تکلیف سوم الگوریتم - ۹۸۲۳۰۳۳

سوال ۱ : برای اینکه جدول DP مربوطه این سوال را ایجاد کرده و نتایج را ذخیره کنیم ابتدا باید base سوال را بررسی کرده و سپس با استفاده از حالت بازگشتی بقیه درایه های جدول را ایجاد کنیم.

طبق فرض سوال می دانیم که اگر k=۱ باشد یعنی گاو نمی تواند هیچ علف خوردنی را در حین خوردن تغییر دهد پس حداکثر m کپه علف می تواند بخورد.
حال اگر جدول DP را به صورت یک آرایه دوبعدی m در n در نظر بگیریم.
درایه goal ما DP[k][n] خواهد بود و سطر اول جدول با توجه به صورت سوال k ایجاد می شود به این شکل خواهد بود:

	۰	۱	-	-	-	-	n
بر حسب k	۰	۱	۲	-	-	-	m
⋮							

حالا برای محاسبه سطرهای بعدی جدول (که باید به صورت بازگشتی محاسبه شوند).
یعنی تعداد حالت هایی که می توانند با یک بار تغییر بیشتر مقداری علف را بخورند باید از سطرهای قبلی جدول استفاده کنیم.
مثلاً اگر گاو بخواهد j کپه علف را با k بار تغییر علف بخورد می توانیم بررسی کنیم که ۱-j کپه علف را با ۱-k تغییر به چند حالت می توانسته بخورد

در نتیجه نتیجه می‌گیریم الان فقط آخری می‌تواند با یک بار تغییر نوع علف j کیسه علف را نیز بخورد. به همین ترتیب پیش می‌رویم تا اگر دو تا می‌توانستند j-m کیسه علف را با k-1 بار تغییر بخورد پس الان می‌توانند j کیسه علف را با k بار تغییر بخورد.

پس به رابطه بازگشتی می‌رسیم:

$$DP[i][j] = \sum_{k=1}^{m} \left(DP[i-1][j-k] \right)$$

حال با استفاده از این رابطه بازگشتی جدول DP که m در n است را پر می‌کنیم. و برای پر کردن هر کدام از خانه‌ها باید m خانه از ردیف قبلی را با هم جمع کنیم پس پیچیدگی زمانی الگوریتم ما از $O(mnk)$ خواهد بود.

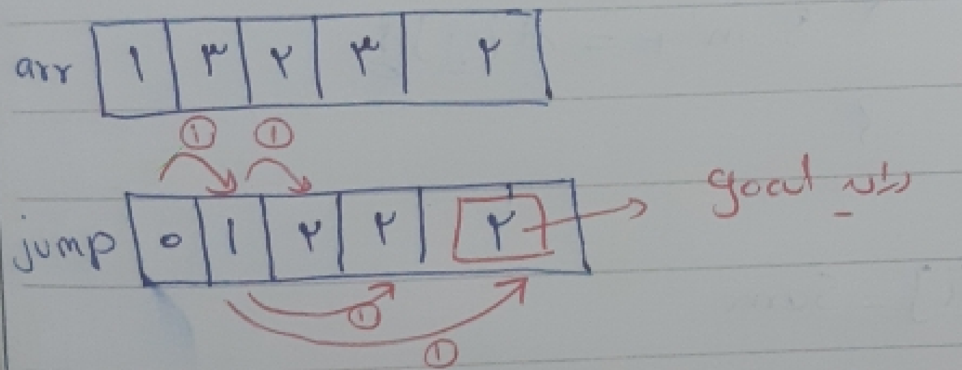
سوال ۲ :

با رویکرد DP نیاز به یک آرایه n تایی jump داریم که jump[i] نشان دهنده

مینیمم تعداد پرش هایی است که نیاز است تا از خونه arr[0] به خونه arr[i]

برسیم . حال باید با استفاده یک راه حل بازگشتی آرایه jump را پر کنیم و درایه

goal ما در واقع همان jump[n-1] خواهد شد .



برای به دست آوردن هر کدام از خانه های آرایه jump مثل jump[i] روی درایه های

از صفر تا i-1 از آرایه arr (آرایه اولیه) پیمایش می کنیم و اگر این شرط

($i <= j + arr[j]$) که $0 \leq j < i$ است برقرار شد مقدار درایه

jump[i] برابر می شود با $min(jump[i], jump[j] + 1)$.

ما از خانه j ام فقط یک پرش است تا به درایه i ام برسیم .

درایه jump[0] همواره صفر است چون نیازی به هیچ پرشی برای رسیدن به خانه اول نیست.

در ابتدا نیز درایه‌های بقیه خانه‌های آرایه غیر از خانه اول را برابر با ∞ قرار می‌دهیم که موقع مینیمم گیری این مقدار نیز اصلاح می‌شود.

در واقع arr[j] نشان دهنده این است که حداکثر تا چند خانه می‌تواند پرش کند که اگر در خانه j ام باشیم و به اندازه arr[j] از خانه‌های بعدی را بتوانیم پرش دهیم باید چک کنیم که آیا درایه i ام در محدوده تحت پرش هست یا نه که اگر بود با یک پرش اضافه‌تر به آن برسیم.

به همین دلیل شرط $i <= j + arr[j]$ چک می‌شود.

پس نیاز به دو حلقه فور تو در تو داریم که یکی شمارنده تا n و روی درایه‌های jump و یکی شمارشگر شمارنده درایه‌های صفر تا i-1 حرکت می‌کند پس از $O(n^2)$ می‌باشد.

```
jump[0] = 0
for i = 1 to n
{
    jump[i] = ∞
    for j = 0 to i
        if (i <= j + arr[j] && jump[j] != ∞)
        jump[i] = min(jump[i], jump[j] + 1)
}
return jump[n-1]
```

سوال ۳ :

یک رشته مانند S را با طول n در نظر می‌گیریم. دو حالت پیش می‌آید ، یا عنصر اول و آخر رشته S (عناصر S[0] و S[n-1] یکی می‌شوند.) در آن صورت طول بلندترین زیررشته متقارن روی S برای عناصر ۰ تا n-1 برابر می‌شود با طول بلندترین زیررشته متقارن روی S برای عناصر ۱ تا n-2 به علاوه ۲ (به خاطر عناصر صفر و n-1 که به علاوه ۲ می‌شود که نشان می‌دهد همچنان زیررشته متقارن است.)

۲ حالت دوم این است که عناصر اول و آخر رشته S با هم برابر نباشند در آن صورت طول بلندترین زیررشته متقارن max طول زیررشته متقارن از عناصر ۰ تا n-2 و عناصر ۱ تا n-1 خواهد بود. (هر دفعه یکی از عناصر ۰ و n-1 را کنار می‌گذاریم و جواب را برای زیررشته باقی مانده به صورت بازگشتی بدست می‌آوریم) .

اگر L(0, n-1) طول بلندترین زیررشته متقارن برای عناصر ۰ تا n-1 در رشته ی S باشد داریم :

$$L(0, n-1) = L(1, n-2) + 2$$

اگر S[0] = S[n-1] باشد :

$$L(0, n-1) = \max\{L(1, n-1), L(0, n-2)\}$$

اگر S[0] != S[n-1] باشد :

می‌توانیم این مقادیر را در یک آرایه‌ی دوبعدی [n][n] L ذخیره کنیم که از ۰ و ۰ شروع می‌کنیم و تا n-1 و φ پیش می‌رویم. نهایتاً مستقیم مقدار [n-1][φ] L را برمی‌گردانیم. یک پیدا کردن ماتریس داریم $O(n^2)$. بقیه عملیات از $O(1)$ می‌باشند.

۲-

الف)

```
int F[۰ ... ۲^n]
for i = ۰ to ۲^n
{   sum = ۰
    for j = ۰ to ۲^n
    {
        andoperation = i ∧ j
        if(andoperation == j)
        {   sum += A[j]
        }
    }
    F[i] = sum
}
```

همه‌ی مقادیر ۰ تا ۲^n را برای F(x) بررسی می‌کند و هر دو فور روی آرایه A حرکت می‌کنند A[i] ها را جمع می‌کند که از $O(۲^n \times ۲^n) = O(۴^n)$ می‌باشد.

می‌توانیم F را یک آرایه یک بعدی ۲^n تایی در نظر بگیریم و هر بار حاصل به دست آمده را در هر خانه متناظرش ذخیره کنیم.

ب) می دانیم که عدد باینری n رقمی اگر با یک عدد باینری m رقمی and شود
هیچ گاه نمی تواند برابر با خودش شود چون ممکنه تعدادی صفر پشت عدد
با تعداد ارقام کمتر قرار بگیرد که موقع اند کردن همه‌ی ارقام بقیه‌اش را صفر می‌کند.
پس لازم است برای محاسبه $f(x)$ هر x را با $A[i]$ هایی اند کنیم که تعداد
ارقامش با تعداد ارقام x برابر است.
پس یک محدودیت که با توجه به این موضوع روی قسمت الف می‌توانیم داشته باشیم
این است که محدوده‌ی حلقه داخلی را کوچک‌تر کنیم.
مثلاً اگر n عدد ۵ باشد باید با اعداد ۹ و ۱۰ و ۱۱ و ۱۲ و ۱۳ و ۱۴ و ۱۵ فقط
اند شود. پس محدوده‌ی مورد نظر را می‌توانیم $(\frac{3}{2})^n$ در نظر بگیریم که در نهایت
با این حساب از $O(2^n \times (\frac{3}{2})^n) = O(3^n)$ خواهد شد.