

Análisis y Diseño de Algoritmos

Programación dinámica

De la memoización a la tabla: mochila, subsecuencias y distancia de edición

Gerson Garrido

Semana 7

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A

1. Fibonacci: los subproblemas se repiten
2. El método, en cuatro pasos
3. Intervalos con pesos
4. Mochila 0/1
5. Pseudopolinomial
6. Una sola fila, y en qué dirección

Sesión B

7. Subsecuencia común más larga
8. Distancia de edición
9. Dónde se usa
10. Demostrar: inducción sobre la tabla

> **Lectura**
>
> Apuntes de la semana 7. Código en `code/week-07`.

SECCIÓN 01

Fibonacci otra vez

01

Lo que nos faltaba

Hasta ahora: una decisión local, sin volver atrás, y una demostración de que no estropea nada.

Lo que nos faltaba

Hasta ahora: una decisión local, sin volver atrás, y una demostración de que no estropea nada.

En la semana 4 dejamos un pendiente: **clases con valor**. Ni “termina antes” ni “mayor valor” funcionan.

Lo que nos faltaba

Hasta ahora: una decisión local, sin volver atrás, y una demostración de que no estropea nada.

En la semana 4 dejamos un pendiente: **clases con valor**. Ni “termina antes” ni “mayor valor” funcionan.

Haría falta decir: *no sé si esta clase va, calculemos las dos posibilidades.*

Lo que nos faltaba

Hasta ahora: una decisión local, sin volver atrás, y una demostración de que no estropea nada.

En la semana 4 dejamos un pendiente: **clases con valor**. Ni “termina antes” ni “mayor valor” funcionan.

Haría falta decir: *no sé si esta clase va, calculemos las dos posibilidades.*

- Hecho sin cuidado: tiempo exponencial.

Lo que nos faltaba

Hasta ahora: una decisión local, sin volver atrás, y una demostración de que no estropea nada.

En la semana 4 dejamos un pendiente: **clases con valor**. Ni “termina antes” ni “mayor valor” funcionan.

Haría falta decir: *no sé si esta clase va, calculemos las dos posibilidades.*

- Hecho sin cuidado: tiempo exponencial.
- Hecho con cuidado: **programación dinámica** *(dynamic programming)*.

Lo que nos faltaba

Hasta ahora: una decisión local, sin volver atrás, y una demostración de que no estropea nada.

En la semana 4 dejamos un pendiente: **clases con valor**. Ni "termina antes" ni "mayor valor" funcionan.

Haría falta decir: *no sé si esta clase va, calculemos las dos posibilidades.*

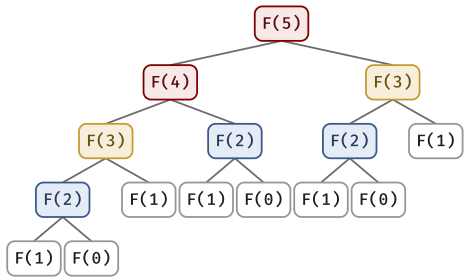
- Hecho sin cuidado: tiempo exponencial.
- Hecho con cuidado: **programación dinámica** *(dynamic programming)*.

> **La idea**
>
> Una recursión que prueba todo suele repetir las mismas preguntas. Si hay pocas preguntas distintas, se responde cada una una vez y se guarda.

La recursión directa repite

$F(n) = F(n-1) + F(n-2)$. Las llamadas de `fib(5)`:

Sin guardar nada: 15 llamadas

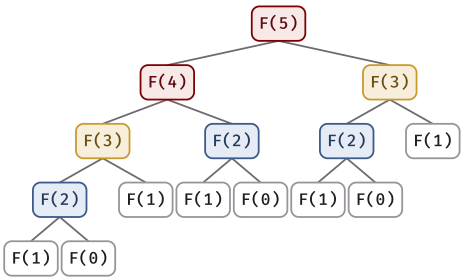


F(3) se calcula 2 veces, y F(2), 3 veces.
Solo hay 6 preguntas distintas.

La recursión directa repite

$F(n) = F(n-1) + F(n-2)$. Las llamadas de `fib(5)`:

Sin guardar nada: 15 llamadas



F(3) se calcula 2 veces, y F(2), 3 veces.
Solo hay 6 preguntas distintas.

Guardando cada respuesta: 9 llamadas



De 2 en adelante, cada F(n) se calcula una vez.
Las repetidas leen la respuesta y no bajan.

Pocas preguntas, muchas repeticiones

terminal

```
$ build/fib_calls
 n          F(n)   sin memoizar   memoizando
 5             5             15            9
10            55            177           19
20          6765          21891           39
30        832040        2692537           59
40     102334155      331160281           79
```

extracto

```
fib(10): cuántas veces se calcula
  F(0): 34
  F(1): 55
  F(2): 34
  F(3): 21
  F(4): 13
  ...
  F(9): 1
  F(10): 1
```

- Sin memoizar: $2F(n+1) - 1$ llamadas. Memoizando: $2n - 1$.
- `fib(10)` hace 177 llamadas, y **solo hay 11 preguntas distintas**.
- El tiempo exponencial viene de olvidar, no de que el problema sea difícil.

De la memoización a la tabla

de arriba hacia abajo

```
algoritmo fib_memo(n, memo)
    si n < 2 entonces
        devolver n
    fin si
    si memo[n] no tiene valor entonces
        memo[n] ← fib_memo(n - 1, memo)
                + fib_memo(n - 2, memo)
    fin si
    devolver memo[n]
fin algoritmo
```

de abajo hacia arriba

```
algoritmo fib_tabla(n)
    tabla ← arreglo de n + 2 ceros
    tabla[1] ← 1
    para k desde 2 hasta n hacer
        tabla[k] ← tabla[k - 1]
                 + tabla[k - 2]
    fin para
    devolver tabla[n]
fin algoritmo
```

- $F(n)$ solo necesita índices menores: se puede llenar en orden, sin recursión.
- Cada casilla lee las dos anteriores: bastan dos variables.

La tabla es el resultado, no la idea: sin una frase que diga qué significa cada casilla, es un acertijo.

SECCIÓN 02

El método

02

El método, en cuatro pasos

1. **Definir el subproblema, en una frase.** Qué significa cada casilla, y cuál es la respuesta. *(subproblem)*

El método, en cuatro pasos

1. **Definir el subproblema, en una frase.** Qué significa cada casilla, y cuál es la respuesta. *(subproblem)*
2. **Escribir la recurrencia.** Una sola decisión: ¿qué pasa con el último elemento? Con casos base.

El método, en cuatro pasos

1. **Definir el subproblema, en una frase.** Qué significa cada casilla, y cuál es la respuesta. *(subproblem)*
2. **Escribir la recurrencia.** Una sola decisión: ¿qué pasa con el último elemento? Con casos base.
3. **Elegir un orden.** Toda casilla que se lee ya está calculada. Con eso, el programa y su costo.

El método, en cuatro pasos

1. **Definir el subproblema, en una frase.** Qué significa cada casilla, y cuál es la respuesta. *(subproblem)*
2. **Escribir la recurrencia.** Una sola decisión: ¿qué pasa con el último elemento? Con casos base.
3. **Elegir un orden.** Toda casilla que se lee ya está calculada. Con eso, el programa y su costo.
4. **Reconstruir.** La tabla da el valor. La solución sale caminando hacia atrás.

El método, en cuatro pasos

1. **Definir el subproblema, en una frase.** Qué significa cada casilla, y cuál es la respuesta. *(subproblem)*
2. **Escribir la recurrencia.** Una sola decisión: ¿qué pasa con el último elemento? Con casos base.
3. **Elegir un orden.** Toda casilla que se lee ya está calculada. Con eso, el programa y su costo.
4. **Reconstruir.** La tabla da el valor. La solución sale caminando hacia atrás.

Y siempre se demuestra: una recurrencia afirma que una frase coincide con una fórmula. Va por inducción, en el orden de la tabla.

El método, en cuatro pasos

1. **Definir el subproblema, en una frase.** Qué significa cada casilla, y cuál es la respuesta. *(subproblem)*
2. **Escribir la recurrencia.** Una sola decisión: ¿qué pasa con el último elemento? Con casos base.
3. **Elegir un orden.** Toda casilla que se lee ya está calculada. Con eso, el programa y su costo.
4. **Reconstruir.** La tabla da el valor. La solución sale caminando hacia atrás.

Y siempre se demuestra: una recurrencia afirma que una frase coincide con una fórmula. Va por inducción, en el orden de la tabla.

> **Subestructura óptima** ***(optimal substructure)***
>
> Una solución óptima se compone de soluciones óptimas de sus subproblemas. No se supone: sale de cada demostración.

SECCIÓN 03

Intervalos con pesos

03

Clases con valor: el greedy deja de servir

Cada clase es $[s, f)$ y tiene un valor $v > 0$. Se busca el **máximo valor total** sin cruces.

build/weighted_demo

```
Una clase valiosa en medio:
  clases: [0,2)v1 [1,6)v10 [5,7)v1
  óptimo por fuerza bruta: 10
  termina antes (semana 4): 2
  mayor valor primero:      10
```

build/weighted_demo

```
Una clase larga que estorba:
  clases: [0,10)v5 [0,3)v4 [3,6)v4 [6,10)v4
  óptimo por fuerza bruta: 12
  termina antes (semana 4): 12
  mayor valor primero:      5
```

Dónde se rompe la demostración de la semana 4

El intercambio conserva el *número* de clases, no el valor: g_j puede ser barata y x_j muy cara.

Paso 1: el subproblema

Se ordenan las clases por hora de fin: $c[0], c[1], \ldots, c[n-1]$.

Subproblema

$T[j]$ es el máximo valor usando solo las primeras j clases del orden, $c[0], \ldots, c[j-1]$. La respuesta es $T[n]$.

Paso 1: el subproblema

Se ordenan las clases por hora de fin: $c[0], c[1], \ldots, c[n-1]$.

Subproblema

$T[j]$ es el máximo valor usando solo las primeras j clases del orden, $c[0], \ldots, c[j-1]$. La respuesta es $T[n]$.

Miremos la última clase disponible, $c[j-1]$:

Paso 1: el subproblema

Se ordenan las clases por hora de fin: $c[0], c[1], \ldots, c[n-1]$.

Subproblema

$T[j]$ es el máximo valor usando solo las primeras j clases del orden, $c[0], \ldots, c[j-1]$. La respuesta es $T[n]$.

Miremos la última clase disponible, $c[j-1]$:

- Si no está: lo mejor es $T[j-1]$.

Paso 1: el subproblema

Se ordenan las clases por hora de fin: $c[0], c[1], \ldots, c[n-1]$.

Subproblema

$T[j]$ es el máximo valor usando solo las primeras j clases del orden, $c[0], \ldots, c[j-1]$. La respuesta es $T[n]$.

Miremos la última clase disponible, $c[j-1]$:

- Si no está: lo mejor es $T[j-1]$.
- Si está: vale $v(c[j-1])$, y el resto tiene que ser compatible con ella.

Paso 1: el subproblema

Se ordenan las clases por hora de fin: $c[0], c[1], \ldots, c[n-1]$.

Subproblema

$T[j]$ es el máximo valor usando solo las primeras j clases del orden, $c[0], \ldots, c[j-1]$. La respuesta es $T[n]$.

Miremos la última clase disponible, $c[j-1]$:

- Si no está: lo mejor es $T[j-1]$.
- Si está: vale $v(c[j-1])$, y el resto tiene que ser compatible con ella.

$p(j)$

Cuántas clases del orden tienen hora de fin menor o igual que el inicio de $c[j-1]$.

Lema: las compatibles son un prefijo

Lema. Las compatibles con $c = c[j-1]$ entre las primeras $j-1$ son exactamente $c[0], \ldots, c[p(j)-1]$, y $p(j) \leq j-1$.

Sea x con índice menor que $j-1$. Por el orden, $f(x) \leq f(c)$. Son compatibles si:

Lema: las compatibles son un prefijo

Lema. Las compatibles con $c = c[j-1]$ entre las primeras $j-1$ son exactamente $c[0], \ldots, c[p(j)-1]$, y $p(j) \leq j-1$.

Sea x con índice menor que $j-1$. Por el orden, $f(x) \leq f(c)$. Son compatibles si:

- $f(c) \leq s(x)$. Imposible: $f(c) \leq s(x) < f(x) \leq f(c)$.

Lema: las compatibles son un prefijo

Lema. Las compatibles con $c = c[j-1]$ entre las primeras $j-1$ son exactamente $c[0], \ldots, c[p(j)-1]$, y $p(j) \leq j-1$.

Sea x con índice menor que $j-1$. Por el orden, $f(x) \leq f(c)$. Son compatibles si:

- $f(c) \leq s(x)$. Imposible: $f(c) \leq s(x) < f(x) \leq f(c)$.
- $f(x) \leq s(c)$. Es la única forma.

Lema: las compatibles son un prefijo

Lema. Las compatibles con $c = c[j-1]$ entre las primeras $j-1$ son exactamente $c[0], \ldots, c[p(j)-1]$, y $p(j) \leq j-1$.

Sea x con índice menor que $j-1$. Por el orden, $f(x) \leq f(c)$. Son compatibles si:

- $f(c) \leq s(x)$. Imposible: $f(c) \leq s(x) < f(x) \leq f(c)$.
- $f(x) \leq s(c)$. Es la única forma.

- Forman un prefijo: si una clase termina antes de $s(c)$, también las anteriores. Son $p(j)$.

Lema: las compatibles son un prefijo

Lema. Las compatibles con $c = c[j-1]$ entre las primeras $j-1$ son exactamente $c[0], \ldots, c[p(j)-1]$, y $p(j) \leq j-1$.

Sea x con índice menor que $j-1$. Por el orden, $f(x) \leq f(c)$. Son compatibles si:

- $f(c) \leq s(x)$. Imposible: $f(c) \leq s(x) < f(x) \leq f(c)$.
- $f(x) \leq s(c)$. Es la única forma.

- Forman un prefijo: si una clase termina antes de $s(c)$, también las anteriores. Son $p(j)$.
- No llega hasta c: con índice $j-1$ o mayor, $f \geq f(c) > s(c)$. Luego $p(j) \leq j-1$. ■

Paso 2: la recurrencia, demostrada

$$T[0] = 0, \qquad T[j] = \max\left(T[j-1],\ \ v(c[j-1]) + T[p(j)] \right)$$

Técnica. Inducción sobre j. $OPT(j)$ es lo que dice la frase. Se supone $T[i] = OPT(i)$ para $i < j$.

Paso 2: la recurrencia, demostrada

$$T[0] = 0, \qquad T[j] = \max\big(\, T[j-1],\;\; v(c[j-1]) + T[p(j)] \,\big)$$

Técnica. Inducción sobre j. $OPT(j)$ es lo que dice la frase. Se supone $T[i] = OPT(i)$ para $i < j$.

- **Base.** Sin clases, solo la solución vacía: 0.

Paso 2: la recurrencia, demostrada

$$T[0] = 0, \qquad T[j] = \max\big(\, T[j-1],\;\; v(c[j-1]) + T[p(j)] \,\big)$$

Técnica. Inducción sobre j. $OPT(j)$ es lo que dice la frase. Se supone $T[i] = OPT(i)$ para $i < j$.

- **Base.** Sin clases, solo la solución vacía: 0.
- **($\leq$)** Sea S óptima para j. Si no usa c: vale a lo sumo $T[j-1]$.

Paso 2: la recurrencia, demostrada

$$T[0] = 0, \qquad T[j] = \max\left(\, T[j-1],\;\; v(c[j-1]) + T[p(j)] \,\right)$$

Técnica. Inducción sobre j. $OPT(j)$ es lo que dice la frase. Se supone $T[i] = OPT(i)$ para $i < j$.

- **Base.** Sin clases, solo la solución vacía: 0.
- **($\leq$)** Sea S óptima para j. Si no usa c: vale a lo sumo $T[j-1]$.
- Si usa c: el resto es compatible y anterior. Por el lema está en las primeras $p(j)$, y S vale a lo sumo $v(c) + T[p(j)]$.

Paso 2: la recurrencia, demostrada

$$T[0] = 0, \qquad T[j] = \max\left(\, T[j-1],\;\; v(c[j-1]) + T[p(j)] \,\right)$$

Técnica. Inducción sobre j. $OPT(j)$ es lo que dice la frase. Se supone $T[i] = OPT(i)$ para $i < j$.

- **Base.** Sin clases, solo la solución vacía: 0.
- **($\leq$)** Sea S óptima para j. Si no usa c: vale a lo sumo $T[j-1]$.
- Si usa c: el resto es compatible y anterior. Por el lema está en las primeras $p(j)$, y S vale a lo sumo $v(c) + T[p(j)]$.
- **($\geq$)** Una solución de $j-1$ también lo es de j.

Paso 2: la recurrencia, demostrada

$$T[0] = 0, \qquad T[j] = \max\big(\, T[j-1],\;\; v(c[j-1]) + T[p(j)] \,\big)$$

Técnica. Inducción sobre j. $OPT(j)$ es lo que dice la frase. Se supone $T[i] = OPT(i)$ para $i < j$.

- **Base.** Sin clases, solo la solución vacía: 0.
- **($\leq$)** Sea S óptima para j. Si no usa c: vale a lo sumo $T[j-1]$.
- Si usa c: el resto es compatible y anterior. Por el lema está en las primeras $p(j)$, y S vale a lo sumo $v(c) + T[p(j)]$.
- **($\geq$)** Una solución de $j-1$ también lo es de j.
- Una óptima de las primeras $p(j)$, más c, es válida por el lema. Como $p(j) \leq j-1$, c no estaba. ■

Paso 3: el orden y el costo

pseudocódigo

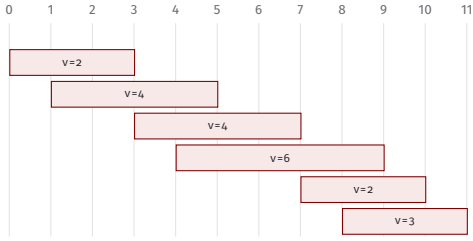
```
algoritmo intervalos_con_pesos(clases, n)
    ordenar clases por hora de fin
    para j desde 1 hasta n hacer
        p[j] ← clases con fin ≤ clases[j - 1].inicio,
               con búsqueda binaria
    fin para
    T[0] ← 0
    para j desde 1 hasta n hacer
        T[j] ← máx(T[j - 1],
                   clases[j - 1].valor + T[p[j]])
    fin para
    devolver T[n]
fin algoritmo
```

$T[j]$ lee $T[j-1]$ y $T[p(j)]$, con $p(j) \leq j-1$.

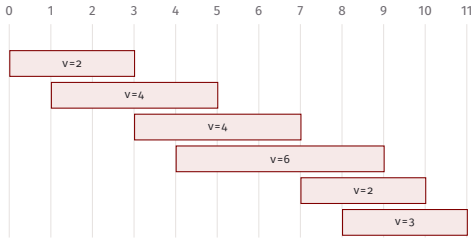
Ordenar: $O(n \log n)$.
n búsquedas: $O(n \log n)$.
Tabla: $\Theta(n)$.

Total: $O(n \log n)$, con $n+1$ subproblemas frente a 2^n subconjuntos.

Un ejemplo: las clases sobre el tiempo

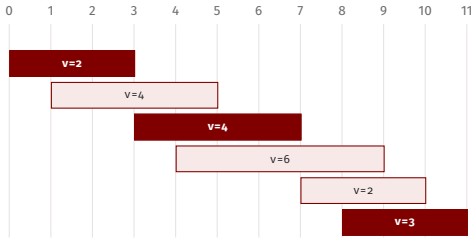


Un ejemplo: las clases sobre el tiempo



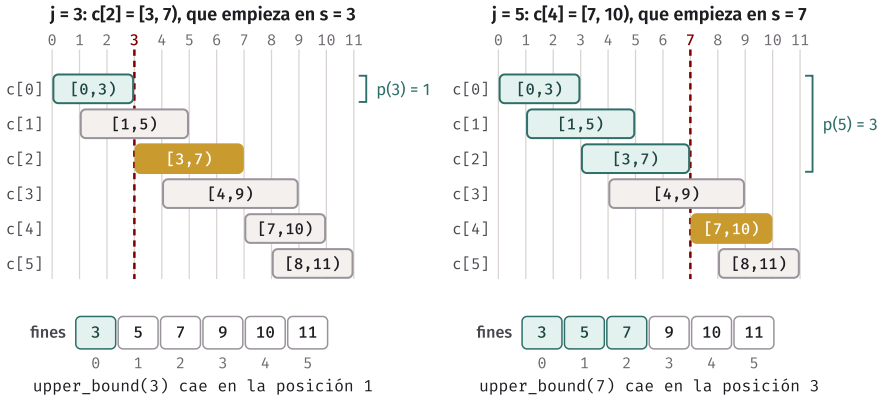
“Termina antes” da 8. “Mayor valor primero” da 8.

Un ejemplo: las clases sobre el tiempo



“Termina antes” da 8. “Mayor valor primero” da 8. El óptimo es 9, y usa dos clases que se tocan en el borde: $[0, 3)$ y $[3, 7)$.

$p(j)$ es un prefijo del orden por fin



Las clases con fin ≤ s son las primeras del orden por fin: un prefijo. El borde cuenta: un fin igual a s es compatible.

La tabla, fila por fila

j	clase	valor	$p(j)$	sin ella	con ella	$T[j]$
0	–	–	–	–	–	0

La tabla, fila por fila

j	clase	valor	$p(j)$	sin ella	con ella	$T[j]$
0	–	–	–	–	–	0
1	$[0,3)$	2	0	0	2	2

La tabla, fila por fila

j	clase	valor	$p(j)$	sin ella	con ella	$T[j]$
0	–	–	–	–	–	0
1	$[0,3)$	2	0	0	2	2
2	$[1,5)$	4	0	2	4	4

La tabla, fila por fila

j	clase	valor	$p(j)$	sin ella	con ella	$T[j]$
0	–	–	–	–	–	0
1	$[0,3)$	2	0	0	2	2
2	$[1,5)$	4	0	2	4	4
3	$[3,7)$	4	1	4	6	6

$[3,7)$ empieza en 3, y $[0,3)$ termina en 3: compatible. $p(3) = 1$.

La tabla, fila por fila

j	clase	valor	$p(j)$	sin ella	con ella	$T[j]$
0	–	–	–	–	–	0
1	$[0,3)$	2	0	0	2	2
2	$[1,5)$	4	0	2	4	4
3	$[3,7)$	4	1	4	6	6
4	$[4,9)$	6	1	6	8	8

La tabla, fila por fila

j	clase	valor	$p(j)$	sin ella	con ella	$T[j]$
0	–	–	–	–	–	0
1	$[0,3)$	2	0	0	2	2
2	$[1,5)$	4	0	2	4	4
3	$[3,7)$	4	1	4	6	6
4	$[4,9)$	6	1	6	8	8
5	$[7,10)$	2	3	8	8	8

Un empate: tomarla da $2 + T[3] = 8$, y no tomarla da $T[4] = 8$.

La tabla, fila por fila

j	clase	valor	$p(j)$	sin ella	con ella	$T[j]$
0	–	–	–	–	–	0
1	$[0, 3)$	2	0	0	2	2
2	$[1, 5)$	4	0	2	4	4
3	$[3, 7)$	4	1	4	6	6
4	$[4, 9)$	6	1	6	8	8
5	$[7, 10)$	2	3	8	8	8
6	$[8, 11)$	3	3	8	9	9

La tabla, fila por fila

j	clase	valor	$p(j)$	sin ella	con ella	$T[j]$
0	–	–	–	–	–	0
1	$[0,3)$	2	0	0	2	2
2	$[1,5)$	4	0	2	4	4
3	$[3,7)$	4	1	4	6	6
4	$[4,9)$	6	1	6	8	8
5	$[7,10)$	2	3	8	8	8
6	$[8,11)$	3	3	8	9	9

Hacia atrás: si $T[j] = T[j-1]$, se baja a $j-1$. Si no, se toma la clase y se salta a $p(j)$.

Paso 4: reconstruir, y un error de borde

terminal

```
Hacia atrás:
  T[6] = 9 > T[5] = 8: se toma [8,11), j = p(6) = 3
  T[3] = 6 > T[2] = 4: se toma [3,7), j = p(3) = 1
  T[1] = 2 > T[0] = 0: se toma [0,3), j = p(1) = 0
Elegidas: [0,3) [3,7) [8,11), valor 9

Si el borde contara como cruce (lower_bound en vez de upper_bound):
  p: 0 0 0 1 2 3
  valor: 8 en vez de 9
```

Error común: `lower_bound`

`lower_bound` no cuenta los fines iguales a s: es tratar los intervalos como cerrados. El programa no falla: da otra respuesta.

SECCIÓN 04

Mochila 0/1

04

Mochila 0/1 *(knapsack)*

n objetos con peso entero $w \geq 1$ y valor $v \geq 0$, y capacidad entera W. Máximo valor con peso a lo sumo W. Cada objeto se toma una vez o ninguna.

terminal

```
$ build/knapsack_demo
Un objeto valioso que no deja sitio:
  capacidad 10, objetos (peso, valor): (6, 30) (5, 20) (5, 20)
  óptimo por fuerza bruta: 40
  mayor valor/peso:        30
  mayor valor:             30
  tabla:                   40
```

(6, 30) tiene el mayor valor/peso y el mayor valor. Con él sobran 4 unidades, y esa capacidad no vale nada. Hay que considerar las dos opciones para cada objeto.

Un subproblema al que le falta información

Primer intento: “lo mejor con los primeros *i* objetos”. Si el siguiente cabe, se agrega.

extracto

```
capacidad 9, objetos (peso, valor): (2, 3) (3, 4) (4, 5) (5, 8)
óptimo por fuerza bruta: 13
mayor valor/peso:        11
sin capacidad en el subproblema: 12
```

Un subproblema al que le falta información

Primer intento: “lo mejor con los primeros *i* objetos”. Si el siguiente cabe, se agrega.

extracto

```
capacidad 9, objetos (peso, valor): (2, 3) (3, 4) (4, 5) (5, 8)
óptimo por fuerza bruta: 13
mayor valor/peso:        11
sin capacidad en el subproblema: 12
```

- Con tres objetos, lo mejor pesa 9 y vale 12. Ya no cabe (5, 8).

Un subproblema al que le falta información

Primer intento: “lo mejor con los primeros *i* objetos”. Si el siguiente cabe, se agrega.

extracto

```
capacidad 9, objetos (peso, valor): (2, 3) (3, 4) (4, 5) (5, 8)
óptimo por fuerza bruta: 13
mayor valor/peso:        11
sin capacidad en el subproblema: 12
```

- Con tres objetos, lo mejor pesa 9 y vale 12. Ya no cabe (5, 8).
- Pero la óptima usa lo mejor con tres objetos *dejando 5 libres*: (4, 5).

Un subproblema al que le falta información

Primer intento: “lo mejor con los primeros *i* objetos”. Si el siguiente cabe, se agrega.

extracto

```
capacidad 9, objetos (peso, valor): (2, 3) (3, 4) (4, 5) (5, 8)
óptimo por fuerza bruta: 13
mayor valor/peso:        11
sin capacidad en el subproblema: 12
```

- Con tres objetos, lo mejor pesa 9 y vale 12. Ya no cabe (5, 8).
- Pero la óptima usa lo mejor con tres objetos *dejando 5 libres*: (4, 5).

Error común

Definir un subproblema que no lleva lo necesario para decidir. La solución es **agregar un parámetro**.

La recurrencia de la mochila

Subproblema

$K(i, c)$: máximo valor con los primeros i objetos y capacidad c. Respuesta: $K(n, W)$.

Último objeto disponible, con peso w y valor v:

$$K(0, c) = 0, \qquad K(i, c) = \begin{cases} K(i-1, c) & \text{si } w > c \\ \max\left(K(i-1, c),\ v + K(i-1, c-w)\right) & \text{si } w \leq c \end{cases}$$

La recurrencia de la mochila

Subproblema

$K(i, c)$: máximo valor con los primeros i objetos y capacidad c. Respuesta: $K(n, W)$.

Último objeto disponible, con peso w y valor v:

$$K(0, c) = 0, \qquad K(i, c) = \begin{cases} K(i-1, c) & \text{si } w > c \\ \max\left(K(i-1, c),\ v + K(i-1, c-w)\right) & \text{si } w \le c \end{cases}$$

Dónde se usa que es 0/1

El segundo término lee la fila $i - 1$: el objeto no se puede volver a tomar. Volveremos a esta línea.

Demostración, por inducción sobre i

$OPT(i, c)$ es el máximo de un subconjunto de los primeros i objetos con peso a lo sumo c.

Demostración, por inducción sobre i

$OPT(i, c)$ es el máximo de un subconjunto de los primeros i objetos con peso a lo sumo c.

- **Base.** Con 0 objetos, solo el vacío: vale 0 y pesa $0 \leq c$.

Demostración, por inducción sobre i

$OPT(i, c)$ es el máximo de un subconjunto de los primeros i objetos con peso a lo sumo c.

- **Base.** Con 0 objetos, solo el vacío: vale 0 y pesa $0 \leq c$.
- **($\leq$)** Sea S óptimo. Si no usa el objeto $i - 1$: vale a lo sumo $OPT(i - 1, c)$.

Demostración, por inducción sobre i

$OPT(i, c)$ es el máximo de un subconjunto de los primeros i objetos con peso a lo sumo c.

- **Base.** Con 0 objetos, solo el vacío: vale 0 y pesa $0 \leq c$.
- **($\leq$)** Sea S óptimo. Si no usa el objeto $i-1$: vale a lo sumo $OPT(i-1, c)$.
- Si lo usa: $w \leq c$. Sin él, queda un subconjunto de los anteriores con peso a lo sumo $c - w$. Vale a lo sumo $v + OPT(i-1, c-w)$.

Demostración, por inducción sobre i

$OPT(i, c)$ es el máximo de un subconjunto de los primeros i objetos con peso a lo sumo c.

- **Base.** Con 0 objetos, solo el vacío: vale 0 y pesa $0 \leq c$.
- **($\leq$)** Sea S óptimo. Si no usa el objeto $i-1$: vale a lo sumo $OPT(i-1, c)$.
- Si lo usa: $w \leq c$. Sin él, queda un subconjunto de los anteriores con peso a lo sumo $c - w$. Vale a lo sumo $v + OPT(i-1, c-w)$.
- **($\geq$)** Un subconjunto de los primeros $i-1$ lo es de los primeros i.

Demostración, por inducción sobre i

$OPT(i, c)$ es el máximo de un subconjunto de los primeros i objetos con peso a lo sumo c.

- **Base.** Con 0 objetos, solo el vacío: vale 0 y pesa $0 \leq c$.
- **($\leq$)** Sea S óptimo. Si no usa el objeto $i-1$: vale a lo sumo $OPT(i-1, c)$.
- Si lo usa: $w \leq c$. Sin él, queda un subconjunto de los anteriores con peso a lo sumo $c - w$. Vale a lo sumo $v + OPT(i-1, c-w)$.
- **($\geq$)** Un subconjunto de los primeros $i-1$ lo es de los primeros i.
- Si $w \leq c$: a uno de los primeros $i-1$ con peso a lo sumo $c-w$ se le agrega el objeto, que no estaba. Pesa a lo sumo c. ■

El programa

knapsack.h

```cpp
inline std::vector<std::vector<long long>> knapsack_table(const std::vector<Item>& items,
                                                          std::size_t capacity) {
    const std::size_t n = items.size();
    std::vector<std::vector<long long>> table(n + 1, std::vector<long long>(capacity + 1, 0));
    for (std::size_t i = 1; i <= n; i++) {
        const Item& item = items[i - 1];
        for (std::size_t c = 0; c <= capacity; c++) {
            table[i][c] = table[i - 1][c];  // sin el objeto
            if (item.weight <= c) {
                table[i][c] = std::max(table[i][c], item.value + table[i - 1][c - item.weight]);
            }
        }
    }
    return table;
}
```

$(n+1)(W+1)$ casillas, cada una $O(1)$: $\Theta(nW)$ en tiempo y en memoria.

La tabla, y una casilla

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
ninguno	0	0	0	0	0	0	0	0	0	0
(2, 3)	0	0	3	3	3	3	3	3	3	3
(3, 4)	0	0	3	4	4	7	7	7	7	7
(4, 5)	0	0	3	4	5	7	8	9	9	12
(5, 8)	0	0	3	4	5	8	8	11	12	13

Filas: el último objeto disponible. Columnas: la capacidad c. Capacidad 9.

La tabla, y una casilla

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
ninguno	0	0	0	0	0	0	0	0	0	0
(2, 3)	0	0	3	3	3	3	3	3	3	3
(3, 4)	0	0	3	4	4	7	7	7	7	7
(4, 5)	0	0	3	4	5	7	8	9	9	12
(5, 8)	0	0	3	4	5	8	8	11	12	13

Última fila, $c = 9$. Sin (5, 8): la casilla de arriba, 12.

La tabla, y una casilla

	0	1	2	3	4	5	6	7	8	9
ninguno	0	0	0	0	0	0	0	0	0	0
(2, 3)	0	0	3	3	3	3	3	3	3	3
(3, 4)	0	0	3	4	4	7	7	7	7	7
(4, 5)	0	0	3	4	5	7	8	9	9	12
(5, 8)	0	0	3	4	5	8	8	11	12	13

Con (5, 8): 8 más la fila anterior en $c = 9 - 5 = 4$, que es 5. Da 13, y se queda con 13.

La tabla, y una casilla

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
ninguno	0	0	0	0	0	0	0	0	0	0
(2, 3)	0	0	3	3	3	3	3	3	3	3
(3, 4)	0	0	3	4	4	7	7	7	7	7
(4, 5)	0	0	3	4	5	7	8	9	9	12
(5, 8)	0	0	3	4	5	8	8	11	12	13

Hacia atrás. $T[4][9] = 13 \neq T[3][9] = 12$: se toma (5, 8), y se sigue en $c = 4$.

La tabla, y una casilla

	0	1	2	3	4	5	6	7	8	9
ninguno	0	0	0	0	0	0	0	0	0	0
(2, 3)	0	0	3	3	3	3	3	3	3	3
(3, 4)	0	0	3	4	4	7	7	7	7	7
(4, 5)	0	0	3	4	5	7	8	9	9	12
(5, 8)	0	0	3	4	5	8	8	11	12	13

$T[3][4] = 5 \neq T[2][4] = 4$: se toma (4, 5), y se sigue en $c = 0$. Lo demás no hace falta. Valor 13.

La tabla, y una casilla

	0	1	2	3	4	5	6	7	8	9
ninguno	0	0	0	0	0	0	0	0	0	0
(2, 3)	0	0	3	3	3	3	3	3	3	3
(3, 4)	0	0	3	4	4	7	7	7	7	7
(4, 5)	0	0	3	4	5	7	8	9	9	12
(5, 8)	0	0	3	4	5	8	8	11	12	13

En la fila (4, 5), $c = 4$ vale 5 y $c = 9$ vale 12: son dos respuestas para los mismos objetos. Por eso hace falta c.

¿Tabla o memoización?

extracto de build/knapsack_demo

```
Con memoización: valor 13, 14 estados calculados de 40 casillas con i >= 1
```

Tabla

Simple y con ciclos rápidos. Llena todas las casillas. Hay que elegir el orden.

Memoización

Calcula solo lo que la recursión visita. Paga la llamada y la búsqueda, y hasta n niveles de pila.

Son dos formas de hacer lo mismo, y en general no cambian el orden del peor caso.

SECCIÓN 05

Pseudopolinomial

05



¿$O(nW)$ **es polinomial?**

Polinomial: acotado por un polinomio en el **tamaño de la entrada**, medido en bits.

¿$O(nW)$ **es polinomial?**

Polinomial: acotado por un polinomio en el **tamaño de la entrada**, medido en bits.

- n **cuenta** objetos. Cada uno ocupa lugar en la entrada.

¿$O(nW)$ **es polinomial?**

Polinomial: acotado por un polinomio en el **tamaño de la entrada**, medido en bits.

- n **cuenta** objetos. Cada uno ocupa lugar en la entrada.
- W es un **valor**. Se escribe con $\lceil \log_2(W+1) \rceil$ bits. 1 000 000 ocupa 20 bits.

¿$O(nW)$ es polinomial?

Polinomial: acotado por un polinomio en el **tamaño de la entrada**, medido en bits.

- n **cuenta** objetos. Cada uno ocupa lugar en la entrada.
- W es un **valor**. Se escribe con $\lceil \log_2(W+1) \rceil$ bits. 1 000 000 ocupa 20 bits.
- Con $W = 2^b - 1$, y pesos y valores de hasta b bits, la entrada mide $\Theta(nb)$ bits.

¿$O(nW)$ es polinomial?

Polinomial: acotado por un polinomio en el **tamaño de la entrada**, medido en bits.

- n **cuenta** objetos. Cada uno ocupa lugar en la entrada.
- W es un **valor**. Se escribe con $\lceil \log_2(W+1) \rceil$ bits. 1 000 000 ocupa 20 bits.
- Con $W = 2^b - 1$, y pesos y valores de hasta b bits, la entrada mide $\Theta(nb)$ bits.
- La tabla tiene $(n+1) \cdot 2^b$ casillas: con n fijo, crece como 2^b.

¿$O(nW)$ es polinomial?

Polinomial: acotado por un polinomio en el **tamaño de la entrada**, medido en bits.

- n **cuenta** objetos. Cada uno ocupa lugar en la entrada.
- W es un **valor**. Se escribe con $\lceil \log_2(W+1) \rceil$ bits. 1 000 000 ocupa 20 bits.
- Con $W = 2^b - 1$, y pesos y valores de hasta b bits, la entrada mide $\Theta(nb)$ bits.
- La tabla tiene $(n+1) \cdot 2^b$ casillas: con n fijo, crece como 2^b.

Pseudopolinomial ***(pseudo-polynomial)***

Costo polinomial en los **valores** de los números de la entrada, y no en la cantidad de bits que los escriben. $O(nW)$ es polinomial en n y en el valor de W, y no lo es en n y $\log W$.

Cada bit de W: cerca del doble

extracto de make bench

```
n = 200, pesos al azar entre 1 y W/2, menor de 3 intentos
 bits de W           W    casillas n·W          ms
        18      131072        26214400         8.6   x2.14   (óptimo 19892176)
        19      262144        52428800        17.9   x2.08   (óptimo 16760028)
        20      524288       104857600        37.3   x2.09   (óptimo 19123141)
        21     1048576       209715200        74.3   x1.99   (óptimo 18208025)
        22     2097152       419430400       150.4   x2.02   (óptimo 18623276)
        23     4194304       838860800       302.4   x2.01   (óptimo 20579996)
        24     8388608      1677721600       684.9   x2.26   (óptimo 18984173)
        25    16777216      3355443200      1484.4   x2.17   (óptimo 16422278)
```

Con `-O2`, una corrida en el contenedor, con la máquina compartida. Los tiempos dependen de la máquina.

La entrada casi no creció: 200 números, con un bit más cada uno. El tiempo, cerca del doble en cada fila de este rango.

De 10 bits a 40

- W puede multiplicarse por cerca de 2^{30}, unos mil millones de veces. La tabla también.

De 10 bits a 40

- W puede multiplicarse por cerca de 2^{30}, unos mil millones de veces. La tabla también.
- Prolongando la última fila hasta 40 bits: unas 13 horas, y más de 4 TB para una sola fila. Es la cuenta, no una medición.

De 10 bits a 40

- W puede multiplicarse por cerca de 2^{30}, unos mil millones de veces. La tabla también.
- Prolongando la última fila hasta 40 bits: unas 13 horas, y más de 4 TB para una sola fila. Es la cuenta, no una medición.

Para no pasarse al otro extremo:

De 10 bits a 40

- W puede multiplicarse por cerca de 2^{30}, unos mil millones de veces. La tabla también.
- Prolongando la última fila hasta 40 bits: unas 13 horas, y más de 4 TB para una sola fila. Es la cuenta, no una medición.

Para no pasarse al otro extremo:

- **No significa inútil.** Con W pequeño, $O(nW)$ es excelente. Dinero en céntimos muchas veces cabe. Los bytes de un disco grande, no.

De 10 bits a 40

- W puede multiplicarse por cerca de 2^{30}, unos mil millones de veces. La tabla también.
- Prolongando la última fila hasta 40 bits: unas 13 horas, y más de 4 TB para una sola fila. Es la cuenta, no una medición.

Para no pasarse al otro extremo:

- **No significa inútil.** Con W pequeño, $O(nW)$ es excelente. Dinero en céntimos muchas veces cabe. Los bytes de un disco grande, no.
- **No es cuestión de un truco.** Nadie conoce un algoritmo polinomial en n y $\log W$. Volverá en la semana 14.

De 10 bits a 40

- W puede multiplicarse por cerca de 2^{30}, unos mil millones de veces. La tabla también.
- Prolongando la última fila hasta 40 bits: unas 13 horas, y más de 4 TB para una sola fila. Es la cuenta, no una medición.

Para no pasarse al otro extremo:

- **No significa inútil.** Con W pequeño, $O(nW)$ es excelente. Dinero en céntimos muchas veces cabe. Los bytes de un disco grande, no.
- **No es cuestión de un truco.** Nadie conoce un algoritmo polinomial en n y $\log W$. Volverá en la semana 14.

Una pregunta por cada letra

¿Cuenta cosas de la entrada, o es el valor de un número de la entrada?

SECCIÓN 06

Una sola fila

06

Ahorrar memoria

Con 1000 objetos y $W = 10^6$: unas 10^9 casillas de 8 bytes, 8 GB. Con una fila, 8 MB.

pseudocódigo

```
algoritmo mochila_una_fila(pesos, valores, n, W)
    fila ← arreglo de W + 1 ceros
    para i desde 0 hasta n - 1 hacer
        c ← W
        mientras c ≥ pesos[i] hacer
            fila[c] ← máx(fila[c],
                  valores[i] + fila[c - pesos[i]])
            c ← c - 1
        fin mientras
    fin para
    devolver fila[W]
fin algoritmo
```

Invariante

Antes de escribir c: las casillas mayores ya son de la fila $i + 1$, y las demás siguen siendo de la fila i.

Como $w \geq 1$, $c - w < c$: lee la fila anterior, y calcula $\max(K(i, c),\ v + K(i, c - w))$.

Las dos direcciones, paso a paso

Un objeto de peso 2 y valor 3, capacidad 6.

	0	1	2	3	4	5	6
de mayor a menor	0	0	0	0	0	0	0
de menor a mayor	0	0	0	0	0	0	0

Las dos direcciones, paso a paso

Un objeto de peso 2 y valor 3, capacidad 6.

	0	1	2	3	4	5	6
de mayor a menor	0	0	0	0	0	0	3
de menor a mayor	0	0	0	0	0	0	0

De mayor a menor: al escribir c, la casilla $c - 2$ sigue en cero. El objeto se cuenta una vez.

Las dos direcciones, paso a paso

Un objeto de peso 2 y valor 3, capacidad 6.

	0	1	2	3	4	5	6
de mayor a menor	0	0	0	0	0	3	3
de menor a mayor	0	0	0	0	0	0	0

De mayor a menor: al escribir c, la casilla $c - 2$ sigue en cero. El objeto se cuenta una vez.

Las dos direcciones, paso a paso

Un objeto de peso 2 y valor 3, capacidad 6.

	0	1	2	3	4	5	6
de mayor a menor	0	0	0	0	3	3	3
de menor a mayor	0	0	0	0	0	0	0

De mayor a menor: al escribir c, la casilla $c - 2$ sigue en cero. El objeto se cuenta una vez.

Las dos direcciones, paso a paso

Un objeto de peso 2 y valor 3, capacidad 6.

	0	1	2	3	4	5	6
de mayor a menor	0	0	0	3	3	3	3
de menor a mayor	0	0	0	0	0	0	0

De mayor a menor: al escribir c, la casilla $c - 2$ sigue en cero. El objeto se cuenta una vez.

Las dos direcciones, paso a paso

Un objeto de peso 2 y valor 3, capacidad 6.

	0	1	2	3	4	5	6
de mayor a menor	0	0	3	3	3	3	3
de menor a mayor	0	0	0	0	0	0	0

De mayor a menor: al escribir c, la casilla $c - 2$ sigue en cero. El objeto se cuenta una vez.

Las dos direcciones, paso a paso

Un objeto de peso 2 y valor 3, capacidad 6.

	0	1	2	3	4	5	6
de mayor a menor	0	0	3	3	3	3	3
de menor a mayor	0	0	3	0	0	0	0

Las dos direcciones, paso a paso

Un objeto de peso 2 y valor 3, capacidad 6.

	0	1	2	3	4	5	6
de mayor a menor	0	0	3	3	3	3	3
de menor a mayor	0	0	3	3	0	0	0

Las dos direcciones, paso a paso

Un objeto de peso 2 y valor 3, capacidad 6.

	0	1	2	3	4	5	6
de mayor a menor	0	0	3	3	3	3	3
de menor a mayor	0	0	3	3	6	0	0

De menor a mayor: al escribir $c = 4$, la casilla 2 **ya incluye el objeto**. Se suma otra vez: 6.

Las dos direcciones, paso a paso

Un objeto de peso 2 y valor 3, capacidad 6.

	0	1	2	3	4	5	6
de mayor a menor	0	0	3	3	3	3	3
de menor a mayor	0	0	3	3	6	6	0

Las dos direcciones, paso a paso

Un objeto de peso 2 y valor 3, capacidad 6.

	0	1	2	3	4	5	6
de mayor a menor	0	0	3	3	3	3	3
de menor a mayor	0	0	3	3	6	6	9

Con $c = 6$ da 9: el mismo objeto tres veces.

Al revés, resuelve otro problema

build/knapsack_direction: los objetos del ejemplo, capacidad 9

```
tabla completa:      13
fila, mayor a menor: 13
fila, menor a mayor: 14
sin límite de copias: 14
```

- Calcula $U(i, c) = \max(U(i-1, c),\ v + U(i, c-w))$: la **mochila sin límite de copias**. Comprobado en 3000 instancias.
- **El precio de una fila:** ya no se puede caminar hacia atrás.

Por qué es peligroso

Compila y da un número plausible. Si repetir objetos no conviene, hasta acierta.

SECCIÓN 07

Subsecuencia común más larga

07

Subsecuencia y subcadena

Subcadena *(substring)*

Un tramo **contiguo**.

Subsecuencia *(subsequence)*

Se borran caracteres, sin cambiar el orden de los que quedan.

a l g o r i t m o

l o g a r i t m o

Subsecuencia y subcadena

Subcadena *(substring)*

Un tramo **contiguo**.

Subsecuencia *(subsequence)*

Se borran caracteres, sin cambiar el orden de los que quedan.



Subcadena común más larga: `ritmo`, de largo 5.

Subsecuencia y subcadena

Subcadena *(substring)*

Un tramo **contiguo**.

Subsecuencia *(subsequence)*

Se borran caracteres, sin cambiar el orden de los que quedan.



Subsecuencia común más larga: `lgritmo`, de largo 7. También `loritmo`: puede haber varias.

Subsecuencia y subcadena

Subcadena *(substring)*

Un tramo **contiguo**.

Subsecuencia *(subsequence)*

Se borran caracteres, sin cambiar el orden de los que quedan.



Subsecuencia común más larga: `lgritmo`, de largo 7. También `loritmo`: puede haber varias.

Error común: confundirlas. En la subcadena, si los caracteres difieren, la casilla vale 0.

El subproblema: dos prefijos

Subproblema

$L(i, j)$: largo de la subsecuencia común más larga de $a[0..i)$ y $b[0..j)$. Respuesta: $L(n, m)$.

$$L(i, 0) = L(0, j) = 0, \qquad L(i, j) = \begin{cases} L(i-1, j-1) + 1 & \text{si } a[i-1] = b[j-1] \\ \max\left(L(i-1, j),\ L(i, j-1)\right) & \text{si } a[i-1] \neq b[j-1] \end{cases}$$

El subproblema: dos prefijos

Subproblema

$L(i, j)$: largo de la subsecuencia común más larga de $a[0..i)$ y $b[0..j)$. Respuesta: $L(n, m)$.

$$L(i, 0) = L(0, j) = 0, \qquad L(i, j) = \begin{cases} L(i-1, j-1) + 1 & \text{si } a[i-1] = b[j-1] \\ \max\left(L(i-1, j),\ L(i, j-1)\right) & \text{si } a[i-1] \neq b[j-1] \end{cases}$$

Si coinciden, los empareja sin mirar alternativas. Eso suena a greedy. ¿Por qué es seguro?

El subproblema: dos prefijos

Subproblema

$L(i, j)$: largo de la subsecuencia común más larga de $a[0..i)$ y $b[0..j)$. Respuesta: $L(n, m)$.

$$L(i, 0) = L(0, j) = 0, \qquad L(i, j) = \begin{cases} L(i-1, j-1) + 1 & \text{si } a[i-1] = b[j-1] \\ \max\left(L(i-1, j),\ L(i, j-1)\right) & \text{si } a[i-1] \neq b[j-1] \end{cases}$$

Si coinciden, los empareja sin mirar alternativas. Eso suena a greedy. ¿Por qué es seguro?

Si difieren, dice que uno de los dos sobra. ¿Por qué no sirven los dos?

Lema de la última posición

Sea z una subsecuencia común de $a[0..i)$ y $b[0..j)$, de largo $k \geq 1$, con posiciones $p_1 < \cdots < p_k$ en a y $q_1 < \cdots < q_k$ en b.

Lema

Solo el último carácter de z puede usar la posición $i-1$ de a o la posición $j-1$ de b. Por ser crecientes las posiciones, los demás están en $a[0..i-1)$ y $b[0..j-1)$.

Lema de la última posición

Sea z una subsecuencia común de $a[0..i)$ y $b[0..j)$, de largo $k \geq 1$, con posiciones $p_1 < \cdots < p_k$ en a y $q_1 < \cdots < q_k$ en b.

Lema

Solo el último carácter de z puede usar la posición $i-1$ de a o la posición $j-1$ de b. Por ser crecientes las posiciones, los demás están en $a[0..i-1)$ y $b[0..j-1)$.

Técnica para el teorema. Inducción sobre el orden de la tabla: fila por fila, y de izquierda a derecha. Las casillas $(i-1, j-1)$, $(i-1, j)$ e $(i, j-1)$ van antes de (i, j).

Lema de la última posición

Sea z una subsecuencia común de $a[0..i)$ y $b[0..j)$, de largo $k \geq 1$, con posiciones $p_1 < \cdots < p_k$ en a y $q_1 < \cdots < q_k$ en b.

Lema

Solo el último carácter de z puede usar la posición $i-1$ de a o la posición $j-1$ de b. Por ser crecientes las posiciones, los demás están en $a[0..i-1)$ y $b[0..j-1)$.

Técnica para el teorema. Inducción sobre el orden de la tabla: fila por fila, y de izquierda a derecha. Las casillas $(i-1, j-1)$, $(i-1, j)$ e $(i, j-1)$ van antes de (i, j).

Base. Si un prefijo es vacío, la única subsecuencia común es la vacía, de largo 0.

Caso igual: emparejar es seguro

Aquí $a[i-1] = b[j-1] = x$.

Caso igual: emparejar es seguro

Aquí $a[i-1] = b[j-1] = x$.

- **($\geq$)** Se toma una LCS de $a[0..i-1)$ y $b[0..j-1)$, y se le agrega x al final. Las posiciones $i-1$ y $j-1$ están después de todas las usadas.

Caso igual: emparejar es seguro

Aquí $a[i-1] = b[j-1] = x$.

- **($\geq$)** Se toma una LCS de $a[0..i-1)$ y $b[0..j-1)$, y se le agrega x al final. Las posiciones $i-1$ y $j-1$ están después de todas las usadas.
- **($\leq$)** Sea z común de $a[0..i)$ y $b[0..j)$, no vacía. Se le quita el último carácter.

Caso igual: emparejar es seguro

Aquí $a[i-1] = b[j-1] = x$.

- **($\geq$)** Se toma una LCS de $a[0..i-1)$ y $b[0..j-1)$, y se le agrega x al final. Las posiciones $i-1$ y $j-1$ están después de todas las usadas.
- **($\leq$)** Sea z común de $a[0..i)$ y $b[0..j)$, no vacía. Se le quita el último carácter.
- Por el lema, lo que queda es común de $a[0..i-1)$ y $b[0..j-1)$. Luego $|z| \leq L(i-1, j-1) + 1$.

Caso igual: emparejar es seguro

Aquí $a[i-1] = b[j-1] = x$.

- **($\geq$)** Se toma una LCS de $a[0..i-1)$ y $b[0..j-1)$, y se le agrega x al final. Las posiciones $i-1$ y $j-1$ están después de todas las usadas.
- **($\leq$)** Sea z común de $a[0..i)$ y $b[0..j)$, no vacía. Se le quita el último carácter.
- Por el lema, lo que queda es común de $a[0..i-1)$ y $b[0..j-1)$. Luego $|z| \leq L(i-1, j-1) + 1$.

La respuesta a la duda

Nadie supera $L(i-1, j-1)+1$, y emparejar lo alcanza. Aunque x aparezca muchas veces y una óptima lo empareje con otro, hay otra óptima del mismo largo que empareja estos dos.

Caso distinto: alguno sobra

$a[i-1] \neq b[j-1]$. Las posiciones de la z de arriba terminan en p_k y q_k.

Caso distinto: alguno sobra

$a[i-1] \neq b[j-1]$. Las posiciones de la z de arriba terminan en p_k y q_k.

- **($\geq$)** Una común de los prefijos acortados también es común de los originales.

Caso distinto: alguno sobra

$a[i-1] \neq b[j-1]$. Las posiciones de la z de arriba terminan en p_k y q_k.

- **($\geq$)** Una común de los prefijos acortados también es común de los originales.
- **($\leq$)** $p_k = i-1$ y $q_k = j-1$ a la vez es imposible: ese carácter sería $a[i-1]$ y $b[j-1]$, que son distintos.

Caso distinto: alguno sobra

$a[i-1] \neq b[j-1]$. Las posiciones de la z de arriba terminan en p_k y q_k.

- **($\geq$)** Una común de los prefijos acortados también es común de los originales.
- **($\leq$)** $p_k = i-1$ y $q_k = j-1$ a la vez es imposible: ese carácter sería $a[i-1]$ y $b[j-1]$, que son distintos.
- Entonces $p_k < i-1$ o $q_k < j-1$.

Caso distinto: alguno sobra

$a[i-1] \neq b[j-1]$. Las posiciones de la z de arriba terminan en p_k y q_k.

- **($\geq$)** Una común de los prefijos acortados también es común de los originales.
- **($\leq$)** $p_k = i-1$ y $q_k = j-1$ a la vez es imposible: ese carácter sería $a[i-1]$ y $b[j-1]$, que son distintos.
- Entonces $p_k < i-1$ o $q_k < j-1$.
- Por el lema: z no usa $a[i-1]$, o no usa $b[j-1]$.

Caso distinto: alguno sobra

$a[i-1] \neq b[j-1]$. Las posiciones de la z de arriba terminan en p_k y q_k.

- **($\geq$)** Una común de los prefijos acortados también es común de los originales.
- **($\leq$)** $p_k = i-1$ y $q_k = j-1$ a la vez es imposible: ese carácter sería $a[i-1]$ y $b[j-1]$, que son distintos.
- Entonces $p_k < i-1$ o $q_k < j-1$.
- Por el lema: z no usa $a[i-1]$, o no usa $b[j-1]$.
- Es común de $a[0..i-1)$ y $b[0..j)$, o de $a[0..i)$ y $b[0..j-1)$. ■

Caso distinto: alguno sobra

$a[i-1] \neq b[j-1]$. Las posiciones de la z de arriba terminan en p_k y q_k.

- **($\geq$)** Una común de los prefijos acortados también es común de los originales.
- **($\leq$)** $p_k = i-1$ y $q_k = j-1$ a la vez es imposible: ese carácter sería $a[i-1]$ y $b[j-1]$, que son distintos.
- Entonces $p_k < i-1$ o $q_k < j-1$.
- Por el lema: z no usa $a[i-1]$, o no usa $b[j-1]$.
- Es común de $a[0..i-1)$ y $b[0..j)$, o de $a[0..i)$ y $b[0..j-1)$. ■

Costo: $(n+1)(m+1)$ casillas, cada una $O(1)$. $\Theta(nm)$. Aquí n y m cuentan caracteres de la entrada: sí es polinomial.

La tabla de `algoritmo` y `logaritmo`

		l	o	g	a	r	i	t	m	o
	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	1	1	1	1	1	1
l	0	1	1	1	1	1	1	1	1	1
g	0	1	1	2	2	2	2	2	2	2
o	0	1	2	2	2	2	2	2	2	3
r	0	1	2	2	2	3	3	3	3	3
i	0	1	2	2	2	3	4	4	4	4
t	0	1	2	2	2	3	4	5	5	5
m	0	1	2	2	2	3	4	5	6	6
o	0	1	2	2	2	3	4	5	6	7

La fila 0 y la columna 0 son los prefijos vacíos.

La tabla de `algoritmo` y `logaritmo`

		l	o	g	a	r	i	t	m	o
	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	1	1	1	1	1	1
l	0	1	1	1	1	1	1	1	1	1
g	0	1	1	2	2	2	2	2	2	2
o	0	1	2	2	2	2	2	2	2	3
r	0	1	2	2	2	3	3	3	3	3
i	0	1	2	2	2	3	4	4	4	4
t	0	1	2	2	2	3	4	5	5	5
m	0	1	2	2	2	3	4	5	6	6
o	0	1	2	2	2	3	4	5	6	7

Casilla (3, 3): g y g coinciden. La diagonal más uno: $1 + 1 = 2$.

La tabla de `algoritmo` y `logaritmo`

		l	o	g	a	r	i	t	m	o
	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	1	1	1	1	1	1
l	0	1	1	1	1	1	1	1	1	1
g	0	1	1	2	2	2	2	2	2	2
o	0	1	2	2	2	2	2	2	2	3
r	0	1	2	2	2	3	3	3	3	3
i	0	1	2	2	2	3	4	4	4	4
t	0	1	2	2	2	3	4	5	5	5
m	0	1	2	2	2	3	4	5	6	6
o	0	1	2	2	2	3	4	5	6	7

Casilla (3, 4): g y a difieren. El máximo de arriba (1) y de la izquierda (2).

La tabla de `algoritmo` y `logaritmo`

		l	o	g	a	r	i	t	m	o
	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	1	1	1	1	1	1
l	0	1	1	1	1	1	1	1	1	1
g	0	1	1	2	2	2	2	2	2	2
o	0	1	2	2	2	2	2	2	2	3
r	0	1	2	2	2	3	3	3	3	3
i	0	1	2	2	2	3	4	4	4	4
t	0	1	2	2	2	3	4	5	5	5
m	0	1	2	2	2	3	4	5	6	6
o	0	1	2	2	2	3	4	5	6	7

Hacia atrás: `o`, `m`, `t`, `i`, `r` por la diagonal. En (4, 4), arriba y a la izquierda valen 2: un empate.

La tabla de `algoritmo` y `logaritmo`

		l	o	g	a	r	i	t	m	o
	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	1	1	1	1	1	1
l	0	1	1	1	1	1	1	1	1	1
g	0	1	1	2	2	2	2	2	2	2
o	0	1	2	2	2	2	2	2	2	3
r	0	1	2	2	2	3	3	3	3	3
i	0	1	2	2	2	3	4	4	4	4
t	0	1	2	2	2	3	4	5	5	5
m	0	1	2	2	2	3	4	5	6	6
o	0	1	2	2	2	3	4	5	6	7

Subiendo: `g` en (3, 3) y `l` en (2, 1). Sale `lgritmo`.

La tabla de `algoritmo` y `logaritmo`

		l	o	g	a	r	i	t	m	o
	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	1	1	1	1	1	1
l	0	1	1	1	1	1	1	1	1	1
g	0	1	1	2	2	2	2	2	2	2
o	0	1	2	2	2	2	2	2	2	3
r	0	1	2	2	2	3	3	3	3	3
i	0	1	2	2	2	3	4	4	4	4
t	0	1	2	2	2	3	4	5	5	5
m	0	1	2	2	2	3	4	5	6	6
o	0	1	2	2	2	3	4	5	6	7

Por la izquierda: `o` en (4, 2) y `l` en (2, 1). Sale `loritmo`. Las dos son correctas.

Reconstruir

pseudocódigo

```
algoritmo reconstruir_lcs(a, n, b, m, L)
    salida ← lista vacía
    i ← n
    j ← m
    mientras i > 0 y j > 0 hacer
        si a[i - 1] = b[j - 1] entonces
            agregar a[i - 1] a salida
            i ← i - 1
            j ← j - 1
        sino si L[i - 1][j] ≥ L[i][j - 1] entonces
            i ← i - 1
        sino
            j ← j - 1
        fin si
    fin mientras
    devolver salida invertida
fin algoritmo
```

Cada paso va a una casilla que explica el valor actual, y cada coincidencia suma 1. Cuesta $O(n + m)$.

Error común

La casilla (i, j) termina en $a[i-1]$ y $b[j-1]$, no en $a[i]$ y $b[j]$.

SECCIÓN 08

Distancia de edición

08

Distancia de edición *(edit distance)*

Definición

Operaciones de costo 1: **insertar**, **borrar** y **cambiar** un carácter. La distancia es el mínimo de operaciones para convertir *a* en *b*.

casa $\rightarrow$ cosa $\rightarrow$ cosas: 2 operaciones. ¿Se puede con menos?

Distancia de edición *(edit distance)*

Definición

Operaciones de costo 1: **insertar**, **borrar** y **cambiar** un carácter. La distancia es el mínimo de operaciones para convertir a en b.

casa $\to$ cosa $\to$ cosas: 2 operaciones. ¿Se puede con menos?

La definición permite rarezas: insertar y luego borrar, cambiar lo mismo tres veces, pasar por cadenas más largas. No deberían ayudar, y hay que demostrarlo.

Distancia de edición *(edit distance)*

Definición

Operaciones de costo 1: **insertar**, **borrar** y **cambiar** un carácter. La distancia es el mínimo de operaciones para convertir a en b.

Alineamiento ***(alignment)***

Dos filas, con huecos, sin columnas de dos huecos. Costo: las columnas en que las filas difieren.

```
c a s a -
c o s a s
```

casa → cosa → cosas: 2 operaciones. ¿Se puede con menos?

La definición permite rarezas: insertar y luego borrar, cambiar lo mismo tres veces, pasar por cadenas más largas. No deberían ayudar, y hay que demostrarlo.

Lema: operaciones y alineamientos

Lema. La distancia de edición es igual al mínimo costo de un alineamiento.

Distancia $\leq$ costo. Se leen las columnas de izquierda a derecha, y cada columna distinta es una operación.

Lema: operaciones y alineamientos

Lema. La distancia de edición es igual al mínimo costo de un alineamiento.

Distancia $\leq$ costo. Se leen las columnas de izquierda a derecha, y cada columna distinta es una operación.

Costo $\leq$ distancia. Si a se convierte en s con k operaciones, hay un alineamiento de a y s de costo a lo sumo k. Por inducción sobre k; con $k = 0$, cada carácter con sí mismo.

Lema: operaciones y alineamientos

Lema. La distancia de edición es igual al mínimo costo de un alineamiento.

Distancia $\leq$ costo. Se leen las columnas de izquierda a derecha, y cada columna distinta es una operación.

Costo $\leq$ distancia. Si a se convierte en s con k operaciones, hay un alineamiento de a y s de costo a lo sumo k. Por inducción sobre k; con $k = 0$, cada carácter con sí mismo.

- **Cambiar:** cambia un carácter de la fila de abajo. Esa columna sube a lo sumo 1.

Lema: operaciones y alineamientos

Lema. La distancia de edición es igual al mínimo costo de un alineamiento.

Distancia $\leq$ costo. Se leen las columnas de izquierda a derecha, y cada columna distinta es una operación.

Costo $\leq$ distancia. Si a se convierte en s con k operaciones, hay un alineamiento de a y s de costo a lo sumo k. Por inducción sobre k; con $k = 0$, cada carácter con sí mismo.

- **Cambiar:** cambia un carácter de la fila de abajo. Esa columna sube a lo sumo 1.
- **Insertar:** una columna nueva con hueco arriba. Sube 1.

Lema: operaciones y alineamientos

Lema. La distancia de edición es igual al mínimo costo de un alineamiento.

Distancia $\leq$ costo. Se leen las columnas de izquierda a derecha, y cada columna distinta es una operación.

Costo $\leq$ distancia. Si a se convierte en s con k operaciones, hay un alineamiento de a y s de costo a lo sumo k. Por inducción sobre k; con $k = 0$, cada carácter con sí mismo.

- **Cambiar:** cambia un carácter de la fila de abajo. Esa columna sube a lo sumo 1.
- **Insertar:** una columna nueva con hueco arriba. Sube 1.
- **Borrar:** queda un hueco abajo. Si arriba hay carácter, sube a lo sumo 1.

Lema: operaciones y alineamientos

Lema. La distancia de edición es igual al mínimo costo de un alineamiento.

Distancia $\leq$ costo. Se leen las columnas de izquierda a derecha, y cada columna distinta es una operación.

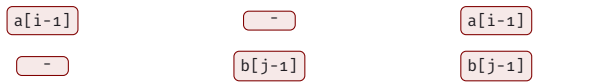
Costo $\leq$ distancia. Si a se convierte en s con k operaciones, hay un alineamiento de a y s de costo a lo sumo k. Por inducción sobre k; con $k = 0$, cada carácter con sí mismo.

- **Cambiar:** cambia un carácter de la fila de abajo. Esa columna sube a lo sumo 1.
- **Insertar:** una columna nueva con hueco arriba. Sube 1.
- **Borrar:** queda un hueco abajo. Si arriba hay carácter, sube a lo sumo 1.
- Si arriba había un hueco, la columna desaparece, y el costo baja. ■

La recurrencia: la última columna

Subproblema

$D(i, j)$: distancia de edición entre $a[0..i)$ y $b[0..j)$. Respuesta: $D(n, m)$.



$$D(i,0) = i, \quad D(0,j) = j, \quad D(i,j) = \min\left(D(i-1,j)+1,\ D(i,j-1)+1,\ D(i-1,j-1)+d\right)$$

con $d = 0$ si $a[i-1] = b[j-1]$, y $d = 1$ si no. La tabla tiene $(n+1)$ **por** $(m+1)$ casillas: $\Theta(nm)$.

Demostración, por inducción sobre el orden de la tabla

Fila por fila y de izquierda a derecha. Arriba, izquierda y diagonal van antes.

Demostración, por inducción sobre el orden de la tabla

Fila por fila y de izquierda a derecha. Arriba, izquierda y diagonal van antes.

- **Bases.** Si $b[0..0)$ está vacía, hay un solo alineamiento: i columnas con hueco abajo, costo i. Igual $D(0, j)$.

Demostración, por inducción sobre el orden de la tabla

Fila por fila y de izquierda a derecha. Arriba, izquierda y diagonal van antes.

- **Bases.** Si $b[0..0)$ está vacía, hay un solo alineamiento: i columnas con hueco abajo, costo i. Igual $D(0, j)$.
- **($\geq$)** Un alineamiento óptimo de $a[0..i)$ y $b[0..j)$ tiene última columna, de una de las tres formas.

Demostración, por inducción sobre el orden de la tabla

Fila por fila y de izquierda a derecha. Arriba, izquierda y diagonal van antes.

- **Bases.** Si $b[0..0)$ está vacía, hay un solo alineamiento: i columnas con hueco abajo, costo i. Igual $D(0, j)$.
- **($\geq$)** Un alineamiento óptimo de $a[0..i)$ y $b[0..j)$ tiene última columna, de una de las tres formas.
- Al quitarla, queda un alineamiento de los prefijos acortados, y el costo baja en 1, 1 o d. Lo que queda cuesta al menos su casilla.

Demostración, por inducción sobre el orden de la tabla

Fila por fila y de izquierda a derecha. Arriba, izquierda y diagonal van antes.

- **Bases.** Si $b[0..0)$ está vacía, hay un solo alineamiento: i columnas con hueco abajo, costo i. Igual $D(0, j)$.
- **($\geq$)** Un alineamiento óptimo de $a[0..i)$ y $b[0..j)$ tiene última columna, de una de las tres formas.
- Al quitarla, queda un alineamiento de los prefijos acortados, y el costo baja en 1, 1 o d. Lo que queda cuesta al menos su casilla.
- **($\leq$)** Cada término se alcanza: a un alineamiento óptimo de los prefijos acortados se le agrega esa columna. ■

Demostración, por inducción sobre el orden de la tabla

Fila por fila y de izquierda a derecha. Arriba, izquierda y diagonal van antes.

- **Bases.** Si $b[0..0)$ está vacía, hay un solo alineamiento: i columnas con hueco abajo, costo i. Igual $D(0, j)$.
- **($\geq$)** Un alineamiento óptimo de $a[0..i)$ y $b[0..j)$ tiene última columna, de una de las tres formas.
- Al quitarla, queda un alineamiento de los prefijos acortados, y el costo baja en 1, 1 o d. Lo que queda cuesta al menos su casilla.
- **($\leq$)** Cada término se alcanza: a un alineamiento óptimo de los prefijos acortados se le agrega esa columna. ■

Comprobado sin alineamientos

Un BFS sobre cadenas, con una arista por operación, es la definición tal cual. Coincide con la tabla en 400 pares de cadenas cortas sobre $\{a, b\}$.

El programa

sequences.h

```cpp
inline Table edit_table(const std::string& a, const std::string& b) {
    Table dist(a.size() + 1, std::vector<int>(b.size() + 1, 0));
    for (std::size_t i = 0; i <= a.size(); i++) dist[i][0] = static_cast<int>(i);
    for (std::size_t j = 0; j <= b.size(); j++) dist[0][j] = static_cast<int>(j);
    for (std::size_t i = 1; i <= a.size(); i++) {
        for (std::size_t j = 1; j <= b.size(); j++) {
            const int change = a[i - 1] == b[j - 1] ? 0 : 1;
            dist[i][j] = std::min({dist[i - 1][j] + 1,          // borrar a[i - 1]
                                   dist[i][j - 1] + 1,          // insertar b[j - 1]
                                   dist[i - 1][j - 1] + change}); // conservar o cambiar
        }
    }
    return dist;
}
```

casa y cosas, casilla por casilla

		c	o	s	a	s
	0	1	2	3	4	5
c	1					
a	2					
s	3					
a	4					

Fila 0: construir cada prefijo de `cosas` desde nada. Columna 0: borrar cada prefijo de `casa`.

casa y cosas, casilla por casilla

		c	o	s	a	s
	0	1	2	3	4	5
c	1	0	1	2	3	4
a	2					
s	3					
a	4					

(1, 1): c y c coinciden. Arriba y a la izquierda dan $1 + 1$, la diagonal da $0 + 0$. Vale 0.

casa y cosas, casilla por casilla

		c	o	s	a	s
	0	1	2	3	4	5
c	1	0	1	2	3	4
a	2	1	1	2	2	3
s	3					
a	4					

(2, 2): a y o difieren. Arriba y a la izquierda dan $1 + 1$, la diagonal da $0 + 1$. Vale 1.

casa y cosas, casilla por casilla

		c	o	s	a	s
	0	1	2	3	4	5
c	1	0	1	2	3	4
a	2	1	1	2	2	3
s	3	2	2	1	2	2
a	4					

(3, 3): s y s coinciden. La diagonal da $1 + 0$. Vale 1.

casa y cosas, casilla por casilla

		c	o	s	a	s
	0	1	2	3	4	5
c	1	0	1	2	3	4
a	2	1	1	2	2	3
s	3	2	2	1	2	2
a	4	3	3	2	1	2

(4, 5): a y s difieren. Arriba $2 + 1$, izquierda $1 + 1$, diagonal $2 + 1$. Vale 2: la distancia.

Reconstruir: una opción que explique el valor

Desde la esquina (n, m), hasta $(0, 0)$. En cada casilla se toma la primera opción que explique su valor:

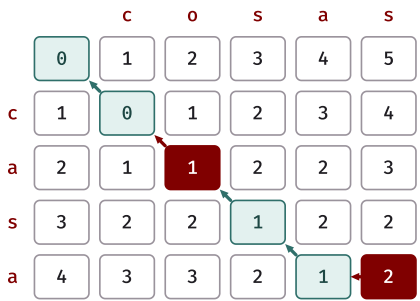
Si	Columna	Sigue en
$a[i-1] = b[j-1]$ y $D[i][j] = D[i-1][j-1]$	conservar $a[i-1]$	$(i-1, j-1)$
$D[i][j] = D[i-1][j-1] + 1$	cambiar $a[i-1]$ por $b[j-1]$	$(i-1, j-1)$
$D[i][j] = D[i-1][j] + 1$	borrar $a[i-1]$	$(i-1, j)$
en otro caso	insertar $b[j-1]$	$(i, j-1)$

La diagonal requiere $i, j > 0$, y borrar requiere $i > 0$. Hasta $n + m$ pasos: $O(n + m)$.

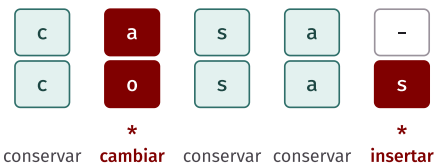
Por cómo se calculó la casilla, alguna opción la explica. Si hay varias, cualquiera da un alineamiento óptimo.

Hacia atrás: el recorrido da el alineamiento

La tabla, y el recorrido desde la esquina



El alineamiento



Costo 2: las columnas en que las filas difieren.

Cada flecha es una columna, de la última a la primera. Verde: cuesta 0. Granate: cuesta 1.

Lo que imprime el programa

terminal

```
$ build/edit_demo
casa -> cosas
     c o s a s
   0 1 2 3 4 5
c  1 0 1 2 3 4
a  2 1 1 2 2 3
s  3 2 2 1 2 2
a  4 3 3 2 1 2
Distancia: 2
Alineamiento:
  casa-
  cosas
   *  *
Operaciones:
  cambiar a por o
  insertar s
Con solo borrar e insertar: 3
```

SECCIÓN 09

Dónde se usa

09

Sin cambios: otra vez la LCS

Afirmación. Solo con borrar e insertar, el mínimo es $n + m - 2L$, con L el largo de la LCS.

Sin cambios: otra vez la LCS

Afirmación. Solo con borrar e insertar, el mínimo es $n + m - 2L$, con L el largo de la LCS.

- Un alineamiento sin columnas de cambio, con k columnas iguales. Esas columnas son una subsecuencia común: $k \leq L$.

Sin cambios: otra vez la LCS

Afirmación. Solo con borrar e insertar, el mínimo es $n + m - 2L$, con L el largo de la LCS.

- Un alineamiento sin columnas de cambio, con k columnas iguales. Esas columnas son una subsecuencia común: $k \leq L$.
- Lo demás se borra o se inserta: $(n - k) + (m - k) = n + m - 2k \geq n + m - 2L$.

Sin cambios: otra vez la LCS

Afirmación. Solo con borrar e insertar, el mínimo es $n + m - 2L$, con L el largo de la LCS.

- Un alineamiento sin columnas de cambio, con k columnas iguales. Esas columnas son una subsecuencia común: $k \leq L$.
- Lo demás se borra o se inserta: $(n - k) + (m - k) = n + m - 2k \geq n + m - 2L$.
- Y se alcanza: se alinea una LCS, y lo demás va con huecos. ■

Sin cambios: otra vez la LCS

Afirmación. Solo con borrar e insertar, el mínimo es $n + m - 2L$, con L el largo de la LCS.

- Un alineamiento sin columnas de cambio, con k columnas iguales. Esas columnas son una subsecuencia común: $k \leq L$.
- Lo demás se borra o se inserta: $(n - k) + (m - k) = n + m - 2k \geq n + m - 2L$.
- Y se alcanza: se alinea una LCS, y lo demás va con huecos. ■

Con `casa` y `cosas`: $4 + 5 - 2 \cdot 3 = 3$, porque el cambio de `a` por `o` ahora cuesta 2.

Dónde se usa

- **`diff`.** La misma cuenta, tomando cada **línea** como un carácter. Las líneas que se conservan son una subsecuencia común. Las herramientas reales no siempre pagan la tabla completa ni garantizan el mínimo.

entorno del curso: GNU diff 3.10

```
$ diff --help | grep minimal
  -d, --minimal            try hard to find a smaller set of changes
```

- **Correctores ortográficos.** Sugerir palabras a poca distancia es una base. Intercambiar dos letras vecinas aquí cuesta 2, y por eso hay variantes.
- **ADN.** Se alinean secuencias con esta tabla, en su forma básica, y puntajes para coincidencias, cambios y huecos en lugar de costos unitarios.

SECCIÓN 10

Demostrar programación dinámica

10



La receta: inducción sobre el orden de la tabla

1. **Qué significa la casilla**, en palabras. Eso es *OPT*.

La receta: inducción sobre el orden de la tabla

1. **Qué significa la casilla**, en palabras. Eso es *OPT*.
2. **El orden.** Cada casilla leída va antes: permite la hipótesis, y el programa no lee vacío.

La receta: inducción sobre el orden de la tabla

1. **Qué significa la casilla**, en palabras. Eso es *OPT*.
2. **El orden.** Cada casilla leída va antes: permite la hipótesis, y el programa no lee vacío.
3. **Casos base**, contra la frase y no contra la fórmula.

La receta: inducción sobre el orden de la tabla

1. **Qué significa la casilla**, en palabras. Eso es *OPT*.
2. **El orden.** Cada casilla leída va antes: permite la hipótesis, y el programa no lee vacío.
3. **Casos base**, contra la frase y no contra la fórmula.
4. **Paso, en dos direcciones.**
   - **Toda solución queda cubierta:** una óptima cualquiera; se miran **todos** los casos de su última decisión.
   - **Cada opción se puede conseguir:** la combinación sigue siendo válida.
   - Al maximizar, la primera es $\leq$ y la segunda $\geq$. Al minimizar, se invierten.

La receta: inducción sobre el orden de la tabla

1. **Qué significa la casilla**, en palabras. Eso es *OPT*.
2. **El orden.** Cada casilla leída va antes: permite la hipótesis, y el programa no lee vacío.
3. **Casos base**, contra la frase y no contra la fórmula.
4. **Paso, en dos direcciones.**
   - **Toda solución queda cubierta:** una óptima cualquiera; se miran **todos** los casos de su última decisión.
   - **Cada opción se puede conseguir:** la combinación sigue siendo válida.
   - Al maximizar, la primera es $\leq$ y la segunda $\geq$. Al minimizar, se invierten.
5. **Costo:** casillas por trabajo en cada una.

La receta: inducción sobre el orden de la tabla

1. **Qué significa la casilla**, en palabras. Eso es *OPT*.
2. **El orden.** Cada casilla leída va antes: permite la hipótesis, y el programa no lee vacío.
3. **Casos base**, contra la frase y no contra la fórmula.
4. **Paso, en dos direcciones.**
   - **Toda solución queda cubierta:** una óptima cualquiera; se miran **todos** los casos de su última decisión.
   - **Cada opción se puede conseguir:** la combinación sigue siendo válida.
   - Al maximizar, la primera es $\leq$ y la segunda $\geq$. Al minimizar, se invierten.
5. **Costo:** casillas por trabajo en cada una.

No cubrir una opción

Al maximizar, da de menos: el subproblema sin capacidad.

La receta: inducción sobre el orden de la tabla

1. **Qué significa la casilla**, en palabras. Eso es *OPT*.
2. **El orden.** Cada casilla leída va antes: permite la hipótesis, y el programa no lee vacío.
3. **Casos base**, contra la frase y no contra la fórmula.
4. **Paso, en dos direcciones.**
   - **Toda solución queda cubierta:** una óptima cualquiera; se miran **todos** los casos de su última decisión.
   - **Cada opción se puede conseguir:** la combinación sigue siendo válida.
   - Al maximizar, la primera es $\leq$ y la segunda $\geq$. Al minimizar, se invierten.
5. **Costo:** casillas por trabajo en cada una.

No cubrir una opción

Al maximizar, da de menos: el subproblema sin capacidad.

Una opción imposible

Al maximizar, da de más: la fila al revés suma dos veces un objeto.

Greedy o programación dinámica

	Greedy	Programación dinámica
Decisión	Una regla elige, sin mirar las demás	Se calculan todas, y se queda con la mejor
Estado	La solución parcial	Una tabla de subproblemas
Costo típico	El de ordenar	Casillas por trabajo en cada una
Demostración	Intercambio	Inducción sobre la tabla
Riesgo	Una regla plausible y falsa	Un subproblema sin la información necesaria

Si un greedy funciona, suele ser más simple. Pero sin demostración, la programación dinámica sirve de árbitro en instancias pequeñas.

Lo que hay que llevarse

- **El método:** subproblema en una frase, recurrencia, orden, reconstrucción, y demostración.
- **Intervalos con pesos:** $T[j] = \max(T[j-1],\ v + T[p(j)])$. $O(n \log n)$, con `upper_bound`.
- **Mochila 0/1:** $\Theta(nW)$. Sin c en el subproblema no se puede decidir. Una fila: la capacidad al revés.
- **Pseudopolinomial:** polinomial en el valor de W, exponencial en sus bits.
- **LCS:** si coinciden, emparejar no pierde nada. Si difieren, alguno sobra. $\Theta(nm)$.
- **Distancia de edición:** tres formas de la última columna. Tabla de $(n+1)$ por $(m+1)$.
- **Solo borrar e insertar:** $n + m - 2L$.

Para la próxima clase

- Leer los apuntes de la semana 7, y hacer sus ejercicios. La semana 8 es el examen parcial.
- **Práctica 2** en la sesión de laboratorio: greedy, Huffman, Dijkstra, árboles de expansión mínima y DFS.
- Opcional: un `diff` pequeño, en `lab/small_diff`.

Preguntas

Calculen a mano la tabla de distancia de edición entre `casa` y `cosas`.
Den una instancia de la mochila 0/1 donde tomar por mayor valor por peso no sea óptimo.
¿Qué pasa con el costo de $O(nW)$ si W pasa de tener 10 bits a 40?
¿Por qué se recorre la capacidad de mayor a menor cuando se usa una sola fila?
Definan en una frase el subproblema de los intervalos con pesos, y escriban su recurrencia.