

UNIVERSIDAD NACIONAL DE INGENIERÍA

Análisis y Diseño de Algoritmos

Más programación dinámica

El DAG de subproblemas: caminos, intervalos y árboles

Gerson Garrido

Semana 9

Hoy

Sesión A

1. Los subproblemas forman un DAG
2. Camino más largo en un DAG
3. Subsecuencia creciente más larga
4. Cadena de matrices: rangos $i..j$
5. El orden que lee casillas vacías
6. Árboles binarios de búsqueda óptimos

Sesión B

7. Un árbol tiene un orden natural
8. Conjunto independiente: dos estados
9. Reconstruir de arriba hacia abajo
10. Cobertura de vértices
11. Evitar la recursión profunda

> **Lectura**
>
> Apuntes de la semana 9. Código en `code/week-09`.

SECCIÓN 01

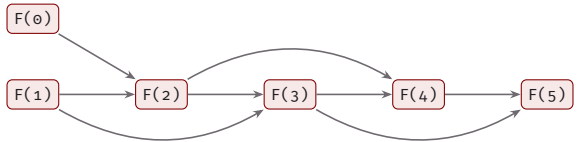
Los subproblemas forman un DAG

01



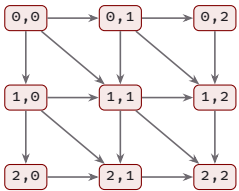
El grafo de subproblemas

Grafo de subproblemas ***(subproblem graph)***
Un nodo por subproblema. Una arista $A \to B$ si la recurrencia de B **lee** a A.



Fibonacci con $n = 5$: 6 nodos y 8 aristas. Cada $F(k)$ lee a los dos anteriores.

La distancia de edición es una cuadrícula



- (i, j) lee tres casillas.
- $(n + 1)(m + 1)$ nodos y $3nm + n + m$ aristas.
- `casa`, `cosas`: 30 nodos, 69 aristas.
- Por eso cuesta $\Theta(nm)$.

Costo

Si leer una casilla cuesta $O(1)$, llenar todo cuesta $O(V + E)$ en este grafo.

Lo que ya conocíamos, leído en el grafo

- **Tiene que ser un DAG.** Un ciclo diría que *A* hace falta para calcular *A*.
- **Un orden de cálculo válido es un orden topológico.** Toda arista va de lo que se lee a lo que lo lee. Llenar por filas es elegir uno de ellos.
- **La memoización es un DFS** que parte de la respuesta y sigue las flechas al revés. La tabla hace de arreglo de visitados.

Lo nuevo de esta semana

El orden pasa a ser parte del problema. Elegirlo es elegir un orden topológico.

¿Qué forma tiene una solución?

Cada opción lee un subproblema: un camino

- Distancia de edición: costo más una casilla.
- Mochila, LIS, camino más largo.

Una opción combina varios: no es un camino

- Cadena de matrices: suma las dos partes.
- Árboles: suma todos los hijos.

build/dag_demo

```
Distancia de edición, casa -> cosas: 30 nodos, 69 aristas
  distancia, como camino en ese DAG: 2
  el camino, en casillas (i, j): (0,0) (1,1) (2,2) (3,3) (4,4) (4,5)

Mochila de la semana 7, capacidad 9, como camino más largo: 13
```

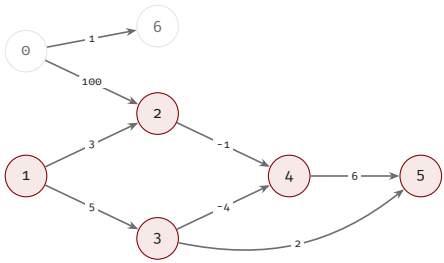
Los mismos 2 y 13 de la semana 7, con un solo programa: el del camino más largo.

SECCIÓN 02

Camino más largo en un DAG

02

El problema, y un greedy que falla



Origen 1. Se busca el mayor peso de un camino hasta cada nodo. Los pesos pueden ser negativos.

- Salir por la arista más pesada: `1 3 5`, largo **7**.
- Lo mejor: `1 2 4 5`, largo **8**, con una arista negativa.

Subproblema y recurrencia

Subproblema

$L(v)$ es el mayor peso de un camino de s a v, y es $-\infty$ si no hay ninguno.

Se mira la última arista del camino:

$$L(s) = 0, \qquad L(v) = \max\{\, L(u) + w(u,v) : u \to v,\ u \text{ se alcanza desde } s \,\}$$

$L(v) = -\infty$ si no hay tal u.

- Un máximo entre opciones, y cada opción lee una sola casilla: es un camino.
- El grafo de subproblemas es el propio grafo de entrada.
- El orden: uno topológico, el de la semana 6.

Por qué es correcto

Técnica: inducción sobre un orden topológico. Cada u que se lee va antes que v.

- **Un hecho previo:** en un DAG ningún recorrido repite un nodo. Todo recorrido es un camino.
- $v = s$: solo el camino vacío. Si una arista entrara desde un nodo alcanzable, habría un ciclo.
- v sin caminos desde s: ningún u con $u \to v$ se alcanza. Vale $-\infty$.
- ($\leq$) Un camino óptimo a v tiene una última arista $u \to v$. Lo anterior es un camino a u, de peso a lo sumo $OPT(u)$.
- ($\geq$) Un camino óptimo a u más la arista $u \to v$ es un recorrido, y por el hecho previo es un camino.

Lo que no se usa

El signo de los pesos. Al llegar a v ya se revisó todo lo que le entra.

El algoritmo: $O(V + E)$

preparar

```
alcanzado ← n valores falso
L ← n ceros
pred ← n valores -1
alcanzado[s] ← verdadero
orden ← orden_topologico(
      grafo, n)
```

recorrer

```
para t desde 0 hasta n - 1 hacer
    u ← orden[t]
    si alcanzado[u] entonces
        para cada arco (u, v, w) de la lista de u hacer
            si no alcanzado[v] o L[u] + w > L[v] entonces
                alcanzado[v] ← verdadero
                L[v] ← L[u] + w
                pred[v] ← u
            fin si
        fin para
    fin si
fin para
devolver L, alcanzado, pred
```

Lo que hace el programa

- La recurrencia mira lo que *entra*, y el programa empuja lo que *sale*. Es lo mismo: al llegar a v ya pasaron todos los que le llegan.
- El orden topológico por DFS cuesta $O(V + E)$, y luego cada arista se mira una vez.
- El camino sale siguiendo `pred` hacia atrás, desde el destino.
- Con mínimo en vez de máximo: camino más corto en un DAG. También con el signo de los pesos cambiado.

Un nodo sin alcanzar no vale cero



El nodo 0 no se alcanza, y tiene una arista de 100 hacia el 2.

Error común

Con todo en 0, cualquier nodo es otro origen: da 105 para el nodo 5, y es 8. Con `LLONG_MIN` y un peso negativo: overflow.

Solución: un arreglo aparte, y no salir nunca de un nodo sin alcanzar.

Sin DAG, no queda un orden en que calcular. El camino más largo es difícil: semana 14.

SECCIÓN 03

Subsecuencia creciente más larga

03

El problema, y el greedy que falla

Subsecuencia creciente más larga *(longest increasing subsequence, LIS)*
Posiciones $p_1 < \cdots < p_k$ con $a[p_1] < \cdots < a[p_k]$, **estrictamente**, y con k máximo.

build/lis_demo

```
Tomar todo lo que sea mayor que lo último tomado
  5 1 2 3: greedy 1, óptimo 3
  1 7 2 3 4: greedy 2, óptimo 4
```

Un elemento grande tomado temprano cierra el camino, incluso si se empezó bien.

Un subproblema que sabe dónde termina

"Lo mejor con los primeros i" no sirve: para agregar $a[i]$ hay que saber en qué valor termina. Es el error de la mochila sin capacidad.

Subproblema

$E(i)$ es el largo de la subsecuencia creciente más larga que **termina justo en** i. La respuesta es $\max_i E(i)$, no $E(n-1)$.

$$E(i) = 1 + \max\{\, E(j) : j < i,\ a[j] < a[i] \,\}, \qquad E(i) = 1 \text{ si no hay tal } j$$

- ($\leq$) Se quita el último de una solución: queda una que termina justo en el penúltimo, j.
- ($\geq$) A una que termina justo en j, con $a[j] < a[i]$, se le puede agregar $a[i]$.
- Cada opción lee una casilla: es un camino más largo en el DAG $j \to i$.

La tabla: $\Theta(n^2)$

cuadrático

```
algoritmo lis_cuadratico(a, n)
    E ← arreglo de n unos
    pred ← arreglo de n valores -1
    para i desde 0 hasta n - 1 hacer
        para j desde 0 hasta i - 1 hacer
            si a[j] < a[i] y E[j] + 1 > E[i]
      entonces
                E[i] ← E[j] + 1
                pred[i] ← j
            fin si
        fin para
    fin para
    devolver E, pred
fin algoritmo
```

build/lis_demo

```
   i:  0  1  2  3  4  5  6  7
a[i]:  3  1  4  1  5  9  2  6
E[i]:  1  1  2  1  3  4  2  4
pred: -1 -1  0 -1  2  4  1  4
Largo: 4
```

Desde la posición 5: `pred` da 4, 2 y 0. Al revés, `3 4 5 9`.

Desde la posición 7: `3 4 5 6`. Hay más de una respuesta.

La idea para mejorar: por cada largo, un número

Al procesar $a[i]$ se pregunta: ¿cuál es la más larga que puedo extender?

Entre dos subsecuencias del mismo largo, **solo importa la que termina en el valor más pequeño**: se puede extender con todo lo que se pueda extender la otra.

> **Arreglo de finales**
>
> Para el prefijo $a[0..i)$, $\mathrm{finales}[k]$ es el **menor último valor** de una subsecuencia creciente de largo $k+1$.

Es estrictamente creciente. La de largo $k+2$ que termina en $\mathrm{finales}[k+1]$ tiene un penúltimo menor que él, y ese penúltimo cierra una de largo $k+1$:

$$\mathrm{finales}[k] \leq \text{penúltimo} < \mathrm{finales}[k+1].$$

Como está ordenado, sirve la búsqueda binaria.

El algoritmo: $O(n \log n)$

con búsqueda binaria

```
algoritmo lis_rapido(a, n)
    finales ← lista vacía
    para i desde 0 hasta n - 1 hacer
        p ← primera posición de finales con finales[p] ≥ a[i], o su largo si no hay
        si p = largo de finales entonces
            agregar a[i] al final de finales
        sino
            finales[p] ← a[i]
        fin si
    fin para
    devolver largo de finales
fin algoritmo
```

En C++, “el primero mayor o igual” es `std::lower_bound`. En la semana 7, con intervalos semiabiertos, tocaba `upper_bound`. Aquí no.

Paso a paso: 3 1 4 1 5 9 2 6

i	x	posición	finales después
0	3	0	3
1	1	0	1
2	4	1	1 4
3	1	0	1 4
4	5	2	1 4 5
5	9	3	1 4 5 9
6	2	1	1 2 5 9
7	6	3	1 2 5 6

Largo 4. El arreglo final, 1 2 5 6, tampoco es la respuesta en general: lo vemos en un momento.

Por qué es correcto: el invariante

Sea $x = a[i]$, y p la primera posición con $\text{finales}[p] \geq x$.

- Las subsecuencias nuevas terminan en x. Una de largo $\ell + 1$ es una de largo ℓ con último $< x$, más x.
- Esa existe si y solo si $\text{finales}[\ell - 1] < x$, es decir, si $\ell - 1 < p$. Los largos nuevos son **de 1 a** $p + 1$.

Casilla por casilla:

- $k < p$: hay nuevas de ese largo, pero terminan en $x > \text{finales}[k]$. No cambia.
- $k = p$: hay una nueva, que termina en x. Antes el mínimo era $\geq x$, o ese largo no existía. Ahora es x.
- $k > p$: no hay nuevas de ese largo. No cambia.

Conclusión

La definición se mantiene. Al terminar, el largo del arreglo es el de la LIS.

`lower_bound` o `upper_bound`, y lo que no son los finales

build/lis_demo

```
lower_bound y upper_bound
  2 2 2: con lower_bound 1, con upper_bound 3
  1 3 3 5: con lower_bound 3, con upper_bound 4
```

Error común

Con `upper_bound` se resuelve otro problema: la más larga que **no decrece**. Sin ninguna señal.

build/lis_demo

```
Los finales no son una subsecuencia
  arreglo: 2 5 3 7 1
  finales: 1 3 7
  reconstruida: 2 3 7
```

Error común

En el arreglo, el 1 está después del 3 y del 7. Cada casilla habla de una subsecuencia distinta. Lo único seguro es el largo.

Reconstruir: guardar índices y predecesores

con la lista

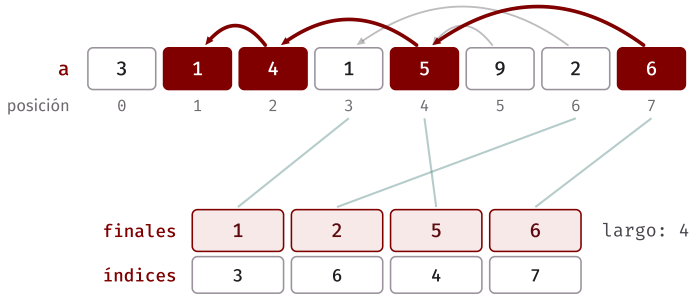
```
para i desde 0 hasta n - 1 hacer
    p ← primera posición con
  finales[p] ≥ a[i]
    si p > 0 entonces
        pred[i] ← indice_final[p -
   1]
    fin si
    si p = largo de finales
  entonces
        agregar a[i] a finales
        agregar i a indice_final
    sino
        finales[p] ← a[i]
        indice_final[p] ← i
    fin si
fin para
```

build/lis_demo, extracto

```
i  x  posición  finales después       índices    pred[i]
...
2  4          1  1 4                  1 2          1
...
4  5          2  1 4 5                3 2 4        2
...
7  6          3  1 2 5 6              3 6 4 7      4
```

Del índice 7: 6, luego 5, luego 4, luego 1. Da 1 4 5 6, aunque al final los índices digan 3 6 4 7. **Por inducción:** si $a[i]$ entra en p, su cadena tiene $p+1$ elementos, y va bajando en posición y en valor.

Los finales se cruzan; la cadena de `pred`, no



Con pred, desde el último índice: posición 7 (el 6), 4 (el 5), 2 (el 4), 1 (el 1).

Al revés: 1 4 5 6, una subsecuencia creciente de largo 4.

Midiendo

```
make bench
```

```
Subsecuencia creciente más larga, valores al azar, menor de 3 intentos
        n   largo   tabla ms   finales ms
    10000     196      203.7          0.4
    20000     270      837.9          0.8
    40000     384     3309.8          2.1
    80000     544    11331.4          4.8
  1000000    1983          -         47.5
  2000000    2819          -        109.1
  4000000    3983          -        217.6
```

- Con `-O2`, en el contenedor del curso, con la máquina compartida. Los tiempos dependen de la máquina.
- Al duplicar n, la tabla cuesta cerca de 4 veces más en las dos primeras duplicaciones, y 3,4 en la tercera.
- Con 80 000 elementos: más de 11 segundos contra menos de 5 milisegundos.

SECCIÓN 04

Intervalos: cadena de matrices

04

El orden de multiplicar cambia el costo

$p \times q$ por $q \times r$ hace $p \cdot q \cdot r$ multiplicaciones de números. El resultado es el mismo. El costo, no.

build/chain_demo

```
Matrices: 10x30, 30x5, 5x60
  i\j       0       1       2
    0       0    1500    4500
    1       .       0    9000
    2       .       .       0
Mínimo: 4500, con ((A0 A1) A2)
  (A0 A1) A2: 4500
  A0 (A1 A2): 27000
```

- A_i mide $d[i] \times d[i+1]$: bastan $n+1$ números.
- Hay tantas formas como el número de Catalan $C(n-1)$. Con 20 matrices: 1 767 263 190.

Lo más barato primero también falla

build/chain_demo

```
Primero la multiplicación más barata
  matrices: 4x1, 1x3, 3x5
  greedy: 72, óptimo por fuerza bruta: 35
```

- El greedy hace $4 \cdot 1 \cdot 3 = 12$, y queda 4×3 por 3×5: 60 más. Total, 72.
- Si se hace primero $1 \cdot 3 \cdot 5 = 15$, queda 4×1 por 1×5: 20 más. Total, 35.
- Lo barato de ahora deja dimensiones grandes para después.

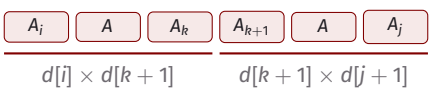
Subproblema: un rango

Se piensa en la **última** multiplicación: parte la cadena en dos pedazos contiguos, más cortos.

Subproblema

$M(i, j)$ es el mínimo de multiplicaciones de números para calcular $A_i \cdots A_j$, con las dos puntas incluidas. La respuesta es $M(0, n-1)$.

$$M(i,i) = 0, \qquad M(i,j) = \min_{i \le k < j} \big(M(i,k) + M(k+1,j) + d[i]\, d[k+1]\, d[j+1] \big)$$



Cada opción lee **dos** casillas. Es un árbol en el DAG, no un camino.

Por qué es correcto

Técnica: inducción sobre el largo $j - i + 1$. Caso base: una matriz cuesta 0.

- **Un hecho:** un producto de A_i a A_k siempre mide $d[i] \times d[k+1]$, sin importar cómo se calculó.
- ($\geq$) Toda forma de calcular $A_i \cdots A_j$ tiene una última multiplicación, en algún k. Antes calcula las dos partes sin mezclarlas. Cuesta al menos $OPT(i,k) + OPT(k+1,j) + d[i]\,d[k+1]\,d[j+1]$.
- ($\leq$) Para cada k, se calcula cada parte con su mejor forma y se multiplica al final. Esa forma cuesta exactamente la opción k.

La receta de la semana 7

Al minimizar, "toda solución queda cubierta" es $\geq$, y "cada opción se puede conseguir" es $\leq$.

Por largo creciente

cadena de matrices, solo el valor

```
M ← matriz de n por n, toda en cero
para largo desde 2 hasta n hacer
    para i desde 0 hasta n - largo hacer
        j ← i + largo - 1
        M[i][j] ← ∞
        para k desde i hasta j - 1 hacer
            costo ← M[i][k] + M[k + 1][j]
                    + d[i]·d[k + 1]·d[j + 1]
            M[i][j] ← mín(M[i][j], costo)
        fin para
    fin para
fin para
devolver M[0][n - 1]
```

- $M(i, j)$ lee $M(i, k)$ y $M(k + 1, j)$: rangos **más cortos**.
- $(n^3 - n)/6$ combinaciones (i, k, j). Con $n = 10$: 165.
- $\Theta(n^3)$ en tiempo, $\Theta(n^2)$ en memoria.
- Una tabla `corte`, con la mejor k, da la parentización: una recursión que abre dos ramas.

La tabla, por diagonales

Matrices 10×20, 20×5, 5×30, 30×2.

	0	1	2	3
0	0	1000	2500	900
1		0	3000	500
2			0	300
3				0

La esquina, $M(0, 3)$:

- $k = 0$: $0 + 500 + 400 = \mathbf{900}$
- $k = 1$: $1000 + 300 + 100 = 1400$
- $k = 2$: $2500 + 0 + 600 = 3100$

Resultado: `(A0 (A1 (A2 A3)))`.

El orden i, j ingenuo lee casillas sin llenar

	0	1	2	3
0	0	1000	?	
1		0	**vacía**	
2			0	
3				0

`i` de menor a mayor, y dentro `j` de menor a mayor: el orden de la semana 7.

- Primero se llena la fila 0.
- Pero $M[0][2]$ lee $M[1][2]$, que tiene $i = 1$, y se llena después.
- El programa la lee igual. Vale 0. No se cae, y no hay ninguna advertencia.

Corriéndolo

build/chain_demo

```
Orden ingenuo: i de menor a mayor, j de menor a mayor, con la tabla en ceros
  10x30, 30x5, 5x60
    M[0][2] lee M[1][2], vacía
    lecturas vacías: 1, respuesta: 4500, correcta: 4500
  10x20, 20x5, 5x30, 30x2
    M[0][2] lee M[1][2], vacía
    M[0][3] lee M[1][3], vacía
    M[0][3] lee M[2][3], vacía
    M[1][3] lee M[2][3], vacía
    lecturas vacías: 4, respuesta: 400, correcta: 900
```

Error común: un orden que lee una casilla antes de llenarla

Por suerte, el primero sale bien. El segundo cuenta gratis $A_1A_2A_3$, que cuesta 500.

¿Hay que llenar por largo?

No necesariamente. Hace falta un orden topológico del DAG, y hay varios.

- **Largo creciente:** todo lo que se lee es más corto. Es el más fácil de justificar.
- i **de mayor a menor,** j **de menor a mayor:** (i, k) tiene la misma fila y una j menor, y $(k+1, j)$ está en una fila anterior del recorrido.
- **Memoización:** la recursión descubre el orden sola, y paga las llamadas y la pila.

build/chain_demo

```
Otro orden válido, i de mayor a menor y j de menor a mayor: 900
```

Midiendo el costo cúbico

```
make bench
```

```
Cadena de matrices, dimensiones al azar entre 1 y 50, menor de 3 intentos
    n   combinaciones        ms
  200         1333300       2.0
  400        10666600      14.3   x7.0
  800        85333200     126.7   x8.9
 1600       682666400    1153.4   x9.1
```

- Con `-O2`, en el contenedor del curso, con la máquina compartida. Los tiempos dependen de la máquina.
- Al duplicar n, las combinaciones se multiplican casi exactamente por 8, y el tiempo, por algo entre 7 y 9,1.
- Tres dimensiones de 2000 ya dan 8×10^9: todo va en `long long`.

SECCIÓN 05

Árboles binarios de búsqueda óptimos

05

Si se sabe qué se busca más

Claves $k_0 < \cdots < k_{n-1}$ que no cambian, con frecuencias de búsqueda $f_0, \ldots, f_{n-1}$.

Costo de un árbol

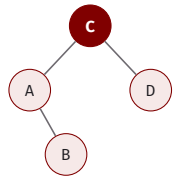
$\text{costo}(T) = \sum_i f_i \cdot \text{profundidad}(k_i)$, con **la raíz en profundidad 1**: cuántos nodos se visitan al buscar.

- Solo contamos **búsquedas exitosas**. CLRS también plantea las que fallan, con la misma idea.
- Se quiere el de mínimo costo: un **árbol binario de búsqueda óptimo** *(optimal binary search tree)*.
- Un subárbol siempre tiene un **rango contiguo** de claves, que puede estar vacío: rangos semiabiertos $[i, j)$.

Los greedy que fallan

build/bst_demo, extracto

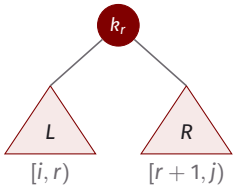
```
Claves y frecuencias: A=2 B=1 C=3 D=4
  ...
  árbol: C(A(- B) D)
  profundidades: A=2 B=3 C=1 D=2
  costo con la definición: 18
  por fuerza bruta: 18
  la más frecuente a la raíz: 20
  balanceado: 19
```



$2 \cdot 2 + 1 \cdot 3 + 3 \cdot 1 + 4 \cdot 2 = 18$

- **La más frecuente a la raíz:** D baja a A, B y C, que juntas pesan más que D.
- **Balanceado:** ignora las frecuencias. Pone a A debajo de B, aunque A se busca el doble.
- **Huffman** tampoco sirve: allá los símbolos pueden quedar en cualquier orden, y aquí no.

Por qué se suma toda la frecuencia del rango



Lema

$\text{costo}(T) = \text{costo}(L) + \text{costo}(R) + F(i,j)$, con $F(i,j) = \sum f$ del rango.

- Cada clave de L o de R queda un nivel más abajo: suma una vez su frecuencia.
- La raíz contribuye $f_r \cdot 1$: también una vez.
- Total: $F(i,j)$, sin importar la raíz.

Error común

Sin $F(i,j)$ da 0 siempre. Con solo f_r, todas cuentan a la misma profundidad.

Recurrencia y demostración

Subproblema

$C(i, j)$ es el mínimo costo de un árbol con las claves del rango $[i, j)$. La respuesta es $C(0, n)$.

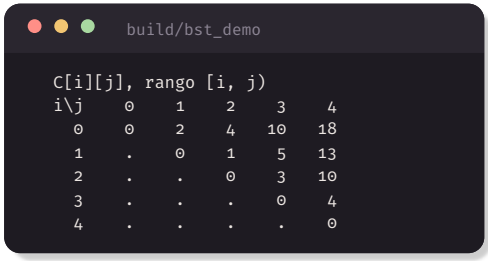
$$C(i, i) = 0, \qquad C(i, j) = F(i, j) + \min_{i \le r < j} \big(C(i, r) + C(r + 1, j)\big)$$

Técnica: inducción sobre el largo, con el lema.

- $(\ge)$ Todo árbol tiene una raíz k_r, y sus lados son árboles de $[i, r)$ y $[r + 1, j)$. Por el lema, cuesta al menos esa opción.
- $(\le)$ Para cada r, se arma el árbol con raíz k_r y los mejores árboles de cada lado. Es un árbol de búsqueda válido, y cuesta exactamente esa opción.

$F(i, j)$ con sumas acumuladas. Por largo creciente: $\Theta(n^3)$ en tiempo, $\Theta(n^2)$ en memoria.

La tabla, y la casilla de la respuesta



$F(0,4) = 2+1+3+4 = 10$.

Raíz	$C(0,r)$	$C(r+1,4)$	Suma
A	0	13	13
B	2	10	12
C	4	4	**8**
D	10	0	10

$C(0,4) = 8+10 = 18$, con raíz C. Las raíces de cada rango dan el árbol: es la misma forma de la cadena de matrices.

SECCIÓN 06

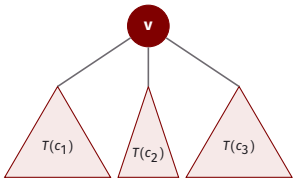
Programación dinámica sobre árboles

06



Un árbol tiene un orden natural

Primero los hijos. $T(v)$ es el subárbol de v: el nodo y todos sus descendientes.



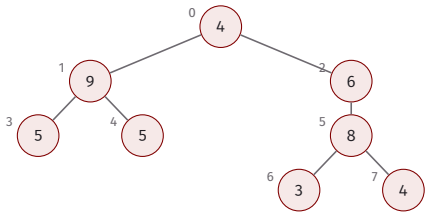
- Los subárboles de los hijos no comparten nodos.
- **No hay aristas entre ellos.** Para pasar de uno a otro hay que cruzar v.
- El DAG de subproblemas es el árbol, con flechas de hijo a padre.
- Un nodo combina a todos sus hijos: no es un camino.

Esa segunda propiedad es la que hace que todo funcione, y la que se pierde con ciclos.

Conjunto independiente de peso máximo

Conjunto independiente ***(independent set)***

Nodos entre los cuales no hay ninguna arista. Cada nodo tiene un valor, que puede ser negativo. El conjunto vacío vale 0.



Piensen en una empresa: nadie quiere ir a una reunión con su jefe directo. El número pequeño es el nodo, y el del círculo, su valor.

Ideas parecidas que fallan

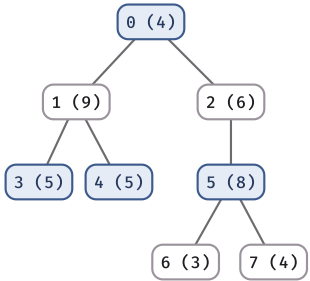
build/tree_demo, extracto

```
El árbol de arriba:
  fuerza bruta:              23
  dos estados:               23
  niveles alternos:          22
  mayor valor primero:       17
  ...
La cadena 10, 1, 1, 10:
  fuerza bruta:              20
  dos estados:               20
  niveles alternos:          11
  mayor valor primero:       20
  ...
Un jefe de valor 5 con tres personas de valor 2:
  fuerza bruta:              6
  dos estados:               6
  niveles alternos:          6
  mayor valor primero:       5
```

- **Niveles alternos:** siempre es independiente. Falla en la cadena: lo mejor son las dos puntas, de distinta paridad.
- **Mayor valor primero:** falla con el jefe. El 5 bloquea a tres de 2, que suman 6.
- En el primer árbol fallan las dos. El óptimo mezcla niveles: elige a los nodos 2, 3, 4, 6 y 7.

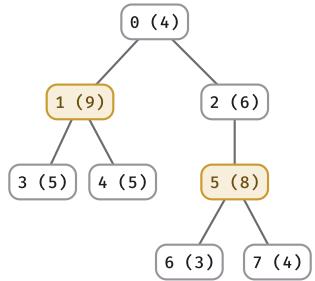
Lo que elige cada idea

Niveles alternos: 22



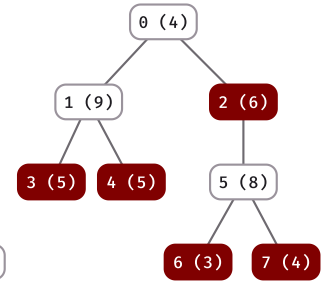
Los impares también dan 22.

Mayor valor primero: 17



El 9 y el 8 bloquean al resto.

Óptimo: 23



No respeta ningún nivel.

Cada nodo se escribe número (valor). Marcados, los que elige cada idea.

Dos estados por nodo

Para decidir sobre v hay que saber qué pasa con sus hijos, y eso depende de si v se eligió. Falta información: otra vez, se agrega al subproblema.

Subproblema

$\mathrm{Con}(v)$: lo mejor en $T(v)$ **conteniendo a** v. $\mathrm{Sin}(v)$: lo mejor en $T(v)$ **sin** v. La respuesta es $\max(\mathrm{Con}(\text{raíz}), \mathrm{Sin}(\text{raíz}))$.

$$\mathrm{Con}(v) = \mathrm{valor}(v) + \sum_{c \text{ hijo}} \mathrm{Sin}(c), \qquad \mathrm{Sin}(v) = \sum_{c \text{ hijo}} \max\big(\mathrm{Con}(c), \mathrm{Sin}(c)\big)$$

- Si v está, cada hijo queda fuera.
- Si v no está, **cada hijo queda libre**: puede estar o no.
- En una hoja: $\mathrm{Con}(v) = \mathrm{valor}(v)$ y $\mathrm{Sin}(v) = 0$. Cada nodo se mira una vez: $\Theta(n)$.

Por qué es correcto

Lema. $X \subseteq T(v)$ es independiente si y solo si cada $X \cap T(c_t)$ lo es, y si $v \in X$, ningún hijo está en X. *Las aristas de $T(v)$ son las de dentro de cada $T(c_t)$ y las de v con sus hijos. No hay más.*

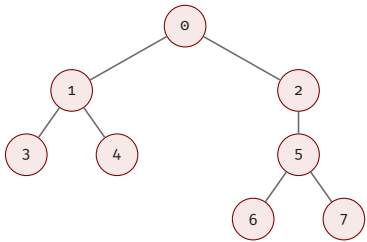
Teorema, por inducción sobre la altura de v:

- $\mathrm{Con}(v)$**.** Por el lema, es v más, en cada $T(c_t)$, un conjunto independiente sin c_t.
- Las elecciones en distintos hijos **no se estorban**. Maximizar una suma de partes independientes es maximizar cada parte: $\mathrm{Sin}(c_t)$.
- $\mathrm{Sin}(v)$**.** Por el lema, en cada $T(c_t)$ vale cualquier conjunto independiente. Lo mejor es $\max(\mathrm{Con}(c_t), \mathrm{Sin}(c_t))$.

Valores negativos

La demostración no usa el signo: no hace falta código aparte. Como $\mathrm{Sin}(v) \geq 0$, la respuesta nunca es negativa.

De abajo hacia arriba



	valor	con	sin
7	4	4	0
6	3	3	0
5	8	8	7
4	5	5	0
3	5	5	0
2	6	13	8
1	9	9	10
0	4	22	**23**

Raíz. Con ella: $4 + 10 + 8 = 22$. Sin ella: $\max(9, 10) + \max(13, 8) = 23$. Gana no elegirla.

Error común: un nodo sin elegir deja libres a sus hijos

build/tree_demo, extracto

```
La cadena 10, 1, 1, 10:
  fuerza bruta:              20
  dos estados:               20
  ...
  sin elegir obliga a hijos: 11
```



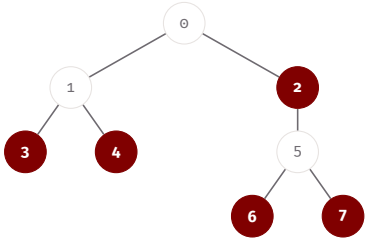
Lo que se escribe mal

$\mathrm{Sin}(v) = \sum \mathrm{Con}(c)$, como si no elegir a un padre obligara a elegir a sus hijos.

Esa recurrencia alterna a la fuerza: toma el 10, saca el 1, obliga al siguiente 1, y saca el otro 10. Da 11.

Lo correcto: dos nodos seguidos pueden quedar sin elegir. Por eso $\max(\mathrm{Con}(c), \mathrm{Sin}(c))$.

Reconstruir: de arriba hacia abajo



La decisión de un nodo depende de su padre. Por eso se baja, en el orden inverso.

- Raíz: $22 \leq 23$, fuera. Nodo 1: $9 \leq 10$, fuera.
- Nodos 3 y 4: $5 > 0$, dentro.
- Nodo 2: $13 > 8$, dentro.
- Nodo 5: su padre está dentro, y queda **fuera**, aunque $8 > 7$. Nodos 6 y 7: padre fuera, dentro.

Resultado: $\{2, 3, 4, 6, 7\}$, suma 23.

Error común: decidir cada nodo por separado

build/tree_demo

```
Mejor valor: 23, con {2, 3, 4, 6, 7}
Cada nodo por separado, sin mirar al padre: {2, 3, 4, 5, 6, 7}
```

Lo que dice cada fila es condicional

El nodo 2 tiene 13 $>$ 8, y el nodo 5 tiene 8 $>$ 7. Pero son vecinos. Esa lista tiene las aristas 2-5, 5-6 y 5-7.

Por inducción sobre la altura: lo elegido en $T(v)$ suma $\mathrm{Sin}(v)$ si el padre se eligió, y $\max(\mathrm{Con}(v), \mathrm{Sin}(v))$ si no. Ningún nodo elegido tiene a su padre elegido.

En un empate, se puede preferir cualquiera de los dos: ambos dan el óptimo.

Cobertura de vértices: el mismo patrón

Cobertura de vértices *(vertex cover)*

Nodos tales que toda arista tiene al menos una punta en el conjunto. Se busca el de peso mínimo.

$$\text{Dentro}(v) = \text{peso}(v) + \sum_{c} \min\big(\text{Dentro}(c), \text{Fuera}(c)\big), \qquad \text{Fuera}(v) = \sum_{c} \text{Dentro}(c)$$

- Si v está dentro, cada hijo queda libre.
- Si v está fuera, cada hijo **tiene** que estar: alguien debe cubrir la arista entre ellos.
- Aquí obligar a los hijos sí es correcto: lo pide la definición.
- La respuesta es $\min(\text{Dentro}(\text{raíz}), \text{Fuera}(\text{raíz}))$.

El lema del complemento

Lema, en cualquier grafo

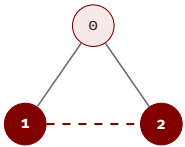
X es independiente si y solo si $V \setminus X$ es una cobertura: que ninguna arista tenga las dos puntas en X es que toda arista tenga una punta fuera.

build/tree_demo

```
Cobertura de peso mínimo del primer árbol
  con su propia recurrencia:         21
  fuerza bruta:                      21
  total 44 menos conjunto independiente: 21
```

Cobertura mínima $= W -$ **conjunto independiente máximo**, también con pesos negativos. Ambos son difíciles en un grafo general: semana 13.

¿Por qué no sirve en un grafo con ciclos?



build/tree_demo

```
Un triángulo con valores 5, 5, 5, usando el árbol de BFS desde 0
  el programa de árboles dice: 10, con {1, 2}
  pero 1 y 2 son vecinos en el triángulo, y lo mejor de verdad es 5
```

- Se rompe el lema: “no hay aristas entre subárboles de hermanos”.
- En un árbol, decidido *v*, los subárboles de los hijos se optimizan por separado.
- Con una arista más, lo que se decide en uno limita lo que se puede hacer en otro. Un bit por nodo ya no alcanza.

La recursión puede agotar la pila

make deep

```
bash -c 'build/deep_chain 100000; echo "código de salida: $?"' 2>&1
orden inverso de BFS, nodos bajo la raíz: 100000
recursión, conjunto independiente: 50000
código de salida: 0
bash -c 'build/deep_chain 1000000; echo "código de salida: $?"' 2>&1
orden inverso de BFS, nodos bajo la raíz: 1000000
bash: line 1: 10307 Segmentation fault      build/deep_chain 1000000
código de salida: 139
```

- Un árbol no tiene por qué ser bajo: una cadena de n nodos tiene profundidad n.
- Con `-O2` y la pila del contenedor, de 8192 KB: un stack overflow. El juez lo reporta como `RE`.
- El 10307 es el número del proceso, y cambia en cada corrida.

Evitarla: primero, un orden

orden de BFS

```
orden ← lista con raiz
padre ← arreglo de n valores -1
t ← 0
mientras t < largo de orden hacer
    v ← orden[t]
    para cada hijo c de v hacer
        padre[c] ← v
        agregar c al final de orden
    fin para
    t ← t + 1
fin mientras
```

- Un nodo entra a la cola cuando se procesa su padre: el padre sale antes.
- **Al revés**, cada nodo llega con sus hijos resueltos.
- Es un orden topológico del DAG, otra vez.

Error común

Procesar por número de nodo. Los números no dicen nada sobre la jerarquía.

Evitarla: después, al revés

hacia arriba, sin recursión

```
Con ← copia de valor
Sin ← arreglo de n ceros
t ← n - 1
mientras t ≥ 1 hacer
    v ← orden[t]
    Con[padre[v]] ← Con[padre[v]] + Sin[v]
    Sin[padre[v]] ← Sin[padre[v]] + máx(Con[v], Sin[v])
    t ← t - 1
fin mientras
devolver máx(Con[raiz], Sin[raiz])
```

- Es la misma suma de la recurrencia, hecha en otro orden: cada nodo le entrega lo suyo a su padre.
- Para reconstruir, se recorre `orden` hacia adelante: ahí el padre va antes.
- Sumas de hasta 10^{15} con 10^6 nodos y valores de hasta 10^9: `long long`.

SECCIÓN 07

Cierre

07

Lo que hay que llevarse

- **DAG de subproblemas:** un orden de cálculo es un orden topológico. Es un camino solo si cada decisión depende de un subproblema.
- **Camino más largo en un DAG:** $O(V + E)$, con pesos negativos. Un nodo sin alcanzar es $-\infty$, no 0.
- **LIS:** $\Theta(n^2)$ con $E(i)$. $O(n \log n)$ con el arreglo de finales, que no es una subsecuencia. `lower_bound` para estrictamente creciente.
- **Cadena de matrices:** un rango, $\Theta(n^3)$. El orden i, j ingenuo lee casillas vacías, y da un número razonable y falso.
- **Árbol de búsqueda óptimo:** se suma $F(i, j)$ porque colgar de una raíz baja un nivel a todo.
- **Árboles:** dos estados. Un nodo sin elegir deja libres a sus hijos. Se reconstruye de arriba hacia abajo.
- **Cobertura** $= W-$ conjunto independiente. Los dos son difíciles en un grafo general.
- **Recursión profunda:** orden inverso de BFS.

Para la próxima clase

- Leer los apuntes de la semana 9 y hacer sus ejercicios.
- Laboratorio: optimizar sobre una jerarquía, en `lab/company_meeting`. Extensión: `lab/company_meeting_list`.
- Empiecen con fuerza bruta en empresas pequeñas, y revisen los límites.

Preguntas

Dibujen el DAG de subproblemas de Fibonacci y el de la distancia de edición.
Calculen la mejor forma de multiplicar matrices de 10×30, 30×5 y 5×60.
¿Por qué hay que llenar la tabla de intervalos por largo creciente? ¿Es el único orden posible?
Escriban la recurrencia del conjunto independiente de peso máximo en un árbol.
¿Por qué ese algoritmo no sirve directamente en un grafo con ciclos?