


UNIVERSIDAD NACIONAL DE INGENIERÍA

Análisis y Diseño de Algoritmos

# Demostrar un algoritmo

## Invariantes, inducción y Gale-Shapley

**Gerson Garrido**

Semana 1

# Hoy

**Sesión A**

1. Cómo funciona el curso
2. Por qué las pruebas no alcanzan
3. Invariante de ciclo: búsqueda binaria
4. Inducción: potencia rápida
5. Terminación

**Sesión B**

6. El matching estable
7. Gale-Shapley y su demostración
8. Quién propone importa
9. Tabla de rangos: $O(n^2)$ contra $O(n^3)$
10. Cupos y usos reales

**Lectura**

Apuntes de la semana 1. Código en `code/week-01`. Apoyo: Erickson, secs. 0.5, 0.6 y 4.5; CLRS, capítulos *Getting Started* y *Matchings in Bipartite Graphs*.

SECCIÓN 01

# Cómo funciona el curso

01

# Qué cambia respecto de Estructuras de Datos

**Allá**

- Usaron Dijkstra para encontrar rutas.
- Usaron Huffman para comprimir.
- Los probaron, y funcionaron.

**Aquí**

- ¿Cómo se diseña un algoritmo para un problema nuevo?
- ¿Cómo se demuestra que siempre da la respuesta correcta?
- ¿Cómo se reconoce un problema para el que quizá no exista ningún algoritmo rápido?

**Hoy**

Empezamos por la segunda pregunta, porque las otras dos la necesitan.

# Cómo se trabaja

- **Teoría**, 2 + 2 h: se diseña y se demuestra.
- **Laboratorio**, 2 h: un problema escrito en palabras, en C++, hasta que pase el **juez**.
- Los laboratorios son para practicar: el juez responde al instante.
- Los apuntes son el texto del curso.

**Prácticas y exámenes**

- 4 prácticas: semanas 4, 7, 11 y 14. Una demostración y un programa.
- Parcial en la semana 8 y final en la 16.

SECCIÓN 02

# Por qué las pruebas no alcanzan

02

# Problema, instancia y algoritmo

- **Problema:** qué entradas son válidas y qué salida le toca a cada una.
- **Instancia:** una entrada concreta. El arreglo 5, 2, 9 es una instancia de ordenar.
- **Algoritmo:** procedimiento paso a paso para cualquier instancia.

**Correctitud** *(correctness)*

Un algoritmo es correcto si, para toda instancia válida, **termina** y devuelve una **salida que el problema acepta**. Son dos cosas, y las dos se demuestran.

# 1000 pruebas de 1000

tests_vs_proof.cpp

```cpp
int best = 0;
for (int value : a) {
    if (value > best) {
        best = value;
    }
}
return best;
```

Un máximo que empieza en 0. Se prueba con 1000 arreglos al azar, de valores entre 1 y 1 000 000, contra `std::max_element`, y con un arreglo más.

terminal

```
$ build/tests_vs_proof
pruebas al azar con valores positivos: 1000 de 1000 correctas
con {-5, -2, -9}: devuelve 0, y el máximo es -2
```

# Lo que dice ese resultado

- Ninguna de las mil pruebas podía encontrar el error: todas sacaban valores del mismo lugar.
- Un millón tampoco. El problema no es la cantidad.

**Una prueba puede mostrar un error, pero no su ausencia**

Un contraejemplo *(counterexample)* refuta un algoritmo. Ninguna cantidad de ejemplos lo demuestra.

**Hacemos las dos cosas**

La demostración dice que el algoritmo está bien. Las pruebas dicen que lo escribimos bien.

SECCIÓN 03

# Invariante de ciclo

03



# Qué se demuestra de un ciclo

**Invariante de ciclo** ***(loop invariant)***

Condición sobre las variables que se cumple cada vez que el ciclo va a evaluar su condición.

1. **Inicio.** Se cumple antes de la primera vuelta.
2. **Conservación.** Si se cumple antes de una vuelta y la condición es verdadera, se cumple después.
3. **Salida.** Si se cumple y la condición es falsa, el resultado es correcto.

**Es inducción sobre las vueltas**

Inicio es el caso base y conservación, el paso inductivo. Se agrega la salida.

# Un primer ejemplo: la suma

pseudocódigo

```
algoritmo suma(a, n)
    s ← 0
    i ← 0
    mientras i < n hacer
        s ← s + a[i]
        i ← i + 1
    fin mientras
    devolver s
fin algoritmo
```

**Invariante.** $0 \leq i \leq n$ y $s = a[0] + \cdots + a[i-1]$.

- **Inicio.** Con $i = 0$ la suma no tiene términos y vale 0. Y $s = 0$.
- **Conservación.** Con $i < n$, el cuerpo suma $a[i]$ y aumenta $i$. Queda $s = a[0] + \cdots + a[i_{\text{nuevo}} - 1]$, y $i_{\text{nuevo}} \leq n$.
- **Salida.** Termina cuando $i \geq n$. Con $i \leq n$, eso obliga a $i = n$, y entonces $s = a[0] + \cdots + a[n-1]$.
- **Terminación.** $n - i$ es un entero, no es negativo y baja en 1 en cada vuelta.

SECCIÓN 04

# Búsqueda binaria

04

# La búsqueda binaria, con precisión

pseudocódigo

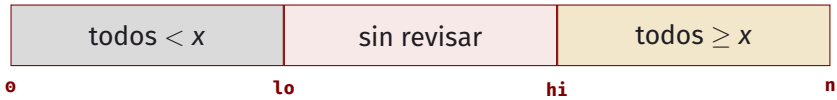
```
algoritmo primero_mayor_o_igual(a, n, x)
    lo ← 0
    hi ← n
    mientras lo < hi hacer
        mid ← lo + (hi - lo) div 2
        si a[mid] < x entonces
            lo ← mid + 1
        sino
            hi ← mid
        fin si
    fin mientras
    devolver lo
fin algoritmo
```

**Problema**

Entrada: a ordenado de menor a mayor, $n \geq 0$, y x.
Salida: el primer i con $a[i] \geq x$, o n si no hay.

- Es `std::lower_bound`.
- ¿Está x? Si el resultado r cumple $r < n$ y $a[r] = x$.
- **Rango semiabierto** [lo, hi): tiene hi − lo elementos, y el rango vacío es lo = hi. Como `end()` en C++.

# La invariante: tres zonas




**Invariante**

**(I1)** $0 \leq lo \leq hi \leq n$.
**(I2)** Para todo i con $0 \leq i < lo$: $a[i] < x$.
**(I3)** Para todo i con $hi \leq i < n$: $a[i] \geq x$.

**Inicio.** Con lo = 0 y hi = n vale (I1). En (I2) e (I3) no hay ningún i: se cumplen porque no hay nada que revisar.

# Conservación: dos casos

Con lo < hi, sea $d = (hi - lo)$ div 2. Entonces $0 \leq d < hi - lo$, así que $lo \leq mid < hi$, y $a[mid]$ existe.

**Caso** $a[mid] < x$

Como a está ordenado, para todo $i \leq mid$: $a[i] \leq a[mid] < x$.

(I2) vale con lo $\leftarrow$ mid + 1. (I3) no cambia.

(I1): mid < hi implica mid + 1 $\leq$ hi.

**Caso** $a[mid] \geq x$

Como a está ordenado, para todo $i \geq mid$: $a[i] \geq a[mid] \geq x$.

(I3) vale con hi $\leftarrow$ mid. (I2) no cambia.

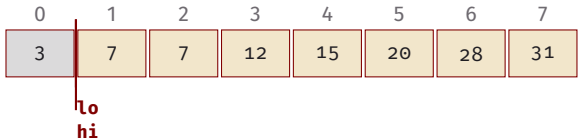
(I1): lo $\leq$ mid.

**Dónde se usa el orden:** en los dos casos. Sin él, mirar un elemento no dice nada de los demás.

# Salida y terminación

- **Salida.** El ciclo termina cuando lo $\geq$ hi. Con (I1), lo = hi. Llamemos r a ese valor.
- (I2): $a[i] < x$ para todo $i < r$. (I3): $a[i] \geq x$ para todo $i \geq r$.
- Si $r < n$, r es el primer índice con $a[i] \geq x$. Si $r = n$, no hay ninguno. Es justo lo que pide el problema.
- **Terminación.** hi $-$ lo es un entero, no negativo por (I1).
  - Primer caso: pasa a hi $-$ mid $-$ 1, menor porque mid $\geq$ lo.
  - Segundo caso: pasa a mid $-$ lo, menor porque mid < hi.
- **Costo.** De s elementos quedan a lo sumo s/2. Para $n \geq 1$: a lo sumo $\lfloor \log n \rfloor + 1$ vueltas.

# Vuelta por vuelta: x = 7




**Salida.** lo = hi = 1: el primer 7. Con x = 16 sale 5, con x = 40 sale 8 = n, con x = 1 sale 0. La demostración cubre todo, sin casos aparte.

> **Compruébenlo**
>
> `build/trace_search` imprime estas vueltas y comprueba la invariante entera con `assert`.

# Una invariante que no alcanza

Cambiemos `hi = mid` por `hi = lo`. Se conserva «$0 \le lo \le hi \le n$ y $a[i] < x$ para $i < lo$», y el programa termina.

terminal

```
$ build/bad_searches
buscar 10 en {2, 4, 6, 8, 10}
  correcta: 4
  invariante débil: 3
buscar 9 en {2, 4}
  correcta: 2
  con lo = mid: -1 (cortada a las 1000 vueltas)
```

Y da mal: $a[3] = 8$ no es mayor o igual que 10. **Error común:** una invariante verdadera que no alcanza. Escriban qué dice cuando la condición es falsa, y comprueben que es el resultado.

# Cuando la cantidad no baja: `lo = mid`

Buscar 9 en {2, 4}, escribiendo `lo = mid` en vez de `lo = mid + 1`:

| Vuelta | lo | hi | mid | a[mid] | Qué pasa |
|---|---|---|---|---|---|
| 1 | 0 | 2 | 1 | 4 | 4 < 9, y lo $\leftarrow$ 1 |
| 2 | 1 | 2 | 1 | 4 | 4 < 9, y lo $\leftarrow$ 1: nada cambió |
| 3 | 1 | 2 | 1 | 4 | igual, para siempre |

**La invariante sigue siendo cierta**

Lo que falla es la terminación: con hi $-$ lo = 1, mid = lo, y hi $-$ lo no baja. En el juez sería un TLE.

SECCIÓN 05

# Inducción y terminación

05

# Potencia rápida

pseudocódigo

```
algoritmo potencia(b, e, m)
    si e = 0 entonces
        devolver 1 mod m
    fin si
    mitad ← potencia(b, e div 2, m)
    resultado ← (mitad * mitad) mod m
    si e mod 2 = 1 entonces
        resultado ← (resultado * b) mod m
    fin si
    devolver resultado
fin algoritmo
```

**Afirmación.** Para $e \geq 0$, termina y devuelve $b^e$ mód $m$. **Técnica:** inducción fuerte sobre e.

- **Base, e = 0.** Devuelve 1 mód $m = b^0$ mód $m$. El caso m = 1 da 0.
- **Paso,** $e \geq 1$. Sea $k = e$ div 2. Como $k < e$, la hipótesis da mitad $= b^k$ mód $m$.
- Entonces mitad$^2 \equiv b^{2k}$ (mód $m$).
- Si $e = 2k$, ya está. Si $e = 2k + 1$, multiplicar por b da $b^{2k+1}$.

**La terminación viene incluida**

La inducción funciona porque $k < e$: la cantidad que baja.

# Cuántos productos

terminal

```
$ build/trace_power
e  3^e mod 1000  productos
0  1  0
1  3  2
3  27  4
6  729  5
13  323  7
2^1000000000 mod 1000000007 = 140625001 con 43 productos
```

$\lfloor \log e \rfloor + 1$ llamadas con $e \geq 1$, cada una con un cuadrado, y un producto más por cada bit en 1 de e. En total, a lo sumo $2(\lfloor \log e \rfloor + 1)$.

# Dos cuidados

**En C++: overflow**

Un resto puede llegar a $m - 1$. La función pide $m \leq 2^{31}$: un producto es menor que $2^{62}$ y cabe en `long long`, no en `int`. Hagan siempre esa cuenta.

**Error común**

Llamar dos veces a `potencia(b, e div 2, m)`. Sigue siendo correcto, pero el número de llamadas se duplica en cada nivel y el costo vuelve a ser proporcional a e.

Correcto y eficiente son dos demostraciones distintas.

# Terminación: la regla general

**Regla**

Una cantidad entera que nunca sea negativa y que baje estrictamente en cada paso. Si empieza en N, hay a lo sumo N pasos.

| Algoritmo | Cantidad | Por qué baja | Pasos |
|---|---|---|---|
| `suma` | n − i | i aumenta en 1 | n |
| Búsqueda binaria | hi − lo | se reduce al menos a la mitad | O(log n) |
| Potencia rápida | e | e div 2 < e, si $e \geq 1$ | O(log e) |
| Gale-Shapley | ? | lo vemos hoy | |

**Una demostración aquí:** qué se afirma, con qué técnica, pasos cortos en prosa cubriendo todos los casos, y dónde se usa cada hipótesis.

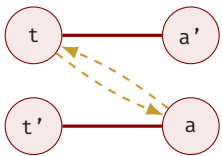
SECCIÓN 06

# Matching estable

06

# El problema

n tesistas y n asesores. Cada uno ordena a **todos** los del otro grupo, sin empates. Hay que hacer parejas.




**Par inestable**

t y a no son pareja, t prefiere a a antes que a su pareja, y a prefiere a t antes que a su pareja.

**Matching estable** ***(stable matching)***

Un matching perfecto sin pares inestables.

No exige que todos estén contentos: exige que nadie tenga con quién irse. ¿Siempre existe uno?

# El ejemplo de la semana

| Tesista | Lista | Asesor | Lista |
|---|---|---|---|
| t0 | a1, a0, a2, a3 | a0 | t0, t3, t2, t1 |
| t1 | a1, a2, a0, a3 | a1 | t1, t0, t2, t3 |
| t2 | a0, a1, a3, a2 | a2 | t0, t1, t2, t3 |
| t3 | a0, a3, a2, a1 | a3 | t3, t2, t0, t1 |

**De más a menos preferido**

En el código son enteros de 0 a $n - 1$ y un `std::vector<std::vector<int>` para cada lado.

# Idea 1 que falla: la primera oferta manda

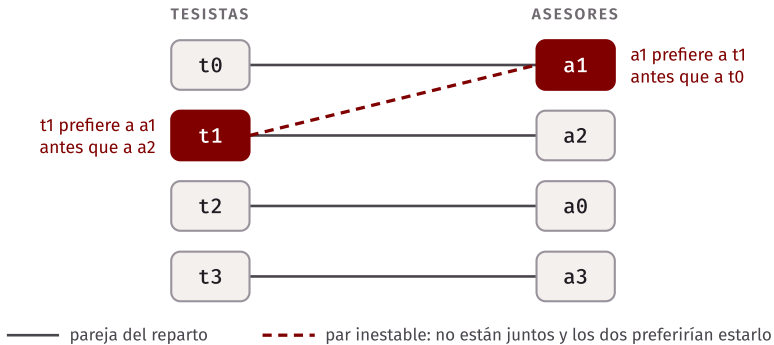
Cada tesista, en orden, se queda con el asesor libre que más le gusta. El asesor acepta a quien llegue primero, y no cambia nunca.

terminal

```
$ build/first_offer
primera oferta: t0-a1 t1-a2 t2-a0 t3-a3
par inestable: t1 y a1
```

- t1 quería a a1 primero, pero t0 llegó antes.
- a1 también prefiere a t1 antes que a t0. Aceptó sin saber qué ofertas vendrían después.

# El par inestable de la primera oferta



# Idea 2 que falla: arreglar lo que esté mal

terminal

```
$ build/repair_loop
a0 a1 a2 a3 par inestable: t1 y a3
a0 a3 a2 a1 par inestable: t2 y a0
a2 a3 a0 a1 par inestable: t3 y a0
a2 a3 a1 a0 par inestable: t0 y a1
a1 a3 a2 a0 par inestable: t2 y a3
a1 a2 a3 a0 par inestable: t3 y a2
a1 a0 a3 a2 par inestable: t1 y a1
a0 a1 a3 a2 par inestable: t2 y a0
a3 a1 a0 a2 par inestable: t0 y a2
a2 a1 a0 a3 par inestable: t1 y a3
volvimos a: a2 a3 a0 a1
matchings estables de esta instancia: 3
```

La idea correcta: **un lado solo avanza, y el otro solo mejora.**

- Tomar el primer par inestable y juntarlo. Las ex parejas quedan juntas.
- Cada línea: el asesor de t0, t1, t2 y t3.
- A los diez arreglos, vuelve a la tercera línea. La regla es determinista: no termina.
- Y hay tres matchings estables. Es un contraejemplo para esta regla.

SECCIÓN 07

# Gale-Shapley

07

# Posponer el sí definitivo

pseudocódigo

```
algoritmo gale_shapley(pref_t, pref_a, n)
    siguiente[t] ← 0 y pareja[a] ← -1 para todo t y todo a
    libres ← todos los tesistas
    mientras libres no esté vacío hacer
        t ← sacar un tesista cualquiera de libres
        a ← pref_t[t][siguiente[t]]
        siguiente[t] ← siguiente[t] + 1
        si pareja[a] = -1 entonces
            pareja[a] ← t
        sino si a prefiere a t sobre pareja[a] entonces
            agregar pareja[a] a libres
            pareja[a] ← t
        sino
            agregar t a libres
        fin si
    fin mientras
    devolver pareja
fin algoritmo
```

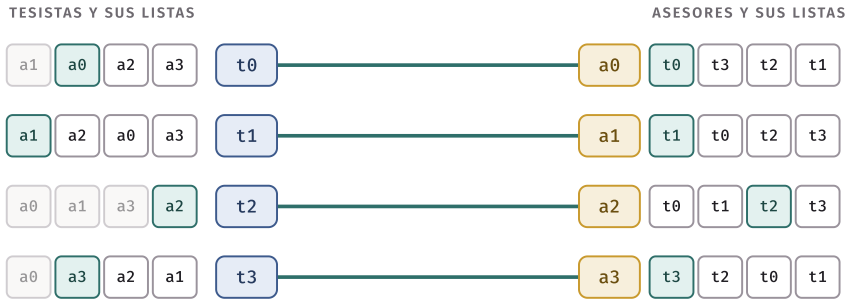
- Un tesista libre le propone al mejor al que no le ha propuesto.
- Un asesor libre acepta, por ahora.
- Un asesor ocupado se queda con el que prefiere de los dos.
- «Cualquiera»: la demostración no usa el orden.

# Vuelta por vuelta

| Paso | Propone | A | a0 | a1 | a2 | a3 | Qué pasa | Libres al final |
|---|---|---|---|---|---|---|---|---|
| 1 | t0 | a1 | | **t0** | | | a1 estaba libre: acepta | t1, t2, t3 |
| 2 | t1 | a1 | | **t1** | | | a1 cambia a t0 por t1 | t0, t2, t3 |
| 3 | t0 | a0 | **t0** | t1 | | | a0 estaba libre: acepta | t2, t3 |
| 4 | t2 | a0 | t0 | t1 | | | a0 se queda con t0 | t2, t3 |
| 5 | t2 | a1 | t0 | t1 | | | a1 se queda con t1 | t2, t3 |
| 6 | t2 | a3 | t0 | t1 | | **t2** | a3 estaba libre: acepta | t3 |
| 7 | t3 | a0 | t0 | t1 | | t2 | a0 se queda con t0 | t3 |
| 8 | t3 | a3 | t0 | t1 | | **t3** | a3 cambia a t2 por t3 | t2 |
| 9 | t2 | a2 | t0 | t1 | **t2** | t3 | a2 estaba libre: acepta | ninguno |

Resultado: t0-a0, t1-a1, t2-a2, t3-a3, con 9 propuestas y ningún par inestable. Cada columna de un asesor solo mejora. Cada tesista solo baja: t2 terminó con el último de su lista.

# Las propuestas, en el dibujo




resultado: t0-a0 t1-a1 t2-a2 t3-a3

propuestas: 9. Pares inestables: ninguno.

Lo apagado en la lista de un tesista solo crece. La pareja marcada en la lista de un asesor solo sube.

SECCIÓN 08

# Por qué funciona

08

# Dos observaciones que sostienen todo

**Lema 1: los asesores solo mejoran**

Una vez que un asesor recibe su primera propuesta, tiene pareja hasta el final, y cada cambio de pareja es a alguien que prefiere.

*Demostración.* `pareja[a]` solo pasa de $-1$ a t, o cambia en la rama donde a prefiere a t. Ninguna línea la pone en $-1$.

**Lema 2: los tesistas solo bajan**

Cada tesista propone en el orden de su lista, a cada asesor como máximo una vez.

*Demostración.* `siguiente[t]` aumenta con cada propuesta y nunca baja, y la lista no repite.

Además, `pareja` es siempre un matching, y todo tesista está libre o es la pareja de exactamente un asesor.

# Un tesista libre siempre tiene a quién proponer

**Lema 3**

Si t está libre, `siguiente[t]` < n: todavía no le ha propuesto a todos.

**Por contradicción.** Supongamos que t está libre y ya les propuso a los n asesores.

1. Por el Lema 1, los n asesores tienen pareja.
2. Como `pareja` es un matching, son n tesistas distintos.
3. Ninguno es t, porque t está libre.
4. Pero solo hay $n - 1$ tesistas aparte de t. Contradicción.

**Dónde se usan las hipótesis:** grupos del mismo tamaño y listas completas. Sin ellas, puede quedarse sin pareja.

# Termina y da un matching perfecto

**Teorema 1**

Gale-Shapley termina después de a lo sumo $n^2$ propuestas.

- Cantidad: $\Phi = \sum_t (n - \texttt{siguiente}[t])$, las propuestas que aún pueden hacerse. Empieza en $n^2$.
- Por el Lema 3, nunca es negativa. Cada vuelta hace una propuesta y baja $\Phi$ en exactamente 1.
- No es el número de libres: ese no baja en el paso 2 del ejemplo.

**Teorema 2**

Al terminar, `pareja` es un matching perfecto.

*Demostración.* `libres` está vacío, así que todos los tesistas tienen pareja. Son n tesistas con n asesores distintos.

# Es estable

**Teorema 3**

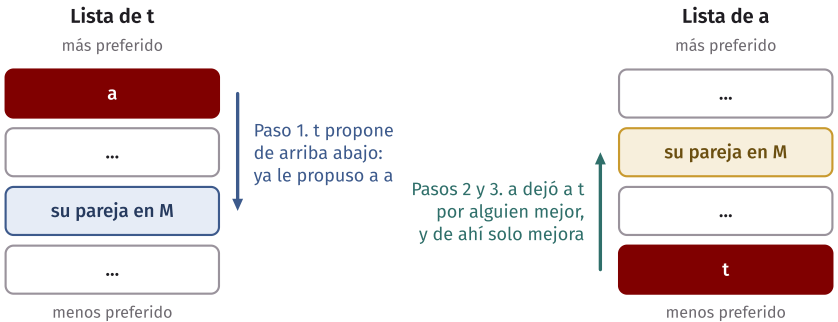
El matching M que devuelve Gale-Shapley no tiene pares inestables. **Técnica:** contradicción.

Supongamos que t y a no están juntos en M, t prefiere a a antes que a su pareja, y a prefiere a t antes que a su pareja.

1. **t le propuso a a.** Por el Lema 2, propone en orden, y su pareja está después de a en su lista.
2. **a dejó a t.** Hay dos maneras, y las dos dicen lo mismo:
   - lo rechazó al proponer, porque tenía a alguien que prefería;
   - lo aceptó, y luego lo cambió por alguien que prefería.
3. **De ahí en adelante, a solo mejora.** Por el Lema 1, a prefiere a su pareja final antes que a t.
4. Eso contradice que a prefiere a t antes que a su pareja. No hay par inestable.

Por lo tanto, **siempre existe un matching estable**. En la Idea 1, sin cambiar de pareja, el paso 3 es falso.

# La demostración en dos listas




La suposición decía que a prefiere a t antes que a su pareja en M. Las listas dicen lo contrario.

Contradicción: no hay par inestable. ■

SECCIÓN 09

# Quién propone importa

09

# Dos matchings estables

| | |
|---|---|
| t0 | a0, a1 |
| t1 | a1, a0 |
| a0 | t1, t0 |
| a1 | t0, t1 |

terminal

```
$ build/who_proposes
instancia cruzada
  estable: t0-a0 t1-a1
  estable: t0-a1 t1-a0
  proponen los tesistas: t0-a0 t1-a1
  proponen los asesores: t0-a1 t1-a0
ejemplo de cuatro
  estable: t0-a0 t1-a1 t2-a2 t3-a3
  proponen los tesistas: t0-a0 t1-a1 t2-a2 t3-a3
  proponen los asesores: t0-a0 t1-a1 t2-a2 t3-a3
```

a es **viable** para t si existe algún matching estable con t y a juntos. El **mejor viable** de t es el viable que t más prefiere.

# Cada tesista queda con su mejor viable

**Teorema 4**

Gale-Shapley le da a cada tesista su mejor viable. El resultado no depende del orden.

**Técnica:** mirar el primer rechazo de un tesista por un asesor viable: a rechaza a t por t', y a es viable para t.

1. Hay un matching estable S con t-a. Sea a' la pareja de t' en S. Como t' $\neq$ t, a' $\neq$ a.
2. Si t' prefiriera a a' antes que a a, ya le habría propuesto, y a' lo habría rechazado. a' es viable para t': sería un rechazo anterior. Así que t' prefiere a a antes que a a'.
3. a prefiere a t' antes que a t: por eso rechazó a t.
4. Entonces (t', a) es un par inestable de S. Contradicción.
5. Luego nadie es rechazado por un viable. Todos los que t prefiere antes que a su pareja lo rechazaron, así que no son viables, y su pareja sí lo es (el resultado es estable).

La definición de mejor viable solo habla de las listas: el orden no importa, ni para el resultado ni para las propuestas.

# El otro lado paga la cuenta

**Teorema 5**
Cada asesor queda con el peor tesista para el cual es viable.

**Por contradicción.** Sea (t, a) del resultado, y S estable donde a está con un t' que le gusta menos que t.

1. Sea a' la pareja de t en S. Como en S a está con t' $\neq$ t, a' $\neq$ a.
2. Por el Teorema 4, a es el mejor viable de t, así que t prefiere a a antes que a a'.
3. a prefiere a t antes que a t'.
4. (t, a) es un par inestable de S. Contradicción.

**Elegir quién propone es una decisión de diseño:** lo mejor para todos los que proponen, y lo peor para todos los que reciben. `test_examples.cpp` lo comprueba enumerando hasta n = 6.

SECCIÓN 10

# Tabla de rangos

10

# Solo una línea no es O(1)

- Sacar un libre: una pila. O(1).
- Siguiente de la lista. O(1).
- Saber quién tiene a a. O(1).
- a prefiere a t antes que a pareja[a].

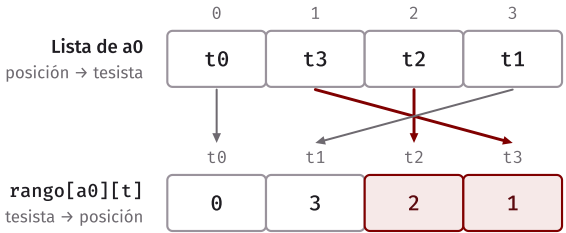
Recorrer la lista cuesta O(n). Con $n^2$ propuestas: $O(n^3)$.

**Tabla de rangos**

rango[a][t] es la posición de t en la lista de a. Se llena una vez, en $O(n^2)$. Comparar es mirar dos números: O(1).

$O(n^2)$ es lo mejor posible en este formato, porque la entrada tiene $2n^2$ números.

# De la lista a la tabla




**¿Prefiere a0 a t3 antes que a t2? rango[a0][t3] = 1 < 2 = rango[a0][t2]: sí, sin recorrer la lista.**

Llenar la tabla cuesta $O(n^2)$ una vez; cada comparación, $O(1)$.

# En C++

matching.h

```
for (int r = 0; r < n; r++) {
    for (int position = 0; position < n;
     position++) {
        rank[r][prefs[r][position]] =
     position;
    }
}
```

matching.h

```
int held = holder[r];
if (held == -1) {
    holder[r] = p;
} else if (prefers(r, p, held)) {
    holder[r] = p;
    free_proposers.push_back(held);
} else {
    free_proposers.push_back(p);
}
```

Con tabla, `prefers` es `rank[r][a] < rank[r][b]`. Con recorrido solo cambia `prefers`, y da lo mismo.

# Contar: la instancia importa

terminal

```
$ build/rank_counts
instancia      n  propuestas  casillas
adversa      250       31375       5177125
adversa      500      125250      41541750
adversa     1000      500500     332833500
adversa     2000     2001000    2664667000
al azar      250        1392         64948
al azar      500        3523        310004
al azar     1000        9150       1391767
al azar     2000       15845       5086685
```

- **Adversa:** todos con la misma lista, y gana el último en llegar. $n(n+1)/2$ propuestas, unas $n^3/3$ casillas: ×8 al duplicar n.
- **Al azar:** con $n = 2000$, 16 mil propuestas de 4 millones posibles.
- Un programa con recorrido, probado al azar, parece estar bien.

**Error común:** comparar recorriendo una lista. Da las mismas respuestas y es $O(n^3)$.

# Medir: el reloj

terminal, depende de la máquina

| n | tabla (s) | razón | ns/propuesta | lista (s) | razón | ns/casilla | lista/tabla |
|---|---|---|---|---|---|---|---|
| 500 | 0.0031 | - | 25.0 | 0.0283 | - | 0.68 | 9 |
| 1000 | 0.0152 | 4.9 | 30.4 | 0.2336 | 8.3 | 0.70 | 15 |
| 2000 | 0.1014 | 6.7 | 50.7 | 1.2885 | 5.5 | 0.48 | 13 |
| 4000 | 0.4745 | 4.7 | 59.3 | 11.3885 | 8.8 | 0.53 | 24 |

Familia adversa, `-O2`, sin sanitizers, mediana de 5 o 3 corridas, en el contenedor del curso. Depende de la máquina y de la carga.

- De 500 a 4000, n se multiplica por 8. **Lista:** tiempo ×400, en el orden de $8^3 = 512$.
- **Tabla:** ×150, y no ×64, aunque las propuestas sí se multiplican casi por 64: crece `ns/propuesta`.
- Causa probable: cada propuesta salta de fila en una tabla de unos 64 MB. Esto no lo comprueba: el reloj no ve fallos de cache.
- $O(n^2)$ cuenta operaciones, y no dice cuánto cuesta cada una. La tabla cambia memoria por tiempo.

SECCIÓN 11

# Cupos y usos reales

11

# Variante con cupos

pseudocódigo

```
algoritmo gale_shapley_con_cupos(pref_p, pref_c, v, A, C)
    siguiente[p] ← 0 y admitidos[c] ← vacío para todo p y toda c
    libres ← todos los postulantes
    mientras libres no esté vacío hacer
        p ← sacar un postulante cualquiera de libres
        si siguiente[p] < C entonces
            c ← pref_p[p][siguiente[p]]
            siguiente[p] ← siguiente[p] + 1
            agregar p a admitidos[c]
            si c tiene más de v[c] admitidos entonces
                q ← el admitido de c que c prefiere menos
                sacar q de admitidos[c]
                agregar q a libres
            fin si
        fin si
    fin mientras
    devolver admitidos
fin algoritmo
```

- A postulantes y C carreras con vacantes. Estar sin carrera es lo peor.
- **Par inestable:** p preferiría c, y c tiene una vacante libre o prefiere a p antes que a un admitido.
- Ya no es un matching perfecto.
- «¿Quién es el peor admitido?» se pregunta muchas veces: una cola de prioridad por carrera.

# Es estable, con el mismo argumento

**Teorema 6**

Termina después de a lo sumo $A \cdot C$ propuestas, y su resultado es una asignación estable.

- **Las carreras solo mejoran.** Desde que una carrera se llena, sigue llena y su peor admitido solo mejora. Si saca a alguien, está llena y le gustan más todos sus admitidos.
- **Por contradicción.** Si p y c forman un par inestable, p ya le propuso a c: propone en orden, y llegó más abajo o agotó la lista.
- Como p no está en c, c lo sacó. En ese momento estaba llena, y todos sus admitidos le gustaban más que p.
- Al final sigue llena, y todos le gustan más que p. Contradicción.

Lo mismo con el Teorema 4: cada postulante recibe la mejor carrera posible en alguna asignación estable. Queda como ejercicio.

# Dónde se usa

- **Médicos residentes.** En Estados Unidos, el National Resident Matching Program usa un algoritmo de esta familia, y proponen los postulantes. Tiene complicaciones reales, como parejas de médicos: con ellas puede no existir un matching estable, y el sistema no es Gale-Shapley sin cambios.
- **Escuelas.** Sistemas escolares públicos, como los de Nueva York y Boston, rediseñaron su asignación con aceptación diferida *(deferred acceptance)*.
- Gale y Shapley publicaron el algoritmo en 1962. Alvin Roth y Lloyd Shapley recibieron el Nobel de Economía de 2012 por el trabajo en esta línea.

**Una propiedad demostrada**

Cuando la decisión afecta a miles de personas, poder decir por qué nadie tiene un reclamo vale más que correr en un segundo.

SECCIÓN 12

# Laboratorio

12

# El juez: `lab/warmup`

terminal, depende de la máquina

```
01          AC     0.00 s
02          WA     0.00 s  palabra 2: esperaba 1, salió 0
03          WA     0.00 s  palabra 2: esperaba 5, salió 4
04          AC     0.00 s
05          WA     0.00 s  palabra 1: esperaba 3, salió 2
06          WA     0.02 s  palabra 32401: esperaba 200000, salió 199999
2/6 casos correctos
```

- `env/check.sh`: todo `ok`. Después, `make test` con un programa dado que tiene un bug.
- El caso 01 es el ejemplo del enunciado, y pasa. Lean el veredicto, corran el caso a mano y corríjanlo.

`AC` correcto `WA` incorrecto `TLE` se pasó del tiempo `RE` terminó con error

# El problema: `lab/admissions`

1. Postulantes que ordenan carreras, y carreras con vacantes que ordenan postulantes.
2. Calcular la asignación estable que más favorece a los postulantes. Empiecen por el pseudocódigo.
3. Pasar todos los casos, incluidos los grandes. Sus entradas tienen millones de números.
4. Responder: ¿quién gana y quién pierde cuando proponen las carreras? Con una demostración o con una ejecución.

**Si da TLE**

Calculen cuántas operaciones hace su programa con ese tamaño. Recuerden que hay dos preguntas que se repiten en cada rechazo.

# Para la próxima clase

- Leer los apuntes de la semana 1 y hacer los ejercicios.
- Terminar `lab/warmup` y `lab/admissions`.
- Próxima semana: recurrencias, teorema maestro y mergesort demostrado.

**Preguntas**

1. ¿Por qué Gale-Shapley hace a lo sumo $n^2$ propuestas?
2. Demuestren, sin mirar, que al terminar no queda ningún par inestable.
3. ¿Qué costo tiene una búsqueda binaria recursiva que pasa el `std::vector` por valor? Piénsenlo para la próxima semana.