

Algoritmi e Strutture Dati

Andrea Broccoletti

INTRODUZIONE AGLI ALGORITMI

ALGORITMO sequenza finita di istruzioni che eseguita a partire da un insieme di dati produce, in un numero finito di passi, altri dati.

- i dati devono poter essere rappresentati in modo finito, ma non devono essere necessariamente di quantità finita.
- ogni istruzione non è ambigua.
- i dati prodotti sono in relazione con i dati: a partire dai quali l'esecuzione della sequenza di istruzioni si applica, mediante una funzione $f: I \rightarrow O$, l'algoritmo deve quindi essere deterministico.

PROBLEMA COMPUTAZIONALE data un'istanza, trovare una soluzione attraverso la funzione definita: istanza \rightarrow soluzione dove $f: I \rightarrow O$ è così definita

e.g. istanza $x \in \mathbb{N}$, soluzione $f(x)$ dove $f: \mathbb{N} \rightarrow \mathbb{N} \quad \forall x \in \mathbb{N}, f(x) = 2^x$

e.g. istanza $(x,y) \in \mathbb{Z} \times \mathbb{N}$, soluzione $f(x)$ dove $f: \mathbb{Z} \times \mathbb{N} \rightarrow \mathbb{Z} \quad \forall x = (x,y) \in I, f(x) = x^y$

TEMPO DI CALCOLO dato x di dimensione n , come varia il tempo di calcolo al variare di n .

NOTAZIONI ASINTOTICHE E ASINTOTI

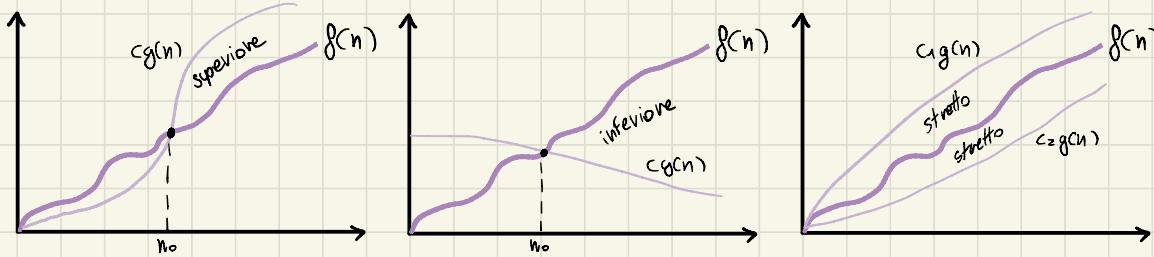
Sia g una funzione non negativa. Definiamo:

LIMITE ASINTOTICO SUPERIORE $O(g(n)) = \{f : \exists c, n_0 > 0, \forall n > n_0, 0 \leq f(n) \leq cg(n)\}$

LIMITE ASINTOTICO INFERIORE $\Omega(g(n)) = \{f : \exists c, n_0 > 0, 0 \leq c \cdot g(n) \leq f(n), \forall n > n_0\}$

LIMITE ASINTOTICO STRETTO $\Theta(g(n)) = \{f : \exists c_1, c_2, n_0 > 0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n > n_0\}$

Graficamente abbiamo, nell'ordine:



POLINOMI IN N un polinomio $f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_0$, esprimibile anche nella forma $p(n) = \sum_{i=0}^k a_i n^i$ ha:

- $\limsup g(n) = n^k$
- $\liminf g(n) = n^k$

Possiamo quindi definire una scala di polinomi: n, n^2, n^3, \dots che sono tutte classi separate tra loro.

COSTANTI sono (per convenzione) considerate $O(1)$, $\Omega(1)$ e $\Theta(1)$

Possiamo dunque definire:

SCALA DI ASINTOTI 1, $\log n$, n , $n \log n$, n^2 , n^3 , ...

OSS nella scala abbiamo solo quelli trattabili in termini teorici, dopo i polinomiali ci sono ..., 2^n , 3^n , ..., $n!$, n^n

COMPLESSITÀ COMPUTAZIONALE

Número di istruzioni, eventualmente pesate, eseguite da un algoritmo su un input.

→ le istruzioni possono avere costi diversi int

OSS è legata al tempo

OSS le istruzioni eseguite sono diverse dalle istruzioni scritte

È dipendente da:

- dimensione dell'input
- a parità di dimensione, dalla specificità, dalla struttura dell'input

TA considerata la notazione $\overbrace{f: I \rightarrow V}^P$ Con algoritmo A che risolve P, introduciamo:
 $T_A^*: I \rightarrow \mathbb{R}^+$. $\forall x \in I$, $T^*(x)$ è il numero di istruzioni, eventualmente pesate, eseguite da A su un input $x \in I$

Parlando di complessità, ci si deve riferire a:

- $x \in I$, dim(x) dimensione dell'input
- $n \in \mathbb{N}$, n° di istruzioni eseguite da A su un input di dimensione $n \in \mathbb{N}$

CASI distinguiamo 3 casi:

- caso peggiore $T_A: \mathbb{N} \rightarrow \mathbb{R}^+$ $\forall n \in \mathbb{N}$, $T_A(n) = \max \{T_A^*(x) : x \in I, \dim(x) = n\}$
- caso migliore $t_a: \mathbb{N} \rightarrow \mathbb{R}$ $\forall n \in \mathbb{N}$ $t_a(n) = \min \{T_A^*(x) : x \in I, \dim(x) = n\}$
- caso medio $\bar{T}_A: \mathbb{N} \rightarrow \mathbb{R}^+$ $\forall n \in \mathbb{N}$ $\bar{T}_A(n)$ è il valore medio di $T^*(x)$ al variare di $x \in I$ con $\dim(x) = n$

e.g. $\{x \in I : \dim(x) = n\} = \{x_1, x_2\}$ tali che: x_1 ha probabilità p_1 [$\frac{1}{4}$]
 x_2 ha probabilità p_2 [$\frac{3}{4}$]

$$\text{quindi } \bar{T}_A(n) = \frac{1}{4} \cdot T^*(x_1) + \frac{3}{4} \cdot T^*(x_2)$$

[PRO] SOMMA ELEMENTI DI UN VETTORE $I = \mathbb{Z}^*$ $\forall v \in I, f(v) = \begin{cases} v_1 + \dots + v_n & \text{se } v = (v_1, \dots, v_n) \text{ con } n \geq 1 \\ 0 & \text{se } v = () \end{cases}$

SommaV(v):

$n = \text{length}(v)$	1	c_1
$\text{sum} = 0$	1	c_2
For $i = 1$ to n	$n+1$	c_3
$\text{sum} = \text{sum} + v[i]$	n	c_4
return sum	1	c_5

Dall'algoritmo abbiamo $T^*(x) = 1 + 1 + (\dim(x) + 1) + \dim(x) + 1$, quindi:

$$T_a(n) = t_a(n) = E_a(n) = T^*(x) = 2n + 4 = \Theta(n)$$

[PRO] ELEMENTO MINIMO DI UN VETTORE

$I = \text{Vettore A di } n \text{ naturali, assunto non vuoto}$

$O = \text{Elemento minimo di A}$

Function Minimo(A):

$\min = A[1]$	c_1	1
For $i = 2$ to $A.length$ do:	c_2	n
if $A[i] < \min$ then:	c_3	$n-1$
$\min = A[i]$	c_4	t in funzione del for $0 \leq t \leq n-1$
return min	c_5	1

Il tempo di calcolo deve essere distinto in migliore e peggiore:

- $T(n)$ [minimo in prima posizione]: $c_1 + c_3 + nc_2 + (n-1)c_3 + 0 \times c_4 = \Theta(n)$
- $t(n)$ [ordinato in senso decrescente]: $c_1 + c_3 + nc_2 + (n-1)c_3 + (n-1)c_4 = \Theta(n)$
- $T(n) = \Theta(n)$, in quanto migliore e peggiore si equivalgono asintoticamente

ALGORITMI DI RICERCA

[PRO] RICERCA SEQUENZIALE

Dato un qualunque vettore di lunghezza almeno 1 di componenti intere tutte diverse tra loro, e dato un valore K , calcolare se esiste la posizione del vettore nella quale si trova K (se tale posizione non esiste, il calcolo dà luogo al valore 0).

$$I = \text{Pippo} \times \mathbb{Z} \quad \text{quindi } x = (v, u), \quad \text{Pizzo} = \{v \in \mathbb{Z}^+ \text{ con } v = (v_1 \dots v_n) : \forall i, j \in \{1 \dots n\} \quad v_i \neq v_j\}$$

$$0 = \mathbb{N}$$

$$f: I \rightarrow \mathbb{O} \quad \forall x \in I, f(v, u) = \begin{cases} 1 & \forall i \in \{1 \dots n\} \quad v_i = u \\ 0 & \forall i \in \{1 \dots n\} \quad v_i \neq u \end{cases}$$

Scan(v, K):

$$n = \text{length}(v)$$

while $v[i] \neq K$ and $i \leq n$:

$i++$

 if $i \leq n$

 return i

 else

 return 0

$v[1]$	$v[2]$	$v[3]$	$v[n]$	$/$
1	2	3	n	$n+1$
0	1	2	$n-1$	n

Dall'algoritmo abbiamo che $\dim(x) = n+1$, ma è influente solo n - length vettore:

$$\bullet \quad T_A^*(x) = \begin{cases} 1 + 1 + \overbrace{1 + 0} + 1 + 1 = 5 \\ 1 + 1 + \overbrace{2 + 1} + 1 + 1 = 7 \\ \dots \\ 1 + 1 + n + n - 1 + 1 + 1 = 2n + 3 \\ 1 + 1 + n + 1 + n + 1 + 1 = 2n + 5 \end{cases}$$

$$\bullet \quad T_A(n) = 2n + 5 = \Theta(n) \quad [\text{l'elemento non è presente}]$$

$$\bullet \quad t_A(n) = 5 = \Theta(1) \quad [\text{elemento in prima posizione}]$$

$$\bullet \quad E_A(n) = \sum 2i + 3 = 2 \sum_{i=1}^{n+1} i + 3 \sum_{i=1}^{n+1} 1 = 2 \cdot \frac{(n+1)(n+2)}{2} + 3(n+1) = n+2 + 3 = n+5$$

Il tempo medio risulta quando K è nella metà dell'avray $\rightarrow 2 \cdot \frac{n}{2} + 5 = n+5$

[PRO] RICERCA SEQUENZIALE ORDINATA Si assume il vettore ordinato

Scan(V, k):

$n = \text{length}(V)$

$i = 1$

while $V[i] < k$ and $i \leq n$:

$i++$

if $i \leq n$ and $V[i] = k$

 return i

else

 return 0

- Caso migliore $\{(v, k) \in I : k \leq v_1\}$

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

Rispetto alla ricevuta non ordinata non sembra essere cambiato niente, in realtà questo insieme è più grande, comprende più casi

- Caso peggiore $\{(v, k) \in I : k > v_n\}$

$$T(n) = 2n + 5$$

In questo caso è un insieme più piccolo, quindi c'è un miglioramento con l'ordinamento.

[PRO] **Ricerca dicotomica** Dato che il vettore è ordinato, possiamo stabilire un range di ricerca più piccolo dell'intero vettore

$\text{Scan}(V, k)$:

$$sx = 1$$

$$dx = \text{length}(v)$$

$$\text{meta} = (sx + dx) \text{ div } 2$$

$$\text{while } V[\text{meta}] \neq k \text{ and } sx \leq dx$$

$$\text{if } V[\text{meta}] < k$$

$$sx = \text{meta} + 1$$

else

$$dx = \text{meta} - 1$$

$$\text{meta} = (sx + dx) \text{ div } 2$$

$$\text{if } V[\text{meta}] == k$$

return meta

else

return 0

1	2	3	4	5	6	7	8
5	7	11	16	25	30	41	44

sx $\underline{\hspace{2cm}}$ meta $\underline{\hspace{2cm}}$ dx

k può essere prima

o dopo la metà

- Caso migliore $\{(v, k) \in I : v_{(n+1) \text{ div } 2} = k\}$

$$t(n) = 3 + 1 + 0 + 2 = 6 = \Theta(1)$$

- Caso peggiore $\{(v, k) \in I : \forall i \ v_i \neq k\}$

$$T(n) = 3 + ? + 2 = 3 + (\log_2 n + 1) + 3 \log_2 n + 2 = 4 \log_2 n + 6 = \Theta(\log n)$$

capire quante volte l'array viene dimezzato:

$$1^{\circ} \text{ ciclo} : \frac{n}{2}$$

$$2^{\circ} \text{ ciclo} = \frac{n}{4}$$

$$3^{\circ} \text{ ciclo} = \frac{n}{8}$$

...

$$m - \text{ciclo} = \frac{n}{2^m} = 1 \text{ poiché una sola cella è rimasta}$$

$$2^m = n$$

$$m = \log_2 n$$

Il caso peggiore è migliorato drasticamente.

PROBLEMA DELL'ORDINAMENTO

$I = \{v \in \bigcup_{n=1}^{\infty} \mathbb{Z}^n\} \rightarrow O = I$ output e input sono lo stesso vettore.

In particolare, $\forall v \in I$, $f(v) = v' \in O$ tale che:

- v' è una permutazione di v
- $v'_1 \leq v'_2 \leq \dots \leq v'_n$ ordinato crescente

STABILE un algoritmo di ordinamento è stabile se e solo se mantiene inalterato l'ordine relativo di elementi (chiavi) uguali.

e.g. Mergesort è stabile, quicksort no.

IN LOCO un algoritmo di ordinamento si dice in loco se utilizza un numero costante di variabili, indipendentemente dalla dimensione dell'input.

e.g. il mergesort non ordina in loco, perché si avale di un array di appoggio, mentre quicksort lo è

ALGORITMI che risolvono il problema:

- Selectionsort
- Bubblesort
- Insertionsort
- Mergesort

ALGORITMI DI ORDINAMENTO

[PRO] **SELECTION SORT** ricordando che $I = O = \{v \in \cup_{n=1}^{\infty} \mathbb{Z}^n\}$, opera selezionando l'elemento minimo, come da esempio.

1	2	3	4	5
5	-2	1	4	0

e.g.

determina l'elemento minimo considerando $V[1..5]$

1	2	3	4	5
-2	5	1	4	0

lo scambia con l'elemento in posto 1 considerando $V[1..4]$

Ripete il procedimento per tutto l'array, fino all'ultimo elemento.

Selesort(V):

$n = \text{length}(V)$

1

for $i = 1$ to $n - 1$:

n

$\text{posmin} = i$

$n - 1$

for $j = i + 1$ to n :

*

if $V[j] < V[\text{posmin}]$:

**

$\text{posmin} = j$

if $\text{posmin} \neq i$:

?? [influenza per la complessità]

$t_{mp} = V[j]$

$V[i] = V[\text{posmin}]$

$n - 1$

$V[\text{posmin}] = t_{mp}$

} Influenza (è già presente in $O(n^2)$ in * o **)

test ciclo for

$$i = 1 \quad j : 2 \dots n \rightarrow n - 2 + 1 = n - 1 + 1 = n$$

$i \quad a \dots b$ sono $(b - a) + 1$

$$i = 2 \quad j : 3 \dots n \rightarrow n - 3 + 1 = n - 2 + 1 = n - 1$$

$$i = 3 \quad j : 4 \dots n \rightarrow n - 4 + 1 = n - 3 + 1 = n - 3$$

$$i \quad j : i + 1 \dots n \rightarrow n - (i + 1) + 1 + 1 = n - i + 1$$

$$\text{Da cui } \sum_{i=1}^{n-1} (n - i + 1) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 = n(n-1) - \frac{(n-1)n}{2} + n - 1 = O(n^2)$$

$$*\sum_{i=1}^{n-1} (n - i) = n(n-1) - \frac{(n-1)n}{2} = O(n^2)$$

- caso migliore $\{v \in I : v_1 \leq v_2 \leq \dots \leq v_n\}$

$$t(n) = 1 + n + n - 1 + * + ** + n - 1 = \Theta(n^2)$$

- caso peggiore $\{v \in I : v_1 > v_2 > \dots > v_n\}$

$$T(n) = 1 + n + n - 1 + * + *** + *** + n - 1 = \Theta(n^2)$$

Da cui abbiamo $\bar{E}(n) = \Theta(n^2)$

Il punto debole dell'algoritmo è che, se il vettore è già ordinato (caso migliore) esegue comunque tutto l'algoritmo.

[PRO] **BUBBLESORT** Sia n la lunghezza di V , se il vettore è ordinato si possono fare meno operazioni, quindi:

- scansiona il vettore $V[1 \dots n-1]$ scambiando ogni elemento con il successivo se quest'ultimo è inferiore. L'ultimo elemento ora è il più grande.
- ripete su ogni sottovettore, senza considerare l'ultimo elemento

Bubblesort (V):

$$n = \text{length}(V)$$

1

For $i = 1$ to $n-1$:

n

For $j = 1$ to $n-i$:

*

If $V[j+1] < V[j]$

* *

Scambia($V[j+1], V[j]$)

dipende da V : O oppure **

* $\sum_{i=1}^{n-1} (n-i+1) = \frac{1}{2} n(n-1) + n - 1$

* * $\sum_{i=1}^{n-1} (n-i) = \frac{1}{2} n(n-1)$

- caso migliore $\{v \in I : v_1 \leq v_2 \leq \dots \leq v_n\}$

$$t(n) = 1 + n + \frac{1}{2} n(n-1) + n - 1 + \frac{1}{2} n(n-1) + 0 = n(n-1) + 2n = n^2 + n = \Theta(n^2)$$

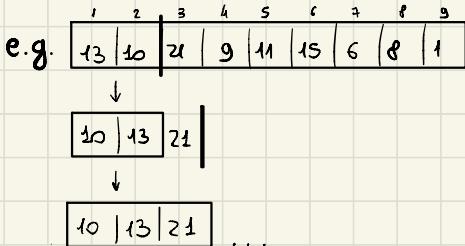
- caso peggiore $\{v \in I : v_1 > v_2 > \dots > v_n\}$

$$t(n) = 1 + n + \frac{1}{2} n(n-1) + n - 1 + \frac{1}{2} n(n-1) + 3 \cdot \frac{1}{2} n(n-1) = \frac{5}{2} n(n-1) + 2n = \frac{5}{2} n^2 - \frac{1}{2} n = \Theta(n^2)$$

Da cui $\bar{E}(n) = \Theta(n^2)$

[PRO] INSERTIONSORT Sia n la lunghezza di V :

- considera $U = V[2..n]$
- inserire in $V[1..1]$ in modo tale che $V[1..2]$ diventi ordinata
- $V[1..2]$ è ordinato, considera $U = V[3..n]$
- inserire in $V[1..2]$ U in modo tale che $V[1..3]$ diventi ordinata



Insertionsort (V):

```

 $n = \text{length}(V)$            1
for  $i = 2$  to  $n$ :          n
     $U = V[i]$               $n-1$ 
     $j = i - 1$              $n-1$ 
    while  $j > 0$  and  $V[j] > U$ :  $\sum_{i=2}^n C_i$ ; *
         $V[j+1] = V[j]$        $\sum_{i=2}^n (C_{i-1})$ 
         $j --$ 
     $V[j+1] = U$              $n-1$ 

```

* Z_i quante volte viene eseguito il test del while dato i (si legge "tao")

- **caso migliore** $\{v \in I : v_1 \leq v_2 \leq \dots \leq v_n\}$

$$\sum_{i=2}^n 1 = n-2+1 = n-1, Z_i = 1$$

$$t(n) = 1 + n + n-1 + n-1 + n-1 + \dots + n-1 = 5n-3 = O(n)$$

- **caso peggiore** $\{v \in I : v_1 > v_2 > \dots > v_n\}$

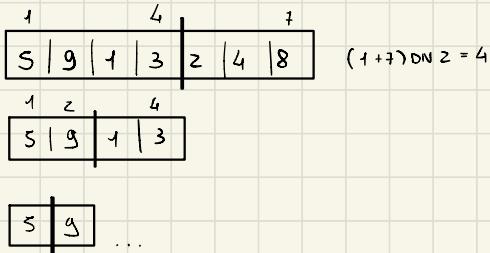
$$Z_i = i-1+1 = i, \sum_{i=2}^n i = \frac{n(n+1)}{2} + n-2$$

$$T(n) = 1 + n + n-1 + n-1 + \frac{n(n+1)}{2} + n-2 + n-1 = \frac{3}{2}n(n+1) + 2n - 3 = O(n^2)$$

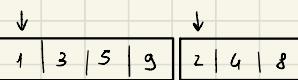
- **caso medio** $Z_i = \frac{i-1}{2} \approx \frac{i}{2}, O(n^2)$

[PRO] MERGESORT Si basa sul divide et impera (et combina), è quindi una procedura ricorsiva.

e.g. Procedura di divide



Procedura d: impera e combina:



confronta a due a due, spostando l'indice
quando viene inserito

Mergesort(V, p, q): Ricorsivamente, $V[p \dots q]$: si ferma quando $p=q$
if $p < q$:

$$m = (p+q) \text{ DN } 2$$

Mergesort(V, p, m)

Mergesort($V, m+1, q$)

merge(V, p, m, q)

Merge(V, p, m, q): assumendo le due metà ordinate

$$sx = p$$

$$dx = m+1$$

$$i = p$$

while $sx \leq m$ and $dx \leq q$:

if $V[sx] \leq V[dx]$

$W[i] = V[sx]$

$sx++$

else:

$W[i] = V[dx]$

$dx++$

$i++$

while $sx \leq m$

$W[i] = V[sx]$

$sx++$

while $dx \leq q$

$W[i] = V[dx]$

$dx++$

For $j = p$ to q :

$V[j] = W[j]$

La procedura è fatta in modo tale che le procedure saranno sempre eseguite, quindi caso peggiore, migliore e medio coincidono.

$T(n)$ costo dell'invocazione su $v[1..n]$:

- **CASO BASE** $n=1$, $T(n)=1$
- **CASO PASSO** $n > 1$, $T(n) = c + T(\frac{n}{2}) + T(\frac{n}{2}) + \Theta(n)$ merge

T è espressa ricorsivamente, vediamo di risolvere l'equazione di ricchezza.

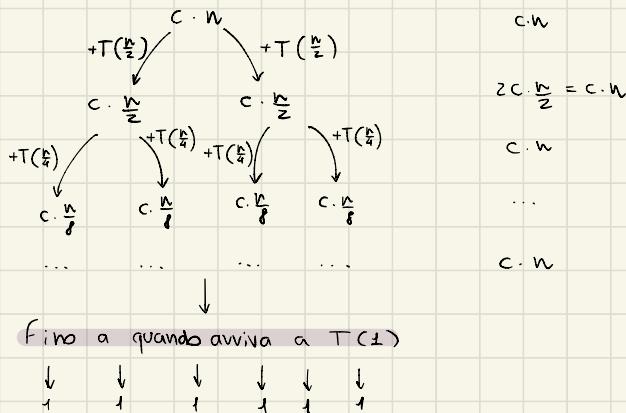
Sì può risolvere con l'albero delle chiamate, assumendo che:

- n sia una potenza di 2
- il costo di merge sia $c \cdot n$

Verrà così:

$$\begin{cases} T(1) = 1 \\ T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + c \cdot n = 2T\left(\frac{n}{2}\right) + c \cdot n \\ T\left(\frac{n}{2}\right) = 2 \cdot T\left(\frac{n}{4}\right) + c \cdot \frac{n}{2} \end{cases}$$

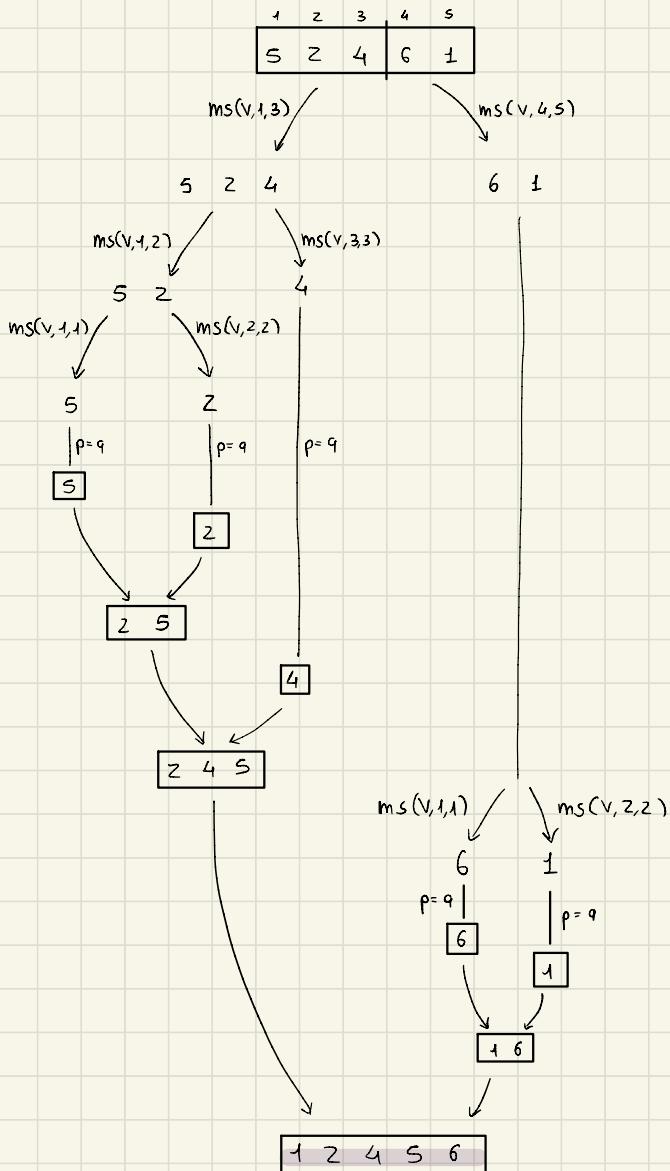
Da cui possiamo costruire l'albero:



Quindi dobbiamo avere $\frac{n}{2^u} = 1$ $2^u = n$ $u = \log_2 n$ con u numero di voci dell'albero, e la complessità $T(n) = c \cdot n \cdot \log_2 n$, da cui abbiamo $T(n) = \Theta(n \log n)$

La procedura è dimostrabile anche senza le assunzioni fatte, secondo il teorema dell'esperto.

e.g. Applicare la procedura mergesort($V, 1, 5$) a:



TEOREMA DELL'ESPERTO

Data $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ con $n \in \mathbb{N}$, $a \geq 1$, $b > 1$:

- 1 se $\exists \varepsilon > 0 : f(n) = O(n^{\log_b a - \varepsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$
- 2 se $f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$
- 3 se $\exists \varepsilon > 0 : f(n) = \Omega(n^{\log_b a + \varepsilon}) \text{ e } \exists c < 1 : f\left(\frac{n}{b}\right) \leq c \cdot f(n) \text{ definitivamente} \Rightarrow T(n) = \Theta(f(n))$

e.g. Supponiamo $T(n) = 2T\left(\frac{n}{2}\right) + 3n - 1$ che soddisfa la formula, quindi:
 $a = 2$
 $b = 2$

$$\log_b a = \log_2 2 = 1$$

- $3n - 1 = O(n^{1-\varepsilon}) \quad \forall \varepsilon > 0$ che soddisfa
- $3n - 1 = \Theta(n^1)$ sì

$$f(n) = 3n - 1$$

$$\Leftrightarrow \Theta(n^1 \cdot \log_2 n)$$

DIVIDE ET IMPERA (ET COMBINA)

Sia P un problema computazionale. Per risolvere P su un generico $x \in I$, si intende progettare una procedura fatta nel modo seguente:

- **divide** dividere P in k sottoproblemi, ciascuno dei quali non è altro che il medesimo P applicato su un input che è più piccolo di x e consiste di una delle k parti di cui x è composto.
- **impera** risolvere ricorsivamente i k sottoproblemi utilizzando questa stessa procedura (che consiste di questi stessi punti) invocata k volte ciascuna su una delle k parti di cui x è composto.
- **combina** si combinano opportunamente le k soluzioni dei k sottoproblemi per costruire la soluzione di P su x .

RICORSIONE

Iniziamo definendo \mathbb{N} come:

① $0 \in \mathbb{N}$

② se $n \in \mathbb{N}$, allora $n+1 [scn] \in \mathbb{N}$

③ niente altro è un numero naturale se non un elemento definito da ① e ②.

e.g. $I = n \in \mathbb{N}$

sol. $f(n) : \mathbb{N} \rightarrow \mathbb{Z}$ dove $\forall n \in \mathbb{N}, f(n) = \begin{cases} 6 & \text{se } n=0 \\ 1-2f(n) & \text{se } n>0 \end{cases}$

$Ric(n)$:

if $n == 0$.

return 6

else

return $1 - 2 * Ric(n-1)$

e.g. $f(n) = n^2 = n^2 - 2n + 1 + 2n - 1$

$f(n-1) = (n-1)^2 = n^2 - 2n + 1$) si deve ritrovare in $f(n)$

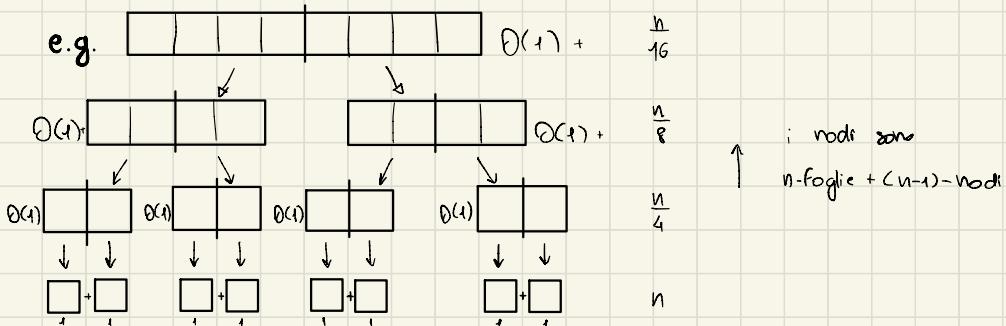
Concludo che $f(n) = f(n-1) + 2n - 1$

[PRO] SOMMA DIVIDE ET IMPERA

$I = \cup_n$ vettore non vuoto di n interi.

$$O = \sum_{i=1}^n A[i]$$

L'approccio iterativo ha complessità $\Theta(n)$, come fare un approccio divide et impera?



Problema implementativo: quanto costa a passare i parametri, ovevo dividere a metà il vettore? La divisione è logica, con l'uso di due parametri che indicano gli estremi.

SommaDI(A, l, r):

if ($l == r$):

return $A[r]$

else:

mid = $(l+r) \text{ div } 2$

return SommaDI(A, l, m) + SommaDI($A, m+1, r$)

$$\left\lfloor \frac{l+r}{2} \right\rfloor$$

Quale è il tempo di calcolo, sia $n = r - l + 1$: intervalllo con estremi inclusi

- caso base un solo elemento:

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

- caso passo

$$T(n) = \Theta(1) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) = \Theta(1) + 2T\left(\frac{n}{2}\right) \rightarrow \text{Supponiamo } n \text{ sempre pari, asintoticamente}$$

$$T(n) = \Theta(1) + \underbrace{n \text{ foglie} + (n-1) \text{ nodi interi}}_{\text{ciascuno} + \text{costante}} > \Theta(n) \quad \text{non cambia nulla}$$

La somma non può scendere sotto il tempo lineare: deve sempre guardare tutti gli n numeri

[PRO] VOCALI e CONSONANTI DIVIDI ET IMPERA

I = vettore non vuoto di n caratteri

O = numero di coppie $A[i], A[i+1]$ con $1 \leq i \leq n$
tali che $A[i]$ è vocale, $A[i+1]$ è consonante

ContaVC(A, l, r):

if ($l == r$)

return 0

mid = ($l+r$) div 2

ris = ContaVC(A, l, mid) + ContaVC(A, mid+1, r)

if $A[mid]$ is vocale \wedge $A[mid+1]$ è consonante then:

ris = ris + 1

return ris

Tempo di calcolo, sia $n = r - l + 1$:

• caso base

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

• caso passo

$$T(n) = \Theta(1) + 2T\left(\frac{n}{2}\right) = \Theta(n) \rightarrow \text{stessa ricchezza della somma divide et impera}$$

PROBLEMI A CASO

CHE HO PRESO A CASO e NON RICORDO A
QUALE ARGOMENTO APPARTENGONO MA TANTO È UGUALE

[PRO] MATRICE SIMMETRICA I: Una matrice quadrata di dimensioni $n \times n$
O: vero se e solo se la matrice è simmetrica

Simmetrica(A):

for $i=1$ to n do:

 for $j=1$ to n do:

 if $A[i][j] \neq A[j][i]$ then:

 return FALSE

return TRUE

<u>t</u>	<u>T</u>
1	$n+1$
2	$n(n+1)$
2	$n \cdot n$
1	0
0	1

$$t(n) = 1 + 2 + 2 + 1 = 6 = O(1) \text{ quando } A[1][2] \neq A[2][1]$$

$$T(n) = n+1 + n(n+1) + n \cdot n + 0 + 1 = O(n^2) \text{ quando } A_{ij} = A_{ji} \forall i, j \leq n$$

Da cui $\bar{T}(n) : T(n) = \underline{\Omega}(1) \rightarrow$ delimitazione inferiore

$T(n) = O(n^2) \rightarrow$ delimitazione superiore

Anche se, in termini di complessità non cambia niente, possiamo migliorare l'algoritmo come segue:

$$\frac{n^2}{2} - \frac{n}{2}$$

evita il controllo
della diagonale

*(non considera la sottoscala
(evita un doppio controllo))*

[PRO] **ESPOENZIALE RICORSIVO** calcola un esponenziale ricorsivamente:

EsponenzialeRic(n):

if $n == 0$:

 return 1

C₁

{ ris = EsponenzialeRic(n-1)

C₂

 ris = ris * z

C₃ + T(n-1)

 return ris

C₄

C₅

Specifico tutte le istruzioni separandole perché sono interessato ai tempi

Non si ragiona in caso migliore e caso peggiore, ma solo in:

- **caso base** $t(0) = c_1 + c_2 = a$ una certa costante
- **caso passo** $t(1) = c_1 + c_3 + T(0) + c_4 + c_5 = b + T(0) = b + a$

$$t(2) = b + T(1) = 2b + a$$

$$t(3) = b + T(2) = 3b + a$$

generalizzando, $T(n) = nb + a = \Theta(n)$

Il problema può essere ottimizzato sfruttando le proprietà:

$$f(n) = \begin{cases} 1 & \text{se } n = 0 \\ 2^n = 2^{\frac{n}{2}} \cdot 2^{\frac{n}{2}} = f\left(\frac{n}{2}\right) \cdot f\left(\frac{n}{2}\right) = [f\left(\frac{n}{2}\right)]^2 & \text{se } n \text{ è pari} \\ 2^n = 2^{\frac{n}{2}} \cdot 2^{\frac{n}{2}} \cdot 2 = 2 \cdot [f\left(\frac{n}{2}\right)]^2 & \text{se } n \text{ è dispari} \end{cases}$$

EspOTT(n):

if $n == 0$:

C₁

 return 1

C₂

 ris = EspOTT($\lfloor \frac{n}{2} \rfloor$)

C₃ + T($\lfloor \frac{n}{2} \rfloor$)

 ris = ris * ris

C₄

 if $n \bmod 2 == 1$:

C₅

 ris = ris * 2

C₆

 return ris

C₇

• **caso base** $T(0) = c_1 + c_2 = a$

• **caso passo** assumendo n pari e > 0 :

$$T(n) = c_1 + c_3 + T\left(\frac{n}{2}\right) + c_4 + c_5 + c_7 = b^1 + T\left(\frac{n}{2}\right)$$

$$T\left(\frac{n}{2}\right) = c_1 + c_3 + T\left(\frac{n}{4}\right) + c_4 + c_5 + c_7 = b^1 + T\left(\frac{n}{4}\right)$$

$$T(n) = b^1 + T\left(\frac{n}{2}\right) = b^1 + b^1 + T\left(\frac{n}{4}\right) = 2b^1 + T\left(\frac{n}{4}\right) =$$

$$= 2b^1 + b^1 + T\left(\frac{n}{8}\right) = 3b^1 + T\left(\frac{n}{8}\right) \dots$$

$$\dots = i \cdot b^1 + T\left(\frac{n}{2^i}\right) \dots \log n \cdot b^1 + T(1) = \Theta(\log n)$$



mi fermo nella ricerca
quando arrivo al caso base,
ovvero:

$$\frac{2}{n^i} = 1$$

$$n = 2^i \text{ da cui } \log n = i$$

L'algoritmo è migliorato dal punto di vista asintotico

[PRO] **ARRAY ORDINATO** I: array A di interi non vuoto
O: true se e solo se A è ordinato

Ordinato(A, l, r);
if $l == r$ then:
 return true
 $mid = (l+r) \text{ div } 2$
 $ord_l = \text{Ordinato}(A, l, mid)$
 $ord_r = \text{Ordinato}(A, mid+1, r)$
return $ord_l \text{ AND } ord_r \text{ AND } A(mid) \leq A(mid+1)$

[PRO] **CERCA CINQUE** I: due vettori V e W di uguali lunghezze
O: (x, y) : $x = \text{TRUE} \Leftrightarrow \text{e solo se } 5 \in V$
 $y = \text{TRUE} \Leftrightarrow \text{e solo se } 5 \in W$

CercaCinque(v, w, l, r);
if $l == r$ then:
 $x = V[l] == 5$
 $y = W[l] == 5$
 return (x, y)
 $mid = (l+r) \text{ div } 2$
 $(x_l, y_l) = \text{CercaCinque}(v, w, l, mid)$
 $(x_r, y_r) = \text{CercaCinque}(v, w, mid+1, r)$
return $(x_l \vee x_r, y_l \vee y_r)$