… previamente en IIC2133

Selection Sort

- **Ordenamos para buscar**
  - Búsqueda binaria es muy eficiente
    - O(log(n))
- **Algoritmo de ordenamiento simple**
  - Selecciona ordenadamente
  - Ubica al final de la salida ordenada
- **Es correcto**
  - Termina
  - Cumple su función (ordena)
    - Lo probamos por inducción
- **Determinamos su complejidad**
  - $O(n^2)$

SelectionSort

input : Secuencia $A[0 \ldots n-1]$, largo $n \geq 2$
output: $\varnothing$

`SelectionSort` (A, n)**:**

```
1   for i = 0 ... n − 2 :
2       min ← i
3       for j = i + 1 ... n − 1 :
4           if A[j] < A[min] :
5               min ← j
6       A[i] ⇋ A[min]
```

Versión *in place* en tiempo $\mathcal{O}(n^2)$

Selection Sort

- Modificaciones
  - Utilizar dos criterios de orden
    - Por valor y luego por pinta
    - Por pinta y luego por valor

- Atributos generales
  - $O(n^2)$
  - O(n) swaps
  - O(1) en memoria – In place
  - No adaptativo – sin mejor caso
  - No estable

¿Cuándo usar Selection Sort?

Insertion sort

Clase 02

IIC 2133 - Sección 2

Prof. Mario Droguett

Sumario

Continuamos ordenando el curso

- El sistema sigue malo, pero al menos ya ordenamos la lista de asistentes
- Sin embargo, ahora la Universidad nos pide cambiar 5 estudiantes de la sección 1 a la sección 2
- Tenemos que actualizar las listas de ambos cursos manteniendo el orden
- ¿Hay alguna forma de actualizar sin tener que ordenar todo de nuevo?

Una posible solución

- En lugar de ordenar desde cero, podemos **insertar** elementos respetando el orden
- Insertar pocos elementos ordenadamente debiera ser (relativamente) barato
- ¿Cómo podemos usar este hecho para ordenar?
- ¿Cómo se deben insertar los elementos para asegurar el orden? ¿Al final?

El algoritmo `InsertionSort`

input : Secuencia de datos A
output: Nueva secuencia de datos B, ordenada

`InsertionSort` *(A)*:

1. Definir secuencia B, inicialmente vacía
2. Tomar el primer dato x de A
3. Sacar x de A
4. Insertar x en B de manera que B quede ordenada
5. Si quedan datos en A, volver a la línea 2

return B

¿En qué se diferencia `InsertionSort` de `SelectionSort`?

Objetivos de la clase

- ☐ Distinguir diferencias de `InsertionSort` y `SelectionSort`
- ☐ Demostrar correctitud de `InsertionSort`
- ☐ Comprender el efecto de la inserción general en las EDD básicas
- ☐ Comprender concepto de inversión y su rol en la ordenación
- ☐ Comprender casos (mejor, peor y promedio) en análisis en complejidad
- ☐ Determinar complejidad por caso para `InsertionSort`

Sumario

El algoritmo `InsertionSort`

input : Secuencia de datos A
output: Nueva secuencia de datos B, ordenada

`InsertionSort` *(A)*:

1. Definir secuencia B, inicialmente vacía
2. Tomar el primer dato x de A
3. Sacar x de A
4. Insertar x en B de manera que B quede ordenada
5. Si quedan datos en A, volver a la línea 2

return B

La idea empleada por los algoritmos es diferente

- `SelectionSort`: *elegir ordenadamente*
- `InsertionSort`: *insertar ordenadamente*

Correctitud de `InsertionSort`

Este algoritmo es correcto y demostraremos su correctitud de manera similar al caso de `SelectionSort`

input : Secuencia de datos A
output: Nueva secuencia de datos B, ordenada

`InsertionSort` *(A)*:

1 Definir secuencia B, inicialmente vacía
2 Tomar el primer dato x de A
3 Sacar x de A
4 Insertar x en B de manera que B quede ordenada
5 Si quedan datos en A, volver a la línea 2
return B

¿En qué se diferencian ambas demostraciones?

Correctitud de `InsertionSort`

Demostración (finitud)

Sea n la cantidad (finita) de valores en la secuencia original A.

En cada iteración del algoritmo se borra un valor de A (línea 3).

Luego se inserta dicho valor en B (línea 4), para lo cual se requiere como máximo recorrer todo B. Como en todo minuto B tiene largo finito, la inserción toma una cantidad finita de pasos.

Luego de n iteraciones, todos los valores en A fueron borrados. La condición de término (línea 5) garantiza que el algoritmo termina.

Por lo tanto, el algoritmo termina en una cantidad finita de pasos. □

Correctitud de `InsertionSort`

Demostración (ordenación)

Consideremos la propiedad

$$\mathbf{P}(\mathbf{n}) \quad := \quad \text{Luego de la } n \text{ -ésima iteración,}\ B \text{ se encuentra ordenada}$$

1. **Caso base.** $\mathbf{P}(\mathbf{1})$ corresponde al estado de la secuencia B luego de insertar el primer elemento. Dado que B solo tiene un elemento, está ordenada.

Correctitud de `InsertionSort`

Demostración (ordenación)

$$\mathbf{P(n)} \quad := \quad \text{Luego de la } n \text{ -ésima iteración,}\\ B \text{ se encuentra ordenada}$$

2. **H.I.** Suponemos que luego de la n-ésima iteración, B está ordenada.

   P.D. Luego de la iteración $(n+1)$-ésima iteración, B sigue ordenada.

   En la iteración $n+1$ extraemos el primer elemento de A, a saber a'. Si el estado de B es

   $$a_1 \leq \cdots \leq a_i \leq a_{i+1} \leq \cdots \leq a_n$$

   y a' cumple $a_i \leq a' \leq a_{i+1}$, lo insertamos de forma que B satisface

   $$a_1 \leq \cdots \leq a_i \leq a' \leq a_{i+1} \leq \cdots \leq a_n$$

   Es decir, luego de la iteración $n+1$, B está ordenada. □

Inserción de datos

`InsertionSort` y `SelectionSort` no solo se diferencian en su estretegia.

También difieren en cómo realizan la inserción del dato en cada iteración

- `SelectionSort` inserta siempre al final de la nueva EDD
- `InsertionSort` inserta **de forma que la nueva EDD quede ordenada**, no necesariamente haciéndolo al final de esta

Nuestro análisis de complejidad de la inserción es insuficiente: interesa poder insertar en **cualquier posición**

¿Cómo cambia nuestro análisis de complejidad de la inserción?

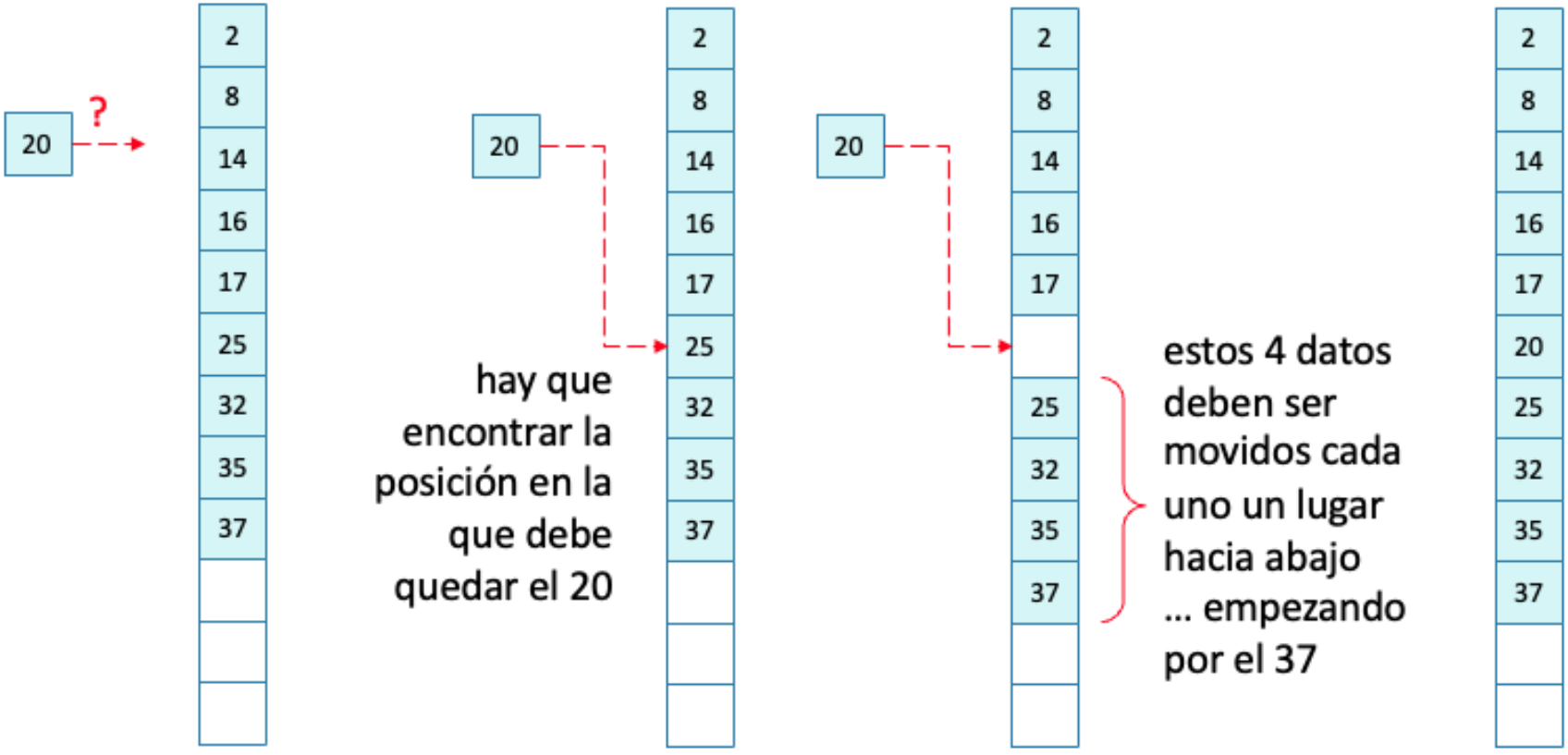
Inserción de datos

Toda inserción de un dato en una estructura involucra dos pasos

1. Búsqueda de la posición en que se debe insertar el dato
2. Inserción propiamente tal, quizás incluyendo mover otros datos

¿Cuál es la complejidad de estos pasos usando arreglos? ¿Y listas ligadas?

Inserción en arreglo ordenado



Inserción en arreglo ordenado

Para un arreglo de largo n

La búsqueda de la posición adecuada requiere un algoritmo de búsqueda

- Si usamos búsqueda binaria, toma tiempo $\mathcal{O}(\log(n))$

Para insertar un elemento en una posición es posible que haya que desplazar elementos ya ordenados

- Si el elemento es mayor que todos, se agrega al final
- En caso contrario, hay que desplazar **todos los mayores a él**
- Esto toma tiempo $\mathcal{O}(n)$

La inserción en arreglos ordenados toma tiempo $\mathcal{O}(n)$

Inserción en lista (doblemente) ligada

Consideramos una lista ligada con n elementos ordenados

La búsqueda de la posición adecuada requiere revisar la lista completa

- Esto toma tiempo $\mathcal{O}(n)$

Para insertar un elemento en una posición solo hay que reasignar punteros

- A lo más involucra 2 elementos previamente ordenados
- Esto toma tiempo $\mathcal{O}(1)$

La inserción en listas ligadas ordenadas toma tiempo $\mathcal{O}(n)$

Complejidad de `InsertionSort`

Ya sabemos que la complejidad de la inserción es $\mathcal{O}(n)$ para EDDs básicas.

Ejercicio

¿Cuál es la complejidad de `InsertionSort`?

input : Secuencia de datos A
output: Nueva secuencia de datos B, ordenada

`InsertionSort` *(A)*:

1 Definir secuencia B, inicialmente vacía
2 Tomar el primer dato x de A
3 Sacar x de A
4 Insertar x en B de manera que B quede ordenada
5 Si quedan datos en A, volver a la línea 2
return B

Ejecución de `InsertionSort`

Para buscar la intuición, consideremos una versión in place del algoritmo

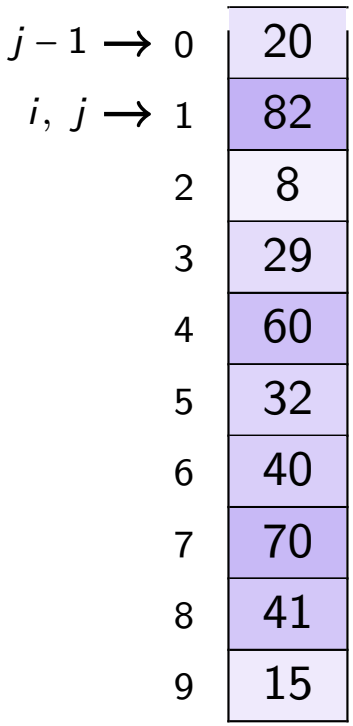
- Tal como con `SelectionSort`, $|A| + |B| = n$ se mantiene luego de cada iteración, i.e. es un **invariante**
- Podemos usar un mismo arreglo para ambas secuencias

input : Secuencia $A[0 \ldots n-1]$, largo $n \geq 2$
output: $\varnothing$

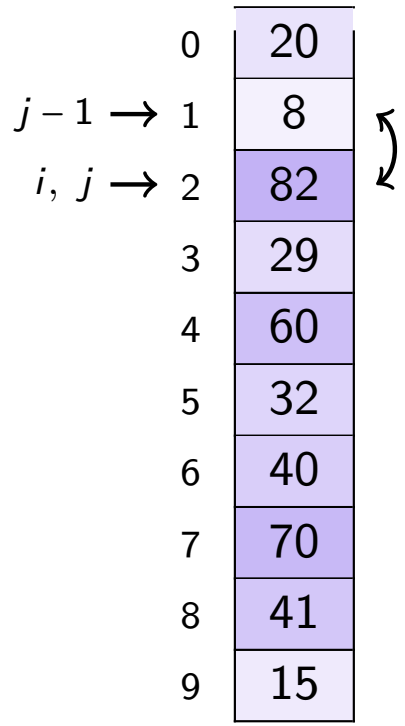
`InsertionSort` (A, n)**:**

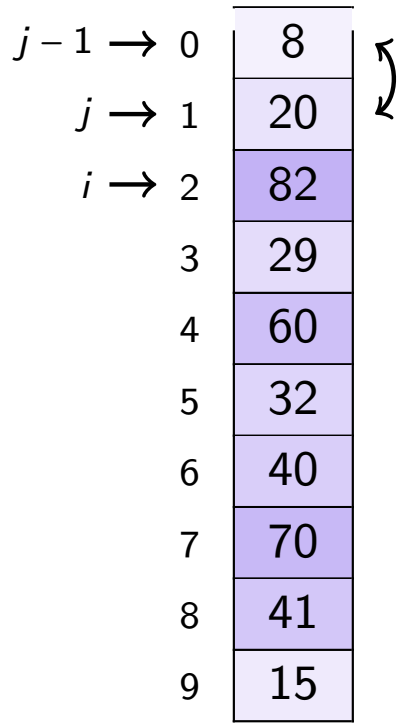
```
1    for i = 1 ... n − 1 :
2        j = i
3        while (j > 0) ∧ (A[j] < A[j − 1]) :
4            Intercambiar A[j] con A[j − 1]
5            j = j − 1
```

Ejecución de `InsertionSort`



Iteración con $i = 1$:
no hay intercambio
entre $A[1]$ y $A[0]$





Iteración con $i = 2$:
hay dos intercambios
$A[2] \leftrightarrow A[1]$ y $A[1] \leftrightarrow A[0]$

`InsertionSort` puede hacer $\mathcal{O}(n)$ intercambios por elemento

Complejidad de `InsertionSort`

Complejidad de `InsertionSort`

input : Secuencia $A[0 \ldots n-1]$, largo $n \geq 2$
output: $\varnothing$

`InsertionSort` (A, n)**:**

```
1    for i = 1 ... n − 1 :
2        j = i
3        while (j > 0) ∧ (A[j] < A[j − 1]) :
4            Intercambiar A[j] con A[j − 1]
5            j = j − 1
```

¿Qué tiempo toma si los datos vienen ordenados?

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

Complejidad de `InsertionSort`

- A diferencia de `SelectionSort` que revisa toda la secuencia A para encontrar el menor elemento, `InsertionSort` siempre toma el primer elemento de A
- Si los datos vienen ordenados, `SelectionSort` no cambia su complejidad
- En `InsertionSort` la inserción se ve beneficiada: sirve recordar dónde termina la lista ordenada para saber dónde insertar
- Pareciera entonces que la complejidad de `InsertionSort` depende de qué tan ordenados están los datos

¿Cómo medir qué tan ordenados vienen los datos?

Sumario

Inversiones

Definición

Sea A un arreglo de n números distintos entre 0 y $n-1$, y sean $0 \leq i, j \leq n-1$ índices del arreglo. Si $i < j$ y $A[i] > A[j]$, entonces diremos que el par ordenado (i, j) es una **inversión**.

Usaremos el número de inversiones en un arreglo como una **métrica de desorden**

¿Esta idea sirve cuando hay repetidos?
¿Y cuando hay números fuera del rango $0 \ldots n-1$?

Inversiones

Definición

Sea A un arreglo de n números y sean $0 \leq i, j \leq n-1$ índices del arreglo. Si $i < j$ y $A[i] > A[j]$, entonces diremos que el par ordenado (i, j) es una **inversión**.

Ejemplo

El arreglo

$A =$

34	8	64	51	32	21
0	1	2	3	4	5

tiene 9 inversiones:

- $(0,1),\ (0,4),\ (0,5)$
- $(2,3),\ (2,4),\ (2,5)$
- $(3,4),\ (3,5)$
- $(4,5)$

Inversiones e `InsertionSort`

Recordemos la versión in place del algoritmo

input : Secuencia $A[0 \ldots n-1]$, largo $n \geq 2$
output: $\varnothing$

`InsertionSort` (A, n)**:**

```
1    for i = 1 ... n − 1 :
2        j = i
3        while (j > 0) ∧ (A[j] < A[j − 1]) :
4            Intercambiar A[j] con A[j − 1]
5            j = j − 1
```

Consideremos un arreglo A de largo n con k inversiones

- ¿Cuántas inversiones se arreglan con un intercambio (línea 4)?
- ¿Cuánto tiempo toma `InsertionSort` en ordenar A?

Inversiones e `InsertionSort`

input : Secuencia $A[0 \ldots n-1]$, largo $n \geq 2$
output: $\varnothing$

`InsertionSort` (A, n)**:**

```
1    for i = 1 ... n − 1 :
2        j = i
3        while (j > 0) ∧ (A[j] < A[j − 1]) :
4            Intercambiar A[j] con A[j − 1]
5            j = j − 1
```

- Antes de cada intercambio, comparan los datos con índices j y $j-1$
- Los datos se intercambian solo si el par $(j-1, j)$ es una inversión
- Por lo tanto, cada intercambio de datos adyacentes corrije exactamente una inversión
- Además, cada dato se compara al menos una vez

Complejidad de `InsertionSort`

De acuerdo a lo anterior, la complejidad es

$$\mathcal{O}(n+k)$$

- n considera que cada dato se compara al menos una vez
- k es el número de inversiones

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

Mejor y peor caso de `InsertionSort`

En el mejor caso, el arreglo tiene $k = 0$ inversiones

- El arreglo está ordenado y la complejidad en el mejor caso es $\mathcal{O}(n)$

Mejor y peor caso de `InsertionSort`

En el peor caso, el arreglo tiene el número máximo de inversiones

- El arreglo está ordenado decrecientemente
- Al comparar cada índice i con $j > i$, se cuenta una inversión

$$k = (n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

- De forma equivalente, el número de inversiones es el número de pares de índices

$$k = \binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}$$

- La complejidad en el peor caso es $\mathcal{O}(n^2)$

¿Qué hay del caso promedio?

Caso promedio de `InsertionSort`

Haremos el análisis de caso promedio abstrayendo el orden del arreglo

Para esto, consideramos un arreglo A que viene en cualquier orden

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

Para dar una definición supondremos que A no tiene elementos repetidos

Definición

Dado el conjunto S con n elementos, una **permutación** P de S es una secuencia

$$s_1, s_2, \ldots, s_n \quad s_i \in S$$

Para S con $|S| = n$, existen $n!$ permutaciones diferentes

- Cada una es una forma de ordenar los elementos de S
- El arreglo A corresponde a una de esas permutaciones

Caso promedio de `InsertionSort`

Definiremos el número promedio de inversiones como el promedio aritmético del número de inversiones de cada una de las $n!$ permutaciones

En lugar de contar inversiones para cada permutación, usaremos un truco

Caso promedio de `InsertionSort`

Definiremos el número promedio de inversiones como el promedio aritmético del número de inversiones de cada una de las $n!$ permutaciones

En lugar de contar inversiones para cada permutación, usaremos un truco

- Para una permutación P, consideremos su permutación inversa P^r
- Existen $n!/2$ pares de permutaciones acompañadas de su inversa

Ejemplo

Para $n = 3$, las $3! = 6$ permutaciones de $S = \{4, 7, 8\}$ son

- $P_1 = 4, 7, 8$ y $P_1^r = 8, 7, 4$
- $P_2 = 7, 4, 8$ y $P_2^r = 8, 4, 7$
- $P_3 = 7, 8, 4$ y $P_3^r = 4, 8, 7$

Caso promedio de `InsertionSort`

Tomando un par de elementos distintos x, y de S y un par de permutaciones P, P^r, los índices de x e y están en inversión en **alguna de las dos permutaciones**

Ejemplo

Para $S = \{4, 7, 8\}$, el par de elementos $x = 4$ e $y = 8$, y el siguiente par de permutaciones

$P_2 =$

7	4	8
0	1	2

$P_2^r =$

8	4	7
0	1	2

los índices de x, y en P_2 son $(1, 2)$ y en P_2^r, $(1, 0)$

- En P_2, como $1 < 2$ y $4 < 8$, $(1, 2)$ no es inversión
- En P_2^r, como $1 \not< 0$ y $4 < 8$, $(0, 1)$ es inversión

Un par de índices (i, j) es inversión en P o en P^r

Caso promedio de `InsertionSort`

Como cada inversión aparece en una de las dos permutaciones del par P, P^r,

$$\#\text{ inv en } P \;+\; \#\text{ inv en } P^r \;=\; \frac{n(n-1)}{2}$$

Luego, tomando los $n!/2$ pares posibles de permutaciones de la forma P_k, P_k^r, el promedio de inversiones es

$$\begin{aligned}
\frac{\sum_P (\#\text{ inv en } P)}{\#\text{ permutaciones}} &= \frac{\sum_{k=1}^{n!/2} (\#\text{ inv en } P_k \;+\; \#\text{ inv en } P_k^r)}{n!} \\
&= \frac{\frac{n(n-1)}{2} \cdot n!/2}{n!} \\
&= \frac{n(n-1)}{4}
\end{aligned}$$

Una permutación promedio tiene una cantidad de inversiones $\mathcal{O}(n^2)$

Caso promedio de `InsertionSort`

Con este resultado podemos concluir la complejidad de `InsertionSort` en el caso promedio

- La complejidad de `InsertionSort` es de la forma $\mathcal{O}(n + k)$
- En el caso promedio $k \in \mathcal{O}(n^2)$
- Luego, en el caso promedio `InsertionSort` toma tiempo $\mathcal{O}(n^2)$

¡Ojo! Esto va más allá de `InsertionSort`

Caso promedio de algoritmos de ordenación

Teorema

Sea $\mathcal{A}$ un algoritmo de ordenación. Si $\mathcal{A}$ corrige una inversión por intercambio, entonces no puede ordenar más rápido que $\mathcal{O}(n^2)$ en el caso promedio.

Corolario

$\mathcal{A}$ no puede ordenar más rápido que $\mathcal{O}(n^2)$ en el peor caso.

Insertion Sort

- Variantes
  - Shell sort – Donald Shell 1959
    - Compara con elementos a distancias que decrecen
    - $O(n^{3/2})$, $O(n^{4/3})$
  - Binary Insertion sort
    - Si las comparaciones son “caras” utiliza binary seach para mejorar el costo de la inserción
      - O(log(n))
    - El desempeño global es $O(n^2)$
  - Library sort – 2006
    - Bender, Martin Farach-Colton, and Mosteiro
    - Deja “espacios” libres
    - ~ O(n log(n))

Insertion Sort

- Atributos generales
  - $O(n^2)$
  - O(1) en memoria – In place
  - Más eficiente en la práctica que otros O(n2)
  - Adaptativo
    - O(n) mejor caso
  - Estable
  - On Line
    - Puede ordenar a medida que recibe los datos

- ¿Cuándo usar Insertion Sort?

Complejidad de algoritmos de ordenación

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	?	?	?	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Quick Sort	?	?	?	?
Merge Sort	?	?	?	?
Heap Sort	?	?	?	?

Ejercicio (propuesto)

Aplicando las estrategias que vimos hoy

- Escriba el pseudocódigo de la versión in place de `SelectionSort`
- Especifique complejidad del mejor, peor y caso promedio de `SelectionSort` de acuerdo a su propuesta

Sumario

Objetivos de la clase

- ☐ Distinguir diferencias de `InsertionSort` y `SelectionSort`
- ☐ Demostrar correctitud de `InsertionSort`
- ☐ Comprender el efecto de la inserción general en las EDD básicas
- ☐ Comprender concepto de inversión y su rol en la ordenación
- ☐ Comprender casos (mejor, peor y promedio) en análisis en complejidad
- ☐ Determinar complejidad por caso para `InsertionSort`