

La pequeña ovejería

- Don Juan tiene 600 ovejas
- Cada oveja tiene asignado un número único
- El rebaño está dividido en 4 lotes de 150 ovejas c/u
- Para cada lote, sus números están registrados en una hoja
- Los números de un lote están en cualquier orden

TAIWAN

4160 T
4022 T
3117 m/T
3058 T
5193 S
2103 R
288 B
5/n T
3150 T
4254 m
5008 m/S
246 m

flaca
super flaca

3195 T
2199 M
6175 T
2181 T
293 T
274 T
5088 M
2063 T
4120 A
2021 T

3094 T
2055 m
229 T
3003 T
6004 T
*2140 m
263 m
*2209 m T
296 m
3091 m
2098 m T
2125 B
4195 T

269 m/n
625 T
251 R
6025 S
3108 T
5/n M
3197 M
3300 T
4163 R
6155 T

6106 m
6191 T
2006 m
267 m/n
2084 m/T
3031 T
2039 T
6063 m/n
5/n m/T
248 m
283 m
3137 m
2078 T

4058 M
2167 T
3032 T
6181 m/n
253 M
4114 T
6169 PARDA
6121 PARDA
4021 T
5209 PARDA

252 m
4083 m
2009 m
3014 T
2036 T
2241 m
6247 T
5/n T
2117 m
4177 m/T
268 m

6222 T
2109 m
3180 m
6197 parda
237 B
4206 T
6055 p
5/n p
5154 p/n
3029 m T

La pequeña ovejería



Don Juan suele encontrar algunas ovejas separadas de sus lotes

¿Cómo puede saber fácilmente a qué lote pertenece una oveja?

Secuencias ordenadas



Una secuencia de números x_1, \dots, x_n se dice **ordenada** (no decrecientemente) si cumple que $x_1 \leq \dots \leq x_n$

¿Qué es entonces **ordenar** una secuencia de números?

El algoritmo de ordenación de Don Juan

1. En la hoja original, encontrar el número más pequeño no tachado
2. Tacharlo
3. Escribirlo al final (en el primer espacio disponible) de la hoja nueva
4. Si quedan números sin tachar en la hoja original, volver al paso 1.

¿Es correcto el algoritmo de Don Juan?

Ahora ... a trabajar ustedes



Demuestra que el algoritmo de Don Juan es correcto:

- termina en una cantidad finita de pasos
- cumple su propósito, es decir, **ordena** los datos

Termina en una cantidad finita de pasos



- La cantidad de números a ordenar es finita, digamos n
- En cada vuelta, tachamos un número en la hoja original y lo escribimos en la hoja nueva
- Después de n vueltas, todos los números en la hoja original están tachados y, debido al paso 4, el algoritmo termina

Cumple su propósito: ordena los datos



Demostración **por inducción**:

- **Caso base.** El primer número tachado en la hoja original y escrito en la hoja nueva es el menor de todos (criterio de selección) y está ordenado (único número en la hoja nueva): ✓
- **Hipótesis inductiva.** Los k primeros números tachados en la hoja original y escritos en la hoja nueva son los k números más chicos y están ordenados
- **Por demostrar** (usando la hipótesis inductiva). ...

Por demostrar, usando la hipótesis inductiva



Los $k+1$ primeros números tachados en la hoja original y escritos en la hoja nueva son los $k+1$ números más chicos y están ordenados:

- los primeros k números en la hoja nueva son los k más chicos (por hipótesis inductiva) y están tachados en la hoja original (por paso 2); el siguiente número que pasa a la hoja nueva es el menor de los no tachados (por criterio de selección) \rightarrow el $k+1$ más chico
- los primeros k números en la hoja nueva están ordenados (por hipótesis inductiva); el siguiente número que se escribe al final de la hoja nueva no es menor que ninguno de los k números que ya están en la hoja nueva (por criterio de selección) \rightarrow queda ordenado

El algoritmo *selection sort*

Para la secuencia inicial de datos, A :

1. Definir una secuencia ordenada, B , inicialmente vacía
2. Buscar el menor dato x en A
3. Sacar x de A e insertarlo al final de B
4. Si quedan elementos en A , volver a 2.

¿Cuál es la complejidad de *selection sort*?



Raciocinio para determinar la complejidad de *selection sort*

Buscar el menor dato en A significa revisar A entero: $O(n)$

Este proceso se hace una vez por cada dato: n veces

La complejidad es entonces $n \cdot O(n) = O(n^2)$

Otra forma de calcular la complejidad de *selection sort*

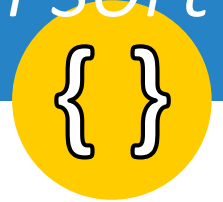
También se puede hacer de manera explícita:

Buscar el mínimo cuesta $n - 1$, y el siguiente $n - 2$, y así:

$$T(n) = \sum_{i=1}^{n-1} i = \frac{n^2 - n}{2}$$

$$T(n) \in O(n^2)$$

Complejidad de memoria de *selection sort*



Selection Sort se puede hacer en un solo **arreglo**, ya que $|A| + |B| = n$

Eso significa que no necesita memoria adicional (excepto ...)

Los algoritmos que hacen esto se conocen como *in place*

Don Juan tiene ahora otro problema



Don Juan quiere cambiar 5 ovejas del lote A al lote B

Necesita actualizar el cambio en ambas hojas

¿Cómo lo hace para no tener que volver a ordenar todo?

Inserción en una lista ordenada



Insertar pocos elementos ordenadamente es ... ¿barato?

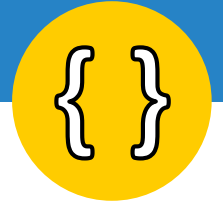
¿Cómo podemos usar este hecho para ordenar?

El algoritmo *insertion sort*

Para la secuencia inicial de datos, A :

1. Definir una secuencia ordenada, B , inicialmente vacía
2. Tomar el primer dato x de A y sacarlo de A
3. Insertar x en B de manera que B quede ordenado
4. Si quedan elementos en A , volver a 2.

¿Cómo se hace una inserción?



Depende de la estructura de datos usada para almacenar la lista

Se suele usar **arreglos**, pero también se puede usar **listas ligadas**

En cualquier caso, el algoritmo no necesita memoria adicional

Los dos pasos de la inserción



1. Primero, hay que buscar donde corresponde insertar el elemento
2. Luego, hay que llevar a cabo la inserción propiamente tal

¿Cuál es la complejidad usando **arreglos**?

¿Y con **listas ligadas**?

Insertar en un arreglo

El primer paso —la búsqueda— podemos hacerlo en $O(\log n)$ con búsqueda binaria

Pero ... para insertar x hay que desplazar todos los elementos $> x$ un lugar hacia la derecha (o hacia abajo) $\rightarrow O(n)$

Por lo tanto, en **arreglos**, **insertar** es $O(n)$

datos
ordenados:
I N O R S T

próximo dato
a ser insertado
ordenadamente: G

último dato
insertado
ordenadamente: N

| | | | | | | | | | | | | | |
|---|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| S | O | R | T | I | N | G | E | X | A | M | P | L | E |
| O | <u>S</u> | R | T | I | N | G | E | X | A | M | P | L | E |
| O | R | <u>S</u> | T | I | N | G | E | X | A | M | P | L | E |
| O | R | S | <u>T</u> | I | N | G | E | X | A | M | P | L | E |
| I | O | R | S | <u>T</u> | N | G | E | X | A | M | P | L | E |
| I | <u>N</u> | O | R | S | <u>T</u> | G | E | X | A | M | P | L | E |
| G | <u>I</u> | <u>N</u> | <u>O</u> | <u>R</u> | <u>S</u> | <u>T</u> | E | X | A | M | P | L | E |
| E | <u>G</u> | <u>I</u> | <u>N</u> | <u>O</u> | <u>R</u> | <u>S</u> | <u>T</u> | X | A | M | P | L | E |
| E | G | I | N | O | R | S | <u>T</u> | X | A | M | P | L | E |
| A | <u>E</u> | <u>G</u> | <u>I</u> | <u>N</u> | <u>O</u> | <u>R</u> | <u>S</u> | <u>T</u> | X | M | P | L | E |
| A | E | G | I | <u>M</u> | <u>N</u> | <u>O</u> | <u>R</u> | <u>S</u> | <u>T</u> | X | P | L | E |
| A | E | G | I | M | N | O | <u>P</u> | <u>R</u> | <u>S</u> | <u>T</u> | X | L | E |
| A | E | G | I | <u>L</u> | <u>M</u> | <u>N</u> | <u>O</u> | <u>P</u> | <u>R</u> | <u>S</u> | <u>T</u> | X | E |
| A | E | <u>E</u> | <u>G</u> | <u>I</u> | <u>L</u> | <u>M</u> | <u>N</u> | <u>O</u> | <u>P</u> | <u>R</u> | <u>S</u> | <u>T</u> | <u>X</u> |
| A | E | E | G | I | L | M | N | O | P | R | S | T | X |

Insertar en una lista (doblemente) ligada

Para el primer paso es necesario revisar toda la lista: $O(n)$

Teniendo el nodo donde corresponde insertar, hacerlo es $O(1)$

Por lo tanto, en **listas ligadas**, **insertar** es $O(n)$

El algoritmo *insertion sort*

Para la secuencia inicial de datos, A :

1. Definir una secuencia ordenada, B , inicialmente vacía
2. Tomar el primer dato x de A y sacarlo de A
3. Insertar x en B de manera que B quede ordenado
4. Si quedan elementos en A , volver a 2.

Demostración de finitud

En cada paso se saca un elemento de A y se inserta en B

Cuando no quedan elementos en A , el algoritmo termina

La inserción requiere como máximo recorrer todo B

Como A y B son finitos, el algoritmo termina en tiempo finito

Demostración, por inducción, de que cumple con su propósito

PD: Al terminar la n -ésima iteración, B se encuentra ordenada

Caso Base: Después de la primera iteración, B tiene un solo dato

→ B está ordenada

Hipótesis Inductiva: Después de la i -ésima iteración, B está ordenada

Demostraremos que después de la iteración $i + 1$, B está ordenada

Extraemos el primer elemento de A , y lo insertamos ordenadamente en B .

Termina el paso $i + 1$ y B tiene $i + 1$ elementos ordenados

En particular, al terminar el algoritmo después del paso n , B está ordenada.

Vimos que *insertion sort* es $O(n^2)$



Pero, ¿qué tiempo toma si los datos vienen ordenados?

A E E G I L M N O P R S T X

insertionSort(A, n):

for $i = 1 \dots n - 1$:

$j = i$

while $(j > 0) \wedge (A[j] < A[j - 1])$:

Intercambiar $A[j]$ con $A[j - 1]$

$j = j - 1$

Complejidad de *insertion sort*



Parecería que la complejidad de *insertion sort* depende de qué tan ordenados vienen los datos

¿Cómo podemos medir “qué tan ordenados vienen los datos”?

Inversiones

Sea A un arreglo con n números distintos de 1 a n

Si $i < j$ pero $A[i] > A[j]$, entonces se dice que el par ordenado (i, j) es una **inversión**

El número de inversiones es una métrica de **desorden**

Inversiones: ejemplo

P.ej., el arreglo

$$A = [34 \ 8 \ 64 \ 51 \ 32 \ 21]$$

tiene 9 inversiones:

(34, 8), (34, 32), (34, 21), (64, 51), (64, 32), (64, 21),
(51, 32), (51, 21), (32, 21)

¿Cómo depende *insertion sort* del número de inversiones?



Tenemos un arreglo A de largo n que tiene k **inversiones**

¿Cuánto tiempo toma *insertion sort* en ordenar A ?

¿Cuántas inversiones se arreglan con un intercambio?

insertionSort(A, n):

for $i = 1 \dots n - 1$:

$j = i$

while $(j > 0) \wedge (A[j] < A[j - 1])$:

Intercambiar $A[j]$ con $A[j - 1]$

$j = j - 1$

Antes de cada intercambio se hace una comparación entre los datos con índices j y $j-1$

Los datos se intercambian sólo si el par de índices $(j-1, j)$ es una inversión

- es decir, si $A[j-1] > A[j]$

Por lo tanto, **cada intercambio de elementos (adyacentes) en el arreglo deshace exactamente una inversión**

Además, cada elemento se compara al menos una vez

Complejidad de *insertion sort*



La complejidad es entonces $O(n + k)$

cada dato se compara
al menos una vez

número de inversiones

¿Qué valor tiene k en el mejor caso? ¿Y el en peor?

- mejor caso: 0 (no hay inversiones)
- peor caso: $(n^2 - n)/2$ (todos los pares posibles son inversiones → el arreglo está ordenado de mayor a menor)

¿Qué hay del *caso promedio*?

El caso promedio



¿Cuál es el número promedio de inversiones en un arreglo con n elementos?

- lo vamos a definir como **el promedio aritmético del número de inversiones en cada una de las $n!$ permutaciones de los n elementos**

Suponemos que

- no hay elementos repetidos*
- todas las permutaciones de los n elementos son igualmente probables (de que aparezcan como input)

(* \rightarrow podemos suponer que los n elementos son los números $0, 1, 2, \dots, n-1$)

¿Cómo calculamos el promedio aritmético del número de inversiones de las $n!$ permutaciones?

- podríamos contar el número de inversiones en cada permutación, luego sumar y finalmente dividir por $n!$

En vez de contar, sumar y dividir, observemos que

... para cualquier permutación L , consideremos la permutación inversa Lr (hay $n!/2$ parejas distintas de permutaciones definidas de esta manera):

- p.ej., si $n = 10$ y $L = \{ 8, 1, 4, 9, 0, 3, 5, 2, 7, 6 \}$
... entonces $Lr = \{ 6, 7, 2, 5, 3, 0, 9, 4, 1, 8 \}$

Tomemos cualquier par de elementos (x, y) , con $y \neq x$

L y Lr tienen la propiedad de que en **exactamente una** de ellas el par ordenado de los índices de x y y es una inversión:

- p.ej., si $(x, y) = (9, 5)$, entonces el par de los índices es $(3, 6)$, y
... $(3, 6)$ **es** una inversión en L , ya que $9 > 5$, pero **no** es una inversión en Lr
- p.ej., si $(x, y) = (0, 7)$, entonces el par de los índices es $(4, 8)$, y
... $(4, 8)$ **no** es una inversión en L , ya que $0 < 7$, pero **sí** es una inversión en Lr

Es decir, un par de índices (i,j) es una inversión en L o es una inversión en Lr (y no cabe otra posibilidad)

... así, el número total de inversiones en L más el número total de inversiones en Lr debe ser igual al número total de pares posibles entre n datos: $n(n-1)/2$

Es decir, cada dos permutaciones — L y su respectiva Lr — el número de inversiones es exactamente $n(n-1)/2$

... por lo tanto, una permutación promedio tiene la mitad de esta cantidad de inversiones: $n(n-1)/4 = O(n^2)$

Complejidad de *insertion sort*

La cantidad de inversiones promedio en un arreglo de n elementos distintos es $O(n^2)$

... por lo que *insertion sort* es $O(n^2)$ en el caso promedio

Ahora, más allá de *insertion sort* ...

Si un algoritmo sólo corrige una inversión por intercambio, no puede ordenar más rápido que $O(n^2)$ en promedio y por lo tanto en el peor caso