Lezione 7

Sommario della Lezione

Sommario della Lezione

- Metodologie per il progetto di algoritmi: La Tecnica Divide et Impera

Sommario della Lezione

- Metodologie per il progetto di algoritmi: La Tecnica Divide et Impera
- Altri esempi

Paradigma generale di algoritmi basati su D&I

Paradigma generale di algoritmi basati su D&I

Gli algoritmi basati sulla tecnica Divide-et-Impera hanno, generalmente, la seguente struttura ricorsiva:

```
Algoritmo D&I(x)
    IF (l'input x é sufficientemente piccolo o semplice) {
        RETURN adhoc(x) (% ovvero risolvi x direttamente)
        }
```

Paradigma generale di algoritmi basati su D&I

Gli algoritmi basati sulla tecnica Divide-et-Impera hanno, generalmente, la seguente struttura ricorsiva:

```
Algoritmo D&I(x)
    IF (l'input x é sufficientemente piccolo o semplice) {
        RETURN adhoc(x) (% ovvero risolvi x direttamente)
        }  ELSE { decomponi l'istanza di input x in k istanze piú
piccole x_1, x_2, ..., x_k
                }
```

Paradigma generale di algoritmi basati su D&I

Gli algoritmi basati sulla tecnica Divide-et-Impera hanno, generalmente, la seguente struttura ricorsiva:

Algoritmo D&I(x)
    IF (l'input x é sufficientemente piccolo o semplice) {
        RETURN *adhoc*(x) (% ovvero risolvi x direttamente)
        } ELSE { decomponi l'istanza di input x in k istanze piú piccole $x_1, x_2, \ldots, x_k$
                }
  s1= Algoritmo D&I(x_1)

Paradigma generale di algoritmi basati su D&I

Gli algoritmi basati sulla tecnica Divide-et-Impera hanno, generalmente, la seguente struttura ricorsiva:

```
Algoritmo D&I(x)
     IF (l'input x é sufficientemente piccolo o semplice) {
          RETURN adhoc(x) (% ovvero risolvi x direttamente)
          } ELSE { decomponi l'istanza di input x in k istanze piú
piccole x1, x2, ..., xk
                    }
  s1= Algoritmo D&I(x1)
  s2= Algoritmo D&I(x2)
```

Paradigma generale di algoritmi basati su D&I

Gli algoritmi basati sulla tecnica Divide-et-Impera hanno, generalmente, la seguente struttura ricorsiva:

Algoritmo D&I(x)
    IF (l'input x é sufficientemente piccolo o semplice) {
        RETURN *adhoc*(x) (% ovvero risolvi x direttamente)
        } ELSE { decomponi l'istanza di input x in k istanze piú piccole $x_1, x_2, \dots, x_k$
                }
  s1= Algoritmo D&I(x_1)
  s2= Algoritmo D&I(x_2)
    $\vdots$
  sk= Algoritmo D&I(x_k)

Paradigma generale di algoritmi basati su D&I

Gli algoritmi basati sulla tecnica Divide-et-Impera hanno, generalmente, la seguente struttura ricorsiva:

```
Algoritmo D&I(x)
    IF (l'input x é sufficientemente piccolo o semplice) {
        RETURN adhoc(x) (% ovvero risolvi x direttamente)
        } ELSE { decomponi l'istanza di input x in k istanze piú
piccole x_1, x_2, ..., x_k
                }
  s1= Algoritmo D&I(x_1)
  s2= Algoritmo D&I(x_2)
    ⋮
  sk= Algoritmo D&I(x_k)
componi le sottosoluzioni s1, s2, ..., sk alle istanze x_i
per ottenere una soluzione globale s alla istanza completa
x
```

Paradigma generale di algoritmi basati su D&I

Gli algoritmi basati sulla tecnica Divide-et-Impera hanno, generalmente, la seguente struttura ricorsiva:

```
Algoritmo D&I(x)
    IF (l'input x é sufficientemente piccolo o semplice) {
        RETURN adhoc(x) (% ovvero risolvi x direttamente)
        } ELSE { decomponi l'istanza di input x in k istanze piú
piccole x_1, x_2, ..., x_k
                }
  s1= Algoritmo D&I(x_1)
  s2= Algoritmo D&I(x_2)
    ⋮
  sk= Algoritmo D&I(x_k)
componi le sottosoluzioni s1, s2, ..., sk alle istanze x_i
per ottenere una soluzione globale s alla istanza completa
x
RETURN(s)
```

Primo esempio: Ordinamento

Primo esempio: ORDINAMENTO

Applichiamo la tecnica Divide-et-Impera al problema dell'ordinamento, così definito:

Input: sequenza a=a[0]a[1]...a[n-1] di numeri
Output: una permutazione $j_0, j_1, \ldots, j_{n-1}$ degli indici 0,1, ... , n-1 tale che a[j_0]$\leq$ a[j_1]$\leq \ldots \leq$ a[j_{n-1}]

Primo esempio: Ordinamento

Applichiamo la tecnica Divide-et-Impera al problema dell'ordinamento, così definito:

Input: sequenza `a=a[0]a[1]...a[n-1]` di numeri
Output: una permutazione $j_0, j_1, \ldots, j_{n-1}$ degli indici `0,1, ... , n-1` tale che `a[`j_0`]`$\leq$ `a[`j_1`]`$\leq \ldots \leq$ `a[`j_{n-1}`]`

Adottando la tecnica Divide et Impera procederemo nel modo seguente:

MergeSort(A)
1. Se $n = 1$, restituisci A.

Primo esempio: ORDINAMENTO

Applichiamo la tecnica Divide-et-Impera al problema dell'ordinamento, così definito:

Input: sequenza `a=a[0]a[1]...a[n-1]` di numeri
Output: una permutazione $j_0, j_1, \ldots, j_{n-1}$ degli indici `0,1, ... , n-1` tale che `a[`j_0`]`$\leq$ `a[`j_1`]`$\leq \ldots \leq$ `a[`j_{n-1}`]`

Adottando la tecnica Divide et Impera procederemo nel modo seguente:

MERGESORT(A)
1. Se $n = 1$, restituisci A.
2. Altrimenti, dividi A in due metà uguali X e Y.

Primo esempio: ORDINAMENTO

Applichiamo la tecnica Divide-et-Impera al problema dell'ordinamento, così definito:

Input: sequenza `a=a[0]a[1]...a[n-1]` di numeri
Output: una permutazione $j_0, j_1, \ldots, j_{n-1}$ degli indici `0,1, ... , n-1` tale che `a`[j_0]$\leq$ `a`[j_1]$\leq \ldots \leq$ `a`[j_{n-1}]

Adottando la tecnica Divide et Impera procederemo nel modo seguente:

MERGESORT(A)
1. Se $n = 1$, restituisci A.
2. Altrimenti, dividi A in due metà uguali X e Y.
3. Ordina X e Y, chiamando ricorsivamente MERGESORT(X) e MERGESORT(Y)

Primo esempio: ORDINAMENTO

Applichiamo la tecnica Divide-et-Impera al problema dell'ordinamento, così definito:

Input: sequenza `a=a[0]a[1]...a[n-1]` di numeri
Output: una permutazione $j_0, j_1, \ldots, j_{n-1}$ degli indici `0,1, ... , n-1` tale che `a`[j_0]$\leq$ `a`[j_1]$\leq \ldots \leq$ `a`[j_{n-1}]

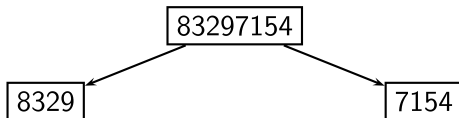
Adottando la tecnica Divide et Impera procederemo nel modo seguente:

MERGESORT(A)
1. Se $n = 1$, restituisci A.
2. Altrimenti, dividi A in due metà uguali X e Y.
3. Ordina X e Y, chiamando ricorsivamente MERGESORT(X) e MERGESORT(Y)
4. Combina i due array ottenuti al passo 3. in un unico array ordinato e restituiscilo

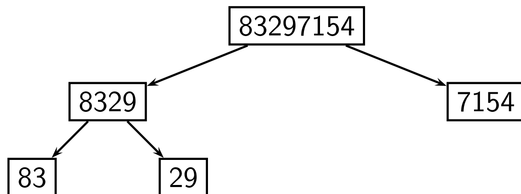
Esempio di esecuzione di MergeSort

83297154

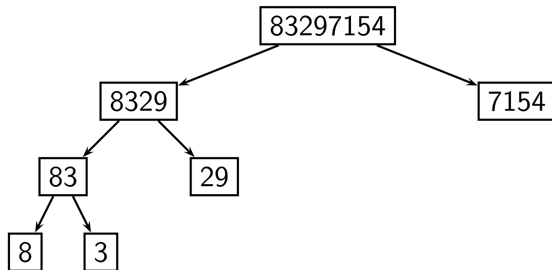
Esempio di esecuzione di MergeSort



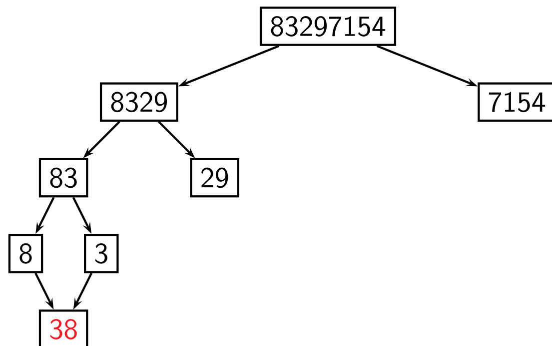
Esempio di esecuzione di MergeSort



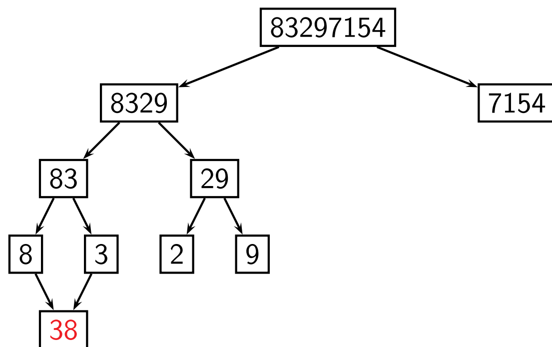
Esempio di esecuzione di MergeSort



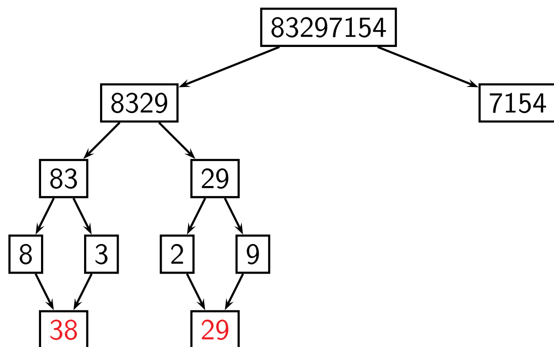
Esempio di esecuzione di MergeSort



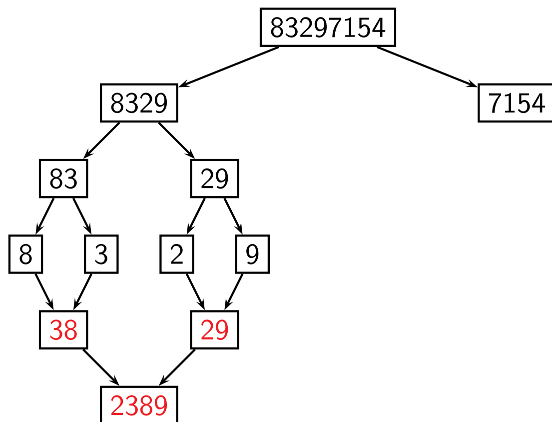
Esempio di esecuzione di MergeSort



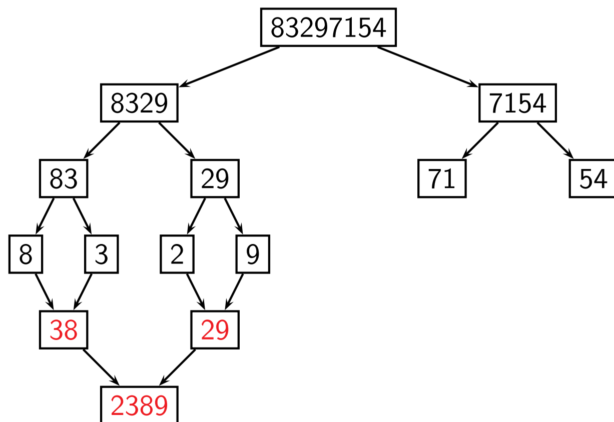
Esempio di esecuzione di MergeSort



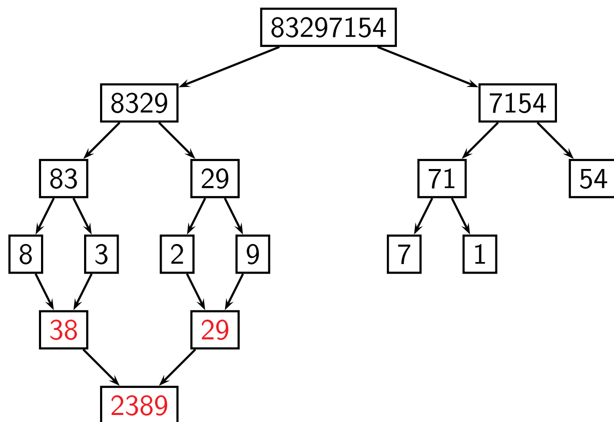
Esempio di esecuzione di MergeSort



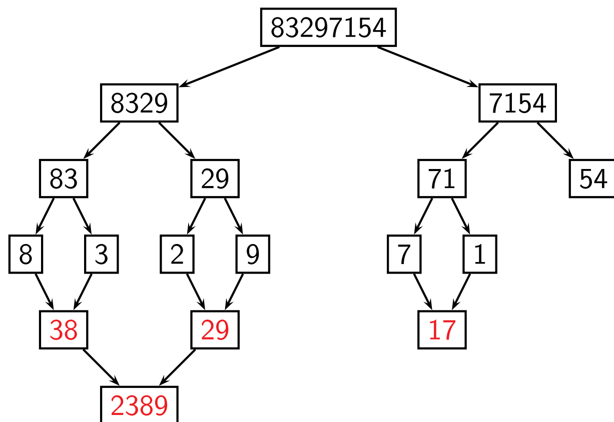
Esempio di esecuzione di MergeSort



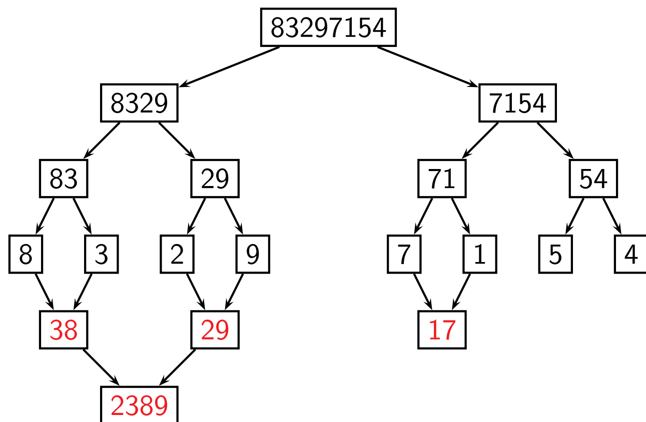
Esempio di esecuzione di MergeSort



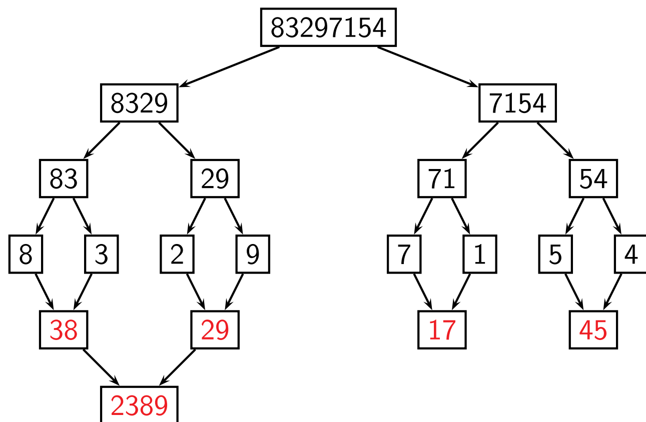
Esempio di esecuzione di MergeSort



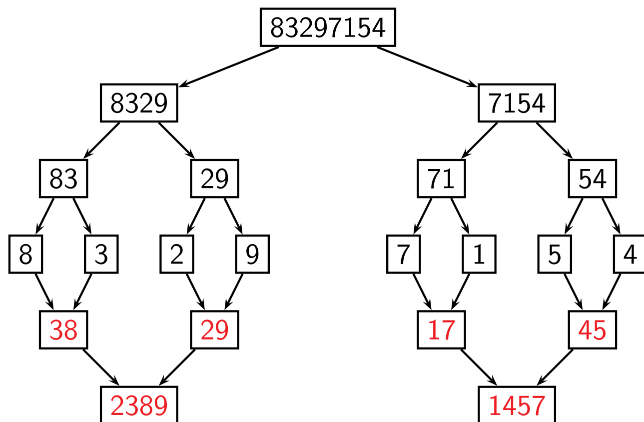
Esempio di esecuzione di MergeSort



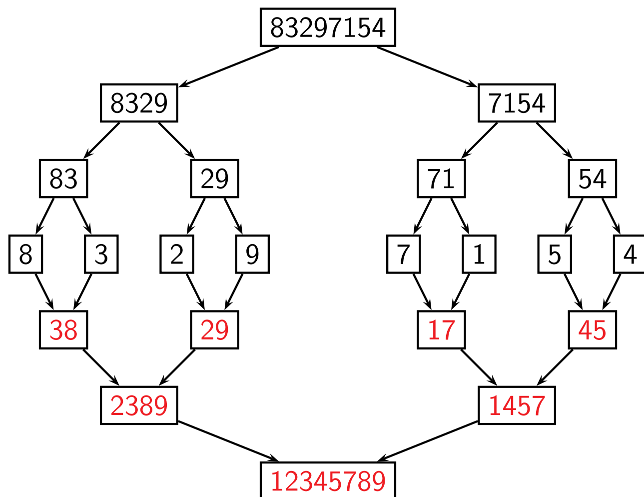
Esempio di esecuzione di MergeSort



Esempio di esecuzione di MergeSort



Esempio di esecuzione di MergeSort



Formalmente, l'algoritmo potrebbe essere di questo tipo:

```
MergeSort(a,i,j)   % 0≤ i ≤ j≤ n-1
```

Formalmente, l'algoritmo potrebbe essere di questo tipo:

```
MergeSort(a,i,j)   % 0≤ i ≤ j≤ n-1
1. IF(i<j) {
2. c=(i+j)/2
```

Formalmente, l'algoritmo potrebbe essere di questo tipo:

```
MergeSort(a,i,j)   % 0≤ i ≤ j≤ n-1
1. IF(i<j) {
2. c=(i+j)/2
3. MergeSort(a,i,c)
```

Formalmente, l'algoritmo potrebbe essere di questo tipo:

```
MergeSort(a,i,j)   % 0≤ i ≤ j≤ n-1
1. IF(i<j) {
2. c=(i+j)/2
3. MergeSort(a,i,c)
4. MergeSort(a,c+1, j)
```

Formalmente, l'algoritmo potrebbe essere di questo tipo:

```
MergeSort(a,i,j)   % 0≤ i ≤ j≤ n-1
1. IF(i<j) {
2. c=(i+j)/2
3. MergeSort(a,i,c)
4. MergeSort(a,c+1, j)
5. MERGE(a,i,c,j)
}
```

Formalmente, l'algoritmo potrebbe essere di questo tipo:

```
MergeSort(a,i,j)   % 0≤ i ≤ j≤ n-1
1. IF(i<j) {
2. c=(i+j)/2
3. MergeSort(a,i,c)
4. MergeSort(a,c+1, j)
5. MERGE(a,i,c,j)
}
```

dove l'algoritmo `MERGE(a,i,c,j)` prende in input le due sottosequenze `a[i]...a[c]` e `a[c+1]...a[j]` *ordinate*

Formalmente, l'algoritmo potrebbe essere di questo tipo:

```
MergeSort(a,i,j)   % 0≤ i ≤ j≤ n-1
1. IF(i<j) {
2. c=(i+j)/2
3. MergeSort(a,i,c)
4. MergeSort(a,c+1, j)
5. MERGE(a,i,c,j)
}
```

dove l'algoritmo `MERGE(a,i,c,j)` prende in input le due sottosequenze `a[i]...a[c]` e `a[c+1]...a[j]` *ordinate* e restituisce un'unica sequenza ordinata contenente tutto gli elementi delle due sottosequenza in input.

L'algoritmo `MERGE(a,i,c,j)` può essere descritto dal seguente pseudocodice, in cui si usa un array ausiliario `b` per memorizzare computazioni intermedie:

```
Merge(a,i,c,j)
```

L'algoritmo `MERGE(a,i,c,j)` può essere descritto dal seguente pseudocodice, in cui si usa un array ausiliario `b` per memorizzare computazioni intermedie:

```
Merge(a,i,c,j)
s=i, d=c+1, k=0
```

L'algoritmo MERGE(a,i,c,j) può essere descritto dal seguente pseudocodice, in cui si usa un array ausiliario b per memorizzare computazioni intermedie:

```
Merge(a,i,c,j)
s=i, d=c+1, k=0
WHILE ((s<=c)&&(d<=j)) {
```

L'algoritmo MERGE(a,i,c,j) può essere descritto dal seguente pseudocodice, in cui si usa un array ausiliario b per memorizzare computazioni intermedie:

```
Merge(a,i,c,j)
s=i, d=c+1, k=0
WHILE ((s<=c)&&(d<=j)) {
    IF (a[s]<=a[d]) {
```

L'algoritmo `MERGE(a,i,c,j)` può essere descritto dal seguente pseudocodice, in cui si usa un array ausiliario `b` per memorizzare computazioni intermedie:

```
Merge(a,i,c,j)
s=i, d=c+1, k=0
WHILE ((s<=c)&&(d<=j)) {
    IF (a[s]<=a[d]) {
      b[k]=a[s],
```

L'algoritmo `MERGE(a,i,c,j)` può essere descritto dal seguente pseudocodice, in cui si usa un array ausiliario `b` per memorizzare computazioni intermedie:

```
Merge(a,i,c,j)
s=i, d=c+1, k=0
WHILE ((s<=c)&&(d<=j)) {
    IF (a[s]<=a[d]) {
      b[k]=a[s], s=s+1
```

L'algoritmo MERGE(a,i,c,j) può essere descritto dal seguente pseudocodice, in cui si usa un array ausiliario b per memorizzare computazioni intermedie:

```
Merge(a,i,c,j)
s=i, d=c+1, k=0
WHILE ((s<=c)&&(d<=j)) {
    IF (a[s]<=a[d]) {
      b[k]=a[s], s=s+1
    } ELSE {
      b[k]=a[d],
```

L'algoritmo `MERGE(a,i,c,j)` può essere descritto dal seguente pseudocodice, in cui si usa un array ausiliario `b` per memorizzare computazioni intermedie:

```
Merge(a,i,c,j)
s=i, d=c+1, k=0
WHILE ((s<=c)&&(d<=j)) {
    IF (a[s]<=a[d]) {
      b[k]=a[s], s=s+1
    } ELSE {
      b[k]=a[d], d=d+1 }
```

L'algoritmo `MERGE(a,i,c,j)` può essere descritto dal seguente pseudocodice, in cui si usa un array ausiliario `b` per memorizzare computazioni intermedie:

```
Merge(a,i,c,j)
s=i, d=c+1, k=0
WHILE ((s<=c)&&(d<=j)) {
    IF (a[s]<=a[d]) {
      b[k]=a[s], s=s+1
    } ELSE {
      b[k]=a[d], d=d+1 }
    k=k+1 }
```

L'algoritmo MERGE(a,i,c,j) può essere descritto dal seguente pseudocodice, in cui si usa un array ausiliario b per memorizzare computazioni intermedie:

```
Merge(a,i,c,j)
s=i, d=c+1, k=0
WHILE ((s<=c)&&(d<=j)) {
    IF (a[s]<=a[d]) {
      b[k]=a[s], s=s+1
    } ELSE {
      b[k]=a[d], d=d+1 }
    k=k+1 }
FOR ( ; s<=c; s=s+1; k=k+1)
    b[k]=a[s]
```

L'algoritmo `MERGE(a,i,c,j)` può essere descritto dal seguente pseudocodice, in cui si usa un array ausiliario `b` per memorizzare computazioni intermedie:

```
Merge(a,i,c,j)
s=i, d=c+1, k=0
WHILE ((s<=c)&&(d<=j)) {
    IF (a[s]<=a[d]) {
      b[k]=a[s], s=s+1
    } ELSE {
      b[k]=a[d], d=d+1 }
    k=k+1 }
FOR ( ; s<=c; s=s+1; k=k+1)
    b[k]=a[s]
FOR ( ; d<=j; d=d+1; k=k+1)
    b[k]=a[d]
```

L'algoritmo `MERGE(a,i,c,j)` può essere descritto dal seguente pseudocodice, in cui si usa un array ausiliario `b` per memorizzare computazioni intermedie:

```
Merge(a,i,c,j)
s=i, d=c+1, k=0
WHILE ((s<=c)&&(d<=j)) {
    IF (a[s]<=a[d]) {
      b[k]=a[s], s=s+1
    } ELSE {
      b[k]=a[d], d=d+1 }
    k=k+1 }
FOR ( ; s<=c; s=s+1; k=k+1)
    b[k]=a[s]
FOR ( ; d<=j; d=d+1; k=k+1)
    b[k]=a[d]
FOR (s=i;s<=j; s=s+1)
    a[s]=b[s-i]
```

L'algoritmo `Merge(a,0,c,n-1)` richiede tempo dn, per qualche costante d

L'algoritmo `MERGE(a,i,c,j)` può essere descritto dal seguente pseudocodice, in cui si usa un array ausiliario `b` per memorizzare computazioni intermedie:

```
Merge(a,i,c,j)
s=i, d=c+1, k=0
WHILE ((s<=c)&&(d<=j)) {
    IF (a[s]<=a[d]) {
      b[k]=a[s], s=s+1
    } ELSE {
      b[k]=a[d], d=d+1 }
    k=k+1 }
FOR ( ; s<=c; s=s+1; k=k+1)
    b[k]=a[s]
FOR ( ; d<=j; d=d+1; k=k+1)
    b[k]=a[d]
FOR (s=i;s<=j; s=s+1)
    a[s]=b[s-i]
```

L'algoritmo `Merge(a,0,c,n-1)` richiede tempo dn, per qualche costante d (e potremmo anche fare a meno dell'array ausiliario `b`).

Analisi di Merge Sort

Analisi di Merge Sort

```
MergeSort(a,i,j)   % 0≤ i ≤ j≤ n-1
1. IF(i<j) {
2. c=(i+j)/2
3. MergeSort(a,i,c)
4. MergeSort(a,c+1, j)
5. MERGE(a,i,c,j)
}
```

Analisi di Merge Sort

```
MergeSort(a,i,j)   % 0≤ i ≤ j≤ n-1
1. IF(i<j) {
2. c=(i+j)/2
3. MergeSort(a,i,c)
4. MergeSort(a,c+1, j)
5. MERGE(a,i,c,j)
}
```

Analisi:

$$T(1) = \Theta(1)$$

Analisi di Merge Sort

```
MergeSort(a,i,j)   % 0≤ i ≤ j≤ n-1
1. IF(i<j) {
2. c=(i+j)/2
3. MergeSort(a,i,c)
4. MergeSort(a,c+1, j)
5. MERGE(a,i,c,j)
}
```

Analisi:

$$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= \end{aligned}$$

Analisi di MERGE SORT

```
MergeSort(a,i,j)   % 0≤ i ≤ j≤ n-1
1. IF(i<j) {
2. c=(i+j)/2
3. MergeSort(a,i,c)
4. MergeSort(a,c+1, j)
5. MERGE(a,i,c,j)
}
```

Analisi:

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

Analisi di MERGE SORT

```
MergeSort(a,i,j)   % 0≤ i ≤ j≤ n-1
1. IF(i<j) {
2. c=(i+j)/2
3. MergeSort(a,i,c)
4. MergeSort(a,c+1, j)
5. MERGE(a,i,c,j)
}
```

Analisi:

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

Analisi di MERGE SORT

```
MergeSort(a,i,j)   % 0≤ i ≤ j≤ n-1
1. IF(i<j) {
2. c=(i+j)/2
3. MergeSort(a,i,c)
4. MergeSort(a,c+1, j)
5. MERGE(a,i,c,j)
}
```

Analisi:

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

da cui $\boxed{T(n) = \Theta(n \log n)}$

Si può fare meglio?

Si può fare meglio?

No.

Vediamo il perchè

Vediamo il perchè

Sia $\mathcal{A}$ un **qualsivoglia** algoritmo di ordinamento, capace di ordinare arbitrarie sequenze `a=a[0]...a[n-1]` mediante confronti tra gli elementi di `a`.

Vediamo il perchè

Sia $\mathcal{A}$ un **qualsivoglia** algoritmo di ordinamento, capace di ordinare arbitrarie sequenze `a=a[0]...a[n-1]` mediante confronti tra gli elementi di `a`.

Il numero di **possibili** ordinamenti di `a=a[0]...a[n-1]` è pari a $n!$

Vediamo il perchè

Sia $\mathcal{A}$ un **qualsivoglia** algoritmo di ordinamento, capace di ordinare arbitrarie sequenze `a=a[0]...a[n-1]` mediante confronti tra gli elementi di `a`.

Il numero di **possibili** ordinamenti di `a=a[0]...a[n-1]` è pari a $n!$

(ad esempio, se $a = a[0]a[1]a[2]$, potremmo avere che
$a[0]$

Vediamo il perchè

Sia $\mathcal{A}$ un **qualsivoglia** algoritmo di ordinamento, capace di ordinare arbitrarie sequenze `a=a[0]...a[n-1]` mediante confronti tra gli elementi di `a`.

Il numero di **possibili** ordinamenti di `a=a[0]...a[n-1]` è pari a n!

(ad esempio, se $a = a[0]a[1]a[2]$, potremmo avere che
$a[0] \leq a[1]$

Vediamo il perchè

Sia $\mathcal{A}$ un **qualsivoglia** algoritmo di ordinamento, capace di ordinare arbitrarie sequenze `a=a[0]...a[n-1]` mediante confronti tra gli elementi di `a`.

Il numero di **possibili** ordinamenti di `a=a[0]...a[n-1]` è pari a $n!$

(ad esempio, se $a = a[0]a[1]a[2]$, potremmo avere che $a[0] \leq a[1] \leq a[2]$, oppure

Vediamo il perchè

Sia $\mathcal{A}$ un **qualsivoglia** algoritmo di ordinamento, capace di ordinare arbitrarie sequenze `a=a[0]...a[n-1]` mediante confronti tra gli elementi di `a`.

Il numero di **possibili** ordinamenti di `a=a[0]...a[n-1]` è pari a n!

(ad esempio, se $a = a[0]a[1]a[2]$, potremmo avere che
$a[0] \leq a[1] \leq a[2]$, oppure
$a[0] \leq a[2] \leq a[1]$, oppure

Vediamo il perchè

Sia $\mathcal{A}$ un **qualsivoglia** algoritmo di ordinamento, capace di ordinare arbitrarie sequenze `a=a[0]...a[n-1]` mediante confronti tra gli elementi di `a`.

Il numero di **possibili** ordinamenti di `a=a[0]...a[n-1]` è pari a $n!$

(ad esempio, se $a = a[0]a[1]a[2]$, potremmo avere che
$a[0] \leq a[1] \leq a[2]$, oppure
$a[0] \leq a[2] \leq a[1]$, oppure
$a[1] \leq a[0] \leq a[2]$, oppure

Vediamo il perchè

Sia $\mathcal{A}$ un **qualsivoglia** algoritmo di ordinamento, capace di ordinare arbitrarie sequenze `a=a[0]...a[n-1]` mediante confronti tra gli elementi di `a`.

Il numero di **possibili** ordinamenti di `a=a[0]...a[n-1]` è pari a $n!$

(ad esempio, se $a = a[0]a[1]a[2]$, potremmo avere che
$a[0] \leq a[1] \leq a[2]$, oppure
$a[0] \leq a[2] \leq a[1]$, oppure
$a[1] \leq a[0] \leq a[2]$, oppure
$a[1] \leq a[2] \leq a[0]$, oppure

Vediamo il perchè

Sia $\mathcal{A}$ un **qualsivoglia** algoritmo di ordinamento, capace di ordinare arbitrarie sequenze `a=a[0]...a[n-1]` mediante confronti tra gli elementi di `a`.

Il numero di **possibili** ordinamenti di `a=a[0]...a[n-1]` è pari a $n!$

(ad esempio, se $a = a[0]a[1]a[2]$, potremmo avere che
$a[0] \leq a[1] \leq a[2]$, oppure
$a[0] \leq a[2] \leq a[1]$, oppure
$a[1] \leq a[0] \leq a[2]$, oppure
$a[1] \leq a[2] \leq a[0]$, oppure
$a[2] \leq a[0] \leq a[1]$, oppure

Vediamo il perchè

Sia $\mathcal{A}$ un **qualsivoglia** algoritmo di ordinamento, capace di ordinare arbitrarie sequenze `a=a[0]...a[n-1]` mediante confronti tra gli elementi di `a`.

Il numero di **possibili** ordinamenti di `a=a[0]...a[n-1]` è pari a n!

(ad esempio, se $a = a[0]a[1]a[2]$, potremmo avere che
$a[0] \leq a[1] \leq a[2]$, oppure
$a[0] \leq a[2] \leq a[1]$, oppure
$a[1] \leq a[0] \leq a[2]$, oppure
$a[1] \leq a[2] \leq a[0]$, oppure
$a[2] \leq a[0] \leq a[1]$, oppure
$a[2] \leq a[1] \leq a[0]$,

Vediamo il perchè

Sia $\mathcal{A}$ un **qualsivoglia** algoritmo di ordinamento, capace di ordinare arbitrarie sequenze `a=a[0]...a[n-1]` mediante confronti tra gli elementi di `a`.

Il numero di **possibili** ordinamenti di `a=a[0]...a[n-1]` è pari a n!

(ad esempio, se $a = a[0]a[1]a[2]$, potremmo avere che
$a[0] \leq a[1] \leq a[2]$, oppure
$a[0] \leq a[2] \leq a[1]$, oppure
$a[1] \leq a[0] \leq a[2]$, oppure
$a[1] \leq a[2] \leq a[0]$, oppure
$a[2] \leq a[0] \leq a[1]$, oppure
$a[2] \leq a[1] \leq a[0]$,

ovvero $3! = 6$ possibili ordinamenti.

E allora?

E allora?

L'algoritmo $\mathcal{A}$ **deve** essere in grado di scoprire quale, tra gli $n!$ possibili ordinamenti, è quello corretto.

E allora?

L'algoritmo $\mathcal{A}$ **deve** essere in grado di scoprire quale, tra gli $n!$ possibili ordinamenti, è quello corretto.

- Sia `(a[i],a[j])` la generica **prima** coppia di elementi che l'algoritmo $\mathcal{A}$ confronterà .

E allora?

L'algoritmo $\mathcal{A}$ **deve** essere in grado di scoprire quale, tra gli $n!$ possibili ordinamenti, è quello corretto.

- Sia (a[i],a[j]) la generica **prima** coppia di elementi che l'algoritmo $\mathcal{A}$ confronterà .
- Sia A l'insieme di tutti i possibili ordinamenti di a=a[0]...a[n-1] in cui a[i]≤a[j]

E allora?

L'algoritmo $\mathcal{A}$ **deve** essere in grado di scoprire quale, tra gli $n!$ possibili ordinamenti, è quello corretto.

- Sia `(a[i],a[j])` la generica **prima** coppia di elementi che l'algoritmo $\mathcal{A}$ confronterà .
- Sia A l'insieme di tutti i possibili ordinamenti di `a=a[0]...a[n-1]` in cui `a[i]`$\leq$`a[j]` e B l'insieme di tutti i possibili ordinamenti di `a=a[0]...a[n-1]` in cui `a[i]`$>$`a[j]`.

E allora?

L'algoritmo $\mathcal{A}$ **deve** essere in grado di scoprire quale, tra gli $n!$ possibili ordinamenti, è quello corretto.

- Sia `(a[i],a[j])` la generica **prima** coppia di elementi che l'algoritmo $\mathcal{A}$ confronterà .
- Sia A l'insieme di tutti i possibili ordinamenti di `a=a[0]...a[n-1]` in cui `a[i]`$\leq$`a[j]` e B l'insieme di tutti i possibili ordinamenti di `a=a[0]...a[n-1]` in cui `a[i]`$>$`a[j]`.
- É ovvio che

$$|A| + |B| = n!$$

E allora?

L'algoritmo $\mathcal{A}$ **deve** essere in grado di scoprire quale, tra gli $n!$ possibili ordinamenti, è quello corretto.

- Sia `(a[i],a[j])` la generica **prima** coppia di elementi che l'algoritmo $\mathcal{A}$ confronterà .
- Sia A l'insieme di tutti i possibili ordinamenti di `a=a[0]...a[n-1]` in cui `a[i]`$\leq$`a[j]` e B l'insieme di tutti i possibili ordinamenti di `a=a[0]...a[n-1]` in cui `a[i]`$>$`a[j]`.
- É ovvio che

  $$|A| + |B| = n!$$

  da cui ne segue che o vale che $|A| \geq n!/2$

E allora?

L'algoritmo $\mathcal{A}$ **deve** essere in grado di scoprire quale, tra gli $n!$ possibili ordinamenti, è quello corretto.

- Sia `(a[i],a[j])` la generica **prima** coppia di elementi che l'algoritmo $\mathcal{A}$ confronterà .
- Sia A l'insieme di tutti i possibili ordinamenti di `a=a[0]...a[n-1]` in cui `a[i]`$\leq$`a[j]` e B l'insieme di tutti i possibili ordinamenti di `a=a[0]...a[n-1]` in cui `a[i]`$>$`a[j]`.
- É ovvio che

  $$|A| + |B| = n!$$

  da cui ne segue che o vale che $|A| \geq n!/2$ oppure vale che $|B| \geq n!/2$.

E allora?

L'algoritmo $\mathcal{A}$ **deve** essere in grado di scoprire quale, tra gli $n!$ possibili ordinamenti, è quello corretto.

- Sia `(a[i],a[j])` la generica **prima** coppia di elementi che l'algoritmo $\mathcal{A}$ confronterà .
- Sia A l'insieme di tutti i possibili ordinamenti di `a=a[0]...a[n-1]` in cui `a[i]`$\leq$`a[j]` e B l'insieme di tutti i possibili ordinamenti di `a=a[0]...a[n-1]` in cui `a[i]`$>$`a[j]`.
- É ovvio che
$$|A| + |B| = n!$$
da cui ne segue che o vale che $|A| \geq n!/2$ oppure vale che $|B| \geq n!/2$.
- Senza perdita di generalità , **assumiamo** che
$$|A| \geq \frac{1}{2} n!$$

- Visto che analizziamo la complessità degli algoritmi in base al loro caso **peggiore**,

- Visto che analizziamo la complessità degli algoritmi in base al loro caso **peggiore**, possiamo assumere che la risposta al confronto tra gli elementi della coppia `(a[i],a[j])` ci dice che `a[i]`$\leq$ `a[j]`,

- Visto che analizziamo la complessità degli algoritmi in base al loro caso **peggiore**, possiamo assumere che la risposta al confronto tra gli elementi della coppia `(a[i],a[j])` ci dice che `a[i]`$\leq$ `a[j]`, in questo modo l'ordinamento incognito di `a=a[0]...a[n-1]` che stiamo cercando **sappiamo** essere in A, che sappiamo anche avere cardinalità $|A| \geq n!/2$.

- Visto che analizziamo la complessità degli algoritmi in base al loro caso **peggiore**, possiamo assumere che la risposta al confronto tra gli elementi della coppia `(a[i],a[j])` ci dice che `a[i]`$\leq$ `a[j]`, in questo modo l'ordinamento incognito di `a=a[0]...a[n-1]` che stiamo cercando **sappiamo** essere in A, che sappiamo anche avere cardinalità $|A| \geq n!/2$.
- Consideriamo allora il secondo confronto che l'algoritmo $\mathcal{A}$ eseguirà,

- Visto che analizziamo la complessità degli algoritmi in base al loro caso **peggiore**, possiamo assumere che la risposta al confronto tra gli elementi della coppia (a[i],a[j]) ci dice che a[i]$\leq$ a[j], in questo modo l'ordinamento incognito di a=a[0]...a[n-1] che stiamo cercando **sappiamo** essere in A, che sappiamo anche avere cardinalità $|A| \geq n!/2$.
- Consideriamo allora il secondo confronto che l'algoritmo $\mathcal{A}$ eseguirà, sia (a[k],a[s]) la coppia di elementi coinvolta nel confronto.

- Visto che analizziamo la complessità degli algoritmi in base al loro caso **peggiore**, possiamo assumere che la risposta al confronto tra gli elementi della coppia `(a[i],a[j])` ci dice che `a[i]`$\leq$ `a[j]`, in questo modo l'ordinamento incognito di `a=a[0]...a[n-1]` che stiamo cercando **sappiamo** essere in A, che sappiamo anche avere cardinalità $|A| \geq n!/2$.
- Consideriamo allora il secondo confronto che l'algoritmo $\mathcal{A}$ eseguirà, sia `(a[k],a[s])` la coppia di elementi coinvolta nel confronto.
- Possiamo ripetere lo stesso ragionamento di prima, e partizionare l'insieme A in due sottoinsiemi C e D,

- Visto che analizziamo la complessità degli algoritmi in base al loro caso **peggiore**, possiamo assumere che la risposta al confronto tra gli elementi della coppia (a[i],a[j]) ci dice che a[i]$\leq$ a[j], in questo modo l'ordinamento incognito di a=a[0]...a[n-1] che stiamo cercando **sappiamo** essere in A, che sappiamo anche avere cardinalità $|A| \geq n!/2$.
- Consideriamo allora il secondo confronto che l'algoritmo $\mathcal{A}$ eseguirà, sia (a[k],a[s]) la coppia di elementi coinvolta nel confronto.
- Possiamo ripetere lo stesso ragionamento di prima, e partizionare l'insieme A in due sottoinsiemi C e D, dove C è fatto da tutti gli ordinamenti in A per cui vale che (a[k]$\leq$a[s]),

- Visto che analizziamo la complessità degli algoritmi in base al loro caso **peggiore**, possiamo assumere che la risposta al confronto tra gli elementi della coppia `(a[i],a[j])` ci dice che `a[i]`$\leq$ `a[j]`, in questo modo l'ordinamento incognito di `a=a[0]...a[n-1]` che stiamo cercando **sappiamo** essere in A, che sappiamo anche avere cardinalità $|A| \geq n!/2$.
- Consideriamo allora il secondo confronto che l'algoritmo $\mathcal{A}$ eseguirà, sia `(a[k],a[s])` la coppia di elementi coinvolta nel confronto.
- Possiamo ripetere lo stesso ragionamento di prima, e partizionare l'insieme A in due sottoinsiemi C e D, dove C è fatto da tutti gli ordinamenti in A per cui vale che `(a[k]`$\leq$`a[s])`, e D è composto da tutti gli ordinamenti in A per cui vale che `(a[k]`$>$`a[s])`.

- Visto che analizziamo la complessità degli algoritmi in base al loro caso **peggiore**, possiamo assumere che la risposta al confronto tra gli elementi della coppia (a[i],a[j]) ci dice che a[i]$\leq$ a[j], in questo modo l'ordinamento incognito di a=a[0]...a[n-1] che stiamo cercando **sappiamo** essere in A, che sappiamo anche avere cardinalità $|A| \geq n!/2$.
- Consideriamo allora il secondo confronto che l'algoritmo $\mathcal{A}$ eseguirà, sia (a[k],a[s]) la coppia di elementi coinvolta nel confronto.
- Possiamo ripetere lo stesso ragionamento di prima, e partizionare l'insieme A in due sottoinsiemi C e D, dove C è fatto da tutti gli ordinamenti in A per cui vale che (a[k]$\leq$a[s]), e D è composto da tutti gli ordinamenti in A per cui vale che (a[k]$>$a[s]). Vale che

$$|C| + |D| = |A|$$

- Visto che analizziamo la complessità degli algoritmi in base al loro caso **peggiore**, possiamo assumere che la risposta al confronto tra gli elementi della coppia (a[i],a[j]) ci dice che a[i]$\leq$ a[j], in questo modo l'ordinamento incognito di a=a[0]...a[n-1] che stiamo cercando **sappiamo** essere in A, che sappiamo anche avere cardinalità $|A| \geq n!/2$.
- Consideriamo allora il secondo confronto che l'algoritmo $\mathcal{A}$ eseguirà, sia (a[k],a[s]) la coppia di elementi coinvolta nel confronto.
- Possiamo ripetere lo stesso ragionamento di prima, e partizionare l'insieme A in due sottoinsiemi C e D, dove C è fatto da tutti gli ordinamenti in A per cui vale che (a[k]$\leq$a[s]), e D è composto da tutti gli ordinamenti in A per cui vale che (a[k]$>$a[s]). Vale che

$$|C| + |D| = |A|$$

- Come prima, **almeno uno degli insiemi** C e D avrà cardinalità

$$\geq \frac{1}{2}|A|$$

- Visto che analizziamo la complessità degli algoritmi in base al loro caso **peggiore**, possiamo assumere che la risposta al confronto tra gli elementi della coppia `(a[i],a[j])` ci dice che `a[i]`$\leq$ `a[j]`, in questo modo l'ordinamento incognito di `a=a[0]...a[n-1]` che stiamo cercando **sappiamo** essere in A, che sappiamo anche avere cardinalità $|A| \geq n!/2$.
- Consideriamo allora il secondo confronto che l'algoritmo $\mathcal{A}$ eseguirà, sia `(a[k],a[s])` la coppia di elementi coinvolta nel confronto.
- Possiamo ripetere lo stesso ragionamento di prima, e partizionare l'insieme A in due sottoinsiemi C e D, dove C è fatto da tutti gli ordinamenti in A per cui vale che `(a[k]`$\leq$`a[s])`, e D è composto da tutti gli ordinamenti in A per cui vale che `(a[k]`$>$`a[s])`. Vale che

$$|C| + |D| = |A|$$

- Come prima, **almeno uno degli insiemi** C e D avrà cardinalità

$$\geq \frac{1}{2}|A| \geq \frac{1}{2} \times \frac{1}{2} n!$$

- Visto che analizziamo la complessità degli algoritmi in base al loro caso **peggiore**, possiamo assumere che la risposta al confronto tra gli elementi della coppia (a[i],a[j]) ci dice che a[i]$\leq$ a[j], in questo modo l'ordinamento incognito di a=a[0]...a[n-1] che stiamo cercando **sappiamo** essere in A, che sappiamo anche avere cardinalità $|A| \geq n!/2$.
- Consideriamo allora il secondo confronto che l'algoritmo $\mathcal{A}$ eseguirà, sia (a[k],a[s]) la coppia di elementi coinvolta nel confronto.
- Possiamo ripetere lo stesso ragionamento di prima, e partizionare l'insieme A in due sottoinsiemi C e D, dove C è fatto da tutti gli ordinamenti in A per cui vale che (a[k]$\leq$a[s]), e D è composto da tutti gli ordinamenti in A per cui vale che (a[k]$>$a[s]). Vale che

$$|C| + |D| = |A|$$

- Come prima, **almeno uno degli insiemi** C e D avrà cardinalità

$$\geq \frac{1}{2}|A| \geq \frac{1}{2} \times \frac{1}{2} n! = \frac{n!}{4}$$

Continuiamo...

- Sempre perchè stiamo valutando il caso peggiore, possiamo supporre che la risposta al confronto tra gli elementi `(a[k],a[s])` ci dirà che l'ordinamento incognito si trova proprio in quel sottoinsieme tra C e D di cardinalità $\geq n!/4$

Continuiamo...

- Sempre perchè stiamo valutando il caso peggiore, possiamo supporre che la risposta al confronto tra gli elementi `(a[k],a[s])` ci dirà che l'ordinamento incognito si trova proprio in quel sottoinsieme tra C e D di cardinalità $\geq n!/4$ (insomma, siamo sempre sfortunati...)

Continuiamo...

- Sempre perchè stiamo valutando il caso peggiore, possiamo supporre che la risposta al confronto tra gli elementi `(a[k],a[s])` ci dirà che l'ordinamento incognito si trova proprio in quel sottoinsieme tra C e D di cardinalità $\geq n!/4$ (insomma, siamo sempre sfortunati...)
- Detto in altri termini, dopo due confronti **può** senz'altro accadere che l'ordinamento incognito si trovi

Continuiamo...

- Sempre perchè stiamo valutando il caso peggiore, possiamo supporre che la risposta al confronto tra gli elementi `(a[k],a[s])` ci dirà che l'ordinamento incognito si trova proprio in quel sottoinsieme tra C e D di cardinalità $\geq n!/4$ (insomma, siamo sempre sfortunati...)
- Detto in altri termini, dopo due confronti **può** senz'altro accadere che l'ordinamento incognito si trovi (nel caso peggiore)

Continuiamo...

- Sempre perchè stiamo valutando il caso peggiore, possiamo supporre che la risposta al confronto tra gli elementi `(a[k],a[s])` ci dirà che l'ordinamento incognito si trova proprio in quel sottoinsieme tra C e D di cardinalità $\geq n!/4$ (insomma, siamo sempre sfortunati...)
- Detto in altri termini, dopo due confronti **può** senz'altro accadere che l'ordinamento incognito si trovi (nel caso peggiore) in un insieme di cardinalità
$$\geq \frac{n!}{4}$$

Continuiamo...

- Sempre perchè stiamo valutando il caso peggiore, possiamo supporre che la risposta al confronto tra gli elementi `(a[k],a[s])` ci dirà che l'ordinamento incognito si trova proprio in quel sottoinsieme tra C e D di cardinalità $\geq n!/4$ (insomma, siamo sempre sfortunati...)
- Detto in altri termini, dopo due confronti **può** senz'altro accadere che l'ordinamento incognito si trovi (nel caso peggiore) in un insieme di cardinalità
$$\geq \frac{n!}{4}$$
- É evidente che possiamo iterare il ragionamento ad un numero arbitrario i di confronti,

Continuiamo...

- Sempre perchè stiamo valutando il caso peggiore, possiamo supporre che la risposta al confronto tra gli elementi `(a[k],a[s])` ci dirà che l'ordinamento incognito si trova proprio in quel sottoinsieme tra C e D di cardinalità $\geq n!/4$ (insomma, siamo sempre sfortunati...)
- Detto in altri termini, dopo due confronti **può** senz'altro accadere che l'ordinamento incognito si trovi (nel caso peggiore) in un insieme di cardinalità
$$\geq \frac{n!}{4}$$
- É evidente che possiamo iterare il ragionamento ad un numero arbitrario i di confronti, e che dopo averle eseguite, l'algoritmo sà **solo** che l'ordinamento incognito si trova, nel caso peggiore,

Continuiamo...

- Sempre perchè stiamo valutando il caso peggiore, possiamo supporre che la risposta al confronto tra gli elementi `(a[k],a[s])` ci dirà che l'ordinamento incognito si trova proprio in quel sottoinsieme tra C e D di cardinalità $\geq n!/4$ (insomma, siamo sempre sfortunati...)
- Detto in altri termini, dopo due confronti **può** senz'altro accadere che l'ordinamento incognito si trovi (nel caso peggiore) in un insieme di cardinalità
$$\geq \frac{n!}{4}$$
- É evidente che possiamo iterare il ragionamento ad un numero arbitrario i di confronti, e che dopo averle eseguite, l'algoritmo sà **solo** che l'ordinamento incognito si trova, nel caso peggiore, in un insieme di cardinalità
$$\geq \frac{n!}{2^i}$$

Continuiamo...

Quando l'algoritmo $\mathcal{A}$ si potrà fermare, dichiarando di aver scoperto con **certezza** l'ordimanento **giusto** di `a=a[0]...a[n-1]`?

Continuiamo...

Quando l'algoritmo $\mathcal{A}$ si potrà fermare, dichiarando di aver scoperto con **certezza** l'ordimanento **giusto** di `a=a[0]...a[n-1]`?

- Lo potrà fare **solo** quando l'insieme in cui è contenuto l'ordinamento incognito ha di cardinalità pari ad 1

Continuiamo...

Quando l'algoritmo $\mathcal{A}$ si potrà fermare, dichiarando di aver scoperto con **certezza** l'ordimanento **giusto** di `a=a[0]...a[n-1]`?

- Lo potrà fare **solo** quando l'insieme in cui è contenuto l'ordinamento incognito ha di cardinalità pari ad 1
- ciò comporta **necessariamente** che il numero i di confronti **deve** essere tale che $n!/2^i \leq 1$,

Continuiamo...

Quando l'algoritmo $\mathcal{A}$ si potrà fermare, dichiarando di aver scoperto con **certezza** l'ordimanento **giusto** di `a=a[0]...a[n-1]`?

- Lo potrà fare **solo** quando l'insieme in cui è contenuto l'ordinamento incognito ha di cardinalità pari ad 1
- ciò comporta **necessariamente** che il numero i di confronti **deve** essere tale che $n!/2^i \leq 1$, ovvero

$$i \geq \log n!$$

Continuiamo...

Quando l'algoritmo $\mathcal{A}$ si potrà fermare, dichiarando di aver scoperto con **certezza** l'ordimanento **giusto** di `a=a[0]...a[n-1]`?

- Lo potrà fare **solo** quando l'insieme in cui è contenuto l'ordinamento incognito ha di cardinalità pari ad 1
- ciò comporta **necessariamente** che il numero i di confronti **deve** essere tale che $n!/2^i \leq 1$, ovvero

$$i \geq \log n!$$

Noi volevamo mostrare che il tempo di esecuzione $O(n \log n)$ di MERGESOT non può essere battuto da nessun altro algoritmo.

Continuiamo...

Quando l'algoritmo $\mathcal{A}$ si potrà fermare, dichiarando di aver scoperto con **certezza** l'ordimanento **giusto** di `a=a[0]...a[n-1]`?

- Lo potrà fare **solo** quando l'insieme in cui è contenuto l'ordinamento incognito ha di cardinalità pari ad 1
- ciò comporta **necessariamente** che il numero i di confronti **deve** essere tale che $n!/2^i \leq 1$, ovvero

$$i \geq \log n!$$

Noi volevamo mostrare che il tempo di esecuzione $O(n \log n)$ di MERGESOT non può essere battuto da nessun altro algoritmo.

Sulla base di quanto prima detto, ci basterà provare che

$$i \geq \log n! = \Omega(n \log n)$$

E allora lo proviamo

Osserviamo che

$$n!$$

E allora lo proviamo

Osserviamo che

$$n! = n \cdot (n-1) \cdots 2 \cdot 1$$

E allora lo proviamo

Osserviamo che

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \geq \underbrace{n \cdot (n-1) \cdots \frac{n}{2}}_{\frac{n}{2} \text{volte}}$$

E allora lo proviamo

Osserviamo che

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \geq \underbrace{n \cdot (n-1) \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} \geq \underbrace{\frac{n}{2} \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}}$$

E allora lo proviamo

Osserviamo che

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \geq \underbrace{n \cdot (n-1) \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} \geq \underbrace{\frac{n}{2} \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} = \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

E allora lo proviamo

Osserviamo che

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \geq \underbrace{n \cdot (n-1) \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} \geq \underbrace{\frac{n}{2} \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} = \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

da cui

$\log n!$

E allora lo proviamo

Osserviamo che

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \geq \underbrace{n \cdot (n-1) \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} \geq \underbrace{\frac{n}{2} \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} = \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

da cui

$$\log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

E allora lo proviamo

Osserviamo che

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \geq \underbrace{n \cdot (n-1) \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} \geq \underbrace{\frac{n}{2} \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} = \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

da cui

$$\log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2}$$

E allora lo proviamo

Osserviamo che

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \geq \underbrace{n \cdot (n-1) \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} \geq \underbrace{\frac{n}{2} \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} = \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

da cui

$$\log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2} = \frac{n}{2}(\log n - \log 2)$$

E allora lo proviamo

Osserviamo che

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \geq \underbrace{n \cdot (n-1) \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} \geq \underbrace{\frac{n}{2} \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} = \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

da cui

$$\log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2} = \frac{n}{2}(\log n - \log 2) = \frac{n}{2} \log n - \frac{n}{2}$$

E allora lo proviamo

Osserviamo che

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \geq \underbrace{n \cdot (n-1) \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} \geq \underbrace{\frac{n}{2} \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} = \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

da cui

$$\log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2} = \frac{n}{2}(\log n - \log 2) = \frac{n}{2} \log n - \frac{n}{2} = \Omega(n \log n).$$

E allora lo proviamo

Osserviamo che

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \geq \underbrace{n \cdot (n-1) \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} \geq \underbrace{\frac{n}{2} \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} = \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

da cui

$$\log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2} = \frac{n}{2}(\log n - \log 2) = \frac{n}{2} \log n - \frac{n}{2} = \Omega(n \log n).$$

Mettendo tutto insieme, **ogni** algoritmo di ordinamento, basato su confronti, richiederà $\Omega(n \log n)$ confronti **nel caso peggiore** per ordinare arbitrarie sequenze composte da n elementi.

E allora lo proviamo

Osserviamo che

$$n! = n \cdot (n-1) \cdots 2 \cdot 1 \geq \underbrace{n \cdot (n-1) \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} \geq \underbrace{\frac{n}{2} \cdots \frac{n}{2}}_{\frac{n}{2}\text{volte}} = \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

da cui

$$\log n! \geq \log \left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2} \log \frac{n}{2} = \frac{n}{2}(\log n - \log 2) = \frac{n}{2} \log n - \frac{n}{2} = \Omega(n \log n).$$

Mettendo tutto insieme, **ogni** algoritmo di ordinamento, basato su confronti, richiederà $\Omega(n \log n)$ confronti **nel caso peggiore** per ordinare arbitrarie sequenze composte da n elementi.

Quindi, `MergeSort` è asintoticamente **ottimo** in quanto esegue un numero di operazioni $O(n \log n)$.

Ottimalità asintotica della Ricerca Binaria

Allo stesso modo possiamo provare l'ottimalità asintotica di altri algoritmi,

Ottimalità asintotica della Ricerca Binaria

Allo stesso modo possiamo provare l'ottimalità asintotica di altri algoritmi, ad esempio l'algoritmo di Ricerca Binaria

Ottimalità asintotica della Ricerca Binaria

Allo stesso modo possiamo provare l'ottimalità asintotica di altri algoritmi, ad esempio l'algoritmo di Ricerca Binaria

Quando cerchiamo la presenza (o meno) di un elemento `x` in un array arbitrari `a=a[0]...a[n-1]`,

Ottimalità asintotica della Ricerca Binaria

Allo stesso modo possiamo provare l'ottimalità asintotica di altri algoritmi, ad esempio l'algoritmo di Ricerca Binaria

Quando cerchiamo la presenza (o meno) di un elemento `x` in un array arbitrari `a=a[0]...a[n-1]`, il numero di possibili risposte (output) è $n + 1$

Ottimalità asintotica della Ricerca Binaria

Allo stesso modo possiamo provare l'ottimalità asintotica di altri algoritmi, ad esempio l'algoritmo di Ricerca Binaria

Quando cerchiamo la presenza (o meno) di un elemento `x` in un array arbitrari `a=a[0]...a[n-1]`, il numero di possibili risposte (output) è $n+1$ (nel caso del problema dell'ordinamento il numero di possibili output è $n!$).

Ottimalità asintotica della Ricerca Binaria

Allo stesso modo possiamo provare l'ottimalità asintotica di altri algoritmi, ad esempio l'algoritmo di Ricerca Binaria

Quando cerchiamo la presenza (o meno) di un elemento `x` in un array arbitrari `a=a[0]...a[n-1]`, il numero di possibili risposte (output) è $n+1$ (nel caso del problema dell'ordinamento il numero di possibili output è $n!$).

Ogni confronto tra elementi ci potrà far eliminare **al più metà** delle possibili risposte.

Ottimalità asintotica della Ricerca Binaria

Allo stesso modo possiamo provare l'ottimalità asintotica di altri algoritmi, ad esempio l'algoritmo di Ricerca Binaria

Quando cerchiamo la presenza (o meno) di un elemento `x` in un array arbitrari `a=a[0]...a[n-1]`, il numero di possibili risposte (output) è $n+1$ (nel caso del problema dell'ordinamento il numero di possibili output è $n!$).

Ogni confronto tra elementi ci potrà far eliminare **al più metà** delle possibili risposte. La condizione di terminazione di ogni algoritmo è che vi sia una **sola possibile** risposta.

Ottimalità asintotica della Ricerca Binaria

Allo stesso modo possiamo provare l'ottimalità asintotica di altri algoritmi, ad esempio l'algoritmo di Ricerca Binaria

Quando cerchiamo la presenza (o meno) di un elemento `x` in un array arbitrari `a=a[0]...a[n-1]`, il numero di possibili risposte (output) è $n+1$ (nel caso del problema dell'ordinamento il numero di possibili output è $n!$).

Ogni confronto tra elementi ci potrà far eliminare **al più metà** delle possibili risposte. La condizione di terminazione di ogni algoritmo è che vi sia una **sola possibile** risposta.

Pertando, avremo bisogno di un numero di confronti i tale che il numero di possibili risposte sia al più pari ad 1,

Ottimalità asintotica della Ricerca Binaria

Allo stesso modo possiamo provare l'ottimalità asintotica di altri algoritmi, ad esempio l'algoritmo di Ricerca Binaria

Quando cerchiamo la presenza (o meno) di un elemento `x` in un array arbitrari `a=a[0]...a[n-1]`, il numero di possibili risposte (output) è $n+1$ (nel caso del problema dell'ordinamento il numero di possibili output è $n!$).

Ogni confronto tra elementi ci potrà far eliminare **al più metà** delle possibili risposte. La condizione di terminazione di ogni algoritmo è che vi sia una **sola possibile** risposta.

Pertando, avremo bisogno di un numero di confronti i tale che il numero di possibili risposte sia al più pari ad 1, ovvero

$$\frac{n+1}{2^i} \leq 1,$$

Ottimalità asintotica della Ricerca Binaria

Allo stesso modo possiamo provare l'ottimalità asintotica di altri algoritmi, ad esempio l'algoritmo di Ricerca Binaria

Quando cerchiamo la presenza (o meno) di un elemento `x` in un array arbitrari `a=a[0]...a[n-1]`, il numero di possibili risposte (output) è $n + 1$ (nel caso del problema dell'ordinamento il numero di possibili output è $n!$).

Ogni confronto tra elementi ci potrà far eliminare **al più metà** delle possibili risposte. La condizione di terminazione di ogni algoritmo è che vi sia una **sola possibile** risposta.

Pertando, avremo bisogno di un numero di confronti i tale che il numero di possibili risposte sia al più pari ad 1, ovvero

$$\frac{n+1}{2^i} \leq 1, \Rightarrow i \geq \log(n+1)$$

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore `a=a[1]a[2]...a[n]`, **ordinato** in senso crescente,

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore `a=a[1]a[2]...a[n]`, **ordinato** in senso crescente, di *n* numeri **distinti**,

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore `a=a[1]a[2]...a[n]`, **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore `a=a[1]a[2]...a[n]`, **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in `a`

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore `a=a[1]a[2]...a[n]`, **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in `a`

Esempio:
`a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]`

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore `a=a[1]a[2]...a[n]`, **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in `a`

Esempio:
`a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11`

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore `a=a[1]a[2]...a[n]`, **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in `a`

Esempio:
`a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11`

manca 5.

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore a=a[1]a[2]...a[n], **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in a

Esempio:
a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11

manca 5.

Analisi del problema:

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore `a=a[1]a[2]...a[n]`, **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in `a`

Esempio:
`a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11`

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore `a=a[1]a[2]...a[n]`

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore `a=a[1]a[2]...a[n]`, **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in `a`

Esempio:
`a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11`

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore `a=a[1]a[2]...a[n]`
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $i > k$ sono memorizzati nella posizione $i-1$-esima del vettore `a=a[1]a[2]...a[n]`.

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore `a=a[1]a[2]...a[n]`, **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in a

Esempio:
`a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11`

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore `a=a[1]a[2]...a[n]`
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $i > k$ sono memorizzati nella posizione $i-1$-esima del vettore `a=a[1]a[2]...a[n]`.

Esempio: `a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]`

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore a=a[1]a[2]...a[n], **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in a

Esempio:
a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore a=a[1]a[2]...a[n]
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $i > k$ sono memorizzati nella posizione $i-1$-esima del vettore a=a[1]a[2]...a[n].

Esempio: a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]
= 1

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore a=a[1]a[2]...a[n], **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in a

Esempio:
a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore a=a[1]a[2]...a[n]
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $i > k$ sono memorizzati nella posizione $i-1$-esima del vettore a=a[1]a[2]...a[n].

Esempio: a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]
= 1 2

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore `a=a[1]a[2]...a[n]`, **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in a

Esempio:
`a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11`

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore `a=a[1]a[2]...a[n]`
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $i > k$ sono memorizzati nella posizione $i-1$-esima del vettore `a=a[1]a[2]...a[n]`.

Esempio: `a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]`
`= 1    2    3`

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore a=a[1]a[2]...a[n], **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in a

Esempio:
a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore a=a[1]a[2]...a[n]
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $i > k$ sono memorizzati nella posizione $i-1$-esima del vettore a=a[1]a[2]...a[n].

Esempio: a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]
= 1 2 3 4

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore `a=a[1]a[2]...a[n]`, **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in a

Esempio:
`a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11`

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore `a=a[1]a[2]...a[n]`
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $i > k$ sono memorizzati nella posizione $i-1$-esima del vettore `a=a[1]a[2]...a[n]`.

Esempio: `a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]`
`= 1 2 3 4 6`

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore a=a[1]a[2]...a[n], **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in a

Esempio:
a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore a=a[1]a[2]...a[n]
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $i > k$ sono memorizzati nella posizione $i-1$-esima del vettore a=a[1]a[2]...a[n].

Esempio: a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]
= 1 2 3 4 6 7

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore a=a[1]a[2]...a[n], **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in a

Esempio:
a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore a=a[1]a[2]...a[n]
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $i > k$ sono memorizzati nella posizione $i-1$-esima del vettore a=a[1]a[2]...a[n].

Esempio: a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]
= 1 2 3 4 6 7 8

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore a=a[1]a[2]...a[n], **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in a

Esempio:
a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore a=a[1]a[2]...a[n]
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $i > k$ sono memorizzati nella posizione $i-1$-esima del vettore a=a[1]a[2]...a[n].

Esempio: a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]
= 1 2 3 4 6 7 8 9

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore `a=a[1]a[2]...a[n]`, **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in a

Esempio:
`a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11`

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore `a=a[1]a[2]...a[n]`
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $i > k$ sono memorizzati nella posizione $i-1$-esima del vettore `a=a[1]a[2]...a[n]`.

Esempio: `a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]`
`= 1 2 3 4 6 7 8 9 10`

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore `a=a[1]a[2]...a[n]`, **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in a

Esempio:
`a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11`

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore `a=a[1]a[2]...a[n]`
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $i > k$ sono memorizzati nella posizione $i-1$-esima del vettore `a=a[1]a[2]...a[n]`.

Esempio: `a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]`
`= 1 2 3 4 6 7 8 9 10 11`

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore a=a[1]a[2]...a[n], **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in a

Esempio:
a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore a=a[1]a[2]...a[n]
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $i > k$ sono memorizzati nella posizione $i-1$-esima del vettore a=a[1]a[2]...a[n].

Esempio: a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]
= 1 2 3 4 6 7 8 9 10 11

Troviamo quindi il **più grande** indice i per cui a[i]=i

Esercizio

Consideriamo il seguente problema algoritmico:
Input: vettore a=a[1]a[2]...a[n], **ordinato** in senso crescente, di n numeri **distinti**, con $a[i] \in \{1, 2, \ldots, n+1\}$, per ogni $i = 1, \ldots, n$
Output: l'unico intero $k \in \{1, 2, \ldots, n+1\}$ che **non** compare in a

Esempio:
a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]= 1 2 3 4 6 7 8 9 10 11

manca 5.

Analisi del problema: Poichè manca un unico valore k,
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $1 \leq i < k$ sono memorizzati nella posizione i-esima del vettore a=a[1]a[2]...a[n]
tutti i valori $i \in \{1, 2, \ldots, n+1\}$, con $i > k$ sono memorizzati nella posizione $i-1$-esima del vettore a=a[1]a[2]...a[n].

Esempio: a=a[1]a[2]a[3]a[4]a[5]a[6]a[7]a[8]a[9]a[10]
= 1 2 3 4 6 7 8 9 10 11

Troviamo quindi il **più grande** indice i per cui a[i]=i ed il valore **mancante** sarà $k = i + 1$.

Analizzando le posizioni di `a[i]...a[j]`, calcoliamo $m = \lceil (i+j)/2 \rceil$.

Analizzando le posizioni di a[i] . . . a[j], calcoliamo $m = \lceil (i + j)/2 \rceil$.

- **Se** a[m]=m

Analizzando le posizioni di `a[i]...a[j]`, calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** `a[m]=m` il **più grande** indice s per cui `a[s]=s` è compreso tra m e j.

Analizzando le posizioni di `a[i]`...`a[j]`, calcoliamo $m = \lceil (i + j)/2 \rceil$.

- **Se** `a[m]=m` il **più grande** indice s per cui `a[s]=s` è compreso tra m e j.
- **Se** `a[m]>m`

Analizzando le posizioni di `a[i]`...`a[j]`, calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** `a[m]=m` il **più grande** indice s per cui `a[s]=s` è compreso tra m e j.
- **Se** `a[m]>m` il **più grande** indice s per cui `a[s]=s` è compreso tra i e $m-1$.

Analizzando le posizioni di `a[i]...a[j]`, calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** `a[m]=m` il **più grande** indice s per cui `a[s]=s` è compreso tra m e j.
- **Se** `a[m]>m` il **più grande** indice s per cui `a[s]=s` è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero,

Analizzando le posizioni di `a[i]...a[j]`, calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** `a[m]=m` il **più grande** indice s per cui `a[s]=s` è compreso tra m e j.
- **Se** `a[m]>m` il **più grande** indice s per cui `a[s]=s` è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$,

Analizzando le posizioni di `a[i]...a[j]`, calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** `a[m]=m` il **più grande** indice s per cui `a[s]=s` è compreso tra m e j.
- **Se** `a[m]>m` il **più grande** indice s per cui `a[s]=s` è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

Analizzando le posizioni di a[i]...a[j], calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** a[m]=m il **più grande** indice s per cui a[s]=s è compreso tra m e j.
- **Se** a[m]>m il **più grande** indice s per cui a[s]=s è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

```
Mancante(a,i,j)   % [cerca in a[i]...a[j], 1≤ i ≤ j≤ n]
```

Analizzando le posizioni di `a[i]...a[j]`, calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** `a[m]=m` il **più grande** indice s per cui `a[s]=s` è compreso tra m e j.
- **Se** `a[m]>m` il **più grande** indice s per cui `a[s]=s` è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

```
Mancante(a,i,j)   % [cerca in a[i]...a[j], 1≤ i ≤ j≤ n]
1. IF(i==j) {
```

Analizzando le posizioni di a[i]...a[j], calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** a[m]=m il **più grande** indice s per cui a[s]=s è compreso tra m e j.
- **Se** a[m]>m il **più grande** indice s per cui a[s]=s è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

```
Mancante(a,i,j)   % [cerca in a[i]...a[j], 1≤ i ≤ j≤ n]
1. IF(i==j) {
2.       IF(a[i]==i) {
```

Analizzando le posizioni di a[i]...a[j], calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** a[m]=m il **più grande** indice s per cui a[s]=s è compreso tra m e j.
- **Se** a[m]>m il **più grande** indice s per cui a[s]=s è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

```
Mancante(a,i,j)   % [cerca in a[i]...a[j], 1≤ i ≤ j≤ n]
1. IF(i==j) {
2.        IF(a[i]==i) {
3.          return i+1
             }
```

Analizzando le posizioni di a[i]...a[j], calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** a[m]=m il **più grande** indice s per cui a[s]=s è compreso tra m e j.
- **Se** a[m]>m il **più grande** indice s per cui a[s]=s è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

```
Mancante(a,i,j)   % [cerca in a[i]...a[j], 1≤ i ≤ j≤ n]
1. IF(i==j) {
2.       IF(a[i]==i) {
3.          return i+1
            } ELSE {
4.          return i
   }     }
```

Analizzando le posizioni di a[i]...a[j], calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** a[m]=m il **più grande** indice s per cui a[s]=s è compreso tra m e j.
- **Se** a[m]>m il **più grande** indice s per cui a[s]=s è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

```
Mancante(a,i,j)   % [cerca in a[i]...a[j], 1≤ i ≤ j≤ n]
1. IF(i==j) {
2.       IF(a[i]==i) {
3.         return i+1
            } ELSE {
4.         return i
   }     }
5. m=(i+j)/2
```

Analizzando le posizioni di a[i]...a[j], calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** a[m]=m il **più grande** indice s per cui a[s]=s è compreso tra m e j.
- **Se** a[m]>m il **più grande** indice s per cui a[s]=s è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

```
Mancante(a,i,j)   % [cerca in a[i]...a[j], 1≤ i ≤ j≤ n]
1. IF(i==j) {
2.        IF(a[i]==i) {
3.           return i+1
             } ELSE {
4.           return i
  }     }
5. m=(i+j)/2
6. IF(a[m]==m)
```

Analizzando le posizioni di a[i]...a[j], calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** a[m]=m il **più grande** indice s per cui a[s]=s è compreso tra m e j.
- **Se** a[m]>m il **più grande** indice s per cui a[s]=s è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

```
Mancante(a,i,j)   % [cerca in a[i]...a[j], 1≤ i ≤ j≤ n]
1. IF(i==j) {
2.        IF(a[i]==i) {
3.           return i+1
              } ELSE {
4.           return i
   }     }
5. m=(i+j)/2
6. IF(a[m]==m) {
7.        return Mancante(a,m+1,j)
          }
```

Analizzando le posizioni di a[i]...a[j], calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** a[m]=m il **più grande** indice s per cui a[s]=s è compreso tra m e j.
- **Se** a[m]>m il **più grande** indice s per cui a[s]=s è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

```
Mancante(a,i,j)   % [cerca in a[i]...a[j], 1≤ i ≤ j≤ n]
1. IF(i==j) {
2.       IF(a[i]==i) {
3.          return i+1
             } ELSE {
4.          return i
   }     }
5. m=(i+j)/2
6. IF(a[m]==m) {
7.       return Mancante(a,m+1,j)
         } ELSE {
8.       return Mancante(a,i,m-1)
}
```

Analizzando le posizioni di a[i]...a[j], calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** a[m]=m il **più grande** indice s per cui a[s]=s è compreso tra m e j.
- **Se** a[m]>m il **più grande** indice s per cui a[s]=s è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

```
Mancante(a,i,j)   % [cerca in a[i]...a[j], 1≤ i ≤ j≤ n]
1. IF(i==j) {
2.        IF(a[i]==i) {
3.           return i+1
               } ELSE {
4.           return i
   }      }
5. m=(i+j)/2
6. IF(a[m]==m) {
7.        return Mancante(a,m+1,j)
          } ELSE {
8.        return Mancante(a,i,m-1)
}
```

Complessità: $T(n)$

Analizzando le posizioni di a[i]...a[j], calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** a[m]=m il **più grande** indice s per cui a[s]=s è compreso tra m e j.
- **Se** a[m]>m il **più grande** indice s per cui a[s]=s è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

```
Mancante(a,i,j)   % [cerca in a[i]...a[j], 1≤ i ≤ j≤ n]
1. IF(i==j) {
2.       IF(a[i]==i) {
3.          return i+1
            } ELSE {
4.          return i
   }     }
5. m=(i+j)/2
6. IF(a[m]==m) {
7.       return Mancante(a,m+1,j)
         } ELSE {
8.       return Mancante(a,i,m-1)
}
```

Complessità: $T(n) = T(n/2)$

Analizzando le posizioni di a[i]...a[j], calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** a[m]=m il **più grande** indice s per cui a[s]=s è compreso tra m e j.
- **Se** a[m]>m il **più grande** indice s per cui a[s]=s è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

```
Mancante(a,i,j)   % [cerca in a[i]...a[j], 1≤ i ≤ j≤ n]
1. IF(i==j) {
2.       IF(a[i]==i) {
3.          return i+1
             } ELSE {
4.          return i
   }     }
5. m=(i+j)/2
6. IF(a[m]==m) {
7.       return Mancante(a,m+1,j)
         } ELSE {
8.       return Mancante(a,i,m-1)
}
```

Complessità: $T(n) = T(n/2) + c$

Analizzando le posizioni di a[i]...a[j], calcoliamo $m = \lceil (i+j)/2 \rceil$.

- **Se** a[m]=m il **più grande** indice s per cui a[s]=s è compreso tra m e j.
- **Se** a[m]>m il **più grande** indice s per cui a[s]=s è compreso tra i e $m-1$.

Quando ci siamo ridotti a considerare un unico numero, ovvero quando $i = j$, abbiamo trovato il numero mancante!

```
Mancante(a,i,j)   % [cerca in a[i]...a[j], 1≤ i ≤ j≤ n]
1. IF(i==j) {
2.       IF(a[i]==i) {
3.          return i+1
             } ELSE {
4.          return i
   }     }
5. m=(i+j)/2
6. IF(a[m]==m) {
7.       return Mancante(a,m+1,j)
         } ELSE {
8.       return Mancante(a,i,m-1)
}
```

Complessità: $T(n) = T(n/2) + c \Rightarrow T(n) = O(\log n)$

Esercizio

Supponiamo di avere un array A di n numeri positivi, che rappresentano il prezzo di una data azione in n giorni consecutivi

Esercizio

Supponiamo di avere un array A di n numeri positivi, che rappresentano il prezzo di una data azione in n giorni consecutivi (cioè $A[i]$ =prezzo dell'azione al giorno i-esimo).

Esercizio

Supponiamo di avere un array A di n numeri positivi, che rappresentano il prezzo di una data azione in n giorni consecutivi (cioè $A[i] =$prezzo dell'azione al giorno i-esimo).
Vogliamo calcolare il *massimo* profitto che possiamo ottenere comprando un'azione nel giorno i e vendendola nel giorno $j \geq i$,

Esercizio

Supponiamo di avere un array A di n numeri positivi, che rappresentano il prezzo di una data azione in n giorni consecutivi (cioè $A[i]$ =prezzo dell'azione al giorno i-esimo).
Vogliamo calcolare il *massimo* profitto che possiamo ottenere comprando un'azione nel giorno i e vendendola nel giorno $j \geq i$, detto in altri temini vogliamo risolvere il seguente problema algoritmico:

Input: Array $A = A[1...n]$ di n numeri interi positivi.
Output:

$$\max_{1 \leq i \leq j \leq n} (A[j] - A[i]).$$

Esercizio

Supponiamo di avere un array A di n numeri positivi, che rappresentano il prezzo di una data azione in n giorni consecutivi (cioè $A[i]$ =prezzo dell'azione al giorno i-esimo).
Vogliamo calcolare il *massimo* profitto che possiamo ottenere comprando un'azione nel giorno i e vendendola nel giorno $j \geq i$, detto in altri temini vogliamo risolvere il seguente problema algoritmico:

Input: Array $A = A[1...n]$ di n numeri interi positivi.
Output:

$$\max_{1 \leq i \leq j \leq n} (A[j] - A[i]).$$

Esempio: Sia $A = A[1...8] = [3, 8, 1, 5, 6, 7, 2, 4]$.

Esercizio

Supponiamo di avere un array A di n numeri positivi, che rappresentano il prezzo di una data azione in n giorni consecutivi (cioè $A[i]$ =prezzo dell'azione al giorno i-esimo).
Vogliamo calcolare il *massimo* profitto che possiamo ottenere comprando un'azione nel giorno i e vendendola nel giorno $j \geq i$, detto in altri temini vogliamo risolvere il seguente problema algoritmico:

Input: Array $A = A[1...n]$ di n numeri interi positivi.
Output:

$$\max_{1 \leq i \leq j \leq n} (A[j] - A[i]).$$

Esempio: Sia $A = A[1...8] = [3, 8, 1, 5, 6, 7, 2, 4]$. Si può vedere che

$$\max_{1 \leq i \leq j \leq 8} (A[j] - A[i]) = A[6] - A[3] = 6.$$

Output: $\max_{1 \leq i \leq j \leq n}(A[j] - A[i])$

Prima idea: calcola *tutti* i valori $A[j] - A[i]$ e ritorna il massimo.

Output: $\max_{1 \leq i \leq j \leq n}(A[j] - A[i])$

Prima idea: calcola *tutti* i valori $A[j] - A[i]$ e ritorna il massimo.

```
MaxProfitto1(A)
   bestProfit = 0
```

Output: $\max_{1 \le i \le j \le n}(A[j] - A[i])$

Prima idea: calcola *tutti* i valori $A[j] - A[i]$ e ritorna il massimo.

```
MaxProfitto1(A)
    bestProfit = 0
    FOR(i = 1; i < n + 1; i = i + 1)
```

Output: $\max_{1 \leq i \leq j \leq n}(A[j] - A[i])$

Prima idea: calcola *tutti* i valori $A[j] - A[i]$ e ritorna il massimo.

```
MaxProfitto1(A)
    bestProfit = 0
    FOR(i = 1; i < n + 1; i = i + 1)
       FOR(j = i; j < n + 1; i = i + 1)
```

Output: $\max_{1\leq i\leq j\leq n}(A[j]-A[i])$

Prima idea: calcola *tutti* i valori $A[j]-A[i]$ e ritorna il massimo.

```
MaxProfitto1(A)
   bestProfit = 0
   FOR(i = 1; i < n + 1; i = i + 1)
     FOR(j = i; j < n + 1; i = i + 1)
       bestProfit = max(bestProfit,A[j]-A[i])
```

Output: $\max_{1 \le i \le j \le n}(A[j] - A[i])$

Prima idea: calcola *tutti* i valori $A[j] - A[i]$ e ritorna il massimo.

```
MaxProfitto1(A)
   bestProfit = 0
   FOR(i = 1; i < n + 1; i = i + 1)
     FOR(j = i; j < n + 1; i = i + 1)
       bestProfit = max(bestProfit,A[j]-A[i])
   return bestProfit
```

Output: $\max_{1 \leq i \leq j \leq n}(A[j] - A[i])$

Prima idea: calcola *tutti* i valori $A[j] - A[i]$ e ritorna il massimo.

```
MaxProfitto1(A)
   bestProfit = 0
   FOR(i = 1; i < n + 1; i = i + 1)
     FOR(j = i; j < n + 1; i = i + 1)
       bestProfit = max(bestProfit,A[j]-A[i])
   return bestProfit
```

Complessità $T(n) = \Theta(n^2)$.

Output: $\max_{1 \leq i \leq j \leq n} (A[j] - A[i])$

Seconda idea: applica D&I

Output: $\max_{1 \le i \le j \le n}(A[j] - A[i])$

Seconda idea: applica D&I

```
MaxProfitto2(A,i,j)
1.    if(i == j) {
```

Output: $\max_{1 \leq i \leq j \leq n}(A[j] - A[i])$

Seconda idea: applica D&I

```
MaxProfitto2(A,i,j)
1.    if(i == j) {
2.      return 0
     }
```

Output: $\max_{1 \le i \le j \le n}(A[j] - A[i])$

Seconda idea: applica D&I

```
MaxProfitto2(A,i,j)
1.    if(i == j) {
2.      return 0
      }
3.    m = (i + j)/2
```

Output: $\max_{1 \le i \le j \le n}(A[j] - A[i])$

Seconda idea: applica D&I

```
MaxProfitto2(A,i,j)
1.   if(i == j) {
2.     return 0
    }
3.   m = (i + j)/2
4.   miglioreaS=MaxProfitto2(A,i,m)
```

Output: $\max_{1 \leq i \leq j \leq n}(A[j] - A[i])$

Seconda idea: applica D&I

```
MaxProfitto2(A,i,j)
1.    if(i == j) {
2.      return 0
      }
3.    m = (i + j)/2
4.    miglioreaS=MaxProfitto2(A,i,m)
5.    miglioreaD=MaxProfitto2(A,m+1,j)
```

Output: $\max_{1 \leq i \leq j \leq n}(A[j] - A[i])$

Seconda idea: applica D&I

```
MaxProfitto2(A,i,j)
1.    if(i == j) {
2.      return 0
      }
3.    m = (i + j)/2
4.    miglioreaS=MaxProfitto2(A,i,m)
5.    miglioreaD=MaxProfitto2(A,m+1,j)
6.    miglioreaC=max(A[m+1...j])-min(A[i...m])
```

Output: $\max_{1 \leq i \leq j \leq n}(A[j] - A[i])$

Seconda idea: applica D&I

```
MaxProfitto2(A,i,j)
1.   if(i == j) {
2.     return 0
     }
3.   m = (i + j)/2
4.   miglioreaS=MaxProfitto2(A,i,m)
5.   miglioreaD=MaxProfitto2(A,m+1,j)
6.   miglioreaC=max(A[m+1...j])-min(A[i...m])
7.   return max(miglioreaS, miglioreaD, miglioreaC)
```

Output: $\max_{1 \le i \le j \le n}(A[j] - A[i])$

Seconda idea: applica D&I

```
MaxProfitto2(A,i,j)
1.    if(i == j) {
2.      return 0
     }
3.    m = (i + j)/2
4.    miglioreaS=MaxProfitto2(A,i,m)
5.    miglioreaD=MaxProfitto2(A,m+1,j)
6.    miglioreaC=max(A[m+1...j])-min(A[i...m])
7.    return max(miglioreaS, miglioreaD, miglioreaC)
```

Complessità $T(n) = 2T(n/2) + O(n)$

Output: $\max_{1 \leq i \leq j \leq n}(A[j] - A[i])$

Seconda idea: applica D&I

```
MaxProfitto2(A,i,j)
1.   if(i == j) {
2.     return 0
     }
3.   m = (i + j)/2
4.   miglioreaS=MaxProfitto2(A,i,m)
5.   miglioreaD=MaxProfitto2(A,m+1,j)
6.   miglioreaC=max(A[m+1...j])-min(A[i...m])
7.   return max(miglioreaS, miglioreaD, miglioreaC)
```

Complessità $T(n) = 2T(n/2) + O(n) = O(n \log n)$

Output: $\max_{1 \leq i \leq j \leq n}(A[j] - A[i])$

Terza idea: applica D&I *anche* per calcolare il `max(A[m+1...j])` e `min(A[i...m])`.

Output: $\max_{1\leq i\leq j\leq n}(A[j] - A[i])$

Terza idea: applica D&I *anche* per calcolare il `max(A[m+1...j])` e `min(A[i...m])`.

```
MaxProfitto3(A,i,j)
1.    if(i == j) {
```

Output: $\max_{1 \leq i \leq j \leq n}(A[j] - A[i])$

Terza idea: applica D&I *anche* per calcolare il `max(A[m+1...j])` e `min(A[i...m])`.

```
MaxProfitto3(A,i,j)
1.   if(i == j) {
2.     return(0,A[i],A[j])
     }
```

Output: $\max_{1\leq i\leq j\leq n}(A[j]-A[i])$

Terza idea: applica D&I *anche* per calcolare il `max(A[m+1...j])` e `min(A[i...m])`.

```
MaxProfitto3(A,i,j)
1.    if(i == j) {
2.      return(0,A[i],A[j])
      }
3.    m = (i + j)/2
```

Output: $\max_{1 \leq i \leq j \leq n}(A[j] - A[i])$

Terza idea: applica D&I *anche* per calcolare il `max(A[m+1...j])` e `min(A[i...m])`.

```
MaxProfitto3(A,i,j)
1.   if(i == j) {
2.     return(0,A[i],A[j])
     }
3.   m = (i + j)/2
4.   (miglioreaS,minaS,maxaS)=MaxProfitto3(A,i,m)
```

Output: $\max_{1 \le i \le j \le n}(A[j] - A[i])$

Terza idea: applica D&I *anche* per calcolare il `max(A[m+1...j])` e `min(A[i...m])`.

```
MaxProfitto3(A,i,j)
1.    if(i == j) {
2.      return(0,A[i],A[j])
      }
3.    m = (i + j)/2
4.    (miglioreaS,minaS,maxaS)=MaxProfitto3(A,i,m)
5.    (miglioreaD,minaD,maxaD)=MaxProfitto3(A,m+1,j)
```

Output: $\max_{1 \leq i \leq j \leq n}(A[j] - A[i])$

Terza idea: applica D&I *anche* per calcolare il `max(A[m+1...j])` e `min(A[i...m])`.

```
MaxProfitto3(A,i,j)
1.    if(i == j) {
2.      return(0,A[i],A[j])
     }
3.    m = (i + j)/2
4.    (miglioreaS,minaS,maxaS)=MaxProfitto3(A,i,m)
5.    (miglioreaD,minaD,maxaD)=MaxProfitto3(A,m+1,j)
6.    migliore=max(miglioreaS,miglioreaD,maxaD-minaS)
```

Output: $\max_{1 \le i \le j \le n}(A[j] - A[i])$

Terza idea: applica D&I *anche* per calcolare il `max(A[m+1...j])` e `min(A[i...m])`.

```
MaxProfitto3(A,i,j)
1.   if(i == j) {
2.     return(0,A[i],A[j])
     }
3.   m = (i + j)/2
4.   (miglioreaS,minaS,maxaS)=MaxProfitto3(A,i,m)
5.   (miglioreaD,minaD,maxaD)=MaxProfitto3(A,m+1,j)
6.   migliore=max(miglioreaS,miglioreaD,maxaD-minaS)
7.   return(migliore,min(minaS,minaD),max(maxaS,maxaD))
```

Output: $\max_{1 \leq i \leq j \leq n}(A[j] - A[i])$

Terza idea: applica D&I *anche* per calcolare il `max(A[m+1...j])` e `min(A[i...m])`.

```
MaxProfitto3(A,i,j)
1.   if(i == j) {
2.     return(0,A[i],A[j])
     }
3.   m = (i + j)/2
4.   (miglioreaS,minaS,maxaS)=MaxProfitto3(A,i,m)
5.   (miglioreaD,minaD,maxaD)=MaxProfitto3(A,m+1,j)
6.   migliore=max(miglioreaS,miglioreaD,maxaD-minaS)
7.   return(migliore,min(minaS,minaD),max(maxaS,maxaD))
```

Complessità $T(n) = 2T(n/2) + O(1)$

Output: $\max_{1 \le i \le j \le n}(A[j] - A[i])$

Terza idea: applica D&I *anche* per calcolare il `max(A[m+1...j])` e `min(A[i...m])`.

```
MaxProfitto3(A,i,j)
1.   if(i == j) {
2.     return(0,A[i],A[j])
     }
3.   m = (i + j)/2
4.   (miglioreaS,minaS,maxaS)=MaxProfitto3(A,i,m)
5.   (miglioreaD,minaD,maxaD)=MaxProfitto3(A,m+1,j)
6.   migliore=max(miglioreaS,miglioreaD,maxaD-minaS)
7.   return(migliore,min(minaS,minaD),max(maxaS,maxaD))
```

Complessità $T(n) = 2T(n/2) + O(1) = O(n)$