Programación dinámica

Top Down y Bottom Up

Tomás Chimenti - Santiago Cifuentes - Joaquin Laks - Ezequiel Companeetz

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

2^{do} Cuatrimestre de 2024

Programa de hoy

Se viene una pedazo de clase práctica con:

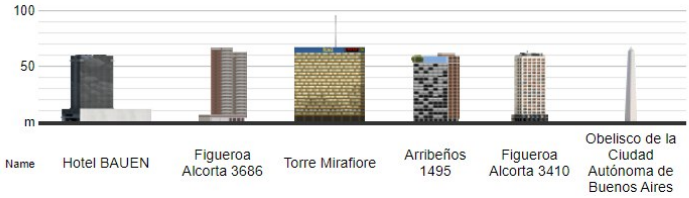
1. Kahoot de Repaso
2. Ejercicios
3. Tips + Conclusiones

Copias en los TP's

- No pueden copiarse código de internet. El único código que está permitido copiarse es el autorizado por la materia, vamos a correr un detector de copias.
- No se copien entre ustedes, es mucho trabajo investigar cada caso para ver quién se copió de quién. Así que no pasen su código a otra gente o a grupos de Telegram/Whatsapp.
- Tengan cuidado si usan Chat GPT, porque les puede dar soluciones muy similares/ iguales a las de otras personas y puede ser que eso salte en el detector de copias. Igual que antes, es muy costoso revisar caso por caso si se copiaron o ambos justo tuvieron un prompt similar.
- No tiene mucho sentido copiarse. Se pierden una instancia de aprendizaje y un momento para practicar, para entrevistas o hacer problemas que consideramos divertidos.

Kahoot Time

Ahora vamos a repasar algunas propiedades útiles de programación dinámica y backtracking.
Para eso van a tener que entrar https://kahoot.it/ e ingresar el código escrito en el pizarrón.



Mi Buenos Aires Crecido

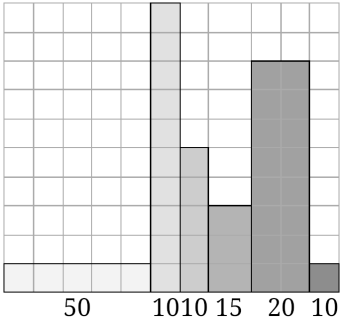
Sasha vive en San Nicolás hace mucho tiempo, y tiene una eterna discusión con su vecina Tasha. Sasha dice que si se mira el horizonte de izquierda a derecha terminando en el obelisco, los edificios están ordenados en un perfil principalmente ascendente para que la altura del obelisco sea más impresionante. Tasha le dice que con las nuevas torres que se construyeron en la zona eso ya no es verdad, y que en realidad ahora menos de la mitad del ancho del horizonte está en orden ascendente.

Entrada

Recibimos una lista de edificios ordenados de izquierda a derecha con dos datos para cada uno, por un lado, el ancho de cada edificio y por el otro el alto.

Buscamos la longitud de la máxima subsecuencia ascendente, es decir, la subsecuencia creciente de edificios que ocupe el mayor espacio horizontal.

Ejemplo



La máxima subsecuencia ascendente tiene 85 de largo, usando 3 edificios. Un edificio por su cuenta es también una subsecuencia ascendente.

Función recursiva

Tenemos N edificios y tenemos las funciones $alto(i)$ y $largo(i)$ que nos dan el alto y largo del edificio i.

Queremos tener una función f que compute el ancho de la máxima subsecuencia ascendente.

Pensemos la función recursiva.
¿Cuál es nuestro caso base?
¿Cómo es el paso recursivo?

Una posible respuesta

Sea $alto(-1) = 0$.

$$f(i, ult) = \begin{cases} 0 & \text{si } i \geq N \\ f(i+1, ult) & \text{si } alto(ult) \geq alto(i) \\ \max\{f(i+1, ult), f(i+1, i) + largo(i)\} & \text{c.c.} \end{cases}$$

La subsecuencia creciente más larga es $f(0, -1)$.

Una posible respuesta

Sea $alto(-1) = 0$.

$$f(i, ult) = \begin{cases} 0 & \text{si } i \geq N \\ f(i+1, ult) & \text{si } alto(ult) \geq alto(i) \\ \max\{f(i+1, ult), f(i+1, i) + largo(i)\} & \text{c.c.} \end{cases}$$

La subsecuencia creciente más larga es $f(0, -1)$.

¿Cuántos llamados recursivos hace esta función?

Una posible respuesta

Sea $alto(-1) = 0$.

$$f(i, ult) = \begin{cases} 0 & \text{si } i \geq N \\ f(i+1, ult) & \text{si } alto(ult) \geq alto(i) \\ \max\{f(i+1, ult), f(i+1, i) + largo(i)\} & \text{c.c.} \end{cases}$$

La subsecuencia creciente más larga es $f(0, -1)$.

¿Cuántos llamados recursivos hace esta función?

En el peor de los casos, los edificios están todos ordenados en forma creciente y el árbol de recursión explora todos los posibles subconjuntos de edificios. Teniendo $\Omega(2^N)$ hojas. (Una bocha).

Pero... ¿Cuántos resultados distintos nos puede dar?

Estados vs llamados recursivos

Las funciones matemáticas tienen una única salida para cada entrada distinta, entonces no puede haber más respuestas que distintas formas de llamarla.

¿De cuántas maneras distintas podemos llamar a $f(i, ult)$?

Estados vs llamados recursivos

Las funciones matemáticas tienen una única salida para cada entrada distinta, entonces no puede haber más respuestas que distintas formas de llamarla.

¿De cuántas maneras distintas podemos llamar a $f(i, ult)$? N^2

Solución Dinámica

Queremos resolver todos los subproblemas ya calculados en $O(1)$, para eso usamos una estructura de memoización.

Queremos una estructura con acceso aleatorio en tiempo constante donde guardar los resultados de cada estado distinto.

Como tenemos una variable que puede tomar valores entre 0 y N, podemos usar una matriz $M \in \mathbb{N}^{N \times N}$ para guardar $M[i][ult] = f(i, ult)$.

Solución Dinámica

Ahora, cada vez que necesitemos algún $f(i, ult)$ nos podemos primero fijar si ya lo tenemos calculado en M y posiblemente usarlo en $O(1)$

Pseudocódigo sin dinámica

F(i, ult):

- Si $i \geq N$ devolver 0
- Si $alto(ult) \geq alto(i)$ devolver $F(i+1, ult)$
- devolver $\max\{F(i+1, ult),\ F(i+1, i) + largo(i)\}$

Pseudocódigo con dinámica

- Sea M una matriz en $\mathbb{N}^{N \times N}$ inicializada con valores indefinidos.

F(i, ult):

- Si $M[i][ult]$ está definido devolver $M[i][ult]$
- Si $i \geq N$ devolver 0
- Si $alto(ult) \geq alto(i)$ devolver $F(i+1, ult)$
- $M[i][ult] \leftarrow \max\{F(i+1, ult),\ F(i+1, i) + largo(i)\}$
- devolver $M[i][ult]$

Nueva complejidad

Todos los llamados a F que ya estén guardados en la matriz los consideramos iguales a simplemente leer memoria y por lo tanto despreciables.

La complejidad de un algoritmo dinámico es la cantidad de estados multiplicado por lo que cuesta resolver internamente cada estado. En nuestro caso, eso es $O(N^2) \cdot O(1) = O(N^2)$ (mucho mejor).

O vs Ω

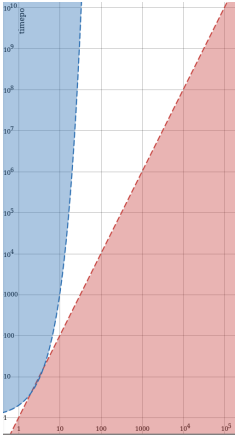


Figura: En escala logarítmica. En azul $\Omega(2^N)$, en rojo $O(N^2)$.

¿Perdimos algo a cambio de esta mejora en complejidad temporal?

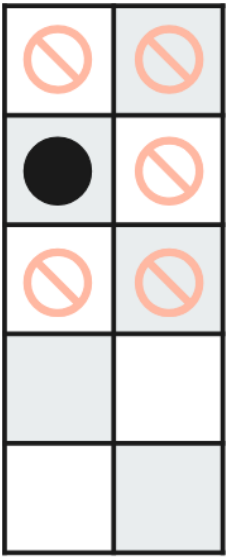
¿Perdimos algo a cambio de esta mejora en complejidad temporal? Ahora tenemos una **complejidad espacial** de $O(N^2)$.

Hay una versión con la misma complejidad temporal pero con complejidad espacial $O(N)$, queda de tarea pensarla.

DEMO

La casa de Damián

Damián tiene una casa de $2 \times N$ metros de largo. Para organizar sus famosas fiestas de *Break Dance*, dividió a su casa en cuadrados de $1m^2$ cada uno, sabiendo que en cada cuadrado solo entra una persona bailando y que, para darle espacio, no puede haber otra persona en los cuadrados adyacentes. Por curiosidad, a Damián le interesaría saber cuántas fiestas distintas puede tener en su casa.



Solución por backtracking

Queremos una función f tal que $f(n)$ sea la cantidad de formas de poner bailarines en una casa de $2 \times N$.

¿Qué decisiones puedo tomar sobre la última fila?
Recursivamente, ¿Cuántas opciones me quedan una vez que tomé cada una de esas decisiones?

Solución por backtracking

Si elegimos no poner a nadie en la última fila, tenemos tantas soluciones posibles como las habría en una casa de largo $n - 1$.

Si elegimos poner a alguien en cualquiera de las dos posiciones, ahora no podemos poner a nadie en la fila $n - 1$ y tenemos tantas opciones como en una casa de largo $n - 2$.

Solución por backtracking

¿Qué podemos usar como caso base?

Solución por backtracking

¿Qué podemos usar como caso base?

$n = 1$ 

$n = 0?$

$f(0) = 1,\ f(1) = 3$

Posible función recursiva

$$f(i) = \begin{cases} 1 & \text{si } i = 0 \\ 3 & \text{si } i = 1 \\ f(i-1) + 2f(i-2) & c.c. \end{cases}$$

Posible función recursiva

$$f(i) = \begin{cases} 1 & \text{si } i = 0 \\ 3 & \text{si } i = 1 \\ f(i-1) + 2f(i-2) & c.c. \end{cases}$$

¿Tenemos superposición de subproblemas?

- ¿Cuántos llamados recursivos hacemos?

Posible función recursiva

$$f(i) = \begin{cases} 1 & \text{si } i = 0 \\ 3 & \text{si } i = 1 \\ f(i-1) + 2f(i-2) & c.c. \end{cases}$$

¿Tenemos superposición de subproblemas?

- ¿Cuántos llamados recursivos hacemos? $\Omega(\sqrt{2}^n)$ (uf)

Posible función recursiva

$$f(i) = \begin{cases} 1 & \text{si } i = 0 \\ 3 & \text{si } i = 1 \\ f(i-1) + 2f(i-2) & c.c. \end{cases}$$

¿Tenemos superposición de subproblemas?

- ¿Cuántos llamados recursivos hacemos? $\Omega(\sqrt{2}^n)$ (uf)
- ¿Cuántos estados distintos hay?

Posible función recursiva

$$f(i) = \begin{cases} 1 & \text{si } i = 0 \\ 3 & \text{si } i = 1 \\ f(i-1) + 2f(i-2) & c.c. \end{cases}$$

¿Tenemos superposición de subproblemas?

- ¿Cuántos llamados recursivos hacemos? $\Omega(\sqrt{2}^n)$ (uf)
- ¿Cuántos estados distintos hay? $O(n)$

Posible función recursiva

$$f(i) = \begin{cases} 1 & \text{si } i = 0 \\ 3 & \text{si } i = 1 \\ f(i-1) + 2f(i-2) & c.c. \end{cases}$$

¿Tenemos superposición de subproblemas?

- ¿Cuántos llamados recursivos hacemos? $\Omega(\sqrt{2}^n)$ (uf)
- ¿Cuántos estados distintos hay? $O(n)$
- $\Omega(\sqrt{2}^n) \gg O(n) \Rightarrow$ tenemos superposición de subproblemas

Con un vector de $\mathbb{N}^n$, como un llamado recursivo toma $O(1)$, la función termina en $O(1) \times O(n) = O(n)$ con complejidad espacial $O(n)$.

Joya, terminamos... ¿No?

NO

Hay una solución mejor todavía.

Solución Bottom Up

En lugar de empezar por el tamaño de entrada que necesitamos y achicarlo hasta llegar al caso base, podemos empezar por el caso base y complejizar la entrada hasta llegar al tamaño que buscamos.

Es decir, empezamos con $i = 0$ y vamos subiendo hasta $i = n$.

Dicho de otra manera, lo estamos resolviendo en el orden en el que podemos obtener los datos.

- $M \leftarrow [1, 3, \perp, \cdots, \perp] \in \mathbb{N}^{n+1}$
- Con i desde 2 hasta n:
  - $M[i] \leftarrow M[i-1] + 2M[i-2]$
- devolver $M[n]$

Complejidad espacial: $O(n)$, complejidad temporal: $O(n)$. Es lo mismo.
¿Entonces por qué lo hicimos?

- $M \leftarrow [1, 3, \perp, \cdots, \perp] \in \mathbb{N}^{n+1}$
- Con i desde 2 hasta n:
  - $M[i] \leftarrow M[i-1] + 2M[i-2]$
- devolver $M[n]$

Cuando llegamos a la iteración I, los datos anteriores a $I - 2$ no se van a usar más, podemos descartarlos.

- $M_0 \leftarrow 1, M_1 \leftarrow 3$
- Con i desde 2 hasta N:
  - $M_i \leftarrow M_0 + 2M_1$
  - $M_0 \leftarrow M_1$
  - $M_1 \leftarrow M_i$
- devolver M_1

Hicimos el mismo cómputo con complejidad espacial $O(1)$.
Podemos hacer esto porque en Bottom Up sabemos el orden en el que se van a necesitar los datos.

¿Hacemos un break?

Ejercicio 3

Rebelión en la granja

Gio está harto de la ciudad, por lo que se fue de vacaciones a la granja de su abuelo. Su abuelo tiene un terreno de N metros de largo, y M de ancho, dividido en celdas de 1 metro cuadrado. En algunas celdas hay una cantidad arbitraria de arvejas. Gio tiene como objetivo recolectar la mayor cantidad de arvejas, respetando que:

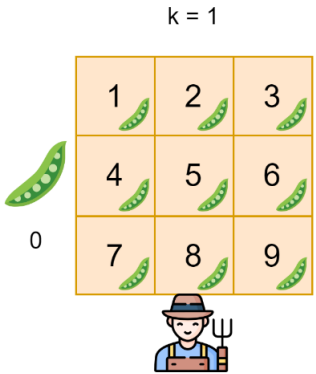
- Empieza desde alguna celda en el comienzo del terreno (y = 0).
- Se puede mover en 2 sentidos:
  - Adelante e izquierda.
  - Adelante y derecha.
- Tiene que llegar al final del terreno (y = N) con una cantidad de arvejas recolectadas divisible por $K + 1$, con K un número fijo dado.

Especificaciones

- El formato del input es una línea que contiene tres enteros n (cantidad de filas), m (cantidad de columnas) y k (número del módulo). Luego tenemos n líneas que contienen m números cada una. Siendo cada uno de estos números la cantidad de arvejas en esa celda.
- Tenemos que devolver -1 si es imposible recolectar la mayor cantidad de arvejas respetando lo pedido. Si no debemos devolver 3 líneas. En la primera, el máximo número de arvejas, divisible por $k + 1$. En la segunda, la posición desde donde debe comenzar. Por último, en la tercera, se debe mostrar la secuencia de movimientos usando I o D en base a qué movimiento tomó.

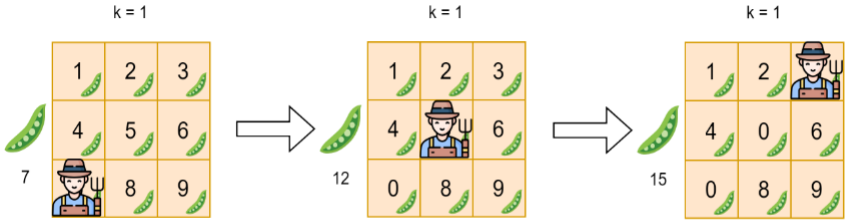
Dibujemos un ejemplo

En este ejemplo nuestros inputs son $N = 3, M = 3, K = 1$. Con esta disposición de arvejas en la granja.



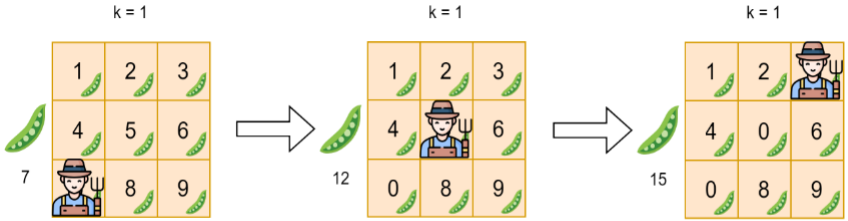
Ejemplo inválido

¿Por qué no es una solución válida?



Ejemplo inválido

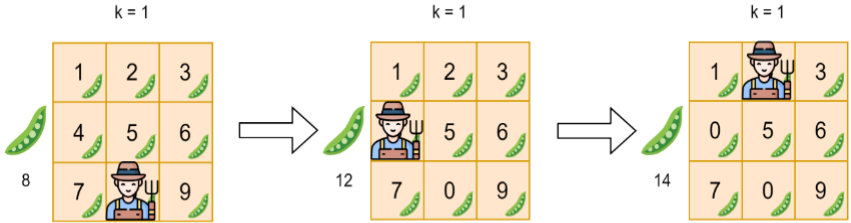
¿Por qué no es una solución válida?



Porque 15 no es divisible por 2.

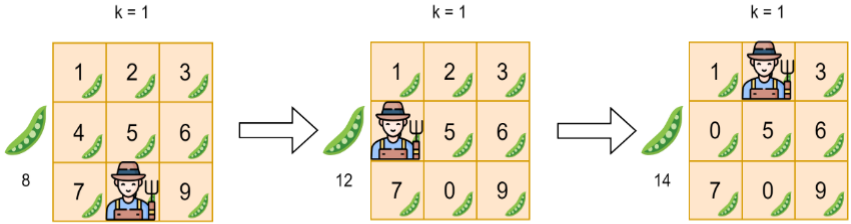
Ejemplo válido

¿Podríamos dar este ejemplo cómo respuesta?



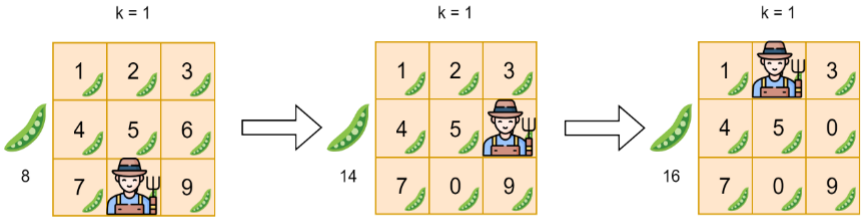
Ejemplo válido

¿Podríamos dar este ejemplo cómo respuesta?



Es una respuesta válida, porque 14 es divisible por 2, pero no es la óptima.

Ejemplo de solución



Es una solución válida, porque 16 es divisible por 2, y es la óptima.

Modelemos el estado del problema

¿Qué necesitamos saber en todo momento para poder decidir cómo calcular la máxima cantidad de arvejas que puede recolectar el granjero?

Modelemos el estado del problema

¿Qué necesitamos saber en todo momento para poder decidir cómo calcular la máxima cantidad de arvejas que puede recolectar el granjero?
Pensemos en las condiciones del problema:

- El granjero en todo momento se encuentra en una celda del terreno.
- El granjero viene acumulando una cantidad de arvejas.

Entonces... para saber qué hacer en cada momento necesitamos:

Modelemos el estado del problema

¿Qué necesitamos saber en todo momento para poder decidir cómo calcular la máxima cantidad de arvejas que puede recolectar el granjero?
Pensemos en las condiciones del problema:

- El granjero en todo momento se encuentra en una celda del terreno.
- El granjero viene acumulando una cantidad de arvejas.

Entonces... para saber qué hacer en cada momento necesitamos:

- Las coordenadas **(x, y)** del terreno.
- La cantidad de arvejas que recolectamos hasta ahora.

Firma de la función

Basado en el estado que hicimos, la función quedaría así:

Firma de la función

Basado en el estado que hicimos, la función quedaría así:

$f(x, y, arv)$ = La máxima cantidad de arvejas divisible por $k + 1$ que puede obtener Gio partiendo desde la posición x, y con arv arvejas.

Tip

Recuerden describir con palabras la firma de su función, ahí van a poder ver si necesitan algún parámetro más o si se pueden estar olvidando de algo.

¿Cómo empezamos?

Firma de la función

Basado en el estado que hicimos, la función quedaría así:

$f(x, y, arv)$ = La máxima cantidad de arvejas divisible por $k + 1$ que puede obtener Gio partiendo desde la posición x, y con arv arvejas.

Tip

Recuerden describir con palabras la firma de su función, ahí van a poder ver si necesitan algún parámetro más o si se pueden estar olvidando de algo.

¿Cómo empezamos? Hay muchas formas de hacerlo, pero en general lo más sencillo son los casos base.

Definiendo los casos base

Pensemos cuando podemos determinar el máximo de arvejas sin recurrir a la recursión.

Definiendo los casos base

Pensemos cuando podemos determinar el máximo de arvejas sin recurrir a la recursión.

Cuando llegamos arriba de todo en el terreno no podemos avanzar más...¿Cuáles son los estados base para nosotros pensando en esto?

- Llegamos a la fila n
  - ¿Importa lo que acumulamos de arvejas?

Definiendo los casos base

Pensemos cuando podemos determinar el máximo de arvejas sin recurrir a la recursión.

Cuando llegamos arriba de todo en el terreno no podemos avanzar más...¿Cuáles son los estados base para nosotros pensando en esto?

- Llegamos a la fila n
  - ¿Importa lo que acumulamos de arvejas? **Sí**
- Tenemos 2 casos entonces, la cantidad de arvejas es:
  1. Válida.
  2. Inválida.

Definiendo los casos base

Pensemos que retornar en cada caso de arvejas:

Definiendo los casos base

Pensemos que retornar en cada caso de arvejas:

1. **Válido**: Retornamos 0 porque no podemos obtener más arvejas desde acá.
2. **Inválido**: Acá casi estamos como en el caso de arriba, pero en una situación inválida.
   - Para saber que retornar, pensemos la naturaleza del problema. Buscamos *maximizar* la cantidad de arvejas.
   - Entonces... ¿Qué retornamos para no afectar la maximización si llegamos a una situación inválida?

Definiendo los casos base

Pensemos que retornar en cada caso de arvejas:

1. **Válido**: Retornamos 0 porque no podemos obtener más arvejas desde acá.
2. **Inválido**: Acá casi estamos como en el caso de arriba, pero en una situación inválida.
   - Para saber que retornar, pensemos la naturaleza del problema. Buscamos *maximizar* la cantidad de arvejas.
   - Entonces... ¿Qué retornamos para no afectar la maximización si llegamos a una situación inválida? **-INFINITO**.

Definiendo los casos base

Actualizando los casos base, la función nos quedaría así:

$$f(x, y, arv) = \begin{cases} 0 & \text{si } y = n \wedge arv \mod (k+1) = 0 \\ -\infty & \text{si } y = n \wedge arv \mod (k+1) \neq 0 \end{cases}$$

Tip

No se olviden de describir con palabras sus casos base y sus pasos recursivos, así uno puede entender qué estaban pensando.

Definiendo los casos recursivos

Ahora vamos a definir los casos recursivos ¿Cuáles son?

Definiendo los casos recursivos

Ahora vamos a definir los casos recursivos ¿Cuáles son?

- Estoy en coordenadas **(0, y)**, o sea, a la izquierda de todo.
  - Puedo únicamente moverme arriba a la derecha.
- Estoy en coordenadas **(M-1, y)**, o sea, a la derecha de todo.
  - Puedo únicamente moverme arriba a la izquierda.
- Estoy en coordenadas **(x, y)** donde $0 < x < M - 1$, o sea, en el medio.
  - Puedo moverme arriba a la izquierda o derecha.

Alternativa

¿Es la única forma de hacerlo?

Definiendo los casos recursivos

Ahora vamos a definir los casos recursivos ¿Cuáles son?

- Estoy en coordenadas **(0, y)**, o sea, a la izquierda de todo.
  - Puedo únicamente moverme arriba a la derecha.
- Estoy en coordenadas **(M-1, y)**, o sea, a la derecha de todo.
  - Puedo únicamente moverme arriba a la izquierda.
- Estoy en coordenadas **(x, y)** donde $0 < x < M - 1$, o sea, en el medio.
  - Puedo moverme arriba a la izquierda o derecha.

Alternativa

¿Es la única forma de hacerlo? ąNo! También podemos incluir en nuestro caso de $-\infty$ cuándo nos vamos del tablero, y así solo nos quedaría el tercer caso. Elegir de que forma de hacerlo es una cuestión de *estilo*.

Definiendo los casos recursivos

Actualizando los casos recursivos, la función nos quedaría así:

$$f(x, y, arv) =$$

$$\begin{cases} 0 & y = n \wedge esValido \\ -\infty & y = n \wedge \neg esValido \\ \text{terr}[x][y] + f(x+1, y+1, sigArv) & x = 0 \\ \text{terr}[x][y] + f(x-1, y+1, sigArv) & x = m-1 \\ \text{terr}[x][y] + \text{máx}(f(x-1, y+1, sigArv), f(x+1, y+1, sigArv)) & \text{sino} \end{cases}$$

$$\text{sigArv} = \text{arv} + \text{terr}[x][y]$$
$$\text{esValido} = arv \mod (k+1) = 0$$

Terminando la solución

Ya casi estamos. ¿Qué nos falta?

Terminando la solución

Ya casi estamos. ¿Qué nos falta? Correcto!

- Inicializamos nuestra estructura de memoización, en este caso podemos usar una matriz de 3 dimensiones, una para cada parámetro.
- Luego hacemos los llamados correspondientes para resolver el problema. En este caso es obtener él $i, 0 \leq i \leq M$ que maximice $(f(i, 0, 0))$. ¿Por qué?
- A partir de eso reconstruimos el resultado (lo vamos a ver más adelante) y devolvemos $f(i, 0, 0)$. ¿Es la única forma de hacerlo?

Terminando la solución

Ya casi estamos. ¿Qué nos falta? Correcto!

- Inicializamos nuestra estructura de memoización, en este caso podemos usar una matriz de 3 dimensiones, una para cada parámetro.
- Luego hacemos los llamados correspondientes para resolver el problema. En este caso es obtener él $i, 0 \leq i \leq M$ que maximice $(f(i, 0, 0))$. ¿Por qué?
- A partir de eso reconstruimos el resultado (lo vamos a ver más adelante) y devolvemos $f(i, 0, 0)$. ¿Es la única forma de hacerlo?

Otras opciones

Esta tampoco es la única forma de hacerlo. A la hora de reconstruir el camino, uno puede querer ir calculándolo a medida que resuelve el problema. Pero en este caso haría menos legible la función y no nos brindaría ninguna ventaja. Hay que ver cuándo eso es conveniente.

Entregando la solución

¿Es correcta nuestra función?

¿Va a pasar el juez?

Entregando la solución

¿Es correcta nuestra función? **Si**

¿Va a pasar el juez? **No** :(

Verdict	Time	Memory
Time limit exceeded on test 24	2000 ms	129100 KB

¿Por qué ocurre esto?

¿Por qué no pasa el juez?

Sabemos que vamos a terminar con una memoria de estas dimensiones:

- Rango de x
  - $O(M)$
- Rango de y
  - $O(N)$
- Cota máxima de arvejas (cada celda tiene a lo sumo 9)
  - $O(MN \cdot 10) = O(MN)$

En total tenemos luego $O(MN \cdot MN) = O(M^2N^2)$, lo cual es muy pesado, agrega mucha complejidad al algoritmo.

Refinemos el estado del problema

La pregunta clave acá es... ¿Necesitamos saber cuántas arvejas venimos acumulando?

Refinemos el estado del problema

La pregunta clave acá es... ¿Necesitamos saber cuántas arvejas venimos acumulando?

- ¿Cambia en algo si $k = 1$ y estoy en una coordenada **(x, y)** con 10 arvejas en un caso, y en otro estado también estoy en **(x, y)** pero tengo 8 por ejemplo?
  - No, porque ambos son divisibles por 2. Y la cantidad final de arvejas ya la acumulo en el resultado de la función.
- Entonces... me basta con saber cuánto es el módulo **(k+1)** de arvejas que vengo recolectando, ¿No?

Empecemos desde el principio reajustando la función recursiva f.

Redefiniendo la función

$f(x, y, \mathit{arvMod}) =$

$$\begin{cases} 0 & y = n \wedge \mathit{esValido} \\ -\infty & y = n \wedge \neg \mathit{esValido} \\ \text{terr}[x][y] + f(x+1, y+1, \mathit{sigArv}) & x = 0 \\ \text{terr}[x][y] + f(x-1, y+1, \mathit{sigArv}) & x = m-1 \\ \text{terr}[x][y] + \text{máx}(f(x-1, y+1, \mathit{sigArv}), f(x+1, y+1, \mathit{sigArv})) & \text{sino} \end{cases}$$

$\text{sigArv} = \mathit{arvMod} + \text{terr}[x][y] \mod (k+1)$
$\text{esValido} = \mathit{arvMod} \mod (k+1) = 0$

Ahora la firma de la función es:

- $f(x, y, \mathit{modArv})$ = La máxima cantidad de arvejas divisible por $k+1$ que puede obtener Gio partiendo desde la posición x, y *con una cantidad de modArv arvejas módulo* $k+1$.

Reconstrucción del problema

Ahora realmente ya casi estamos, solo nos falta la última parte de nuestro algoritmo.

- Inicializamos nuestra estructura de memoización, en este caso podemos usar una matriz de 3 dimensiones, una para cada parámetro.
- Luego hacemos los llamados correspondientes para resolver el problema. En este caso es obtener él $i, 0 \leq i \leq M$ que maximice $(f(i, 0, 0))$.
- A partir de eso reconstruimos el resultado (lo vamos a ver más adelante) y devolvemos $f(i, 0, 0)$

¿Cómo reconstruimos nuestra solución??

Reconstrucción del problema

1. Comenzamos desde la posición $f(i, 0, 0)$, siendo i el máximo que encontramos antes. ¿Cómo sabemos si Gio fue hacia la derecha o la izquierda?

Reconstrucción del problema

1. Comenzamos desde la posición $f(i, 0, 0)$, siendo i el máximo que encontramos antes. ¿Cómo sabemos si Gio fue hacia la derecha o la izquierda?
2. Inicializamos las variables x, y, arv en $x = i, y = 0, arv = 0$.
3. Ahora vemos cuál de las dos posiciones siguientes, $j = \{i - 1, i + 1\}$, cumple qué $f(j, y + 1, arv + \text{terr}[x][y]) = f(x, y, arv) - \text{terr}[x][y]$. Esa nos va a decir que decisión tomamos, o sea si fuimos hacia la izquierda o la derecha.
4. Luego actualizamos las variables con sus nuevos valores, $x = j, y = y + 1, arv = \text{terr}[x][y]$. Ahora repetimos este paso hasta llegar al final del camino, si en algún momento no podemos ir hacia la izquierda/derecha ya sabemos la decisión por descarte.
5. Esto lo podemos hacer, puesto que al calcular f inicialmente ya pre calculamos todos los valores que vamos a necesitar. Así que el costo es meramente $O(N)$.

Implementación en Python

```
def dp(x, y, arvMod):
    if y == n:
        if arvMod == 0:
            return 0
        return -INF
    arvMod  = (arvMod + grid[y][x]) % (k+1)
    if memoria[y][x][arvMod] != -1:
        return memoria[y][x][arvMod]

    maxArvs = -INF
    if x > 0:
        maxArvs = max(maxArvs, dp(x-1, y+1, arvMod))
    if x < m-1:
        maxArvs = max(maxArvs, dp(x+1, y+1, arvMod))

    maxArvs += grid[y][x]
    memoria[y][x][arvMod] = maxArvs
    return maxArvs

memoria = [[[-1 for arv in range(k + 2)] for x in range(m + 1)]
         for y in range(n+1)]

optimo = -1
for c in range(m):
    res = dp(c, 0, 0)
    optimo = max(optimo, res)
```

Comparando BT y PD

¿Cómo mostramos si se cumple la superposición de problemas para este caso?

1. Calculamos una cota inferior $\Omega(g(n))$ de la complejidad en el peor caso de nuestra solución usando backtracking.
2. Calculamos la complejidad $O(f(n))$ en el peor caso de nuestra solución usando programación dinámica.
3. Comparamos ambas complejidades y a partir de ellas sacamos una conclusión. Queremos ver cuándo se cumple que $f(n) << g(n)$.

Cota superior e inferior

Con estas complejidades estamos diciendo que la PD toma **a lo sumo** $a_1 \cdot f(n) + b_1$ operaciones en el peor caso, y que el BT toma **al menos** $a_2 \cdot g(n) + b_2$ operaciones en el peor caso. La PD seguro va a ser mejor que el BT cuando $a_1 \cdot f(n) + b_1 < a_2 \cdot g(n) + b_2$. Abusamos de notación diciendo que esto se cumple cuando $f(n) << g(n)$.

Complejidad de Backtracking

- Vamos a buscar una cota inferior para el peor caso.
- Nuestro peor caso es tener más filas que columnas, y que uno casi siempre pueda ir a la izquierda o la derecha. No vamos a tomar cómo peor caso cuando M=2.
- Sabemos que en el peor caso la función va a tener 2 llamados recursivos, y se van a hacer cómo mucho N llamados.
- Además sabemos que luego de un llamado recursivo en el cual solo tengas una opción, en el siguiente vas a tener 2. Así que va a haber por lo menos $N/2$ llamados en los cuales tengas 2 opciones.
- La función inicial se llama M veces, una por cada posible comienzo.
- A partir de esto llegamos a que la cota inferior para la complejidad es $\Omega(M * 2^{N/2})$.

Complejidad de Programación Dinámica

- Vamos a buscar una cota superior para el peor caso, este el mismo.
- Ahora lo que queremos calcular es la cantidad de estados posibles y cuánto cuesta computar cada estado.
- Cantidad de estados = $O(MNK)$, debido a las posibles combinaciones que tenemos. Costo de computarlo = $O(1)$, ya que todas las operaciones que se realizan en cada caso son $O(1)$.
- Por lo que la complejidad total de nuestra solución de *PD* es $O(\textit{Inicializar matriz}) + O(\textit{Calcular todos los estados}) + O(\textit{Hacer los M llamados}) = O(MNK) + O(MNK\dot{1}) + O(M) = O(MNK)$. Además, vamos a usar el mismo caso para ambas soluciones.

Cómo casi siempre, se puede mejorar

Complejidad de Programación Dinámica

- Vamos a buscar una cota superior para el peor caso, este el mismo.
- Ahora lo que queremos calcular es la cantidad de estados posibles y cuánto cuesta computar cada estado.
- Cantidad de estados = $O(MNK)$, debido a las posibles combinaciones que tenemos. Costo de computarlo = $O(1)$, ya que todas las operaciones que se realizan en cada caso son $O(1)$.
- Por lo que la complejidad total de nuestra solución de *PD* es $O(\textit{Inicializar matriz}) + O(\textit{Calcular todos los estados}) + O(\textit{Hacer los M llamados}) = O(MNK) + O(MNK\dot{1}) + O(M) = O(MNK)$. Además, vamos a usar el mismo caso para ambas soluciones.

Cómo casi siempre, se puede mejorar

Hay una mejor solución, en la cual se puede usar solo $O(MK)$ de memoria si solo vamos guardando las últimas 2 filas en memoria. Pero aun así, la complejidad temporal no cambia.

Comparación final

- A partir de esto llegamos a la inecuación $O(MNK) << \Omega(M * 2^{N/2})$.
- Para hacer la comparación no vamos a utilizar la notación de complejidad, y solo vamos a ver cuándo se cumple qué $K < 2^{N/2}/N$.
- ¿Siempre se cumple esto?

Comparación final

- A partir de esto llegamos a la inecuación $O(MNK) << \Omega(M * 2^{N/2})$.
- Para hacer la comparación no vamos a utilizar la notación de complejidad, y solo vamos a ver cuándo se cumple qué $K < 2^{N/2}/N$.
- ¿Siempre se cumple esto? No necesariamente, pero cuándo esto se cumple, entonces tenemos superposición de subproblemas.
- Para nuestro problema puntual, sacado de CodeForces, hay cotas para N, M y K. $2 \leq n, m \leq 100, 0 \leq k \leq 10$.
- Por lo que se cumple qué $10 < 2^{100}/100$.
- Pero si no tenemos ninguna cota para los posibles valores de nuestros parámetros, entonces con la inecuación que conseguimos arriba alcanza para determinar si hay o no superposición (con los parámetros instanciados).

Ejercicio 4

Artículos de biblioteca

Hay un conjunto de n artículos académicos, y una biblioteca está interesada en adquirir algunos de ellos para enriquecer su colección. Cada artículo puede comprarse de forma individual o bien adquiriendo el volumen al que pertenece (en cuyo caso también se adquieren otros artículos).

Cada artículo $1 \leq i \leq n$ tiene un costo c_i y un valor v_i, y aparte pertenece a algún volumen e_i, con $1 \leq e_i \leq m$ (es decir, hay m volúmenes). Cada volumen $1 \leq j \leq m$ tiene a su vez un costo p_j.

Dado un cierto capital Q la biblioteca quiere saber cuál es la mayor cantidad de valor que puede obtener adquiriendo volúmenes y/o artículos.

Planteando la recursión

- ¿De qué forma podemos reducir este problema a uno similar pero más chico? ¿Qué significa más chico?

Planteando la recursión

- ¿De qué forma podemos reducir este problema a uno similar pero más chico? ¿Qué significa más chico?
- Dado un cierto artículo, hay dos opciones: lo compramos, o no. Nuestra recursión puede ser en la cantidad de artículos restantes, con el capital que nos queda.

Planteando la recursión

- ¿De qué forma podemos reducir este problema a uno similar pero más chico? ¿Qué significa más chico?
- Dado un cierto artículo, hay dos opciones: lo compramos, o no. Nuestra recursión puede ser en la cantidad de artículos restantes, con el capital que nos queda.
- Pero falta el tema de los volúmenes... Dado un artículo hay 3 opciones en realidad: lo compramos individualmente, lo compramos con su volumen o no lo compramos.

Planteando la recursión

- ¿De qué forma podemos reducir este problema a uno similar pero más chico? ¿Qué significa más chico?
- Dado un cierto artículo, hay dos opciones: lo compramos, o no. Nuestra recursión puede ser en la cantidad de artículos restantes, con el capital que nos queda.
- Pero falta el tema de los volúmenes... Dado un artículo hay 3 opciones en realidad: lo compramos individualmente, lo compramos con su volumen o no lo compramos.
- Pero ojo que lo podemos comprar individualmente si no compramos el volumen, y análogamente lo podemos comprar en conjunto solo si no lo compramos individualmente.

Planteando la recursión

- Podemos plantear una recursión ordenada, que vaya por volumen.

Planteando la recursión

- Podemos plantear una recursión ordenada, que vaya por volumen.
- A grandes rasgos sería:
  1. Por cada volumen, decido si comprarlo o no.
  2. Si no lo compre, tengo que decidir por cada libro individual.

Planteando la recursión

- Podemos plantear una recursión ordenada, que vaya por volumen.
- A grandes rasgos sería:
  1. Por cada volumen, decido si comprarlo o no.
  2. Si no lo compre, tengo que decidir por cada libro individual.
- Podemos escribir una función recursiva compacta que implemente esta idea si ordenamos los artículos por volumen.

Función recursiva

$$val(i,q) = \begin{cases} -\infty & q < 0 \\ 0 & i = n \\ \max(val(i+1,q), val(i+1,q-c_i) + v_i) & e_i = e_{i-1} \\ \max(val(i+1,q), val(i+1,q-c_i) + v_i, & \\ \qquad val(fin_{e_i}, q - p_{e_i}) + v'_{e_i}) & \end{cases}$$

donde v'_e indica el valor del volumen v' y fin_e el índice a partir del cual empieza el volumen siguiente a e. La condición $e_i = e_{i-1}$ me indica que no estoy al principio de un volumen (si $i = 0$ debería devolver true para que esta función ande).

Complejidad

- ¿Cuál es la cantidad de llamados que se hacen al llamar a la función como $val(0, Q)$?

Complejidad

- ¿Cuál es la cantidad de llamados que se hacen al llamar a la función como $val(0, Q)$? $O(3^n)$ es una cota gruesa, pero seguro encontramos casos donde el árbol tiene al menos $O(2^n)$ nodos (cuando Q es muy grande).

Complejidad

- ¿Cuál es la cantidad de llamados que se hacen al llamar a la función como $val(0, Q)$? $O(3^n)$ es una cota gruesa, pero seguro encontramos casos donde el árbol tiene al menos $O(2^n)$ nodos (cuando Q es muy grande).
- ¿Cantidad de estados distintos de la recursión?

Complejidad

- ¿Cuál es la cantidad de llamados que se hacen al llamar a la función como $val(0, Q)$? $O(3^n)$ es una cota gruesa, pero seguro encontramos casos donde el árbol tiene al menos $O(2^n)$ nodos (cuando Q es muy grande).
- ¿Cantidad de estados distintos de la recursión? $O(nq)$

Complejidad

- ¿Cuál es la cantidad de llamados que se hacen al llamar a la función como $val(0, Q)$? $O(3^n)$ es una cota gruesa, pero seguro encontramos casos donde el árbol tiene al menos $O(2^n)$ nodos (cuando Q es muy grande).
- ¿Cantidad de estados distintos de la recursión? $O(nq)$
- ¿Costo por estado?

Complejidad

- ¿Cuál es la cantidad de llamados que se hacen al llamar a la función como $val(0, Q)$? $O(3^n)$ es una cota gruesa, pero seguro encontramos casos donde el árbol tiene al menos $O(2^n)$ nodos (cuando Q es muy grande).
- ¿Cantidad de estados distintos de la recursión? $O(nq)$
- ¿Costo por estado? Depende la implementación. ¿Qué tan rápido se puede hacer?

Complejidad

- ¿Cuál es la cantidad de llamados que se hacen al llamar a la función como $val(0, Q)$? $O(3^n)$ es una cota gruesa, pero seguro encontramos casos donde el árbol tiene al menos $O(2^n)$ nodos (cuando Q es muy grande).
- ¿Cantidad de estados distintos de la recursión? $O(nq)$
- ¿Costo por estado? Depende la implementación. ¿Qué tan rápido se puede hacer? $O(1)$ precalculando el costo de cada volumen y los índices.

Complejidad

- ¿Cuál es la cantidad de llamados que se hacen al llamar a la función como $val(0, Q)$? $O(3^n)$ es una cota gruesa, pero seguro encontramos casos donde el árbol tiene al menos $O(2^n)$ nodos (cuando Q es muy grande).
- ¿Cantidad de estados distintos de la recursión? $O(nq)$
- ¿Costo por estado? Depende la implementación. ¿Qué tan rápido se puede hacer? $O(1)$ precalculando el costo de cada volumen y los índices.
- También hay que ordenar en función del volumen...

Complejidad

- ¿Cuál es la cantidad de llamados que se hacen al llamar a la función como $val(0, Q)$? $O(3^n)$ es una cota gruesa, pero seguro encontramos casos donde el árbol tiene al menos $O(2^n)$ nodos (cuando Q es muy grande).
- ¿Cantidad de estados distintos de la recursión? $O(nq)$
- ¿Costo por estado? Depende la implementación. ¿Qué tan rápido se puede hacer? $O(1)$ precalculando el costo de cada volumen y los índices.
- También hay que ordenar en función del volumen... Esto se hace en $O(m)$ con selection sort.

Comparando complejidades

- Las complejidades en peor caso son $O(3^n)$ contra $O(nq)$. ¿Cuándo es mejor la implementación de la función recursiva con dinámica?

Comparando complejidades

- Las complejidades en peor caso son $O(3^n)$ contra $O(nq)$. ¿Cuándo es mejor la implementación de la función recursiva con dinámica?
- Tiene que valer que $q << 3^n$. Caso contrario, ambas implementaciones son iguales (salvo en memoria, ¿no?).

Bonus track

Matrix chaining

Cuando se multiplican dos matrices $A \in \mathbb{R}^{n \times d}$ y $B \in \mathbb{R}^{d \times m}$ con el algoritmo tradicional se obtiene la matriz $AB \in \mathbb{R}^{n \times m}$ luego de realizar ndm productos. Supongamos que tenemos una serie de matrices $A_1, \dots A_k$ y queremos calcular

$$A_1 \dots A_k = \prod_{i=1}^{n} A_i$$

donde $A_i \in \mathbb{R}^{n_i \times m_i}$ (naturalmente, vale que $m_i = n_{i+1}$). Como el producto de matrices es asociativo, esta operación se puede hacer de muchas formas.

Bonus track

Matrix chaining

Por ejemplo, si $A_1 \in \mathbb{R}^{2\times 3}$, $A_2 \in \mathbb{R}^{3\times 5}$ y $A_3 \in \mathbb{R}^{5\times 1}$ el producto se puede calcular como

$$(A_1 A_2)A_3$$

o bien como

$$A_1(A_2 A_3)$$

De la primera forma se hacen 40 multiplicaciones, mientras que de la segunda solo 21. Dada la lista de dimensiones $\{(n_i, m_i)\}_{1\leq i\leq k}$ debemos encontrar el mejor orden para realizar las multiplicaciones.

Función recursiva

- Como siempre, partimos de una función recursiva. ¿Hay una primera decisión que podríamos hacer?

Función recursiva

- Como siempre, partimos de una función recursiva. ¿Hay una primera decisión que podríamos hacer?
- Observar que tenemos que elegir algo así como un **punto de corte**: un índice *i* tal que las matrices de la izquierda de *i* se van a multiplicar entre sí antes de multiplicarse con la de las de la derecha.

Función recursiva

- Como siempre, partimos de una función recursiva. ¿Hay una primera decisión que podríamos hacer?
- Observar que tenemos que elegir algo así como un **punto de corte**: un índice *i* tal que las matrices de la izquierda de *i* se van a multiplicar entre sí antes de multiplicarse con la de las de la derecha.
- Una vez que elegimos ese punto de corte, quedan dos problemas disjuntos... ąRecursión!

Función recursiva

- Como siempre, partimos de una función recursiva. ¿Hay una primera decisión que podríamos hacer?
- Observar que tenemos que elegir algo así como un **punto de corte**: un índice *i* tal que las matrices de la izquierda de *i* se van a multiplicar entre sí antes de multiplicarse con la de las de la derecha.
- Una vez que elegimos ese punto de corte, quedan dos problemas disjuntos... ąRecursión!
- ¿Cuántos puntos de corte hay?

Función recursiva

- Como siempre, partimos de una función recursiva. ¿Hay una primera decisión que podríamos hacer?
- Observar que tenemos que elegir algo así como un **punto de corte**: un índice i tal que las matrices de la izquierda de i se van a multiplicar entre sí antes de multiplicarse con la de las de la derecha.
- Una vez que elegimos ese punto de corte, quedan dos problemas disjuntos... ąRecursión!
- ¿Cuántos puntos de corte hay? $n - 1$.
- ¿Cuál sería el estado en cada paso?

Función recursiva

- Como siempre, partimos de una función recursiva. ¿Hay una primera decisión que podríamos hacer?
- Observar que tenemos que elegir algo así como un **punto de corte**: un índice i tal que las matrices de la izquierda de i se van a multiplicar entre sí antes de multiplicarse con la de las de la derecha.
- Una vez que elegimos ese punto de corte, quedan dos problemas disjuntos... ąRecursión!
- ¿Cuántos puntos de corte hay? $n - 1$.
- ¿Cuál sería el estado en cada paso? El subconjunto de matrices sobre el que estamos trabajando, que es un intervalo continuo.

Función recursiva

$$mult(i,j) = \begin{cases} 0 & i+1=j \\ \min_{i+1 \leq k < j}(mult(i,k) + mult(k,j) + n_i n_k m_j) \end{cases}$$

El análisis de la complejidad lo pueden encontrar en el Cormen. Veamos el código con reconstrucción.

Tips

- Resolver ejemplos pequeños a mano puede ayudar a ganar una intuición de que es lo que buscamos plasmar en la función.
- Clave poder explicar la firma de su función en palabras, utilizando cada parámetro para la explicación.
- Piensen con cuidado cuáles pueden ser sus casos borde y base, para no olvidarse ninguno.
- Recuerden de utilizar el peor caso para la complejidad de dinámica y backtracking.
- Para su solución bottom up, fíjense que valores tienen que tener pre calculados para saber cómo tienen que ir llenando la matriz de memoización.
- Hagan los ejercicios de las guías y consulten, aprovechen los espacios de clase.

Referencias

- Ejercicio 1 : https://vjudge.net/problem/UVA-11790
- Ejercicio 3 : https://codeforces.com/problemset/problem/41/D
- Ejercicio bonus: Ejercicio mega picante