

Programación Dinámica

Top Down

FCEyN UBA

Técnicas de Diseño Algorítmico, 1C 2024

Serie de Fibonacci

Cada número es la suma de los dos anteriores:

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

Esta ecuación nos da el *i*ésimo número en la serie:

$$f(0) = 1, f(1) = 1, f(2) = 2, f(3) = 3, f(4) = 5, f(5) = 8 \dots$$

Serie de Fibonacci

Cada número es la suma de los dos anteriores:

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

Esta ecuación nos da el *i*ésimo número en la serie:

$$f(0) = 1, f(1) = 1, f(2) = 2, f(3) = 3, f(4) = 5, f(5) = 8 \dots$$

¿Cómo podemos implementar esto en código?

Serie de Fibonacci

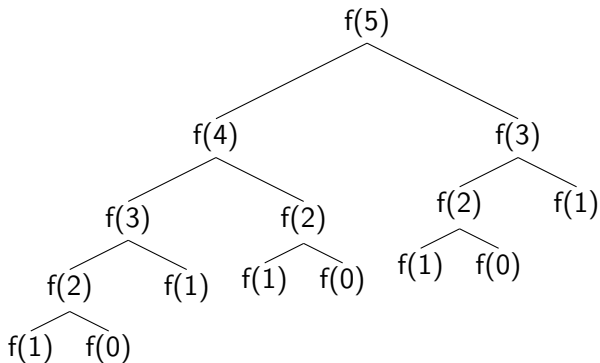
```
1: if  $i \leq 1$  then  
2:   return 1  
3: else  
4:   return  $\text{fibonacci}(i - 2) + \text{fibonacci}(i - 1)$   
5: end if
```

Serie de Fibonacci

```
1: if  $i \leq 1$  then  
2:   return 1  
3: else  
4:   return  $\text{fibonacci}(i - 2) + \text{fibonacci}(i - 1)$   
5: end if
```

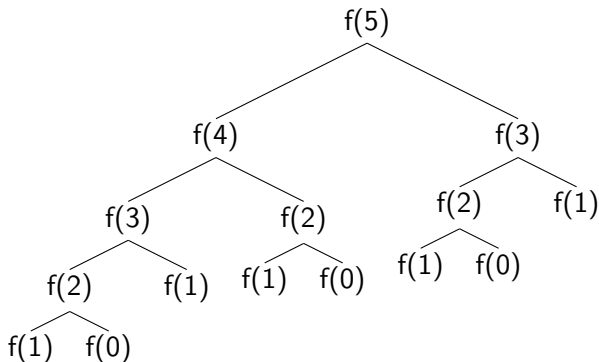
¿Cuál es la complejidad de este algoritmo?

Serie de Fibonacci • árbol de recursión



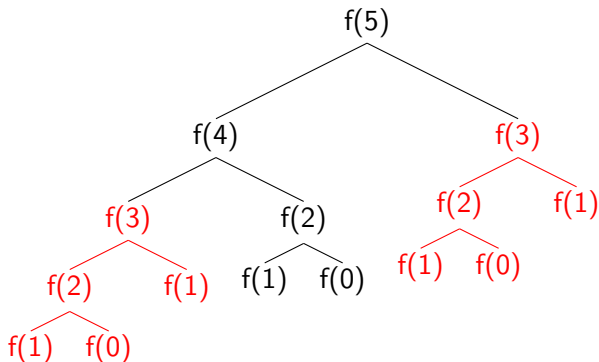
¿Cuántos llamados recursivos hacemos?

Serie de Fibonacci • árbol de recursión



¿Cuántos llamados recursivos hacemos? $\Omega(2^{\frac{n}{2}})$
Nuestro algoritmo corre en tiempo exponencial (☹)

Serie de Fibonacci



Una función matemática siempre devuelve lo mismo con la misma entrada, y estamos calculando más de una vez dos valores que sabemos de antemano que van a ser iguales. Esto es una **superposición de subproblemas**.

¿Cómo probamos que tenemos superposición de subproblemas?

Serie de Fibonacci

No alcanza con mostrar la superposición de subproblemas para una entrada, tenemos que mostrar que **siempre** van a haber más estados que llamados recursivos.

- ¿Cuántos llamados **distintos** hace la función con entrada 5?

Serie de Fibonacci

No alcanza con mostrar la superposición de subproblemas para una entrada, tenemos que mostrar que **siempre** van a haber más estados que llamados recursivos.

- ¿Cuántos llamados **distintos** hace la función con entrada 5? 5

Serie de Fibonacci

No alcanza con mostrar la superposición de subproblemas para una entrada, tenemos que mostrar que **siempre** van a haber más estados que llamados recursivos.

- ¿Cuántos llamados **distintos** hace la función con entrada 5? 5
- ¿Y con entrada n ?

Serie de Fibonacci

No alcanza con mostrar la superposición de subproblemas para una entrada, tenemos que mostrar que **siempre** van a haber más estados que llamados recursivos.

- ¿Cuántos llamados **distintos** hace la función con entrada 5? 5
- ¿Y con entrada n ? $\Theta(n)$

Serie de Fibonacci

No alcanza con mostrar la superposición de subproblemas para una entrada, tenemos que mostrar que **siempre** van a haber más estados que llamados recursivos.

- ¿Cuántos llamados **distintos** hace la función con entrada 5? 5
- ¿Y con entrada n ? $\Theta(n)$

Antes dijimos que hacemos $\Omega(2^{\frac{n}{2}})$ llamados recursivos.

$n \ll 2^{\frac{n}{2}} \Rightarrow$ **siempre** hay superposición de subproblemas.

Entonces sabemos que vale la pena aplicar programación dinámica.

Serie de Fibonacci

Nos interesaría resolver todos los subproblemas ya calculados en $O(1)$.
¿Cómo podemos hacerlo?

Serie de Fibonacci

Nos interesaría resolver todos los subproblemas ya calculados en $O(1)$.

¿Cómo podemos hacerlo?

Con una **estructura de memorización**.

Vamos a guardar todos los valores ya calculados en una estructura y leerlos cuando los volvamos a necesitar.

¿Qué estructura podemos usar en este caso?

Serie de Fibonacci

Nos interesaría resolver todos los subproblemas ya calculados en $O(1)$.

¿Cómo podemos hacerlo?

Con una **estructura de memorización**.

Vamos a guardar todos los valores ya calculados en una estructura y leerlos cuando los volvamos a necesitar.

¿Qué estructura podemos usar en este caso?

Un vector $M \in \mathbb{N}^n$ que guarde $M[i] = f(i)$.

Serie de Fibonacci

¿Cómo cambia nuestro algoritmo?

Serie de Fibonacci

¿Cómo cambia nuestro algoritmo?

Sea $M = (\perp, \dots, \perp) \in \mathbb{N}^n$

```
1: if  $i \leq 1$  then  
2:   return 1  
3: end if  
4:  
5: if  $M[i] \neq \perp$  then  
6:   return  $M[i]$   
7: end if  
8:  
9:  $M[i] := fibonacci(i - 2) + fibonacci(i - 1)$   
10: return  $M[i]$ 
```

¿Cuál es la nueva complejidad de nuestro algoritmo?

- Si llamamos a la función con una entrada que ya fue calculada antes, la respuesta va a estar en M , y se va a resolver en $O(1)$.
- Entonces no puede haber más llamados no triviales que estados.
- Cada llamado se resuelve internamente en $O(1)$

Entonces la nueva complejidad de nuestro algoritmo es la cantidad de estados multiplicada por la complejidad de cada llamado.

$$O(n) \cdot O(1) = O(n)$$

Pasamos de una complejidad exponencial a una lineal. (☺)

¿Pagamos algo por esta mejora en complejidad temporal?

¿Pagamos algo por esta mejora en complejidad temporal?

Estamos guardando más información en la memoria.

Es decir, aumentó nuestra **complejidad espacial**

Ahora necesitamos guardar un vector de $\Theta(n)$ elementos.

Enunciado

- Tenemos un CD con capacidad P y una lista de N canciones, donde la canción i tiene un peso p_i .
- Queremos saber cuánto es lo máximo que podemos llenar el CD de canciones sin pasarnos del peso máximo.
- No nos importa cuántas o qué canciones usamos, solo nos importa llenar lo más posible el disco.

Problema del CD

Con backtracking podemos llegar a una solución matemática de este tipo:

$$cd(i, k) = \begin{cases} -\infty & \text{si } i = N \wedge k < 0 \\ 0 & \text{si } i = N \wedge k \geq 0 \\ \max\{cd(i+1, k), cd(i+1, k - p_i) + p_i\} & \text{c.c.} \end{cases}$$

Y un algoritmo que corría en tiempo $\Theta(2^N)$.

¿Tenemos una superposición de problemas?

Pasos para evaluar superposición de subproblemas

Paso 1. Llamados recursivos

Vimos antes que el árbol de recursión de nuestro algoritmo sin podas es un árbol binario completo de altura N .

Entonces hacemos $\Omega(2^N)$ llamados recursivos.

Pasos para evaluar superposición de subproblemas

Paso 1. Llamados recursivos

Vimos antes que el árbol de recursión de nuestro algoritmo sin podas es un árbol binario completo de altura N .

Entonces hacemos $\Omega(2^N)$ llamados recursivos.

Paso 2. Cantidad de subproblemas a resolver

- Tenemos dos parámetros: i y k .
- $i \in [0, N]$
- $k \in [0, P]$
- Entonces solo podemos resolver $O(NP)$ subproblemas distintos.

Pasos para evaluar superposición de subproblemas

Paso 1. Llamados recursivos

Vimos antes que el árbol de recursión de nuestro algoritmo sin podas es un árbol binario completo de altura N .

Entonces hacemos $\Omega(2^N)$ llamados recursivos.

Paso 2. Cantidad de subproblemas a resolver

- Tenemos dos parámetros: i y k .
- $i \in [0, N]$
- $k \in [0, P]$
- Entonces solo podemos resolver $O(NP)$ subproblemas distintos.

Paso 3. Comparar las cotas

¿Cuándo vale que $NP \ll 2^N$? Cuando $P < \frac{2^N}{N}$

En ese caso **tenemos superposición de subproblemas**

¿Qué estructura de memorización podemos usar para guardar los subproblemas ya resueltos?

Problema del CD

¿Qué estructura de memorización podemos usar para guardar los subproblemas ya resueltos?

La solución suele ser una matriz con una dimensión para cada parámetro de entrada.

En nuestro caso, una matriz M de $N \times P$ donde $M[i, k] = cd(i, k)$

Problema del CD

Código por backtracking:

```
1: if  $i = N \wedge k < 0$  then  
2:   return  $-\infty$   
3: end if  
4: if  $i = N \wedge k \geq 0$  then  
5:   return 0  
6: end if  
7:  
8:  $\text{sinAgregar} := \text{cd}(i + 1, k)$   
9:  $\text{agregando} := \text{cd}(i + 1, k - p_i) + p_i$   
10: return  $\max\{\text{sinAgregar}, \text{agregando}\}$ 
```

¿Cómo lo modificamos para usar nuestra estructura de memorización?

Problema del CD

Código por programación dinámica:

```
1: if  $i = N \wedge k < 0$  then  
2:   return  $-\infty$   
3: end if  
4: if  $i = N \wedge k \geq 0$  then  
5:   return 0  
6: end if  
7: if  $M[i, k] \neq \perp$  then  
8:   return  $M[i, k]$   
9: end if  
10:  
11:  $\text{sinAgregar} := cd(i + 1, k)$   
12:  $\text{agregando} := cd(i + 1, k - p_i) + p_i$   
13:  $M[i, k] := \max\{\text{sinAgregar}, \text{agregando}\}$   
14: return  $M[i, k]$ 
```

- ¿Cuál es la nueva complejidad temporal de nuestro algoritmo?

Problema del CD

- ¿Cuál es la nueva complejidad temporal de nuestro algoritmo?
Tenemos $O(NP)$ llamados no triviales que se resuelven internamente en $O(1)$. Entonces es $O(NP) \cdot O(1) = O(NP)$.
- ¿Cuál es la nueva complejidad espacial de nuestro algoritmo?

Problema del CD

- ¿Cuál es la nueva complejidad temporal de nuestro algoritmo?
Tenemos $O(NP)$ llamados no triviales que se resuelven internamente en $O(1)$. Entonces es $O(NP) \cdot O(1) = O(NP)$.
- ¿Cuál es la nueva complejidad espacial de nuestro algoritmo? $O(NP)$
- ¿El algoritmo corre en tiempo polinomial?

Problema del CD

- ¿Cuál es la nueva complejidad temporal de nuestro algoritmo?
Tenemos $O(NP)$ llamados no triviales que se resuelven internamente en $O(1)$. Entonces es $O(NP) \cdot O(1) = O(NP)$.
- ¿Cuál es la nueva complejidad espacial de nuestro algoritmo? $O(NP)$
- ¿El algoritmo corre en tiempo polinomial? No.

P no depende del tamaño de la entrada, sino que depende del **valor numérico**. Por eso nuestro nuevo tiempo es **pseudopolinomial**.

DEMO

¿Y si el enunciado además del mayor peso posible a llenar nos preguntara **qué** canciones usamos?

¿Podemos aprovechar nuestra estructura de memorización?

Reconstruyendo la solución

- Si queremos saber si usamos o no la canción 0, nos podríamos fijar si la mejor solución usándola es mejor que la mejor solución que no la usa. ($cd(1, P - p_i) + p_i >? cd(1, P)$)
- Si nos conviene usar la primer canción, podemos hacer lo mismo siguiendo por la rama que la usa, si no nos conviene seguimos por la rama que no la usa.
- En general, recorreremos nuestro árbol de recursión buscando el mejor camino posible en cada paso.
- Todos estos cálculos ya los hicimos en el algoritmo, así que están en M y los podemos hacer en $O(1)$.

Enunciado

Tenemos **N** días de vacaciones, donde podemos hacer actividades:

- Hay dos actividades distintas: gimnasio y competencias de programación.
- Tenemos un calendario que nos dice por cada día si podemos hacer ninguna, alguna, o ambas.





En un día podemos:

- Descansar
- Hacer una actividad disponible, siempre que no la hayamos hecho el día anterior.

Buscamos hacer la mayor cantidad de actividades posibles.

Calendario de ejemplo

Entrada de ejemplo ($N = 4$)

Día 1	Día 2	Día 3	Día 4
			
			

Estas son las actividades que **es posible** hacer cada día.

- ¿Qué soluciones son válidas para este ejemplo?
- ¿Qué soluciones son inválidas?
- ¿Cuál sería una solución óptima?

Solución por backtracking

Tratemos de dividirlo en subproblemas idénticos, así podemos formular una función recursiva.

- Queremos pararnos en un día, elegir una actividad, y después resolver el subproblema del resto del calendario habiendo hecho esa actividad hoy.
- ¿Qué información necesitamos saber para saber las actividades posibles?

Solución por backtracking

Tratemos de dividirlo en subproblemas idénticos, así podemos formular una función recursiva.

- Queremos pararnos en un día, elegir una actividad, y después resolver el subproblema del resto del calendario habiendo hecho esa actividad hoy.
- ¿Qué información necesitamos saber para saber las actividades posibles?

El **día de hoy**, y la **actividad que hicimos ayer**.

Solución por backtracking

- Parados en el día 0, podemos ver qué pasa si hacemos cada una de las actividades posibles o descansamos.
- La mayor cantidad de actividades que podemos hacer en todo el viaje es la mayor cantidad que podemos hacer a partir del segundo día, habiendo hecho cualquiera de las actividades posibles el primero.
- En general, para el día i queremos ver cuál resulta mejor de todas las posibles actividades que podemos hacer ese día.

¿Qué actividades están disponibles cada día?

¿Qué pasa si ayer fui al gimnasio? ¿Y si descansé?

Vamos a hacer recursión sobre los días, del 0 a N

- ¿Qué parámetros toma nuestra función?

Vamos a hacer recursión sobre los días, del 0 a N

- ¿Qué parámetros toma nuestra función? día y últimaActividad

Vamos a hacer recursión sobre los días, del 0 a N

- ¿Qué parámetros toma nuestra función? día y últimaActividad
- ¿Qué devuelve la función?

Vamos a hacer recursión sobre los días, del 0 a N

- ¿Qué parámetros toma nuestra función? día y últimaActividad
- ¿Qué devuelve la función? el mayor número de actividades posible

Vamos a hacer recursión sobre los días, del 0 a N

- ¿Qué parámetros toma nuestra función? día y últimaActividad
- ¿Qué devuelve la función? el mayor número de actividades posible
- ¿Cuál es nuestro caso base?

Vamos a hacer recursión sobre los días, del 0 a N

- ¿Qué parámetros toma nuestra función? día y últimaActividad
- ¿Qué devuelve la función? el mayor número de actividades posible
- ¿Cuál es nuestro caso base? día = N

Vamos a hacer recursión sobre los días, del 0 a N

- ¿Qué parámetros toma nuestra función? día y últimaActividad
- ¿Qué devuelve la función? el mayor número de actividades posible
- ¿Cuál es nuestro caso base? día = N
- ¿Qué devuelve el caso base?

Vamos a hacer recursión sobre los días, del 0 a N

- ¿Qué parámetros toma nuestra función? día y últimaActividad
- ¿Qué devuelve la función? el mayor número de actividades posible
- ¿Cuál es nuestro caso base? día = N
- ¿Qué devuelve el caso base? 0

Con esta información, ¿cómo puede ser una función matemática?

Posible función matemática

$$f(dia, ultAct) = \begin{cases} 0 & dia = N \\ \max\{desc(dia, ultAct), gym(dia, ultAct), \\ comp(dia, ultAct)\} & c.c. \end{cases}$$

Donde:

$$desc(dia, ultAct) = f(dia + 1, DESC)$$

$$gym(dia, ultAct) = \begin{cases} f(dia + 1, GYM) + 1 & hayGym(dia) \wedge ultAct \neq GYM \\ -\infty & c.c. \end{cases}$$

$$comp(dia, ultAct) = \begin{cases} f(dia + 1, COM) + 1 & hayComp(dia) \wedge ultAct \neq COM \\ -\infty & c.c. \end{cases}$$

Llamando $f(0, DESC)$ para resolver el problema.

Superposición de problemas

Llamados recursivos en el peor caso

- Siempre llegamos al día N , y avanzamos de a uno. Entonces nuestro árbol de recursión tiene altura N .
- En el peor caso siempre podemos hacer todas las actividades, entonces cada función hace 2 llamados recursivos.

El algoritmo hace $\Omega(2^N)$ llamados.

Cantidad de subproblemas distintos

- $dia \in [0, N]$
- $ultAct \in \{DESC, GYM, COM\}$

Tenemos $O(3N) = O(N)$ estados.

¿Cuándo es menor la cantidad de estados?

Siempre. $N \ll 2^N$.

Usamos una estructura de memorización $M \in \mathbb{N}^{N \times 3}$ donde $M[\text{dia}][\text{ultAct}] = f(\text{dia}, \text{ultAct})$.

- Nueva complejidad temporal: $O(N)$ (☺)
- Nueva complejidad espacial: $O(N)$

Enunciado

Julio César tiene una imponente legión de **patos** y de **dodos**, que quiere poner en fila.

- Tiene P patos y D dodos.
- Como los animales se llevan mal en grupos, no puede haber más de MP patos más de MD dodos en fila seguidos.

Queremos saber **todas las combinaciones** posibles para formar las tropas.

Ejemplo de legión



$P = 3, MP = 2$



$D = 2, MD = 1$

Definición recursiva

- La cantidad de formas posibles de ordenar la legión es la cantidad de formas posibles poniendo primero un pato más la cantidad de formas posibles poniendo primero un dodo.
- Después de poner un pato, nos quedan $P-1$ patos a poner, y solo podemos poner $MP-1$ patos seguidos después de ese.
- Después de poner un dodo, nos quedan $D-1$ patos a poner, y solo podemos poner $MD-1$ patos seguidos después de ese.

Definición recursiva

$$f(n_P, n_D, k_P, k_D) = \begin{cases} 1 & (n_P + n_D) = 0 \\ \text{ponerP}(n_P, n_D, k_P, k_D) + & \\ \text{ponerD}(n_P, n_D, k_P, k_D) & c.c. \end{cases}$$

¿Cómo definimos ponerPy ponerD?

Definición recursiva

$$f(n_P, n_D, k_P, k_D) = \begin{cases} 1 & (n_P + n_D) = 0 \\ \text{ponerP}(n_P, n_D, k_P, k_D) + \text{ponerD}(n_P, n_D, k_P, k_D) & \text{c.c.} \end{cases}$$

¿Cómo definimos ponerP y ponerD?

$$\text{ponerP}(n_P, n_D, k_P, k_D) = \begin{cases} f(n_P - 1, n_D, k_P - 1, MD) & n_P > 0 \wedge k_P > 0 \\ 0 & \text{c.c.} \end{cases}$$

$$\text{ponerD}(n_P, n_D, k_P, k_D) = \begin{cases} f(n_P, n_D - 1, MP, k_D - 1) & n_D > 0 \wedge k_D > 0 \\ 0 & \text{c.c.} \end{cases}$$

¡PROBLEMA!

El juez nos dice que la complejidad espacial máxima es $O(PD)$

Si aplicamos programación dinámica como veníamos haciendo, nos va a quedar algo de $O(P \cdot D \cdot MP \cdot MD)$.

Vamos a tener que buscar formas de reducir el estado.

¿Hace falta guardar k_P y k_D a la vez?

- Nunca va a pasarnos que usemos las dos, porque solo importa mantener el conteo de la última tropa que viene repitiéndose. Cambiemos el nombre de la variable entonces a solo k .
- Para eso vamos a necesitar guardar cuál fue la última tropa que se puso en la fila.

$$f(n_P, n_D, k, ultima) = \begin{cases} 1 & (n_P + n_D) = 0 \\ ponerP(n_P, n_D, k, ultima) + \\ ponerD(n_P, n_D, k, ultima) & c.c. \end{cases}$$

$$ponerP(n_P, n_D, k, ultima) = \begin{cases} f(n_P - 1, n_D, k - 1, PATO) & ultima = PATO \wedge k, n_P > 0 \\ f(n_P - 1, n_D, MP - 1, PATO) & ultima = DODO \wedge n_P > 0 \\ 0 & c.c. \end{cases}$$

$$ponerD(n_P, n_D, k, ultima) = \begin{cases} f(n_P, n_D - 1, k - 1, DODO) & ultima = DODO \wedge k, k_D > 0 \\ f(n_P, n_D - 1, MD - 1, DODO) & ultima = PATO \wedge k_D > 0 \\ 0 & c.c. \end{cases}$$

Pasos grandes

¿Hace falta que cada paso recursivo cambie de a una tropa?

En lugar de ir agregando de a una sola tropa, podemos en cada paso recursivo ver que pasa si ponemos 1, 2, ..., etc. tropas del mismo tipo.

¿Qué parámetro nos deja sacar este cambio?

$$f(n_P, n_D, ultima) = \begin{cases} 1 & (n_P + n_D) = 0 \\ ponerP(n_P, n_D, ultima) + \\ ponerD(n_P, n_D, ultima) & c.c. \end{cases}$$

$$ponerP(n_P, n_D, ultima) = \begin{cases} \sum_{i=1}^{\min(n_P, MP)} f(n_P - i, n_D, PATO) & ult = DODO \\ 0 & c.c. \end{cases}$$

$$ponerD(n_P, n_D, k, ultima) = \begin{cases} \sum_{i=1}^{\min(n_D, MD)} f(n_P, n_D - i, DODO) & ult = PATO \\ 0 & c.c. \end{cases}$$

Llamando $f(P, D, PATO) + f(P, D, DODO)$ para resolver el problema.

Superposición de problemas

Llamados recursivos en el peor caso

- El árbol tiene $P+D$ de altura
- En cada paso hay por lo menos 2 llamados

El algoritmo hace $\Omega(2^{P+D})$ llamados.

Cantidad de subproblemas distintos

- $n_P \in [0, P]$
- $n_D \in [0, D]$
- $ultimo \in \{PATO, DODO\}$

Tenemos $O(2^{PD}) = O(PD)$ estados.

Complejidad de la función

- Hay $O(PD)$ estados.
- El costo interno de la función en peor caso ahora es el máximo tamaño de la sumatoria.
 $O(\max\{MP, MD\})$

Entonces la complejidad temporal del algoritmo es $O(PD \cdot \max(MP, MD))$