Relazioni di ricorrenza

Data una funzione ricorsiva, come calcoliamo l'ordine di grandezza?
Esempio:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{altrimenti} \end{cases}$$

con tempo computazionale:

$$T(n) = \begin{cases} c & \text{se } n = 0 \\ T(n-1) + d & \text{altrimenti} \end{cases}$$

Dunque un algoritmo ricorsivo ha un tempo computazionale **ricorsivo**

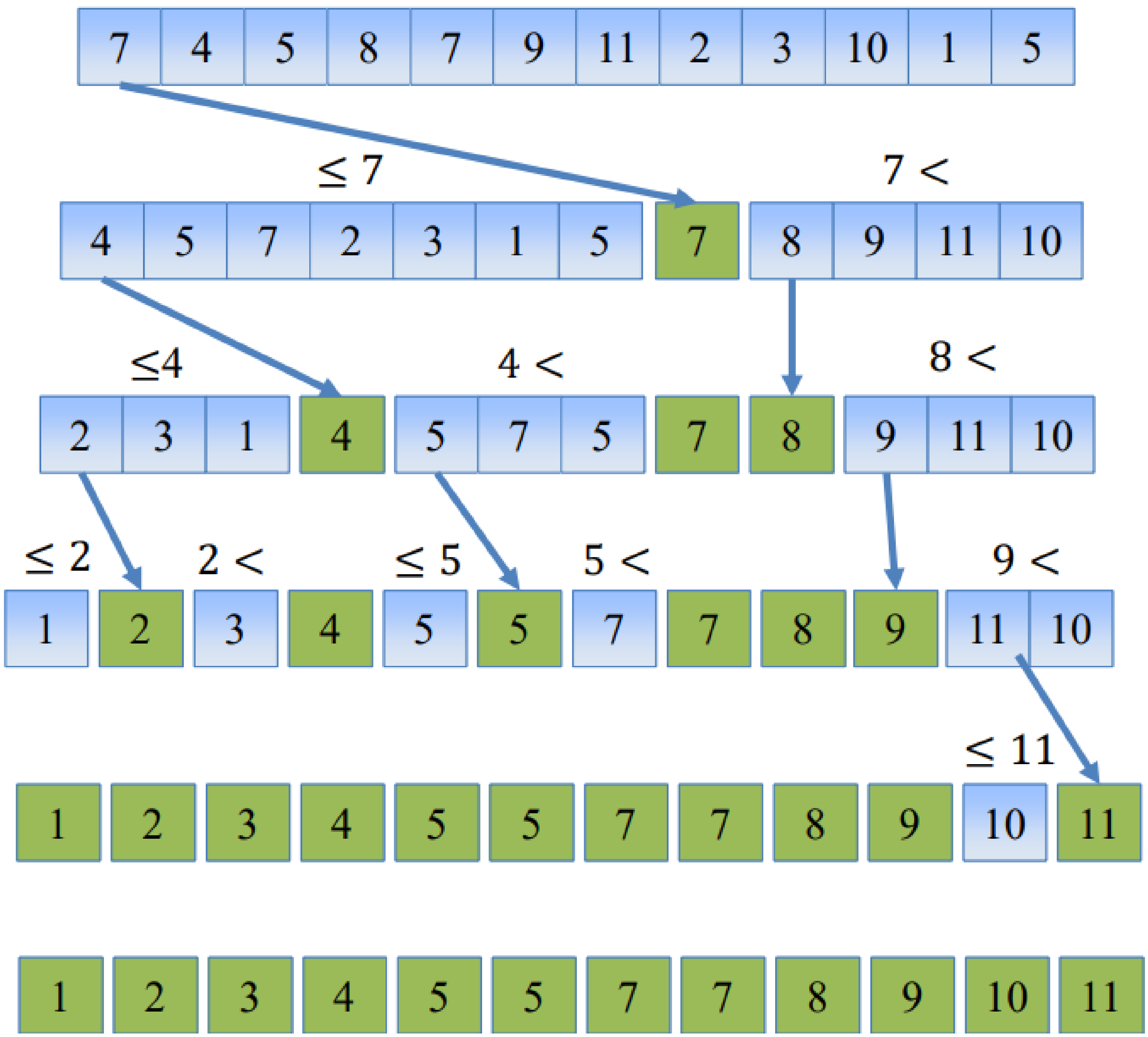
Però qual è esattamente l'ordine di grandezza di $T(n)$?

- **Metodo dell'iterazione:**
  Applichiamo ripetutamente la relazione di ricorrenza fino a trovare la soluzione.
  $T(n) = T(n-1) + d = T(n-2) + d + d = \ldots = T(n-k) + kd = T(0 + nd) = c +$
  con $k \leq n$ scegliendo $k = n$
- **Metodo di sostituzione:**
  Ipotizziamo inizialmente una soluzione e, induttiviamente, la verifichiamo
  Soluzione ipotizzata: $T(n) = c + nd$
  Caso base: $c + 0 \cdot d = c = T(0)$
  Passo induttivo: dimostriamo che
  - $T(n) = c + nd \rightarrow T(n+1) + c + (n+1)d$
  - Secondo la relazione di ricorrenza:
    $T(n+1) = T(n) + d =$
  - Utilizzando l'ipotesi induttiva:
    $= c + nd + d = c + (n+1)d$

Quick-Sort

Idea di base dato A[1..n]:

- se $n \leq 1$ il caso è banale e non faccio niente(vettore ordinato)
- scelgo un elemento del vettore q, chiamato **perno**
- riorganizzo il vettore in modo che
  - gli elementi a sinistra di q siano $\leq q$
  - gli elementi a destra di q siano $\geq q$
    - **Attenzione:** gli elementi a sinistra e a destra possono essere anche in disordine, basta soltanto che siano $\leq$(o $\geq$) di q
- ciò implica che q sia al posto giusto
- sia p la posizione di q, abbiamo dunque: $A[1..p-1] \leq A[p] \leq A[p+1..n]$
- reitero il procedimento su $A[1..p-1]$ e $A[p+1..n]$



Funzionamento del Quick-Sort con partizionamento generico

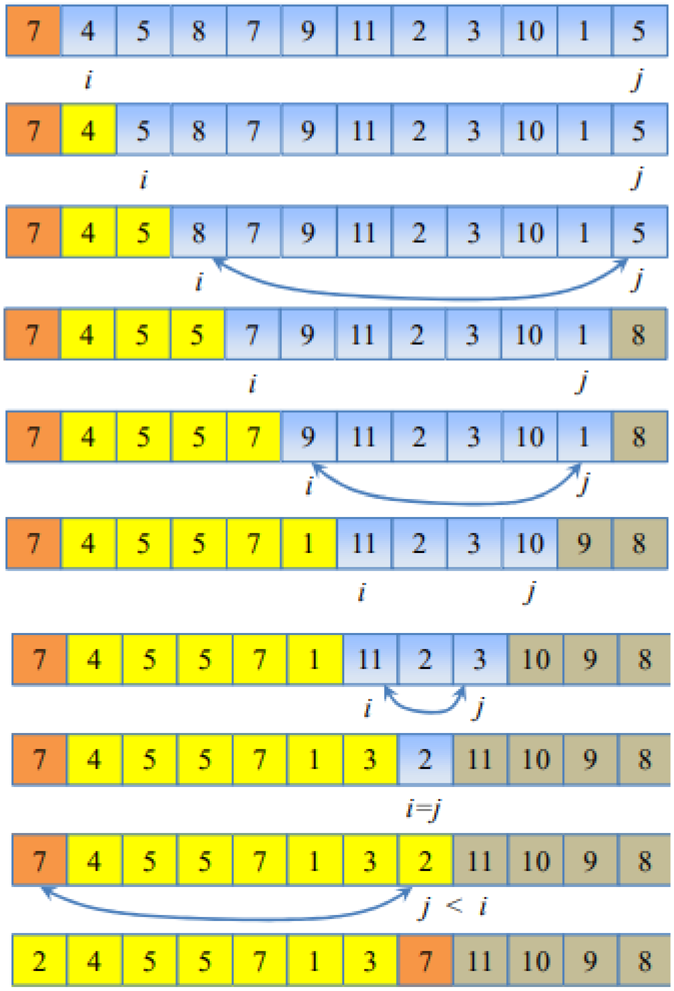
Nota bene! Con A[1..n] si intende dal primo elemento(A[0] all'ultimo elemento (A[n-1])

Partizionamento del Quick-Sort

Partizionamento in $A[1..n]$*(Attenzione alla nota sopra!)*

- Scegliamo come elemento **perno** $A[1]$
- Scegliamo due **indici di partizionamento** i e j tali che:
  - Gli elementi in $A[2..i-1]$ sono già esaminati e $A[2..i-1] \leq A[1]$
  - Gli elementi in $A[i..j]$ sono elemento ancora da esaminare
  - Gli elementi in $A[j+1..n]$ sono già esaminati e $A[1] < A[j+1..n]$
- Scegliamo $i = 2$ e $j = n$
- i e j si spostano, se $i \leq j$, nel seguente modo:
  - se $A[i] \leq A[1]$ incremento i
  - se $A[i] > A[1] \wedge A[j] > A[1]$ decremento j
  - se $A[i] > A[1] \wedge A[j] \leq A[1]$ scambio $A[i]$ e $A[j]$ e incremento i e decremento j
- Quando per la prima volta $i > j$, scambio $A[1]$ con $A[j]$

In questo modo,arriverò a un punto dove gli elementi prima di i sono minori del perno e gli elementi dopo di j sono maggiori del perno. Quando i "supera" j scambio il perno con $A[j]$ così da render vera la prima fase



Esempio di funzionamento di partizionamento nel quicksort

Correttezza del partizionamento

```
PARTITION(A[1..n])
i ← 2, j ← n
while i ≤ j do
    if A[i] ≤ A[1] then
        i ← i + 1
    else
        if A[j] > A[1] then
            j ← j − 1
        else
            scambia A[i] con A[j]
            i ← i + 1, j ← j − 1
        end if
    end if
end while
scambia A[1] con A[j]
return j
```

Invariante di ciclo: $A[2..i-1] \leq A[1] \wedge A[1] < A[j+1..n]$

Ovvero tutti gli elementi prima di i sono minori dell'elemento perno e tutti gli elementi dopo j sono maggiori dell'elemento perno

Inizializzazione:
Con $i = 2$ e $j = n$ i due sottovettori sono vuoti. Dunque è vero

Mantenimento:
Il ciclo viene eseguito finchè $i \leq j$ ed effettua **soltanto** una tra le seguenti operazioni:

- se $A[i] \leq A[1]$ incremento i altrimenti
- se $A[1] < A[j]$ decremento j altrimenti
- scambio $A[i]$ e $A[j]$, incremento i, decremento j

ed è evidente(visto che esegue soltanto una di quelle operazioni) che il ciclo viene **mantenuto**

Correttezza del Quick Sort

```
QUICK-SORT(A[1..n])
if n > 1 then
    p ← PARTITION(A[1..n])       ▷ partizionamento porta il perno nella posizione p
    if p > 2 then                ▷ se prima del perno ci sono almeno 2 elementi
        QUICK-SORT(A[1..p − 1])
    end if
    if p < n − 1 then            ▷ se dopo il perno ci sono almeno 2 elementi
        QUICK-SORT(A[p + 1..n])
    end if
end if
```

Essendo un algoritmo **ricorsivo**, per dimostrare la sua correttezza bisogna usare una induzione completa, assumendo che il partizionamento funzioni correttamente.

Caso base: con $n \leq 1$ il vettore in input è **ordinato**
Passo induttivo:
HP: Corretto con dimensione $< n$ ⇒ Corretto con dimensione $= n$

- Dopo aver effettuato la partizione: $A[1..p-1] \leq A[p] < A[p+1..n]$
- Dall'ipotesi induttiva:
  - Se $A[1..p-1]$ ha più di 1 elemento, sarà ordinato correttamente alla prima chiamata ricorsiva di Quick-Sort perchè ha meno di n elementi
  - Se $A[p+1..n]$ ha più di 1 elemento, sarà ordinato correttamente alla

prima chiamata ricorsiva di Quick-Sort perchè ha meno di n elementi
$A[1..n]$ è dunque ordinato

Complessità del Quick-Sort

- **Partizionamento:**
  Il partizionamento scansiona una volta il vettore su cui opera.
  Nel caso peggiore, $T_p(n) \in O(n)$ infatti $T_p(n) = an$

- Per quanto riguarda il **Quick-Sort**:
  Dopo aver partizionato i vettori, le due chiamate ricorsive lavorano su $p-1$ elementi e $n-p$ elementi

- Le due situazioni estreme sono:
  - $A[1..p-1]$ e $A[p+1..n]$ hanno circa lo stesso numero di elementi
  - $A[1..p-1]$ ha $n-1$ elementi e $A[p+1..n]$ è vuoto(o viceversa)

La seconda casistica dà luogo alla seguente relazione di ricorrenza:

$$T(n) = \begin{cases} c & n = 1 \\ T(n-1) + T_P(n) + b & n > 1 \end{cases} = \begin{cases} c & n = 1 \\ T(n-1) + an + b & n > 1 \end{cases}$$

Relazioni di ricorrenza a partizione costante

Il quick-sort, come altri algoritmi che abbiamo visto, vengono definiti da una relazione ricorrenza a **partizione costante.**
Esiste inoltre un teorema, detto **Teorema master per relazioni lineari a partizioni costanti** che ci fornisce una formula per calcolare "facilmente" il tempo di complessita rispetto a O :

$$T(n) = \begin{cases} d & \text{se } n \leq m \leq h \\ \sum_{1 \leq i \leq h} a_i T(n-i) + cn^{\beta} & \text{se } n > m \end{cases}$$

Teorema. Se $c > 0, \beta \geq 0, a = \sum_{1 \leq i \leq h} a_i$ allora $\begin{cases} T(n) \in O(n^{\beta+1}) & \text{se } a = 1 \\ T(n) \in O(a^n n^{\beta}) & \text{se } a \geq 2 \end{cases}$

dove:

- aè una **costante** e indica il **numero di volte** che viene effettuata la chiamata ricorsiva dentro il codice(nel caso del **quick-sort**, a dovrebbe valere 2 ma invece vale 1 visto che una chiamata si può trascurare visto che ha complessita $O(1)$)
- β è l'ordine di complessità della funzione "ausiliaria"(nel caso del quick-sort, β si riferisce alla complessità del partizionamento,quindi 1 visto che il partizionamento è **lineare**)
- c è una costante moltiplicativa legata alla funzione ausiliaria sopra descritta

Viene detto a **partizione costante** perchè l'input diminuisce costantemente e NON in funzione di n ($T(n-i)$)

Relazioni di ricorrenza Divide et Impera

"Per meglio dominare occorre dividere gli avversari"
Ossia: suddividi il problema in sottoproblemi di dimensione circa uguale; risolvi i sottoproblemi in maniera ricorsiva, infine combina i risultati

$$T(n) = \begin{cases} d & \text{se } n \le k \\ D(n) + C(n) + \sum_{i=1}^{h} T(n_i) & \text{se } n > k \end{cases}$$

Tempo per dividere

Tempo per combinare

Somma dei tempi dei sottoproblemi

DI Min Max

```
DI-Min-Max (A, p, q)
1  if p = q then return (A[p], A[p])
2  if p = q – 1 then
3        if A[p] < A[q] then return (A[p], A[q])
4        else return (A[q], A[p])
5  r ← (p + q)/2
6  (min1, max1) ← DI-Min-Max (A, p, r)
7  (min2, max2) ← DI-Min-Max (A, r +1, q)
8  return (min (min1, min2), max(max1, max2))
```

Questo algoritmo calcola il **minimo** e il **massimo** in $A[p..q]$
Le **prime due righe** sono casi banali:

- **Riga 1:** Se il vettore contiene un solo elemento, restituisco semplicemente l'unico elemento disponibile
- **Riga 2:** Se il vettore contiene due elementi, li restituisco entrambi, facendo attenzione alla loro relazione di $< o >$

Se il vettore contiene più di due elementi, divido in due sottovettori il vettore originale, e richiamo il metodo su essi(**Divide et Impera**)

Infine calcolo il minimo tra i valori minimi dei due sottovettori
Stessa cosa per il valore massimo

Relazione di ricorrenza di DI Min-Max

Se $n = q - p + 1$ allora i confronti $C(n)$ sono:

$$C(n) = \begin{cases} 0 & \text{se } n = 1 \\ 1 & \text{se } n = 2 \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + 2 & \text{se } n > 2 \end{cases}$$

La **relazione di ricorrenza** è definita nel seguente modo:

- Se il vettore ha 0 elementi, banalmente effettuo 0 confronti
- Se il vettore ha 2 elementi, banalmente effettuo 1 confronto(per capire il valore minimo e il valore massimo).
- Invece se il vettore ha $n > 2$ elementi, effettuo 2 confronti più i confronti necessari per ogni sottovettore($C(n/2) + C(n/2)$)

Ponendo $n = 2^k$ per semplificarci i conti, sviluppando la relazione di ricorrenza otteniamo:

$$C(n) = 2C\left(\frac{n}{2}\right) + 2 = 2\left(2C\left(\frac{n}{4}\right) + 2\right) + 2 = 4C\left(\frac{n}{4}\right) + 4 + 2 =$$

$$= 4\left(2C\left(\frac{n}{8}\right) + 2\right) + 4 + 2 = 8C\left(\frac{n}{8}\right) + 8 + 4 + 2 = \cdots$$

$$= 2^j C\left(\frac{n}{2^j}\right) + 2^j + 2^{j-1} + \cdots + 2 =$$

$$\text{con } \frac{n}{2^j} = 2 \Rightarrow j = \log_2 n - 1$$

$$C(n) = 2^{\log_2 n - 1} C\left(\frac{n}{2^{\log_2 n - 1}}\right) + 2^{\log_2 n - 1} + 2^{\log_2 n - 2} + \cdots + 2 =$$

$$C(n) = 2^{\log_2 n-1} C\left(\frac{n}{2^{\log_2 n-1}}\right) + 2^{\log_2 n-1} + 2^{\log_2 n-2} + \cdots + 2 =$$
$$= \frac{n}{2} C(2) + \frac{n}{2} + \frac{n}{4} + \cdots + 2 = \frac{n}{2} + \left(\frac{n}{2} + \frac{n}{4} + \cdots + 2 + 1\right) - 1 =$$
$$= \frac{n}{2} + \left(2^{k-1} + 2^{k-2} + \cdots + 2 + 1\right) - 1 =$$
$$= \frac{n}{2} + n - 1 - 1 = \frac{3}{2} n - 2$$

Ora possiamo dunque, applicando la relazione iterativamente, a dimostrarla con l'**induzione:**

Caso base: $C(2) = \frac{3}{2} 2 - 2 = 3 - 2 = 1$ **verificato!**
Passo induttivo: $C(\frac{n}{2}) = \frac{3}{2} \cdot \frac{n}{2} - 2 \Rightarrow C(n) = \frac{3}{2} - 2$
$C(n) = 2C(\frac{n}{2}) + 2 = \ldots = \frac{3}{2} n - 2$

Binary Search DI

```
BinSearch-Ric(x, A, i, j)
    ▷ Pre: A[i..j] ordinato
    ▷ Post: true se x ∈ A[i..j]
if i > j then        ▷ A[i..j] = ∅
    return false
else
    m ← ⌊(i + j)/2⌋
    if x = A[m] then
        return true
    else
        if x < A[m] then
            return BinSearch-Ric(x, A, i, m − 1)
        else      ▷ A[m] < x
            return BinSearch-Ric(x, A, m + 1, j)
        end if
    end if
end if
```

$$T(n) = \begin{cases} c & \text{se } n \leq 1 \\ T\left(\frac{n}{2}\right) + d & \text{altrimenti} \end{cases}$$

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + d \\ &= T\left(\frac{n}{4}\right) + d + d = T\left(\frac{n}{4}\right) + 2d \\ &\dots \\ &= T\left(\frac{n}{2^k}\right) + dk \quad \text{se} \quad k \leq \log_2 n \end{aligned}$$

con $k = \log_2 n$ (assumiamo n sia potenza di 2)

$$= T(1) + d \cdot \log_2 n = c + d \cdot \log_2 n$$

quindi

$T(n) \in \Theta(\log n)$

(con $T(n) = T(n/b) + d$ viene lo stesso ordine di grandezza)

Relazioni di ricorrenza a partizione bilanciata

Esiste il **Teorema master per le relazioni di ricorrenza a partizione bilanciata**:

Teorema. Se $a \geq 1; b \geq 2; c > 0; d, \beta \geq 0$, posto $\alpha = \log a / \log b$ allora:

$$\begin{cases} T(n) \in O(n^{\alpha}) & \text{se } \alpha > \beta \\ T(n) \in O(n^{\alpha} \log n) & \text{se } \alpha = \beta \\ T(n) \in O(n^{\beta}) & \text{se } \alpha < \beta \end{cases}$$

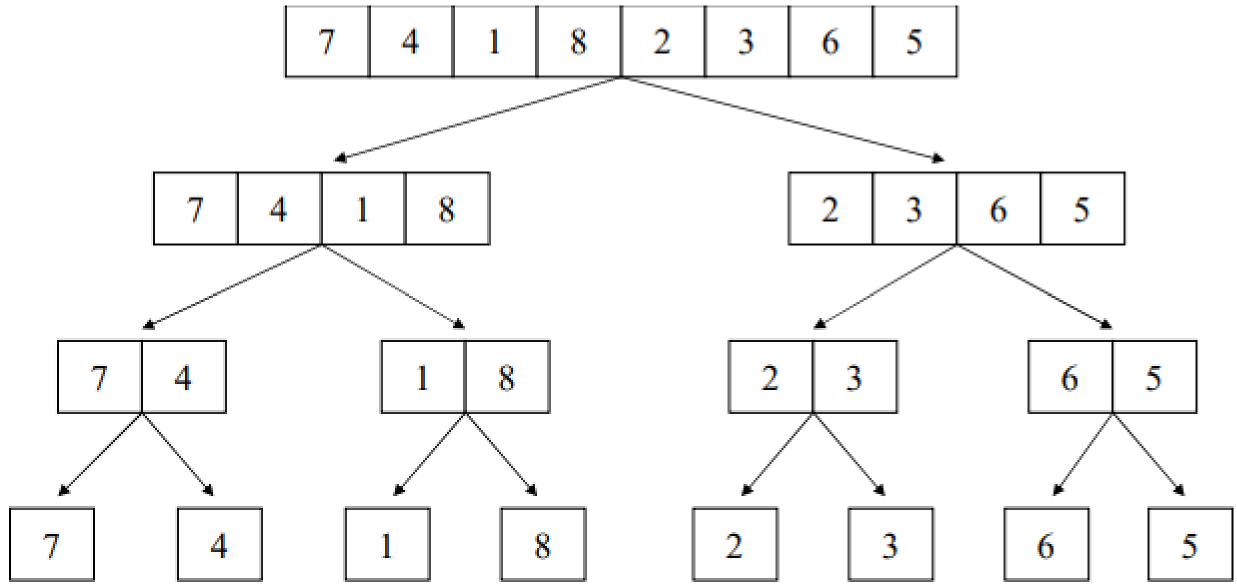
che ci permette di calcolare "facilmente" la complessità di un algoritmo in funzione di O.
Viene detta a partizione bilanciata perchè l'input diminuisce in funzione di n, come nella seguente(generica) relazione di ricorrenza:

$$T(n) = \begin{cases} d & \text{se } n = 1 \\ aT(n/b) + cn^{\beta} & \text{se } n > 1 \end{cases}$$

dove l'input diminuisce di n/b ogni volta.

Algoritmo per fusione: Merge-Sort

Anche quest'algoritmo si basa sul principio Divide et Impera.
Idea di base:
Consiste nel **dividere** l'array in due ogni volta, finchè non arriviamo ad avere array **composti da un singolo elemento.**

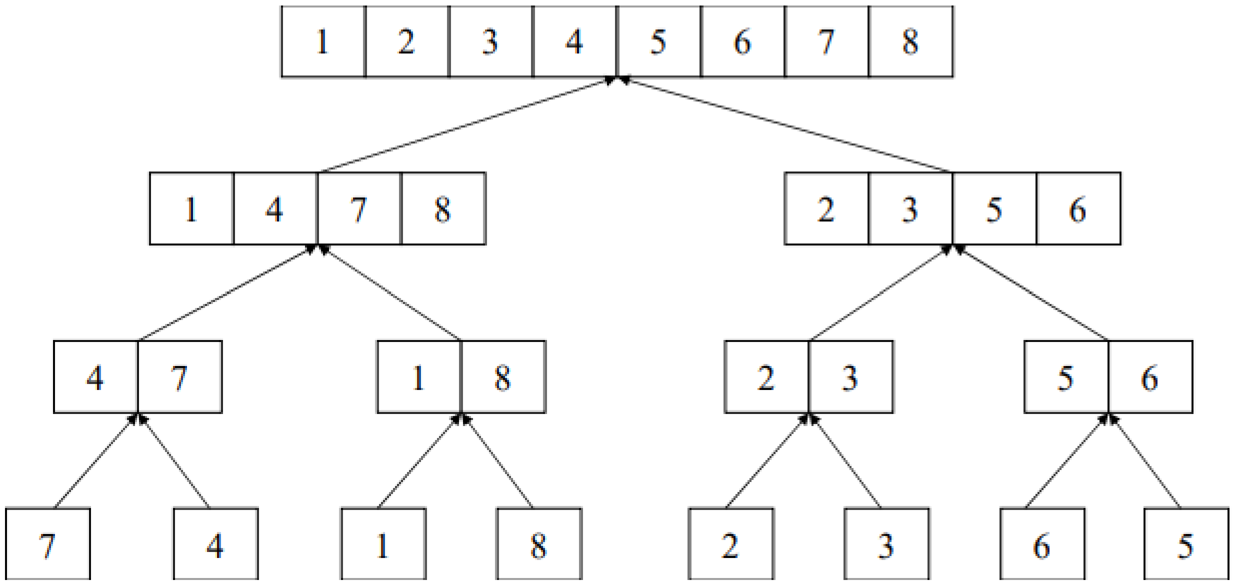


Array scomposto in sottovettori atomici(composti da un solo elemento)

Una volta che abbiamo l'array scomposto in sottosequenze di singoli elementi, prendiamo questi a coppia di due in due e li **fondiamo**(da questo deriva il nome **Merge**).
Ora abbiamo molteplici array composti da due elementi ciascuno.
Reiteriamo questo procedimento finchè non abbiamo il nostro array originale ordinato.

Abbiamo i due sottovettori A e B e il vettore C ordinato che vogliamo restituire

- Se $A[0] \leq B[0]$
  - $C[0] = A[0]$
  - Controllo il prossimo elemento di A
- Se $A[0] > B[0]$
  - $C[0] = B[0]$
  - Controllo il prossimo elemento di B



Fondo i sottovettori(all'inizio atomici), a coppia di due in due, in vettori ordinati.
Reiterando questo processo, arrivo ad ottenere l'array originale ordinato

Codice del Merge-Sort

$\textsc{Merge-Sort}(A)$
if $length(A) = 1$ **then**
    return A
else
    $k \leftarrow \lfloor length(A)/2 \rfloor$
    $B \leftarrow \textsc{Merge-Sort}(A[1..k])$
    $C \leftarrow \textsc{Merge-Sort}(A[k+1..length(A)])$
    return $\textsc{Merge}(B, C)$
end if

Con A array da ordinare:

- Se la lunghezza del vettore A è 1, significa che è un vettore atomico, e quindi, banalmente, è già ordinato e lo restituisco
- Altrimenti procedo a dividere A in due sottoparti ordinate
  e infine le fondo ordinatamente.

Funzione ausiliaria di **Merge**:

$$
\begin{array}{l}
\textsc{Merge}(B, C) \\
\textbf{if } B = [] \textbf{ then} \\
\quad \textbf{return } C \\
\textbf{else} \\
\quad \textbf{if } C = [] \textbf{ then} \\
\quad\quad \textbf{return } B \\
\quad \textbf{else} \\
\quad\quad \textbf{if } B[1] \leq C[1] \textbf{ then} \\
\quad\quad\quad \textbf{return } [B[1], \textsc{Merge}(B[2..length(B)], C)] \\
\quad\quad \textbf{else} \\
\quad\quad\quad \textbf{return } [C[1], \textsc{Merge}(B, C[2..length(C)])] \\
\quad\quad \textbf{end if} \\
\quad \textbf{end if} \\
\textbf{end if}
\end{array}
$$

Con B e C due sottovettori da ordinare:

- Se uno dei due sottovettori è vuoto, semplicemente ritorno l'altro non vuoto
- Altrimenti
  - Se il primo elemento di B è $\leq$ di C, ritorno un sottovettore il cui primo elemento è $B[0]$($B[1]$ in figura) seguito dalla fusione dei restanti sottovettori(questa volta considero B a partire dal secondo elemento)
  - Altrimenti, analogamente, ritorno un sottovettore il cui primo elemento è $C[0]$ seguito dalla fusione dei restanti sottovettori(questa volta considero C a partire dal secondo elemento)

Tempo computazione del Merge-Sort

Il tempo computazionale del **Merge-Sort** può esser definito dalla seguente relazione di ricorrenza:

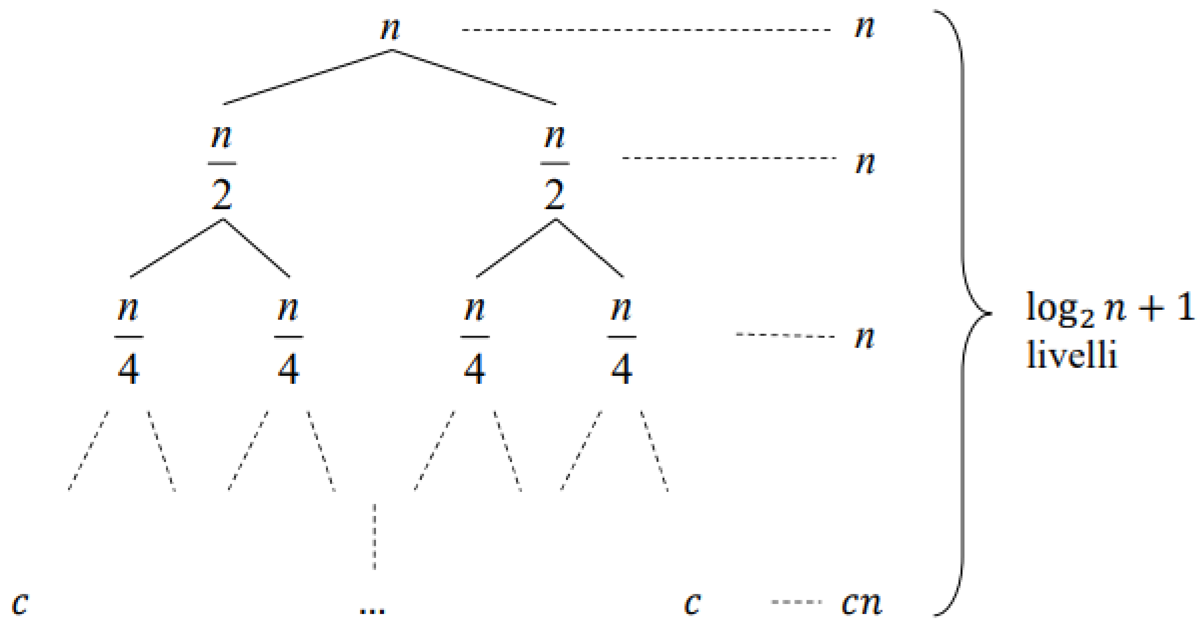
$$T(n) = 2T(n/2) + T_{\text{Merge}}(n)$$

Infatti il suo **costo computazionale** su un array lungo n ($T(n)$) è dato da due volte il costo su un vettore lungo la metà $2T(n/2)$ più il costo computazionale della funzione ausiliaria di **Merge**
Considerando la versione iterativa della funzione ausiliaria di **Merge:**
```
function merge (a[], left, center, right)
   i ← left
   j ← center + 1
   k ← 0
   b ← array temp size= right-left+1

   while i ≤ center and j ≤ right do
      if a[i] ≤ a[j] then
         b[k] ← a[i]
         i ← i + 1
      else
         b[k] ← a[j]
         j ← j + 1
      k ← k + 1
   end while

   while i ≤ center do
      b[k] ← a[i]
      i ← i + 1
      k ← k + 1
   end while

   while j ≤ right do
      b[k] ← a[j]
      j ← j + 1
      k ← k + 1
   end while

   for k ← left to right do
      a[k] ← b[k-left]
```
Possiamo facilmente notare che $T_{Merge}(n) \in \Theta(n)$

Dunque:

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

Analizzando l'albero di ricorsione è evidente che il tempo computazionale totale è $\in \Theta(n \cdot \log n)$



Ricollegandoci al teorema master per le partizioni bilanciate:

$$T(n) = \begin{cases} d & \text{se } n = 1 \\ aT(n/b) + cn^{\beta} & \text{se } n > 1 \end{cases}$$

Merge Sort

$$T(n) = 2T(n/2) + cn$$

$$a = b = 2, \alpha = \log 2 / \log 2 = 1 = \beta$$

$$\boxed{\begin{aligned} T(n) &\in O(n^{\alpha} \log n) \\ &= O(n \log n) \end{aligned}}$$

Caso medio del Quick Sort

Prima di tutto, cosa si intende per **Caso Medio?**
Per **Caso Medio** si intende che prendiamo in considerazione ogni possibile **input** con la stessa probabilità.

Nel caso del Quick-Sort, significa prendere in considerazione ogni possibile **posizione** dell'elemento **pivot**.
Ciò è dato dalla seguente relazione di ricorrenza:

$$T(n) = \begin{cases} a & \text{se } n \leq 1 \\ \frac{1}{n} \sum_{k=1}^{n} (T(k-1) + T(n-k)) + bn + c & \text{altrimenti} \end{cases}$$

Attraverso *varie* manipolazioni, arriviamo ad ottenere che
$T(n) \in O(n \cdot \log n)$