Divide et impera (Divide and Conquer)

Dividi il problema in sottoproblemi piu` semplici e risolvili ricorsivamente

Divide et impera - Schema generale

Divide-et-impera (P, n)

if $n \leq k$ **then** "risolvi direttamente"

else "dividi P in sottoproblemi $P_1, P_2, \ldots, P_h$
di dimensione $n_1, n_2, \ldots, n_h$ risp."

for $i \leftarrow 1$ **to** h **do**

Divide-et-impera (P_i, n_i)

"combina i risultati di $P_1, \ldots, P_h$
per ottenere quello di P"

Divide et impera - Complessità

Il costo dell'algoritmo è di solito costante per i casi in cui la soluzione viene costruita direttamente; per i casi in cui non è immediata, il costo può essere espresso come somma di h + 2 costi:

- il costo di **dividere** il problema nei sottoproblemi
- il costo di **combinare** i risultati dei sottoproblemi per ottenere il risultato del problema
- gli h costi delle **soluzioni** dei sottoproblemi

$$
\begin{aligned}
T(n) &= c && n \leq k \\
&= D(n) + C(n) + \Sigma_{i=1..h}\, T(n_i) && n > k
\end{aligned}
$$

Tre modi per trovare la soluzione della relazione di ricorrenza :

- metodo di sostituzione
- metodo iterativo
- metodo principale

Il metodo di sostituzione

- Ipotizza un limite asintotico
- verifica l'ipotesi con una dimostrazione per induzione

esempio: $T(n) = 4T(n/2) + n$

proviamo **$T(n) = O(n^3)$**

mostrando che esistono c ed n_0 :

$$T(n) \leq c\, n^3 \text{ per tutti gli } n \geq n_0$$

Supponendo $T(k) \leq c\, k^3$ per tutti i $k < n$, si ottiene

$$\begin{aligned} T(n) &= 4T(n/2) + n \\ &\leq 4c(n/2)^3 + n \\ &= cn^3 - (c/2\, n^3 - n) \\ &\leq cn^3 \text{ per } c \geq 2 \text{ e } n \geq 1 \end{aligned}$$

Anche $T(1)$ deve essere $\leq c\, 1^3$

un tentativo migliore permette comunque di dimostrare:

$T(n) = O(n^2)$

Dimostriamo che la ricorrenza

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

appartiene all'insieme $\mathbf{O}(n \lg n)$

Proviamo $T(n) \leq c\, n \lg n$

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n && (\lfloor n/2 \rfloor \leq n/2) \\ &\leq 2\,(c\,(n/2)\lg(n/2)) + n \\ &= c\,n \lg n - c\,n \lg 2 + n \\ &= c\,n \lg n - c\,n + n \\ &\leq c\,n \lg n && \text{per } c \geq 1 \end{aligned}$$

Bisogna fare attenzione alle condizioni al contorno.
Non è sempre scontato che esse siano limitate da c f(n).

Ad esempio supponiamo che sia T(1) = 1 per la ricorrenza
$T(n) = 2\,T(\lfloor n/2 \rfloor) + n$ per la quale abbiamo dimostrato
$T(n) \leq c\,n \lg n$

Non si ottiene $T(1) \leq c\,1 \lg 1$ per nessun c, essendo $\lg 1 = 0$

La difficoltà può essere superata ricordando che la notazione asintotica richiede di dimostrare $T(n) \leq c\,n \lg n$ per gli $n \geq n_0$, dove n_0 è una costante.

Consideriamo allora T(2) e T(3), in quanto per n > 3, la relazione non dipende più da T(1).

$T(2) = 4 \leq 2 \lg 2$
$T(3) = 5 \leq 3 \lg 3$ → Basta scegliere $c \geq 2$

Inconveniente : non esiste una regola per trovare la soluzione corretta per ogni ricorrenza.

Di solito bisogna provare più tentativi, ma se la ricorrenza è simile ad una di cui si conosce la soluzione, si può provare la stessa soluzione.

Ad esempio: $T(n) = 2T(\lfloor n/2 \rfloor + 15) + n$

Intuitivamente per n grande la differenza tra $\lfloor n/2 \rfloor$ e $\lfloor n/2 \rfloor + 15$ non è molto significativa .
Si può perciò tentare con la soluzione $T(n) = O(n \lg n)$, che sappiamo risolvere la ricorrenza $T(n) = 2T(\lfloor n/2 \rfloor) + n$.

Il metodo iterativo

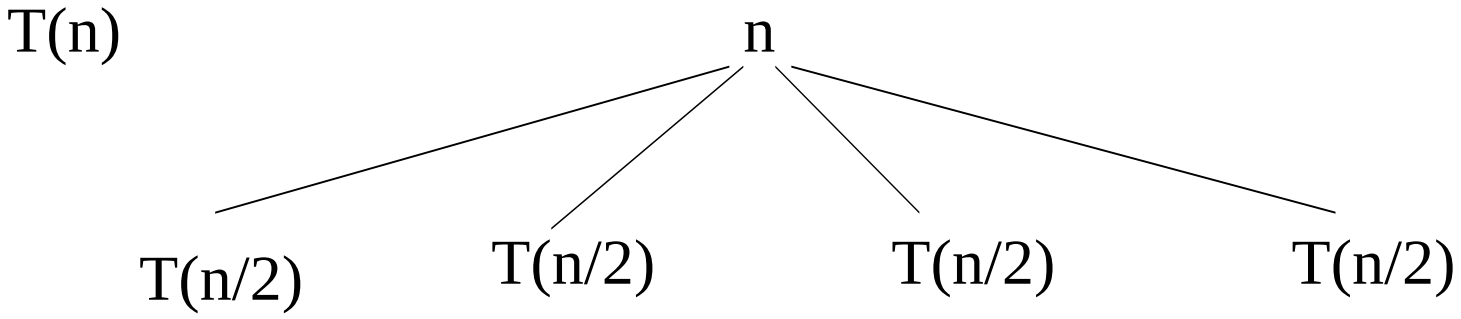
- Sviluppare la ricorrenza ed esprimerla come somma di termini che dipendono solo da n e dalle condizioni al contorno.
  Per risolvere la ricorrenza si applicano le tecniche per limitare le sommatorie.

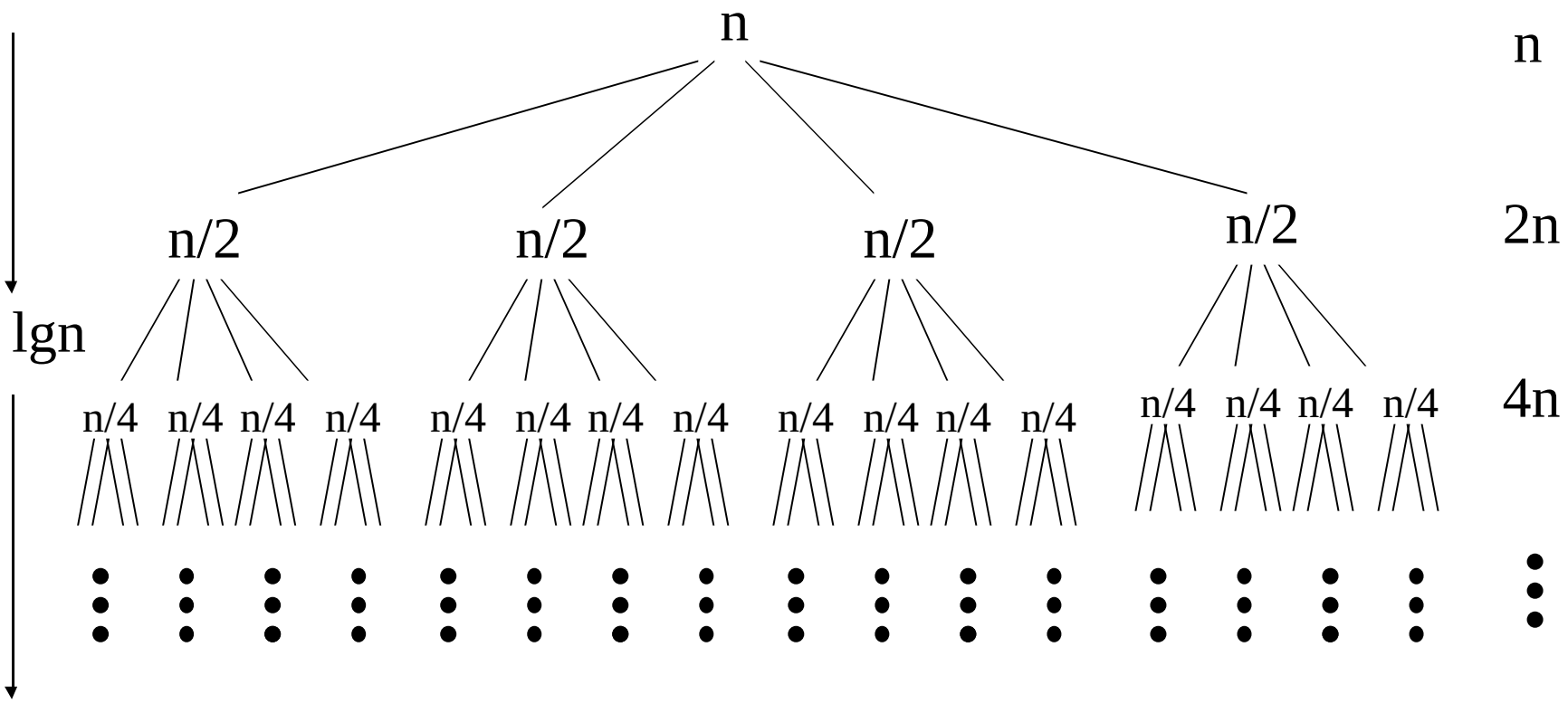
Consideriamo di nuovo l'equazione $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= n + 4T(n/2) \\
&= n + 4 (n/2 + 4 T(n/4)) \\
&= n + 2n + 4^2(n/4 + 4 T(n/8)) \\
&= n + 2n + 4n + 4^3(n/8 + 4 T(n/16)) \\
&= n + 2n + \ldots + 2^{\lg n-1} n + 2^{\lg n}T(1) \\
&= n (1 + 2 + \ldots + 2^{\lg n-1}) + n \Theta(1) \\
&= n (2^{\lg n} - 1) + \Theta(n) \\
&= n^2 - n + \Theta(n) \\
&= \Theta(n^2)
\end{aligned}
$$

Alberi di ricorsione

Disegnamo l’unfolding della ricorrenza
T(n) = n + 4T(n/2)

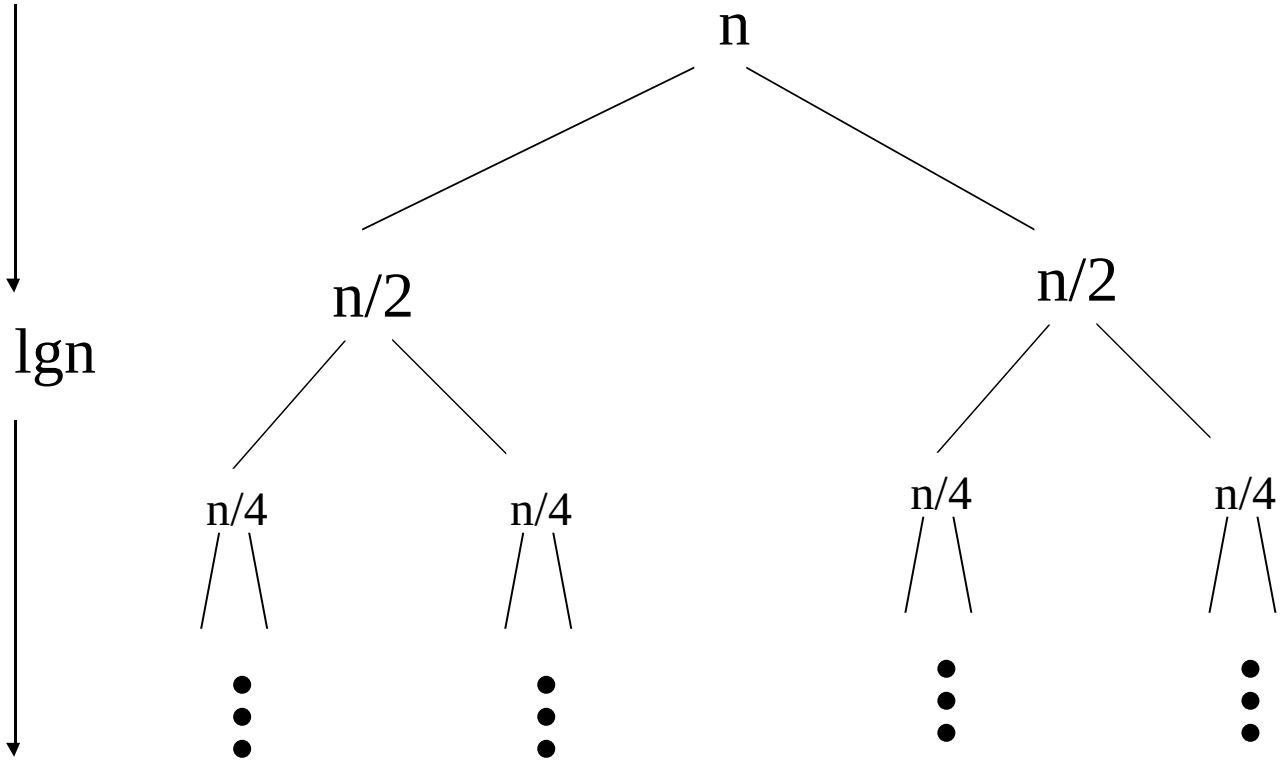




Totale: $n \sum_{i=0}^{\lg n} 2^i = n(2n-1) = \Theta(n^2)$

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

$T(n) = 2T(n/2) + \Theta(n)$ per $n \geq 2$



n

n

n

$= \Theta(n \lg n)$

Il metodo principale

Teorema principale - versione semplificata

Siano $a \geq 1$, $b > 1$ e $c \geq 0$ delle costanti.

$T(n)$ sia definita dalla seguente ricorrenza:

$$T(n) = a\,T(n/b) + \Theta(n^c)$$

dove n/b rappresenta $\lfloor n/b \rfloor$ oppure $\lceil n/b \rceil$.

Allora $T(n)$ è asintoticamente limitato nel modo seguente:

- se $c < \log_b a$ allora $T(n) = \Theta(n^{\log_b a})$
- se $c = \log_b a$ allora $T(n) = \Theta(n^c \lg n)$
- se $c > \log_b a$ allora $T(n) = \Theta(n^c)$

Esempi

$$T(n) = 2\,T(n/2) + \Theta(1)$$

$a = b = 2$

$c = 0 < 1 = \log_b a$

Caso 1 del teorema principale:

$$T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$$

$$T(n) = 2\,T(n/2) + \Theta(n)$$

$a = b = 2$

$c = 1 = \log_b a$

Caso 2 del teorema principale:

$$T(n) = \Theta(n^c \lg n) = \Theta(n \lg n)$$

$$T(n) = 8\,T(n/2) + n^2$$

$a = 8 \quad b = 2$
$c = 2 < 3 = \log_b a$

Caso 1 del teorema principale:
$T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

$a = 7 \quad b = 2$
$c = 2 < \log_b a \approx 2.801$

Caso 1 del teorema principale:
$T(n) = \Theta(n^{\log_2 7}) = O(n^3)$

Per le equazioni di ricorrenza:

1. $T_1(n) = 4\,T(n/2) + n$
2. $T_2(n) = 4\,T(n/2) + n^2$
3. $T_3(n) = 4\,T(n/2) + n^3$

Si applicano rispettivamente i casi 1, 2 e 3 del teorema principale:

$a = 4 \quad b = 2$

$c = i \quad \log_b a = 2$

Si ha allora 

1. $T_1(n) = \Theta(n^2)$
2. $T_2(n) = \Theta(n^2 \lg n)$
3. $T_3(n) = \Theta(n^3)$

```
{length[A] ≥ 1}
Insertion-sort (A)
   {A[1..j-1] è ordinato}
   for j ← 2 to length[A] do
        key ← A[j]
        {inserisci A[j] nella sequenza A[1. .j-1]
         spostando a destra gli elementi > di A[j]}
        i ← j-1
        while i > 0 and A[i] > key do
             A[i+1] ← A[i]
             i ← i-1
        A[i+1] ← key
{A[1..length[A]] è ordinato}
```

Insertion-sort = $\mathbf{O}(n^2)$

n = *length*[A]

Un algoritmo di ordinamento Divide et Impera

Merge-Sort (A, p, r)

n = r - p + 1

if p < r **then**

q ← ⌊(p + r)/2⌋

Merge-Sort (A, p, q)

Merge-Sort (A, q+1, r)

Merge (A, p, q, r)

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) \qquad \text{per } n \geq 2$$

$T(n) = \mathbf{Q}(n \lg n)$

L’algoritmo **Merge-Sort** è perciò asintoticamente migliore dell’algoritmo **Insertion-sort**

Si può inoltre dimostrare che $\Theta(n \lg n)$ confronti sono necessari nel caso peggiore per ordinare n numeri per qualunque algoritmo basato sui confronti.

Quindi l’algoritmo **Merge-Sort** è (asintoticamente) ottimo.

```
{p ≤ r}
Merge-Sort (A, p, r)
    if p < r then
        q ← ⌊(p + r)/2⌋
        {p ≤ q}
        Merge-Sort (A, p, q)
        {A[p..q] ordinato}
        {q+1 ≤ r}
        Merge-Sort (A, q+1, r)
        {A[p..q] ordinato & A[q+1..r] ordinato}
        Merge (A, p, q, r)
        {A[p..r] ordinato}
{ A[p..r] ordinato }
```

{A[p..q] ordinato & A[q +1..r] ordinato}

Merge (A, p, r, q)

i ← p

j ← q+1

k ← p

{B[p..k-1] ordinato & A[i..q] ordinato & A[j..r] ordinato &
& {A[i..q]} ≥ {B[p..k-1]} & {A[j..r]} ≥ {B[p..k-1]}}

while i ≤ q **and** j ≤ r **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

{B[p..k-1] ordinato & A[i..q] ordinato & A[j..r] ordinato &
& {A[i..q]} ≥ {B[p..k-1]} & {A[j..r]} ≥ {B[p..k-1]} &
& (i > q ∨ j > r) }

$t \leftarrow k$

{B[p..k-1] ordinato & A[i..q] ordinato & A[j..r] ordinato &
& {A[i..q]} ≥ {B[p..k-1]} & {A[j..r]} ≥ {B[p..k-1]} &
& (i > q ∨ j > r) }

{j > r & i ≤ q}

for h ← i **to** q **do**
A[t] ← A[h]
t ← t + 1

{i > q & j ≤ r}

{B[p..k-1] ordinato & A[k..r] ordinato & {A[k..r]} ≥ {B[p..k-1]}

for t ← p **to** k-1 **do**
A[t] ← B[t]

{A[p..r] ordinato}

{A[p..q] ordinato & A[q+1..r] ordinato}

```
Merge (A, p, q, r)
    i ← p
    j ← q+1
    k ← p
    while i ≤ q and  j ≤ r 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
    t ← k
    for h ← i to q do
                A[t] ← A[h]
                t ← t+1
    for t ← p to k-1 do
                A[t] ← B[t]
```

{A[p..r] ordinato}

Un altro algoritmo di ordinamento Divide et Impera

{true}

Quicksort (A, p, r)

if p < r **then**

{p < r}

q ← ***Partition*** (A, p, r)

{{A[p..q]} ≤ {A[q+1..r]}}

Quicksort (A, p, q)

{A[p..q] ordinato & {A[p..q]} ≤ {A[q+1..r]}}

Quicksort (A, q+1, r)

{A[p..q] ordinato & A[q+1..r] ordinato &

& {A[p..q]} ≤ {A[q+1..r]}}

{A[p..r] ordinato}

```
{p < r}
Partition (A, p, r)
        x ← A[p]
        i ← p-1
        j ← r+1
        while true do
            repeat j ← j-1 until A[j] ≤ x
            repeat i ← i+1 until A[i] ≥ x
            if i < j
                then “scambia A[i] con A[j]”
                else return j
{{A[p..j]} ≤ {A[j+1..r]}}
```

{true}

Quicksort-1 (A, p, r)

if p < r **then**

{p < r}

q ← ***Partition-1*** (A, p, r)

{{A[p..q-1]} ≤ A[q] ≤ {A[q+1..r]}}

Quicksort-1 (A, p, q-1)

{A[p..q-1] ordinato & {A[p..q-1]} ≤ {A[q]}}

Quicksort-1 (A, q+1, r)

{A[p..q-1] ordinato & A[q+1..r] ordinato &

& {A[p.. q-1]} ≤ A[q] ≤ {A[q+1..r]}}

{A[p..r] ordinato}

```
{p < r}
Partition-1 (A, p, r)
        j ← p
        x ← A[p]
        {{A[p+1..j]} ≤ A[p] ≤ {A[j+1..i-1]}}
        for i ← p to r do
            if A[i] < x
                then  j ← j+1
                              if i ≠ j then "scambia A[i] con A[j]"
        {{A[p+1..j]} ≤ A[p] ≤ {A[j+1..r]}}
        "scambia A[p] con A[j]"
        return j
{{A[p..j-1]} ≤ A[j] ≤ {A[j+1..r]}}
```

Analisi di complessità

Siano T(n) e P(n) le complessita` (il numero di confronti tra elementi di A) rispettivamente di ***Quicksort-1*** e ***Partition-1***.
Poiche` P(n) = n e le due porzioni di A sulle quali avviene la chiamata ricorsiva hanno j - 1 e n - j elementi otteniamo:

$$T(n) = 0 \qquad \text{per } n < 2$$
$$T(n) = n + T(j - 1) + T(n - j) \qquad \text{per } n \geq 2$$

Caso migliore

A e` sempre suddiviso a meta`: la porzione di A con elementi minori di A[p] e quella con elementi maggiori o uguali contiene (n-1)/2 elementi.

L'equazione di ricorrenza assume la forma:

$$T(n) = 0 \qquad \text{per } n < 2$$
$$T(n) = 2\,T((n-1)/2) + n \qquad \text{per } n \geq 2$$

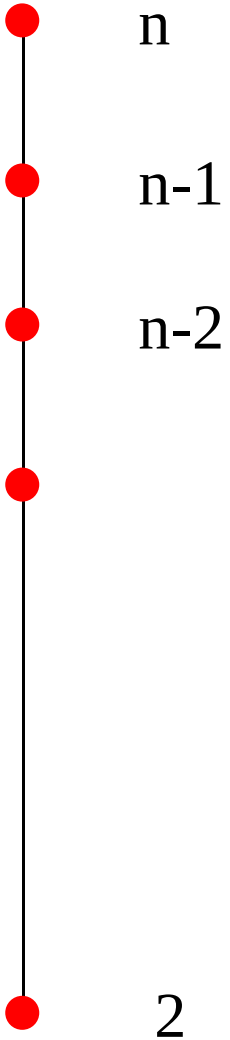
$$\mathbf{T(n) = Q(n \lg n)}$$

Caso peggiore

A[p] e` sempre l'elemento minore: la porzione di A con elementi minori di A[p] e` vuota, mentre la porzione di A con elementi maggiori o uguali contiene n-1 elementi.

L'equazione di ricorrenza assume la forma:

$$T(n) = 0 \qquad \text{per } n < 2$$
$$T(n) = T(n - 1) + n \qquad \text{per } n \geq 2$$



$T(n) = Q(n^2)$

Caso medio nell'ipotesi che tutte le permutazioni in input siano ugualmente probabili.

Partition produce un misto di suddivisioni "buone" e "cattive", che sono distribuite lungo l'albero in modo casuale.

$$T(n) = n + 1/n \sum_{j=1}^{n} \left(T(j - 1) + T(n - j)\right)$$

$$= n + 2/n \sum_{j=1}^{n-1} T(j) = 2\,(n+1) \sum_{k=3}^{n+1} 1/k$$

$$= 2(n+1)\,\lg(n+1)$$

$$\mathbf{T(n) = O(n \lg n)}$$

Quicksort randomizzato

Randomized-Partition-1 (A, p, r)

    i ← *Random* (p, r)
    "scambia A[p] con A[i]"
    return *Partition-1* (A, p, r)

Randomized-Quicksort-1 (A, p, r)

    if p < r
        then q ← *Randomized-Partition-1* (A, p, r)
            Randomized-Quicksort-1 (A, p, q-1)
            Randomized-Quicksort-1 (A, q+1, r)

Ordinamento in tempo lineare:

l'algoritmo *Counting-Sort*

Idee guida

- contare per ogni elemento quanti ce ne sono di uguali
- contare quanti ce ne sono di minori o uguali
- sistemare ogni elemento nella posizione corretta

Si usano due array supplementari:

B per memorizzare l'output ordinato

C per contare gli elementi

Counting-Sort (A, B, k)

for i ← 1 **to** k **do** — **O**(k)

  C[i] ← 0

for j ← 1 **to** *length*[A] **do** — **O**(n)

  C[A[j]] ← C[A[j]] + 1

for i ← 2 **to** k **do** — **O**(k)

  C[i] ← C[i] + C[i-1]

for j ← *length*[A] **downto** 1 **do** — **O**(n)

  B[C[A[j]]] ← A[j]

  C[A[j]] ← C[A[j]] - 1

O(n + k) $\xrightarrow{k = \mathbf{O}(n)}$ **O**(n)

Selezione dell'i-esimo elemento

Input: n numeri distinti e un numero $1 \leq i \leq n$

Output: l'elemento x che è maggiore di esattamente i-1 dei numeri dati

Esiste ovviamente un algoritmo di complessità $\mathbf{\Theta}(n \lg n)$

Select (A, i)
    Merge-Sort (A, 1, n)
    $x \leftarrow A[i]$

Se usiamo la tecnica **divide-et-impera** otteniamo un algoritmo di complessità $\mathbf{\Theta}(n)$

{p ≤ r & 1 ≤ i ≤ n}

Randomized-Select (A, p, r, i)

if p = r **then return** A[p]

q ← *Randomized-Partition* (A, p, r)

k ← q - p + 1

if i ≤ k **then return** *Randomized-Select* (A, p, q, i)

else return *Randomized-Select* (A, q+1, r, i-k)

{esistono n-1 elementi di A ≤ a quello restituito}

L'algoritmo funziona bene nel caso medio, ma poichè è "randomizzato" nessun particolare input provoca il comportamento del caso peggiore.

Troviamo un limite superiore alla complessità nel caso medio, che si presenta quando l'i-esimo elemento deve sempre essere cercato sul lato più grande della partizione.

$$T(n) \le 1/n \sum_{k=1}^{n-1} T(\max(k, n-k)) + n$$

$$\le 2/n \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + n$$

Dimostriamo per sostituzione che $T(n) = \mathbf{O}(n)$

$$T(n) \le c\, n$$

$$T(n) \le 2/n \sum_{k=\lceil n/2 \rceil}^{n-1} c\, k + n$$

$$= 2c/n \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \right) + n$$

$$\le 2c/n \left((n-1)n/2 - (n/2-1)(n/2)/2 \right) + n = (3/4\, c+1)\, n - c/2$$

$$\le (3/4\, c+1)\, n \le c\, n \quad \text{per } c \ge 4$$