

UNIVERSIDAD NACIONAL DE INGENIERÍA

Estructuras de Datos

Ordenamiento y búsqueda binaria

Pagar una vez por ordenar, y buscar en $\Theta(\log n)$

Gerson Garrido

Semana 6

Hoy

1. Mergesort y su árbol de llamadas
2. Quicksort: partición, peor caso y promedio
3. Pivote al azar y partición en tres
4. Estabilidad
5. La cota $\Omega(n \log n)$
6. Qué usan las bibliotecas, y los comparadores
7. Búsqueda binaria con invariante
8. Bytes, UTF-8 y rangos de prefijos

> **Lectura**
>
> Apuntes de la semana 6. Código en `code/week-06`.

El problema: autocompletar

- 28 039 palabras, cada una con su frecuencia.
- El usuario escribe `ca`. Hay que proponer las 5 más frecuentes que empiezan así.
- **Opción 1:** revisar todo el archivo en cada consulta.
- **Opción 2:** ordenar una vez, y hallar los candidatos con unas 30 lecturas.

Idea clave

Ordenar es una inversión: cuesta $\Theta(n \log n)$ una vez, y deja cada búsqueda en $\Theta(\log n)$.

Qué significa ordenar

Ordenar *(sorting)*

Reacomodar a[0], …, a[n − 1] de modo que para ningún i ocurra que a[i + 1] va antes que a[i].

- El resultado es una **permutación**: no se pierde ni se inventa nada.
- La condición admite elementos **equivalentes**: dos personas con la misma nota.
- Hoy: algoritmos **por comparación**. Solo preguntan "¿a va antes que b?".
- Inserción cuesta $\Theta(n^2)$ en el peor caso. Para bajar de ahí hace falta dividir.

SECCIÓN 01

Mergesort

01



Mergesort: dividir, ordenar y mezclar

pseudocódigo

```
algoritmo merge_sort(a, lo, hi)
    si hi - lo < 2 entonces
        devolver
    fin si
    mid ← lo + (hi - lo) div 2
    merge_sort(a, lo, mid)
    merge_sort(a, mid, hi)
    mezclar(a, lo, mid, hi)
fin algoritmo
```

- Rangos semiabiertos: `lo` entra y `hi` no.
- Caso base: 0 o 1 elementos ya están ordenados.
- Observación clave: mezclar dos arreglos **ya ordenados** es fácil.
- Como juntar dos pilas de exámenes: se mira el primero de cada una, y se saca el menor.

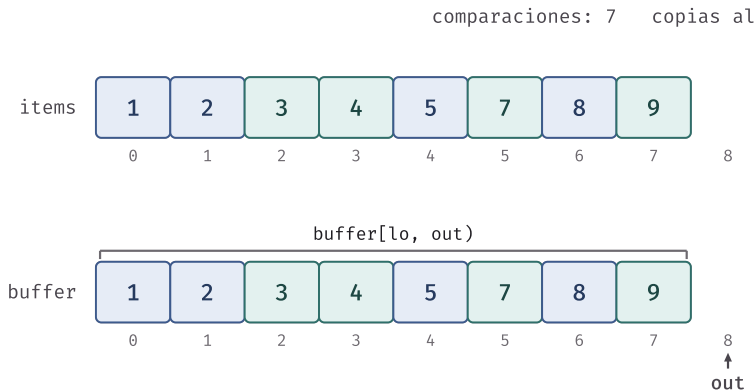
La mezcla

int_sorts.h

```
1  std::size_t left = lo;
2  std::size_t right = mid;
3  std::size_t out = lo;
4  while (left < mid && right < hi) {
5      comparisons++;
6      if (items[right] < items[left]) {
7          buffer[out++] = items[right++];
8      } else {
9          buffer[out++] = items[left++];
10     }
11 }
12 while (left < mid) {
13     buffer[out++] = items[left++];
14 }
15 while (right < hi) {
16     buffer[out++] = items[right++];
17 }
```

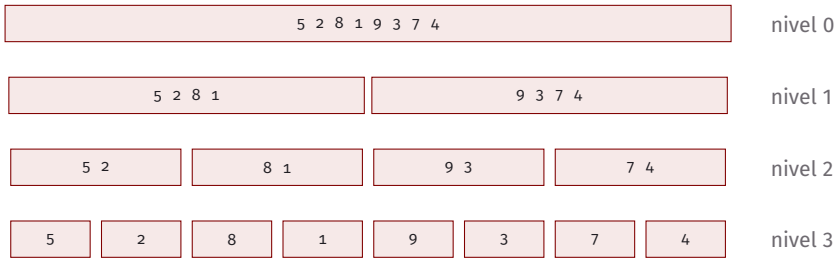
- **Invariante:** `buffer[lo, out)` tiene en orden los más pequeños.
- Cada comparación saca un elemento: a lo sumo $m - 1$ comparaciones.
- Exactamente m copias al buffer.
- Se toma el de la derecha solo si es estrictamente menor. En un empate gana el de la izquierda.

Una mezcla, paso a paso



Al final, el buffer se copia de vuelta a items: 7 comparaciones y 8 copias al buffer

El árbol de llamadas con n = 8



- Los rangos de un nivel no se tocan entre sí, y juntos tienen los n elementos.
- Los tamaños son 8, 4, 2, 1: se divide por 2 unas $\log_2$ n veces.

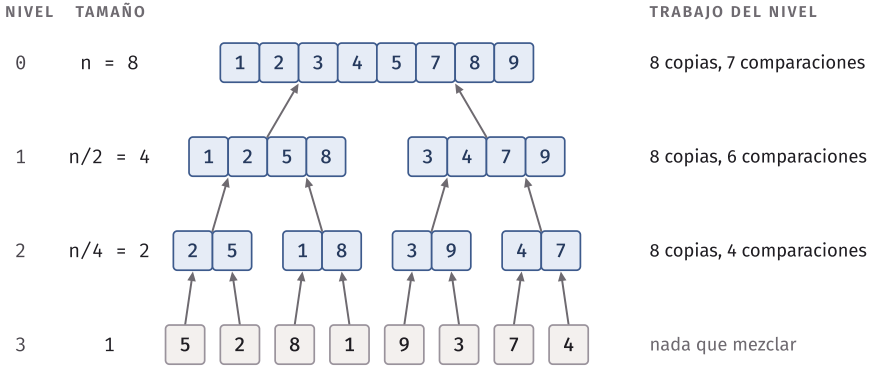
De abajo arriba: las mezclas

terminal

```
$ build/trace_merge
dividir
  nivel 0: [5 2 8 1 9 3 7 4]
  nivel 1: [5 2 8 1] [9 3 7 4]
  nivel 2: [5 2] [8 1] [9 3] [7 4]
  nivel 3: [5] [2] [8] [1] [9] [3] [7] [4]
mezclar
  nivel 2: [2 5] [1 8] [3 9] [4 7]
           4 comparaciones, 8 elementos copiados
  nivel 1: [1 2 5 8] [3 4 7 9]
           6 comparaciones, 8 elementos copiados
  nivel 0: [1 2 3 4 5 7 8 9]
           7 comparaciones, 8 elementos copiados
```

- 8 copias **en cada nivel**.
- Menos de 8 comparaciones en cada nivel.
- $\log_2$ n niveles $\times$ Θ(n) de trabajo.
- Θ(n log n) en cualquier caso, sin recurrencias ni inducción.

Se suma por niveles



3 niveles con mezclas × 8 copias = 24 copias, y 4 + 6 + 7 = 17 comparaciones

Lo que cuesta: memoria extra

- Si la mezcla escribiera sobre `items`, pisaría lo que todavía no leyó.
- Buffer de **n elementos**, más unos $\log_2 n$ marcos de la recursión. No es “en el lugar”.
- El buffer se reserva una sola vez afuera. Si cada mezcla reservara el suyo: mismo O, y unas n reservas.
- A cambio: $\Theta(n \log n)$ en el peor caso, y es **estable**.

Idea clave

Para analizar un algoritmo que divide, sumen por niveles, no por llamadas. Las llamadas de un nivel son muchas y chicas, pero juntas tocan cada elemento una vez.

SECCIÓN 02

Quicksort

02

Quicksort: lo mismo, al revés

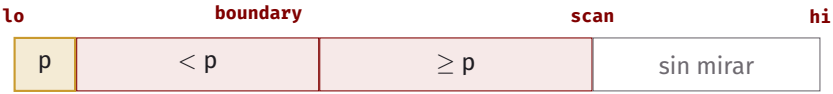
pseudocódigo

```
algoritmo quick_sort(a, lo, hi)
    si hi - lo < 2 entonces
        devolver
    fin si
    p ← partir(a, lo, hi)
    quick_sort(a, lo, p)
    quick_sort(a, p + 1, hi)
fin algoritmo
```

- Mergesort: dividir es trivial, el trabajo está en juntar.
- Quicksort: el trabajo está en dividir, y juntar es trivial.
- No necesita buffer.

Partir *(partition)* respecto de un pivote p: primero los menores, luego p, y luego los mayores o iguales. p queda en su lugar definitivo.

La invariante de la partición

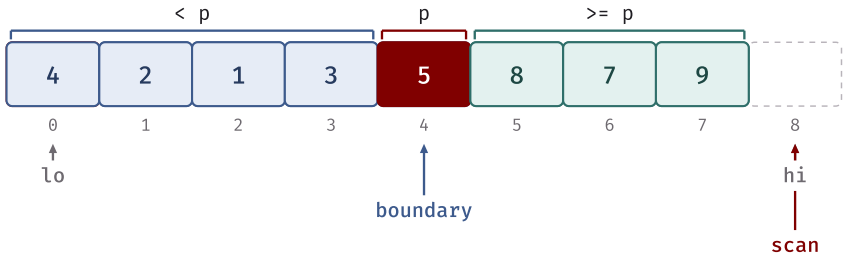


int_sorts.h

```
1  int pivot = items[lo];
2  std::size_t boundary = lo;
3  for (std::size_t scan = lo + 1; scan < hi; scan++) {
4      comparisons++;
5      if (items[scan] < pivot) {
6          boundary++;
7          std::swap(items[boundary], items[scan]);
8      }
9  }
10 std::swap(items[lo], items[boundary]);
```

- **Al inicio:** las dos zonas están vacías.
- **En cada paso:** si es $\geq$ p ya está bien. Si es menor, cambia con el primero de la zona $\geq$.
- **Al final:** no queda nada sin mirar. El último `swap` pone al pivote entre las zonas.

La partición, paso a paso



scan llegó a hi: el 5 se intercambia con items[boundary] y queda en su lugar definitivo

Sigámosla

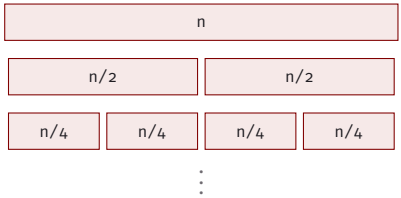
terminal

```
$ build/trace_partition | head -9
pivote = primer elemento = 5
  scan=1 boundary=1  5 2 8 1 9 3 7 4
  scan=2 boundary=1  5 2 8 1 9 3 7 4
  scan=3 boundary=2  5 2 1 8 9 3 7 4
  scan=4 boundary=2  5 2 1 8 9 3 7 4
  scan=5 boundary=3  5 2 1 3 9 8 7 4
  scan=6 boundary=3  5 2 1 3 9 8 7 4
  scan=7 boundary=4  5 2 1 3 4 8 7 9
  pivote a su lugar 4 2 1 3 5 8 7 9  (posición 4)
```

- El 5 quedó donde estará al final.
- `4 2 1 3` y `8 7 9` no están ordenados, pero ya no se mezclarán entre sí.

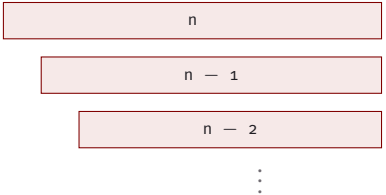
La forma del árbol lo decide todo

Pivote en el medio



$\log_2 n$ niveles con unas n comparaciones: $\Theta(n \log n)$.

Pivote siempre un extremo



$(n-1) + (n-2) + \ldots + 1 = n(n-1)/2$: $\Theta(n^2)$.

¿Qué entrada hace eso con el primer elemento?

Una **ya ordenada**: es siempre el menor. La inversa alterna entre el mayor y el menor. Y con todos iguales, ninguno es menor.

El conteo coincide exactamente

terminal

```
$ build/count_sorts | sed -n '9,13p'
quicksort con el primero como pivote, entrada ordenada
     n  comparac.     n(n-1)/2
  1000     499500       499500
  2000    1999000      1999000
  4000    7998000      7998000
```

- Al duplicar n, las comparaciones se multiplican por 4. Es el sello de $\Theta(n^2)$.
- Es un conteo, no un tiempo: no depende de la máquina.

Y además, la pila

La recursión también tiene n niveles. Aquí, con la pila reducida a 1 MB para no esperar.

terminal

```
$ build/deep_recursion 20000
ordenó 20000 elementos
$ ulimit -s 1024; build/deep_recursion 20000 2>&1 | sed -n '3p;$p' | cut -c1-74
==26406==ERROR: AddressSanitizer: stack-overflow on address 0xffffe0083ff0
==26406==ABORTING
```

- Con los 8 MB normales, 100 000 elementos dan el mismo error, tras más de medio minuto.

Error común

Creer que el peor caso "solo" es lento. Puede **tumbar el programa**.

El caso promedio: contar pares

Sean $z_1 < z_2 < \ldots < z_n$ los elementos en orden, distintos. Tomen el grupo $z_i, \ldots, z_j$:



- Dos elementos se comparan solo si uno es el pivote, y a lo sumo una vez.
- Si el primer pivote del grupo es z_i o z_j, el par se compara.
- Si es uno del medio, se separan para siempre.
- P(se comparan) = 2 / (j − i + 1). Los vecinos siempre; los lejanos casi nunca.
- Sumando: alrededor de $2n \ln n \approx 1.39\, n \log_2 n$.

Se probó contando: con n = 100 000, razón 0.878 respecto de $2n \ln n$, y se acerca a 1 lentamente (apuntes).

SECCIÓN 03

Pivote al azar

03

Pivote al azar: dos líneas

int_sorts.h

```
1  std::uniform_int_distribution<std::size_t> pick(lo, hi - 1);
2  std::swap(items[lo], items[pick(rng)]);
3  std::size_t p = partition_first(items, lo, hi, comparisons);
```

- Lo aleatorio pasa a ser **el algoritmo**, no los datos.
- Con elementos distintos, toda entrada cuesta $\Theta(n \log n)$ en esperanza.
- Una mala racha de $\Theta(n^2)$ sigue existiendo, pero depende de la suerte, no de lo que mandó el usuario.
- `uniform_int_distribution` usa el rango **cerrado** [a, b]: por eso va `hi - 1`.

Todo junto: comparaciones con n = 4096

terminal

```
$ build/count_sorts | sed -n '1,7p'
comparaciones con n = 4096
entrada        merge q.primero    q.azar    q.tres std::sort    stable
al azar        43983      56479     55320     88183     59551     48346
ordenada       24576    8386560     61392     94040     55831     28668
inversa        24576    8386560     56008     87440     41482     21654
10 valores     42540     850773    855607     26340     41311     47160
iguales        24576    8386560   8386560      8192     37643     28668
```

- `merge`: siempre por debajo de $n \log_2 n = 49\,152$.
- `q.primero`: 8 386 560 es $n(n-1)/2$.
- `q.azar` arregla `ordenada`. Pero miren `iguales`: el azar no ayuda.

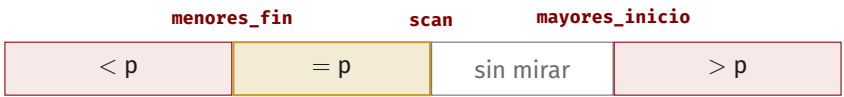
SECCIÓN 04

Elementos repetidos

04

Los empates son el problema normal

- Nuestra partición manda a la derecha todo lo que es $\geq$ pivote.
- Todos iguales: cada partición saca solo al pivote, sin importar cuál se elija.
- Con solo 10 valores distintos: más de 850 000 comparaciones.
- Y es lo común: de nuestras 28 039 palabras, 14 110 aparecen una sola vez.



Los iguales ya están en su lugar definitivo. Se recurre solo sobre menores y mayores.

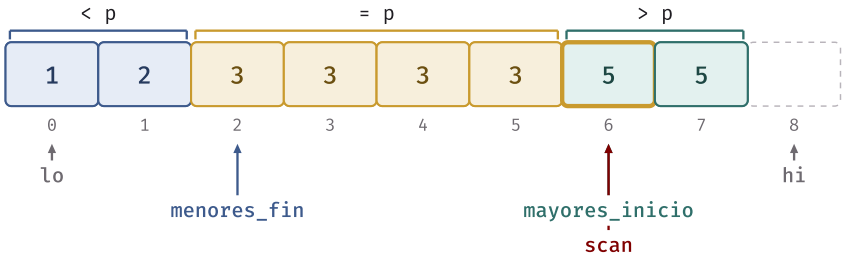
La partición en tres

pseudocódigo

```
mientras scan < mayores_inicio hacer
    si a[scan] < p entonces
        intercambiar a[menores_fin] con a[scan]
        menores_fin ← menores_fin + 1
        scan ← scan + 1
    sino si p < a[scan] entonces
        mayores_inicio ← mayores_inicio - 1
        intercambiar a[scan] con a[mayores_inicio]
    sino
        scan ← scan + 1
    fin si
fin mientras
```

- **Mayor:** `scan` no avanza. Lo que llegó desde el final no se ha mirado.
- **Menor:** sí avanza. Lo que llega desde `menores_fin` es un igual ya revisado.
- Todos iguales: **una pasada**. En la tabla, de 8 386 560 a 8192.
- El precio: dos preguntas por elemento. Con datos distintos, 88 183 contra 55 320.

La partición en tres, paso a paso



items[6] = 5 es mayor. scan = mayores_inicio: los cuatro 3 no vuelven a tocarse

Sigámosla

terminal

```
$ build/trace_partition | sed -n '10,18p'
pivote = 3
  menores_fin=0 scan=1 mayores_inicio=8  3 5 3 1 3 2 5 3
  menores_fin=0 scan=1 mayores_inicio=7  3 3 3 1 3 2 5 5
...
  menores_fin=2 scan=6 mayores_inicio=7  1 2 3 3 3 3 5 5
  menores_fin=2 scan=6 mayores_inicio=6  1 2 3 3 3 3 5 5
```

Idea clave

El pivote al azar arregla las entradas ordenadas, y la partición en tres, los empates. Ninguno sirve para el otro.

Lo que dice el reloj

`-O2`, sin sanitizers, mediana de 5. Depende de la máquina; las razones se repitieron en 3 corridas.

terminal

```
$ build/time_sorts
milisegundos con n = 16000
entrada       merge q.primero   q.azar   q.tres std::sort   stable
al azar        0.89     0.81     0.87     1.09     0.68     0.72
ordenada       0.24    70.71     0.34     0.57     0.12     0.07
10 valores     0.50     9.02     9.38     0.12     0.18     0.30
iguales        0.24    58.64    76.78     0.03     0.07     0.07

entrada al azar, milisegundos al duplicar n
       n    merge   q.azar   q.tres std::sort   stable
  250000     20.1     17.9     22.7      14.8     16.8
  500000     42.8     36.9     47.6      31.0     35.5
 1000000     89.1     77.6     99.4      66.5     75.4
```

- `q.primero` con ordenada: de 200 a 270 veces `q.azar`. `merge`: las copias no salen en el conteo.

SECCIÓN 05

Estabilidad

05

Estabilidad

Ordenamiento estable ***(stable sort)***

Dos elementos equivalentes quedan en el resultado en el mismo orden relativo que tenían al principio.

- Importa cuando se ordena por una **clave** y el elemento lleva algo más.
- Tabla ordenada por nombre, y clic en “nota”: se espera que los empates sigan por nombre.
- **Regla:** ordenar por la clave secundaria, y luego de manera estable por la principal.
- Mergesort es estable por una sola decisión: `items[right] < items[left]`. En un empate gana el de la izquierda.
- Error común: confiar en una prueba chica. En libstdc++, hasta 16 elementos se ordenan solo por inserción.

Quicksort no es estable

terminal

```
$ build/stability ../../data/terminos.tsv | head -2
antes:    2a 3b 3c 1d
partido:  1d 2a 3c 3b
```

`3b` iba delante de `3c`. Un `swap` salta por encima de otros iguales.
Con datos reales: términos por texto, y luego por frecuencia. El archivo está justo en ese orden.

terminal

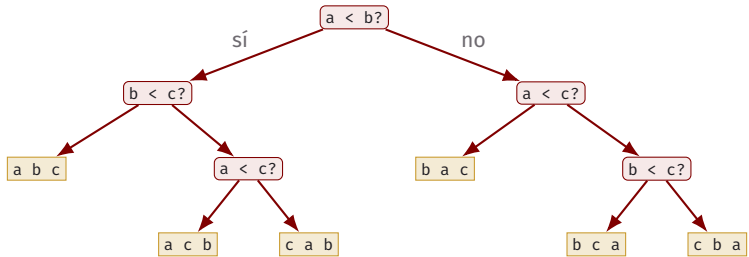
```
$ build/stability ../../data/terminos.tsv | tail -3
líneas: 28039
std::stable_sort, iguales al archivo: 28039
std::sort,        iguales al archivo: 344
```

SECCIÓN 06

La cota

06

El árbol de decisiones, con n = 3



- Cada nodo es una pregunta, y cada hoja dice “el orden correcto es este”.
- **Hacen falta al menos n! hojas:** si dos órdenes llegaran a la misma hoja, se equivocaría con uno.
- **Un árbol de altura h tiene a lo sumo 2^h hojas:** cada pregunta como mucho duplica los caminos.

De ahí sale la cota

- $2^h \geq n!$, o sea $h \geq \log_2(n!)$. La altura es el número de comparaciones en el peor caso.
- La mitad de los factores de $n!$ son al menos $n/2$, así que $n! \geq (n/2)^{n/2}$.
- $\log_2(n!) \geq (n/2)\log_2(n/2)$, que es Ω**(n log n)**.
- Con $n = 4096$: $\log_2(4096!) \approx 43\,250$. Mergesort, con datos al azar, hizo 43 983: a menos de un 2 %.

Idea clave

Hay que distinguir entre $n!$ posibilidades, y cada comparación da un solo bit de información. No es la demostración completa, pero esa es la idea.

Lo que la cota dice y lo que no dice

- **Se aplica** a quien obtiene información sobre el orden **solo comparando pares**: mergesort, quicksort, heapsort, inserción.
- **Habla del peor caso.** Con la entrada ordenada, mergesort hizo 24 576: una entrada recorre un solo camino del árbol.
- **No se aplica** a quien usa los datos de otra manera.

Ejemplo: enteros de 0 a 99

Contar cuántos hay de cada uno en un arreglo de 100 casillas cuesta $\Theta(n + 100)$, sin comparar elementos entre sí. Usa el valor como índice, lo que un comparador no puede hacer. A cambio, necesita conocer el rango.

SECCIÓN 07

Bibliotecas y comparadores

07



Lo que garantiza C++20

Función	Garantía	Requiere
`std::sort`	O(N log N) comparaciones. No garantiza estabilidad	Acceso aleatorio
`std::stable_sort`	Estable. N log N con memoria extra; si no, N $\log^2$ N	Acceso aleatorio
`std::lower_bound`	A lo sumo $\log_2$ N + O(1) comparaciones	Rango partido

- El estándar no dice qué algoritmo usar.
- `std::sort` no funciona sobre una lista enlazada. `std::list` tiene su propio `sort`, estable.
- **Regla práctica:** usen `std::sort`, y `std::stable_sort` si importan los empates.

Lo que hace libstdc++ 14

Esto no lo pide el estándar. Se puede leer en `bits/stl_algo.h`.

`std::sort`: introsort

- Quicksort con la mediana de tres candidatos como pivote.
- Si la profundidad pasa de $2 \lfloor \log_2 n \rfloor$, cambia a heapsort para ese rango.
- Deja sin ordenar los rangos de hasta 16 elementos, y al final pasa inserción.

`std::stable_sort`

Mergesort con un buffer temporal. Si no consigue memoria, usa un método en el lugar, más lento.

Comparadores

comparators.cpp

```cpp
std::sort(terms.begin(), terms.end(), by_text);
std::sort(terms.begin(), terms.end(), [](const Term& a, const Term&
      b) {
    if (a.weight != b.weight) {
        return a.weight > b.weight;
    }
    return a.text < b.text;
});
std::sort(values.begin(), values.end(), std::greater<int>());
```

- Se llama con dos elementos y devuelve `bool`.
- Como `qsort`, sin `void *`.
- Lambda con `[]`: no usa nada de alrededor.
- Varias claves en un solo comparador.

Por frecuencia, con los empates por texto: `tacna(95) arequipa(90) cusco(90) lima(40) puno(40)`.

El comparador tiene que ser un orden estricto

Orden débil estricto *(strict weak ordering)*: como <. `comp(a, a)` debe ser falso.

errors/bad_comparator.cpp

```cpp
std::vector<int> items(100, 7);
std::sort(items.begin(), items.end(), [](int a, int b) { return a <= b; });
```

terminal

```
$ build/bad_comparator
...
    #6 0xaaaaac5b9538 in main errors/bad_comparator.cpp:8
```

Es comportamiento indefinido

Esta corrida leyó fuera del arreglo. Para ordenar de mayor a menor se escribe >, no >=.

SECCIÓN 08

Búsqueda binaria

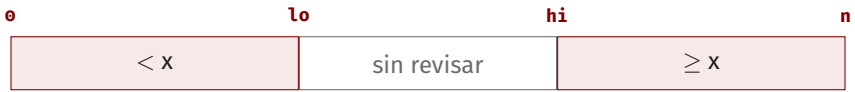
08

La pregunta que siempre tiene respuesta

Límite inferior *(lower bound)* de x

La posición del primer elemento mayor o igual a x, o n si no hay ninguno.

Esa posición siempre existe, esté x o no. Y la invariante se escribe sola:



Todo en `[0, lo)` es menor que x. Todo en `[hi, n)` es mayor o igual a x.

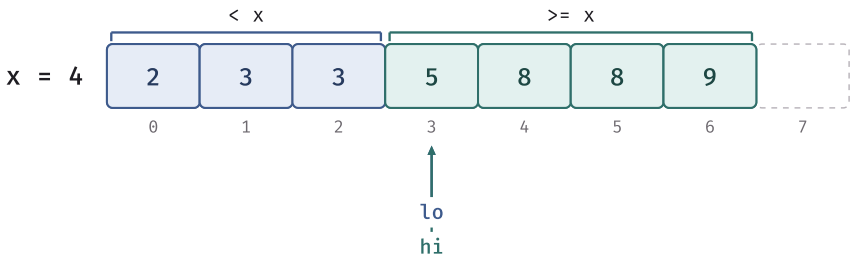
El código sale de la invariante

int_sorts.h

```
1   std::size_t lo = 0;
2   std::size_t hi = items.size();
3   while (lo < hi) {
4       std::size_t mid = lo + (hi - lo) / 2;
5       probes++;
6       if (items[mid] < x) {
7           lo = mid + 1;
8       } else {
9           hi = mid;
10      }
11  }
12  return lo;
```

- **Al inicio:** las dos zonas están vacías.
- **En cada paso:** si `items[mid] < x`, todo hasta `mid` es menor. Si no, `mid` y lo que sigue es $\geq$ x.
- **Al final:** `lo == hi`, y las zonas se tocan. `lo` es el primero $\geq$ x. Sin casos aparte.
- **Termina:** `lo <= mid < hi`, así que el rango siempre se reduce, al menos a la mitad. A lo sumo $\lfloor \log_2 n \rfloor + 1$ vueltas.

La búsqueda, paso a paso



lo = hi = 3: el 4 no está, y esa es la posición donde iría

Sigámosla, en 2 3 3 5 8 8 9

terminal

```
$ build/trace_search
x = 8
  lo=0 hi=7 mid=3 items[mid]=5 < x
  lo=4 hi=7 mid=5 items[mid]=8 >= x
  lo=4 hi=5 mid=4 items[mid]=8 >= x
  termina con lo=hi=4
x = 4
  lo=0 hi=7 mid=3 items[mid]=5 >= x
  lo=0 hi=3 mid=1 items[mid]=3 < x
  lo=2 hi=3 mid=2 items[mid]=3 < x
  termina con lo=hi=3
x = 10
...
  lo=6 hi=7 mid=6 items[mid]=9 < x
  termina con lo=hi=7
```

- **x = 8:** encontró un 8 y no se detuvo, porque podría haber otro antes. Devuelve el primero.
- **x = 4:** no está. Devuelve 3, donde iría.
- **x = 10:** devuelve 7, que es n, y es correcto.
- ¿Está x? Revisen además `lo < n` e `items[lo] == x`.

Error común: overflow en el punto medio

terminal

```
$ timeout 2 build/midpoint_overflow 2>&1 | cut -c1-110
errors/midpoint_overflow.cpp:18:23: runtime error: signed integer overflow: 1000000001 + 2000000000 cannot be
```

La suma no cabe en un `int`. Se busca sobre una función, en un rango de dos mil millones.

- `(lo + hi) / 2` con `int`: comportamiento indefinido. Aquí, además, no termina.
- `lo + (hi - lo) / 2` nunca pasa de `hi`.
- Con `std::size_t` de 64 bits no pasará en un arreglo real. Pero es igual de corta, y funciona siempre.
- C++20 trae `std::midpoint`, en `<numeric>`.

Error común: el rango que no se reduce

terminal

```
$ timeout 2 build/stuck_search; echo "código de salida: $?"
lo=0 hi=4 mid=2
lo=2 hi=4 mid=3
lo=3 hi=4 mid=3
lo=3 hi=4 mid=3
código de salida: 124
```

- Con `hi - lo == 1`, `mid` vale `lo`, y `lo = mid` no cambia nada.
- 124 es el código de `timeout`: lo tuvo que cortar.

No mezclen convenciones

`[lo, hi)` va con `hi = mid` y `lo = mid + 1`. Prueben con 0, 1 y 2 elementos.

std::lower_bound

comparators.cpp

```cpp
auto found = std::lower_bound(terms.begin(), terms.end(), wanted,
                              [](const Term& term, const std::string& x) { return term.text < x;
    });
```

Primero el elemento y después el valor. También: `std::upper_bound` y `std::equal_range`.

terminal

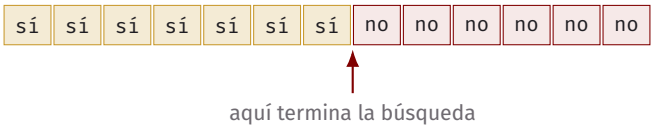
```
$ build/word_search ../../data/palabras.txt rodillas desta
1045757 palabras; ordenado por bytes: sí
rodillas   20 comparaciones, no está; la siguiente es rodillazo
desta      20 comparaciones, no está; la siguiente es destaca
```

$\log_2 1\,045\,757 \approx 20$. Límites de los datos: falta `rodillas`, y `desta` es una grafía antigua.

La idea general

Idea clave

La búsqueda binaria no necesita de verdad un arreglo ordenado por x. Le basta que el rango esté **partido** respecto de la pregunta: primero todos los “sí”, y después todos los “no”.



- En la biblioteca: `std::partition_point`.
- En el laboratorio: la segunda búsqueda no pregunta por un valor, sino por algo que se decide mirando el comienzo de cada término.

SECCIÓN 09

Bytes y UTF-8

09



`std::string` guarda bytes

terminal

```
$ build/byte_order_unsigned ../../data/terminos.tsv | head -4
a = 61, size() = 1
á = c3 a1, size() = 2
ñ = c3 b1, size() = 2
z = 7a, size() = 1
```

- `size()` cuenta bytes, no letras. Un ciclo sobre `char` parte la `á` en dos.
- `0xC3` es mayor que cualquier letra sin tilde.
- Por eso `árbol` y `ñandú` van después de `zorro`.

No es orden alfabético. Pero **los que empiezan con los mismos bytes quedan contiguos**, y para buscar eso basta.

Rangos de prefijos, sobre el archivo real

terminal

```
$ build/byte_order_unsigned ../../data/terminos.tsv | sed -n '8,14p'
a     3471 términos en [0, 3471): de a a aún
á       41 términos en [27927, 27968): de á a ávila
ca     819 términos en [4324, 5143): de ca a cañón
cá      21 términos en [7561, 7582): de cádiz a cáñamo
cañ     12 términos en [5131, 5143): de caña a cañón
z      100 términos en [27827, 27927): de z a zárate
ñ       3 términos en [28016, 28019): de ño a ñustas
```

- `a` no incluye `árbol`: `á` es otro rango, casi al final. `ca` no incluye `cámara`.
- Cada rango es contiguo: una búsqueda binaria lo puede encontrar.
- `ca` incluye `cañón`, al final, después de todo lo que sigue con `z`.
- (Las columnas se corren en las filas con tilde: `printf` cuenta bytes.)

La trampa de “hasta cazzzz”

terminal

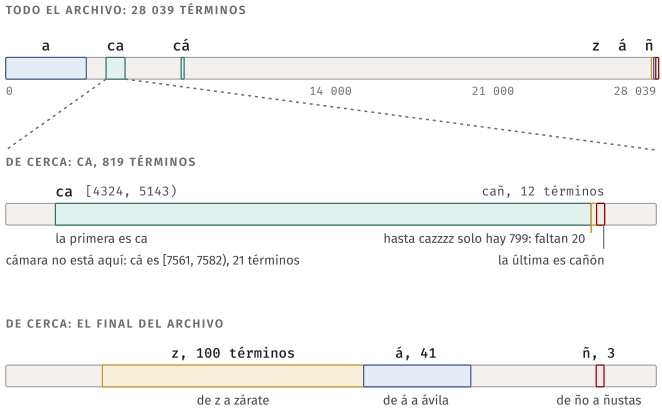
```
$ build/byte_order_unsigned ../../data/terminos.tsv | tail -1
entre ca y cazzzz: 799 términos
```

- Faltan 20 de los 819: `cañón`, `caía`… Se pierden sin que nada falle.
- “Sumarle 1 al último byte” falla en los casos extremos.
- La pregunta correcta se hace sobre el **comienzo** de cada término: `compare(0, n, prefijo)`.

Idea clave

Este autocompletado busca por bytes, y es correcto en ese sentido. Pero `ca` no ofrece `cámara`. Sepan qué hace su programa, y díganlo.

Los rangos, a escala



En orden de bytes, á empieza con 0xC3: va después de la z. Por eso el prefijo a no incluye a árbol

El signo de `char`

En Linux con g++: con signo en x86-64, y sin signo en ARM de 64 bits. El mismo programa, de las dos maneras:

terminal

```
$ build/byte_order_signed ../../data/terminos.tsv | sed -n '5,7p'
char con signo: sí
con std::string: cama cazar caña cámara zorro árbol ñandú
char por char : árbol ñandú cámara caña cama cazar zorro
$ build/byte_order_unsigned ../../data/terminos.tsv | sed -n '7p'
char por char : cama cazar caña cámara zorro árbol ñandú
```

- Si `char` tiene signo, `0xC3` vale −61: `á` queda antes que `a`.
- El estándar define `char_traits<char>::lt` como `<` sobre `unsigned char`. `operator<`, `compare` y `starts_with` no dependen del signo.
- No comparen `char` por `char`. Las pruebas corren también con `-fsigned-char`.

SECCIÓN 10

Laboratorio

10

Autocompletado

Sesión 1: `merge_sort`, `quick_sort` y `slow_input`, y medir.
Sesión 2: `sort_by_text`, `prefix_range`, `top_k` y `top_k_scan`, y medir.

- Las pruebas **cuentan**, no miden tiempos: comparaciones, copias y lecturas. Usen índices.
- `merge_sort` es estable y no copia.
- `prefix_range` se compara con un oráculo, con prefijos cortados por bytes.
- Los empates de `top_k` van por texto. ¿Qué propiedad les da eso gratis?
- `lab/report.md`: ¿con qué prefijos gana de verdad la búsqueda binaria? No lo den por hecho. Midan.

terminal

```
make test-sorts
make test-autocomplete
make test-lab
make bench-sorts
make bench-query
```

Para la próxima clase

- Leer los apuntes de la semana 7: diccionarios y tablas hash.
- Laboratorio: autocompletado.
- Escriban la invariante de `prefix_range` antes de escribir el código.

Pregunta

Ordenar nos dejó buscar en $\Theta(\log n)$, y además hallar todo lo que empieza con un prefijo. Si solo les importara saber si una palabra exacta está o no, ¿pagarían por tener todo ordenado?