به نام خدا

تکلیف دوم آلگوریتم پیشرفته

فرهنیه علیدادی - 810101236

(1)

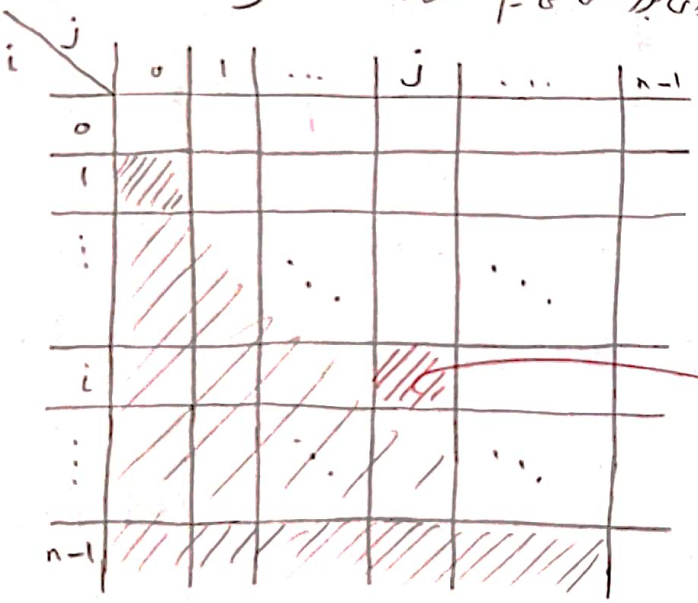
ابتدا تمام فاصله ها و تب فاصله های حله ورودی را حذف می‌کنیم. و حروف جمله ی باقی مانده را که فقط شامل حروف و punctuation است، برابر n در نظر می‌گیریم.

⊕ از یک جدول دوبعدی کمک می‌گیریم: A_{nxn}

درایه ی A[i][j] در این جدول، این را نشان می‌دهد که آیا دنباله ی letter ها از letter i تا letter j در لغت نامه، توکن بندی شود (T)، یا نه (F).

می‌تواند با استفاده از لغت نامه، توکن بندی شود (T)، یا نه (F).

با توجه به اینکه دنباله ی حروف را باید به ترتیب ورودی بررسی می‌کنیم، همواره باید ز یکی باشد. پس نقطه ی قطر اصلی



A:

و درایه های بالای قطر اصلی مقدار دارند و درایه های زیر قطر اصلی، استفاده نمی‌شوند و معتبر نیستند.

نحوه ی ترتیب پر کردن درایه ها:

1. ابتدا قطر اصلی را پر می‌کنیم. درایه ی A[i][i] به ازای $0 \le i \le n-1$، token ها یک حرف را تشخیص می‌دهند.
2. سپس درایه های بالای قطر اصلی یعنی A[i][i+1] را به ازای $0 \le i < n-1$ پر می‌کنیم. که token های باطول 2 حرف را نشان می‌دهند.
3. به همین ترتیب هر بار token های باطول بیشتر را تشخیص می‌دهیم. تا اینکه به خانه ی A[0][n-1] برسیم. این خانه، قابل توکن بندی بودن (T) یا نبودن (F) کل جمله را نشان می‌دهد.

⊕ علاوه بر این جدول، از یک جدول سه بعدی دیگر هم همزمان کمک می‌گیریم: B_{nxnxn}

در این جدول به ازای هر درایه A[i][j]، یک آرایه یک بعدی متناظر B[i][j] داریم. که درایه های این آرایه، این را نشان

ادامه‌ی ①

می‌دهد که اگر دنباله‌ی حروف نا تمام، توکن بندی می‌شود، از چه نقاطی قابل split است. برفرض مثال اگر یکی از این آرایه‌ها، مقدار K را داشته باشد، نشان دهنده‌ی این است که دنباله‌ی letter ها از شماره‌ی ① تا K، با توجه به لغت نامه‌ی موجود، قابل توکن بندی است؛ و همینطور دنباله‌ی letterها از شماره‌ی K+1 تا آخر قابل توکن بندی است. (یعنی یک یا چند کلمه‌ی موجود در لغت نامه، بخش بندی می‌شود.)

⊛ نحوه‌ی پر کردن دو آرایه A_{nxn} و B_{nxnxn} :

گفتیم برای مقداردهی به درایه‌های A، از مقطع اصلی شروع می‌کنیم. هرکدام از حروف که عیناً در لغت نامه وجود داشت، در درایه مورد نظر، مقدار T می‌گذاریم. در غیراین صورت، F می‌گذاریم.

سپس درایه‌های بالای مقطع اصلی را مقداردهی می‌کنیم. ابتدا بدون split کردن، وجود کلمه‌ی i تا i+1 را در لغت نامه بررسی می‌کنیم. اگر وجود داشت، در A[i][i+1] مقدار T می‌گذاریم. و در اولین درایه‌ی خالی آرایه B[i][i+1] مقدار -1 را ذخیره می‌کنیم؛ که به این معنی است که این کلمه بدون split شدن، در لغت نامه وجود دارد. سپس با split کردن هم، وجود بخش هایش در لغت نامه را بررسی می‌کنیم. از نقطه‌ی i آن را split می‌کنیم. با چک کردن دو درایه A[i][i] و A[i+1][i+1]، اگر هردو T بودند، یعنی دو کلمه‌ی i تا i+1 و i+1 تا آخر عیناً در لغت نامه وجود دارند. پس A[i][i+1] را T می‌کنیم. و در اولین درایه خالی آرایه B[i][i+1]، مقدار i را ذخیره می‌کنیم؛ به این معنی که با split کردن در نقطه‌ی i از حرف i تا دو بخش ایجاد شده، توسط لغت نامه موجود، توکن بندی می‌شوند.

این کار را تا انتها ادامه می‌دهیم و هر بار دنباله‌ی حروف را split نشده و split شده در تمام نقاط، در نظر می‌گیریم، و به همین ترتیب که در بالا گفته شد، A و B را پر می‌کنیم.

⊛ حال اگر A[0][n] برابر T بود، یعنی جمله توکن بندی می‌شود. و برای اینکه جمله های توکن بندی شده را بدست بیاوریم، از B_{nxnxn} کمک می‌گیریم:

آرایه B[0][n] را در نظر می‌گیریم. عددهای ذخیره شده در درایه‌های آن، این را نشان می‌دهند که با split کردن جمله، در چه نقاطی، جمله توکن بندی می‌شود. همه‌ی حالت های split را در نظر می‌گیریم. و دوباره بخش هایی

ادامه‌ی ①

که بعداز split داریم را هم، آرایه نظیرشان را در B بررسی می‌کنیم. و نحوه‌ی split شدن آن‌ها را هم بدست می‌آوریم. این کار را تا جایی ادامه می‌دهیم که آرایه‌های موجود برای بخش‌های مختلفی که داریم، فقط شامل 1- باشند (که یعنی اگر این بخش را یکپارچه درنظر بگیریم، هنوز یک کلمه است و split نمی‌شود و شامل چند واحد معنی‌دار نیست. بلکه یک واحد معنی‌دار است)، یا اینکه بخش‌بندی آن را در جایی دیگر، در یک توکن‌بندی دیگر درنظر گرفته باشیم.

⊛ پیچیدگی حافظه:

$$O(n^3) + O(n^2) = \boxed{O(n^3)}$$

$O(n^3)$ ← B
$O(n^2)$ ← A

⊛ پیچیدگی زمانی:

$$\left[O(n^2) \times O(n)\right] + O(n^3) = \boxed{O(n^3)}$$

$O(n^2)$: ساخت A
$O(n)$: بازای هر درایه A، یک آرایه یک‌بعدی در B محاسبه می‌شود.
$O(n^3)$: تولید توکن‌بندی‌های ممکن

⊛ حل مثال با این روش:

⊛ ورودی: مجموعه

0	1	2	3	4	5	6
f	a	t	s	a	n	d

⊛ لغت‌نامه:

{sand, an, fats, fat, and, a, s, at}

⊛ $A_{7\times7}$:

	0	1	2	3	4	5	6
0	F	F	T	T	T	T	T
1		T	T	T	T	T	T
2			F	F	F	F	F
3				T	T	T	T
4					T	T	T
5						F	F
6							F

یک خانه از ← A[1][2] = at

1) بدون split شدن، در لغت‌نامه وجود دارد:
A[1][2] = T & B[1][2] = -1 | ...

2) split در 1
A[1][1] = T
A[2][2] = F
& → X
X ← update، B و A می‌شوند

اطاعتی ①

* $B_{7\times7\times7}$: برای راحتی، ابتدایی شکل نشان دادم :

B[1][2] = [-1]

B[0][2] = [-1]

B[4][6] = [-1]

B[3][6] = [-1 | 3]

B[0][5] = [2 | 3]

B[3][4] = [3]

B[1][3] = [2]

B[0][3] = [-1 | 2]

B[0][4] = [2 | 3]

B[1][6] = [2 | 3]

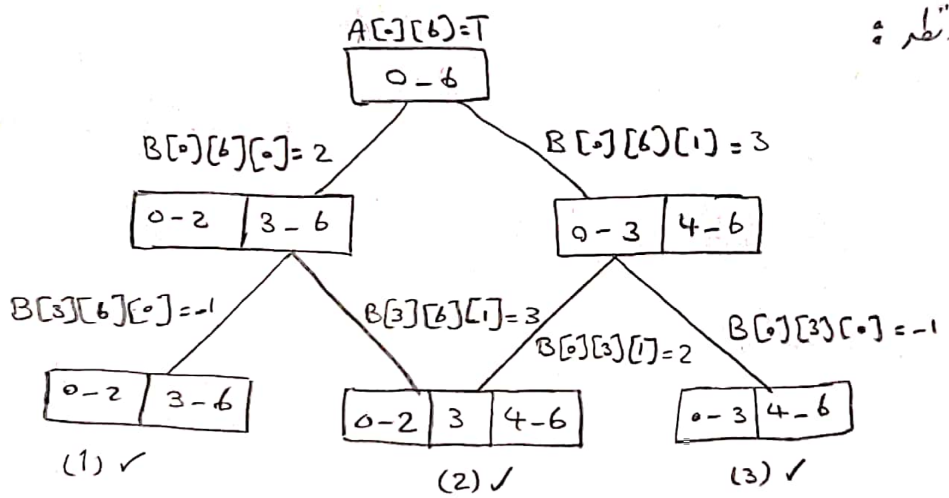
B[4][5] = [-1]

B[3][5] = [3]

B[1][4] = [2 | 3]

B[1][5] = [2 | 3]

B[0][6] = [2 | 3]



* تولید تمام ترکیب بندی‌های ممکن جمله مورد نظر :

⟹
(1) fat sand
(2) fat s and
(3) fats and

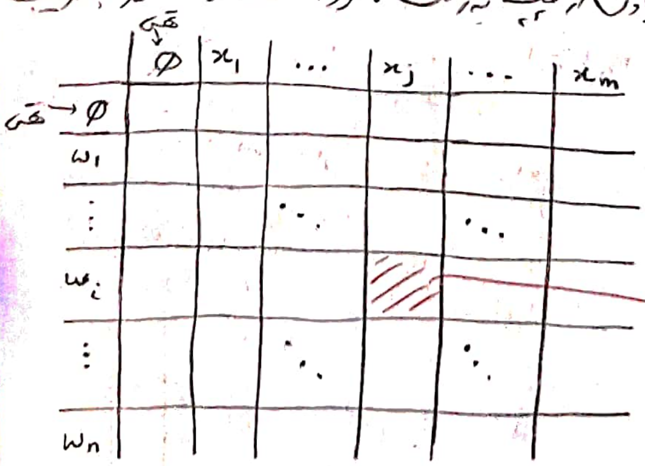
✓

---

②

کلمه‌ی دریافت شده را با W نشان می‌دهیم و طولش را برابر n درنظر می‌گیریم. و الگوی داده شده را با X نشان می‌دهیم و طولش را برابر m درنظر می‌گیریم.

⊕ یک جدول $(n+1)\times(m+1)$ به اسم A درنظر می‌گیریم. بالای سطر اول از چپ به راست، حروف متناظر الگو را به ترتیب قرار می‌دهیم. در سمت چپ ستون اول نیز، حروف متناظر کلمه‌ی داده شده را به ترتیب از بالا به پایین قرار می‌دهیم :



→ A[i][j]

$W = w_1\ w_2 \cdots w_i \cdots w_n$

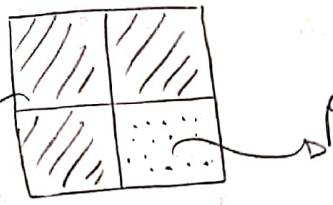
& $X = x_1\ x_2 \cdots x_j \cdots x_m$

ادامه‌ی ②

در درایه‌های این جدول، T یا F قرار می‌دهیم. A[i][j] نشان‌دهنده‌ی این است که آیا $x_1 x_2 \dots x_j$ با $w_1 w_2 \dots w_i$ قابل تطبیق است (T)، یا خیر (F).

با توجه به اینکه X شامل * است، یک X می‌تواند با یک W با طول بزرگتر یا مساوی طول خودش تطبیق شود. یعنی همواره برای $x_1 x_2 \dots x_j$ ای که با $w_1 w_2 \dots w_i$، قابل تطبیق بودن بررسی می‌شود، باید داشته باشیم: $j \le i$

درایه‌ها را به نحوی پر می‌کنیم، که هر لحظه که می‌خواهیم یک درایه را پر کنیم، درایه‌ی بالایش و درایه‌ی سمت چپش و درایه‌ی بالای درایه‌ی سمت چپش، پر باشند.



دو عبارت تهی را قابل تطبیق می‌گیریم. برای همین، درایه‌ی A[ϕ][ϕ]، T قرار می‌دهیم. و اگر یکی عبارت تهی باشد و عبارت دیگر تهی نباشد، قابل تطبیق نیستند. پس:

$$\forall_{i=1}^{n} \; A[i][\phi] = F$$

$$\& \quad \forall_{j=1}^{m} \; A[\phi][j] = F \qquad \& \qquad A[\phi][\phi] = T$$

درایه‌ی A[i][j] را به این صورت پر می‌کنیم: x_j و w_i را با هم مقایسه می‌کنیم. یکی از 4 حالت زیر ممکن است رخ بدهد:

1. اگر $x_j \neq *$، $x_j \neq ?$، $w_i = x_j$ می‌توان حرف x_j را از X حذف کرد و حرف w_i را هم از W حذف کرد. و $x_1 x_2 \dots x_{j-1}$ و $w_1 w_2 \dots w_{i-1}$ را با هم مقایسه کرد. اگر قابل تطبیق بودند، در این درایه T قرار می‌دهیم. در غیر این صورت، F قرار می‌دهیم:

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

2. اگر $x_j \neq *$، $x_j \neq ?$، $w_i \neq x_j$ در این صورت دو عبارت قابل تطبیق نیستند. در درایه‌ی A[i][j]، F قرار می‌دهیم:

$$A[i][j] = F$$

3 اگر $x_j = ?$ $\Leftarrow$ $x = ?$ باتوجه به اینکه ? با هر حرفی match می‌شود، $w_i = \text{any letter}$ را درنظر می‌گیریم. و $x_1 x_2 \dots x_{j-1}$ و $w_1 w_2 \dots w_{i-1}$ را باهم مقایسه می‌کنیم. اگر قابل تطبیق بودند، درایه‌ی T مقدار می‌دهیم. در غیراین صورت، F قرار می‌دهیم:

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

4 اگر $x = *$ $\Leftarrow$ باتوجه به اینکه * به دو صورت می‌تواند match شود، یکی به این شکل که با x_j match شود، و دیگری با این شکل که با w_i و یک سری letter دیگر از W، می‌توان دو حالت درنظر گرفت:

(1) x_j با w_i match می‌شود. پس آن را حذف می‌کنیم و $x_1 x_2 \dots x_{j-1}$ را با $w_1 w_2 \dots w_i$ مقایسه می‌کنیم. اگر قابل تطبیق بودند، T قرار می‌دهیم. در غیراین صورت F می‌گذاریم.

(2) x_j با w_i و یک سری letter دیگر قبل از w_i در W، match می‌شود. پس x_j را نگه می‌داریم، w_i را حذف می‌کنیم. و $x_1 x_2 \dots x_j$ را با $w_1 w_2 \dots w_{i-1}$ مقایسه می‌کنیم. اگر قابل تطبیق بودند، T قرار می‌دهیم. در غیراینصورت F می‌گذاریم:

$$A[i][j] = A[i][j-1] \quad \vee \quad A[i-1][j]$$

(1) (2)

* اگر در درایه‌ی $A[n][m]$، T ذخیره شده باشد، کلمه و الگوی داده شده، قابل تطبیق هستند. اگر بخواهیم نحوه‌ی تطبیق دو عبارت را مشخص کنیم، در هنگام پرکردن هر درایه، اگر مقدار آن برابر T شد، با یک فلش مشخص می‌کنیم که این مقدار T را از کدام یک از 3 درایه‌ای که از آن‌ها مقدار می‌گیرد، گرفته است.

* پیچیدگی زمانی و حافظه: $O(m \times n)$

که اندازه‌ی جدول - برای پرکردن هر درایه‌ی جدول، هزینه‌ی زمانی $O(1)$ لازم است.

ادامه‌ی (2)

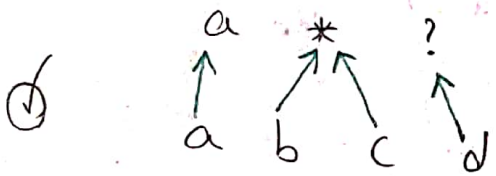
مثال (4):

$W = a\,b\,c\,d$ $\qquad$ $X = a*?$

	φ	a	*	?
φ	T	F	F	F
a	F	T	T	F
b	F	F	T	T
c	F	F	T	T
d	F	F	T	T

پس X و W با جمله‌ی تطبیق می‌شوند.

با پیاده‌سازی برعکس مسیر، نحوه‌ی تطبیق را بدست می‌آوریم:



---

(3)

الف) فرض می‌کنیم همه را در K خط به طور optimal نوشته‌ایم. هزینه‌ی کل در بدترین حالت است و از این بهتر نمی‌شود.

دو خط K و K-1 ام را به عنوان یک زیرمسئله در نظر می‌گیریم. فرض می‌کنیم که این بخش از حل مان بهینه نیست و می‌تواند از این بهتر باشد. کارهایی که می‌توانیم انجام بدهیم این است که [1] کلمات را از آخر خط K-1 ام برداریم و به ابتدای خط K ام اضافه کنیم. یا [2] کلمات را از ابتدای خط K ام برداریم و به خط K-1 ام اضافه کنیم.

+ درحالت [1]: با توجه به اینکه فضای بیشتری در انتهای خط K-1 ام اضافه می‌شود، هزینه‌ی این خط افزایش می‌یابد. خط K ام هم که همواره هزینه‌اش $\frac{1}{2}$ است و تغییری ندارد. در این صورت، با توجه به اینکه هزینه‌ی خط K-1 ام افزایش یافته، هزینه‌ی خط K ام ثابت مانده، مجموع هزینه‌ی دو خط K و K-1 ام نیز افزایش پیدا می‌کند. با توجه به اینکه هزینه‌ی کل مسئله، از مجموع هزینه‌های خطوط بدست می‌آید، هزینه‌ی کل هم افزایش پیدا می‌کند. پس بهتر از قبل نیست. و همان حالت قبل، optimal بوده است.

ادامه ی ③ الف)

+ در حالت 2 : فضای بیشتری در خط K ام اضافه می شود. پس چون خط آخر است، همچنان هزینه اش = باقی می ماند و تغییری نمی کند. و فضای موجود در انتهای خط K-1 ام هم کاهش پیدا می کند. در این صورت هزینه ی خط K-1 ام کاهش می یابد. با توجه به اینکه هزینه ی کل مسئله، از مجموع هزینه های خطوط بدست می آید، هزینه ی کل هم کاهش پیدا می کند. پس هزینه ای که قبلا" برای مسئله داشتیم، بهینه بوده. پس فرض مسئله را نقض کردیم. پس هیچ گاه این اتفاق نمی افتد و می توان یک زیرمسئله را از چیزی که هست، بهینه تر کرد، زیرا خودش بهینه است و sub-optimality برقرار است.

برای هر ی زیرمسئله این مسئله، به همین صورت، sub-optimality اثبات می شود.

ب)

$$\text{CostFunc}(i,j) = \begin{cases} \infty & \text{if } \left(M - j + i - \sum_{k=i}^{j} \text{length}(w_k)\right) < 0 \\ 0 & \text{if } \left(M - j + i - \sum_{k=i}^{j} \text{length}(w_k)\right) > 0 \text{ and } j = n-1 \\ \left(M - j + i - \sum_{k=i}^{j} \text{length}(w_k)\right)^3 & \text{else} \end{cases}$$

$$\Downarrow \text{ recursive}$$

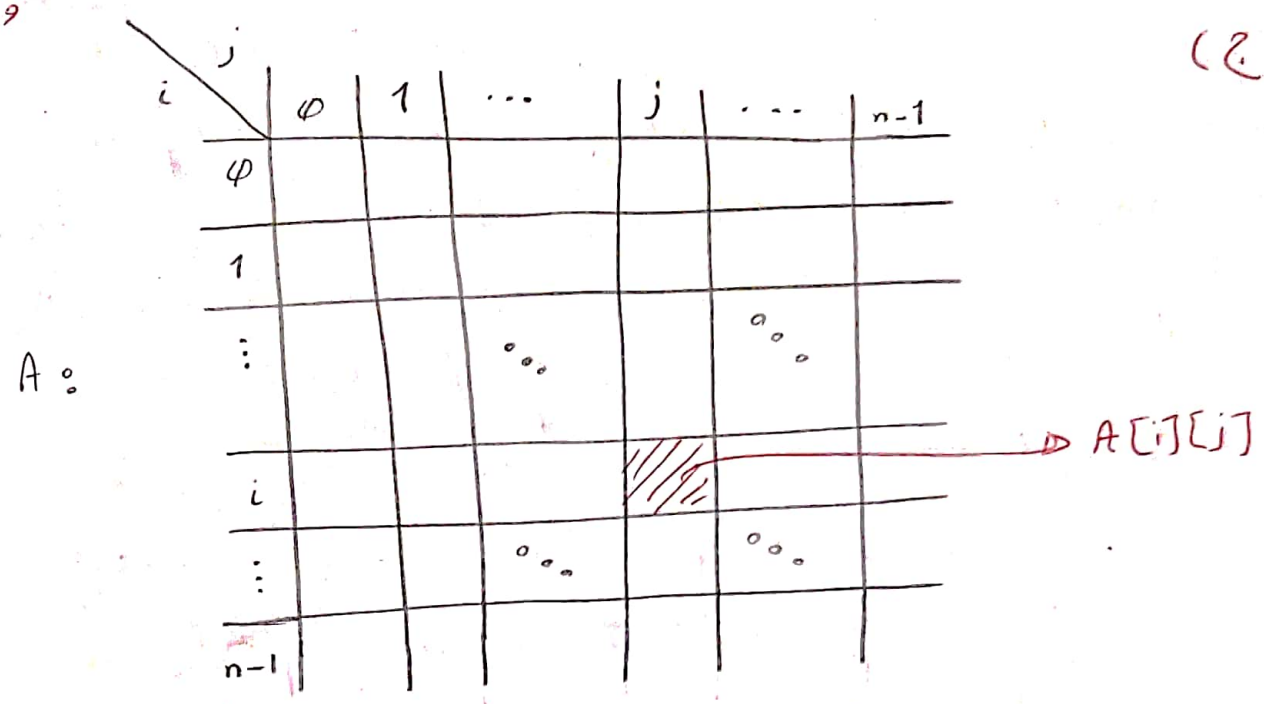
$$\text{CostFunc}(i, n-1) = \min\left(A[i][n-i],\ \min_{i<j<n-1}\left(A[i][j] + \text{Cost}[j+1]\right)\right)$$

(آرایه ها در بخش بعدی توصیف شده اند)

ج)

برای حل این سوال به روش DP، از یک جدول nxn به اسم A کمک می گیریم. که n نشان دهنده ی تعداد کلمات موجود در جمله است. و درایه ی A[i][j] مشخص می کند که یک خط شامل کلمه ی i تا کلمه ی j، چه هزینه ای خواهد داشت. اگر این کلمات در یک خط جا نشوند، ∞ قرار می دهیم. اگر خط آخر باشند، 0 قرار می دهیم. و در غیر این صورت، طبق تعداد space های آن خط را به توان 3 میرسانیم.

الف) ③



باتوجه به اینکه i تا از یک sequence از کلمات است و ترتیب آن ها هم است، همواره باید j ها باشند. بنابراین فقط درایه های قطر اصلی و بالای قطر اصلی در جدول مقداردهی می شوند و معتبر هستند.

باتوجه به اینکه هزینه ی خط آخر ۰ است، هرجا می خواهیم هزینه ی i تا n-1 را حساب کنیم، اگر j=n-1 (یعنی خط مورد نظر شامل کلمه ی آخر باشد) باشد، و اگر این کلمات بتوانند در یک خط قرار بگیرند، این خط همان خط آخر بوده و هزینه ی آن ۰ برآورد می شود.

پس از پر کردن این جدول، برای محاسبه ی پاسخ بهینه و هزینه ی آن، به دو آرایه ی یک بعدی به طول n نیاز داریم. که یکی برای بدست آوردن هزینه است با نام cost و cost[0] هزینه ی پاسخ بهینه را نشان می دهد. و دیگری با نام SplitPoint، برای بدست آوردن پاسخ بهینه استفاده می شود. عدد ذخیره شده در SplitPoint[i] نشان می دهد که از i تا SplitPoint[i] در یک خط قرار می گیرند. پس با پیمایش آرایه ی SplitPoint از ۰ تا n-1، نقاط شکستن متن به خطوط بدست می آید. که همان پاسخ بهینه است.

+ نحوه ی پر کردن دو آرایه ی cost و SplitPoint:

روی درایه ی cost[n-1]، دو انتهای نون را set می کنیم. و کم کم به خطی که در ادامه توصیف شده، آن ها را روی آرایه به عقب حرکت می دهیم. این درایه، هزینه ی تراز شدن کلمه ی n-1 ام در یک خط را نشان می دهد. درایه A[n-1][n-1]

نظری ③ (2

را بررسی می‌کنیم. این درایه برابر ∞ خواهد بود. چون با توجه به شامل بودن کلاس n-1 ام، همان خط آخر جلسه می‌شود. مقدار آن را در Cost[n-1] قرار می‌دهیم و SplitPoint[n-1] را مساوی n-1 قرار می‌دهیم. به این صفتی که از n-1 تا n-1 در یک خط قرار گرفته است. سپس i را یکی به عقب بررسی می‌کنیم: j=n-1 و i=n-2. پس درایه‌ی A[n-2][n-1] را بررسی می‌کنیم. اگر در کلاس n-1 و n-2 در یک خط قرار گیرند، درایه‌ی A[n-2][n-1] برابر ∞ خواهد بود. آن را در Cost[n-2] ذخیره خواهیم کرد. اما اگر در یک خط قرار نگیرند، A[n-2][n-1] برابر ∞ خواهد بود. بدین صورت ز را یکی کم می‌کنیم: i=n-2 ، j=n-2 . (هرگاه A[i][j] برابر ∞ باشد، ز را یکی کم می‌کنیم. به شرطی که ز یکی از n نشود.) این به این معنی است که n-2 در یک خط قرار می‌گیرد و n-1 هم در یک خط نیز قرار می‌گیرند. قرار می‌دهیم:

Cost[n-2] = A[n-2][n-2] + Cost[n-1]
SplitPoint[n-2] = j = n-2

سپس دوباره ز را یکی به عقب می‌آوریم و درایه آخر آرایه بررسی کرده یعنی: i=n-3 ، j=n-1 و A[n-3][n-1] را بررسی می‌کنیم. (که یا ∞ خواهد بود. اگر در یک خط قرار گیرند، چون خط آخر است، ∞ می‌شود. و اگر در یک خط قرار نگیرند، ∞ خواهد بود.) قرار می‌دهیم:

Cost[n-3] = A[n-3][n-3]
SplitPoint[n-3] = j = n-1

سپس ز را یکی به عقب می‌آوریم ∞ ، i=n-3 و j=n-2 و A[n-3][n-2] را بررسی می‌کنیم. اگر Cost[n-3] > A[n-3][n-2] + Cost[n-1] باشد، قرار می‌دهیم:

Cost[n-3] = A[n-3][n-2] + Cost[n-1]
SplitPoint[n-3] = j = n-2

(بدین صفتی که از n-3 تا n-2، در یک خط قرار می‌گیرند. و n-1 در خط بعدی.)

سپس ز را دوباره به عقب می‌آوریم: i=j=n-3. پس A[n-3][n-3] را بررسی می‌کنیم. اگر Cost[n-3] > A[n-3][n-3] + Cost[n-2] باشد، قرار می‌دهیم:

Cost[n-3] = A[n-3][n-3] + Cost[n-2]
SplitPoint[n-3] = j = n-3

(بدین صفتی که n-3 در یک خط و n-2 و n-1 در خط بعدی قرار می‌گیرند.)

الگوریتم ③ (؟)

این روند را به همین شکل تا به ابتدای آرایه ادامه می‌دهیم. هربار به ازای یک i، تمام j های از i تا n-1 را در نظر می‌گیریم و این update عملیات را بر روی دو آرایه‌ی Cost و Split Point انجام می‌دهیم.

+ پس از تمام شدن این عملیات دو آرایه، Cost[0] هزینه‌ی پاسخ همه‌ی را نشان می‌دهد. و برای پیدا کردن مسیر همه، SplitPoint را به این شکل از همه به راست پیمایش می‌کنیم: عدد ذخیره شده در SplitPoint[0]، نشان دهنده‌ی این است که از کلمه‌ی ۰ تا کلمه‌ی SplitPoint[0] در یک خط قرار می‌گیرند. سپس SplitPoint[SplitPoint[0]+1] را در نظر می‌گیریم. عدد ذخیره شده در آن، نشان می‌دهد که از کلمه‌ی SplitPoint[0]+1 تا SplitPoint[SplitPoint[0]+1] در یک خط قرار می‌گیرند. این روند را به همین شکل ادامه می‌دهیم ... و کلمات را در خطوط، طوری شکستن همه به خطوط با هزینه‌ی همه را بدست می‌آوریم.

Anxn Split Cost

$O(n^2)+O(n)+O(n) = O(n^2)$: پیچیدگی حافظه ✱

$O(n^2)+O(n^2) = O(n^2)$: پیچیدگی زمانی ← ✱

Anxn Cost, Split

M=10 , n=5 : مثال ← ✱

Hello	I	am	Marzieh	Alidadi
0	1	2	3	4

*A_{5X3}:

	0	1	2	3	4
0	$5^3=125$	$3^3=27$	0	∞	∞
1		$9^3=729$	$6^3=216$	∞	∞
2			$8^3=512$	0	∞
3				$3^3=27$	∞
4					0

Cost:

0	1	2	3	4
27	270	0	54	27

Split Point:

0	1	2	3	4
1	2	3	3	4

(2. ③ ص ~61

$i=j=4 \Rightarrow Cost[4] = A[4][4] = 0$

SplitPoint = 4

$i=3, j=4 \Rightarrow Cost[3] = A[3][4] = \infty$

$i=3, j=3 \Rightarrow Cost[3] > A[3][3] + Cost[4]$

So ↘ $Cost[3] = A[3][3] + Cost[4] = 27+27 = 54$

SplitPoint [3] = 3

$i=2, j=4 \Rightarrow Cost[2] = A[2][4] = \infty$

$i=2, j=3 \Rightarrow Cost[2] > A[2][3] + Cost[4] = 0+0 = 0$

So ↘ $Cost[2] = A[2][3] + Cost[4] = 0$

SplitPoint [2] = j = 3

$i=2, j=2 \Rightarrow Cost[2] < A[2][2] + Cost[3] = 512+54$

So ↘ ✗ نمی‌تونه update

$i=1, j=4 \Rightarrow Cost[1] = A[1][4] = \infty$

$i=1, j=3 \Rightarrow Cost[1] = A[1][3] + Cost[4] = \infty$

$i=1, j=2 \Rightarrow Cost[1] > A[1][2] + Cost[3] = 216+54 = 270$

So ↘ $Cost[1] = A[1][2] + Cost[3] = 270$

SplitPoint [1] = j = 2

$i=1, j=1 \Rightarrow Cost[1] < A[1][1] + Cost[2] = 729+0 = 729$

So ↘ نمی‌تونه update

$i=0, j=4 \Rightarrow Cost[0] = A[0][4] = \infty$

$i=0, j=3 \Rightarrow Cost[0] = A[0][3] + Cost[4] = \infty$

$i=0, j=2 \Rightarrow Cost[0] > A[0][2] + Cost[3] = 0+54 = 54$

So ↘ $Cost[0] = A[0][2] + Cost[3] = 54$

SplitPoint[0] = j = 2

$i=0, j=1 \Rightarrow Cost[0] > A[0][1] + Cost[2] = 27+0 = 27$

So ↘ $Cost[0] = A[0][1] + Cost[2] = 27$

SplitPoint [0] = 1

$i=0, j=0 \Rightarrow Cost[0] < A[0][0] + Cost[1] = 125+270$

So ↘ نمی‌تونه update

* حالا که آرایه‌ها تکمیل شدند، می‌ریم سراغ جواب:

هزینه‌ی بهینه = $Cost[0] = 27$

مسیر بهینه: $Split[0] = 1$

$Split[1+1] = 3$

$Split[3+1] = 4$

Hello I → $3^3 = 27$

am Marzieh → .

Alidadi → .

27 ✓

(4)

اینجا در واقع باید امید ریاضی هرکدام از n سوال را بدست بیاوریم. که با توجه به اینکه p_i احتمال سلام با شعله‌ور است، باید $1-p_i$ را ضربدر کنیم. و امید ریاضی به این شکل می‌شود: $(1-p_i)\,d_i$

پس [1] برای هرکدام از این n تا سوال، $f_i=(1-p_i)\,d_i$ را محاسبه می‌کنیم. [2] و به صورت نزولی آن‌ها را بر حسب f_i مرتب می‌کنیم. [3] از بزرگ به کوچک، یکی یکی سوال‌ها را انتخاب می‌کنیم. ← این ترتیب انتخاب، بهترین حالت برای حداکثر کردن جایزه است.

⊛ مثال:

	d_i	p_i	f_i	ترتیب انتخاب
1	10	0,1	$(10\times 0,9)=9$	[3]
2	20	0,5	$(20\times 0,5)=10$	[2]
3	30	0,3	$(30\times 0,7)=21$	[1]
4	40	0,9	$(40\times 0,1)=4$	[4]

* اثبات درستی: با توجه به مفهوم امید ریاضی، این ترتیبی که با انتخاب بیشترین امید ریاضی بدست می‌آید، بهترین ترتیب است. اگر همواره سوال با بیشترین امید ریاضی را انتخاب نکنیم، مجموع امید ریاضی‌های سوال‌های انتخاب شده، maximum نمی‌شود. و بهترین ترتیب انتخاب بدان را خواهیم داشت. و جواب بهینه خواهد بود ☑

---

(5)

در این سوال اگر کوچکترین (min) و بزرگترین (max) عدد موجود را در نظر بگیریم، عبارت $\left\lceil \frac{(max-min)}{2} \right\rceil$ به عنوان threshold بیشترین تعداد بازه‌های به طول 2 است، که در بدترین حالت برای دسته‌بندی تمام

استفاده می‌شود. هر جوابی که به دست می‌آوریم، باید از این threshold کمتر مساوی باشد.

* مثال:

1 و 1,5 و 2 و 2,5 و 3 و 3,5 و 4 و 4,5 و 5 و 5,5

$$\left\lceil \frac{max - min}{2} \right\rceil = \left\lceil \frac{5,5 - 1}{2} \right\rceil = \left\lceil \frac{4,5}{2} \right\rceil = \lceil 2,25 \rceil = 3$$

⊛ روش greedy: نقاط به صورت sort شده صعودی در آرایه قرار دارند. عدد اول که کوچکترین است را به عنوان نقطه‌ی شروع بازه‌ی اول در نظر می‌گیریم. و با توجه به اینکه طول بازه برابر 2 است، نقطه‌ی پایانش برابر آن نقطه + 2 خواهد بود. تمام نقاطی که در این بازه قرار می‌گیرند را در این دسته قرار می‌دهیم و از آرایه حذف می‌کنیم. همچنان اعداد باقی مانده به صورت صعودی مرتب هستند. دوباره عدد اول که minimum عدد بین باقی مانده هاست را انتخاب می‌کنیم و ابتدای بازه قرار می‌دهیم و اعدادی که در این بازه قرار می‌گیرند را حذف می‌کنیم. و ... همین کار را تا تمام شدن نقاط درون آرایه ادامه می‌دهیم و تعداد دسته هایی که به وجود می‌آیند را می‌شماریم.

⊛ اثبات درستی: اثبات می‌شود که این روش همواره جواب optimal را به ما می‌دهد. فرضاً داریم $x_1 x_2 x_3 x_4$ نقاط ما هستند و به صورت صعودی نشان داده شده‌اند. در روش greedy مطرح شده، گفتیم همواره x_1 را به عنوان ابتدای بازه قرار می‌دهیم و ... فرضاً این جا با قرار دادن x_1 به عنوان شروع بازه، بازه‌ی اول شامل سه عدد $x_3 x_2 x_1$ می‌شود و بازه‌ی بعدی شامل x_4. حال فرض می‌کنیم که x_1 شروع بازه نباشد و x_2 شروع بازه باشد. در بهترین حالت، اگر این بازه‌ی جدید شروع شود از x_2، شامل x_4 هم می‌شود، دوباره‌ی (x_4) $(x_2 x_3)$ (x_1) را خواهیم داشت. که همچنان مشاهده می‌شود که هیچگاه با انتخاب عدد دیگری

ادامه‌ی (5)

به غیر از عدد اول، به عنوان شروع بازه، به تعداد عددی بازه برای دسته‌بندی نقاط می‌رسیم. در بهترین حالت، تعداد بازه‌ها به اندازه‌ی وقتی است که هر بار عدد اول (min) را به عنوان شروع بازه انتخاب کنیم.

پس جواب حاصل از روش greedy همواره optimal است. ممکن است یک تقسیم‌بندی دیگر با بازه‌های دیگر وجود داشته باشد؛ ولی تعداد بازه optimal، همین تعداد یکسان بدست آمده از روش greedy است.