

Introducción

Motivación

Ordenación por selección

Número de operaciones de un programa (función ops)

Ordenación por inserción

Resumen

Algoritmos y Estructuras de Datos II

Ordenación elemental

11 y 13 de marzo de 2019

Contenidos

- 1 Introducción
- 2 Motivación
- 3 Ordenación por selección
 - Idea
 - Ejemplo
 - Algoritmo
 - Comando for
 - Análisis
- 4 Número de operaciones de un programa (función ops)
- 5 Ordenación por inserción
 - Ejemplo
 - Algoritmo
 - Análisis

Generalidades

Toda la información sobre la materia se encuentra en la wiki,
accesible desde

wiki.cs.famaf.unc.edu.ar

Algoritmos y Estructuras de Datos

Programación imperativa:

- Algoritmos y Estructuras de Datos I
 - pre- y post- condiciones
 - “qué” hace un algoritmo
- Algoritmos y Estructuras de Datos II
 - “cómo” hace el algoritmo

Ejemplo de “qué” y “cómo” de un algoritmo

Ejemplo:

un algoritmo para contar los ceros de una secuencia finita de enteros.

- ¿**Qué** hace?

devuelve (calcula, computa) el número de ocurrencias del cero en la secuencia dada.

- ¿**Cómo** lo hace? Hay varias posibilidades, por ejemplo:

recorre la secuencia de izquierda a derecha incrementando un contador cada vez que observa un cero.

Análisis de algoritmos

Analizar el “cómo” permite

- predecir el tiempo de ejecución (eficiencia en tiempo)
- predecir el uso de memoria (eficiencia en espacio)
- predecir el uso de otros recursos
- comparar distintos algoritmos para un mismo problema

Organización de la materia

La materia está organizada en tres partes:

- Análisis de algoritmos.
 - Cómo se ejecutan los algoritmos y estimar cuánto trabajo realiza.
- Estructuras de datos.
 - Tipos de datos concretos y abstractos.
- Algoritmos avanzados.
 - Algunas técnicas para resolver problemas algorítmicos.

Problema del pintor

Un pintor tarda una hora y media en pintar una pared de 3 metros de largo. ¿Cuánto tardará en pintar una de 5 metros de largo?

3 metros	↔	90 minutos
1 metro	↔	30 minutos
5 metros	↔	150 minutos

Solución: dos horas y media.

El trabajo de pintar la pared es **proporcional** a su longitud.

Problema del profe de Algoritmos 2

El profe de esta materia tarda media hora en ordenar alfabéticamente 100 exámenes. ¿Cuánto tardará en ordenar 200 exámenes?

Razonamiento similar

100 exámenes \longleftrightarrow 1/2 hora

200 exámenes \longleftrightarrow 1 hora

Solución: una hora.

¿Está bien? ¿Es el trabajo de ordenar exámenes **proporcional** a la cantidad de exámenes a ordenar?

Otros problemas

del pintor

*Un pintor tarda una hora y media en pintar una pared **cuadrada** de 3 metros de lado. ¿Cuánto tardará en pintar una de 5 metros de lado?*

9 metros cuadrados	↔	90 minutos
1 metro cuadrado	↔	10 minutos
25 metros cuadrados	↔	250 minutos

Solución: cuatro horas y 10 minutos.

El trabajo de pintar la pared cuadrada es **proporcional** a su superficie, que es proporcional al cuadrado del lado.

Otros problemas

el del globo esférico

Si lleva cinco horas inflar un globo aerostático esférico de 2 metros de diámetro, ¿cuánto llevará inflar uno de 4 metros de diámetro?

El trabajo de inflar el globo es **proporcional** a su volumen, que es proporcional al cubo del diámetro ($V = \frac{\pi d^3}{6}$).

diámetro = 2	↔	k metros cúbicos	↔	5 horas
diámetro = 4	↔	8k metros cúbicos	↔	40 horas

Solución: cuarenta horas.

Algoritmos de ordenación

Para resolver el problema del profe de esta materia, es necesario

- establecer a qué es proporcional la tarea de ordenar exámenes,
- estudiar/inventar métodos de ordenación,
- asumiremos la existencia de elementos o items a ordenar,
- relacionados por un orden total,
- que deben ordenarse de menor a mayor y
- que no necesariamente son diferentes entre sí.

¿Cómo?

Reflexionemos sobre lo siguiente:

- ¿Qué significa que una secuencia de exámenes, números, palabras, etc. esté ordenada?
- ¿Cómo hacen para controlar si una secuencia de números está ordenada?
 - (a esta pregunta la vamos a continuar en el práctico y en el laboratorio)
- ¿Cómo harían para ordenar de menor a mayor ciertos datos o ciertas cosas físicas que están desordenados/as?
 - números
 - cartas de un juego,
 - palabras,
 - exámenes.

Ordenación por selección

Idea

- Es el algoritmo de ordenación más sencillo (pero no el más rápido),
- **selecciona** el menor de todos, lo coloca en el primer lugar apartándolo del resto,
- **selecciona** el menor de todos **los restantes**, lo coloca en el segundo lugar apartándolo del resto,
- **selecciona** el menor de todos **los restantes**, lo coloca en el tercer lugar apartándolo del resto,
- ... (*en cada uno de estos pasos ordena un elemento*) ...
- hasta terminar.

Ordenación por selección

Ejemplo

[9, 3, 1, 3, 5, 2, 7]

```
[      9, 3, 1, 3, 5, 2, 7]
[1,      9, 3, 3, 5, 2, 7]
[1, 2,      9, 3, 3, 5, 7]
[1, 2, 3,      9, 3, 5, 7]
[1, 2, 3, 3,      9, 5, 7]
[1, 2, 3, 3, 5,      9, 7]
[1, 2, 3, 3, 5, 7,      9]
```

[1, 2, 3, 3, 5, 7, 9]

Ordenación por selección

En Haskell

```
ssort :: [Integer] -> [Integer]
```

```
ssort [] = []
```

```
ssort ns = m : ssort r
```

```
    where m = minimum ns
```

```
          r = erase m ns
```

```
erase :: Integer -> [Integer] -> [Integer]
```

```
erase m (n:ns) | m == n = ns
```

```
               | otherwise = n : erase m ns
```


Ordenación por selección

Ejecución en Haskell

- cronometrar su ejecución con las secuencias
 - 5000..1
 - 10000..1
- y comparar los tiempos obtenidos.
- resulta que en el segundo caso demora el cuádruple!

Ordenación por selección

Ejemplo: ahora con arreglos

9	3	1	3	5	2	7
9	3	1	3	5	2	7
1	3	9	3	5	2	7
1	3	9	3	5	2	7
1	2	9	3	5	3	7
1	2	9	3	5	3	7
1	2	3	9	5	3	7

1	2	3	9	5	3	7
1	2	3	3	5	9	7
1	2	3	3	5	9	7
1	2	3	3	5	9	7
1	2	3	3	5	9	7
1	2	3	3	5	7	9

Introducción

Motivación

Ordenación por selección

Número de operaciones de un programa (función ops)

Ordenación por inserción

Resumen

Idea

Ejemplo

Algoritmo

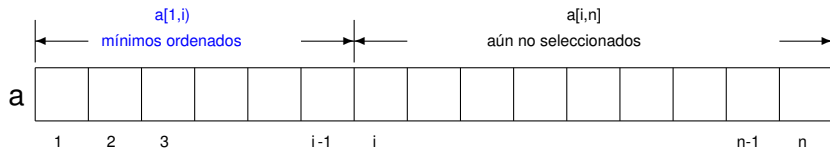
Comando for

Análisis

Demo (www.sorting-algorithms.com)

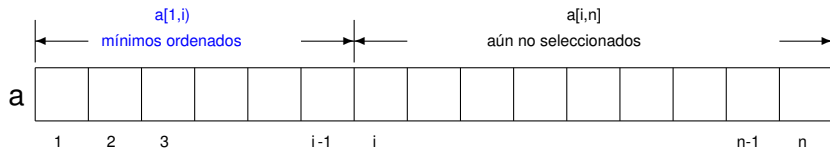
Ordenación por selección

En un arreglo



Ordenación por selección

Invariante



- Invariante:

- el arreglo a es una permutación del original,
- un segmento inicial $a[1,i]$ del arreglo está ordenado, y
- dicho segmento contiene los elementos mínimos del arreglo.

Ordenación por selección

Swap o intercambio

{Pre: $a = A \wedge 1 \leq i, j \leq n$ }

proc swap (**in/out** a: **array**[1..n] **of** T, **in** i,j: **nat**)

var tmp: T

 tmp:= a[i]

 a[i]:= a[j]

 a[j]:= tmp

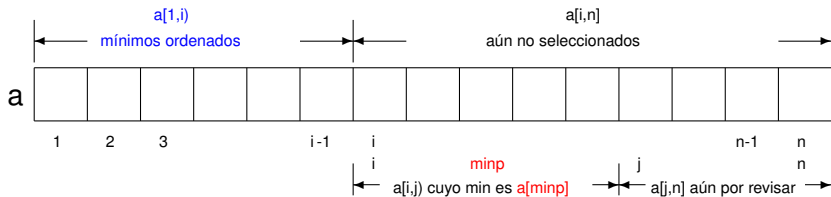
end proc

{Post: $a[i] = A[j] \wedge a[j] = A[i] \wedge \forall k. k \notin \{i, j\} \Rightarrow a[k]=A[k]$ }

¡Garantiza permutación!

Ordenación por selección

Invariante de la función de selección



- Invariante:

- invariante anterior, y
- el mínimo del segmento $a[i, j]$ está en la posición minp .

Comando for

Fragmentos de la siguiente forma aparecen con frecuencia:

```
k:= n
do k ≤ m → C
    k:= k+1
od
```

Por simplicidad, lo reemplazaremos por

```
for k:= n to m do C od
```

siempre que k no se modifique en C .

Además, asumiremos que el **for** declara la variable k , cuya vida dura sólo durante la ejecución del ciclo.

Comando for

Reemplazo en min_pos_from

```

fun min_pos_from (a: array[1..n] of T, i: nat) ret minp: nat
  var j: nat
  minp:= i
  j:= i+1
  do j ≤ n → if a[j] < a[minp] then minp:= j fi
    j:= j+1
  od
end fun

fun min_pos_from (a: array[1..n] of T, i: nat) ret minp: nat
  minp:= i
  for j:= i+1 to n do if a[j] < a[minp] then minp:= j fi
  od
end fun

```

Comando for

Reemplazo en selection_sort

```
proc selection_sort (in/out a: array[1..n] of T)
  var i, minp: nat
  i := 1
  do i < n → minp := min_pos_from(a,i)
    swap(a,i,minp)
    i := i+1
  od
end proc
```

Comando for

En selection_sort

```
proc selection_sort (in/out a: array[1..n] of T)
```

```
  var minp: nat
```

```
  for i:= 1 to n do
```

```
    minp:= min_pos_from(a,i)
```

```
    swap(a,i,minp)
```

```
  od
```

```
end proc
```

```
fun min_pos_from (a: array[1..n] of T, i: nat) ret minp: nat
```

```
  minp:= i
```

```
  for j:= i+1 to n do if a[j] < a[minp] then minp:= j fi
```

```
  od
```

```
end fun
```

Comando for ... downto

Fragmentos de la siguiente forma también aparecen con cierta frecuencia:

```

k:= m
do k ≥ n → C
    k:= k-1
od
  
```

Por simplicidad, lo reemplazaremos por

```

for k:= m downto n do C od
  
```

siempre que k no se modifique en C.

Ordenación por selección en lenguajes de programación

- En python.
- En c.

Ordenación por selección en python

```
def min_pos_from(c,fr):  
    min_pos = fr  
    for j in range(fr+1,len(c)):  
        if c[j] < c[min_pos]:  
            min_pos = j  
    return min_pos
```

```
def swap(b,x,y):  
    b[x], b[y] = b[y], b[x]  
  
for i in range(0,len(a)):  
    minp = min_pos_from(a,i)  
    swap(a,i,minp)
```


Ordenación por selección en c

```
#define n 7
```

```
void swap(int * b, unsigned int x, unsigned int y) {  
    int tmp = b[x];  
    b[x] = b[y];  
    b[y] = tmp;  
}
```

```
int min_pos_from(int * c, unsigned int fr) {  
    unsigned int min_pos = fr;  
    for (unsigned int j = fr+1; j < n; j++) {  
        if (c[j] < c[min_pos]) min_pos = j;  
    }  
    return min_pos;  
}
```

Ordenación por selección en c

```
int a[n] = {9, 3, 1, 3, 5, 2, 7};
```

```
int main() {  
    unsigned int minp = 0;  
    for (unsigned int i = 0; i < n; i++) {  
        minp = min_pos_from(a,i);  
        swap(a,i,minp);  
    }  
}
```

Problema del profe

Cuando el algoritmo es la ordenación por selección

- ¿Cómo se respondería el problema del profe si el algoritmo utilizado por él fuera el de ordenación por selección?
- ¿Cuánto más trabajo resulta ordenar 200 exámenes que 100 con este algoritmo?
- ¿Cuánto trabajo es ordenar 200 exámenes (con este algoritmo)?
- ¿Cuánto trabajo es ordenar 100 exámenes (con este algoritmo)?
- ¿Cuánto trabajo es ordenar n exámenes (con este algoritmo)?

Problema del profe

Análisis

- Para contestar estas preguntas habría que **analizar** el algoritmo de ordenación por selección, es decir, contar cuántas operaciones elementales realiza.
- Cuántas sumas, asignaciones, llamadas a funciones, comparaciones, intercambios, etc.
- En vez de eso, se elige una operación **representativa**.
- ¿Qué es una operación **representativa**?
- Una tal que se repite más que o tanto como cualquier otra.
- Hay que buscar la que **más se repite**.

Analizando el procedimiento selection_sort

- selection_sort **contiene un ciclo**,
- **allí** debe estar la operación que más se repite,
- encontramos **una llamada** a la función min_pos_from y **una llamada** al procedimiento swap,
- el procedimiento swap **es constante** (siempre realiza 3 asignaciones elementales),
- la función min_pos_from, en cambio, **tiene un ciclo**,
- nuevamente **allí** debe estar la operación que más se repite,
- encontramos **una comparación** entre elementos de a, y **una asignación** (condicionada al resultado de la comparación).

Analizando ordenación por selección

Conclusión

- La **operación que más se repite es la comparación** entre elementos de a ,
- **toda otra operación se repite a lo sumo de manera proporcional** a esa,
- por lo tanto, **la comparación** entre elementos de a **es representativa** del trabajo de la ordenación por selección.
- Esto es habitual: para medir la eficiencia de los algoritmos de ordenación es habitual considerar el número de comparaciones entre elementos del arreglo.
- Veremos luego que acceder (o modificar) una celda de un arreglo es **constante**: su costo no depende de cuál es la celda, ni de la longitud del arreglo.

¿Cuántas comparaciones realiza la ordenación por selección?

- Al llamarse a `min_pos_from(a,i)` se realizan $n-i$ comparaciones.
- `selection_sort` llama a `min_pos_from(a,i)` para $i \in \{1, 2, \dots, n-1\}$.
- por lo tanto, en total son $(n-1) + (n-2) + \dots + (n-(n-1))$ comparaciones.
- es decir, $(n-1) + (n-2) + \dots + 1 = \frac{n*(n-1)}{2}$ comparaciones.

Resolviendo el problema del profe

Con la fórmula obtenida

Para un arreglo de tamaño n , son $\frac{n*(n-1)}{2}$ comparaciones.

100 exámenes	↔	4950 comparaciones	↔	1/2 hora
200 exámenes	↔	19900 comparaciones	↔	2 horas

Solución: 2 horas.

Resolviendo el problema del profe

Con una fórmula simplificada

Como $\frac{n*(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$, el número de comparaciones es proporcional a n^2 .

100 exámenes	↔	10000 comparaciones	↔	1/2 hora
200 exámenes	↔	40000 comparaciones	↔	2 horas

Solución: 2 horas.

Conviene utilizar la expresión n^2 para contestar la pregunta; es más sencillo y da el mismo resultado.

Número de operaciones de un programa

- Una vez que uno sabe qué **operación** quiere contar, debe imaginar una ejecución arbitraria, genérica del programa intentando contar el número de veces que esa ejecución arbitraria realizará **dicha operación**.
- Ése es el verdadero método para contar.
- Es imprescindible comprender **cómo** se ejecuta el programa.
- A modo de ayuda, en las filminas que siguen se da un método imperfecto para ir aprendiendo.
- El método supone que ya sabemos cuál **operación** queremos contar.

Número de operaciones de un programa

Secuencia de comandos

- Una secuencia de comandos se ejecuta de manera secuencial, del primero al último.
- La secuencia se puede escribir horizontalmente:
 $C_1; C_2; \dots; C_n$
- o verticalmente

 C_1 C_2 \vdots C_n

Número de operaciones de un programa

Secuencia de comandos

- Para contar cuántas veces se ejecuta **la operación**, entonces, se cuenta cuántas veces se ejecuta en el primero, cuántas en el segundo, etc. y luego se suman los números obtenidos:

- $\text{ops}(C_1; C_2; \dots; C_n) = \text{ops}(C_1) + \text{ops}(C_2) + \dots + \text{ops}(C_n)$

- $\text{ops} \left(\begin{array}{c} C_1 \\ C_2 \\ \vdots \\ C_n \end{array} \right) = \text{ops}(C_1) + \text{ops}(C_2) + \dots + \text{ops}(C_n)$

Número de operaciones de un programa

Comando **skip**

- El comando **skip** equivale a una secuencia vacía:
- $\text{ops}(\mathbf{skip}) = 0$

Número de operaciones de un programa

Comando `for`

- El comando **for** $k := n$ **to** m **do** $C(k)$ **od** “equivale” también a una secuencia:
- **for** $k := n$ **to** m **do** $C(k)$ **od** “equivale” a

 $C(n)$ $C(n+1)$ \vdots $C(m)$

Número de operaciones de un programa

Comando `for`

- De esta “equivalencia” resulta

$$\begin{aligned} \text{ops}(\mathbf{for\ } k:= n \mathbf{ to\ } m \mathbf{ do\ } C(k) \mathbf{ od}) &= \\ &= \text{ops}(C(n)) + \text{ops}(C(n+1)) + \dots + \text{ops}(C(m)) \end{aligned}$$

- que también se puede escribir

$$\text{ops}(\mathbf{for\ } k:= n \mathbf{ to\ } m \mathbf{ do\ } C(k) \mathbf{ od}) = \sum_{k=n}^m \text{ops}(C(k))$$

Número de operaciones de un programa

Comando **for** (una salvedad importante)

La ecuación

$$\text{ops}(\text{for } k:= n \text{ to } m \text{ do } C(k) \text{ od}) = \sum_{k=n}^m \text{ops}(C(k))$$

solamente vale cuando **no hay interés en contar las operaciones que involucran el índice k** implícitas en el **for**: inicialización, comparación con la cota m, incremento; ni el cómputo de los límites n y m. Por eso escribimos “equivale” entre comillas.

Número de operaciones de un programa

Comando condicional if

- El comando **if b then C else D fi** se ejecuta evaluando la condición b y luego, en función del valor de verdad que se obtenga, ejecutando C (caso verdadero) o D (caso falso).
- Para contar cuántas veces se ejecuta **la operación**, entonces, se cuenta cuántas veces se la ejecuta durante la evaluación de b y luego cuántas en la ejecución de C o D
- $\text{ops}(\text{if } b \text{ then } C \text{ else } D \text{ fi}) = \begin{cases} \text{ops}(b) + \text{ops}(C) & \text{caso } b \text{ V} \\ \text{ops}(b) + \text{ops}(D) & \text{caso } b \text{ F} \end{cases}$

Número de operaciones de un programa

Asignación

- El comando $x:=e$ se ejecuta evaluando la expresión e y modificando la posición de memoria donde se aloja la variable x con el valor de e .



$$\text{ops}(x:=e) = \begin{cases} \text{ops}(e)+1 & \text{si se desea contar la asignación} \\ & \text{o las modificaciones de memoria} \\ \text{ops}(e) & \text{en caso contrario} \end{cases}$$

- Tener en cuenta que la evaluación de e puede implicar la llamada a funciones auxiliares cuyas operaciones deben ser también contadas.

Número de operaciones de una expresión

- Similares ecuaciones se pueden obtener para la evaluación de expresiones.
- Por ejemplo, para evaluar la expresión $e < f$, primero se evalúa la expresión e , luego se evalúa la expresión f y luego se comparan dichos valores.
-

$$\text{ops}(e < f) = \begin{cases} \text{ops}(e) + \text{ops}(f) + 1 & \text{si se cuentan comparaciones} \\ \text{ops}(e) + \text{ops}(f) & \text{caso contrario} \end{cases}$$

Ejemplo: número de comparaciones de la ordenación por selección

```
proc selection_sort (in/out a: array[1..n] of T)
  var minp: nat
  for i:= 1 to n do
    minp:= min_pos_from(a,i)
    swap(a,i,minp)
  od
end proc

fun min_pos_from (a: array[1..n] of T, i: nat) ret minp: nat
  minp:= i
  for j:= i+1 to n do if a[j] < a[minp] then minp:= j fi
  od
end fun
```

Ejemplo: número de comparaciones de la ordenación por selección

$$\begin{aligned}
 & \text{ops}(\text{selection_sort}(a)) \\
 = & \text{ops}(\mathbf{for\ } i := 1 \mathbf{ to\ } n \mathbf{ do\ } \text{minp} := \text{min_pos_fr} \dots ; \text{swap} \dots \mathbf{od}) \\
 = & \sum_{i=1}^n \text{ops}(\text{minp} := \text{min_pos_from}(a,i); \text{swap}(a,i,\text{minp})) \\
 = & \sum_{i=1}^n (\text{ops}(\text{minp} := \text{min_pos_from}(a,i)) + \text{ops}(\text{swap}(a,i,\text{minp}))) \\
 = & \sum_{i=1}^n \text{ops}(\text{minp} := \text{min_pos_from}(a,i)) \\
 = & \sum_{i=1}^n \text{ops}(\text{min_pos_from}(a,i)) \\
 = & \sum_{i=1}^n \text{ops}(\text{minp} := i; \mathbf{for\ } j := i+1 \mathbf{ to\ } n \mathbf{ do\ } \text{if} \dots \mathbf{fi\ } \mathbf{od})
 \end{aligned}$$

Ejemplo: número de comparaciones de la ordenación por selección

$$\begin{aligned}
 & \text{ops}(\text{selection_sort}(a)) \\
 = & \sum_{i=1}^n \text{ops}(\text{minp}:= i; \text{for } j:= i+1 \text{ to } n \text{ do if } \dots \text{fi od}) \\
 = & \sum_{i=1}^n (\text{ops}(\text{minp}:= i) + \text{ops}(\text{for } j:= i+1 \text{ to } n \text{ do if } \dots \text{fi od})) \\
 = & \sum_{i=1}^n \text{ops}(\text{for } j:= i+1 \text{ to } n \text{ do if } \dots \text{fi od}) \\
 = & \sum_{i=1}^n \sum_{j=i+1}^n \text{ops}(\text{if } a[j] < a[\text{minp}] \text{ then minp}:= j \text{ fi}) \\
 = & \sum_{i=1}^n \sum_{j=i+1}^n (\text{ops}(a[j] < a[\text{minp}]) + \text{ops}(\text{minp}:= j)) \text{ o ops}(\text{skip}) \\
 = & \sum_{i=1}^n \sum_{j=i+1}^n \text{ops}(a[j] < a[\text{minp}])
 \end{aligned}$$

Ejemplo: número de comparaciones de la ordenación por selección

$$\begin{aligned}
 \text{ops}(\text{selection_sort}(a)) &= \sum_{i=1}^n \sum_{j=i+1}^n \text{ops}(a[j] < a[\text{minp}]) \\
 &= \sum_{i=1}^n \sum_{j=i+1}^n 1 \\
 &= \sum_{i=1}^n (n-i) \\
 &= \sum_{i=0}^{n-1} i \\
 &= \frac{n \cdot (n-1)}{2} \\
 &= \frac{n^2}{2} - \frac{n}{2}
 \end{aligned}$$

Ejemplo: número de intercambios de la ordenación por selección

$$\begin{aligned}
 & \text{ops}(\text{selection_sort}(a)) \\
 = & \text{ops}(\mathbf{for} \ i:= 1 \ \mathbf{to} \ n \ \mathbf{do} \ \text{minp}:= \text{min_pos_fr} \dots ; \text{swap} \dots \ \mathbf{od}) \\
 = & \sum_{i=1}^n \text{ops}(\text{minp}:= \text{min_pos_from}(a,i); \text{swap}(a,i,\text{minp})) \\
 = & \sum_{i=1}^n (\text{ops}(\text{minp}:= \text{min_pos_from}(a,i)) + \text{ops}(\text{swap}(a,i,\text{minp}))) \\
 = & \dots = \sum_{i=1}^n (0 + \text{ops}(\text{swap}(a,i,\text{minp}))) \\
 = & \sum_{i=1}^n \text{ops}(\text{swap}(a,i,\text{minp})) \\
 = & \sum_{i=1}^n 1 \\
 = & n
 \end{aligned}$$

Conclusión del ejemplo

- Número de comparaciones de la ordenación por selección:
 $\frac{n^2}{2} - \frac{n}{2}$
- Número de intercambios de la ordenación por selección: n
- Esto significa que la operación de **intercambio no es representativa** del comportamiento de la ordenación por selección, ya que el número de comparaciones crece más que proporcionalmente respecto a los intercambios.
- Por otro lado, pudimos contar las operaciones de manera **exacta**.

Ordenación por inserción

- **No siempre** es posible contar el **número exacto** de operaciones.
- Un ejemplo de ello lo brinda otro algoritmo de ordenación: la ordenación por inserción.
- Es un algoritmo que se utiliza por ejemplo en juegos de cartas, cuando es necesario mantener un gran número de cartas en las manos, en forma ordenada.
- Cada carta que se levanta de la mesa, se inserta en el lugar correspondiente entre las que ya están en las manos, manteniéndolas ordenadas.

Ordenación por inserción

Ejemplo

9	3	1	3	5	2	7
9	3	1	3	5	2	7
3	9	1	3	5	2	7
3	1	9	3	5	2	7
1	3	9	3	5	2	7
1	3	3	9	5	2	7
1	3	3	5	9	2	7

1	3	3	5	2	9	7
1	3	3	2	5	9	7
1	3	2	3	5	9	7
1	2	3	3	5	9	7
1	2	3	3	5	7	9

Introducción

Motivación

Ordenación por selección

Número de operaciones de un programa (función ops)

Ordenación por inserción

Resumen

Ejemplo

Algoritmo

Análisis

Demo (www.sorting-algorithms.com)

Ordenación por inserción

En Haskell

```
isort :: [Integer] -> [Integer]
```

```
isort [] = []
```

```
isort (n:ns) = insert n (isort ns)
```

```
insert :: Integer -> [Integer] -> [Integer]
```

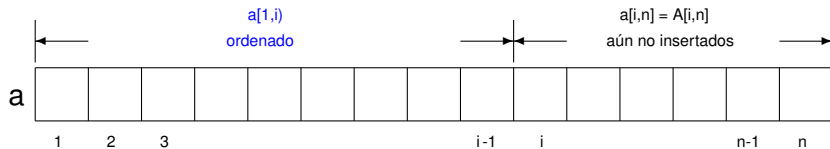
```
insert n [] = [n]
```

```
insert n (m:ms) | n <= m = n : m : ms
```

```
                | otherwise = m : insert n ms
```

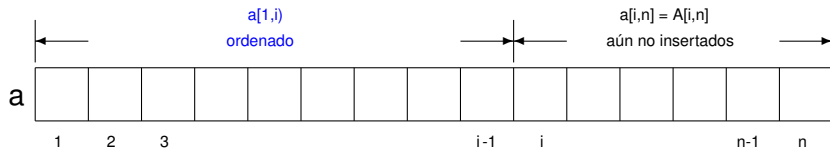
Ordenación por inserción

En un arreglo



Ordenación por inserción

Invariante



- Invariante:

- el arreglo a es una permutación del original y
- un segmento inicial $a[1, i]$ del arreglo está ordenado.
- (pero en general $a[1, i]$ **no** contiene los mínimos del arreglo)

Ordenación por inserción

Pseudocódigo

{Pre: $n \geq 0 \wedge a = A$ }

proc insertion_sort (**in/out** a: **array**[1..n] **of** T)

for i:= 2 **to** n **do**

 {Inv: Invariante de recién}

 insert(a,i)

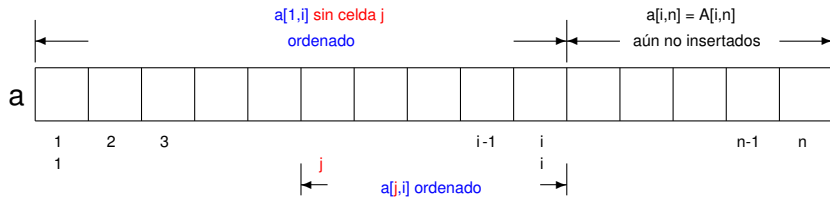
od

end proc

{Post: a está ordenado y es permutación de A}

Ordenación por inserción

Invariante del procedimiento de inserción



- Invariante:

- el arreglo a es una permutación del original
- $a[1,j]$ sin celda j está ordenado, y
- $a[j,i]$ también está ordenado.

Ordenación por inserción

Procedimiento de inserción

{Pre: $0 < i \leq n \wedge a = A$ }

proc insert (in/out a: array[1..n] of T, in i: nat)

var j: nat

j:= i

{Inv: Invariante de recién}

do $j > 1 \wedge a[j] < a[j - 1]$ \rightarrow swap(a,j-1,j)

j:= j-1

od

end proc

{Post: a[1,i] está ordenado \wedge a es permutación de A}

Ordenación por inserción

Todo junto

```
proc insertion_sort (in/out a: array[1..n] of T)
```

```
  for i:= 2 to n do
```

```
    insert(a,i)
```

```
  od
```

```
end proc
```

```
proc insert (in/out a: array[1..n] of T, in i: nat)
```

```
  var j: nat
```

```
  j:= i
```

```
  do  $j > 1 \wedge a[j] < a[j - 1]$   $\rightarrow$  swap(a,j-1,j)
```

```
    j:= j-1
```

```
  od
```

```
end proc
```

Ordenación por inserción en python

```
def swap(b,x,y):  
    b[x], b[y] = b[y], b[x]  
  
def insert(c,k):  
    j = k  
    while j > 0 and c[j] < c[j-1]:  
        swap(c,j,j-1)  
        j = j - 1  
  
for i in range(1,len(a)):  
    insert(a,i)
```

Ordenación por inserción en c

```
void insert(int * c, unsigned int k) {  
    unsigned int j = k;  
    while (j > 0 && c[j] < c[j-1]) {  
        swap(a,j,j-1);  
        j:= j-1;  
    }  
}  
  
int main() {  
    for (unsigned int i = 1; i < n; i++) {  
        insert(a,i);  
    }  
}
```

Número de Comparaciones e intercambios

Procedimiento insert(a,i)

si el valor de i es ...	comparaciones		intercambios	
	mín	máx	mín	máx
2	1	1	0	1
3	1	2	0	2
4	1	3	0	3
⋮	⋮	⋮	⋮	⋮
n	1	n-1	0	n-1
total insertion_sort	n - 1	$\frac{n^2}{2} - \frac{n}{2}$	0	$\frac{n^2}{2} - \frac{n}{2}$

Ordenación por inserción, casos

- mejor caso: arreglo ordenado, n comparaciones y 0 intercambios.
- peor caso: arreglo ordenado al revés, $\frac{n^2}{2} - \frac{n}{2}$ comparaciones e intercambios, es decir, del orden de n^2 .
- caso promedio: del orden de n^2 .

Número de operaciones de un programa

El ciclo `do`

- El ciclo `do b → C od` (o equivalente `while b do C od`) se ejecuta evaluando la condición `b`, y dependiendo de si su valor es `V` o `F` se continúa de la siguiente manera:
 - si su valor fue `F`, la ejecución termina inmediatamente
 - si su valor fue `V`, la ejecución continúa con la ejecución del cuerpo `C` del ciclo, y luego de eso vuelve a ejecutarse todo el ciclo nuevamente.
- Es decir que su ejecución es una secuencia de evaluaciones de la condición `b` y ejecuciones del cuerpo `C` que finaliza con la primera evaluación de `b` que dé `F`.

Número de operaciones de un programa

El ciclo **do**

Es decir, la ejecución del ciclo **do** $b \rightarrow C$ **od** “equivale” a la ejecución de

```

if  $b$  then  $C$ 
    if  $b$  then  $C$ 
        if  $b$  then  $C$ 
            if  $b$  then  $C$ 
                ...  $jj$  indefinidamente !!
            else skip
        else skip
    else skip
else skip
  
```

Número de operaciones de un programa

El ciclo **do**

$$\text{ops}(\mathbf{do\ } b \rightarrow \mathbf{C\ } \mathbf{od}) = \text{ops}(b) + \sum_{k=1}^n d_k$$

donde

- n es el número de veces que se ejecuta el cuerpo del **do**
- d_k es el número de operaciones que realiza la k -ésima ejecución del cuerpo C del ciclo y la subsiguiente evaluación de la condición o guarda b

Resumen

- Hemos analizado dos algoritmos de ordenación
 - ordenación por selección
 - ordenación por inserción
- la ordenación por selección hace siempre el mismo número de comparaciones, del orden de n^2 .
- la ordenación por inserción también es del orden de n^2 en el peor caso (arreglo ordenado al revés) y en el caso medio,
- la ordenación por inserción es del orden de n en el mejor caso (arreglo ordenado),
- la ordenación por inserción realiza del orden de n^2 swaps (contra n de la ordenación por selección) en el peor caso.

Problema del profe de algoritmos 2

- Con cualquiera de los dos algoritmos la respuesta es 2 horas,
- salvo que se trate de un conjunto ya ordenado o casi ordenado, en cuyo caso:
 - ordenación por inserción es del orden de n ,
 - y por ello la respuesta sería: 1 hora.

Repaso de la ordenación por selección

```
proc selection_sort (in/out a: array[1..n] of T)
```

```
  var minp: nat
```

```
  for i:= 1 to n do
```

```
    minp:= min_pos_from(a,i)
```

```
    swap(a,i,minp)
```

```
  od
```

```
end proc
```

```
fun min_pos_from (a: array[1..n] of T, i: nat) ret minp: nat
```

```
  minp:= i
```

```
  for j:= i+1 to n do if a[j] < a[minp] then minp:= j fi
```

```
  od
```

```
end fun
```

Se lo puede abreviar omitiendo la función auxiliar.

Forma abreviada de la ordenación por selección

```
proc selection_sort (in/out a: array[1..n] of T)
  var minp: nat
  for i:= 1 to n do
    minp:= i
    for j:= i+1 to n do
      if a[j] < a[minp] then minp:= j fi
    od
    swap(a,i,minp)
  od
end proc
```

Repaso de la ordenación por inserción

```

proc insertion_sort (in/out a: array[1..n] of T)
  for i:= 2 to n do
    insert(a,i)
  od
end proc

```

```

proc insert (in/out a: array[1..n] of T, in i: nat)
  j:= i
  do  $j > 1 \wedge a[j] < a[j - 1]$   $\rightarrow$  swap(a,j-1,j)
    j:= j-1
  od
end proc

```

También puede abreviarse omitiendo el procedimiento auxiliar.

Forma abreviada de la ordenación por inserción

```
proc insertion_sort (in/out a: array[1..n] of T)
  for i:= 2 to n do
    j:= i
    do  $j > 1 \wedge a[j] < a[j - 1]$   $\rightarrow$  swap(a,j-1,j)
      j:= j-1
    od
  od
end proc
```


Demo (www.sorting-algorithms.com)

- Ejecución de ordenación por selección
 - entrada aleatoria
 - casi ordenada
 - invertida
 - con repeticiones
- Ejecución de ordenación por inserción
 - entrada aleatoria
 - casi ordenada
 - invertida
 - con repeticiones
- Comparación y conclusiones.

Introducción

Motivación

Ordenación por selección

Número de operaciones de un programa (función ops)

Ordenación por inserción

Resumen

Reflexión sobre paralelismo

¿Qué provecho podríamos sacar a los algoritmos que hemos visto si contáramos con varios o muchos procesadores capaces de cooperar entre ellos?