Programación dinámica

Top Down

Joaquín Laks - Ezequiel Companeetz

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

2^{do} Cuatrimestre de 2025

El retorno del rey

El rey Cambyses está interesado en armar ejércitos en una serie de días consecutivos.

Mas aún, le interesa que el número de personas de su ejército en el dia d_i sea equivalente a la suma del número de personas del ejército que formó el dia $i - 1$ e $i - 2$.

La excepción para esto es en el día 0 y 1, en cuyo caso la cantidad de personas en esos día va a ser siempre 1.

Para él es muy complicado determinar este número, entonces nos pidió que lo ayudemos.

Dado un día N, tenemos que devolver el número de personas de su ejército.

Función recursiva

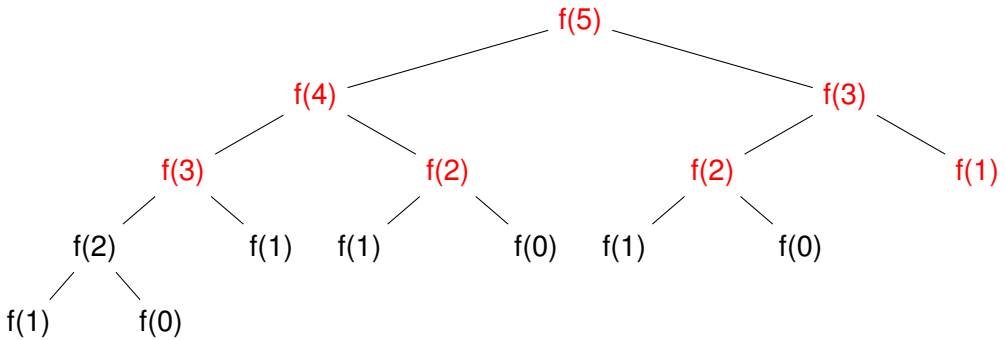
$f(i) =$ cantidad de soldados el día i.

Función recursiva

$$f(i) = \begin{cases} 1 & \text{si } i \leq 1 \\ f(i-1) + f(i-2) & \text{cc} \end{cases}$$

¿La conocen? ¿Cuántos llamados recursivos hace?

Llamados recursivos



Hay un subárbol binario completo que tiene la altura de la rama más corta ($\lfloor \frac{n}{2} \rfloor$).
Entonces el árbol tiene $> 2^{\lfloor \frac{n}{2} \rfloor}$ nodos. ($\Omega(2^{\frac{n}{2}})$).

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 $f(i)$?

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 $f(i)$? $O(n)$

O vs Ω

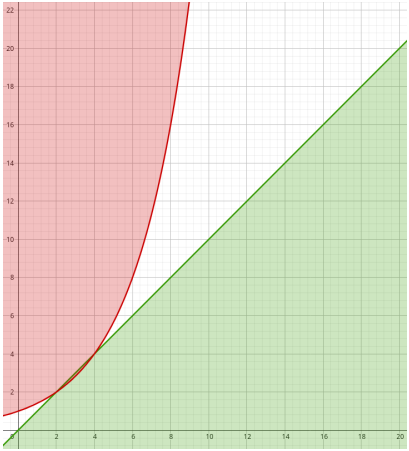


Figura: En rojo $\Omega(2^{\frac{n}{2}})$, en verde $O(N)$.

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 $M \in \mathbb{N}^n$ para guardar $M[i] = f(i)$.

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):

- Si $i \leq 1$ devolver 1
- Si no, devolver F(i-1) + F(i-2)

Pseudocódigo con dinámica

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

F(i):

- Si $M[i]$ está definido devolver $M[i]$
- Si $i \leq 1$ devolver 1
- $M[i] \leftarrow F(i-1) + F(i-2)$
- devolver $M[i]$

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 $O(n) \cdot O(1) = O(n)$ (mucho mejor).

¿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 $O(n)$.

Astro Trade

Astro Trade

Lu se dedica a la compra de asteroides. Sea $p \in \mathbb{N}^n$ tal que p_i es el precio de un asteroide el i-ésimo día en una secuencia de n días. Lu quiere comprar y vender asteroides durante esos n días de manera tal de obtener la mayor ganancia neta posible. Debido a las dificultades que existen en el transporte y almacenamiento de asteroides, Lu puede comprar a lo sumo un asteroide cada día, puede vender a lo sumo un asteroide cada día y comienza sin asteroides. Además, el Ente Regulador Asteroidal impide que Lu venda un asteroide que no haya comprado. Queremos encontrar la máxima ganancia neta que puede obtener Lu respetando las restricciones indicadas. Por ejemplo, si $p = (3, 2, 5, 6)$ el resultado es 6 y si $p = (3, 6, 10)$ el resultado es 7. Notar que en una solución óptima, Lu debe terminar sin asteroides.

Astro Trade - Incisos

1. Encontrar los casos base y los pasos recursivos para calcular la máxima ganancia neta (m.g.n.) que puede obtener Lu
2. Escribir matemáticamente la formulación recursiva enunciada en el primer punto. Dar los valores de los casos base en función de la restricción de que comienza sin asteroides.
3. Indicar qué dato es la respuesta al problema con esa formulación recursiva.
4. Diseñar un algoritmo de PD *top-down* que resuelva el problema. Explicar su complejidad temporal y espacial auxiliar. Decidir si se cumple la propiedad de superposición de subproblemas.
5. Formalmente, el problema consiste en determinar el máximo $g = \sum_{i=1}^{n} x_i p_i$ para un vector $x = (x_1, \ldots, x_n)$ tal que: $x_i \in \{-1, 0, 1\}$ para todo $1 \leq i \leq n$ y $\sum_{i=1}^{j} x_i \leq 0$ para todo $1 \leq j \leq n$. Demostrar que la formulación recursiva es correcta.

Casos base y pasos recursivos - I

- ¿Que necesitamos para definir los casos de nuestra función?

Casos base y pasos recursivos - I

- ¿Que necesitamos para definir los casos de nuestra función? Los parámetros de la función, que en este caso van a ser *a* asteroides y *d* días.

Casos base y pasos recursivos - I

- ¿Que necesitamos para definir los casos de nuestra función? Los parámetros de la función, que en este caso van a ser *a* asteroides y *d* días.
- Ahora definamos la semántica de nuestra función:

Casos base y pasos recursivos - I

- ¿Que necesitamos para definir los casos de nuestra función? Los parámetros de la función, que en este caso van a ser *a* asteroides y *d* días.
- Ahora definamos la semántica de nuestra función:
  - $mgn(a, d) =$ la máxima ganancia neta que puede obtener Lu si tiene *a* asteroides al final del día *d*.
  - Con esta semántica, ¿cuál va a ser el llamado que resuelva nuestro problema?

Casos base y pasos recursivos - I

- ¿Que necesitamos para definir los casos de nuestra función? Los parámetros de la función, que en este caso van a ser a asteroides y d días.
- Ahora definamos la semántica de nuestra función:
  - $mgn(a, d) =$ la máxima ganancia neta que puede obtener Lu si tiene a asteroides al final del día d.
  - Con esta semántica, ¿cuál va a ser el llamado que resuelva nuestro problema? $mgn(0, |p|)$

Casos base y pasos recursivos - II

- ¿Cuáles son nuestros casos base?

Casos base y pasos recursivos - II

- ¿Cuáles son nuestros casos base?
  - Si tenemos más asteroides que días, no vamos a poder vender todos, por lo que es una solución inválida.
  - Si vendimos un asteroide cuándo no teníamos ninguno rompemos la restricción, por lo que es una solución inválida.
  - Si ya no quedan días, nuestra m.g.n es 0.
- ¿Cuál es nuestro paso recursivo en el día d con a asteroides?

Casos base y pasos recursivos - II

- ¿Cuáles son nuestros casos base?
  - Si tenemos más asteroides que días, no vamos a poder vender todos, por lo que es una solución inválida.
  - Si vendimos un asteroide cuándo no teníamos ninguno rompemos la restricción, por lo que es una solución inválida.
  - Si ya no quedan días, nuestra m.g.n es 0.
- ¿Cuál es nuestro paso recursivo en el día d con a asteroides? El máximo entre:
  - la m.g.n. de finalizar el día $d-1$ con $a-1$ asteroides y comprar uno en el día d,
  - la m.g.n. de finalizar el día $d-1$ con $a+1$ asteroides y vender uno en el día d,
  - la m.g.n. de finalizar el día $d-1$ con a asteroides y no operar el día d.

Función recursiva

- ¿Que valor representa el inválido en nuestro problema?

Función recursiva

- ¿Que valor representa el inválido en nuestro problema? Cómo estamos buscando maximizar un valor, si devolvemos $-\infty$ este valor nunca será seleccionado.
- Ahora lo que vamos a hacer es pasar cada caso/paso previamente escrito a una función. ¿Cómo lo harían?

Función recursiva

- ¿Que valor representa el inválido en nuestro problema? Cómo estamos buscando maximizar un valor, si devolvemos $-\infty$ este valor nunca será seleccionado.
- Ahora lo que vamos a hacer es pasar cada caso/paso previamente escrito a una función. ¿Cómo lo harían?

$$mgn(a,d) = \begin{cases} -\infty & \text{si } a < 0 \\ -\infty & \text{si } a > d \\ -\infty & \text{si } d = 0 \text{ y } a < 0 \\ 0 & \text{si } d = 0 \\ \max(mgn(a-1,d-1) - p[d], & \text{si no} \\ \quad \max(mgn(a+1,d-1) + p[d], & \\ \quad mgn(a,d-1))) & \end{cases}$$

Función recursiva

- ¿Que valor representa el inválido en nuestro problema? Cómo estamos buscando maximizar un valor, si devolvemos $-\infty$ este valor nunca será seleccionado.
- Ahora lo que vamos a hacer es pasar cada caso/paso previamente escrito a una función. ¿Cómo lo harían?

$$mgn(a,d) = \begin{cases} -\infty & \text{si } a < 0 \\ -\infty & \text{si } a > d \\ -\infty & \text{si } d = 0 \text{ y } a < 0 \\ 0 & \text{si } d = 0 \\ \text{máx}(mgn(a-1,d-1) - p[d], & \text{si no} \\ \quad \text{máx}(mgn(a+1,d-1) + p[d], & \\ \quad mgn(a,d-1))) & \end{cases}$$

- ¿Se puede hacer un poco más concisa?

Función recursiva simplificada

$$mgn(a,d) = \begin{cases} -\infty & \text{si } a < 0 \text{ o } a > d \\ 0 & \text{si } d = 0 \\ \max(mgn(a-1,d-1) - p[d], & \text{si no} \\ \quad mgn(a+1,d-1) + p[d], & \\ \quad mgn(a,d-1)) & \end{cases}$$

Solución del problema

Solución del problema

- En este caso la llamada que resuelve el problema es $mgn(0, |p|)$, ya que:
  - $|p|$ son todos los días en los cuáles podemos intercambiar asteroides
  - La semántica de la función es: $mgn(a, d) =$ la máxima ganancia neta que puede obtener Lu si tiene a asteroides al final del día d.
- Por lo tanto $mgn(0, |p|)$ es la máxima ganancia neta que puede obtener Lu.

Solución Backtracking

Pseudocódigo sin dinámica

Función mgn(asteroid, day):

- Si $asteroid < 0$ o $asteroid > day$ devolver $-\infty$
- Si $day = 0$ devolver 0
- Sea $p \leftarrow prices[day]$
- $ans \leftarrow \max(mgn(asteroid, day - 1,\ mgn(asteroid - 1, day - 1) - p\, mgn(asteroid + 1, day - 1) + p)$
- devolver *ans*

¿Cómo la paso a PD?

- Igual que antes, la estructura de memoización ("memoria") la construyo en base a los estados que tengo.

¿Cómo la paso a PD?

- Igual que antes, la estructura de memoización ("memoria") la construyo en base a los estados que tengo.
- Para nuestros casos suele ser una matriz la memoria, pero no siempre es así.

¿Cómo la paso a PD?

- Igual que antes, la estructura de memoización ("memoria") la construyo en base a los estados que tengo.
- Para nuestros casos suele ser una matriz la memoria, pero no siempre es así.
- Los pasos a seguir son:
  - Inicializar la memoria.
  - Agregar un chequeo al principio de la función para saber si ya resolvimos ese llamado recursivo.
  - Antes de retornar el resultado, guardamos lo calculado en la posición correspondiente de la memoria.
-

Solución PD

Pseudocódigo con dinámica

- Sea M una matriz de tamaño $(|p|+1) \times (|p|+1)$ inicializada en $-\infty$.

Función mgn(asteroid, day):

- Si $asteroid < 0$ o $asteroid > day$ devolver $-\infty$
- Si $day = 0$ devolver 0
- Si $M[day][asteroid]$ está definido devolver $M[day][asteroid]$
- Sea $p \leftarrow prices[day]$
- $ans \leftarrow \max(mgn(asteroid, day-1,\ mgn(asteroid-1, day-1) - p\, mgn(asteroid+1, day-1) + p)$
- $M[day][asteroid] \leftarrow ans$
- devolver ans

Función recursiva `mgn`

```
1  // Includes, global variables and constants declaration
2  int mgn(int asteroid, int day) {
3      if (asteroid < 0 || asteroid > day) return NEG;
4      if (day == 0) return 0;
5
6      int &memo = gain_per_day_and_asteroid[day][asteroid];
7      if (memo != NEG) return memo;
8
9      int ans = mgn(asteroid, day - 1);
10     ans = max(ans, mgn(asteroid - 1, day - 1) - prices[day]);
11     ans = max(ans, mgn(asteroid + 1, day - 1) + prices[day]);
12
13     memo = ans;
14     return memo;
15 }
```

Función `main`

```
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    if (!(cin >> n)) return 0;

    prices.assign(n + 1, 0);
    for (int i = 1; i <= n; ++i) cin >> prices[i];

    gain_per_day_and_asteroid.assign(n + 1,
        vector<int>(n + 1, NEG));

    cout << mgn(0, n) << "\n";
    return 0;
}
```

Complejidades

Complejidades

- Complejidad espacial: utilizamos una matriz de tamaño $O(|p|^2)$ la cuál tiene datos de tamaño $O(1)$ en cada posición, por lo que la complejidad espacial es $O(|p|^2)$
- Complejidad temporal:

Complejidades

- Complejidad espacial: utilizamos una matriz de tamaño $O(|p|^2)$ la cuál tiene datos de tamaño $O(1)$ en cada posición, por lo que la complejidad espacial es $O(|p|^2)$
- Complejidad temporal:
  - Las operaciones dentro de cada llamado cuestan $O(1)$, pues son todas operaciones elementales
  - Backtracking: En base al árbol de llamados recursivos podemos decir que la cantidad de llamadas que hace la función esta acotada superiormente por $O(3^{|p|})$, así que la complejidad temporal es $O(3^{|p|})$.
  - Dinámica: La cantidad de estados esta acotada superiormente por $O(|p|^2)$, así que la complejidad temporal es $O(|p|^2)$.

Complejidades

- Complejidad espacial: utilizamos una matriz de tamaño $O(|p|^2)$ la cuál tiene datos de tamaño $O(1)$ en cada posición, por lo que la complejidad espacial es $O(|p|^2)$
- Complejidad temporal:
  - Las operaciones dentro de cada llamado cuestan $O(1)$, pues son todas operaciones elementales
  - Backtracking: En base al árbol de llamados recursivos podemos decir que la cantidad de llamadas que hace la función esta acotada superiormente por $O(3^{|p|})$, así que la complejidad temporal es $O(3^{|p|})$.
  - Dinámica: La cantidad de estados esta acotada superiormente por $O(|p|^2)$, así que la complejidad temporal es $O(|p|^2)$.
- ¿Se puede tener una complejidad espacial auxiliar menor estricta a $O(n^2)$? Lo veremos en la clase de Bottom Up

Superposición de subproblemas

- Un error común es "probar"la superposición de subproblemas mostrando un caso puntual, lo que esperamos es que muestren cuándo se cumple que #estados « #subproblemas.

Superposición de subproblemas

- Un error común es "probar"la superposición de subproblemas mostrando un caso puntual, lo que esperamos es que muestren cuándo se cumple que #estados « #subproblemas.
- En este caso podemos acotar ambas con:

Superposición de subproblemas

- Un error común es "probar"la superposición de subproblemas mostrando un caso puntual, lo que esperamos es que muestren cuándo se cumple que #estados « #subproblemas.
- En este caso podemos acotar ambas con:
  - Subproblemas: En base al árbol de llamados recursivos podemos decir que la cantidad de llamadas que hace la función esta acotada inferiormente por $\Omega(2^{|p|})$.
  - Dinámica: La cantidad de estados esta acotada superiormente por $O(|p|^2)$.
- Por lo que se cumple la superposición de subproblemas cuándo $p^2 << 2^p$

Demostración - Consigna

Enunciado

Formalmente, el problema consiste en determinar el máximo $g = \sum_{i=1}^{n} x_i p_i$ para un vector $x = (x_1, \ldots, x_n)$ tal que: $x_i \in \{-1, 0, 1\}$ para todo $1 \leq i \leq n$ y $\sum_{i=1}^{j} x_i \leq 0$ para todo $1 \leq j \leq n$. Demostrar que la formulación recursiva es correcta.

Solución óptima - Asteroides finales

- Queremos demostrar que existe una solución óptima en la que Astro Void se queda sin asteroides al final del día n.

Solución óptima - Asteroides finales

- Queremos demostrar que existe una solución óptima en la que Astro Void se queda sin asteroides al final del día n.
- Sea o una solución óptima cualquiera, con $k = -\sum_{x_i \in o} x_i$ la cantidad de asteroides de Lu al finalizar el día n.

Solución óptima - Asteroides finales

- Queremos demostrar que existe una solución óptima en la que Astro Void se queda sin asteroides al final del día n.
- Sea o una solución óptima cualquiera, con $k = -\sum_{x_i \in o} x_i$ la cantidad de asteroides de Lu al finalizar el día n.
  - Si $k = 0$, entonces la solución óptima tiene 0 asteroides.

Solución óptima - Asteroides finales

- Queremos demostrar que existe una solución óptima en la que Astro Void se queda sin asteroides al final del día n.
- Sea o una solución óptima cualquiera, con $k = -\sum_{x_i \in o} x_i$ la cantidad de asteroides de Lu al finalizar el día n.
  - Si $k = 0$, entonces la solución óptima tiene 0 asteroides.
  - Si $k < 0$, no se cumple la restricción de vender lo que no tenes, por lo que no es una solución válida, así que k no puede ser menor a 0.

Solución óptima - Asteroides finales

- Queremos demostrar que existe una solución óptima en la que Astro Void se queda sin asteroides al final del día n.
- Sea o una solución óptima cualquiera, con $k = -\sum_{x_i \in o} x_i$ la cantidad de asteroides de Lu al finalizar el día n.
  - Si $k = 0$, entonces la solución óptima tiene 0 asteroides.
  - Si $k < 0$, no se cumple la restricción de vender lo que no tenes, por lo que no es una solución válida, así que k no puede ser menor a 0.
  - Si $k > 0$, eso significa que nos quedó un asteroide por vender. Sea i el último día que Lu compró un asteroide, podemos construir una solución o' en la cuál no compró ese día. Luego la ganancia g de o' es $g(o') = g(o) + p_i$, ya que que ese día no compró el asteroide. Por lo tanto $g(o') > g(o)$, ABS!, pues o es óptima. Por lo que k no puede ser mayor a 0.

Corrección de la recurrencia por inducción (I)

- **Proposición:** Para todo $d \in \{0, \ldots, n\}$ y todo a, la función $mgn(a, d)$ devuelve la máxima ganancia neta alcanzable al final del día d poseyendo exactamente a asteroides. Vamos a realizar una inducción en d, la cantidad de días.
- **Caso base ($d = 0$):**
  - El único estado posible es $a = 0$ (por lo probado anteriormente), con ganancia 0.
  - Si $a \neq 0$, el estado es imposible y se define como $-\infty$.
  - Esto coincide con la definición del problema: al inicio no se poseen asteroides ni se han hecho transacciones.
- **Hipótesis inductiva:** Supongamos que la proposición es válida para $d - 1$.

Corrección de la recurrencia por inducción (II)

- **Paso inductivo (d):** Consideremos el día d y un número de asteroides a. Toda secuencia factible que lleva al estado (a, d) proviene de un estado factible en el día $d-1$ mediante exactamente una de estas acciones:
  - **No hacer nada:** estado previo $(a, d-1)$, ganancia $mgn(a, d-1)$.
  - **Comprar un asteroide:** estado previo $(a-1, d-1)$, ganancia $mgn(a-1, d-1) - p_d$.
  - **Vender un asteroide:** estado previo $(a+1, d-1)$, ganancia $mgn(a+1, d-1) + p_d$.
- Por hipótesis inductiva, cada término anterior ya representa la ganancia máxima posible en el estado previo correspondiente.
- Si $a < 0$ o $a > d$ caemos en un caso base y no hay que probar nada, pues no sería una solución válida u óptima (ya que no termina con 0 asteroides).
- Luego, tomar el máximo de los tres casos equivale a seleccionar la mejor acción en el día d.
- **Conclusión:** $mgn(a, d)$ computa correctamente la máxima ganancia neta alcanzable al final del día d con a asteroides.

Ejercicio 3

Rebelión en la granja

Tobi 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. Tobi tiene como objetivo recolectar la mayor cantidad de arvejas, respetando que:

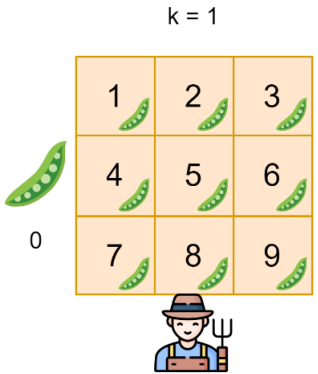
- Empieza desde alguna celda en el comienzo del terreno (y = 0).
- Se puede mover en 2 sentidos:
  - Adelante e izquierda.
  - Adelante y derecha.
- Tiene que llegar al final del terreno (y = N) con una cantidad de arvejas recolectadas divisible por $K + 1$, con K un número fijo dado.

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.

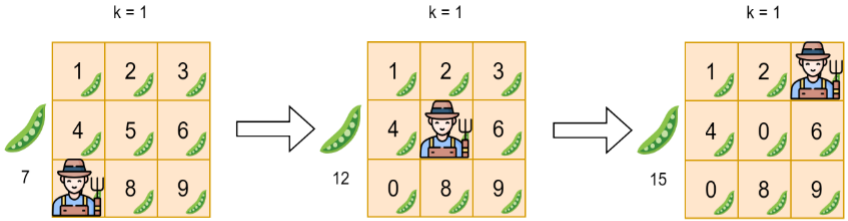
Dibujemos un ejemplo

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



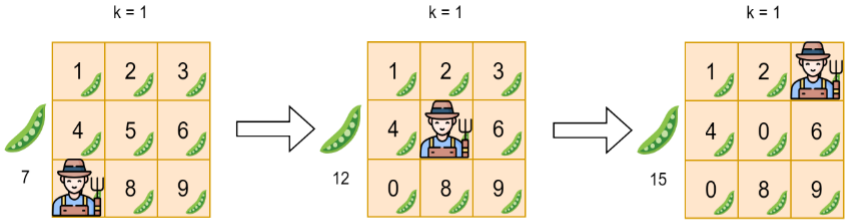
Ejemplo inválido

¿Por qué no es una solución válida?



Ejemplo inválido

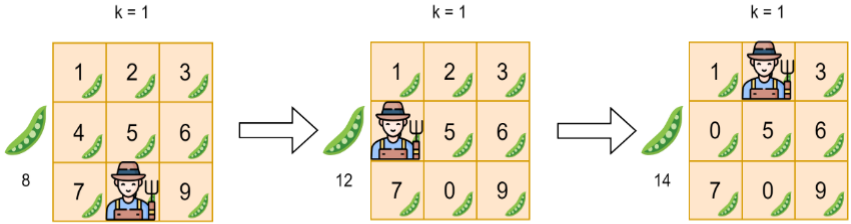
¿Por qué no es una solución válida?



Porque 15 no es divisible por 2.

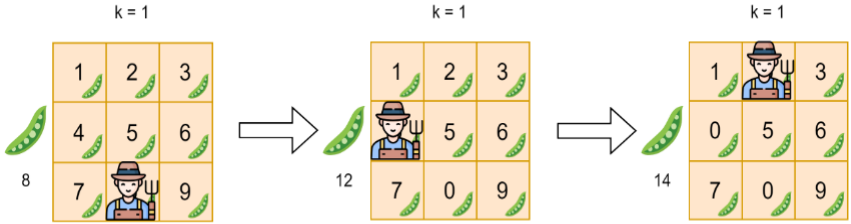
Ejemplo válido

¿Podríamos dar este ejemplo cómo respuesta?



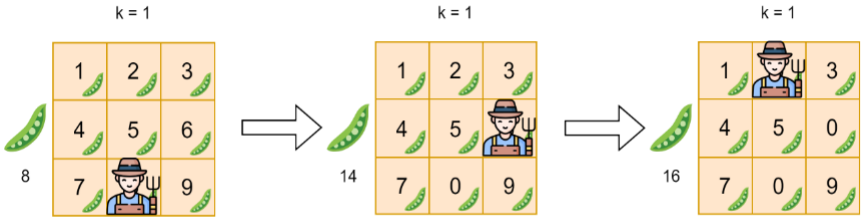
Ejemplo válido

¿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 Tobi 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 Tobi 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? ¡No! 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 Tobi 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 Tobi 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 Tobi 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).

¿Hacemos un break?