Programación dinámica

Santiago Cifuentes

Marzo 2023

Volviendo a Fibonacci

- La técnica de programación dinámica se aplica sobre una función recursiva. La idea es *cachear* los resultados de los llamados intermedios, para luego evitar recalcularlos

$$fibo(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ fibo(n-1) + fibo(n-2) & n > 2 \end{cases}$$

Volviendo a Fibonacci

$$fibo(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ fibo(n-1) + fibo(n-2) & n > 2 \end{cases}$$

- En el caso de fibonacci, se podía demostrar que la cantidad de llamados que se realizan es exponencial, mientras que la cantidad de llamados distintos es lineal.

Volviendo a Fibonacci

$$fibo(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ fibo(n-1) + fibo(n-2) & n > 2 \end{cases}$$

- En el caso de fibonacci, se podía demostrar que la cantidad de llamados que se realizan es exponencial, mientras que la cantidad de llamados distintos es lineal.
- Si los memorizamos, cada estado se calcula una sola vez, y como solo se los visita a lo sumo dos veces, la complejidad temporal queda $O(n)$.

Volviendo a Fibonacci

$$fibo(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ fibo(n-1) + fibo(n-2) & n > 2 \end{cases}$$

- En el caso de fibonacci, se podía demostrar que la cantidad de llamados que se realizan es exponencial, mientras que la cantidad de llamados distintos es lineal.
- Si los memorizamos, cada estado se calcula una sola vez, y como solo se los visita a lo sumo dos veces, la complejidad temporal queda $O(n)$.
- Si pensamos el árbol de llamados, vemos que vamos de la raíz a las hojas, y después volvemos ¿Es posible empezar directamente en las hojas y subir a la raíz?

Volviendo a Fibonacci

- En el caso de fibonacci, es fácil empezar de $i = 0, 1$, e ir subiendo el árbol hasta n.

```
int fibo(int N) {
    int mem[N+1];
    mem[0] = 0; mem[1] = 1;

    for (int i=2; i<=N; i++) mem[i] = mem[i-1] + mem[i-2];

    return mem[N];
}
```

Volviendo a Fibonacci

- En el caso de fibonacci, es fácil empezar de $i = 0, 1$, e ir subiendo el árbol hasta n.

```
int fibo(int N) {
    int mem[N+1];
    mem[0] = 0; mem[1] = 1;

    for (int i=2; i<=N; i++) mem[i] = mem[i-1] + mem[i-2];

    return mem[N];
}
```

- ¿Cambian las complejidades?

Volviendo a Fibonacci

- En el caso de fibonacci, es fácil empezar de $i = 0, 1$, e ir subiendo el árbol hasta n.

```
int fibo(int N) {
    int mem[N+1];
    mem[0] = 0; mem[1] = 1;

    for (int i=2; i<=N; i++) mem[i] = mem[i-1] + mem[i-2];

    return mem[N];
}
```

- ¿Cambian las complejidades? No, pero es más rápido en la práctica.

Volviendo a Fibonacci

- ¿Se puede hacer alguna optimización?

Volviendo a Fibonacci

- ¿Se puede hacer alguna optimización?
- Para calcular $fibo(n)$ solo hace falta tener en memoria $fibo(n-1)$ y $fibo(n-2)$. Luego, para calcular $fibo(n+1)$, alcanza con tener $fibo(n)$ y $f(n-1)$. Podemos ir olvidando los valores.

Volviendo a Fibonacci

- ¿Se puede hacer alguna optimización?
- Para calcular $fibo(n)$ solo hace falta tener en memoria $fibo(n-1)$ y $fibo(n-2)$. Luego, para calcular $fibo(n+1)$, alcanza con tener $fibo(n)$ y $f(n-1)$. Podemos ir olvidando los valores.

```
int fibo(int N) {
    if (N == 0) return 0;
    if (N == 1) return 1;

    int fibN_2 = 0; int fibN_1 = 1;

    while (N >= 2) {
        int fibN = fibN_2 + fibN_1;
        fibN_2 = fibN_1; fibN_1 = fibN;
        N--;
    }

    return fibN_1;
}
```

Volviendo a Fibonacci

- ¿Cómo son ahora las complejidades temporales y espaciales?

Volviendo a Fibonacci

- ¿Cómo son ahora las complejidades temporales y espaciales? Realiza $O(n)$ operaciones, y consume $O(1)$ de espacio.

Volviendo a Fibonacci

- ¿Cómo son ahora las complejidades temporales y espaciales? Realiza $O(n)$ operaciones, y consume $O(1)$ de espacio.
- Para pensar:

Volviendo a Fibonacci

- ¿Cómo son ahora las complejidades temporales y espaciales? Realiza $O(n)$ operaciones, y consume $O(1)$ de espacio.
- Para pensar:
  - ¿Es este algoritmo mejor que el anterior?

Volviendo a Fibonacci

- ¿Cómo son ahora las complejidades temporales y espaciales? Realiza $O(n)$ operaciones, y consume $O(1)$ de espacio.
- Para pensar:
  - ¿Es este algoritmo mejor que el anterior?
  - ¿Es este algoritmo polinomial?

Volviendo a Fibonacci

- ¿Cómo son ahora las complejidades temporales y espaciales? Realiza $O(n)$ operaciones, y consume $O(1)$ de espacio.
- Para pensar:
  - ¿Es este algoritmo mejor que el anterior?
  - ¿Es este algoritmo polinomial?
  - ¿Se puede calcular $\mathit{fibo}(n)$ en tiempo polinomial (es decir, en $\mathit{poly}(\log(n))$)?

Volviendo a Fibonacci

- ¿Cómo son ahora las complejidades temporales y espaciales? Realiza $O(n)$ operaciones, y consume $O(1)$ de espacio.
- Para pensar:
  - ¿Es este algoritmo mejor que el anterior?
  - ¿Es este algoritmo polinomial?
  - ¿Se puede calcular $\mathit{fibo}(n)$ en tiempo polinomial (es decir, en $\mathit{poly}(\log(n))$)?
  - Si ahora nos piden $\mathit{fibo}(k)$ con $k < n$, ¿Cuánto nos cuesta calcularlo?

Volviendo a Fibonacci

- Resumiendo, este nuevo algoritmo recorre el árbol de llamados de abajo hacia arriba. Es decir, *bottom-up*.

Volviendo a Fibonacci

- Resumiendo, este nuevo algoritmo recorre el árbol de llamados de abajo hacia arriba. Es decir, *bottom-up*.
- ¿En qué nos beneficia esto?

Volviendo a Fibonacci

- Resumiendo, este nuevo algoritmo recorre el árbol de llamados de abajo hacia arriba. Es decir, *bottom-up*.
- ¿En qué nos beneficia esto?
  - Es más rápido en la práctica, ya que hacer recursión es más caro que iterar.

Volviendo a Fibonacci

- Resumiendo, este nuevo algoritmo recorre el árbol de llamados de abajo hacia arriba. Es decir, *bottom-up*.
- ¿En qué nos beneficia esto?
  - Es más rápido en la práctica, ya que hacer recursión es más caro que iterar.
  - Nos permitió mejorar la complejidad espacial.

Volviendo a Fibonacci

- Resumiendo, este nuevo algoritmo recorre el árbol de llamados de abajo hacia arriba. Es decir, *bottom-up*.
- ¿En qué nos beneficia esto?
  - Es más rápido en la práctica, ya que hacer recursión es más caro que iterar.
  - Nos permitió mejorar la complejidad espacial.
  - Podría facilitar cuentas, en caso de que sea difícil calcular los sucesores de los estados con un enfoque *top-down*.

Volviendo a Fibonacci

- Ahora, lo malo:

Volviendo a Fibonacci

- Ahora, lo malo:
  - No es tan sencillo pasar del planteo *top-down* a *bottom-up* (como si lo era pasar de *backtracking* a PD).

Volviendo a Fibonacci

- Ahora, lo malo:
  - No es tan sencillo pasar del planteo *top-down* a *bottom-up* (como si lo era pasar de *backtracking* a PD).
  - Requiere encontrar un buen orden para recorrer los estados (*orden topológico*).

Ticketpricing[1]

Enunciado

Faltan W semanas para que vuele un cierto avión. En él entran hasta N pasajeros, y queremos determinar un precio de venta para las entradas (el cual puede variar semana a semana) que maximice la ganancia de la aerolínea. Para cada semana $0 \leq i \leq W$ nos pasan una lista de pares $l_i = (p_1, s_1) \ldots (p_{k_i}, s_{k_i})$ que indica los posibles precios p de venta, junto a la máxima cantidad de boletos s que se venden a ese precio.
Hay que devolver la máxima ganancia posible.

[1] https://open.kattis.com/problems/ticketpricing

Ticketpricing

Sample Input 1

```
50 2
1 437 47
3 357 803 830 13 45 46
1 611 14
```

Sample Output 1

```
23029
437
```

Observación: hay que elegir **algún precio** para cada día

Ticketpricing

Sample Input 2

```
100 3
4 195 223 439 852 92 63 15 1
2 811 893 76 27
1 638 3
1 940 38
```

Sample Output 2

```
83202
852
```

Ticketpricing

- Comencemos planteando una función que resuelva el problema ¿Tiene este alguna cualidad recursiva?

Ticketpricing

- Comencemos planteando una función que resuelva el problema ¿Tiene este alguna cualidad recursiva?
- Si ya sabemos que el primer día se va a elegir el par (p, s), entonces hay que resolver un problema similar al original, pero considerando solo el resto de los días.

Ticketpricing

- Comencemos planteando una función que resuelva el problema ¿Tiene este alguna cualidad recursiva?
- Si ya sabemos que el primer día se va a elegir el par (p, s), entonces hay que resolver un problema similar al original, pero considerando solo el resto de los días.

$$solucion([0 \dots W], N) = p * s + solucion([1 \dots W], N - s)$$

Ticketpricing

- Comencemos planteando una función que resuelva el problema ¿Tiene este alguna cualidad recursiva?
- Si ya sabemos que el primer día se va a elegir el par (p, s), entonces hay que resolver un problema similar al original, pero considerando solo el resto de los días.

$$solucion([0 \ldots W], N) = p * s + solucion([1 \ldots W], N - s)$$

- ¿Cómo elijo el mejor par (p, s)?

Ticketpricing

- Comencemos planteando una función que resuelva el problema ¿Tiene este alguna cualidad recursiva?
- Si ya sabemos que el primer día se va a elegir el par (p, s), entonces hay que resolver un problema similar al original, pero considerando solo el resto de los días.

$$solucion([0 \ldots W], N) = p * s + solucion([1 \ldots W], N - s)$$

- ¿Cómo elijo el mejor par (p, s)? Pruebo todos y me quedo con el máximo.

Ticketpricing

- Comencemos planteando una función que resuelva el problema ¿Tiene este alguna cualidad recursiva?
- Si ya sabemos que el primer día se va a elegir el par (p, s), entonces hay que resolver un problema similar al original, pero considerando solo el resto de los días.

$$solucion([0 \ldots W], N) = p * s + solucion([1 \ldots W], N - s)$$

- ¿Cómo elijo el mejor par (p, s)? Pruebo todos y me quedo con el máximo.
- ¿Cuál sería el caso base?

Ticketpricing

- Comencemos planteando una función que resuelva el problema ¿Tiene este alguna cualidad recursiva?
- Si ya sabemos que el primer día se va a elegir el par (p, s), entonces hay que resolver un problema similar al original, pero considerando solo el resto de los días.

$$solucion([0 \ldots W], N) = p * s + solucion([1 \ldots W], N - s)$$

- ¿Cómo elijo el mejor par (p, s)? Pruebo todos y me quedo con el máximo.
- ¿Cuál sería el caso base? Cuando no quedan más días.

Ticketpricing

- Comencemos planteando una función que resuelva el problema ¿Tiene este alguna cualidad recursiva?
- Si ya sabemos que el primer día se va a elegir el par (p, s), entonces hay que resolver un problema similar al original, pero considerando solo el resto de los días.

$$solucion([0 \dots W], N) = p * s + solucion([1 \dots W], N - s)$$

- ¿Cómo elijo el mejor par (p, s)? Pruebo todos y me quedo con el máximo.
- ¿Cuál sería el caso base? Cuando no quedan más días.
- ¿Siempre se pueden vender las s entradas?

Ticketpricing

- Comencemos planteando una función que resuelva el problema ¿Tiene este alguna cualidad recursiva?
- Si ya sabemos que el primer día se va a elegir el par (p, s), entonces hay que resolver un problema similar al original, pero considerando solo el resto de los días.

$$solucion([0 \dots W], N) = p * s + solucion([1 \dots W], N - s)$$

- ¿Cómo elijo el mejor par (p, s)? Pruebo todos y me quedo con el máximo.
- ¿Cuál sería el caso base? Cuando no quedan más días.
- ¿Siempre se pueden vender las s entradas? No, solo se pueden vender $\min(N, s)$.

Ticketpricing

Una función recursiva que resuelve el problema es:

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

Ticketpricing

Una función recursiva que resuelve el problema es:

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

Observación: ¿Qué pasa cuando $n = 0$?

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

- La función devuelve "La máxima ganancia alcanzable empezando del día i y teniendo n boletos disponibles".

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n-\min(n,s))\} & \text{caso contrario} \end{cases}$$

- La función devuelve “La máxima ganancia alcanzable empezando del día i y teniendo n boletos disponibles”.
- ¿Qué llamado resuelve el problema?

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n-\min(n,s))\} & \text{caso contrario} \end{cases}$$

- La función devuelve “La máxima ganancia alcanzable empezando del día i y teniendo n boletos disponibles”.
- ¿Qué llamado resuelve el problema? $TP(0, N)$.

Ticketpricing

$$TP(i, n) = \begin{cases} 0 & i = W + 1 \\ \max\limits_{(p,s) \in I_i} \{p \times \min(n, s) + TP(i + 1, n - \min(n, s))\} & \text{caso contrario} \end{cases}$$

- La función devuelve "La máxima ganancia alcanzable empezando del día i y teniendo n boletos disponibles".
- ¿Qué llamado resuelve el problema? $TP(0, N)$.
- Complejidad de la función:

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n-\min(n,s))\} & \text{caso contrario} \end{cases}$$

- La función devuelve "La máxima ganancia alcanzable empezando del día i y teniendo n boletos disponibles".
- ¿Qué llamado resuelve el problema? $TP(0, N)$.
- Complejidad de la función:
  - ¿Cuántas operaciones se hacen por estado?

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

- La función devuelve "La máxima ganancia alcanzable empezando del día i y teniendo n boletos disponibles".
- ¿Qué llamado resuelve el problema? $TP(0, N)$.
- Complejidad de la función:
  - ¿Cuántas operaciones se hacen por estado? A lo sumo $K = \max\limits_{0 \leq i \leq W} \{k_i\}$.
  - ¿Cuántos nodos tiene el árbol?

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n-\min(n,s))\} & \text{caso contrario} \end{cases}$$

- La función devuelve "La máxima ganancia alcanzable empezando del día i y teniendo n boletos disponibles".
- ¿Qué llamado resuelve el problema? $TP(0,N)$.
- Complejidad de la función:
  - ¿Cuántas operaciones se hacen por estado? A lo sumo $K = \max\limits_{0 \le i \le W} \{k_i\}$.
  - ¿Cuántos nodos tiene el árbol? Es un árbol de profundidad W con *branching* K, por lo que tiene $O(K^W)$.

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n-\min(n,s))\} & \text{caso contrario} \end{cases}$$

- La función devuelve "La máxima ganancia alcanzable empezando del día i y teniendo n boletos disponibles".
- ¿Qué llamado resuelve el problema? $TP(0, N)$.
- Complejidad de la función:
  - ¿Cuántas operaciones se hacen por estado? A lo sumo $K = \max\limits_{0 \leq i \leq W} \{k_i\}$.
  - ¿Cuántos nodos tiene el árbol? Es un árbol de profundidad W con *branching* K, por lo que tiene $O(K^W)$.
- Luego, la complejidad temporal es $O(K^W K)$.

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

- Contemos la cantidad de estados distintos para ver si vale la pena hacer programación dinámica ¿Cuántos hay?

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

- Contemos la cantidad de estados distintos para ver si vale la pena hacer programación dinámica ¿Cuántos hay?
- Hay un orden de WN estados. Si comparamos con la cantidad de llamados recursivos vemos que:

$$WN \text{ vs } K^W$$

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

- Contemos la cantidad de estados distintos para ver si vale la pena hacer programación dinámica ¿Cuántos hay?
- Hay un orden de WN estados. Si comparamos con la cantidad de llamados recursivos vemos que:

$$WN \text{ vs } K^W$$

- La desigualdad depende del valor de N. El enunciado dice que $0 \leq N \leq 100, 0 \leq W \leq 52, 0 \leq K \leq 100$. Luego:

$$52 \times 100 <<<< 100^{52}$$

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W + 1 \\ \max\limits_{(p,s) \in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

- Contemos la cantidad de estados distintos para ver si vale la pena hacer programación dinámica ¿Cuántos hay?
- Hay un orden de WN estados. Si comparamos con la cantidad de llamados recursivos vemos que:

$$WN \text{ vs } K^W$$

- La desigualdad depende del valor de N. El enunciado dice que $0 \leq N \leq 100, 0 \leq W \leq 52, 0 \leq K \leq 100$. Luego:

$$52 \times 100 <<<< 100^{52}$$

- Por lo tanto, el algoritmo de PD genera un árbol mucho más chico.

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n-\min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Qué estructura usamos para memorizar las soluciones?

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Qué estructura usamos para memorizar las soluciones? Una matriz de $W \times N$.

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n-\min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Qué estructura usamos para memorizar las soluciones? Una matriz de $W \times N$.
- ¿Cuál es la complejidad del algoritmo de PD?

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Qué estructura usamos para memorizar las soluciones? Una matriz de $W \times N$.
- ¿Cuál es la complejidad del algoritmo de PD?
    - Costo por estado:

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Qué estructura usamos para memorizar las soluciones? Una matriz de $W \times N$.
- ¿Cuál es la complejidad del algoritmo de PD?
  - Costo por estado: $O(K)$

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Qué estructura usamos para memorizar las soluciones? Una matriz de $W \times N$.
- ¿Cuál es la complejidad del algoritmo de PD?
  - Costo por estado: $O(K)$
  - Cantidad de estados:

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Qué estructura usamos para memorizar las soluciones? Una matriz de $W \times N$.
- ¿Cuál es la complejidad del algoritmo de PD?
  - Costo por estado: $O(K)$
  - Cantidad de estados: $O(WN)$

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n-\min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Qué estructura usamos para memorizar las soluciones? Una matriz de $W \times N$.
- ¿Cuál es la complejidad del algoritmo de PD?
  - Costo por estado: $O(K)$
  - Cantidad de estados: $O(WN)$
- Luego, la complejidad temporal es $O(WNK)$.

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Qué estructura usamos para memorizar las soluciones? Una matriz de $W \times N$.
- ¿Cuál es la complejidad del algoritmo de PD?
  - Costo por estado: $O(K)$
  - Cantidad de estados: $O(WN)$
- Luego, la complejidad temporal es $O(WNK)$.
- Veamos el código.

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n-\min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Hay alguna forma de computar la respuesta de forma *bottom up*?

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n-\min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Hay alguna forma de computar la respuesta de forma *bottom up*?
- Podemos avanzar semana a semana, empezando desde la última (el caso base)

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n-\min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Hay alguna forma de computar la respuesta de forma *bottom up*?
- Podemos avanzar semana a semana, empezando desde la última (el caso base) ¿Se puede optimizar la memoria de alguna forma?

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n-\min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Hay alguna forma de computar la respuesta de forma *bottom up*?
- Podemos avanzar semana a semana, empezando desde la última (el caso base) ¿Se puede optimizar la memoria de alguna forma?
- Para calcular los valores de la semana i-ésima alcanza con conocer los valores para la semana $i+1$-ésima.

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n-\min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Hay alguna forma de computar la respuesta de forma *bottom up*?
- Podemos avanzar semana a semana, empezando desde la última (el caso base) ¿Se puede optimizar la memoria de alguna forma?
- Para calcular los valores de la semana i-ésima alcanza con conocer los valores para la semana $i+1$-ésima.
- Podemos mantener un único vector con la i-ésima fila, iniciándolo en la $W+1$-ésima.

Ticketpricing

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n-\min(n,s))\} & \text{caso contrario} \end{cases}$$

- ¿Hay alguna forma de computar la respuesta de forma *bottom up*?
- Podemos avanzar semana a semana, empezando desde la última (el caso base) ¿Se puede optimizar la memoria de alguna forma?
- Para calcular los valores de la semana i-ésima alcanza con conocer los valores para la semana $i+1$-ésima.
- Podemos mantener un único vector con la i-ésima fila, iniciándolo en la $W+1$-ésima.
- Código.

Construyendo la solución

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

- Supongamos que ahora no solo nos piden la ganancia, sino que también tenemos que devolver los precios que deben usarse para alcanzar esa ganancia máxima.

Construyendo la solución

$$TP(i,n) = \begin{cases} 0 & i = W+1 \\ \max\limits_{(p,s)\in I_i} \{p \times \min(n,s) + TP(i+1, n - \min(n,s))\} & \text{caso contrario} \end{cases}$$

- Supongamos que ahora no solo nos piden la ganancia, sino que también tenemos que devolver los precios que deben usarse para alcanzar esa ganancia máxima.
- ¿Cómo podríamos saber, por lo menos, el precio que se asigna a la primera semana?

Construyendo la solución

- Justamente tenemos que encontrar el par $(p, c) \in I_0$ que cumple:

$$TP(0, N) = p \times \min(n, s) + TP(1, N - \min(N, s))$$

Construyendo la solución

- Justamente tenemos que encontrar el par $(p, c) \in I_0$ que cumple:

$$TP(0, N) = p \times \min(n, s) + TP(1, N - \min(N, s))$$

- Si tenemos todos los valores guardados en la memoria dp, alcanza con buscar un par que cumpla:

$$dp[0, N] = p \times \min(n, s) + dp[1, N - \min(N, s)]$$

Construyendo la solución

- Justamente tenemos que encontrar el par $(p, c) \in I_0$ que cumple:

$$TP(0, N) = p \times \min(n, s) + TP(1, N - \min(N, s))$$

- Si tenemos todos los valores guardados en la memoria dp, alcanza con buscar un par que cumpla:

$$dp[0, N] = p \times \min(n, s) + dp[1, N - \min(N, s)]$$

- ¿Cuánto nos cuesta hacer eso?

Construyendo la solución

- Justamente tenemos que encontrar el par $(p, c) \in I_0$ que cumple:

$$TP(0, N) = p \times \min(n, s) + TP(1, N - \min(N, s))$$

- Si tenemos todos los valores guardados en la memoria dp, alcanza con buscar un par que cumpla:

$$dp[0, N] = p \times \min(n, s) + dp[1, N - \min(N, s)]$$

- ¿Cuánto nos cuesta hacer eso? $O(K)$

Construyendo la solución

- ¿Y cómo hacemos para descubrir el precio óptimo de la segunda semana?

Construyendo la solución

- ¿Y cómo hacemos para descubrir el precio óptimo de la segunda semana?
- Supongamos que la selección óptima de la primera semana nos dejó en el estado $(1, n)$. Solo tenemos que repetir el proceso para este nuevo estado. Es decir, buscamos un par $(p', s') \in I_1$ que cumpla

$$TP(1, n) = p' \times \min(n, s') + TP(2, n - \min(n, s'))$$

Construyendo la solución

- ¿Y cómo hacemos para descubrir el precio óptimo de la segunda semana?
- Supongamos que la selección óptima de la primera semana nos dejó en el estado $(1, n)$. Solo tenemos que repetir el proceso para este nuevo estado. Es decir, buscamos un par $(p', s') \in I_1$ que cumpla

$$TP(1, n) = p' \times \min(n, s') + TP(2, n - \min(n, s'))$$

- Asi vamos armando la solución recursivamente.

Construyendo la solución

- ¿Y cómo hacemos para descubrir el precio óptimo de la segunda semana?
- Supongamos que la selección óptima de la primera semana nos dejó en el estado $(1, n)$. Solo tenemos que repetir el proceso para este nuevo estado. Es decir, buscamos un par $(p', s') \in I_1$ que cumpla

$$TP(1, n) = p' \times \min(n, s') + TP(2, n - \min(n, s'))$$

- Asi vamos armando la solución recursivamente.
- ¿Qué complejidad toma?

Construyendo la solución

- ¿Y cómo hacemos para descubrir el precio óptimo de la segunda semana?
- Supongamos que la selección óptima de la primera semana nos dejó en el estado $(1, n)$. Solo tenemos que repetir el proceso para este nuevo estado. Es decir, buscamos un par $(p', s') \in I_1$ que cumpla

$$TP(1, n) = p' \times \min(n, s') + TP(2, n - \min(n, s'))$$

- Asi vamos armando la solución recursivamente.
- ¿Qué complejidad toma? $O(WK)$

Construyendo la solución

- ¿Y cómo hacemos para descubrir el precio óptimo de la segunda semana?
- Supongamos que la selección óptima de la primera semana nos dejó en el estado $(1, n)$. Solo tenemos que repetir el proceso para este nuevo estado. Es decir, buscamos un par $(p', s') \in I_1$ que cumpla

$$TP(1, n) = p' \times \min(n, s') + TP(2, n - \min(n, s'))$$

- Asi vamos armando la solución recursivamente.
- ¿Qué complejidad toma? $O(WK)$
- Código.

Construyendo la solución

Resumiendo:

Construyendo la solución

Resumiendo:

- Si tenemos la matriz con todos los estados, no es difícil reconstruir la solución recursivamente.

Construyendo la solución

Resumiendo:

- Si tenemos la matriz con todos los estados, no es difícil reconstruir la solución recursivamente.
- ¿Podemos hacerlo con la implementación *bottom up* que gasta menos memoria?

Construyendo la solución

Resumiendo:

- Si tenemos la matriz con todos los estados, no es difícil reconstruir la solución recursivamente.
- ¿Podemos hacerlo con la implementación *bottom up* que gasta menos memoria?
- ¿Qué complejidad nos toma?

Construyendo la solución

Resumiendo:

- Si tenemos la matriz con todos los estados, no es difícil reconstruir la solución recursivamente.
- ¿Podemos hacerlo con la implementación *bottom up* que gasta menos memoria?
- ¿Qué complejidad nos toma? En general, a lo sumo la misma que tomó calcular la matriz. Solo estamos repitiendo la toma de decisiones, prestando atención a qué valores se van eligiendo.

Dados indistinguibles

Enunciado

Queremos saber cuántas formas hay de sumar s usando n dados de k caras, cuando estos son indistinguibles.
Nos piden un algoritmo que tenga complejidad temporal $O(nsk)$.

Dados indistinguibles

Enunciado

Queremos saber cuántas formas hay de sumar s usando n dados de k caras, cuando estos son indistinguibles.
Nos piden un algoritmo que tenga complejidad temporal $O(nsk)$.

Por ejemplo, si $s = k = 5$ y $n = 3$, se tienen 2 formas:

$$122$$

$$131$$

Dados indistinguibles

- ¿Hay alguna recursión que permita descomponer el problema?

Dados indistinguibles

- ¿Hay alguna recursión que permita descomponer el problema?
- Fijar el primer dado es tentador, pero es difícil evitar contar muchas veces las mismas soluciones.

Dados indistinguibles

- ¿Hay alguna recursión que permita descomponer el problema?
- Fijar el primer dado es tentador, pero es difícil evitar contar muchas veces las mismas soluciones.
- ¿Hay alguna forma de representar las tiradas que evite contar repetidos? Es decir, ¿Tal que las tiradas 131 y 311 sean la misma?

Dados indistinguibles

- ¿Hay alguna recursión que permita descomponer el problema?
- Fijar el primer dado es tentador, pero es difícil evitar contar muchas veces las mismas soluciones.
- ¿Hay alguna forma de representar las tiradas que evite contar repetidos? Es decir, ¿Tal que las tiradas 131 y 311 sean la misma?
- Podemos **ordenarlas**:

$$122 \implies 221$$

$$131 \implies 311$$

Dados indistinguibles

- ¿Hay alguna recursión que permita descomponer el problema?
- Fijar el primer dado es tentador, pero es difícil evitar contar muchas veces las mismas soluciones.
- ¿Hay alguna forma de representar las tiradas que evite contar repetidos? Es decir, ¿Tal que las tiradas 131 y 311 sean la misma?
- Podemos **ordenarlas**:

$$122 \implies 221$$

$$131 \implies 311$$

- Ahora nos enfrentamos al problema de contar secuencias decrecientes de longitud n acotadas por k que suman s.

Dados indistinguibles

- ¿Cómo podemos contar estas secuencias? ¿Hay alguna recursión posible?

Dados indistinguibles

- ¿Cómo podemos contar estas secuencias? ¿Hay alguna recursión posible?
- Si elegimos como primer elemento al valor más alto, k ¿Qué sabemos que tiene que cumplir el resto de la secuencia?

Dados indistinguibles

- ¿Cómo podemos contar estas secuencias? ¿Hay alguna recursión posible?
- Si elegimos como primer elemento al valor más alto, k ¿Qué sabemos que tiene que cumplir el resto de la secuencia?
- Será una secuencia ordenada de longitud $n-1$ con todos valores menores o iguales a k ¿Cuánto tienen que sumar sus valores?

Dados indistinguibles

- ¿Cómo podemos contar estas secuencias? ¿Hay alguna recursión posible?
- Si elegimos como primer elemento al valor más alto, k ¿Qué sabemos que tiene que cumplir el resto de la secuencia?
- Será una secuencia ordenada de longitud $n-1$ con todos valores menores o iguales a k ¿Cuánto tienen que sumar sus valores? $s-k$.

Dados indistinguibles

- ¿Cómo podemos contar estas secuencias? ¿Hay alguna recursión posible?
- Si elegimos como primer elemento al valor más alto, k ¿Qué sabemos que tiene que cumplir el resto de la secuencia?
- Será una secuencia ordenada de longitud $n-1$ con todos valores menores o iguales a k ¿Cuánto tienen que sumar sus valores? $s-k$.
- Ahora, si no elegimos el valor k, ¿Qué forma tendrá nuestra secuencia?

Dados indistinguibles

- ¿Cómo podemos contar estas secuencias? ¿Hay alguna recursión posible?
- Si elegimos como primer elemento al valor más alto, k ¿Qué sabemos que tiene que cumplir el resto de la secuencia?
- Será una secuencia ordenada de longitud $n-1$ con todos valores menores o iguales a k ¿Cuánto tienen que sumar sus valores? $s-k$.
- Ahora, si no elegimos el valor k, ¿Qué forma tendrá nuestra secuencia? Será una secuencia decreciente de longitud n, que suma s, pero acotada por $k-1$.

Dados indistinguibles

- ¿Cómo podemos contar estas secuencias? ¿Hay alguna recursión posible?
- Si elegimos como primer elemento al valor más alto, k ¿Qué sabemos que tiene que cumplir el resto de la secuencia?
- Será una secuencia ordenada de longitud $n-1$ con todos valores menores o iguales a k ¿Cuánto tienen que sumar sus valores? $s-k$.
- Ahora, si no elegimos el valor k, ¿Qué forma tendrá nuestra secuencia? Será una secuencia decreciente de longitud n, que suma s, pero acotada por $k-1$.
- ¿Qué parámetros son importantes?

Dados indistinguibles

- ¿Cómo podemos contar estas secuencias? ¿Hay alguna recursión posible?
- Si elegimos como primer elemento al valor más alto, k ¿Qué sabemos que tiene que cumplir el resto de la secuencia?
- Será una secuencia ordenada de longitud $n-1$ con todos valores menores o iguales a k ¿Cuánto tienen que sumar sus valores? $s-k$.
- Ahora, si no elegimos el valor k, ¿Qué forma tendrá nuestra secuencia? Será una secuencia decreciente de longitud n, que suma s, pero acotada por $k-1$.
- ¿Qué parámetros son importantes? La longitud n, el objetivo s y el límite k.

Dados indistinguibles

- ¿Cómo podemos contar estas secuencias? ¿Hay alguna recursión posible?
- Si elegimos como primer elemento al valor más alto, k ¿Qué sabemos que tiene que cumplir el resto de la secuencia?
- Será una secuencia ordenada de longitud $n-1$ con todos valores menores o iguales a k ¿Cuánto tienen que sumar sus valores? $s-k$.
- Ahora, si no elegimos el valor k, ¿Qué forma tendrá nuestra secuencia? Será una secuencia decreciente de longitud n, que suma s, pero acotada por $k-1$.
- ¿Qué parámetros son importantes? La longitud n, el objetivo s y el límite k.
- ¿Casos base?

Dados indistinguibles

- ¿Cómo podemos contar estas secuencias? ¿Hay alguna recursión posible?
- Si elegimos como primer elemento al valor más alto, k ¿Qué sabemos que tiene que cumplir el resto de la secuencia?
- Será una secuencia ordenada de longitud $n-1$ con todos valores menores o iguales a k ¿Cuánto tienen que sumar sus valores? $s-k$.
- Ahora, si no elegimos el valor k, ¿Qué forma tendrá nuestra secuencia? Será una secuencia decreciente de longitud n, que suma s, pero acotada por $k-1$.
- ¿Qué parámetros son importantes? La longitud n, el objetivo s y el límite k.
- ¿Casos base? La secuencia vacía. Hay que ver que hayamos alcanzado el objetivo, y solo usar números válidos entre 1 y k.

Dados indistinguibles

Resumiendo todo en una expresión, se podría escribir:

$$DI(i,r,l) = \begin{cases} 0 & i = 0 \neq r \\ 1 & i = 0 = r \\ DI(i-1, r-l, l) & i \neq 0, l = 1 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

Complejidad de la función:

Dados indistinguibles

Resumiendo todo en una expresión, se podría escribir:

$$DI(i,r,l) = \begin{cases} 0 & i = 0 \neq r \\ 1 & i = 0 = r \\ DI(i-1, r-l, l) & i \neq 0, l = 1 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

Complejidad de la función:

- Cantidad de llamados recursivos:

Dados indistinguibles

Resumiendo todo en una expresión, se podría escribir:

$$DI(i,r,l) = \begin{cases} 0 & i = 0 \neq r \\ 1 & i = 0 = r \\ DI(i-1, r-l, l) & i \neq 0, l = 1 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

Complejidad de la función:

- Cantidad de llamados recursivos: si $l \sim i$, habrá por lo menos $\Omega(2^i)$ llamados recursivos.

Dados indistinguibles

Resumiendo todo en una expresión, se podría escribir:

$$DI(i,r,l) = \begin{cases} 0 & i = 0 \neq r \\ 1 & i = 0 = r \\ DI(i-1, r-l, l) & i \neq 0, l = 1 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

Complejidad de la función:

- Cantidad de llamados recursivos: si $l \sim i$, habrá por lo menos $\Omega(2^i)$ llamados recursivos.
- Costo por llamado:

Dados indistinguibles

Resumiendo todo en una expresión, se podría escribir:

$$DI(i,r,l) = \begin{cases} 0 & i = 0 \neq r \\ 1 & i = 0 = r \\ DI(i-1, r-l, l) & i \neq 0, l = 1 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

Complejidad de la función:

- Cantidad de llamados recursivos: si $l \sim i$, habrá por lo menos $\Omega(2^i)$ llamados recursivos.
- Costo por llamado: $O(1)$

Dados indistinguibles

Resumiendo todo en una expresión, se podría escribir:

$$DI(i,r,l) = \begin{cases} 0 & i = 0 \neq r \\ 1 & i = 0 = r \\ DI(i-1, r-l, l) & i \neq 0, l = 1 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

Complejidad de la función:

- Cantidad de llamados recursivos: si $l \sim i$, habrá por lo menos $\Omega(2^i)$ llamados recursivos.
- Costo por llamado: $O(1)$

¿Qué llamado resuelve el problema?

Dados indistinguibles

Resumiendo todo en una expresión, se podría escribir:

$$DI(i,r,l) = \begin{cases} 0 & i = 0 \neq r \\ 1 & i = 0 = r \\ DI(i-1, r-l, l) & i \neq 0, l = 1 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

Complejidad de la función:

- Cantidad de llamados recursivos: si $l \sim i$, habrá por lo menos $\Omega(2^i)$ llamados recursivos.
- Costo por llamado: $O(1)$

¿Qué llamado resuelve el problema? $DI(n, s, k)$

Dados indistinguibles

Resumiendo todo en una expresión, se podría escribir:

$$DI(i,r,l) = \begin{cases} 0 & i = 0 \neq r \\ 1 & i = 0 = r \\ DI(i-1, r-l, l) & i \neq 0, l = 1 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

Complejidad de la función:

- Cantidad de llamados recursivos: si $l \sim i$, habrá por lo menos $\Omega(2^i)$ llamados recursivos.
- Costo por llamado: $O(1)$

¿Qué llamado resuelve el problema? $DI(n, s, k)$
Ya vemos entonces que calcular la función directamente no alcanza la complejidad pedida.

Dados indistinguibles

$$DI(i,r,l)=\begin{cases}0 & i=0\neq r\\ 1 & i=0=r\\ DI(i-1,r-l,l) & i\neq 0, l=1\\ DI(i-1,r-l,l)+DI(i,r,l-1) & \text{caso contrario}\end{cases}$$

- Ahora, ¿Cuántos llamados distintos pueden hacerse a esta función?

Dados indistinguibles

$$DI(i,r,l)=\begin{cases}0 & i=0\neq r\\ 1 & i=0=r\\ DI(i-1,r-l,l) & i\neq 0, l=1\\ DI(i-1,r-l,l)+DI(i,r,l-1) & \text{caso contrario}\end{cases}$$

- Ahora, ¿Cuántos llamados distintos pueden hacerse a esta función?
- En un principio no están acotados, porque r puede ser negativo.

Dados indistinguibles

$$DI(i,r,l) = \begin{cases} 0 & i = 0 \neq r \\ 1 & i = 0 = r \\ DI(i-1, r-l, l) & i \neq 0, l = 1 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- Ahora, ¿Cuántos llamados distintos pueden hacerse a esta función?
- En un principio no están acotados, porque r puede ser negativo. Arreglemos estos casos:

$$DI(i,r,l) = \begin{cases} 0 & r < 0 \vee l < 1 \\ r = 0 & i = 0 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

Dados indistinguibles

$$DI(i,r,l) = \begin{cases} 0 & i = 0 \neq r \\ 1 & i = 0 = r \\ DI(i-1, r-l, l) & i \neq 0, l = 1 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- Ahora, ¿Cuántos llamados distintos pueden hacerse a esta función?
- En un principio no están acotados, porque r puede ser negativo. Arreglemos estos casos:

$$DI(i,r,l) = \begin{cases} 0 & r < 0 \vee l < 1 \\ r = 0 & i = 0 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- Ahora, ¿Cuántos llamados distintos hay?

Dados indistinguibles

$$DI(i,r,l) = \begin{cases} 0 & i = 0 \neq r \\ 1 & i = 0 = r \\ DI(i-1, r-l, l) & i \neq 0, l = 1 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- Ahora, ¿Cuántos llamados distintos pueden hacerse a esta función?
- En un principio no están acotados, porque r puede ser negativo. Arreglemos estos casos:

$$DI(i,r,l) = \begin{cases} 0 & r < 0 \vee l < 1 \\ r = 0 & i = 0 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- Ahora, ¿Cuántos llamados distintos hay? $O(irl)$.

Dados indistinguibles

$$DI(i,r,l) = \begin{cases} 0 & r < 0 \vee l < 1 \\ r = 0 & i = 0 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- Hay a lo sumo $O(irl)$ llamados distintos ¿Cuánto cuesta resolver cada llamado?

Dados indistinguibles

$$DI(i,r,l) = \begin{cases} 0 & r < 0 \lor l < 1 \\ r = 0 & i = 0 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- Hay a lo sumo $O(irl)$ llamados distintos ¿Cuánto cuesta resolver cada llamado? $O(1)$.

Dados indistinguibles

$$DI(i, r, l) = \begin{cases} 0 & r < 0 \vee l < 1 \\ r = 0 & i = 0 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- Hay a lo sumo $O(irl)$ llamados distintos ¿Cuánto cuesta resolver cada llamado? $O(1)$.
- Si usamos una matriz de $i \times r \times l$ para guardar los resultados intemedios ¿Cuál es la complejidad resultante?

Dados indistinguibles

$$DI(i,r,l) = \begin{cases} 0 & r < 0 \vee l < 1 \\ r = 0 & i = 0 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- Hay a lo sumo $O(irl)$ llamados distintos ¿Cuánto cuesta resolver cada llamado? $O(1)$.
- Si usamos una matriz de $i \times r \times l$ para guardar los resultados intemedios ¿Cuál es la complejidad resultante? $O(irl)$.

Dados indistinguibles

$$DI(i,r,l) = \begin{cases} 0 & r < 0 \lor l < 1 \\ r = 0 & i = 0 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- Hay a lo sumo $O(irl)$ llamados distintos ¿Cuánto cuesta resolver cada llamado? $O(1)$.
- Si usamos una matriz de $i \times r \times l$ para guardar los resultados intemedios ¿Cuál es la complejidad resultante? $O(irl)$.
- Para nuestro problema particular reemplazamos $i = n$, $r = s$ y $l = k$.

Dados indistinguibles

- Observación: ¿Hay algún otro caso trivial de resolver? ¿Qué pasa si s es muy grande?

Dados indistinguibles

- Observación: ¿Hay algún otro caso trivial de resolver? ¿Qué pasa si s es muy grande?
- Si $s > nk$ la respuesta es trivialmente 1. Si agregamos esta condición al algoritmo antes de cargar la memoria, bajamos la complejidad a $O(n \min(s, nk)k)$.

Dados indistinguibles

- Observación: ¿Hay algún otro caso trivial de resolver? ¿Qué pasa si s es muy grande?
- Si $s > nk$ la respuesta es trivialmente 1. Si agregamos esta condición al algoritmo antes de cargar la memoria, bajamos la complejidad a $O(n \min(s, nk)k)$.
- Código.

Dados indistinguibles

$$DI(i,r,l) = \begin{cases} 0 & r < 0 \vee l < 1 \\ r = 0 & i = 0 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- ¿Hay alguna forma de calcular la función de forma *bottom up*?

Dados indistinguibles

$$DI(i, r, l) = \begin{cases} 0 & r < 0 \lor l < 1 \\ r = 0 & i = 0 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- ¿Hay alguna forma de calcular la función de forma *bottom up*?
- Para calcular la cantidad de cadenas de longitud i necesitamos las de longitud $i-1$, pero también las de i con menor limite l.

Dados indistinguibles

$$DI(i,r,l) = \begin{cases} 0 & r < 0 \lor l < 1 \\ r = 0 & i = 0 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- ¿Hay alguna forma de calcular la función de forma *bottom up*?
- Para calcular la cantidad de cadenas de longitud i necesitamos las de longitud $i-1$, pero también las de i con menor limite l.
- O sea, podemos calcular de $i = 0$ a N, siempre calculando primero los valores con cotas menores (i.e. de $l = 0$ a $l = k$).

Dados indistinguibles

$$DI(i,r,l) = \begin{cases} 0 & r < 0 \vee l < 1 \\ r = 0 & i = 0 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- ¿Hay alguna forma de calcular la función de forma *bottom up*?
- Para calcular la cantidad de cadenas de longitud i necesitamos las de longitud $i-1$, pero también las de i con menor limite l.
- O sea, podemos calcular de $i = 0$ a N, siempre calculando primero los valores con cotas menores (i.e. de $l = 0$ a $l = k$).
- ¿Cuántos valores mantenemos en memoria?

Dados indistinguibles

$$DI(i,r,l) = \begin{cases} 0 & r < 0 \vee l < 1 \\ r = 0 & i = 0 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- ¿Hay alguna forma de calcular la función de forma *bottom up*?
- Para calcular la cantidad de cadenas de longitud i necesitamos las de longitud $i-1$, pero también las de i con menor limite l.
- O sea, podemos calcular de $i = 0$ a N, siempre calculando primero los valores con cotas menores (i.e. de $l = 0$ a $l = k$).
- ¿Cuántos valores mantenemos en memoria? $O(sk)$.

Dados indistinguibles

$$DI(i, r, l) = \begin{cases} 0 & r < 0 \vee l < 1 \\ r = 0 & i = 0 \\ DI(i-1, r-l, l) + DI(i, r, l-1) & \text{caso contrario} \end{cases}$$

- ¿Hay alguna forma de calcular la función de forma *bottom up*?
- Para calcular la cantidad de cadenas de longitud i necesitamos las de longitud $i-1$, pero también las de i con menor limite l.
- O sea, podemos calcular de $i = 0$ a N, siempre calculando primero los valores con cotas menores (i.e. de $l = 0$ a $l = k$).
- ¿Cuántos valores mantenemos en memoria? $O(sk)$.
- Código.

Conclusión

Hoy vimos:

Conclusión

Hoy vimos:

- Que en algunos casos es posible encontrar un orden de cómputo de los estados que evita hacer recursión. Esto mejora la complejidad del algoritmo en la práctica, y a veces nos permite ahorrar memoria.

Conclusión

Hoy vimos:

- Que en algunos casos es posible encontrar un orden de cómputo de los estados que evita hacer recursión. Esto mejora la complejidad del algoritmo en la práctica, y a veces nos permite ahorrar memoria.
- Es posible reconstruir la solución del problema revisando las decisiones que se tomaron durante la recursión. Para esto, se pueden usar las respuestas memorizadas.

Conclusión

Hoy vimos:

- Que en algunos casos es posible encontrar un orden de cómputo de los estados que evita hacer recursión. Esto mejora la complejidad del algoritmo en la práctica, y a veces nos permite ahorrar memoria.
- Es posible reconstruir la solución del problema revisando las decisiones que se tomaron durante la recursión. Para esto, se pueden usar las respuestas memorizadas.
- Pensando en el problema de los dados, a veces conviene repensar el espacio de soluciones haciendo analogías con otros problemas.

Datos curiosos

- Se puede probar que el árbol de recursión de $fibo(n)$ tiene exactamente $2fibo(n+1)-1$ nodos. Como $fibo(n+1)=\Theta(\phi^n)$, esto demuestra que siempre hay una cantidad exponencial de llamados.

Datos curiosos

- Se puede probar que el árbol de recursión de $\mathit{fibo}(n)$ tiene exactamente $2\mathit{fibo}(n+1)-1$ nodos. Como $\mathit{fibo}(n+1)=\Theta(\phi^n)$, esto demuestra que siempre hay una cantidad exponencial de llamados.
- Si bien no se puede calcular $\mathit{fibo}(n)$ en $\mathit{poly}(\log(n))$ por el tamaño del número, podemos platenar el problema de calcular $\mathit{fibo}(n) \mod r$ ¿Los algoritmos que vimos resuelven este problema en $\mathit{poly}(\log(n)+\log(r))$?

Datos curiosos

- Se puede probar que el árbol de recursión de $\mathit{fibo}(n)$ tiene exactamente $2\mathit{fibo}(n+1)-1$ nodos. Como $\mathit{fibo}(n+1)=\Theta(\phi^n)$, esto demuestra que siempre hay una cantidad exponencial de llamados.
- Si bien no se puede calcular $\mathit{fibo}(n)$ en $\mathit{poly}(\log(n))$ por el tamaño del número, podemos platenar el problema de calcular $\mathit{fibo}(n) \mod r$ ¿Los algoritmos que vimos resuelven este problema en $\mathit{poly}(\log(n)+\log(r))$? No, porque hacen $O(n)$ cuentas.

Datos curiosos

- Se puede probar que el árbol de recursión de $\mathit{fibo}(n)$ tiene exactamente $2\mathit{fibo}(n+1) - 1$ nodos. Como $\mathit{fibo}(n+1) = \Theta(\phi^n)$, esto demuestra que siempre hay una cantidad exponencial de llamados.
- Si bien no se puede calcular $\mathit{fibo}(n)$ en $\mathit{poly}(\log(n))$ por el tamaño del número, podemos platenar el problema de calcular $\mathit{fibo}(n) \mod r$ ¿Los algoritmos que vimos resuelven este problema en $\mathit{poly}(\log(n) + \log(r))$? No, porque hacen $O(n)$ cuentas.
- No obstante, se puede calcular usando la siguiente propiedad:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$$

Datos curiosos

- Se puede probar que el árbol de recursión de $\mathit{fibo}(n)$ tiene exactamente $2\mathit{fibo}(n+1)-1$ nodos. Como $\mathit{fibo}(n+1)=\Theta(\phi^n)$, esto demuestra que siempre hay una cantidad exponencial de llamados.
- Si bien no se puede calcular $\mathit{fibo}(n)$ en $\mathit{poly}(\log(n))$ por el tamaño del número, podemos platenar el problema de calcular $\mathit{fibo}(n) \mod r$ ¿Los algoritmos que vimos resuelven este problema en $\mathit{poly}(\log(n)+\log(r))$? No, porque hacen $O(n)$ cuentas.
- No obstante, se puede calcular usando la siguiente propiedad:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$$

- O bien, usando la siguiente identidad:

$$F_{2l} = F_l^2 + 2F_l F_{l-1}$$

$$F_{2l+1} = F_{l+1}^2 + F_l^2$$