Algoritmi e Strutture di Dati I

Massimo Franceschet
http://www.sci.unich.it/~francesc
m.franceschet@unich.it

Problema dell'ordinamento

Data in ingresso una sequenza

$$\langle a_1, a_2, \ldots, a_n \rangle$$

fornire in uscita una permutazione

$$\langle a'_1, a'_2, \ldots, a'_n \rangle$$

della sequenza di ingresso tale che

$$a'_1 \leq a'_2, \ldots, \leq a'_n$$

Ordinamento per confronto

Un algoritmo di **ordinamento per confronto** utilizza il confronto tra elementi del vettore per determinare l'ordine relativo degli elementi del vettore.

Dati due elementi a_i e a_j del vettore, un confronto è un test del tipo $a_i \leq a_j$.

Ordinamento sul posto

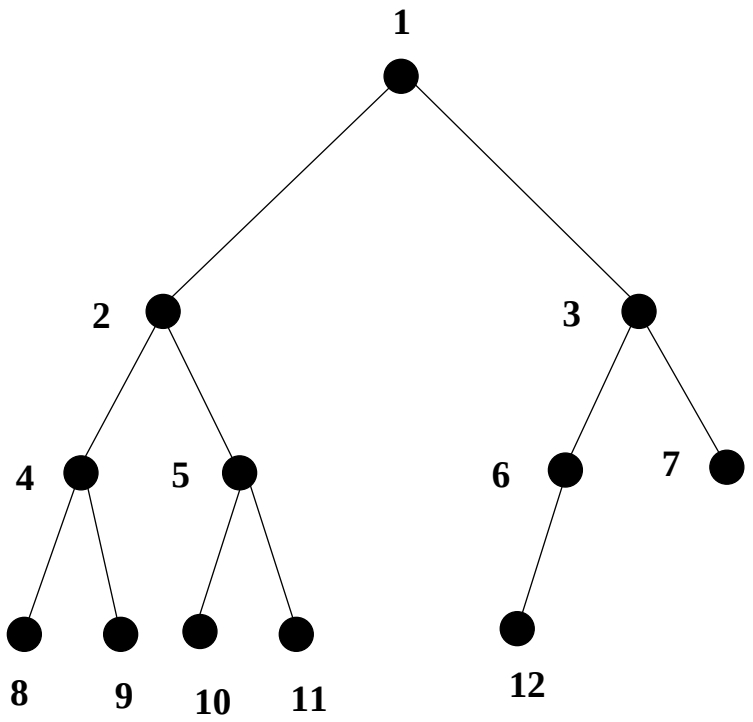
Un algoritmo di ordinamento di un vettore A viene detto **sul posto** (**in-place**) se gli elementi di A vengono riarrangiati all'interno del vettore A, e, in ogni istante, solo un numero costante di elementi vengono tenuti fuori da A.

HeapSort

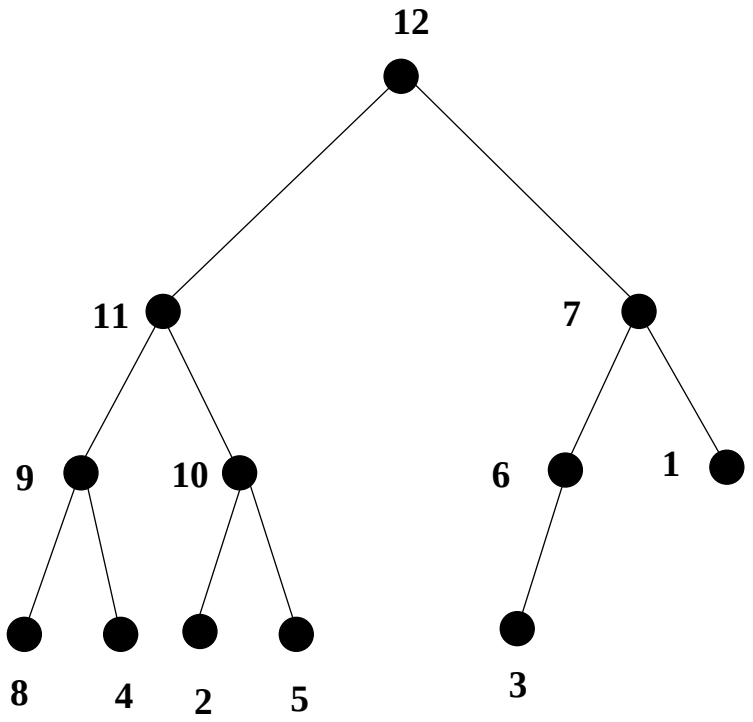
HeapSort è un algoritmo di ordinamento **iterativo**, **per confronto**, e **sul posto** che usa la struttura di dati heap.

HeapSort(A)

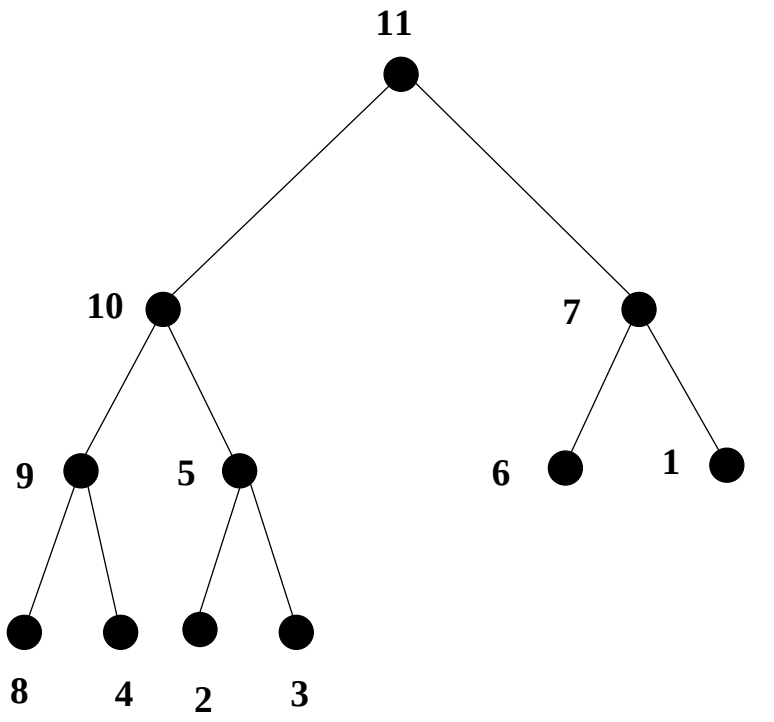
1: $HeapBuild(A)$
2: **for** $i \leftarrow length(A)$ **downto** 2 **do**
3: $\quad Exchange(A, 1, i)$
4: $\quad heapsize(A) \leftarrow heapsize(A) - 1$
5: $\quad Heapify(A, 1)$
6: **end for**



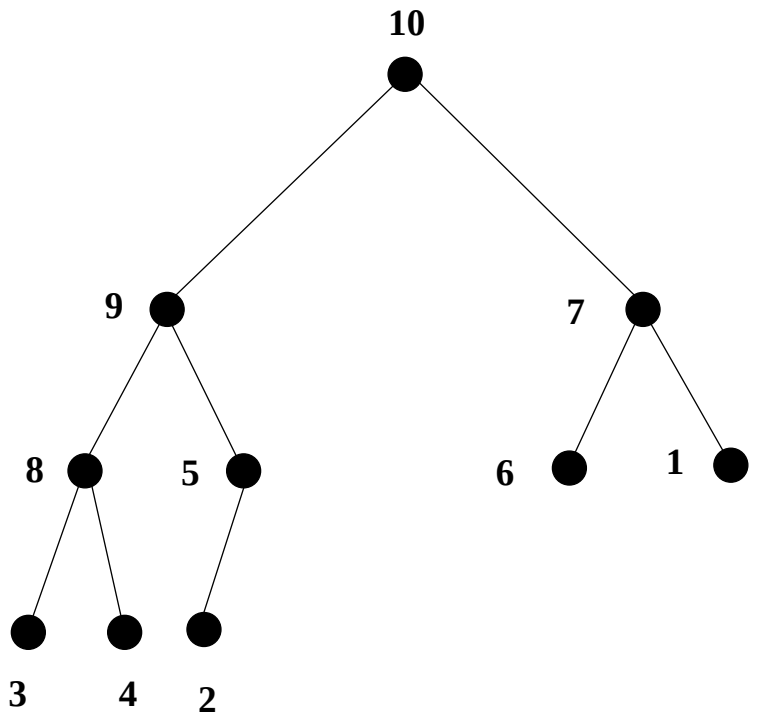
1 2 3 4 5 6 7 8 9 10 11 12



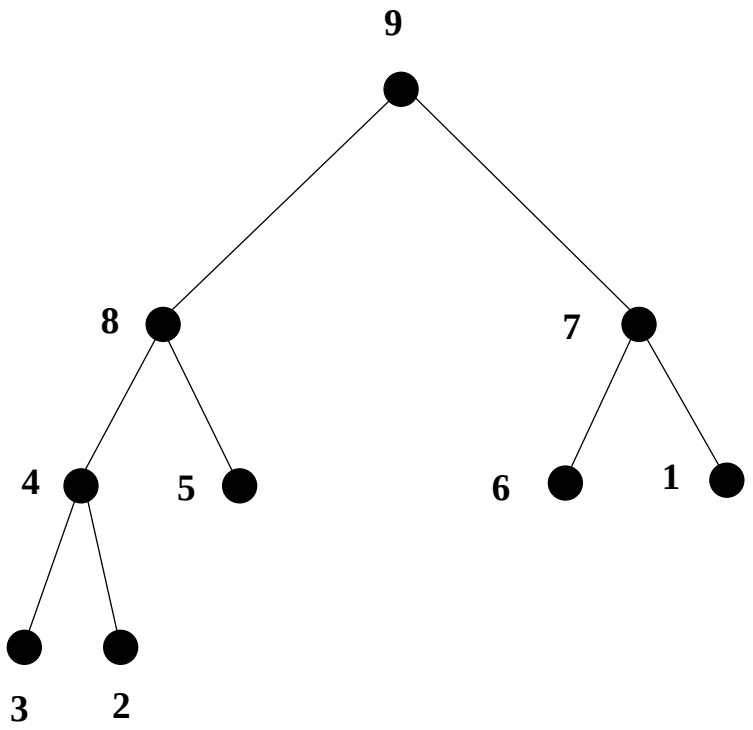
12 11 7 9 10 6 1 8 4 2 5 3



11 10 7 9 5 6 1 8 4 2 3 12



10	9	7	8	5	6	1	3	4	2	11	12



9	8	7	4	5	6	1	3	2	10	11	12

1 2 3 4 5 6 7 8 9 10 11 12

Complessità di HeapSort

Sia $n = length(A)$. La complessità pessima di $HeapSort$ è:

- $HeapBuild$ costa $\Theta(n)$;
- il ciclo for esegue $n - 1$ volte. Ogni iterazione ha complessità $O(\log n)$.

Dunque $HeapSort$ costa

$$O(n) + (n-1) \cdot O(\log n) = O(n + n \log n) = O(n \log n)$$

Vale un limite stretto?

Complessità di HeapSort

HeapSort non è **sensibile alla forma** della sequenza di ingresso.

Ciò significa che il costo di *HeapSort* è indipendente dal numero di coppie che sono già ordinate in partenza.

Infatti, si può dimostrare che, se gli elementi del vettore sono tutti distinti, la **complessità ottima** di *HeapSort* è $\Theta(n \log n)$.

Quindi anche la complessità **media** e quella **pessima** di *HeapSort* sono $\Theta(n \log n)$.

TreeSort

TreeSort è un algoritmo di ordinamento **iterativo**, **per confronto** e non **sul posto** che usa la struttura di dati albero binario di ricerca.

La procedura $TreeSort(A)$ consiste di due passi:

1. gli elementi di A vengono inseriti in un albero binario di ricerca T usando la procedura $TreeInsert$;
2. l'albero T viene visitato in ordine intermedio e gli elementi di T vengono reinseriti nel vettore A.

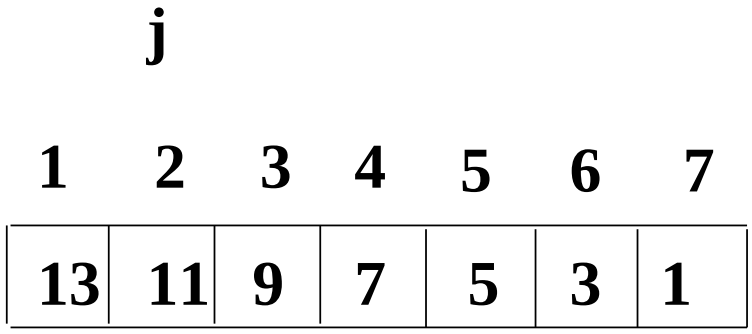
La complessità pessima di questa procedure è $\Theta(n^2)$, quella media e quella ottima sono $\Theta(n \log n)$.

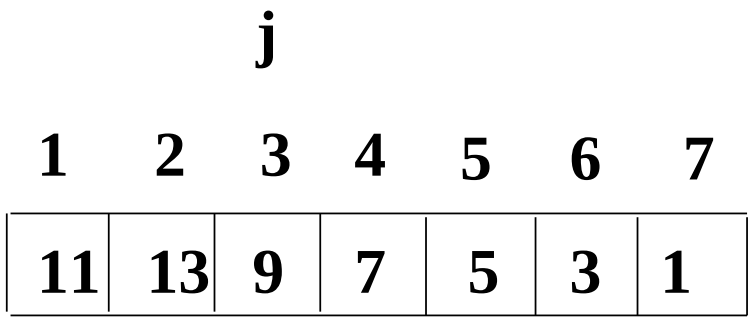
InsertionSort

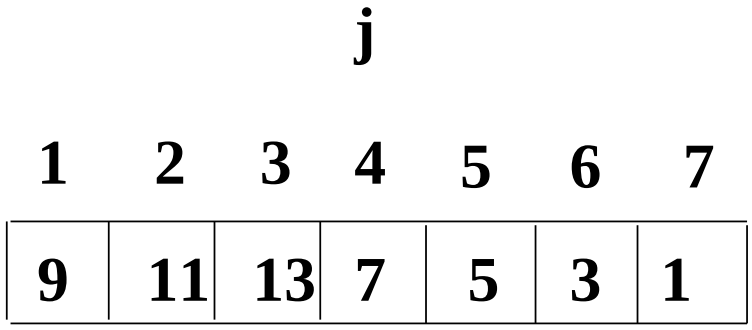
InsertionSort è un algoritmo di ordinamento **iterativo**, **per confronto**, e **sul posto**.

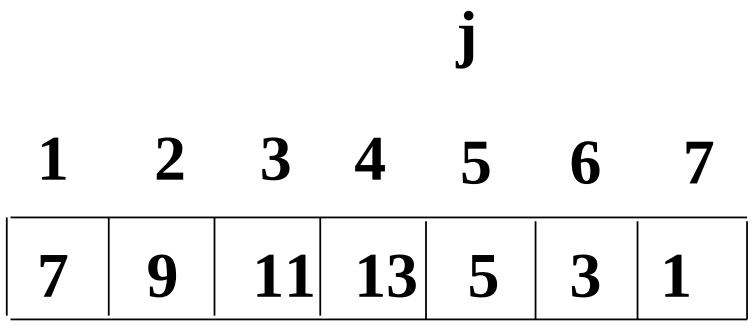
InsertionSort(A)

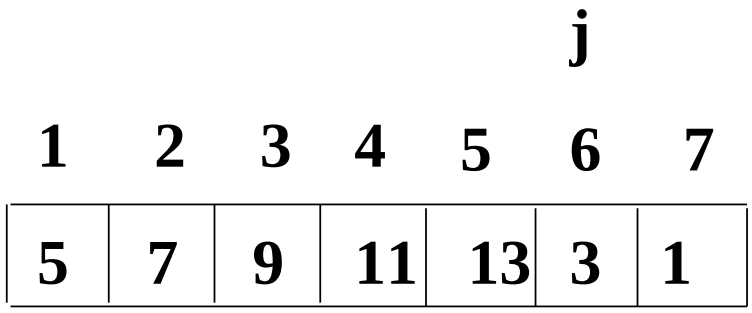
```
1: for j ← 2 to length(A) do
2:     key ← A[j]
3:     i ← j − 1
4:     while (i > 0) and (key < A[i]) do
5:         A[i + 1] ← A[i]
6:         i ← i − 1
7:     end while
8:     A[i + 1] ← key
9: end for
```

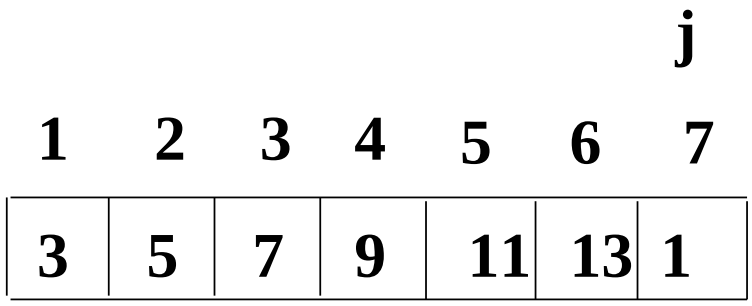

j
1 2 3 4 5 6 7
13 11 9 7 5 3 1

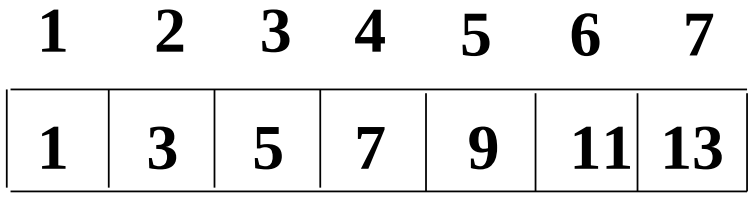

j
1 2 3 4 5 6 7
11 13 9 7 5 3 1


j
1 2 3 4 5 6 7
9 11 13 7 5 3 1


j
1 2 3 4 5 6 7
7 9 11 13 5 3 1


j
1 2 3 4 5 6 7
5 7 9 11 13 3 1


j
1 2 3 4 5 6 7
3 5 7 9 11 13 1


1 2 3 4 5 6 7
1 3 5 7 9 11 13

Complessità ottima di InsertionSort

Sia $n = length(A)$. Il **caso ottimo** si ha quando A è inizialmente ordinato in senso crescente.

In tal caso, il ciclo while non viene mai eseguito, e dunque ogni iterazione del ciclo for ha complessità costante.

Dato che ci sono $n - 1$ iterazioni del ciclo for, la complessità ottima risulta

$$(n-1)\Theta(1) = \Theta(n-1) = \Theta(n)$$

Complessità pessima di InsertionSort

Il **caso pessimo** si ha quando A è inizialmente ordinato in senso decrescente.

In tal caso, il ciclo while viene sempre eseguito fino all'ultimo, cioè da $i = j - 1$ fino a $i = 0$. Dunque ogni iterazione del ciclo for ha complessità $\Theta(j)$.

Dato che il ciclo for itera da $j = 2$ fino a $j = n$, la complessità pessima risulta

$$\sum_{j=2}^{n} \Theta(j) = \Theta(\sum_{j=2}^{n} j) = \Theta(n(n+1)/2 - 1) = \Theta(n^2)$$

Complessità media di InsertionSort

Nel **caso medio**, l'elemento chiave $A[j]$ è più grande della metà degli elementi che lo precedono, e quindi il ciclo while costa $\Theta(j/2)$.

La complessità media risulta

$$\sum_{j=2}^{n} \Theta(j/2) = \Theta(1/2 \sum_{j=2}^{n} j) = \Theta(n(n+1)/4 - 1/2) = \Theta(n^2)$$

QuickSort

QuickSort è un algoritmo di ordinamento **ricorsivo**, **per confronto**, e **sul posto**.

QuickSort usa una procedura ausiliaria $Partition(A, p, r)$ che partiziona il sottovettore $A[p, \ldots, r]$ in due sottovettori

$$A[p, \ldots, q-1]$$

e

$$A[q+1, \ldots, r]$$

per qualche $p \leq q \leq r$, tali che, se $x = A[q]$, per ogni $p \leq i \leq q-1$,

$$A[i] \leq x$$

e, per ogni $q+1 \leq i \leq r$,

$$x \leq A[i]$$

L'elemento $x = A[q]$ viene detto **perno** (o **pivot**) e la procedura ritorna l'indice q.

Partition(A,p,r)

```
1: x ← A[r]
2: i ← p − 1
3: for j ← p to r − 1 do
4:     if A[j] ≤ x then
5:         i ← i + 1
6:         Exchange(A, i, j)
7:     end if
8: end for
9: i ← i + 1
10: Exchange(A, i, r)
11: return i
```

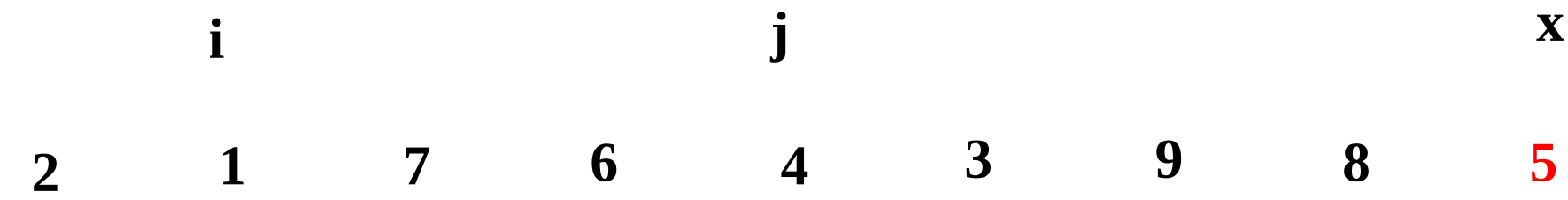
La complessità di $Partition(A, p, r)$ è $\Theta(n)$, ove $n = r - p + 1$ è la dimensione del sottovettore $A[p, \ldots, r]$.


x
i
j
2
1
7
6
4
3
9
8
5


i
j
x
2
1
7
6
4
3
9
8
5


i
j
x
2
1
7
6
4
3
9
8
5

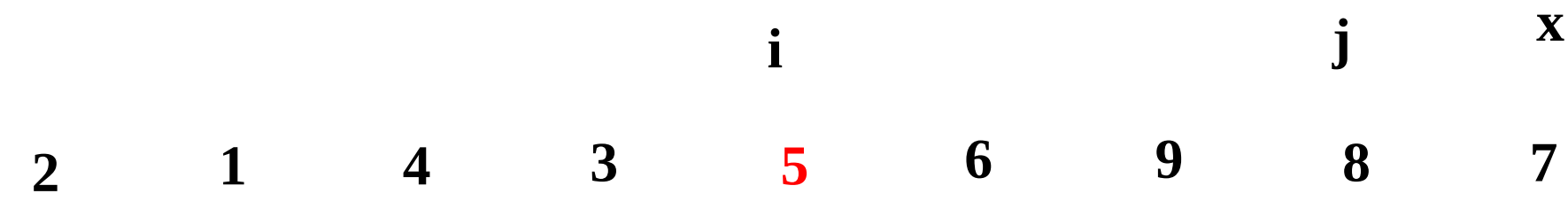

i
j
x
2
1
7
6
4
3
9
8
5


i
j
x
2
1
7
6
4
3
9
8
5


i
j
x
2
1
4
6
7
3
9
8
5


i
j
x
2
1
4
3
7
6
9
8
5


i
j
x
2
1
4
3
7
6
9
8
5


i
j
x
2
1
4
3
5
6
9
8
7

QuickSort

QuickSort(A,p,r)

1: **if** $p < r$ **then**
2: $q \leftarrow Partition(A, p, r)$
3: $QuickSort(A, p, q-1)$
4: $QuickSort(A, q+1, r)$
5: **end if**

2 1 7 6 4 3 9 8 5


2 1 4 3
5
6 9 8 7


1
2
3
4
5
6
7
8
9

Complessità pessima di QuickSort

Nel caso pessimo la procedura $Partition(A, p, r)$ ritorna in tempo $\Theta(n)$ un estremo (p o r), cioè il vettore $A[p, \ldots, r]$ viene suddiviso in un sottovettore di dimensione $n-1$ e uno di dimensione 0, ove $n = r - p + 1$.

Dunque la complessità pessima di $QuickSort$ è espressa dalla seguente equazione ricorsiva:

$$\begin{aligned} C(0) &= \Theta(1) \\ C(n) &= C(n-1) + C(0) + \Theta(n) = C(n-1) + \Theta(n) \end{aligned}$$

per $n > 0$. Dunque

$$C(n) = \Theta(n^2)$$

Complessità ottima di QuickSort

Nel caso ottimo la procedura $Partition(A, p, r)$ ritorna in tempo $\Theta(n)$ l'indice della **mediana** del vettore $A[p, \ldots, r]$. Dunque il vettore $A[p, \ldots, r]$ viene suddiviso in due parti di dimensione circa $n/2$, ove $n = r - p + 1$.

Dunque la complessità ottima di *QuickSort* è espressa dalla seguente equazione ricorsiva:

$$\begin{aligned} C(0) &= \Theta(1) \\ C(n) &= 2C(n/2) + \Theta(n) \end{aligned}$$

per $n > 0$. Dunque

$$C(n) = \Theta(n \log n)$$

Complessità media di QuickSort

Nel caso medio la procedura $Partition(A, p, r)$ ritorna in tempo $\Theta(n)$ l'indice della **mediana** del vettore $A[p, \ldots, r]$.

Infatti, scegliendo a caso un elemento x in un vettore, in media, metà degli elementi del vettore sarà minore di x e l'altra metà sarà maggiore di x.

Dunque il caso medio equivale al caso ottimo, e la complessità media di $QuickSort$ è $\Theta(n \log n)$.

Velocizziamo QuickSort

Si supponga di avere una procedura $Median(A, p, r)$ *di complessità lineare che restituisce l'indice della mediana in* $A[p, r]$*. Su usi questa procedura per implementare una versione di QuickSort con complessità pessima* $O(n \log n)$*.*

QuickSort(A,p,r)

1: **if** $p < r$ **then**
2: $i \leftarrow Median(A, p, r)$
3: $Exchange(A, i, r)$
4: $q \leftarrow Partition(A, p, r)$
5: $QuickSort(A, p, q - 1)$
6: $QuickSort(A, q + 1, r)$
7: **end if**

MergeSort

MergeSort è un algoritmo di ordinamento **ricorsivo**, **per confronto**, e **non sul posto**.

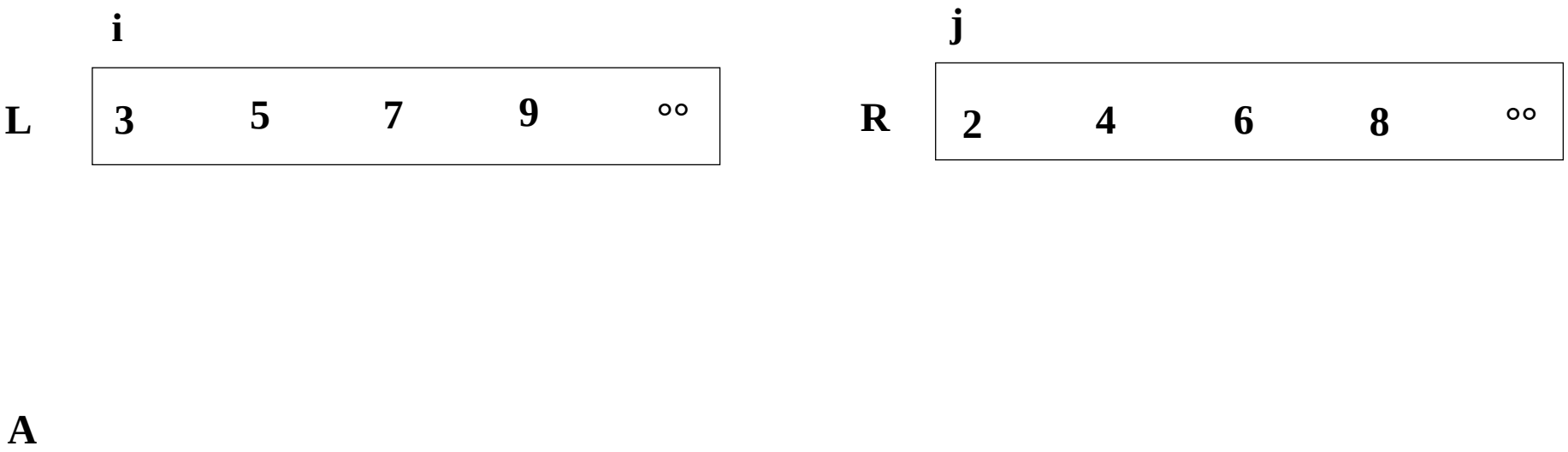
$MergeSort$ usa una procedura ausiliaria $Merge(A, p, q, r)$ che presuppone che i sottovettori $A[p, \ldots, q]$ e $A[q+1, \ldots, r]$ siamo ordinati (in senso crescente).

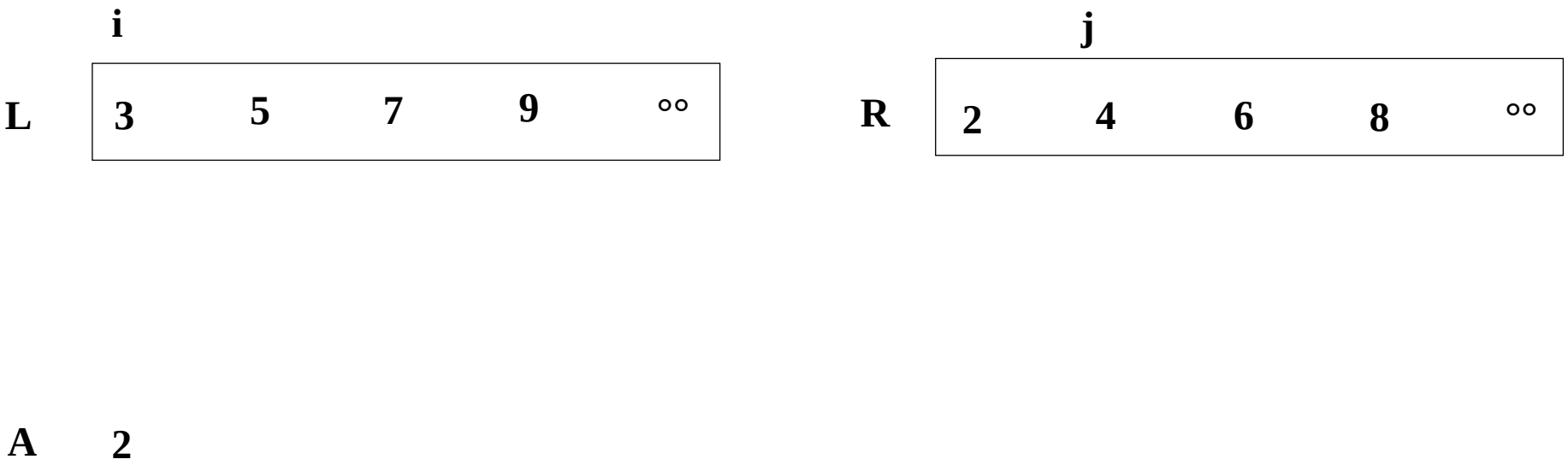
Il compito di $Merge(A, p, q, r)$ è quello di odinare il vettore $A[p, \ldots, r]$.

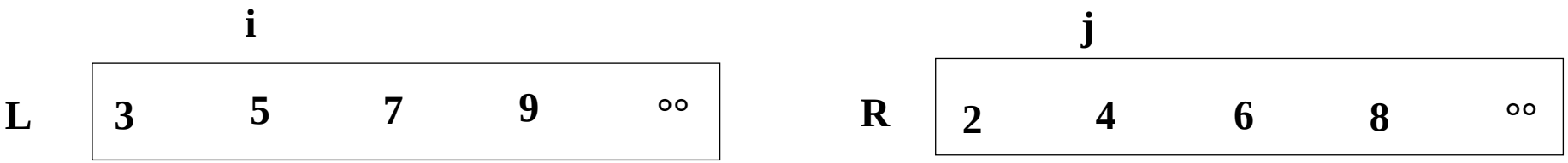
1: **Merge**$(\mathbf{A}, \mathbf{p}, \mathbf{q}, \mathbf{r})$
2: $n_1 \leftarrow q - p + 1$
3: $n_2 \leftarrow r - q$
4: **for** $i \leftarrow 1$ **to** n_1 **do**
5: $\quad L[i] \leftarrow A[p + i - 1]$
6: **end for**
7: **for** $j \leftarrow 1$ **to** n_2 **do**
8: $\quad R[j] \leftarrow A[q + j]$
9: **end for**

10: $L[n_1 + 1] \leftarrow \infty$
11: $R[n_2 + 1] \leftarrow \infty$
12: $i \leftarrow 1$
13: $j \leftarrow 1$
14: **for** $k \leftarrow p$ **to** r **do**
15: **if** $L[i] \leq R[j]$ **then**
16: $A[k] \leftarrow L[i]$
17: $i \leftarrow i + 1$
18: **else**
19: $A[k] \leftarrow R[j]$
20: $j \leftarrow j + 1$
21: **end if**
22: **end for**

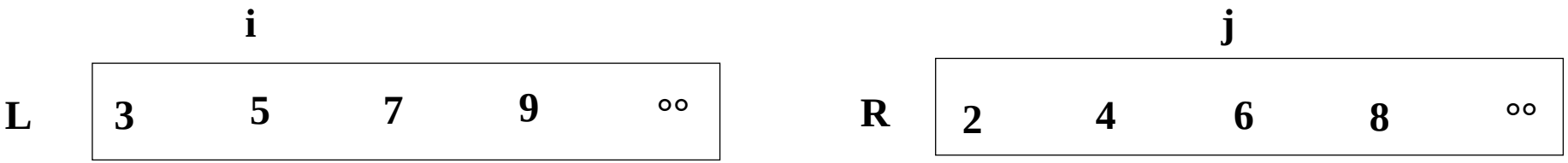
La complessità di $Merge(A, p, q, r)$ è $\Theta(n)$, over $n = r - p + 1$ è la dimensione del sottovettore $A[p, \ldots, r]$.


i
j
L
3
5
7
9
∞∞
R
2
4
6
8
∞∞
A

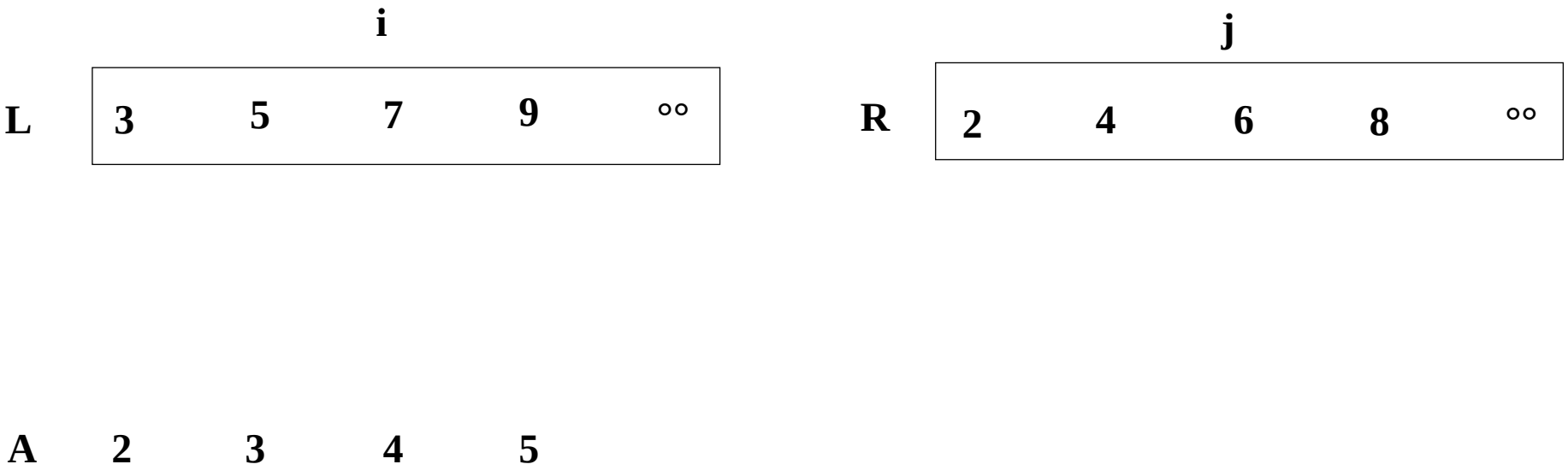

i
j
L
3
5
7
9
∞∞
R
2
4
6
8
∞∞
A
2

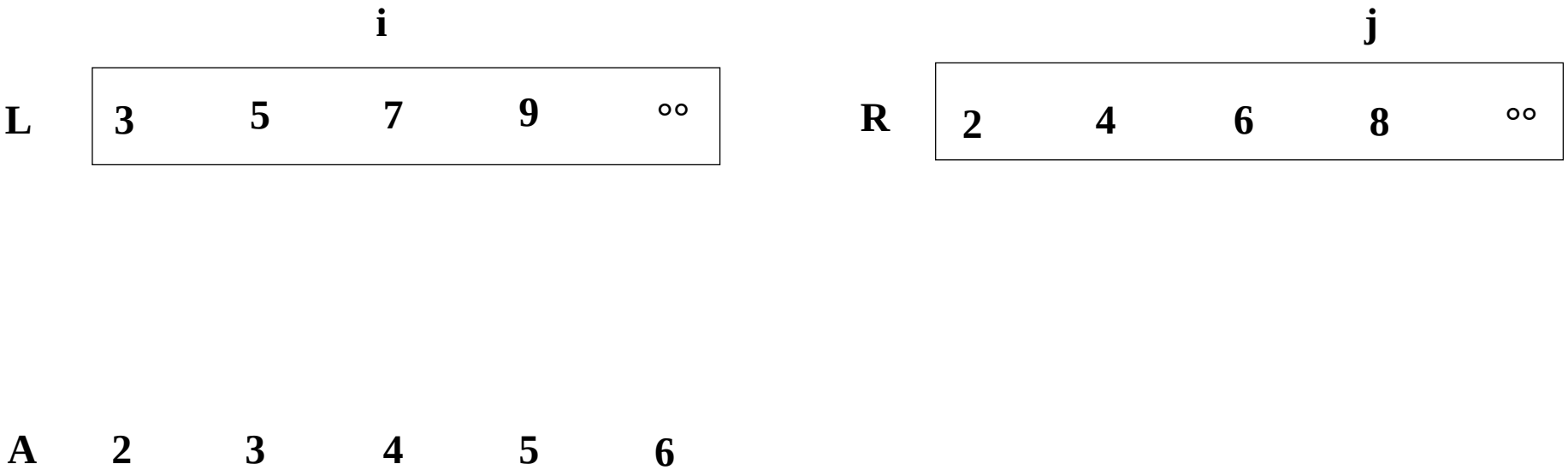


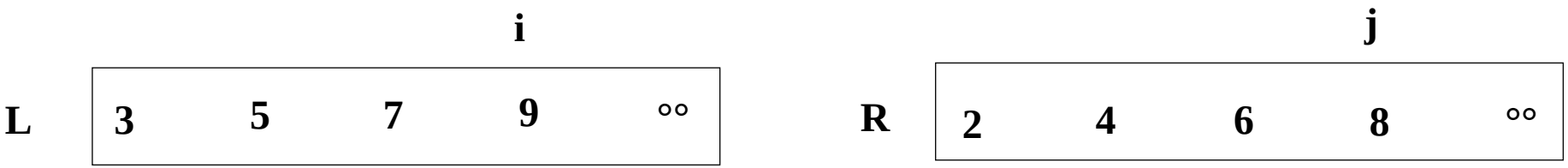
A **2** **3**



A	2	3	4


i
j
L
3
5
7
9
∞∞
R
2
4
6
8
∞∞
A
2
3
4
5


i
j
L
3
5
7
9
∞∞
R
2
4
6
8
∞∞
A
2
3
4
5
6


i
j
L
3
5
7
9
∞∞
R
2
4
6
8
∞∞


A
2
3
4
5
6
7


i
j
L
3
5
7
9
∞∞
R
2
4
6
8
∞∞


A
2
3
4
5
6
7
8

A **2** **3** **4** **5** **6** **7** **8** **9**

MergeSort

MergeSort(A, p, r)

1: **if** $p < r$ **then**
2: $q \leftarrow (p + r)\ div\ 2$
3: $MergeSort(A, p, q)$
4: $MergeSort(A, q + 1, r)$
5: $Merge(A, p, q, r)$
6: **end if**

Complessità di MergeSort

La complessità di *MergeSort* è espressa dalla seguente equazione ricorsiva:

$$\begin{aligned} C(0) &= \Theta(1) \\ C(n) &= 2C(n/2) + \Theta(n) \end{aligned}$$

per $n > 0$. Dunque

$$C(n) = \Theta(n \log n)$$

MergeSort non distingue tra caso ottimo, medio e pessimo.

Confronto tra gli algoritmi per confronto

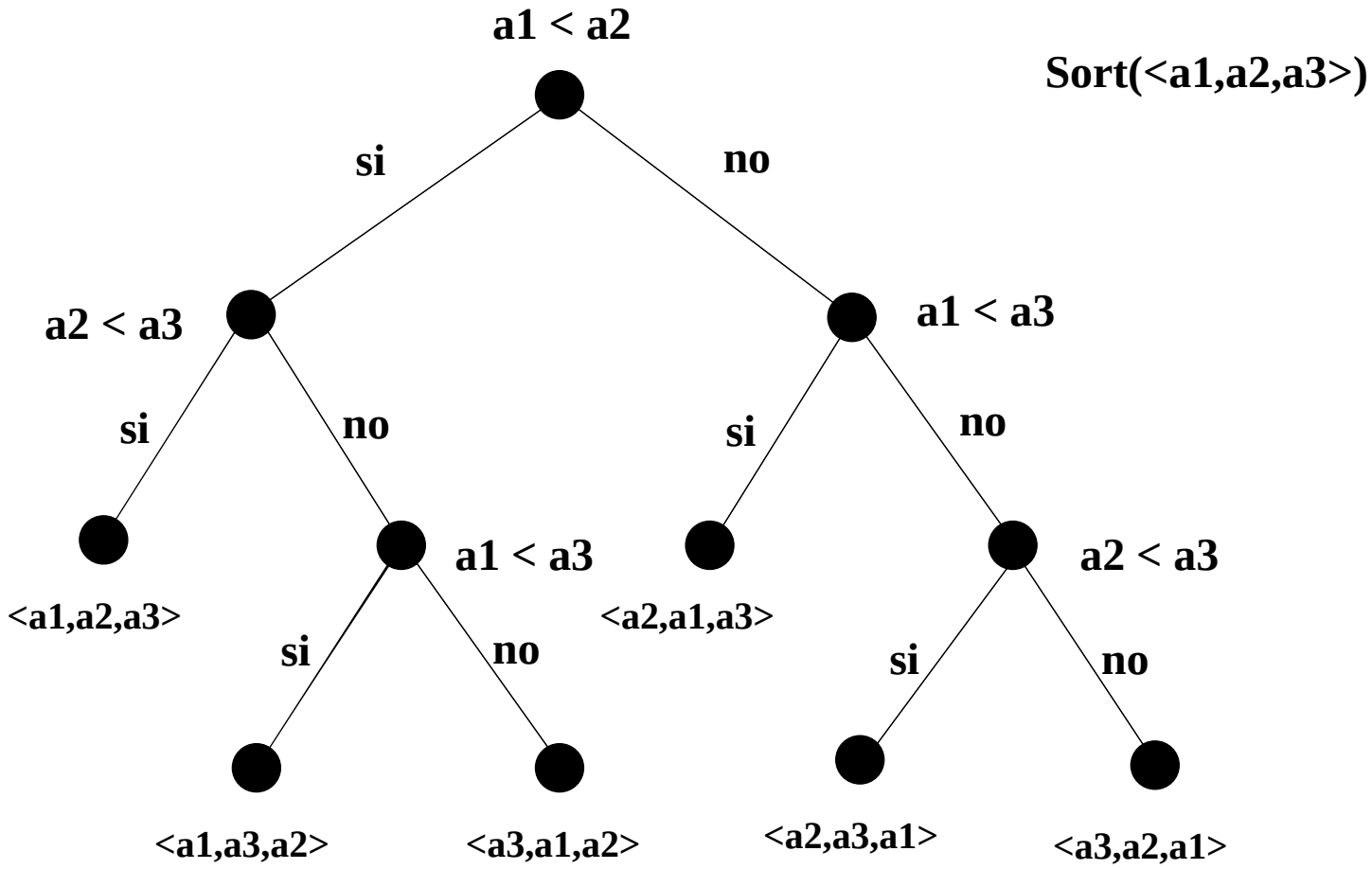
Algoritmo	Ottima	Media	Pessima
InsertionSort	n	n^2	n^2
TreeSort	$n \log n$	$n \log n$	n^2
HeapSort	$n \log n$	$n \log n$	$n \log n$
QuickSort	$n \log n$	$n \log n$	n^2
MergeSort	$n \log n$	$n \log n$	$n \log n$

Complessità dell'ordinamento per confronto

Teorema Sia *Sort* un qualsiasi algoritmo di ordinamento per confronto di una sequenza di n elementi. Allora:

- La **complessità pessima** di *Sort* è $\Omega(n \log n)$;
- la **complessità media** di *Sort* è $\Omega(n \log n)$;
- La **complessità ottima** di *Sort* è $\Omega(n)$.

Dimostrazione Dimostriamo il teorema per il caso pessimo. Tutte le possibile esecuzioni dell'algoritmo generico *Sort* possono essere rappresentate da un albero binario pieno chiamato **albero di decisione**.


a1 < a2
Sort(<a1,a2,a3>)
si
no
a2 < a3
a1 < a3
si
no
si
no
<a1,a2,a3>
a1 < a3
<a2,a1,a3>
a2 < a3
si
no
si
no
<a1,a3,a2>
<a3,a1,a2>
<a2,a3,a1>
<a3,a2,a1>

Una esecuzione dell'algoritmo generico *Sort* corrisponde ad un cammino dalla radice dell'albero di decisione ad una foglia.

La complessità di una esecuzione dell'algoritmo corrisponde al numero di confronti fatti prima di fornire il risultato. Essa corrisponde alla **lunghezza del cammino** associato all'esecuzione.

La **complessità pessima** è dunque la lunghezza del cammino più lungo dalla radice a una foglia, cioè l'**altezza** h dell'albero di decisione.

Il numero di foglie dell'albero di decisione è $n!$. L'albero di decisione è un albero binario pieno. Il massimo numero di foglie di un albero binario pieno di altezza h è 2^h. Quindi deve valere che

$$n! \leq 2^h$$

cioè

$$h \geq \log(n!) = \Theta(n \log n)$$

In particolare, vale che

$$h = \Omega(n \log n)$$

CountingSort

CountingSort è un algoritmo di ordinamento che **non si basa sui confronti**. Sotto opportune ipotesi, l'algoritmo *CountingSort* ha **complessità pessima lineare** $\Theta(n)$.

CountingSort si basa sulla seguente osservazione:

> *Sia x un elemento da ordinare e sia i il numero di elementi da ordinare minori o uguali a x (includendo x stesso nel conto). Allora, nella sequenza ordinata, l'elemento x occuperà la posizione i.*

CountingSort assume che gli elementi da ordinare siano numeri interi compresi nell'intervallo $[0, k]$, cioè per ogni indice i del vettore A da ordinare vale che

$$0 \leq A[i] \leq k$$

CountingSort(A, k)

```
1: for x ← 0 to k do
2:     C[x] ← 0
3: end for
4: for i ← 1 to length(A) do
5:     x ← A[i]
6:     C[x] ← C[x] + 1
7: end for
8: for x ← 1 to k do
9:     C[x] ← C[x − 1] + C[x]
10: end for
11: for i ← length(A) downto 1 do
12:     x ← A[i]
13:     B[C[x]] ← x
14:     C[x] ← C[x] − 1
15: end for
16: for i ← 1 to length(A) do
17:     A[i] ← B[i]
18: end for
```

Correttezza di CountingSort

- Dopo il primo ciclo for l'elemento $C[x]$ contiene zero;
- dopo il secondo ciclo for l'elemento $C[x]$ contiene il numero di elementi di A che sono **uguali** a x;
- dopo il terzo ciclo for l'elemento $C[x]$ contiene il numero di elementi di A che sono **minori o uguali** a x;
- a questo punto $C[x]$ è la posizione di x nel vettore ordinato.

```
     1 2 3 4 5
A = [4,1,4,2,1]   n = 5   k = 4

     0 1 2 3 4
C = [0,0,0,0,0]

     0 1 2 3 4
C = [0,2,1,0,2]

     0 1 2 3 4
C = [0,2,3,3,5]

     1 2 3 4 5                1 2 3 4 5
B = [1,1,2,4,4]          A = [1,1,2,4,4]
```

Complessità di CountingSort

Sia $n = length(A)$. La complessità di $CountingSort(A, k)$ è

$$\Theta(3n + 2k) = \Theta(n + k)$$

Qualora

$$k = O(n)$$

la complessità è **lineare** $\Theta(n)$.