Lezione 6

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
- Vari 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, ..., 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 x_1, x_2, ..., x_k
                    }
  s1= Algoritmo D&I(x_1)
  s2= Algoritmo D&I(x_2)
```

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)
```

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)
```

Quando analizzeremo un generico algoritmo $\mathcal{A}$ progettato in accordo alla tecnica Divide-et-Impera, il numero di operazioni elementari eseguiti dall'algoritmo $\mathcal{A}$ sarà, in generale, esprimibile mediante un'equazione di ricorrenza,

Quando analizzeremo un generico algoritmo $\mathcal{A}$ progettato in accordo alla tecnica Divide-et-Impera, il numero di operazioni elementari eseguiti dall'algoritmo $\mathcal{A}$ sarà, in generale, esprimibile mediante un'equazione di ricorrenza, del tipo:

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq n_0 \\ kT(f(n)) + g(n) & \text{altrimenti} \end{cases}$$

Quando analizzeremo un generico algoritmo $\mathcal{A}$ progettato in accordo alla tecnica Divide-et-Impera, il numero di operazioni elementari eseguiti dall'algoritmo $\mathcal{A}$ sarà, in generale, esprimibile mediante un'equazione di ricorrenza, del tipo:

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq n_0 \\ kT(f(n)) + g(n) & \text{altrimenti} \end{cases}$$

dove

- c_0 è una costante che conta il numero di operazioni elementari eseguiti dall'algoritmo $\mathcal{A}$ quando l'input è di dimensione $\leq n_0$

Quando analizzeremo un generico algoritmo $\mathcal{A}$ progettato in accordo alla tecnica Divide-et-Impera, il numero di operazioni elementari eseguiti dall'algoritmo $\mathcal{A}$ sarà, in generale, esprimibile mediante un'equazione di ricorrenza, del tipo:

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq n_0 \\ kT(f(n)) + g(n) & \text{altrimenti} \end{cases}$$

dove

- c_0 è una costante che conta il numero di operazioni elementari eseguiti dall'algoritmo $\mathcal{A}$ quando l'input è di dimensione $\leq n_0$
- k è il numero di chiamate ricorsive di $\mathcal{A}$

Quando analizzeremo un generico algoritmo $\mathcal{A}$ progettato in accordo alla tecnica Divide-et-Impera, il numero di operazioni elementari eseguiti dall'algoritmo $\mathcal{A}$ sarà, in generale, esprimibile mediante un'equazione di ricorrenza, del tipo:

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq n_0 \\ kT(f(n)) + g(n) & \text{altrimenti} \end{cases}$$

dove

- c_0 è una costante che conta il numero di operazioni elementari eseguiti dall'algoritmo $\mathcal{A}$ quando l'input è di dimensione $\leq n_0$
- k è il numero di chiamate ricorsive di $\mathcal{A}$
- $f(n)$ è (una limitazione superiore al)la dimensione dell'input di ogni chiamata ricorsiva di $\mathcal{A}$

Quando analizzeremo un generico algoritmo $\mathcal{A}$ progettato in accordo alla tecnica Divide-et-Impera, il numero di operazioni elementari eseguiti dall'algoritmo $\mathcal{A}$ sarà, in generale, esprimibile mediante un'equazione di ricorrenza, del tipo:

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq n_0 \\ kT(f(n)) + g(n) & \text{altrimenti} \end{cases}$$

dove

- c_0 è una costante che conta il numero di operazioni elementari eseguiti dall'algoritmo $\mathcal{A}$ quando l'input è di dimensione $\leq n_0$
- k è il numero di chiamate ricorsive di $\mathcal{A}$
- $f(n)$ è (una limitazione superiore al)la dimensione dell'input di ogni chiamata ricorsiva di $\mathcal{A}$
- $g(n)$ è il numero di operazioni elementari eseguiti dall'algoritmo $\mathcal{A}$ al di fuori della ricorsione,

Ricerca Binaria in un array ordinato.

Ricerca Binaria in un array ordinato.

Input: coppia (a, k), dove a è array $a = a[0]a[1] \dots a[n-1]$ di numeri, ordinato in senso non decrescente,

Ricerca Binaria in un array ordinato.

Input: coppia (a, k), dove a è array $a = a[0]a[1] \ldots a[n-1]$ di numeri, ordinato in senso non decrescente, e k è un arbtrario numero.

Ricerca Binaria in un array ordinato.

Input: coppia (a, k), dove a è array $a = a[0]a[1] \dots a[n-1]$ di numeri, ordinato in senso non decrescente, e k è un arbtrario numero.
Output: un valore $i \in \{0, 1, \dots, n-1\}$, se $k = a[i]$,

Ricerca Binaria in un array ordinato.

Input: coppia (a, k), dove a è array $a = a[0]a[1]\ldots a[n-1]$ di numeri, ordinato in senso non decrescente, e k è un arbtrario numero.
Output: un valore $i \in \{0, 1, \ldots, n-1\}$, se $k = a[i]$, ``non c'è'' se $k \neq a[i]$, per ogni $i \in \{0, 1, \ldots, n-1\}$.

Ricerca Binaria in un array ordinato.

Input: coppia (a, k), dove a è array $a = a[0]a[1]\ldots a[n-1]$ di numeri, ordinato in senso non decrescente, e k è un arbtrario numero.
Output: un valore $i \in \{0, 1, \ldots, n-1\}$, se $k = a[i]$, ``non c'è'' se $k \neq a[i]$, per ogni $i \in \{0, 1, \ldots, n-1\}$.

`RicercaBinaria`(a, k, s, d) %[cerca k in $a[s]\ldots a[d]$, con $s \leq d$]

Ricerca Binaria in un array ordinato.

Input: coppia (a, k), dove a è array $a = a[0]a[1]\ldots a[n-1]$ di numeri, ordinato in senso non decrescente, e k è un arbtrario numero.
Output: un valore $i \in \{0, 1, \ldots, n-1\}$, se $k = a[i]$, ``non c'è'' se $k \neq a[i]$, per ogni $i \in \{0, 1, \ldots, n-1\}$.

```
RicercaBinaria(a, k, s, d) %[cerca k in a[s]...a[d], con s ≤ d]
  IF(s == d) {
```

Ricerca Binaria in un array ordinato.

Input: coppia (a, k), dove a è array $a = a[0]a[1]\ldots a[n-1]$ di numeri, ordinato in senso non decrescente, e k è un arbtrario numero.
Output: un valore $i \in \{0, 1, \ldots, n-1\}$, se $k = a[i]$, ``non c'è'' se $k \neq a[i]$, per ogni $i \in \{0, 1, \ldots, n-1\}$.

```
RicercaBinaria(a, k, s, d) %[cerca k in a[s]...a[d], con s ≤ d]
  IF(s == d) {
    IF(k == a[s]) {
```

Ricerca Binaria in un array ordinato.

Input: coppia (a, k), dove a è array $a = a[0]a[1]\ldots a[n-1]$ di numeri, ordinato in senso non decrescente, e k è un arbtrario numero.
Output: un valore $i \in \{0, 1, \ldots, n-1\}$, se $k = a[i]$, ``non c'è'' se $k \neq a[i]$, per ogni $i \in \{0, 1, \ldots, n-1\}$.

```
RicercaBinaria(a, k, s, d) %[cerca k in a[s]...a[d], con s ≤ d]
  IF(s == d) {
    IF(k == a[s]) {
      RETURN(s)
```

Ricerca Binaria in un array ordinato.

Input: coppia (a, k), dove a è array $a = a[0]a[1] \dots a[n-1]$ di numeri, ordinato in senso non decrescente, e k è un arbtrario numero.
Output: un valore $i \in \{0, 1, \dots, n-1\}$, se $k = a[i]$, ``non c'è'' se $k \neq a[i]$, per ogni $i \in \{0, 1, \dots, n-1\}$.

```
RicercaBinaria(a, k, s, d) %[cerca k in a[s]...a[d], con s ≤ d]
  IF(s == d) {
    IF(k == a[s]) {
      RETURN(s)
    } ELSE {
    RETURN ``non c'è''
    }
  }
```

Ricerca Binaria in un array ordinato.

Input: coppia (a, k), dove a è array $a = a[0]a[1] \dots a[n-1]$ di numeri, ordinato in senso non decrescente, e k è un arbtrario numero.
Output: un valore $i \in \{0, 1, \dots, n-1\}$, se $k = a[i]$, ‘‘non c’è’’ se $k \neq a[i]$, per ogni $i \in \{0, 1, \dots, n-1\}$.

```
RicercaBinaria(a, k, s, d) %[cerca k in a[s]...a[d], con s ≤ d]
  IF(s == d) {
    IF(k == a[s]) {
      RETURN(s)
    } ELSE {
    RETURN ‘‘non c’è’’
    }
  }
  c = (s + d)/2
```

Ricerca Binaria in un array ordinato.

Input: coppia (a, k), dove a è array $a = a[0]a[1] \dots a[n-1]$ di numeri, ordinato in senso non decrescente, e k è un arbtrario numero.
Output: un valore $i \in \{0, 1, \dots, n-1\}$, se $k = a[i]$, ‘‘non c’è’’ se $k \neq a[i]$, per ogni $i \in \{0, 1, \dots, n-1\}$.

```
RicercaBinaria(a, k, s, d) %[cerca k in a[s]...a[d], con s ≤ d]
  IF(s == d) {
    IF(k == a[s]) {
      RETURN(s)
    } ELSE {
    RETURN ‘‘non c’è’’
    }
  }
  c = (s + d)/2
  IF (k ≤ a[c]) {
```

Ricerca Binaria in un array ordinato.

Input: coppia (a, k), dove a è array $a = a[0]a[1] \dots a[n-1]$ di numeri, ordinato in senso non decrescente, e k è un arbtrario numero.
Output: un valore $i \in \{0, 1, \dots, n-1\}$, se $k = a[i]$, ``non c'è'' se $k \neq a[i]$, per ogni $i \in \{0, 1, \dots, n-1\}$.

```
RicercaBinaria(a, k, s, d) %[cerca k in a[s]...a[d], con s ≤ d]
  IF(s == d) {
    IF(k == a[s]) {
      RETURN(s)
    } ELSE {
    RETURN ``non c'è''
    }
  }
  c = (s + d)/2
  IF (k ≤ a[c]) {
    RETURN(RicercaBinaria(a, k, s, c)
  }
```

Ricerca Binaria in un array ordinato.

Input: coppia (a, k), dove a è array $a = a[0]a[1] \dots a[n-1]$ di numeri, ordinato in senso non decrescente, e k è un arbtrario numero.
Output: un valore $i \in \{0, 1, \dots, n-1\}$, se $k = a[i]$, ‘‘non c’è’’ se $k \neq a[i]$, per ogni $i \in \{0, 1, \dots, n-1\}$.

```
RicercaBinaria(a, k, s, d) %[cerca k in a[s]...a[d], con s ≤ d]
  IF(s == d) {
    IF(k == a[s]) {
      RETURN(s)
    } ELSE {
    RETURN ‘‘non c’è’’
    }
  }
  c = (s + d)/2
  IF (k ≤ a[c]) {
    RETURN(RicercaBinaria(a, k, s, c)
  } ELSE {
    RETURN(RicercaBinaria(a, k, c + 1, d)
  }
```

Analisi della Ricerca Binaria in un array ordinato.

```
RicercaBinaria(a, k, s, d) %[cerca k in a[s] ... a[d], con s ≤ d]
  IF(s == d) {
    IF(k == a[s]) {
      RETURN(s)
    } ELSE {
    RETURN ‘‘non c’è’’
    }
  }
  c = (s + d)/2
  IF (k ≤ a[c]) {
    RETURN(RicercaBinaria(a, k, s, c)
  } ELSE {
    RETURN(RicercaBinaria(a, k, c + 1, d)    }
```

Analisi della Ricerca Binaria in un array ordinato.

```
RicercaBinaria(a, k, s, d) %[cerca k in a[s] ... a[d], con s ≤ d]
  IF(s == d) {
    IF(k == a[s]) {
      RETURN(s)
    } ELSE {
    RETURN ‘‘non c’è’’
    }
  }
  c = (s + d)/2
  IF (k ≤ a[c]) {
    RETURN(RicercaBinaria(a, k, s, c)
  } ELSE {
    RETURN(RicercaBinaria(a, k, c + 1, d)    }
```

Detta $T(n)$ la complessità di `RicercaBinaria`$(a, k, 0, n-1)$, si ha che

$$T(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T(n/2) + c & \text{altrimenti} \end{cases}$$

Analisi della Ricerca Binaria in un array ordinato.

```
RicercaBinaria(a, k, s, d) %[cerca k in a[s] ... a[d], con s ≤ d]
  IF(s == d) {
    IF(k == a[s]) {
      RETURN(s)
    } ELSE {
    RETURN ‘‘non c’è’’
    }
  }
  c = (s + d)/2
  IF (k ≤ a[c]) {
    RETURN(RicercaBinaria(a, k, s, c)
  } ELSE {
    RETURN(RicercaBinaria(a, k, c + 1, d)    }
```

Detta $T(n)$ la complessità di `RicercaBinaria`$(a, k, 0, n-1)$, si ha che

$$T(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T(n/2) + c & \text{altrimenti} \end{cases}$$

per costanti c_0 e c opportune.

Analisi della Ricerca Binaria in un array ordinato.

```
RicercaBinaria(a, k, s, d) %[cerca k in a[s] ... a[d], con s ≤ d]
  IF(s == d) {
    IF(k == a[s]) {
      RETURN(s)
    } ELSE {
    RETURN ‘‘non c’è’’
    }
  }
  c = (s + d)/2
  IF (k ≤ a[c]) {
    RETURN(RicercaBinaria(a, k, s, c)
  } ELSE {
    RETURN(RicercaBinaria(a, k, c + 1, d)    }
```

Detta $T(n)$ la complessità di `RicercaBinaria`$(a, k, 0, n-1)$, si ha che

$$T(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T(n/2) + c & \text{altrimenti} \end{cases}$$

per costanti c_0 e c opportune. Dai risultati visti nella lezione scorsa, la soluzione dell'equazione di ricorrenza sopra riportata è $T(n) = O(\log n)$.

Un'applicazione della ricerca binaria.

Un'applicazione della ricerca binaria.

Il docente di un corso ha collezionato gli esiti della prova scritta in un vettore $a = a[0]a[1]\ldots a[n-1]$, dove $a[i]$ è il voto in trentesimi riportato dallo studente i-esimo alla prova.

Un'applicazione della ricerca binaria.

Il docente di un corso ha collezionato gli esiti della prova scritta in un vettore $a = a[0]a[1]\ldots a[n-1]$, dove $a[i]$ è il voto in trentesimi riportato dallo studente i-esimo alla prova.
Gli elementi di a sono stati ordinati: $a[0] \leq a[1] \leq \ldots \leq a[n-1]$.

Un'applicazione della ricerca binaria.

Il docente di un corso ha collezionato gli esiti della prova scritta in un vettore $a = a[0]a[1]\ldots a[n-1]$, dove $a[i]$ è il voto in trentesimi riportato dallo studente i-esimo alla prova.
Gli elementi di a sono stati ordinati: $a[0] \leq a[1] \leq \ldots \leq a[n-1]$. Il docente stabilisce che **solo** gli studenti che hanno ottenuto una votazione $> k$ possano sostenere l'esame orale.

Un'applicazione della ricerca binaria.

Il docente di un corso ha collezionato gli esiti della prova scritta in un vettore $a = a[0]a[1]\ldots a[n-1]$, dove $a[i]$ è il voto in trentesimi riportato dallo studente i-esimo alla prova.
Gli elementi di a sono stati ordinati: $a[0] \leq a[1] \leq \ldots \leq a[n-1]$. Il docente stabilisce che **solo** gli studenti che hanno ottenuto una votazione $> k$ possano sostenere l'esame orale. Il Prof. vuole calcolare, dati a e k, quanti sono gli studenti ammessi all'orale.

Un'applicazione della ricerca binaria.

Il docente di un corso ha collezionato gli esiti della prova scritta in un vettore $a = a[0]a[1]\ldots a[n-1]$, dove $a[i]$ è il voto in trentesimi riportato dallo studente i-esimo alla prova.
Gli elementi di a sono stati ordinati: $a[0] \leq a[1] \leq \ldots \leq a[n-1]$. Il docente stabilisce che **solo** gli studenti che hanno ottenuto una votazione $> k$ possano sostenere l'esame orale. Il Prof. vuole calcolare, dati a e k, quanti sono gli studenti ammessi all'orale. Ha quindi il seguente problema algoritmico:
Input: vettore $a = a[0]a[1]\ldots a[n-1]$ tale che $a[0] \leq a[1] \leq \ldots \leq a[n-1]$, ed un intero k.

Un'applicazione della ricerca binaria.

Il docente di un corso ha collezionato gli esiti della prova scritta in un vettore $a = a[0]a[1]\ldots a[n-1]$, dove $a[i]$ è il voto in trentesimi riportato dallo studente i-esimo alla prova.
Gli elementi di a sono stati ordinati: $a[0] \leq a[1] \leq \ldots \leq a[n-1]$. Il docente stabilisce che **solo** gli studenti che hanno ottenuto una votazione $> k$ possano sostenere l'esame orale. Il Prof. vuole calcolare, dati a e k, quanti sono gli studenti ammessi all'orale. Ha quindi il seguente problema algoritmico:
Input: vettore $a = a[0]a[1]\ldots a[n-1]$ tale che $a[0] \leq a[1] \leq \ldots \leq a[n-1]$, ed un intero k.
Output: il numero di elementi in a che hanno valore $> k$.

Un'applicazione della ricerca binaria.

Il docente di un corso ha collezionato gli esiti della prova scritta in un vettore $a = a[0]a[1]\ldots a[n-1]$, dove $a[i]$ è il voto in trentesimi riportato dallo studente i-esimo alla prova.
Gli elementi di a sono stati ordinati: $a[0] \leq a[1] \leq \ldots \leq a[n-1]$. Il docente stabilisce che **solo** gli studenti che hanno ottenuto una votazione $> k$ possano sostenere l'esame orale. Il Prof. vuole calcolare, dati a e k, quanti sono gli studenti ammessi all'orale. Ha quindi il seguente problema algoritmico:
Input: vettore $a = a[0]a[1]\ldots a[n-1]$ tale che $a[0] \leq a[1] \leq \ldots \leq a[n-1]$, ed un intero k.
Output: il numero di elementi in a che hanno valore $> k$.

Il Prof. propone la seguente soluzione al problema: si effettui una scansione di tutto l'array, contando quanti sono gli elementi maggiori di k.

Un'applicazione della ricerca binaria.

Il docente di un corso ha collezionato gli esiti della prova scritta in un vettore $a = a[0]a[1]\ldots a[n-1]$, dove $a[i]$ è il voto in trentesimi riportato dallo studente i-esimo alla prova.
Gli elementi di a sono stati ordinati: $a[0] \leq a[1] \leq \ldots \leq a[n-1]$. Il docente stabilisce che **solo** gli studenti che hanno ottenuto una votazione $> k$ possano sostenere l'esame orale. Il Prof. vuole calcolare, dati a e k, quanti sono gli studenti ammessi all'orale. Ha quindi il seguente problema algoritmico:
Input: vettore $a = a[0]a[1]\ldots a[n-1]$ tale che $a[0] \leq a[1] \leq \ldots \leq a[n-1]$, ed un intero k.
Output: il numero di elementi in a che hanno valore $> k$.

Il Prof. propone la seguente soluzione al problema: si effettui una scansione di tutto l'array, contando quanti sono gli elementi maggiori di k.

```
CONTAPROMOSSI1(a, k)
```

Un'applicazione della ricerca binaria.

Il docente di un corso ha collezionato gli esiti della prova scritta in un vettore $a = a[0]a[1]\ldots a[n-1]$, dove $a[i]$ è il voto in trentesimi riportato dallo studente i-esimo alla prova.
Gli elementi di a sono stati ordinati: $a[0] \leq a[1] \leq \ldots \leq a[n-1]$. Il docente stabilisce che **solo** gli studenti che hanno ottenuto una votazione $> k$ possano sostenere l'esame orale. Il Prof. vuole calcolare, dati a e k, quanti sono gli studenti ammessi all'orale. Ha quindi il seguente problema algoritmico:
Input: vettore $a = a[0]a[1]\ldots a[n-1]$ tale che $a[0] \leq a[1] \leq \ldots \leq a[n-1]$, ed un intero k.
Output: il numero di elementi in a che hanno valore $> k$.

Il Prof. propone la seguente soluzione al problema: si effettui una scansione di tutto l'array, contando quanti sono gli elementi maggiori di k.

```
CONTAPROMOSSI1(a, k)
1. c = 0
```

Un'applicazione della ricerca binaria.

Il docente di un corso ha collezionato gli esiti della prova scritta in un vettore $a = a[0]a[1]\ldots a[n-1]$, dove $a[i]$ è il voto in trentesimi riportato dallo studente i-esimo alla prova.
Gli elementi di a sono stati ordinati: $a[0] \leq a[1] \leq \ldots \leq a[n-1]$. Il docente stabilisce che **solo** gli studenti che hanno ottenuto una votazione $> k$ possano sostenere l'esame orale. Il Prof. vuole calcolare, dati a e k, quanti sono gli studenti ammessi all'orale. Ha quindi il seguente problema algoritmico:
Input: vettore $a = a[0]a[1]\ldots a[n-1]$ tale che $a[0] \leq a[1] \leq \ldots \leq a[n-1]$, ed un intero k.
Output: il numero di elementi in a che hanno valore $> k$.

Il Prof. propone la seguente soluzione al problema: si effettui una scansione di tutto l'array, contando quanti sono gli elementi maggiori di k.

```
CONTAPROMOSSI1(a, k)
1. c = 0
2. FOR(i = 0; i < n; i = i + 1) {
```

Un'applicazione della ricerca binaria.

Il docente di un corso ha collezionato gli esiti della prova scritta in un vettore $a = a[0]a[1]\ldots a[n-1]$, dove $a[i]$ è il voto in trentesimi riportato dallo studente i-esimo alla prova.
Gli elementi di a sono stati ordinati: $a[0] \leq a[1] \leq \ldots \leq a[n-1]$. Il docente stabilisce che **solo** gli studenti che hanno ottenuto una votazione $> k$ possano sostenere l'esame orale. Il Prof. vuole calcolare, dati a e k, quanti sono gli studenti ammessi all'orale. Ha quindi il seguente problema algoritmico:
Input: vettore $a = a[0]a[1]\ldots a[n-1]$ tale che $a[0] \leq a[1] \leq \ldots \leq a[n-1]$, ed un intero k.
Output: il numero di elementi in a che hanno valore $> k$.

Il Prof. propone la seguente soluzione al problema: si effettui una scansione di tutto l'array, contando quanti sono gli elementi maggiori di k.

```
CONTAPROMOSSI1(a, k)
1. c = 0
2. FOR(i = 0; i < n; i = i + 1) {
3.     IF (a[i] > k) { c = c + 1
       }
```

Un'applicazione della ricerca binaria.

Il docente di un corso ha collezionato gli esiti della prova scritta in un vettore $a = a[0]a[1]\ldots a[n-1]$, dove $a[i]$ è il voto in trentesimi riportato dallo studente i-esimo alla prova.
Gli elementi di a sono stati ordinati: $a[0] \leq a[1] \leq \ldots \leq a[n-1]$. Il docente stabilisce che **solo** gli studenti che hanno ottenuto una votazione $> k$ possano sostenere l'esame orale. Il Prof. vuole calcolare, dati a e k, quanti sono gli studenti ammessi all'orale. Ha quindi il seguente problema algoritmico:
Input: vettore $a = a[0]a[1]\ldots a[n-1]$ tale che $a[0] \leq a[1] \leq \ldots \leq a[n-1]$, ed un intero k.
Output: il numero di elementi in a che hanno valore $> k$.

Il Prof. propone la seguente soluzione al problema: si effettui una scansione di tutto l'array, contando quanti sono gli elementi maggiori di k.

```
CONTAPROMOSSI1(a, k)
1. c = 0
2. FOR(i = 0; i < n; i = i + 1) {
3.    IF (a[i] > k) { c = c + 1
        }       }
RETURN c
```

Un'applicazione della ricerca binaria.

Il docente di un corso ha collezionato gli esiti della prova scritta in un vettore $a = a[0]a[1]\ldots a[n-1]$, dove $a[i]$ è il voto in trentesimi riportato dallo studente i-esimo alla prova.
Gli elementi di a sono stati ordinati: $a[0] \leq a[1] \leq \ldots \leq a[n-1]$. Il docente stabilisce che **solo** gli studenti che hanno ottenuto una votazione $> k$ possano sostenere l'esame orale. Il Prof. vuole calcolare, dati a e k, quanti sono gli studenti ammessi all'orale. Ha quindi il seguente problema algoritmico:
Input: vettore $a = a[0]a[1]\ldots a[n-1]$ tale che $a[0] \leq a[1] \leq \ldots \leq a[n-1]$, ed un intero k.
Output: il numero di elementi in a che hanno valore $> k$.

Il Prof. propone la seguente soluzione al problema: si effettui una scansione di tutto l'array, contando quanti sono gli elementi maggiori di k.

```
CONTAPROMOSSI1(a, k)
1. c = 0
2. FOR(i = 0; i < n; i = i + 1) {
3.     IF (a[i] > k) { c = c + 1
       }       }
RETURN c
```

La complessità dell'algoritmo è chiaramente $\Theta(n)$.

Vediamo una soluzione (marginalmente) piu efficiente.

Vediamo una soluzione (marginalmente) piu efficiente. Sfruttando il fatto che l'array a è ordinato, si può evitare la scansione completa e ci si può fermare al primo elemento il cui valore è $> k$;

Vediamo una soluzione (marginalmente) piu efficiente. Sfruttando il fatto che l'array a è ordinato, si può evitare la scansione completa e ci si può fermare al primo elemento il cui valore è $> k$; infatti si sà che tutti gli elementi successivi saranno anch'essi maggiori di k.

```
CONTAPROMOSSI2(a, k)
1. i = 0
2. WHILE ((i < n)&&(a[i] ≤ k)) {
3.         i = i + 1
       }
4. RETURN n − i
```

Vediamo una soluzione (marginalmente) piu efficiente. Sfruttando il fatto che l'array a è ordinato, si può evitare la scansione completa e ci si può fermare al primo elemento il cui valore è $> k$; infatti si sà che tutti gli elementi successivi saranno anch'essi maggiori di k.

```
CONTAPROMOSSI2(a, k)
1. i = 0
2. WHILE ((i < n)&&(a[i] ≤ k)) {
3.         i = i + 1
       }
4. RETURN n − i
```

L'algoritmo abbia complessità $\Theta(n)$ nel caso peggiore,

Vediamo una soluzione (marginalmente) piu efficiente. Sfruttando il fatto che l'array a è ordinato, si può evitare la scansione completa e ci si può fermare al primo elemento il cui valore è $> k$; infatti si sà che tutti gli elementi successivi saranno anch'essi maggiori di k.

```
CONTAPROMOSSI2(a, k)
1. i = 0
2. WHILE ((i < n)&&(a[i] ≤ k)) {
3.         i = i + 1
       }
4. RETURN n − i
```

L'algoritmo abbia complessità $\Theta(n)$ nel caso peggiore, e ciò corrisponde al caso in cui nessuno degli studenti abbia ricevuto un voto $> k$.

Vediamo una soluzione (marginalmente) piu efficiente. Sfruttando il fatto che l'array a è ordinato, si può evitare la scansione completa e ci si può fermare al primo elemento il cui valore è $> k$; infatti si sà che tutti gli elementi successivi saranno anch'essi maggiori di k.

```
CONTAPROMOSSI2(a, k)
1. i = 0
2. WHILE ((i < n)&&(a[i] ≤ k)) {
3.       i = i + 1
     }
4. RETURN n − i
```

L'algoritmo abbia complessità $\Theta(n)$ nel caso peggiore, e ciò corrisponde al caso in cui nessuno degli studenti abbia ricevuto un voto $> k$.

L'algoritmo `CONTAPROMOSSI2` potrebbe terminare anche dopo un solo passo, e ciò nel caso in cui $a[0] > k$. Abbiamo qui un altra situazione in cui il comportamento di un algoritmo nel caso *peggiore* può differire significativamente dal suo comportamento nel caso *migliore*.

La soluzione efficiente è basata sulla ricerca binaria.

La soluzione efficiente è basata sulla ricerca binaria. La funzione `CONTAPROMOSSI3`(a, k, i, j) restituisce il numero degli elementi del sottovettore $a[i] \ldots a[j]$ che risultano maggiori di k (tenendo sempre presente che a è ordinato in senso crescente).

La soluzione efficiente è basata sulla ricerca binaria. La funzione CONTAPROMOSSI3(a, k, i, j) restituisce il numero degli elementi del sottovettore $a[i] \ldots a[j]$ che risultano maggiori di k (tenendo sempre presente che a è ordinato in senso crescente).

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    }
```

La soluzione efficiente è basata sulla ricerca binaria. La funzione CONTAPROMOSSI3(a, k, i, j) restituisce il numero degli elementi del sottovettore $a[i] \ldots a[j]$ che risultano maggiori di k (tenendo sempre presente che a è ordinato in senso crescente).

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.          m = ⌊(i + j)/2)⌋
```

La soluzione efficiente è basata sulla ricerca binaria. La funzione CONTAPROMOSSI3(a, k, i, j) restituisce il numero degli elementi del sottovettore $a[i] \ldots a[j]$ che risultano maggiori di k (tenendo sempre presente che a è ordinato in senso crescente).

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.          m = ⌊(i + j)/2)⌋
5.          IF { (a[m] ≤ k) {
```

La soluzione efficiente è basata sulla ricerca binaria. La funzione CONTAPROMOSSI3(a, k, i, j) restituisce il numero degli elementi del sottovettore $a[i] \ldots a[j]$ che risultano maggiori di k (tenendo sempre presente che a è ordinato in senso crescente).

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.         m = ⌊(i + j)/2)⌋
5.         IF { (a[m] ≤ k) {
6.         RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.              }
```

La soluzione efficiente è basata sulla ricerca binaria. La funzione CONTAPROMOSSI3(a, k, i, j) restituisce il numero degli elementi del sottovettore $a[i] \ldots a[j]$ che risultano maggiori di k (tenendo sempre presente che a è ordinato in senso crescente).

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.         m = ⌊(i + j)/2)⌋
5.         IF { (a[m] ≤ k) {
6.         RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.              } ELSE {
8.           RETURN (j − m + 1)+ CONTAPROMOSSI3(a, k, i, m − 1)
```

La soluzione efficiente è basata sulla ricerca binaria. La funzione CONTAPROMOSSI3(a, k, i, j) restituisce il numero degli elementi del sottovettore $a[i] \ldots a[j]$ che risultano maggiori di k (tenendo sempre presente che a è ordinato in senso crescente).

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.         m = ⌊(i + j)/2)⌋
5.         IF { (a[m] ≤ k) {
6.         RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.              } ELSE {
8.           RETURN (j − m + 1)+ CONTAPROMOSSI3(a, k, i, m − 1)
           }
       }
```

La soluzione efficiente è basata sulla ricerca binaria. La funzione CONTAPROMOSSI3(a, k, i, j) restituisce il numero degli elementi del sottovettore $a[i] \ldots a[j]$ che risultano maggiori di k (tenendo sempre presente che a è ordinato in senso crescente).

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.         m = ⌊(i + j)/2)⌋
5.         IF { (a[m] ≤ k) {
6.         RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.              } ELSE {
8.           RETURN (j − m + 1)+ CONTAPROMOSSI3(a, k, i, m − 1)
           }
       }
```

L'algoritmo viene inizialmente invocato con CONTAPROMOSSI3$(a, k, 0, n-1)$

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.   RETURN 0
3.   } ELSE {
4.       m = ⌊(i + j)/2)⌋
5.       IF { (a[m] ≤ k) {
6.       RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.           } ELSE {
8.         RETURN (j − m + 1)
                + CONTAPROMOSSI3(a, k, i, m − 1)
         }
       }
```

- se $i > j$, allora stiamo effettuando il conteggio sul sottovettore vuoto, in cui il risultato dell'algoritmo e zero;

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.   RETURN 0
3.   } ELSE {
4.        m = ⌊(i + j)/2)⌋
5.        IF { (a[m] ≤ k) {
6.        RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.             } ELSE {
8.          RETURN (j − m + 1)
                  + CONTAPROMOSSI3(a, k, i, m − 1)
          }
       }
```

- se $i > j$, allora stiamo effettuando il conteggio sul sottovettore vuoto, in cui il risultato dell'algoritmo e zero;
- se $i \leq j$ determiniamo la posizione m dell'elemento centrale, come nella ricerca binaria.

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.   RETURN 0
3.   } ELSE {
4.       m = ⌊(i + j)/2)⌋
5.       IF { (a[m] ≤ k) {
6.       RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.           } ELSE {
8.         RETURN (j − m + 1)
                + CONTAPROMOSSI3(a, k, i, m − 1)
         }
      }
```

- se $i > j$, allora stiamo effettuando il conteggio sul sottovettore vuoto, in cui il risultato dell'algoritmo e zero;
- se $i \leq j$ determiniamo la posizione m dell'elemento centrale, come nella ricerca binaria. Distinguiamo due sottocasi:
  - se $a[m] \leq k$, l'elemento centrale è sotto la soglia.

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.         m = ⌊(i + j)/2)⌋
5.         IF { (a[m] ≤ k) {
6.         RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.              } ELSE {
8.           RETURN (j − m + 1)
                    + CONTAPROMOSSI3(a, k, i, m − 1)
           }
       }
```

- se $i > j$, allora stiamo effettuando il conteggio sul sottovettore vuoto, in cui il risultato dell'algoritmo e zero;
- se $i \leq j$ determiniamo la posizione m dell'elemento centrale, come nella ricerca binaria. Distinguiamo due sottocasi:
  - se $a[m] \leq k$, l'elemento centrale è sotto la soglia. Quindi il conteggio prosegue considerando *esclusivamente* gli elementi che stanno a destra di $a[m]$,

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.         m = ⌊(i + j)/2)⌋
5.         IF { (a[m] ≤ k) {
6.         RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.              } ELSE {
8.           RETURN (j − m + 1)
                    + CONTAPROMOSSI3(a, k, i, m − 1)
           }
       }
```

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.         m = ⌊(i + j)/2)⌋
5.         IF { (a[m] ≤ k) {
6.         RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.              } ELSE {
8.           RETURN (j − m + 1)
                    + CONTAPROMOSSI3(a, k, i, m − 1)
           }
       }
```

- se $i > j$, allora stiamo effettuando il conteggio sul sottovettore vuoto, in cui il risultato dell'algoritmo e zero;
- se $i \leq j$ determiniamo la posizione m dell'elemento centrale, come nella ricerca binaria. Distinguiamo due sottocasi:
  - se $a[m] \leq k$, l'elemento centrale è sotto la soglia. Quindi il conteggio prosegue considerando *esclusivamente* gli elementi che stanno a destra di $a[m]$, ossia quelli nel sottovettore $a[m+1] \ldots a[j]$

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.        m = ⌊(i + j)/2)⌋
5.        IF { (a[m] ≤ k) {
6.        RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.             } ELSE {
8.          RETURN (j − m + 1)
                  + CONTAPROMOSSI3(a, k, i, m − 1)
          }
       }
```

- se $i > j$, allora stiamo effettuando il conteggio sul sottovettore vuoto, in cui il risultato dell'algoritmo e zero;
- se $i \leq j$ determiniamo la posizione m dell'elemento centrale, come nella ricerca binaria. Distinguiamo due sottocasi:
  - se $a[m] \leq k$, l'elemento centrale è sotto la soglia. Quindi il conteggio prosegue considerando *esclusivamente* gli elementi che stanno a destra di $a[m]$, ossia quelli nel sottovettore $a[m+1] \ldots a[j]$
  - se $a[m] > k$, l'elemento centrale è sopra la soglia.

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.         m = ⌊(i + j)/2)⌋
5.         IF { (a[m] ≤ k) {
6.         RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.              } ELSE {
8.           RETURN (j − m + 1)
                    + CONTAPROMOSSI3(a, k, i, m − 1)
           }
       }
```

- se $i > j$, allora stiamo effettuando il conteggio sul sottovettore vuoto, in cui il risultato dell'algoritmo e zero;
- se $i \leq j$ determiniamo la posizione m dell'elemento centrale, come nella ricerca binaria. Distinguiamo due sottocasi:
  - se $a[m] \leq k$, l'elemento centrale è sotto la soglia. Quindi il conteggio prosegue considerando *esclusivamente* gli elementi che stanno a destra di $a[m]$, ossia quelli nel sottovettore $a[m+1] \ldots a[j]$
  - se $a[m] > k$, l'elemento centrale è sopra la soglia. Quindi, essendo l'array ordinato, tutti gli $(j - m + 1)$ elementi in $a[m] \ldots a[j]$ risultano *sopra* la soglia.

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.         m = ⌊(i + j)/2)⌋
5.         IF { (a[m] ≤ k) {
6.         RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.              } ELSE {
8.           RETURN (j − m + 1)
                    + CONTAPROMOSSI3(a, k, i, m − 1)
           }
       }
```

- se $i > j$, allora stiamo effettuando il conteggio sul sottovettore vuoto, in cui il risultato dell'algoritmo e zero;
- se $i \leq j$ determiniamo la posizione m dell'elemento centrale, come nella ricerca binaria. Distinguiamo due sottocasi:
  - se $a[m] \leq k$, l'elemento centrale è sotto la soglia. Quindi il conteggio prosegue considerando *esclusivamente* gli elementi che stanno a destra di $a[m]$, ossia quelli nel sottovettore $a[m+1] \ldots a[j]$
  - se $a[m] > k$, l'elemento centrale è sopra la soglia. Quindi, essendo l'array ordinato, tutti gli $(j - m + 1)$ elementi in $a[m] \ldots a[j]$ risultano *sopra* la soglia. Il numero di tali elementi, sommato al conteggio di quelli sopra soglia in $a[i] \ldots a[m-1]$

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.         m = ⌊(i + j)/2)⌋
5.         IF { (a[m] ≤ k) {
6.         RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.              } ELSE {
8.           RETURN (j − m + 1)
                    + CONTAPROMOSSI3(a, k, i, m − 1)
           }
       }
```

- se $i > j$, allora stiamo effettuando il conteggio sul sottovettore vuoto, in cui il risultato dell'algoritmo e zero;
- se $i \leq j$ determiniamo la posizione m dell'elemento centrale, come nella ricerca binaria. Distinguiamo due sottocasi:
  - se $a[m] \leq k$, l'elemento centrale è sotto la soglia. Quindi il conteggio prosegue considerando *esclusivamente* gli elementi che stanno a destra di $a[m]$, ossia quelli nel sottovettore $a[m+1] \ldots a[j]$
  - se $a[m] > k$, l'elemento centrale è sopra la soglia. Quindi, essendo l'array ordinato, tutti gli $(j - m + 1)$ elementi in $a[m] \ldots a[j]$ risultano *sopra* la soglia. Il numero di tali elementi, sommato al conteggio di quelli sopra soglia in $a[i] \ldots a[m-1]$ (che viene calcolato dalla chiamata ricorsiva) produce il risultato.

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.        m = ⌊(i + j)/2)⌋
5.        IF { (a[m] ≤ k) {
6.        RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.             } ELSE {
8.          RETURN (j − m + 1)
                  + CONTAPROMOSSI3(a, k, i, m − 1)
          }
       }
```

- se $i > j$, allora stiamo effettuando il conteggio sul sottovettore vuoto, in cui il risultato dell'algoritmo e zero;
- se $i \leq j$ determiniamo la posizione m dell'elemento centrale, come nella ricerca binaria. Distinguiamo due sottocasi:
  - se $a[m] \leq k$, l'elemento centrale è sotto la soglia. Quindi il conteggio prosegue considerando *esclusivamente* gli elementi che stanno a destra di $a[m]$, ossia quelli nel sottovettore $a[m+1] \ldots a[j]$
  - se $a[m] > k$, l'elemento centrale è sopra la soglia. Quindi, essendo l'array ordinato, tutti gli $(j - m + 1)$ elementi in $a[m] \ldots a[j]$ risultano *sopra* la soglia. Il numero di tali elementi, sommato al conteggio di quelli sopra soglia in $a[i] \ldots a[m-1]$ (che viene calcolato dalla chiamata ricorsiva) produce il risultato.

Sia $T(n)$ la complessità dell'algoritmo CONTAPROMOSSI3$(a, k, 0, n - 1)$,

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.        m = ⌊(i + j)/2)⌋
5.        IF { (a[m] ≤ k) {
6.        RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.             } ELSE {
8.          RETURN (j − m + 1)
                  + CONTAPROMOSSI3(a, k, i, m − 1)
          }
       }
```

- se $i > j$, allora stiamo effettuando il conteggio sul sottovettore vuoto, in cui il risultato dell'algoritmo e zero;
- se $i \leq j$ determiniamo la posizione m dell'elemento centrale, come nella ricerca binaria. Distinguiamo due sottocasi:
  - se $a[m] \leq k$, l'elemento centrale è sotto la soglia. Quindi il conteggio prosegue considerando *esclusivamente* gli elementi che stanno a destra di $a[m]$, ossia quelli nel sottovettore $a[m+1] \ldots a[j]$
  - se $a[m] > k$, l'elemento centrale è sopra la soglia. Quindi, essendo l'array ordinato, tutti gli $(j - m + 1)$ elementi in $a[m] \ldots a[j]$ risultano *sopra* la soglia. Il numero di tali elementi, sommato al conteggio di quelli sopra soglia in $a[i] \ldots a[m-1]$ (che viene calcolato dalla chiamata ricorsiva) produce il risultato.

Sia $T(n)$ la complessità dell'algoritmo CONTAPROMOSSI3$(a, k, 0, n-1)$, essa soddisfa la equazione di ricorrenza

$$T(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T(n/2) + c & \text{altrimenti} \end{cases}$$

per costanti c_0 e c opportune.

```
CONTAPROMOSSI3(a, k, i, j)
1. IF (i > j) {
2.    RETURN 0
3.    } ELSE {
4.        m = ⌊(i + j)/2)⌋
5.        IF { (a[m] ≤ k) {
6.        RETURN CONTAPROMOSSI3(a, k, m + 1, j)
7.             } ELSE {
8.          RETURN (j − m + 1)
                  + CONTAPROMOSSI3(a, k, i, m − 1)
          }
       }
```

- se $i > j$, allora stiamo effettuando il conteggio sul sottovettore vuoto, in cui il risultato dell'algoritmo e zero;
- se $i \leq j$ determiniamo la posizione m dell'elemento centrale, come nella ricerca binaria. Distinguiamo due sottocasi:
  - se $a[m] \leq k$, l'elemento centrale è sotto la soglia. Quindi il conteggio prosegue considerando *esclusivamente* gli elementi che stanno a destra di $a[m]$, ossia quelli nel sottovettore $a[m+1] \ldots a[j]$
  - se $a[m] > k$, l'elemento centrale è sopra la soglia. Quindi, essendo l'array ordinato, tutti gli $(j - m + 1)$ elementi in $a[m] \ldots a[j]$ risultano *sopra* la soglia. Il numero di tali elementi, sommato al conteggio di quelli sopra soglia in $a[i] \ldots a[m-1]$ (che viene calcolato dalla chiamata ricorsiva) produce il risultato.

Sia $T(n)$ la complessità dell'algoritmo CONTAPROMOSSI3$(a, k, 0, n-1)$, essa soddisfa la equazione di ricorrenza

$$T(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T(n/2) + c & \text{altrimenti} \end{cases}$$

per costanti c_0 e c opportune. La soluzione dell'equazione di ricorrenza è $T(n) = O(\log n)$

Problema: Dato un numero a ed un intero positivo n, calcolare a^n usando il minor numero di moltiplicazioni.

Problema: Dato un numero a ed un intero positivo n, calcolare a^n usando il minor numero di moltiplicazioni.

Algoritmo semplice

`SlowPower`(a, n)

Problema: Dato un numero a ed un intero positivo n, calcolare a^n usando il minor numero di moltiplicazioni.

Algoritmo semplice

```
SlowPower(a, n)
  x = a
```

Problema: Dato un numero a ed un intero positivo n, calcolare a^n usando il minor numero di moltiplicazioni.

Algoritmo semplice

```
SlowPower(a, n)
  x = a
  FOR(i = 2; i < n + 1; i = i + 1){
```

Problema: Dato un numero a ed un intero positivo n, calcolare a^n usando il minor numero di moltiplicazioni.

Algoritmo semplice

```
SlowPower(a, n)
  x = a
  FOR(i = 2; i < n + 1; i = i + 1){
    x = x × a
    }
```

Problema: Dato un numero a ed un intero positivo n, calcolare a^n usando il minor numero di moltiplicazioni.

Algoritmo semplice

```
SlowPower(a, n)
  x = a
  FOR(i = 2; i < n + 1; i = i + 1){
    x = x × a
    }
  RETURN(x)
```

Problema: Dato un numero a ed un intero positivo n, calcolare a^n usando il minor numero di moltiplicazioni.

Algoritmo semplice

```
SlowPower(a, n)
  x = a
  FOR(i = 2; i < n + 1; i = i + 1){
    x = x × a
    }
  RETURN(x)
```

Il numero di moltiplicazioni effettuate da `SlowPower`(a, n) è chiaramente pari a $n - 1$.

Per applicare Divide-et-Impera osserviamo che

$$a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}.$$

Per applicare Divide-et-Impera osserviamo che

$$a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}.$$

Inoltre, $a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor}$ se n é pari,

Per applicare Divide-et-Impera osserviamo che

$$a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}.$$

Inoltre, $a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor}$ se n é pari, e $a^{\lceil n/2 \rceil} = a \times a^{\lfloor n/2 \rfloor}$ se n é dispari.

Per applicare Divide-et-Impera osserviamo che

$$a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}.$$

Inoltre, $a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor}$ se n é pari, e $a^{\lceil n/2 \rceil} = a \times a^{\lfloor n/2 \rfloor}$ se n é dispari.
Sulla base di queste osservazioni, possiamo progettare il seguente algoritmo.

`FastPower`(a, n)

Per applicare Divide-et-Impera osserviamo che

$$a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}.$$

Inoltre, $a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor}$ se n é pari, e $a^{\lceil n/2 \rceil} = a \times a^{\lfloor n/2 \rfloor}$ se n é dispari.
Sulla base di queste osservazioni, possiamo progettare il seguente algoritmo.

```
FastPower(a, n)
  IF(n == 1) {
    RETURN(a)
    }
```

Per applicare Divide-et-Impera osserviamo che

$$a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}.$$

Inoltre, $a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor}$ se n é pari, e $a^{\lceil n/2 \rceil} = a \times a^{\lfloor n/2 \rfloor}$ se n é dispari.
Sulla base di queste osservazioni, possiamo progettare il seguente algoritmo.

```
FastPower(a, n)
  IF(n == 1) {
    RETURN(a)
    } ELSE {
      y = FastPower(a, ⌊n/2⌋)
```

Per applicare Divide-et-Impera osserviamo che

$$a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}.$$

Inoltre, $a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor}$ se n é pari, e $a^{\lceil n/2 \rceil} = a \times a^{\lfloor n/2 \rfloor}$ se n é dispari.
Sulla base di queste osservazioni, possiamo progettare il seguente algoritmo.

```
FastPower(a, n)
  IF(n == 1) {
    RETURN(a)
    } ELSE {
      y = FastPower(a, ⌊n/2⌋)
        IF (n é pari) RETURN(y × y)
```

Per applicare Divide-et-Impera osserviamo che

$$a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}.$$

Inoltre, $a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor}$ se n é pari, e $a^{\lceil n/2 \rceil} = a \times a^{\lfloor n/2 \rfloor}$ se n é dispari.
Sulla base di queste osservazioni, possiamo progettare il seguente algoritmo.

```
FastPower(a, n)
  IF(n == 1) {
    RETURN(a)
    } ELSE {
      y = FastPower(a, ⌊n/2⌋)
        IF (n é pari) RETURN(y × y)
          ELSE RETURN(y × y × a)
      }
```

Per applicare Divide-et-Impera osserviamo che

$$a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}.$$

Inoltre, $a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor}$ se n é pari, e $a^{\lceil n/2 \rceil} = a \times a^{\lfloor n/2 \rfloor}$ se n é dispari.
Sulla base di queste osservazioni, possiamo progettare il seguente algoritmo.

```
FastPower(a, n)
  IF(n == 1) {
    RETURN(a)
    } ELSE {
      y = FastPower(a, ⌊n/2⌋)
        IF (n é pari) RETURN(y × y)
          ELSE RETURN(y × y × a)
     }
```

Sia $T(n)$ il numero di moltiplicazioni effettuate dall'algoritmo `FastPower`(a, n).

Per applicare Divide-et-Impera osserviamo che

$$a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}.$$

Inoltre, $a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor}$ se n é pari, e $a^{\lceil n/2 \rceil} = a \times a^{\lfloor n/2 \rfloor}$ se n é dispari.
Sulla base di queste osservazioni, possiamo progettare il seguente algoritmo.

```
FastPower(a, n)
  IF(n == 1) {
    RETURN(a)
    } ELSE {
      y = FastPower(a, ⌊n/2⌋)
        IF (n é pari) RETURN(y × y)
          ELSE RETURN(y × y × a)
    }
```

Sia $T(n)$ il numero di moltiplicazioni effettuate dall'algoritmo `FastPower`(a, n).
É ovvio che

$$T(n) \leq \begin{cases} 0 & \text{se } n = 1 \\ T(n/2) + 2 & \text{altrimenti} \end{cases}$$

Per applicare Divide-et-Impera osserviamo che

$$a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}.$$

Inoltre, $a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor}$ se n é pari, e $a^{\lceil n/2 \rceil} = a \times a^{\lfloor n/2 \rfloor}$ se n é dispari.
Sulla base di queste osservazioni, possiamo progettare il seguente algoritmo.

```
FastPower(a, n)
  IF(n == 1) {
    RETURN(a)
    } ELSE {
      y = FastPower(a, ⌊n/2⌋)
        IF (n é pari) RETURN(y × y)
          ELSE RETURN(y × y × a)
     }
```

Sia $T(n)$ il numero di moltiplicazioni effettuate dall'algoritmo `FastPower`(a, n).
É ovvio che

$$T(n) \leq \begin{cases} 0 & \text{se } n = 1 \\ T(n/2) + 2 & \text{altrimenti} \end{cases}$$

da cui ne discende che $T(n) = O(\log n)$, migliorando considerevolmente rispetto all'algoritmo `SlowPower`.

Consideriamo ora il seguente problema.
Input: sequenza di interi $a = a[0]a[1] \dots a[n-1]$

Consideriamo ora il seguente problema.
Input: sequenza di interi $a = a[0]a[1]\ldots a[n-1]$
Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.

Consideriamo ora il seguente problema.
Input: sequenza di interi $a = a[0]a[1] \dots a[n-1]$
Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.
Scriviamo un algoritmo `CoppieIdentiche`(a, i, j) che restituisce il numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$ (che chiameremo coppie identiche) nella sottosequenza $a[i] \dots a[j]$.

Consideriamo ora il seguente problema.
Input: sequenza di interi $a = a[0]a[1] \dots a[n-1]$
Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.
Scriviamo un algoritmo `CoppieIdentiche`(a, i, j) che restituisce il numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$ (che chiameremo coppie identiche) nella sottosequenza $a[i] \dots a[j]$. Per risolvere il problema di partenza chiameremo l'algoritmo `CoppieIdentiche`$(a, 0, n-1)$

`CoppieIdentiche`(a, i, j)

Consideriamo ora il seguente problema.
Input: sequenza di interi $a = a[0]a[1] \dots a[n-1]$
Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.
Scriviamo un algoritmo `CoppieIdentiche`(a, i, j) che restituisce il numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$ (che chiameremo coppie identiche) nella sottosequenza $a[i] \dots a[j]$. Per risolvere il problema di partenza chiameremo l'algoritmo `CoppieIdentiche`$(a, 0, n-1)$

```
CoppieIdentiche(a, i, j)
  IF(j − i == 1) {
    IF (a[i] = a[i + 1]) {RETURN(1)
```

Consideriamo ora il seguente problema.
Input: sequenza di interi $a = a[0]a[1] \dots a[n-1]$
Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.
Scriviamo un algoritmo `CoppieIdentiche`(a, i, j) che restituisce il numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$ (che chiameremo coppie identiche) nella sottosequenza $a[i] \dots a[j]$. Per risolvere il problema di partenza chiameremo l'algoritmo `CoppieIdentiche`$(a, 0, n-1)$

```
CoppieIdentiche(a, i, j)
  IF(j − i == 1) {
    IF (a[i] = a[i + 1]) {RETURN(1)
                          ELSE RETURN(0)
        }
```

Consideriamo ora il seguente problema.
Input: sequenza di interi $a = a[0]a[1] \dots a[n-1]$
Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.
Scriviamo un algoritmo `CoppieIdentiche`(a, i, j) che restituisce il numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$ (che chiameremo coppie identiche) nella sottosequenza $a[i] \dots a[j]$. Per risolvere il problema di partenza chiameremo l'algoritmo `CoppieIdentiche`$(a, 0, n-1)$

```
CoppieIdentiche(a, i, j)
  IF(j − i == 1) {
    IF (a[i] = a[i + 1]) {RETURN(1)
                          ELSE RETURN(0)
        } ELSE k = ⌊(i + j)/2⌋
```

Consideriamo ora il seguente problema.
Input: sequenza di interi $a = a[0]a[1]\ldots a[n-1]$
Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.
Scriviamo un algoritmo $\texttt{CoppieIdentiche}(a, i, j)$ che restituisce il numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$ (che chiameremo coppie identiche) nella sottosequenza $a[i]\ldots a[j]$. Per risolvere il problema di partenza chiameremo l'algoritmo $\texttt{CoppieIdentiche}(a, 0, n-1)$

```
CoppieIdentiche(a, i, j)
  IF(j − i == 1) {
    IF (a[i] = a[i + 1]) {RETURN(1)
                          ELSE RETURN(0)
        } ELSE k = ⌊(i + j)/2⌋

RETURN(CoppieIdentiche(a, i, k)+CoppieIdentiche(a, k + 1, j))
```

Consideriamo ora il seguente problema.
Input: sequenza di interi $a = a[0]a[1] \dots a[n-1]$
Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.
Scriviamo un algoritmo `CoppieIdentiche`(a, i, j) che restituisce il numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$ (che chiameremo coppie identiche) nella sottosequenza $a[i] \dots a[j]$. Per risolvere il problema di partenza chiameremo l'algoritmo `CoppieIdentiche`$(a, 0, n-1)$

```
CoppieIdentiche(a, i, j)
  IF(j − i == 1) {
    IF (a[i] = a[i + 1]) {RETURN(1)
                          ELSE RETURN(0)
        } ELSE k = ⌊(i + j)/2⌋

RETURN(CoppieIdentiche(a, i, k)+CoppieIdentiche(a, k + 1, j))
```

L'algoritmo non è corretto,

Consideriamo ora il seguente problema.
Input: sequenza di interi $a = a[0]a[1] \dots a[n-1]$
Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.
Scriviamo un algoritmo `CoppieIdentiche`(a, i, j) che restituisce il numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$ (che chiameremo coppie identiche) nella sottosequenza $a[i] \dots a[j]$. Per risolvere il problema di partenza chiameremo l'algoritmo `CoppieIdentiche`$(a, 0, n-1)$

```
CoppieIdentiche(a, i, j)
  IF(j − i == 1) {
    IF (a[i] = a[i + 1]) {RETURN(1)
                          ELSE RETURN(0)
        } ELSE k = ⌊(i + j)/2⌋

RETURN(CoppieIdentiche(a, i, k)+CoppieIdentiche(a, k + 1, j))
```

L'algoritmo non è corretto, perchè?

Consideriamo ora il seguente problema.
Input: sequenza di interi $a = a[0]a[1] \dots a[n-1]$
Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.
Scriviamo un algoritmo `CoppieIdentiche`(a, i, j) che restituisce il numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$ (che chiameremo coppie identiche) nella sottosequenza $a[i] \dots a[j]$. Per risolvere il problema di partenza chiameremo l'algoritmo `CoppieIdentiche`$(a, 0, n-1)$

```
CoppieIdentiche(a, i, j)
  IF(j − i == 1) {
    IF (a[i] = a[i + 1]) {RETURN(1)
                          ELSE RETURN(0)
        } ELSE k = ⌊(i + j)/2⌋

RETURN(CoppieIdentiche(a, i, k)+CoppieIdentiche(a, k + 1, j))
```

L'algoritmo non è corretto, perchè?

Questa è la versione corretta.
Input: sequenza di interi $a = a[0]a[1]\ldots a[n-1]$
Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.

```
CoppieIdentiche(a, i, j)
  IF(j − i == 1) {
    IF (a[i] = a[i + 1]) {RETURN(1)
                          ELSE RETURN(0)
        } ELSE k = ⌊(i + j)/2⌋

RETURN(CoppieIdentiche(a, i, k)+CoppieIdentiche(a, k, j))
```

Questa è la versione corretta.
Input: sequenza di interi $a = a[0]a[1]\ldots a[n-1]$
Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.

```
CoppieIdentiche(a, i, j)
  IF(j − i == 1) {
    IF (a[i] = a[i + 1]) {RETURN(1)
                          ELSE RETURN(0)
        } ELSE k = ⌊(i + j)/2⌋

RETURN(CoppieIdentiche(a, i, k)+CoppieIdentiche(a, k, j))
```

Sia $T(n)$ il numero di operazioni effettuate dall'algoritmo `CoppieIdentiche`$(a, 0, n-1)$.

Questa è la versione corretta.
Input: sequenza di interi $a = a[0]a[1]\ldots a[n-1]$
Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.

```
CoppieIdentiche(a, i, j)
  IF(j − i == 1) {
    IF (a[i] = a[i + 1]) {RETURN(1)
                          ELSE RETURN(0)
        } ELSE k = ⌊(i + j)/2⌋

RETURN(CoppieIdentiche(a, i, k)+CoppieIdentiche(a, k, j))
```

Sia $T(n)$ il numero di operazioni effettuate dall'algoritmo `CoppieIdentiche`$(a, 0, n-1)$. É ovvio che

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 2 \\ 2T(n/2) + c & \text{altrimenti} \end{cases}$$

per opportune costanti c_0 e c,

Questa è la versione corretta.
Input: sequenza di interi $a = a[0]a[1]\ldots a[n-1]$
Output: numero di coppie $(a[i], a[i+1])$ per cui $a[i] = a[i+1])$.

```
CoppieIdentiche(a, i, j)
  IF(j − i == 1) {
    IF (a[i] = a[i + 1]) {RETURN(1)
                          ELSE RETURN(0)
        } ELSE k = ⌊(i + j)/2⌋

RETURN(CoppieIdentiche(a, i, k)+CoppieIdentiche(a, k, j))
```

Sia $T(n)$ il numero di operazioni effettuate dall'algoritmo `CoppieIdentiche`$(a, 0, n-1)$. É ovvio che

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq 2 \\ 2T(n/2) + c & \text{altrimenti} \end{cases}$$

per opportune costanti c_0 e c, da cui ne discende che $T(n) = O(n)$.

Input: numero intero $x \geq 0$

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

- $\lfloor \sqrt{0} \rfloor = 0$

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

- $\lfloor \sqrt{0} \rfloor = 0$
- $\lfloor \sqrt{1} \rfloor = 1$

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

- $\lfloor \sqrt{0} \rfloor = 0$
- $\lfloor \sqrt{1} \rfloor = 1$
- $\lfloor \sqrt{2} \rfloor = 1$

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

- $\lfloor \sqrt{0} \rfloor = 0$
- $\lfloor \sqrt{1} \rfloor = 1$
- $\lfloor \sqrt{2} \rfloor = 1$
- $\lfloor \sqrt{3} \rfloor = 1$

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

- $\lfloor \sqrt{0} \rfloor = 0$
- $\lfloor \sqrt{1} \rfloor = 1$
- $\lfloor \sqrt{2} \rfloor = 1$
- $\lfloor \sqrt{3} \rfloor = 1$
- $\lfloor \sqrt{4} \rfloor = 2$

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

- $\lfloor \sqrt{0} \rfloor = 0$
- $\lfloor \sqrt{1} \rfloor = 1$
- $\lfloor \sqrt{2} \rfloor = 1$
- $\lfloor \sqrt{3} \rfloor = 1$
- $\lfloor \sqrt{4} \rfloor = 2$
- $\lfloor \sqrt{5} \rfloor = 2$
- $\lfloor \sqrt{6} \rfloor = 2$
- $\lfloor \sqrt{7} \rfloor = 2$
- $\lfloor \sqrt{8} \rfloor = 2$

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

- $\lfloor \sqrt{0} \rfloor = 0$
- $\lfloor \sqrt{1} \rfloor = 1$
- $\lfloor \sqrt{2} \rfloor = 1$
- $\lfloor \sqrt{3} \rfloor = 1$
- $\lfloor \sqrt{4} \rfloor = 2$
- $\lfloor \sqrt{5} \rfloor = 2$
- $\lfloor \sqrt{6} \rfloor = 2$
- $\lfloor \sqrt{7} \rfloor = 2$
- $\lfloor \sqrt{8} \rfloor = 2$
- $\lfloor \sqrt{9} \rfloor = 3$

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

- $\lfloor \sqrt{0} \rfloor = 0$
- $\lfloor \sqrt{1} \rfloor = 1$
- $\lfloor \sqrt{2} \rfloor = 1$
- $\lfloor \sqrt{3} \rfloor = 1$
- $\lfloor \sqrt{4} \rfloor = 2$
- $\lfloor \sqrt{5} \rfloor = 2$
- $\lfloor \sqrt{6} \rfloor = 2$
- $\lfloor \sqrt{7} \rfloor = 2$
- $\lfloor \sqrt{8} \rfloor = 2$
- $\lfloor \sqrt{9} \rfloor = 3$
- $\lfloor \sqrt{10} \rfloor = 3$
- $\lfloor \sqrt{11} \rfloor = 3$
- $\lfloor \sqrt{12} \rfloor = 3$
- $\lfloor \sqrt{13} \rfloor = 3$
- $\lfloor \sqrt{14} \rfloor = 3$
- $\lfloor \sqrt{15} \rfloor = 3$

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

- $\lfloor \sqrt{0} \rfloor = 0$
- $\lfloor \sqrt{1} \rfloor = 1$
- $\lfloor \sqrt{2} \rfloor = 1$
- $\lfloor \sqrt{3} \rfloor = 1$
- $\lfloor \sqrt{4} \rfloor = 2$
- $\lfloor \sqrt{5} \rfloor = 2$
- $\lfloor \sqrt{6} \rfloor = 2$
- $\lfloor \sqrt{7} \rfloor = 2$
- $\lfloor \sqrt{8} \rfloor = 2$
- $\lfloor \sqrt{9} \rfloor = 3$
- $\lfloor \sqrt{10} \rfloor = 3$
- $\lfloor \sqrt{11} \rfloor = 3$
- $\lfloor \sqrt{12} \rfloor = 3$
- $\lfloor \sqrt{13} \rfloor = 3$
- $\lfloor \sqrt{14} \rfloor = 3$
- $\lfloor \sqrt{15} \rfloor = 3$
- $\lfloor \sqrt{16} \rfloor = 4$

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

```
Radice(x)
  IF (x == 0||x == 1)
```

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

```
Radice(x)
  IF (x == 0||x == 1)
    RETURN(x)
```

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

```
Radice(x)
  IF (x == 0||x == 1)
    RETURN(x)
  s = 1, d = x/2
```

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

```
Radice(x)
  IF (x == 0||x == 1)
    RETURN(x)
  s = 1, d = x/2
  WHILE (s ≤ d) {
```

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

```
Radice(x)
  IF (x == 0||x == 1)
    RETURN(x)
  s = 1, d = x/2
  WHILE (s ≤ d) {
    m = (s + d)/2
```

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

```
Radice(x)
  IF (x == 0||x == 1)
    RETURN(x)
  s = 1, d = x/2
  WHILE (s ≤ d) {
    m = (s + d)/2
    IF (m × m == x)
```

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

```
Radice(x)
  IF (x == 0||x == 1)
    RETURN(x)
  s = 1, d = x/2
  WHILE (s ≤ d) {
    m = (s + d)/2
    IF (m × m == x) {
    RETURN(m)
```

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

```
Radice(x)
  IF (x == 0||x == 1)
    RETURN(x)
  s = 1, d = x/2
  WHILE (s ≤ d) {
    m = (s + d)/2
    IF (m × m == x) {
    RETURN(m)
  }
    IF (m × m < x) {
```

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

```
Radice(x)
  IF (x == 0||x == 1)
    RETURN(x)
  s = 1, d = x/2
  WHILE (s ≤ d) {
    m = (s + d)/2
    IF (m × m == x) {
    RETURN(m)
  }
    IF (m × m < x) {
    s = m + 1,
```

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

```
Radice(x)
  IF (x == 0||x == 1)
    RETURN(x)
  s = 1, d = x/2
  WHILE (s ≤ d) {
    m = (s + d)/2
    IF (m × m == x) {
    RETURN(m)
  }
    IF (m × m < x) {
    s = m + 1, r = m
```

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

```
Radice(x)
  IF (x == 0||x == 1)
    RETURN(x)
  s = 1, d = x/2
  WHILE (s ≤ d) {
    m = (s + d)/2
    IF (m × m == x) {
    RETURN(m)
  }
    IF (m × m < x) {
    s = m + 1, r = m
    } ELSE{d = m − 1
```

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

```
Radice(x)
  IF (x == 0||x == 1)
    RETURN(x)
  s = 1, d = x/2
  WHILE (s ≤ d) {
    m = (s + d)/2
    IF (m × m == x) {
    RETURN(m)
  }
    IF (m × m < x) {
    s = m + 1, r = m
    } ELSE{d = m − 1
    }
  }
  RETURN(r)
```

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

```
Radice(x)
  IF (x == 0||x == 1)
    RETURN(x)
  s = 1, d = x/2
  WHILE (s ≤ d) {
    m = (s + d)/2
    IF (m × m == x) {
    RETURN(m)
  }
    IF (m × m < x) {
    s = m + 1, r = m
    } ELSE{d = m − 1
    }
  }
  RETURN(r)
```

Complessità: $T(n) = T(n/2) + c$

Input: numero intero $x \geq 0$
Output: $\lfloor \sqrt{x} \rfloor$

```
Radice(x)
  IF (x == 0||x == 1)
    RETURN(x)
  s = 1, d = x/2
  WHILE (s ≤ d) {
    m = (s + d)/2
    IF (m × m == x) {
    RETURN(m)
  }
    IF (m × m < x) {
    s = m + 1, r = m
    } ELSE{d = m − 1
    }
  }
  RETURN(r)
```

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