

UNIVERSIDAD NACIONAL DE INGENIERÍA

Análisis y Diseño de Algoritmos

Cotas inferiores y primeros greedy

Ordenar comparando, quicksort con azar y clases en aulas

Gerson Garrido

Semana 4

Hoy

Sesión A

1. Qué es una cota inferior
2. El árbol de decisiones
3. $\log_2(n!) = \Omega(n \log n)$, completo
4. Counting sort no contradice la cota
5. Quicksort con pivote al azar

Sesión B

6. Greedy: decidir sin volver atrás
7. Tres reglas que fallan
8. La que termina antes, demostrada
9. El mínimo de aulas

> **Lectura**
>
> Apuntes de la semana 4. Código en `code/week-04`.

SECCIÓN 01

Qué es una cota inferior

01

Una afirmación sobre todos los algoritmos

Cota superior

Sobre un algoritmo: este programa nunca se pasa de tanto.

Se demuestra analizando ese programa.

Cota inferior ***(lower bound)***

Sobre un problema: ningún algoritmo de cierto tipo lo hace mejor en el peor caso.

Vale también para los que nadie ha inventado.

Hay que fijar el modelo

Qué operaciones puede hacer el algoritmo, y cuál se cuenta. Probar diez programas lentos no demuestra nada.

Calentamiento: hallar el máximo

Afirmación. Hallar el máximo de n números distintos comparando de a dos requiere al menos $n - 1$ comparaciones.

Técnica. Contar lo que debe ocurrir, sin mirar qué algoritmo es.

1. El algoritmo responde que el máximo es m. Tomemos $x \neq m$.
2. Si x nunca perdió, aumentamos x por encima de todos. Las respuestas no cambian.
3. Entonces el algoritmo vuelve a decir m, y ahora se equivoca.
4. Así que cada uno de los $n - 1$ elementos distintos de m perdió al menos una vez.
5. Cada comparación tiene un solo perdedor: al menos $n - 1$ comparaciones. ■

Problema cerrado

El ciclo de siempre hace exactamente $n - 1$. Nada que mejorar en este modelo.

SECCIÓN 02

El árbol de decisiones

02

El modelo: ordenar comparando

Ordenamiento por comparaciones ***(comparison sort)***

Solo conoce el orden de sus datos preguntando "¿a[i] < a[j]?". Se cuentan las preguntas.

- Lo son: insertion sort, mergesort, heapsort, quicksort, `std::sort`.
- No puede usar un elemento como índice, ni mirar sus bits, ni sumarlo.
- Primero: **deterministas**. Misma entrada, mismas comparaciones.
- Basta mirar n elementos **distintos**: son entradas válidas, y cada pregunta tiene solo dos respuestas.

Un algoritmo para tres elementos

pseudocódigo

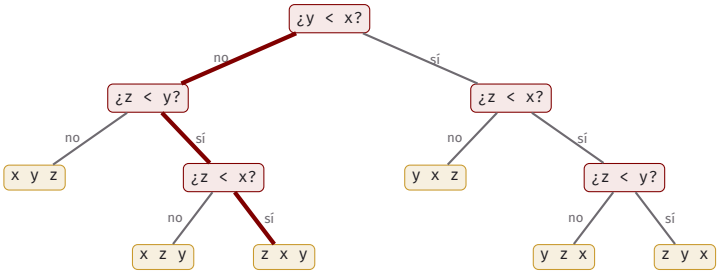
```
algoritmo ordenar_tres(a)
    si a[1] < a[0] entonces
        intercambiar a[0] y a[1]
    fin si
    si a[2] < a[1] entonces
        intercambiar a[1] y a[2]
        si a[1] < a[0] entonces
            intercambiar a[0] y a[1]
        fin si
    fin si
fin algoritmo
```

terminal

```
$ build/sort_three
entrada  respuestas  comparaciones  salida
1 2 3    NN          2              1 2 3
1 3 2    NSN         3              1 2 3
2 1 3    SN          2              1 2 3
2 3 1    NSS         3              1 2 3
3 1 2    SSN         3              1 2 3
3 2 1    SSS         3              1 2 3
```

Seis entradas, seis cadenas de respuestas distintas.

Todas las ejecuciones juntas



Con la entrada 2 3 1 (x=2, y=3, z=1): no, sí, sí. Hoja z x y: 1 2 3.

- Una ejecución es un camino de la raíz a una hoja.
- Comparaciones de esa ejecución: la profundidad de su hoja.
- Peor caso del algoritmo: la altura del árbol.

SECCIÓN 03

La cota inferior para ordenar

03

El teorema

Teorema

Para todo $n \geq 1$ y todo ordenamiento por comparaciones determinista, existe una entrada de n elementos distintos con la que el algoritmo hace al menos $\lceil \log_2(n!) \rceil$ comparaciones.

- **El orden importa:** para *cada* algoritmo hay una entrada mala. Puede ser distinta de un algoritmo a otro.
- **Es sobre el peor caso.** No dice nada de las demás entradas.
- **Técnica:** contar hojas. El árbol necesita muchas, y muchas hojas obligan a mucha altura.

Paso 1: al menos $n!$ hojas alcanzadas

1. Hay $n!$ formas de disponer n elementos distintos en la entrada.
2. Supongamos que dos distintas, P y Q, llegan a la misma hoja.
3. Mismas respuestas, así que mismos movimientos: la misma reorganización.
4. Aplicada a dos arreglos distintos, da dos salidas distintas.
5. A lo sumo una está ordenada, y el algoritmo falla con la otra. Imposible.
6. Luego las $n!$ disposiciones llegan a $n!$ hojas distintas.

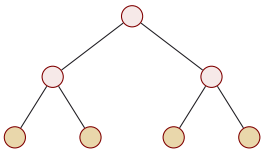
Hojas que nadie alcanza

El árbol completo puede tener más hojas y ramas que estas entradas no usan. Miramos solo la parte que recorren: tiene exactamente $n!$ hojas.

Paso 2 y Paso 3

Paso 2. Un árbol binario de altura h tiene a lo sumo 2^h hojas.

- $h = 0$: una sola hoja, y $2^0 = 1$.
- $h > 0$: la raíz tiene a lo sumo dos subárboles de altura a lo sumo $h - 1$. Cada uno tiene a lo sumo 2^{h-1} hojas.



$h = 2$: a lo sumo 4 hojas.

Paso 3. Sea h la altura de la parte recorrida:

$$n! \leq 2^h \quad \Longrightarrow \quad h \geq \log_2(n!)$$

Como h es entero, $h \geq \lceil \log_2(n!) \rceil$. La hoja más honda es una entrada concreta. ■

$\log_2(n!) = \Omega(n \log n)$, línea por línea

Técnica. Tirar los factores chicos de $n! = 1 \cdot 2 \cdots n$ y acotar por debajo los que quedan.

1. Nos quedamos con los factores de $\lceil n/2 \rceil$ a n.
2. Son $n - \lceil n/2 \rceil + 1 = \lfloor n/2 \rfloor + 1 \geq n/2$ factores, con n par o impar.
3. Cada uno vale al menos $n/2$. Por lo tanto, $n! \geq (n/2)^{n/2}$.
4. Con logaritmos: $\log_2(n!) \geq \frac{n}{2} \log_2 \frac{n}{2} = \frac{n}{2}(\log_2 n - 1)$.
5. Si $n \geq 4$, entonces $\log_2 n \geq 2$, y así $\log_2 n - 1 \geq \frac{1}{2} \log_2 n$.
6. Luego $\log_2(n!) \geq \frac{n}{4} \log_2 n$. ■

Y no se puede subir de orden

$n! \leq n^n$ da $\log_2(n!) \leq n \log_2 n$. Entonces $\log_2(n!) = \Theta(n \log n)$, y mergesort es óptimo en este modelo, salvo la constante.

Tres elementos: exactamente 3

- $\lceil \log_2 6 \rceil = 3$.
- Con 2 comparaciones: a lo sumo $2^2 = 4$ hojas.
- Hacen falta 6. Imposible con 2, sea cual sea el algoritmo.
- `ordenar_tres` usa a lo sumo 3.

Entonces 3 es el número exacto para $n = 3$.

No siempre tan justo

Para $n = 5$ la cota es 7, y mergesort llega a 8. La cota no promete que un algoritmo concreto la alcance. Dice que nadie baja de ahí.

La misma cota, como un juego

Adversario. No fija la entrada. Guarda las permutaciones coherentes, y responde lo que deje más.

- Cada respuesta deja al menos la mitad.
- Tras q preguntas quedan al menos $n!/2^q$.
- No se puede terminar con dos vivas: $q \geq \log_2(n!)$.
- La entrada mala depende del algoritmo.

terminal

```
$ build/adversary_demo
n = 4, inserción. Permutaciones posibles: 24
  ¿e1 < e0? sí, quedan 12
  ¿e2 < e0? sí, quedan 8
  ¿e2 < e1? sí, quedan 4
  ¿e3 < e0? sí, quedan 3
  ¿e3 < e1? sí, quedan 2
  ¿e3 < e2? sí, quedan 1
```

extracto

```
n = 8, cota 16
  inserción  28 comparaciones, posibles al final: 1, entrada: 7 6 5 4 3 2 1 0
  mergesort   17 comparaciones, posibles al final: 1, entrada: 7 3 5 1 6 2 4 0
```

Frente a algoritmos reales

extracto

```
$ build/lower_bound_table
 n       n!  cota  inserción       mergesort       hojas distintas
                   mejor  peor     mejor  peor     ins.     merge
 5      120     7       4    10       5     8      120       120
 8    40320    16       7    28      12    17    40320     40320
 9   362880    19       8    36      13    21   362880    362880
```

Las hojas valen $n!$: es el Paso 1. Las columnas “peor” nunca bajan de la cota. Las “mejor” sí.

Error común: “insertion sort rompe la cota”

La cota habla del **peor caso**: en cada algoritmo existe *alguna* entrada cara. Romperla exige ordenar **todas** con menos.

SECCIÓN 04

Counting sort

04



Ordenar sin comparar

pseudocódigo

```
algoritmo counting_sort(a, n, k)
    cuenta ⟵ arreglo de k ceros
    para i desde 0 hasta n - 1 hacer
        cuenta[a[i]] ⟵ cuenta[a[i]] + 1
    fin para
    siguiente ⟵ 0
    para v desde 0 hasta k - 1 hacer
        para c desde 1 hasta cuenta[v] hacer
            a[siguiente] ⟵ v
            siguiente ⟵ siguiente + 1
        fin para
    fin para
fin algoritmo
```

- Enteros de 0 a $k-1$. Costo $\Theta(n+k)$.
- ¿Contradice el teorema? **No: no compara.**
- `cuenta[a[i]]` usa el valor como índice: una operación con k resultados, no dos.
- El árbol binario no la modela, y el Paso 2 no se aplica.

El precio: enteros en un rango chico

terminal

```
$ build/counting_bench
        n          k  counting ms  std::sort ms
  1000000       1000          1.1          50.4
  4000000       1000          4.2         214.3
 16000000       1000         21.0         853.6
  1000000    1000000          9.6          88.6
  1000000   10000000         25.9          87.3
     1000   10000000         12.4           0.0
```

Con `-O2`, el menor de 5 intentos, en una máquina compartida. Los tiempos dependen de la máquina.

Una cota vale dentro de su modelo. Salir de él tiene condiciones: la última fila recorre diez millones de casillas para ordenar mil números.

SECCIÓN 05

Quicksort con pivote al azar

05

El algoritmo, ya conocido

pseudocódigo

```
algoritmo quicksort(a, lo, hi)
    si hi - lo < 2 entonces
        devolver
    fin si
    r ← número al azar entre lo y hi - 1
    intercambiar a[lo] y a[r]
    pivote ← a[lo]
    p ← lo
    para i desde lo + 1 hasta hi - 1 hacer
        si a[i] < pivote entonces
            p ← p + 1
            intercambiar a[i] y a[p]
        fin si
    fin para
    intercambiar a[lo] y a[p]
    quicksort(a, lo, p)
    quicksort(a, p + 1, hi)
fin algoritmo
```

- Rango semiabierto $[lo, hi)$.
- Al salir: $[lo, p)$ menores que el pivote, $[p + 1, hi)$ mayores o iguales.
- Compara, así que la cota le aplica.

La pregunta de hoy

¿Por qué elegir el pivote al azar protege de las entradas malas?

El azar está en el algoritmo, no en los datos

Pivote fijo, entrada al azar

Funciona bien en promedio *si* los datos llegan en orden uniforme.

Es una suposición sobre el mundo. Los datos reales suelen llegar casi ordenados.

Pivote al azar, entrada cualquiera

Las monedas se tiran después de recibir la entrada.

El promedio es sobre las monedas, y vale para cada entrada.

Nadie puede preparar la entrada mala

Un pivote malo puede salir, pero por mala suerte, y tiene que seguir saliendo muchas veces para hacer daño.

La intuición: la mitad de los pivotes son razonables



1. Pivote bueno: deja cada lado con a lo sumo $3m/4$, más o menos.
2. Un pivote al azar es bueno con probabilidad de al menos $1/2$, más o menos.
3. Sigamos a un elemento: tras t pivotes buenos, su rango mide a lo sumo $n(3/4)^t$. Llega a 1 con $t = \log_{4/3} n \approx 2{,}41 \log_2 n$.
4. Salen uno de cada dos, en promedio: unos $4{,}8 \log_2 n$ niveles. Se pasa de larga.
5. En cada nivel, a lo sumo n comparaciones. Se esperan $O(n \log n)$.

Esto es intuición, no demostración

El análisis formal necesita Probabilidad. En su lugar, medimos.

Tamaños por nivel, con 16 elementos ordenados

primer elemento

```
nivel 1: 16
nivel 2: 15
nivel 3: 14
nivel 4: 13
nivel 5: 12
nivel 6: 11
nivel 7: 10
nivel 8: 9
nivel 9: 8
nivel 10: 7
nivel 11: 6
nivel 12: 5
nivel 13: 4
nivel 14: 3
nivel 15: 2
nivel 16: .
```

al azar, semilla 7

```
nivel 1: 16
nivel 2: 15
nivel 3: 7 7
nivel 4: 6 . 5
nivel 5: 2 3 3 .
nivel 6: . 2 2
nivel 7: . .
```

Un punto es un rango de un elemento.

Al azar también salió un pivote malo: el nivel 1 dejó 15.

El siguiente partió 15 en 7 y 7. Costó un nivel, no arruinó la ejecución.

Medir: los conteos son exactos

extracto

```
$ build/quicksort_levels
pivote y partición  entrada       n  comparac.  niveles  c/nlogn      ms
primero, en dos     ordenada    16000   127992000   15999  572.79    80.8
azar, en dos        ordenada    16000      270137      31    1.21     0.8

primero, en dos     ordenada    64000  2047968000   63999 2004.25  1177.5
primero, en dos     al azar     64000     1208356      38    1.18     5.4
azar, en dos        ordenada    64000     1245835      37    1.22     3.0
azar, en dos        al azar     64000     1182428      36    1.16     6.0
```

Semilla fija: los conteos se repiten. Los tiempos son de una corrida con `-O2` y dependen de la máquina.

- Primero, ordenada: exactamente $n(n-1)/2$ comparaciones y $n-1$ niveles.
- Misma entrada, pivote al azar: más de mil veces menos.
- $n \times 4$: el cuadrático crece $\times 16$; al azar, cerca de $\times 4{,}6$.
- Primero, entrada al azar: le va bien. Y eso engaña, porque depende de los datos.

Solo cambian las monedas

terminal

```
entrada ordenada, n = 64000, 1000 semillas
comparaciones: mínimo 1145604, promedio 1235073, máximo 1382522
niveles:       mínimo 33, promedio 37, máximo 46
pivotes buenos, en llamadas de 16 o más elementos: 52.1 %
```

- En mil corridas, el máximo está un 12 % sobre el promedio.
- Niveles: promedio 37, por debajo de la estimación gruesa de unos 77.
- Pivotes buenos: un poco más de la mitad, como se esperaba.
- El peor caso sigue siendo $\Theta(n^2)$: requiere una racha de monedas muy improbable.

El azar no arregla todo

extracto

pivote y partición	entrada	n	comparac.	niveles	c/nlogn	ms
azar, en dos	iguales	64000	2047968000	63999	2004.25	1075.3
azar, en tres	iguales	64000	127998	1	0.13	0.1
azar, en tres	ordenada	64000	1945879	37	1.90	3.8

Error común

La cuenta de pivotes buenos supone elementos **distintos**. Con todos iguales, esta partición deja todo a la derecha.

- La partición en tres, de Estructuras de Datos: menores, iguales, mayores.
- Los iguales ya no se tocan: 1 nivel. Con dos preguntas por elemento, sube a 1,9.

¿Y la cota inferior? ¿Y `std::sort`?

- Con las monedas fijas, quicksort es un algoritmo determinista: tiene una entrada con al menos $\lceil \log_2(n!) \rceil$ comparaciones.
- Esa entrada mala depende de las monedas. Nadie la prepara sin conocerlas.
- Nuestro teorema es para deterministas. También se sabe que el número *esperado* es $\Omega(n \log n)$ en la peor entrada: aquí lo enunciamos sin demostrarlo.

En C++

`std::sort` debe hacer $O(n \log n)$ comparaciones desde C++11. En libstdc++ es introsort (`__introsort_loop`): un quicksort que cambia a heapsort si la recursión se hunde.

SECCIÓN 06

Greedy: selección de intervalos

06

Decidir sin volver atrás

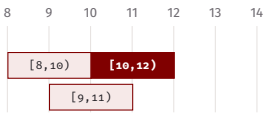
Greedy

En cada paso toma la opción que hoy parece mejor según una regla, y nunca la deshace.

- Muy corto y muy rápido. Esa es la mitad fácil.
- La mitad difícil: la mayoría de las reglas que suenan bien son incorrectas.
- Probar con unos cuantos ejemplos no alcanza para detectarlo.
- En este curso: todo greedy viene con su demostración, y antes, con una regla parecida que falla.

Clases en una sola aula

Se quiere dictar **la mayor cantidad posible** de clases, sin que dos se crucen.



se tocan: compatibles

se cruza con ambas

Intervalo semiabierto $[s, f)$**, con** $s < f$

Ocupa el aula desde s, incluido, hasta justo antes de f. Una clase que termina a las 10 no se cruza con una que empieza a las 10.

Se cruzan, válida, óptima

Se cruzan

$[s_1, f_1)$ y $[s_2, f_2)$ se cruzan si y solo si $s_1 < f_2$ y $s_2 < f_1$. Si no, son compatibles.

intervals.h

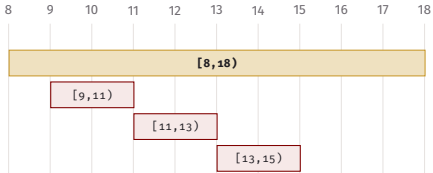
```
1  inline bool overlap(const Interval& a, const Interval& b) {
2      return a.start < b.finish && b.start < a.finish;
3  }
4  // assert(!overlap({8, 10}, {10, 12}));   el borde no es un cruce
```

Válida y óptima son cosas distintas

Válida: compatibles de a pares. **Óptima:** válida y con el mayor número de clases.

Primera regla: la que empieza antes

No dejar el aula vacía. Pero:

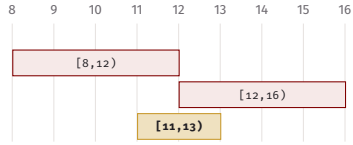


extracto

```
Contra empieza antes:
  clases: [8,18) [9,11) [11,13) [13,15)
  óptimo por fuerza bruta: 3
  empieza antes: 1 (válida) -> [8,18)
  termina antes: 3 (válida) -> [9,11) [11,13) [13,15)
```

Segunda regla: la que dura menos

Si la larga era el problema, tomemos las cortas. Pero una corta puede estorbar a dos largas:

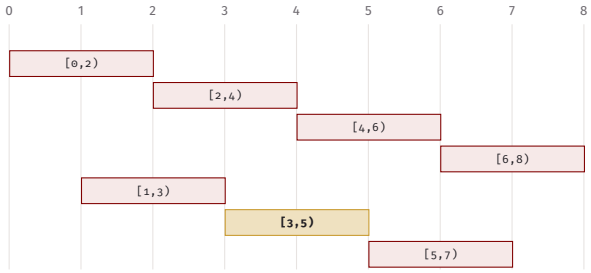


extracto

```
Contra dura menos:
  clases: [8,12) [12,16) [11,13)
  óptimo por fuerza bruta: 2
  dura menos   : 1 (válida) -> [11,13)
  termina antes: 2 (válida) -> [8,12) [12,16)
```

Tercera regla: la que tiene menos cruces

Tomar la que menos estorba, sacar las que se cruzan con ella, y repetir volviendo a contar.



- $[3, 5)$ tiene 2 cruces, y es la única con tan pocos. Se toma primero, sin empate posible.
- No está en ninguna solución óptima: la primera decisión ya perdió.

La tercera regla, corrida

extracto

```
Contra menos cruces:
  clases: [0,2) [2,4) [4,6) [6,8) [1,3) [1,3) [1,3) [3,5) [5,7) [5,7) [5,7)
  óptimo por fuerza bruta: 4
  menos cruces : 3 (válida) -> [3,5) [0,2) [6,8)
  termina antes: 4 (válida) -> [0,2) [2,4) [4,6) [6,8)

En esa lista, cruces de cada clase:
  [0,2): 3
  [2,4): 4
  [1,3): 4  (no está en ninguna solución óptima)
  [3,5): 2  (no está en ninguna solución óptima)
Menos cruces, con cualquier forma de romper empates: entre 3 y 3
```

Sin depender de los empates: el programa prueba todas las formas de romperlos.

Válida no es óptima

terminal

```
$ build/rules_random
20000 instancias de 2 a 12 clases
respuestas inválidas de cualquier regla: 0
instancias en que la regla no llega al óptimo:
  empieza antes   5423  (27.11 %)
  dura menos       684  (3.42 %)
  menos cruces       2  (0.01 %)
  termina antes      0  (0.00 %)
```

Error común

“Nunca toma dos clases que se crucen, así que es correcto.” Vale para las tres reglas falsas. Hay que demostrar que **ninguna otra solución válida es más grande**.

Y una prueba al azar tampoco basta: “menos cruces” falla 2 veces de 20 000.

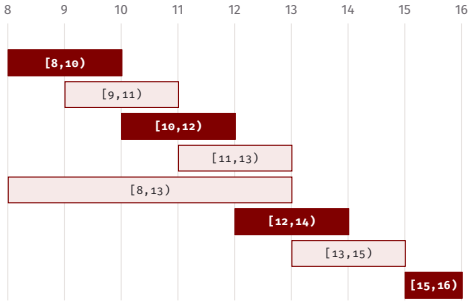
La regla correcta: la que termina antes

pseudocódigo

```
algoritmo seleccionar_clases(clases, n)
    ordenar clases por hora de fin
    elegidas ← lista vacía
    libre ← -∞
    para i desde 0 hasta n - 1 hacer
        si libre ≤ clases[i].inicio entonces
            agregar clases[i] a elegidas
            libre ← clases[i].fin
        fin si
    fin para
    devolver elegidas
fin algoritmo
```

- Lo que una clase le quita al resto del día es el aula hasta su hora de fin.
- El $\leq$ dice que se puede empezar justo cuando la anterior termina.
- Basta comparar con la última elegida: es la que termina más tarde.
- Costo: $O(n \log n)$, por ordenar.

Paso a paso, en orden de fin



Clases ordenadas por hora de fin. Empezamos con $\texttt{libre} = -\infty$. Se toma $[8, 10)$. Ahora $\texttt{libre} = 10$. Se descarta $[9, 11)$: $10 \leq 9$ es falso. $10 \leq 10$: se toma $[10, 12)$, justo en el borde. $\texttt{libre} = 12$. Se descartan $[11, 13)$ y $[8, 13)$. $12 \leq 12$: se toma $[12, 14)$, otra vez en el borde. $\texttt{libre} = 14$. Se descarta $[13, 15)$. $14 \leq 15$: se toma $[15, 16)$. Son **4 clases**, igual que la fuerza bruta. Con el borde como cruce serían 3.

Qué vamos a demostrar

Afirmación 1: validez

Las elegidas $g_1, \ldots, g_k$ son compatibles. Cada una empieza en o después del fin de la anterior.

Afirmación 2: optimalidad

Ninguna solución válida tiene más de k clases. Esta es la que separa la regla correcta de las falsas.

- **Técnica:** argumento de intercambio *(exchange argument)*. Transformar una óptima en la del greedy sin que deje de ser válida ni pierda clases.
- **Preparación:** $O = \{x_1, \ldots, x_m\}$ óptima, ordenada por inicio, que también es el orden por fin. Como el greedy es válido, $m \geq k$. Queremos $m = k$.

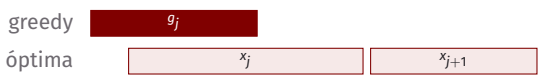
La primera diferencia

Supongamos $x_i = g_i$ para $i < j$, y $x_j \neq g_j$, con $j \leq k$.

Primero: g_j termina a más tardar cuando termina x_j.

1. El greedy eligió g_j como la primera, en orden de fin, que empieza en o después de $f(g_{j-1})$.
2. x_j cumple esa condición: $s(x_j) \geq f(x_{j-1}) = f(g_{j-1})$. Si $j = 1$, no hay condición.
3. Entonces x_j era candidata, y el greedy eligió a la de menor fin:

$$\boxed{f(g_j) \leq f(x_j)}$$



El cambio no rompe nada

Sea $O' = \{x_1, \ldots, x_{j-1},\ g_j,\ x_{j+1}, \ldots, x_m\}$.

- **Hacia atrás:** g_j empieza en o después de $f(g_{j-1}) = f(x_{j-1})$, porque el greedy lo exigió.
- **Hacia adelante:** toda x_l con $l > j$ empieza en o después de $f(x_j) \geq f(g_j)$.
- **Mismo tamaño:** g_j no es ninguna de esas x_l, que empiezan en o después de $f(g_j)$.

Resultado

O' es válida y tiene m clases, así que también es óptima, y coincide con el greedy en una posición más. Repetimos hasta k veces.

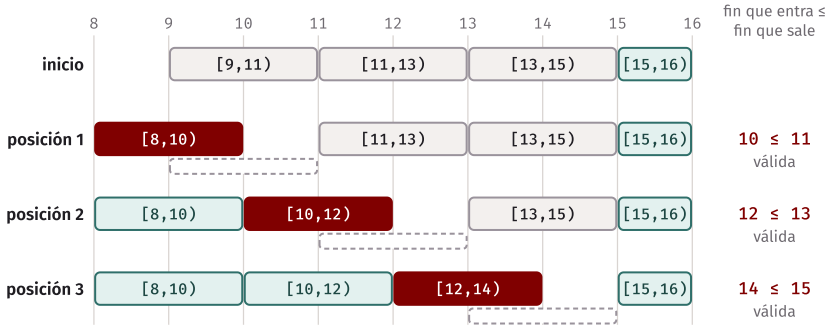
El intercambio, con las ocho clases

extracto

```
Selección (4 clases, óptimo 4): [8,10) [10,12) [12,14) [15,16)
Intercambio, empezando en: [9,11) [11,13) [13,15) [15,16)
  posición 1: [8,10) [11,13) [13,15) [15,16)  válida
  posición 2: [8,10) [10,12) [13,15) [15,16)  válida
  posición 3: [8,10) [10,12) [12,14) [15,16)  válida
```

- En cada paso, la clase que entra termina antes que la que sale.
- Por eso lo que viene después sigue cabiendo.
- Al final, la solución óptima se convirtió en la del greedy sin perder ninguna clase.

El intercambio, sobre el eje de tiempo



La que entra termina antes que la que sale, y por eso lo siguiente sigue cabiendo.

El último paso, que no se puede omitir

Tenemos O^*, óptima, cuyas primeras k clases son $g_1, \ldots, g_k$. ¿Tiene clases de más?

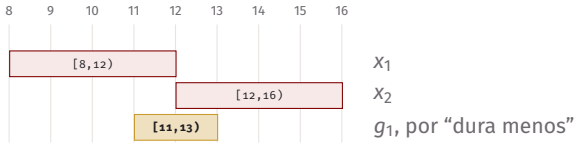
1. Supongamos que $m > k$. Entonces existe x_{k+1}, y empieza en o después de $f(g_k)$.
2. Termina después de empezar: $f(x_{k+1}) > f(g_k)$. Aparece después de g_k en el orden del greedy.
3. Cuando el recorrido llegó a ella, `libre` valía $f(g_k)$, porque g_k fue la última elegida.
4. La condición se cumplía, y el greedy la habría tomado. Contradicción.
5. Luego $m = k$. ■

El patrón

Tomar una óptima. Cambiar su primera diferencia por la elección greedy. Comprobar que sigue siendo válida y no empeora. Repetir. Revisar que no sobre nada.

Dónde se rompen las reglas falsas

Toda la demostración descansa en una línea: $f(g_j) \leq f(x_j)$.



- Aquí $f(g_1) = 13$ y $f(x_1) = 12$: la línea es falsa.
- Al cambiar x_1 por g_1, esta choca con x_2.
- “Empieza antes” y “menos cruces” se rompen en el mismo sitio.

SECCIÓN 07

Partición de intervalos

07

Dictar todas: el mínimo de aulas

Ahora no se descarta nada, y lo que se cuenta son las aulas.

Profundidad *(depth)*

El mayor número de intervalos que contienen un mismo instante. $[s, f)$ contiene a t si $s \leq t < f$.

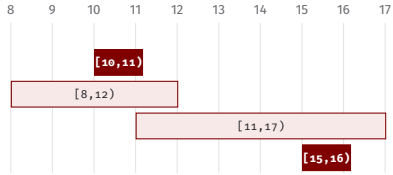
Afirmación. Si la profundidad es d, hacen falta al menos d aulas.

- d clases se dictan a la vez en t, así que se cruzan de a pares.
- Dos que se cruzan no comparten aula. Con menos de d aulas, dos caerían juntas. ■

Otra cota inferior

Vale para toda asignación. Si en algún momento hay 4 clases a la vez, hacen falta al menos 4 aulas. ¿Bastan?

Una idea que falla: llenar aula por aula



primera aula: la selección óptima
quedan dos que se cruzan: 3 aulas

extracto

```
Clases [10,11) [8,12) [11,17) [15,16)
  llenar aula por aula con la selección: 3 aulas
  por orden de inicio: 2 aulas
  mínimo por fuerza bruta: 2 aulas
  máximo simultáneo: 2
```

Bastaban $\{[8,12),[15,16)\}$ y $\{[10,11),[11,17)\}$. Lo mejor para la primera aula perjudicó a las demás.

El algoritmo: por hora de inicio

pseudocódigo

```
algoritmo asignar_aulas(clases, n)
    ordenar clases por hora de inicio
    H ← heap de mínimos: cuándo se libera cada aula
    aulas ← 0
    para i desde 0 hasta n - 1 hacer
        si H no está vacío y mínimo de H ≤ clases[i].inicio
      entonces
            a ← aula del mínimo de H
            sacar el mínimo de H
        sino
            a ← aulas
            aulas ← aulas + 1
        fin si
        aula[i] ← a
        insertar (clases[i].fin, a) en H
    fin para
    devolver aulas
fin algoritmo
```

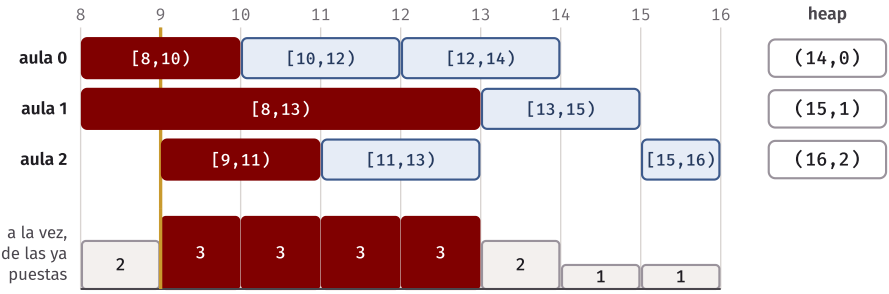
- Cada clase va a un aula libre, si hay.
- Solo se abre una nueva si no hay ninguna libre.
- Si alguna está libre, la que se libera primero también.
- Costo: $O(n \log n)$.

Paso a paso, en orden de inicio

Clase	Mínimo	¿Libre?	Aula	Heap después
$[8, 10)$	vacío	no	0, nueva	(10,0)
$[8, 13)$	(10,0)	$10 \leq 8$, no	1, nueva	(10,0) (13,1)
$[9, 11)$	(10,0)	$10 \leq 9$, no	2, nueva	(10,0) (11,2) (13,1)
$[10, 12)$	(10,0)	$10 \leq 10$, sí	0	(11,2) (12,0) (13,1)
$[11, 13)$	(11,2)	$11 \leq 11$, sí	2	(12,0) (13,1) (13,2)
$[12, 14)$	(12,0)	$12 \leq 12$, sí	0	(13,1) (13,2) (14,0)
$[13, 15)$	(13,1)	$13 \leq 13$, sí	1	(13,2) (14,0) (15,1)
$[15, 16)$	(13,2)	$13 \leq 15$, sí	2	(14,0) (15,1) (16,2)

3 aulas, y el máximo simultáneo también es 3: a las 9, por ejemplo, hay $[8, 10)$, $[8, 13)$ y $[9, 11)$.

Las aulas y la profundidad



A las 9 hay [8, 10), [8, 13) y [9, 11) a la vez: hacen falta 3 aulas. El algoritmo usa 3: también bastan.

Profundidad 3 = 3 aulas.

Demostración: alcanzar la cota

Teorema. `asignar_aulas` da una asignación válida con exactamente d aulas.

- **Validez.** Una clase entra a un aula solo si esta se libera en o antes de su inicio. Dentro de un aula, cada clase empieza en o después del fin de la anterior.
- **A lo sumo d.** Se abre el aula número k al procesar la clase c.
  - Las $k-1$ aulas abiertas se liberan después de $s(c)$. Cada una tiene una última clase c' con $f(c') > s(c)$.
  - c' ya se procesó, y se procesa por inicio: $s(c') \leq s(c)$. Es decir, c' contiene a $s(c)$.
  - Esas $k-1$ clases y c son k clases en el instante $s(c)$. Luego $d \geq k$.
- **Al menos d.** La cota inferior vale para toda asignación. ■

> **Otra forma de demostrar un greedy**
>
> Hallar una cota que toda solución debe pagar, y mostrar que el algoritmo la paga exactamente.

Contar sin asignar, y el borde otra vez

max_simultaneous, en intervals.h

```cpp
std::vector<std::pair<int, int>> events;
for (const Interval& c : classes) {
    events.push_back({c.start, +1});
    events.push_back({c.finish, -1});
}
std::sort(events.begin(), events.end());
int now = 0;
int most = 0;
for (const auto& event : events) {
    now += event.second;
    most = std::max(most, now);
}
```

Error común

Procesar los inicios antes que los fines cuando coinciden. Con $[8, 10)$ y $[10, 12)$, cuenta 2 a la vez y da un aula de más.

Como $-1 < +1$, ordenar los pares deja los fines primero. Es lo que dice el intervalo semiabierto.

El patrón de una demostración greedy

1. Escribir la regla con precisión, con los empates incluidos.
2. Buscar un contraejemplo a la regla y a las parecidas. Si no aparece, no prueba nada.
3. Demostrar la validez. Suele ser corta.
4. Demostrar la optimalidad:
   - **Intercambio:** transformar una óptima en la del greedy sin empeorarla.
   - **Cota inferior alcanzada:** un número que toda solución paga, y el greedy paga exactamente.
5. Analizar el costo, que casi siempre es el de ordenar.

Un cambio pequeño lo rompe

Si cada clase tiene un valor y se busca el máximo valor total, “termina antes” ya no sirve. Eso se resuelve con programación dinámica, más adelante.

Para la próxima clase

- Leer los apuntes de la semana 4, y hacer sus ejercicios.
- **Práctica 1** en la sesión de laboratorio: invariantes e inducción, recurrencias y divide y vencerás.
- Opcional: el problema de horarios y aulas, en `lab/class_schedule`.

Preguntas

¿Por qué hacen falta 3 comparaciones para ordenar 3 elementos?
¿Qué línea de la demostración de "termina antes" deja de valer con "dura menos"?
Si en algún momento hay 4 clases a la vez, ¿por qué bastan 4 aulas si nunca hay más?