را برابر با 1 بگذاریم و dp[0][j] $(1 \le j \le 2n)$ را برابر با 0 بگذاریم.
$dp[0][j] = 1 \ if \ j == 0 \ else \ 0$
برای پر کردن جدول به صورت Row-Major عمل می‌کنیم. ابتدا یک سطر را پر کرده و سپس به سطر بعدی می‌رویم.
با توجه به اینکه هر خانه جدول را یک بار پیمایش می‌کنیم و این پیمایش در زمان $O(1)$ انجام می‌شود، مرتبه زمانی این الگوریتم برابر با $O(n^2)$ خواهد بود.
همانطور که مشخص است، حافظه مصرفی این الگوریتم برابر با $O(n^2)$ است. در الگوریتم ذکر شده واضح است که برای محاسبه هر خانه جدول، فقط به سطر قبلی نیاز داریم. در نتیجه می‌توانیم زمانی که پر کردن یک سطر جدید را شروع می‌کنیم، فقط آخرین سطر پر شده را ذخیره کرده و با این کار حافظه مصرفی را به $O(n)$ کاهش دهیم.

```
function BraceArrangementCount(openBraces, n) do
    set forcedOpenBraces[2 * n + 1] = { false }
    foreach (index in openBraces) do
        forcedOpenBraces[index] = true
    end

    set prev[2 * n + 1] = { 0 } // memory: O(n)
    prev[0] = 1

    for (i = 1; i ≤ 2 * n; ++i) do
        declare current[2 * n + 1]

        for (j = 0; j ≤ 2 * n; ++j) do
            if (forcedOpenBraces[i]) then do
                if (j == 0) then do
                    current[j] = 0
                else
                    current[j] = prev[j - 1]
                end
            else
                if (j == 0) then do
                    current[j] = prev[j + 1]
                else
                    current[j] = prev[j - 1] + prev[j + 1]
                end
            end
        end

        prev = current
    end
    return prev[0]
end
```

5- برای حل این سوال از راه برنامه‌ریزی پویا، از روش زیر استفاده می‌کنیم:

آرایه دو بعدی dp را با اندازه $n \times n$ تعریف می‌کنیم به صورتی که dp[i][j] نشان‌دهنده بیشترین سود ناشی از فروش پنیرهای شماره i تا j از سال y-ام به بعد است. با توجه به اینکه تعداد کل پنیرها n بوده و تعداد پنیرهای باقی‌مانده j-i+1 است، y برابر با n-(j-i) خواهد بود. واضح است که $j \geq i$ است. همچنین برای بازیابی مراحل فروش پنیر، آرایه دو بعدی sell با اندازه $n \times n$ را به صورت 0 و 1 تعریف می‌کنیم به طوری که اگر sell[i][j] برابر با 0 باشد، در هنگام فروش پنیرهای شماره i تا j، فروش پنیر اول (i) سود بیشتری دارد و اگر برابر با 1 باشد، فروش پنیر آخر (j) سود بیشتری دارد.

مطلوب مسئله بیشترین سود ناشی از فروش پنیرهای 0 تا n-1 است. در نتیجه برای محاسبه این مقدار، کافیست مقدار dp[0][n-1] را محاسبه کرده و return کنیم.

در محاسبه رابطه بازگشتی، برای هر خانه (زنجیره پنیرهای i تا j) دو انتخاب وجود دارد، یا می‌توانیم پنیر اول (i) را بفروشیم و پنیرهای i+1 تا j را در سال‌های بعد به فروش برسانیم، و یا اینکه پنیر آخر (j) را بفروشیم و پنیرهای i تا j-1 را در سال‌های آینده به فروش برسانیم. در نهایت حالتی را انتخاب می‌کنیم که بیشترین سود را داشته باشد.

$$y = n - (j - i)$$
$$sellBeg = p_i \times y + dp[i+1][j]$$
$$sellEnd = p_j \times y + dp[i][j-1]$$
$$dp[i][j] = max(sellBeg, sellEnd)$$
$$sell[i][j] = argmax(sellBeg, sellEnd)$$

حالت پایه را زمانی در نظر می‌گیریم که فقط یک پنیر مانده باشد و فقط یک انتخاب داریم که آن هم فروش پنیر باقی‌مانده است. با توجه به اینکه تعداد کل پنیرها برابر با n بوده است و در حال حاضر فقط یک پنیر باقی مانده است، زمان فروش این پنیر سال n-ام است و قیمت آن n برابر شده است. در نتیجه حالات پایه را به صورت زیر مقداردهی می‌کنیم:

$$dp[i][i] = n \times p_i \ (0 \leq i \leq n-1)$$

همانطور که در رابطه بازگشتی مشاهده می‌شود، برای محاسبه هر خانه جدول که معادل با فروش یک زنجیره پنیر به طول n است، به زنجیره‌های پنیر با طول n-1 نیاز داریم. در نتیجه جدول را با توجه به طول زنجیره‌های پنیر (اختلاف i و j) و به صورت صعودی پر می‌کنیم. به عبارت دیگر می‌توان گفت جدول به صورت قطری پر می‌شود.

هر خانه جدول حداکثر یک بار و در زمان $O(1)$ پیمایش می‌شود و در نتیجه مرتبه زمانی الگوریتم $O(n^2)$ خواهد بود. واضح است که حافظه مصرفی هم $O(n^2)$ است.

				j			
	n*p[i]						Answer
		n*p[i]					
			n*p[i]			?	
i				n*p[i]			
					n*p[i]		
						n*p[i]	
							n*p[i]

```
function MaxCheeseProfit(p, n) do
    declare dp[n][n], sell[n][n]
    for (i = 0; i < n; ++i) do // base case
        dp[i][i] = p[i] * n
    end
    for (l = 2; l ≤ n; ++l) do
        for (i = 0; i ≤ n - l; ++i) do
            set j = i + l - 1
            set year = n - (j - i)
            set sellBeg = p[i] * year + dp[i + 1][j]
            set sellEnd = p[j] * year + dp[i][j - 1]
            dp[i][j] = max(sellBeg, sellEnd)
            sell[i][j] = argmax(sellBeg, sellEnd)
        end
    end
    return dp[0][n - 1], sell
end

function FindMaxProfit(p, n) do
    set maxProfit, sell[n][n] = MaxCheeseProfit(p, n)
    print("{maxprofit}\n")
    set i = 0, j = n - 1
    while (i ≤ j) do
        if (sell[i][j] == 0) then do
            print("{i++}\t") // first
        else
            print("{j--}\t") // last
        end
    end
end
```

برای حل این سوال به کمک الگوریتم بازگشتی حافظه‌دار، از همان رابطه بازگشتی و حالات پایه ذکر شده استفاده می‌کنیم که در این صورت، شبه کد الگوریتم به این صورت خواهد بود:

```
function MaxCheeseProfitMemoization(p, n, table, sell, i, j) do
    if (table[i][j] ≠ -1) do
        return table[i][j]
    end
    if (i == j) do // base case
        table[i][j] = p[i] * n
        return table[i][j]
    end
    set year = n - (j - i)
    set sellBeg = p[i] * year + MaxCheeseProfitMemoization(p, n, table, sell, i + 1, j)
    set sellEnd = p[j] * year + MaxCheeseProfitMemoization(p, n, table, sell, i, j - 1)
    table[i][j] = max(sellBeg, sellEnd)
    sell[i][j] = argmax(sellBeg, sellEnd)
    return table[i][j]
end

function FindMaxProfitMemoization(p, n) do
    set table[n][n] = { -1 }
    declare sell[n][n]
    set maxProfit = MaxCheeseProfitMemoization(p, n, table, sell, 0, n - 1)
    print("{maxprofit}\n")
    set i = 0, j = n - 1
    while (i ≤ j) do
        if (sell[i][j] == 0) then do
            print("{i++}\t") // first
        else
            print("{j--}\t") // last
        end
    end
end
```

6- آرایه دو بعدی dp را در اندازه $n \times 9$ تعریف می‌کنیم به طوری که dp[i][j] نشان‌دهنده تعداد حالات ساخت رشته با طول i+1 و با شروع از کلید با عدد j+1 باشد.

با توجه به اینکه مطلوب سوال، تعداد حالات ساخت رشته‌های به طول n با شروع از هر کلیدی است، پاسخ نهایی مسئله برابر با $\sum_{i=0}^{8} dp[n-1][i]$ خواهد بود.

زمانی که می‌خواهیم رشته‌ای با طول n و با شروع از عدد i بسازیم، با کلیک کردن بر روی دکمه عدد i، رشته به طول 1 ساخته شده و کافیست رشته به طول n-1 را بسازیم. شروع این رشته به طول n-1 می‌تواند هر کدام از همسایه‌های چپ، راست، بالا و یا پایین کلید i باشد. در نتیجه تعداد حالات ساخت رشته به طول n با شروع از کلید i برابر است با جمع تمام تعداد حالات ساخت رشته به طول n-1 و شروع از هر کدام از همسایه‌های کلید i.

$$dp[m][i] = \sum dp[m-1][j] \quad (for\ each\ j\ that\ is\ i's\ neighbour\ on\ the\ left, right, up\ or\ down)$$

برای حالت پایه، رشته با طول 1 را در نظر می‌گیریم که با شروع از هر کلید i، فقط یک حالت خواهد داشت.

جدول را به صورت Row-Major پر می‌کنیم. یعنی ابتدا یک سطر را به طور کامل پر کرده و سپس به سراغ پر کردن سطر بعدی می‌رویم.

هر خانه جدول یک بار و در زمان $O(1)$ محاسبه می‌شود و در نتیجه مرتبه زمانی انجام کل الگوریتم برابر با $O(n)$ خواهد بود.

در حال حاضر حافظه مصرفی الگوریتم $O(n)$ است، اما همانطور که در رابطه بازگشتی مشاهده می‌شود، برای محاسبه هر سطر فقط به سطر قبلی نیاز داریم. در نتیجه می‌توانیم به جای ذخیره کل جدول، فقط سطر قبلی را ذخیره کنیم که در کنار سطری که در حال حاضر در حال پر شدن است، کل حافظه مصرفی را به دو سطر یا 18 خانه کاهش می‌دهد که نتیجه آن، حافظه مصرفی $O(1)$ خواهد بود.

```
function StringCount(n) do
    set numpad[3][3] = {
        { 1, 2, 3 },
        { 4, 5, 6 },
        { 7, 8, 9 }
    }
    set moves = {
        (1, 0),
        (0, 1),
        (-1, 0),
        (0, -1)
    }
    // memory: O(1)
    set prev[9] = { 1 } // base case: string of length 1
    for (i = 1; i < n; ++i) do // O(n)
        set current[9] = { 0 }
        for (row = 0; row < 3; ++row) do // O(1)
            for (col = 0; col < 3; ++col) do // O(1)
                for (move = 0; move < 4; ++move) do // O(1)
                    set (newRow, newCol) = (row, col) + moves[move]
                    if (0 ⩽ newRow < 3 and 0 ⩽ newCol < 3) do
                        // numpad[row][col] is the digit
                        // numpad[row][col] - 1 = row * 3 + col
                        current[numpad[row][col] - 1] += prev[numpad[newRow][newCol] - 1]
                    end
                end
            end
        end
        prev = current
    end
    set result = 0
    for (i = 0; i < 9; ++i) do
        result += prev[i]
    end
    return result
end
```