Appunti di: Algoritmi e Strutture dati

(By Ulissis)

Questi sono gli appunti presi in classe nelle lezioni della prof.ssa Pinotti (aa 12/13)

NB: possono contenere errori

Sommario

Algoritmo: Sequenza di passi ben definiti con un input e un output.
L'algoritmo deve terminare, quindi deve:
- avere un numero di passi finito
- terminare in un tempo finito
- essere CORRETTO, prove per contraddizione (grafi), per induzione …
- essere EFFICIENTE, si misura guardando le risorse che utilizza (memoria, numero di passi ..)

L'algoritmo ottimo richiede il minor numero di passi a risolvere il problema

Il problema dell'ordinamento
Si suppone di dover ordinare una sequenza di numeri.
Esistono vari tipi di ordinamento, tra questi abbiamo:
1) Insertion-Sort
2) Merge-Sort
3) Heap-Sort
4) Quick-Sort
5) Counting-Sort
6) Bucket-Sort

1) Insertion-Sort (A)

```
1| for j ← 2 to n do {
2|    z ← A[j]; i ← j-1;
3|    while (A[i] > z) and (i > 0) do {
4|          A[i+1] ← A[i]; i--;
5|    }
6|    A[i+1] ← z;
7| }
```

In questo ordinamento si avrà, dato un j segnato, l'array A[1] .. A[j-1] ordinato, e cerco dove inserire l'elemento A[j]

Contiamo il numero di passi:
t_i = costo in tempo

riga 1: n – 2 + 1 (poiché ci sono gli estremi compresi) => = n – 1
si avrà quindi (n – 1) t_1 + t_1
(Questo ultimo t_1 è dovuto al fallimento della guardia)

2: (n - 2 + 1)t_2

3: (while) fissato un $\overline{j}$ si considera i ← $\overline{j}$ - 1
se z è il massimo valore => il while viene eseguito una sola volta
Studiamo la complessità nel caso pessimo (worst-case) => viene eseguito ($\overline{j}$ - 1)t_3
Il costo complessivo è:

$$\sum_{j=2}^{n}(j-1)t_3+\sum_{j=2}^{n}t_3=t_3\sum_{j=2}^{n}(j-1)=t_3\sum_{j'=1}^{n-1}j'=\frac{n(n-1)}{2}$$

4: t_4 $\frac{(n-1)n}{2}$ // A[i+1] ← A[i];

t_5 $\frac{(n-1)n}{2}$ // i--

6: t_6 (n – 1)

$T_A(n)$ indica il tempo dell'algoritmo, T sta appunto per tempo, A per l'algoritmo studiato, e n è la dimensione del problema
Si ricava sommando tutti i vari passi delle varie righe studiate

$T_A(n) = n\,t_1 + (n-1)\,(t_2 + t_3 + t_6) + \frac{(n-1)n}{2}\,(t_3 + t_4 + t_5)$

Si ha quindi la prima parte, composta dai primi due elementi della somma che corrispondo al costo del for (compresa la guardia del while) mentre il terzo elemento è il costo del corpo del while

Avremo quindi:

$$T_A(n) \leqslant (n+n-1\frac{n(n-1)}{2})t' = (\frac{n^2}{2}+\frac{3}{2}n-1)t' < n^2 \cdot t'$$ t' = max {t_1 ... t_6}

Quindi cresce a velocità quadratica.

Note: Gauss

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2}$$

Prova per induzione:

P(1) $\sum_{j=1}^{1} j = 1 = \frac{1\cdot 2}{2} = 1$

P(n) vera da ipotesi

P(n + 1)

$$\sum_{j=1}^{n+1} j = \sum_{j=1}^{n} j + (n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{n+1}{2}(n+2) = P(n+1)$$

Esempio con indici diversi:

$$\sum_{j=5}^{2n} j = \sum_{j=1}^{2n} j - \sum_{j=1}^{4} j = \frac{2n(2n+1)}{2} - \frac{4\cdot 5}{2}$$

Min – Sort (A)

```
1| for i ← 1 to (n – 1) do {
2|    smallpos ← i;
3|    smallest = A[smallpos];
4|    for j ← (i + 1) to n do {
5|          if (A[j] < smallest) then {
6|                smallest ← j;
7|                smallest = A[smallpos];
8|          }
9|    }
10|   A[smallpos] = A[i];
11|   A[i] ← smallest;
12| }
```

Considerando l'array A[1] … A[i] … A[n] cerca tra i+1 e n il minimo elemento e lo scambia con “i” tra A[1] ... A[i – 1]

Complessità:

1: $\sum_{i=1}^{n-1} t_1 + t_1$

2: (n – 1) t_2

3: (n – 1) t_3

4: $\sum_{j=i+1}^{n} t_4 + t_4$ fissato i → (n – i)t_4 + t_4 = (n – i + 1) t_4

=> non considerando più i fissato avremo

$$\sum_{i=1}^{n-1}(n-i+1)t_4 = t_4\sum_{i=1}^{n-1}(n-i+1) = t_4\sum_{i=1}^{n}(i-1) = t_4\left(\frac{n(n+1)}{2}-1\right)$$

5: $$\sum_{i=1}^{n-1}(n-(i+1)+1)t_5 = \sum_{i=1}^{n-1}(n-i)t_5 = \sum_{i=1}^{n} i\,t_5 = \frac{n(n-1)}{2}t_5$$

6 & 7: (worst-case) $\frac{n(n-1)}{2}(t_6+t_7)$

10 & 11: $(n-1)(t_8+t_9)$

$$T_A(n) \leqslant t'\left(n+(n-1)4+\frac{n(n+1)}{2}-1+3\frac{n(n+1)}{2}\right) = t'\left(5n-4+\frac{n^2}{2}+\frac{n}{2}-1+\frac{3n^2}{2}-\frac{3n}{2}\right) = t'(2n^2+4n-5)$$

Crea matrice identità (A)

```
A: [n x n] of integer
1| for i ← 1 to n do
2|    for j ← 1 to n do
3|          A[i,j] ← 0;
4| for i ← 1 to n do
5|    A[i,i] ← 1;
```

Dimensione: n^2

input: z = n^2 → algoritmo lineare nel numero di celle in A

→ quadratico nel numero di righe

Complessità:

1: n t + t

2: $$\sum_{i=1}^{n}(nt+t) = \sum_{i=1}^{n} nt + \sum_{i=1}^{n} t = n^2 t + nt$$

3: n (n t)

4: (n + 1) t

5: n t

$$T_A(n) = (2n^2+4n+2)t$$

Complessità con richiamo a funzione esterne

(Scriviamo come commento la complessità di ogni riga)

```
main: {
      read (n) \\ t
      a ← 0; \\ t
      x ← foo(a,n) \\ T_foo (x,n)
      printf ('hello!') \\ t
}

foo (x, n int) : int {
      i ← 1; x ← 0; \\ t
      for i ← 1 to n do \\ n t + t
            x ← x + bar(i, n) \\ T_bar(x,n)
      return x; \\ t
}

bar (x, n int) : int {
      for i ← 1 to n do \\ n t + t
            x ← x + i \\ n t + t
      return x; \\ t
}
```

$t = \max\{t_i\}$

$T_{bar}(x,n) = (2n + 3)\ t$

$T_{foo}(x,n) = (2n^2 + 4n + 3)\ t$

$T_{main} = (2n^2 + 4n + 6)\ t$

Ricerca binaria iterativa (A)

```
funzione (A, x, p, q) : bool {
      while( p≤q ) do {
            m ← ⌊(p+q)/2⌋ \\ t
      if ( x = A [m] ) \\ t
            then return true;
            else if x < A[m]
                  then p ← m + 1; \\ t (solo t perché ne può far solo
                                        \\uno dei due)
                  else q ← m -1;
      }
      return false;
}
```

Dim: q – p + 1

1° volta = q – p + 1 = n

2° volta ci sono due possibilità

→ q – m - 1 + 1

→ m – 1 – p + 1

$= \lfloor \frac{n}{2} \rfloor$ = q - $\lfloor \frac{p-q}{2} \rfloor = \frac{q-p}{2}$

$\bar{i} : n = 2^{(\bar{i}-1)} \rightarrow \log_2(n+1)$

Crescita delle funzioni

- $f(x) \in O(g(x))$

indica che a partire da un certo valore x_0 è limitata superiormente da g(x) o da un suo multiplo.

Notazione impropria: $f(x) \leqslant O(g(x))$

$\exists c>0, \exists x_0 \geqslant 0 : \forall x \geqslant x_0 \quad si\,ha \quad 0 < f(x) \leqslant c \cdot g(x)$

$x \geqslant x_0$ asintotica, si utilizza una x molto grande

Prima di x_0 può anche essere negativa o superare g(x)

Esempio:

1) $f(x)=2\log_2(x+1)$ $\qquad g(x)=2x$

x	f(x)	g(x)
1	1	2
2	3	4
4	5	8

$\exists c=1, \ \exists x_0=1, \ \forall x \geqslant 1 \quad 0<2\log_2(x+1) \leqslant 2x$

2) $f(x)=2\log_2(x+1)$ $\qquad g(x)=x$

$f(x) \in O(g(x))$? $\qquad si$ scegliendo $c=2$ e $x_0=1$

- $f(x) \in \Omega(g(x))$

vuol dire che f(x) a partire da x_0 è limitata inferiormente da g(x)

Notazione impropria: $f(x) \geqslant \Omega(g(x))$

3)

$f(n)=2\log_2(n)$ $\qquad g(n)=\sqrt{n}$

$f(n) \in \Omega(g(n))$?

$\exists c>0, \ \exists x_0 \geqslant 0, \ \forall x \geqslant x_0 \quad 0 \leqslant c \cdot 2\log_2(n) \leqslant \sqrt{n}$

n_0 deve almeno 1

n	f(n)	g(n)
1	0	1
2	1	1,4
4	2	2
8	3	2,8
16	4	4
32	5	5,6

=> $n_0 \geqslant 16$ $con\, c=1$

se c=2 n_0 dovrà essere molto maggiore di 16

4) f(x)= 3x + 5 g(x)= x

$\exists c\,?, \exists n_0 \geq 0 : \forall x \geq x_0, 0 \leq c \cdot x \leq 3x+5$

c= 1 … 3 va bene

c=4 o maggiore non va bene

5) $\frac{n}{1000} \in O(n)$? *si con* $c = n_0 = 1$, $0 \leq \frac{n}{1000} \leq n$

$\frac{n}{1000} \in \Omega(n)$? *si con* $c = \frac{1}{1000}$ *e* $n_0 \geq 0$

- se f è limitata sia superiormente che inferiormente allora si avrà che:

$f(n) \in \Theta(g(n))$ $\exists\, c_1, c_2, n_0 \geq 0 : \forall\, n \geq n_0$ *si ha* $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2\, g(n)$

Notazione impropria: f(n) = g(n)

Es: 5)* (continuo esempio precedente) quindi si avrà:

$0 \leq \frac{1}{2000} n \leq \frac{n}{1000} \leq n$ $c_1 = \frac{1}{2000}$ *e* $c_2 = 1$

6) $f(n) = 1000 \cdot n^2 + 100 \cdot n \in O(n^3)$

ma $\notin \Omega(n^3)$ → $f(n) \notin \Theta(g(n))$

- $f(n) \in o(g(n))$ → $0 \leq f(n) < c \cdot g(n)$
- $f(n) \in \omega(g(n))$ → $0 \leq c \cdot g(n) < f(n)$

In questi due casi le due funzioni f non toccheranno mai la funzione g.

Esempio:

1) $f(n) = \frac{n(n+1)}{2} \in \Theta(n^2)$?

$f(n) = \frac{n^2}{2} + \frac{n}{2}$ $\exists c_1 \wedge c_2 > 0, n_0 \geq 0 : \forall\, n \geq n_0$ $0 \leq \frac{1}{2} n^2 \leq \frac{n^2}{2} + \frac{n}{2} \leq n^2$

per Ω(n) si avrebbe $n_0 = 0$

per O(n) invece si avrebbe $\frac{n}{2} \leq \frac{n^2}{2} \rightarrow \frac{1}{2} \leq \frac{n}{2} \rightarrow$ $n_0 = 1$

OSS: Se si hanno n_0 diversi si sceglie quello maggiore

quindi si avrà $c_1 = \frac{1}{2}$ $c_2 = 1$ $n_0 = 1$

2) $f(n) = \sum_{i=1}^{\lceil \frac{n}{2} \rceil} i \in \Theta(n^2)$?

$f(n) = \frac{\lceil \frac{n}{2} \rceil (\lceil \frac{n}{2} \rceil + 1)}{2}$

$f(n) = $ ¿$O(n^2)$? **[Note:** $x \leq \lceil x \rceil < x+1$ **]**

$\sum_{i=1}^{\lceil \frac{n}{2} \rceil} i \leq \frac{(\frac{n}{2}+1)(\frac{n}{2}+2)}{2} = \frac{n^2}{8} + \frac{3}{4} n + 1 < n^2$ → $\in O(n^2)$ *con* $c_2 = 1$

risolvo la disuguaglianza per trovare n_0:

$\frac{7}{8}n^2-\frac{3}{4}n-1>0$ da cui troviamo r_1 e r_2

visto che si prende il maggiore avremo $n \geq r_2$

OSS: se r_1 e r_2 sono minori di 0 si prende $n_0 > 0$

$f(n) \in \Omega(n^2)$? $\quad \exists c>0,\ n_0 \geq 0,\ \forall n \geq n_0$

$$\sum_{i=1}^{\lceil \frac{n}{2} \rceil} i \geq c \cdot n^2 \geq 0$$

$$\frac{\lceil \frac{n}{2} \rceil (\lceil \frac{n}{2} \rceil + 1)}{2} \geq \frac{\frac{n}{2}(\frac{n}{2}+1)}{2} = \frac{n^2}{8}+\frac{n}{4} \geq \frac{n^2}{8} \quad \in \Omega(n^2)$$

$\rightarrow f(n) \in \Theta(n^2)$

Proprietà

1) Transitiva

$f(n) \in O(g(n)) \quad g(n) \in O(h(n)) \rightarrow f(n) \in O(h(n))$

2) Simmetrica

$f(n) \in \Theta(g(n)) \rightarrow g(n) \in \Theta(f(n))$

3) Trasposta simmetrica

$f(n) \in O(g(n)) \leftrightarrow g(n) \in \Omega(f(n))$

4) $max\{f(n), g(n)\} \in \Theta(f(n)+g(n))$

Dimostrazione:

$f(n)+g(n)\ (per\ la\ prop.\ simmetrica) \in \Theta(max\{f(n), g(n)\})$

$0 \leq c_1 \cdot max\{f(n), g(n)\} \leq f(n)+g(n) \leq c_2 \cdot max\{f(n), g(n)\}$

con ad esempio: $c_1 = 1$ e $c_2 = 2$

Funzioni possono essere:

- polinomiali $\quad n^k = n^{(o(1))}$
- poli-logaritmiche $\quad \log^k n = \log^{(o(1))} n$
- esponenziali $\quad b^{(f(n))}$

OSS: per controllare che una funzione sia un polinomio si applica il logaritmo

Es: $2^{(5 \cdot \log_2 n)} \rightarrow \log_2 \cdot 2^{(5 \cdot \log_2 n)} = 5 \cdot \log_2 n$ => è un polinomio

Esempi:

1) $\log_2 n \quad \ln n \quad \log_c n \quad c>1$

a) $\log_2 n \in \Theta(\ln n)$?

b) $\ln n \in \Theta(\log_c n)$?

a) $O(\ln n) \rightarrow 0 \leq \log_2 n \leq c \cdot \ln n$ vero se $n_0=1$

[Note: $\log_b n = \frac{\log_c n}{\log_c b}$ **]**

$$\rightarrow \log_2 n = \frac{\ln n}{\ln 2} = \frac{1}{\ln 2} \cdot \ln n \rightarrow c = \frac{1}{\ln 2}$$

$\Omega(\ln n) \rightarrow 0 \leq c \cdot \ln n \leq \log_2 n$ (si svolge come sopra)

b) usiamo lo stesso procedimento trasformando però $\ln n = \frac{\log_c n}{\log_c e}$

per la proprietà transitiva => $\log_2 n \in \Theta(\log_c n)$

OSS: Quindi la base del logaritmo per noi non è mai importante perchè sono sempre limitate

2) $f(n)\ e\ g(n) \in \Theta(2^n)$?

a) $f(n) = 2^{(n+1)}$

b) $g(n) = 2^{(2n)}$

a) $\exists c_1, c_2, n_0 > 0 : \forall n \geq n_0 \quad 0 \leq c_1 \cdot 2^{2n} \leq 2^{(n+1)} \leq c_2 \, codt \, 2^n \quad \text{con } c_1 \leq 2 \ \ e \ \ c_2 \geq 2$

b) $0 \leq c_1 \cdot 2^n \leq 2^{2n} \leq c_2 \cdot 2^n$

non esiste un c_2 che vada bene per ogni valore di n poiché

$2^{2n} = 2^n \cdot 2^n \quad \rightarrow \notin O(2^n)$ ma $\in \Omega(2^n) \rightarrow \notin \Theta(2^n)$

3) $f(n) = 2^{(5\log_2 n)} \in \Theta(2^n)$?

$2^{(5\log_2 n)} = (2^{(\log_2 n)})^5 = n^5$

$\rightarrow \notin \Omega(2^n)$ ma $\in O(2^n)$

$0 \leq c \cdot 2^n \leq n^5 \qquad n_0 \geq 0$

$c \leq \frac{n^5}{2^n}$ → non esiste una costante c per Ω

4) $3^{(\log_9 n)} : n$ → $3^{(\log_9 n)} < n$

$3^{(\log_9 n)} = 3^{(\frac{\log_3 n}{\log_3 9})} = (3^{(\log_3 n)})^{(\frac{1}{2})} = \sqrt{n}$ → $\in O(n)$ *ma* $\notin \Omega(n)$ poichè $0 \leq c \cdot n \leq \sqrt{n}$ → $c \leq \frac{1}{\sqrt{n}}$

5) $(\sqrt{2})^{(\log_8 n)}$ è polinomiale ? si perchè = $\frac{1}{2}\log_8 n$

$f \in \Theta(2^n)$? no perchè sappiamo già che appartiene a $O(n^{(o(1))})$

Cerchiamo il polinomio più vicino:

$(\sqrt{2})^{(\frac{\log_2 n}{\log_2 8})} = 2^{(\frac{1}{2}\frac{\log_2 n}{3})} = \sqrt[6]{n}$

- $\log^*(n)$: { $\log^{(i)}(n) \leq 1$

Def: $\log^{(i)}(n)$ = → log n, se i =1

→ $\log(\log^{(1)}(n))$, se i =2

→ $\log(\log^{(i-1)}(n))$ *se* $i \geq 3$

Es: $\log^*(16) = 3$, $\log^*(2^{16}) = 4$

OSS: $\Theta(b^n) --- \Theta(n^n) --- \Theta(\log^n n) --- \Theta(\log^i n) --- \Theta(1)$

b > 1

Esempi:

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

$\rightarrow \in \Omega(n^2)$ $\rightarrow \in O(n^2)$

$n^2 \leq \sum_{i=1}^{n} i^2 \leq \sum_{i=1}^{n} n^2 = n \cdot n^2$

n^2 perché c'è almeno un termine che vale tale

$n \cdot n^2$ perché sono *n* termini al quadrato

$\sum_{i=1}^{n} i^2 \geq \sum_{i=\frac{n}{2}}^{n} i^2 \geq \sum_{i=\frac{n}{2}}^{n} \left(\frac{n}{2}\right)^2 \geq \left(\frac{n}{2}\right) \cdot \left(\frac{n}{2}\right)^2 = \frac{1}{8} n^3$

$n \geq 2 (¿ n_0)$ $c = \frac{1}{8}$ → $\sum \in \Omega(n^3)$

Note:

$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \ldots \in \Theta(n^3)$

prova per induzione:

P(1) = $\sum_{i=1}^{1} 1^2 = 1$ $\frac{1(2)(3)}{6} = 1$

P(n) vera per ipotesi induttiva

$P(n+1) = P(n) + (n+1)^2$

$\frac{n(n+1)(2n+1)}{6} + (n+1)^2 = (n+1)\left(\frac{n(2n+1)}{6} + (n+1)\right) = (n+1)\left(\frac{2n^2+7n+6}{6}\right) = P(n+1)$

---

2) Provare per induzione che

$\sum_{i=1}^{n} i^2 \in O(n^3)$

caso base i=1 P(1)=1 $\in O(n^3)$

Ipotesi induttiva

$P(n) \in O(n^3)$ $\exists c > 0, n \geq 0 : \forall n \geq n_0$ $P(n) \leq c \cdot n^3$

$P(n+1) = P(n) + (n+1)^2$

$P(n+1) \leq c \cdot n^3 + (n+1)^2 \leq c \cdot (n+1)^3$

$(n+1)^2 \leq 3n^2 + 3n + 1$

$c \geq \frac{n^2+1+2n}{3n^2+3n+1} = \frac{1}{3}$

[Note: $\sum_{i=0}^{n} a^i = \frac{a^{(n+1)} - 1}{a - 1}$ $\forall\ a \in \mathbb{N}^+$ **]**

3) $\sum_{i=0}^{n} 2^i = 2^n + 2^n - 1 = 2^{(n+1)} - 1 \in \Theta(2^n)$

dimostrazione per induzione

$$\sum_{k=1}^{\log_2 n}\left(\frac{n}{2^k}\right)=n\cdot\sum_{k=1}^{\log_2 n}\left(\frac{1}{2^k}\right)=n\cdot\sum_{k=1}^{\log_2 n}\left(\frac{1}{2}\right)^k\le n\cdot\sum_{k=1}^{\infty}\left(\frac{1}{2}\right)^k=\frac{n}{1-\frac{1}{2}}=2\mathrm{n}$$

4) $x<1 \qquad \sum_{k=0}^{n} k\cdot x^k\le\sum_{k=0}^{\infty} k\cdot x^k=\frac{x}{(1-2)^2}$

$$\frac{d}{dx}\,x^k=k\cdot x^{(k-1)}=\sum_{k=0}^{n} x\left(k\cdot x^{(k-1)}\right)=x\cdot\sum_{k=0}^{n}\left(k\cdot x^{(k-1)}\right)=x\cdot\sum_{k=0}^{n}\left(\frac{d}{dx}\cdot x^k\right)=x\frac{d}{dx}\sum_{k=0}^{n}x^k=x\frac{d}{dx}\frac{1}{1-x}$$

5) $$\sum_{k=1}^{n}\left(\frac{1}{k(k+1)}\right)=\sum_{k=1}^{n}\left(\frac{1}{k}-\frac{1}{k+1}\right)=1-\frac{1}{k+1}$$

6) $$\sum_{k=1}^{n}\left(\frac{1}{k}\right)=\sum_{i=0}^{\lfloor\log_2 n\rfloor}\ \sum_{j=0}^{2^i-1}\left(\frac{1}{2^i+j}\right)\le\sum_{i=0}^{\lfloor\log_2 n\rfloor}\ \sum_{j=0}^{2^i-1}\left(\frac{1}{2^i}\right)=\sum_{i=0}^{\lfloor\log_2 n\rfloor}\left(\frac{1}{2^i}2^i\right)=\lfloor\log_2 n\rfloor+1$$

Esercizio:

Ordiniamo le seguenti f in ordine crescente

$\sqrt{n};\quad 2^{(\log_d n)}(\text{con d}\ge 2);\quad 2^{(5+3\log_2(2n))};\quad 2^{(\sqrt{\log_2 n})};\quad 2^{(\log_2(\sqrt{n}))}$

sono tutte f polinomiali → $\in o(2^n)$

- $\sqrt{n}=n^{(\frac{1}{2})}$
- $2^{(\log_d n)}=2^{(\frac{\log_2 n}{\log_2 d})}=\left(2^{(\log_2 n)}\right)^{(\log_2 d)}=n^{(\frac{1}{\log_2 d})}$

  usando il cambio di base potremmo dire che $\frac{1}{(\frac{\log_d d}{\log_d 2})}=\log_d 2$ → $n^{(\log_d 2)}$

però a noi risulta più comodo mantenerlo come prima poiché possiamo dire che:

$\frac{1}{\log_d d}$ → $\sqrt{n}$ $\quad per\ d=4$ (1)

→ n^c $,c<\frac{1}{2}$ $\quad per\ d>4$ (2)

→ n^c $,c>\frac{1}{2}$ $\quad per\ d<4$ (3)

(1) $\sqrt{n}=2^{(\log_4 n)}$

(2) $d>4 \quad \sqrt{n}\in\Omega(n^c)$

(3) $d<4 \quad \sqrt{n}\in O(n^c)$

→ $2^{(\log_d n)}\in o(n)$

- $2^{(5+3\log_2(2n))}=2^{(5+3[\log_2 2+\log_2 n])}=2^{(5+3+3\log_2 n)}=2^{(8+3\log_2 n)}=2^8\left(2^{(\log_2 n)}\right)^3=2^8\ n^3$ → $\in\Theta(n^3)$

  → viene dopo $\sqrt{n}$ e $2^{(\log_d n)}$ **[Note:** $\log(ab)=\log a+\log b$ **]**

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

- $2^{(\sqrt{\log_2 n})}$ (questa f è meno che $\log_2$ → è meno che polinomiale)

$2^{(\frac{\log_2 n}{\sqrt{\log_2 n}})} = n^{(\frac{1}{\sqrt{\log_2 n}})} \in O(1)$ *ma* $\notin \Omega(1)$ *è più piccola di* $n^{(\frac{1}{2})}$

=> $2^{(\sqrt{\log_2 n})} < 2^{(\log_d n)}$ (con d > 4) $< \sqrt{n} < 2^{(\log_2(\sqrt{n}))} < 2^{(\log_d n)}$ (con d < 4) $< 2^{(5+3\log_2(2n))}$

Studiamo $\log(n!) : n^2$

$n! \geq 1^n = 1$

$n! \geq 2^{(n-1)} = \frac{1}{2} 2^n \in \Omega(2^n)$

$n! \geq 3^{(n-2)} = \frac{1}{9} 3^n \in \Omega(3^n)$

dalla seconda riga ricaviamo che $c \cdot 2^n \leq n! \leq n^n$ → $n! \in O(n^n)$ *ma* $\in \Omega(n^n)$?

no perché: $n! \geq (\frac{n}{2})^{(\frac{n}{2})} = (\frac{1}{2})^{(\frac{n}{2})} \cdot n^{(\frac{n}{2})} = (\frac{1}{2n})^{(\frac{n}{2})} \cdot n^n \geq c \cdot n^n$

non si ha quindi una costante poiché c dipende da n =>

$\notin \Omega(n^n)$ *ma* $\in \Omega(2^n)$

Es: 6! = 6*5*4*3*2*1

=> abbiamo $(\frac{n}{2})^{(\frac{n}{2}+1)}$

con cui abbiamo sviluppato il passaggio per la ricerca della costante

applicando il logaritmo avremo

$\log(c \cdot 2^n) \leq \log(n!) \leq \log(n^n)$

$n \cdot \log c \cdot 2$ → $\log(c \cdot 2) = c'$ → $c' \cdot n \leq \log(n!) \leq n \cdot \log n$

$\Omega(n) \leq \log(n!) \leq O(n \cdot \log n)$

$\log(n!) \in \Theta(n \cdot \log n)$?

$(\frac{n}{2})^{(\frac{n}{2})} \leq n! \leq n^n$

$\frac{n}{2} \cdot \log(\frac{n}{2}) \leq \log(n!) \leq n \cdot \log n$

$\frac{n}{2} \cdot \log(\frac{n}{2}) \geq \frac{n}{4} \cdot \log n$ (a partire da un certo n_0)

perché $\frac{n}{2} \cdot \log(\frac{n}{2}) = \frac{n}{2}(\log n - \log 2) \geq \frac{n}{2}(\frac{\log n}{2})$ → $\frac{1}{4} \cdot n(\log n)$

[Note: $\log(\frac{c}{d}) = \log c - \log d$ **]**

→ $c \cdot n \cdot \log n \leq \log(n!) \leq n \cdot \log n$ → $\log(n!) \in \Theta(n \cdot \log n)$

Es:

```
procedure  esempio (n : int) {
      int k ← 0; // O(1)
      (1)while(k  ≤  n) {    //O(n) perché viene eseguito n+1+1
      (2)   for(j ← 1; j  ≤  3^k; j++)    //3^k +1
                  t ++;              //3^k
```

```
(3)  for(r ← 1; r  ≤  k; r ++)   //k+1
     (3)       for(j ← 1; s  ≤  k; s++)    //k(k+1) perché viene eseguito
                    A[r, s] ← 0            //k² per ogni valore di r
          k++;      //O(1)
}
```

(2) $\sum_{j=1}^{3k} 1+1 = c\cdot 3^k+1$ poiché il costo del corpo di questo for è 1 dovendo solamente incrementare t

(3) $\sum_{r=1}^{k}(\sum_{s=1}^{k} O(1)+1)+1 = \sum_{r=1}^{k}(c\cdot k+1)+1 = k(c\cdot k+1)+1 = c\cdot k^2+k+1 \le (c+1)k^2$ → $\in O(k^2)$

(1) while =>

$$T_A(n) \le \sum_{k=0}^{n}(3^k+1+c'\cdot k^2)+1 = \sum_{k=0}^{n} 3^k + \sum_{k=0}^{n} 1 + c'\cdot\sum_{k=0}^{n} k^2+1 \le \frac{3^{(n+1)}}{2}+(n+1)+n^2(n+1)+1 \in O(3^n)$$

ci siamo arrivati tramite le seguenti operazioni:

1) $\sum_{k=0}^{n} 3^k = \frac{3^{(n+1)}-1}{3-1} = \frac{3^{(n+1)}-1}{2} \le \frac{3}{2}3^n = c\cdot 3^n \in O(3^n)$

2) $\sum_{k=0}^{n} k^2 = 0^2+1^2+\ldots+n^2$ → ciascuno dei termini è più piccolo di n^2 → $\le (n+1)n^2$

usando la regola $f(n)+g(n) \in O(max\, f(n), g(n))$ → $\in O(3^n)$

Quale è il suo Ω ?

non c'è un caso migliore però possiamo dire che:

$$T_A(n) \ge \sum_{k=0}^{n}(3^k+1+c\cdot k^2+k+1)+1 \qquad (c\cdot k^2+k+1 \ge c\cdot k^2)$$

$$\ge \sum_{k=0}^{n} 3^k+1+\sum_{k=0}^{n} c\cdot k^2+1 = \frac{3^{(n+1)}-1}{2}+(n+1)+c\cdot\sum_{k=0}^{n} k^2+1 \qquad (\sum_{k=0}^{n} k^2 \ge n^2)$$

→ $\ge \frac{3^{(n+1)}-1}{2} \ge \frac{3^{(n+2)}}{2} \ge \frac{3^{(n+1)}}{2}-1 \ge \frac{3^{(n+1)}}{4} = \frac{3}{4}3^n$

Equazioni di ricorrenza

```
(1) es
Procedure S(i : integer) {
     if i = 1 then return 2;
     if i = 2 then return 3;
     else  S ← S(i-1)*S(i-2)
}
```

$$T_S(i) = \rightarrow \Theta(1) \quad se\ i=1,2$$

$$\rightarrow T_S(i-1)+T_S(i-2)+\Theta(1) \quad i \ge 3$$

(il secondo theta indica il costo per la moltiplicazione e per l'assegnazione)

Useremo il **metodo per sostituzione**
=> sappiamo che
$T(i-1)=T(i-2)+T(i-3)+c$
$T(i-2)=T(i-3)+T(i-4)+c$
=> T(i) (con i'=1) = $T(i-2)+2T(i-3)+T(i-4)+3c(\leftarrow(2^2-1)c)$
i'=2 => $T(i-3)+T(i-4)+c+2(T(i-4)+T(i-5)+c)+T(i-5)+T(i-6)+c+3c$
= $T(i-3)+(2^1+2^0)T(i-4)+(2^1+2^0)T(i-5)+T(i-6)+(2^3-1)c$
i'=3 => $T(i-4)+T(i-5)+c+(2^1+2^0)[T(i-5)+T(i-6)+c]+(2^1+2^0)[T(i-6)+T(i-7)+c]+$
$+2^0[T(i-7)+T(i-8)+c]+(2^3-1)c$
= $T(i-4)+4T(i-5)+6T(i-6)+4T(i-7)+T(i-8)+(2^4-1)c$
=> $(2^i-1)c$
avremo un triangolo di tartaglia di altezza i , fino ad arrivare a T(1) e T(2) ($\in\Theta(1)$) +
$+(2^i-1)$ c ($\in O(2^i)$) ← ci da una dimensione del numero di passi che facciamo

T(i) arriva a una costante quando:

$$i-i'=2 \rightarrow i'=i-2 \rightarrow 2^{(i-2)}=\frac{2^i}{4}\in O(2^i)$$

i' = numero di iterazioni
Dopo i' = i-2 sono sicuro che tutti i T(i) sono termini costanti avendo
$(2^{(i'+1)}-1)c=(2^{(i-1)}-1)c$

Altro metodo, sempre con sostituzione, ma più semplice:
Posso osservare che
T(i) = $T(i-1)+T(i-2)+c2\geq T(i-1)+c$
=> $T(i-1)2\geq T(i-2)+c$
Vado a sostituire
I sost. => $\leq 2[2T(i-2)+c]+c=20T(i-2)+3c$
T(i-2) $\leq 2T(i-3)+c$
II sost. => $\leq 2^2[2T(i-3)+c]+3c=2^3T(i-3)+7c$
III sost. => $\leq 2^3[2T(i-4)+c]+7c=2^4T(i-4)+15c$ ($\leftarrow 15c=(2^4-1)$)

=> i' sost. avremo $\leq 2^{(i'+1)}T(i-(i'+1))+(2^{(i'+1)}-1)c$
T è una costante quando $i'-(i+1)=2$ cioè quando $i'=i-3$
=> $2^{(i-3+1)}T(2)+(2^{(i-3+1)}-1)c\leq\bar{c}\,2^i$
$T(i)\leq 2^{(i-2)}T(2)+2^{(i-2)}c$ (trascuro -1 perchè ho maggiorato)
=> $=2^{(i-2)}\Theta(1)+2^{(i-2)}c=2^i(\frac{c'}{4}+\frac{c}{4})\in O(2^i)$

una volta trovata la stima della nostra T bisogna provarla per induzione
$T(i)\leq 2T(i-1)+c\in O(2^i)$
$\exists c>0, i_0>0 : \forall i\geq i_0 \quad T(i)\leq c(2^i-1)$
base induttiva:
$T(1)\leq c\cdot 2^1=\Theta(1)$ vero
$T(2)\leq c\cdot 2^2=\Theta(1)$ vero
Ipotesi induttiva:
$T(i-1)\leq c(2^i-1)$

=> $T(i) \leq 2c(2^{(i-1)}-1)+c \leq c \cdot 2^i - 2c + c \leq c \cdot 2^i - c = c(2^i - 1)$ → $\in O(2^i)$

(2) es

```
Procedure S(n : int){
      if (n ≤ 100) then stampa("Svegliati!");
       else S ← S (n/2) + 123;
}
```

```
T(n) = → 1       se n ≤ 100
     →T(n/2)+c  se n > 100              ('c' conta la somma e
l'assegnamento dell'else)
```

Metodo sostituzione

Possiamo assumere che $n=2^k$ poiché tra n e 2n c'è sicuramente una potenza di 2, questo ci libera dal portarci dietro la parte intera inferiore

$T(n) \leq T(\frac{n}{2})+c$

I sost. $\leq (T(\frac{n}{4})+c)+c$

i' sost. $\leq T(\frac{n}{2^{(i+1)}})+(i-1)c$ → $T(\leq 100)+(i'+1)c = \lceil \log_2(\frac{n}{100}) \rceil c$ → $\in O(\log_2 n)$

La conclusione della precedente equazione è data dalla ricerca del valore di i per cui T diventa una costante, questo i lo scriviamo come i'

$\frac{n}{2^{(i+1)}} \leq 100$ → $2^{(i+1)} over 2n \geq \frac{1}{100}$ → $2^{(i+1)} \geq \frac{n}{100}$ → $i+1 \geq \log_2(\frac{n}{100})$ → $i \geq \lceil \log_2(\frac{n}{100}) \rceil - 1$

Dimostrazione per induzione

$\exists c>0, n_0>0 : \forall n \geq n_0 \qquad T(n) \leq c \cdot \log_2 n$

Base induttiva:

$T(100) \leq c \cdot \log_2 100 \in O(1)$

Hp:

$T(\frac{n}{2}) \leq c \cdot \log_2(\frac{n}{2})$

=> $T(n) \leq T(\frac{n}{2})+c \leq c \cdot \log_2(\frac{n}{2})+c = c(\log_2 n - \log_2 2) = c \cdot \log_2 n$ vero

Metodo dell'albero della ricorsione

$T(n) \leq T(\frac{n}{2})+c$

radice → dell' albero — T(n) => (costo) c

chiamata → che viene fatta — T(n/2) => c

…

T(≤ 100)

Il costo complessivo è il costo delle foglie che una costante più tutti gli 'c' ,sapendo l'altezza dell'albero so quanti costi sommare.

L'altezza quando $(\frac{n}{2}) \leq 100$ → $i \geq \lceil \log_2(\frac{n}{100}) \rceil$ → costo= $c \cdot \lceil \log_2(\frac{n}{100}) \rceil$

Esempio 1:

$T(n) \leq 2T(n-1)+c$

```
                T(n)  => c
                /     \
            T(n-1)      T(n-1)  => c + c = 2c
              /     \         /     \
                T(n-2)      “     “       “   => 4c
                …
```

i-esima riga avremo => 2^i c

devo scoprire quanto vale 'i' (che è l'altezza dell'albero):

$n-i \leq 2 \rightarrow i \geq n-2$

=> costo complessivo $2^{(n-1)}\, O(1)+\sum_{i=0}^{n-3} 2^i \cdot c = 2^{(n-1)}+2^{(n-2)}-1$

Esempio 2:

T(n) = → 2 T(n-5) + 6 se $n \geq 10$

→ 10 se $n < 10$

```
            T(n)          => 6
          /     \
             T(n-5)  “    => 2*6          T(n-5) = 2 T(n-5-5) + 6
          /     \
     T(n-10)               => 4*6          T(n-10) = 2 T(n-15) + 6
        /     \
           …
```

$T(n-5 \cdot i)$

al passo i avrò 2^i chiamate ricorsive => ho un costo di $6 \cdot 2^i$

a quale iterazione i $n-5i<10$?

$\frac{n-10}{5} < i \rightarrow i > \lfloor \frac{n}{5} \rfloor - 2$

quando $i = \lceil \frac{n}{5} \rceil - 2$ T(<10) ha un costo costante di 10

il costo complessivo dell'algoritmo è:

$T(n) = \frac{2^{(\lceil \frac{n}{5} \rceil)}}{4} 10 + \sum_{i=0}^{\lceil \frac{n}{5} \rceil - 3} 6 \cdot 2^i \leq \frac{2^{(\lceil \frac{n}{5} \rceil)}}{4} 10 + 6 \frac{2^{(\lceil \frac{n}{5} \rceil)}}{4} \leq 4 \cdot 2^{(\lceil \frac{n}{5} \rceil)} \in O(2^n)$ **[**

$\sum_{i=0}^{t} b^i = \frac{b^{(t+1)}-1}{b-1}$ (per b>1) **]**

ho usato una maggiorazione così da non dover riscrivere “-1” nel risultato della sommatoria

Esempio 3:

```
Procedure ES(n : int){
     if(n ≤ 5) then
          for i ← 1 to n do     // O(1) (*1)
               A[i] ← 7;
          else  a ← 2*ES(n-5);
}
```

(*1) perché il for per essere eseguito ha bisogno di n ≤ 5, quindi avrà sempre un numero costante di passi

T(n) → O(1) se n ≤ 5
→ T(n-5) + $\Theta(1)$ se n > 5

T(n) => $\Theta(1)$
|
T(n-5) => $\Theta(1)$
|
T(n-10) => $\Theta(1)$
|
. . .
|
T(n-5 · i)
|

T(<5) → $n-5i<5$ → $i>\lceil\frac{n}{5}\rceil-1$

Costo complessivo:

$$T(n)=\sum_{i=0}^{\lceil\frac{n}{5}\rceil-1}\Theta(1)+O(1)=\lceil\frac{n}{5}\rceil\Theta(1)+O(1)$$

=> $\leq c(\lceil\frac{n}{5}\rceil+1)lec\cdot n(\text{con } n_0>5)$ → $\in O(n)$

=> $\geq c\cdot\lceil\frac{n}{5}\rceil\geq c\frac{n}{5}$ → $\in\Omega(n)$

Prova per induzione

$T(n)\in O(n)$

$\exists c>0, n_0>0: \forall n\geq n_0 \quad T(n)\leq c\cdot n$

Base induttiva:

$n\leq 5 \quad T(n)\leq c\cdot n\leq c\cdot 5=O(1)$

Hp:

$T(n-5)\leq c(n-5)$

$$T(n)=T(n-5)+\Theta(1)\leq c(n-5)+\Theta(1)=c\cdot n-5\cdot c+\Theta(1)\leq c\cdot n \leftrightarrow -5+\Theta(1)<0$$
$$\rightarrow 5\cdot c-\Theta(1)>0 \rightarrow c>\frac{\Theta(1)}{5}$$

Verifichiamo per induzione anche $\Omega(n)$

$\exists c>0, n_0>0: \forall n\geq n_0 \quad T(n)\geq c\cdot n\geq 0$

Base: $n\leq 5 \quad T(n)=O(1)\geq c\cdot n \in\Omega(1)$

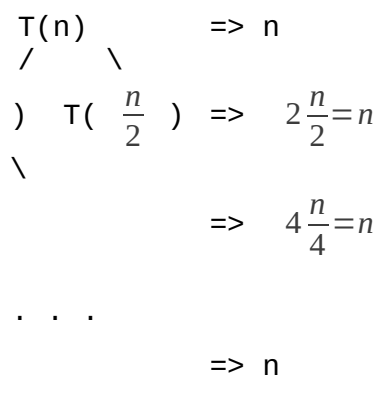
Hp: $T(n-5)\geq c(n-5)$

$$T(n)=T(n-5)+\Theta(1)\geq c(n-5)+\Theta(1)\geq c\cdot n \leftrightarrow -5c+\Theta(1)>0 \leftrightarrow c<\frac{\Theta(1)}{5}$$

Esempio 4:

T(n) → O(1) se n ≤ 4

→ $2T(\frac{n}{2})+n$ se n > 4



$\frac{n}{2^i} \le 4 \rightarrow 2^i \ge \frac{n}{4} \rightarrow i \ge \lceil \log_2(\frac{n}{4}) \rceil$

$$T(n) = \sum_{i=0}^{\lceil \log_2(\frac{n}{4}) \rceil - 1} n + \frac{n}{4} = n\lceil \log_2(\frac{n}{4}) \rceil + \frac{n}{4} = n(\lceil \log_2(\frac{n}{4}) \rceil + \frac{1}{4}) \in O(n \cdot \log_2 n)$$

Prova per induzione:

$\exists c>0, n_0>0 : \forall n \ge n_0 \quad T(n) \le c \cdot n \cdot \log_2 n$

Hp: $T(\frac{n}{2}) \le c \frac{n}{2} \log_2 \frac{n}{2}$

$$T(n) = 2T(\frac{n}{2}) + n \le 2 \cdot c \frac{n}{2} \log_2 \frac{n}{2} + n \le c \cdot n(\log_2 n - \log_2 2) + n = c \cdot n \log_2 n - c \cdot n + n \le c \cdot n \cdot \log_2 n$$

$\leftrightarrow \rightarrow -c \cdot n + n < 0 \rightarrow c > 1$

Esempio 5:

```
ES( n : integer ){
     int k ← 0;       // O(1)
     while(k ≤ n) {       //*(1)
          for(int r=1; r ≤ 4; r++)        //*(2)
               for(int s=1; s ≤ k-1;  s++)          //*(3)
                    a++;
          k ← k + 2;
     }
     if( n > 5 ) return ES( ⌊n/2⌋ ) + ES( ⌊n/2⌋ ) + 3;
     else return 5;
}
```

O(?)

T(n) → n ≤ 5 O(1)

→ n > 5 2T($\lfloor \frac{n}{2} \rfloor$)

*(1)
i' = passo in cui mi fermo (cioè in cui non entro nel while)

i' = min{ i : 2(i-1) > n } = 2i –2>n => $i>\lceil\frac{n+2}{2}\rceil$ usiamo l'upper-bound perchè “i” deve essere un intero

$\sum_{t=1}^{\lceil\frac{n+2}{2}\rceil-1}$ → $\sum_{t=1}^{\lceil\frac{n}{2}\rceil}(\ \ )+1$ usiamo il “-1” sopra per contare quante volte entro nel while

la sommatoria diventa così perché => $\lceil\frac{n+2}{2}\rceil=\lceil\frac{n}{2}\rceil\frac{2}{2}$

*(2)
$\sum_{r=1}^{k}$ → $\sum_{r=1}^{2(t-1)-1}(\ \ )+1$ k si modifica perché devo scriverlo in funzione dei passaggi e quindi k=2(t – 1)

*(3)
$\sum_{s=1}^{2(t-1)-1} O(1)+1$

$$\Rightarrow \quad T(n)=\sum_{t=1}^{\lceil\frac{n}{2}\rceil}(\sum_{r=1}^{2t-3}(\sum_{s=1}^{2t-3}1+1)+1)+1+2T(\lfloor\frac{n}{2}\rfloor)$$

$$=\sum_{t=1}^{\lceil\frac{n}{2}\rceil}(\sum_{r=1}^{2t-3}(\sum_{s=1}^{2t-3}1+\sum_{s=1}^{2t-3}1)+1)+1+2T(\lfloor\frac{n}{2}\rfloor)$$

$$=\sum_{t=1}^{\lceil\frac{n}{2}\rceil}\sum_{r=1}^{2t-3}\sum_{s=1}^{2t-3}1+\sum_{t=1}^{\lceil\frac{n}{2}\rceil}\sum_{r=1}^{2t-3}1+\sum_{r=1}^{\lceil\frac{n}{2}\rceil}1+1+2T(\lfloor\frac{n}{2}\rfloor)$$

$$=\sum_{t=1}^{\lceil\frac{n}{2}\rceil}(2t-3)^2+\sum_{t=1}^{\lceil\frac{n}{2}\rceil}(2t-3)+\lceil\frac{n}{2}\rceil+1+2T(\lfloor\frac{n}{2}\rfloor)$$

$((2t-3)^2=4t^2+9-12t\le 4t^2$ gia da $t_0=1$ è vero perche al crescere di t la parte negativa aumenta$)$

$$\le\sum_{t=1}^{\lceil\frac{n}{2}\rceil}4t^2+\sum_{t=1}^{\lceil\frac{n}{2}\rceil}2t+\lceil\frac{n}{2}\rceil+1+2T(\lfloor\frac{n}{2}\rfloor)$$

$$\le 4(\lceil\frac{n}{2}\rceil)^2\lceil\frac{n}{2}\rceil+2\lceil\frac{n}{2}\rceil\lceil\frac{n}{2}\rceil+\lceil\frac{n}{2}\rceil+1+2T(\lfloor\frac{n}{2}\rfloor)$$

da → $(\sum_{t=1}^{\lceil\frac{n}{2}\rceil}4t^2\le\sum_{t=1}^{\lceil\frac{n}{2}\rceil}4(\lceil\frac{n}{2}\rceil)^2)$

$$\le 8n^3+2T(\lfloor\frac{n}{2}\rfloor)$$

da $n_0=2$, perchè ogni elemento è sicuramente più piccolo di n^3 da $\le(\frac{n}{2}+1)^3\le n^3$

=> T(n) → n ≤ 5*(4)

→ n > 5 $8n^3 + 2T(\lfloor\frac{n}{2}\rfloor)$

*(4) avrei $8n^3$ ma con n ≤ 5 funziona solo la parte iterativa quindi non è più un n che cresce

=> $8n^3\le 8\cdot 5^3$ => O(1)

$T(n) \leq 8n^3 ... \leq 8n^3 + 2T(\frac{n}{2})$

calcoliamo la stima tramite albero di ricorsione

T(n) => $8n^3$

/ \

$T(\frac{n}{2})$ => $2 \cdot 8(\frac{n}{2})^3$

/ \

$T(\frac{n}{4})$ =>

. . . => $2^i \cdot 8(\frac{n}{2^i})^3$

$T(\frac{n}{2^i})$

altezza albero: $\frac{n}{2^i} \leq 5$ → $\frac{n}{5} \leq 2^i$ → $\lceil \log_2 \frac{n}{5} \rceil \leq i$

#foglie 2^i → $2^{\lceil \log_2 \frac{n}{5} \rceil} \leq 2^{(\log_2 \frac{n}{5} + 1)} \leq 2\frac{n}{5}$

per ciascuna foglia si ha T(foglia) = O(1)

=> lavoro alle foglie #foglie * T(foglia) $\leq \frac{2}{3} O(1) n \leq c \cdot n \in O(n)$

costo complessivo:

$T(n) = \text{costo foglie} + \text{costo ai livelli} = O(n) + 8 \cdot \sum_{i=0}^{\lceil \log_2 \frac{n}{5} \rceil - 1} (2^i \frac{n^3}{2^{(3i)}}) = 8 \cdot \sum_{i=0}^{\lceil \log_2 \frac{n}{5} \rceil - 1} (\frac{n^3}{2^{(2i)}}) + O(n)$ → $\in O(n^3)$

facendo il rapporto tra “i” e “i-1” si vede se è una geometrica =>

$\frac{n^3}{2^{(3i)}} \frac{2^{(3(i-1))}}{n^3} = \frac{1}{4}$ ← è una costante => è geometrica, la cui soluzione della sommatoria quindi è $\frac{1}{1 - \frac{1}{4}} = \frac{4}{3}$

=> la soluzione sarebbe $\frac{4}{3} n^3$ → $\in O(n^3)$

Esempio 6:

T(n) → O(1) se n ≤ 1

→ $T(\lceil \frac{n}{2} \rceil) + a$ a > 0, se n > 1

all'i-esimo passo di questa funzione avremo $T(\lceil \frac{n}{2} \rceil) + a$

$\sum_{i=0}^{\lceil \log_2 n \rceil} a + O(1) \leq c \cdot \log_2 n$ → $\in O(\log n)$ con $c = max\{a, O(1)\}$

$T(n) = T(\lceil \frac{n}{2} \rceil) + a \leq T(\frac{n}{2}) + a$ ~non si può scrivere perchè non vale $\lceil x \rceil \leq x$ ma $\lceil x \rceil \leq x + 1$

=>

$\leq T(\frac{n}{2}+1)+a \leq T(\frac{2}{3}n) \in O(\log n)$ a partire da un $n_0=6$ $(\frac{n}{2}+1 \leq \frac{2}{3}n \rightarrow 3n+6 \leq 4n \rightarrow n \geq 6)$

Lo dimostriamo per induzione:

Tesi: $T(n)=T(\lceil \frac{n}{2} \rceil)+a \in O(\log n)$

Hp: $T(\frac{2n}{3}) \leq c \cdot \log(\frac{2n}{3})$

$T(n)=T(\lceil \frac{n}{2} \rceil)+a \leq T(\frac{n}{2}+1)+a \leq T(\frac{2}{3}n)+a \leq c \cdot \log(\frac{2}{3}n) \leq c \cdot \log_2 + c \cdot \log \frac{2}{3} + a \leq c \cdot \log_2 n$

questo è vero solo se $c \cdot \log \frac{2}{3}+a<0 \rightarrow -c \cdot \log \frac{2}{3} - a \rightarrow c > \frac{a}{-\log(\frac{2}{3})}$

Teorema dell'esperto

$T(n)=aT(\frac{n}{b})+f(n)$ (a ≥1 e b>1 costanti)

Possiamo applicare questo teorema se ci troviamo in uno dei seguenti casi:

1) $f(n)=O(n^{(\log_b a-\epsilon)})$ $(\epsilon>0)$ → $T(n)=\Theta(n^{(\log_b a)})$ (ad ogni livello faccio meno lavoro delle foglie)

2) $f(n)=\Theta(n^{(\log_b a)})$ → $T(n)=\Theta(n^{(\log_b a)} \log n)$ (ad ogni livello lavoro quanto le foglie)

3) $f(n)=\Omega(n^{(\log_b a+\epsilon)})$ $(\epsilon>0)$ → $T(n)=\Theta(f(n))$ (ad ogni livello faccio più lavoro delle foglie)

questo terzo caso è vero solo se vale anche la seguente condizione:

$a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ (con c<1)

dimostrazione nel caso $n=b^k$ così da poter trascurare le parti intere superiori e inferiori

T(n) => f(n)
/ / \ \ } a figli
T($\frac{n}{b}$) => $a \cdot f(\frac{n}{b})$
/ / \ \ } a figli per ogni figlio precedente
T($\frac{n}{b^2}$)

. . . => $a^i \cdot f(\frac{n}{b^i})$

i-esimo T($\frac{n}{b^i}$)

$\frac{n}{b^i} \leq 1$ (assumendo che T(n) per n=1 ∈O(1)) → $i \geq \log_b n$ → $i \geq k$ (da $n=b^k$)

→ $a^{(\log_b(n-1))} \cdot f(\frac{n}{b^{(\log_b(n-1))}})$

$T(n)=O(1) \cdot a^{(og_b n)} + \sum_{i=0}^{\log_b n-1} a^i \cdot f(\frac{n}{b^i})$ facciamo un cambio di base

→ $= a^{(\frac{\log_a n}{\log_a b})} + \sum_{i=0}^{\log_b n-1} a^i \cdot f(\frac{n}{b^i}) = (a^{(\log_a n)})^{(\frac{1}{\log_a b})} + \sum_{i=0}^{\log_b n-1} a^i \cdot f(\frac{n}{b^i}) = c \cdot n^{(\log_b a)} + \sum_{i=0}^{\log_b n-1} a^i \cdot f(\frac{n}{b^i})$

abbiamo fatto un altro cambio di base per l'ultimo passaggio

c indica il costo di ciascuna foglia

Dimostrazione:

1) $T(n)=O(1)\cdot n^{(\log_b a)}+\sum_{i=0}^{\log_b n-1} a^i\cdot O\left(\left(\frac{n}{b^i}\right)^{(\log_b a-\epsilon)}\right) \le O(1)\cdot n^{(\log_b a)}+d\cdot\sum_{i=0}^{\log_b n-1} a^i\cdot\left(\frac{n}{b^i}\right)^{(\log_b a-\epsilon)}$

maggioriamo rispetto a un dato $n\ge n_0$

d è la costante della O() che stava dentro la sommatoria

Studiamo a parte la sommatoria:

$\sum_{i=0}^{\log_b n-1} a^i\frac{n^{(\log_b a-\epsilon)}}{b^{(i(\log_b a-\epsilon))}}=\sum_{i=0}^{\log_b n-1} a^i\frac{n^{(\log_b a-\epsilon)}}{b^{(i\cdot\log_b a)}\cdot b^{(-\epsilon\cdot i)}}$ $\quad (b^{(i\cdot\log_b a)}=a^i)$

→ $=n^{(\log_b a-\epsilon)}\cdot\sum_{i=0}^{\log_b n-1}\left(b^\epsilon\right)^i=n^{(\log_b a-\epsilon)}\frac{\left(b^\epsilon\right)^{(\log_b n)}-1}{\left(b^\epsilon\right)-1}=n^{(\log_b a-\epsilon)}\frac{n^\epsilon-1}{b^\epsilon-1}<n^{(\log_b a-\epsilon)}\cdot n^\epsilon$

$\left(\frac{n^\epsilon-1}{b^\epsilon-1}>\frac{n^\epsilon}{2(b^\epsilon-1)}\right)$ → $n^{(\log_b a-\epsilon)}\frac{1}{2(b^\epsilon-1)}\le n^{(\log_b a-\epsilon)}\frac{n^\epsilon-1}{b^\epsilon-1}<n^{(\log_b a)}$ (considerando che $\frac{1}{2(b^\epsilon-1)}$ è una costante)

2) $T(n)=n^{(\log_b a)}+\Theta\left(\sum_{i=0}^{\log_b n-1} a^i\cdot\left(\frac{n}{b^i}\right)^{(\log_b a)}\right)$

possiamo dire che $n^{(\log_b a)}+d(\quad)\le T(n)\le n^{(\log_b a)}+c(\quad)$

Studiamo a parte la sommatoria:

$\sum_{i=0}^{\log_b n-1} a^i\frac{n^{(\log_b a)}}{\left(b^{(\log_b a)}\right)^i}=n^{(\log_b a)}\sum_{i=0}^{\log_b n-1} 1=n^{(\log_b a)}\cdot\log_b n$

3) $T(n)=n^{(\log_b a)}+\sum_{i=0}^{\log_b n-1} a^i\cdot f\left(\frac{n}{b^i}\right)$

deve valere la condizione di regolarità

(con c<1) $a\cdot f\left(\frac{n}{b}\right)\le c\cdot f(n)$ → $f\left(\frac{n}{b}\right)\le\frac{c}{a}\cdot f(n)$

(consideriamo n=$\frac{n}{b}$) → $f\left(\frac{n}{b^2}\right)\le\frac{c}{a}\cdot f\left(\frac{n}{b}\right)\le\frac{c}{a}\left[\frac{c}{a}\cdot f(n)\right]=\left(\frac{c}{a}\right)^2\cdot f(n)$

lavorando per induzione → $f\left(\frac{n}{b^i}\right)\le\left(\frac{c}{a}\right)^i\cdot f(n)$

→ $T(n)\le n^{(\log_b a)}+\sum_{i=0}^{\log_b n-1} a^i\cdot\left(\frac{c}{a}\right)^i\cdot f(n)\le n^{(\log_b a)}+f(n)\cdot\sum_{i=0}^{\log_b n-1} c^i$

(essendo c<1) → $\le n^{(\log_b a)}+f(n)\frac{a}{1-c}$

essendo $\Omega\left(n^{(\log_b a+\epsilon)}\right)$ → $f(n)$ pesa più di $n^{(\log_b a)}$ → $\le d\cdot f(n)$

Esempio 1:

$T(n)=T\left(\frac{n}{2}\right)+n \qquad a=1 \quad b=2 \quad f(n)=n$

conto il numero di foglie

$n^{(\log_b a)}=n^{(\log_2 1)}=n^0=1$ → una foglia sola

valuto il loro peso rispetto a $f(n)$ → $1: f(n)=n$

$f(n)\in O\left(n^{(0+1)}\right)$ → $\epsilon=1$ → si applica il 3° caso

dobbiamo prima verificare la condizione:

$1\,\frac{n}{2}\le c\cdot n$ → si con $c=\frac{1}{2}$ → $\in\Theta(n)$

Esempio 2:

$T(n)=3T(\frac{n}{3})+\sqrt{n}$

#foglie = $n^{(\log_b a)}=n^{(\log_3 3)}=n$

$n : \sqrt{n}$ → caso 1, infatti $\sqrt{n} ¿= n^{(1-\frac{1}{2})}$ con $\epsilon=\frac{1}{2}$ → $T(n)\in\Theta(n)$

Esempio 3:

$T(n)=3T(\frac{n}{3})+\frac{n}{\log n}$ $f(n)\notin O(n^{(1-\epsilon)})$ perchè $\frac{n}{\log n}\leq\frac{n^1}{n^\epsilon}$ non è vero

allora non si può trovare una ϵ => non si può risolvere con il M.T. (Master Theorem)

Esempio 4:

$T(n)=7T(\frac{n}{6})+n^{(\frac{3}{2})}$

#foglie => $n^{(\log_b a)}=n^{(\log_6 7)}$ (che è un pò più di n^1) => caso 3

$n^{(\log_6 7+(\frac{3}{2}-\log_6 7))}$

verifico la condizione

$a\cdot f(\frac{n}{b})=7(\frac{n}{6})^{(\frac{3}{2})}\leq c\cdot n^{(\frac{3}{2})}?$ si con $c=7\frac{1}{6^{(\frac{3}{2})}}<1$ → $T(n)\in\Theta(n^{(\frac{3}{2})})$

Esempio 5:

$T(n)=T(\frac{n}{2})+\log n$

#foglie = 1

f(n) = log n > 1 quindi dovremmo essere nel 3° caso però $\notin\Omega(n^{(0+\epsilon)})$ perché log n cresce più della costante 1, ma cresce meno di n^ϵ anche con ϵ molto piccolo.

Quindi usiamo il metodo dell'albero di ricorsione perché non si può applicare il M.T.

T(n) => log n
|
T($\frac{n}{2}$) => $\log\frac{n}{2}$
|
. . . => $\log\frac{n}{2^i}$
|
T($\frac{n}{2^i}$) $i\geq\log_2 n$

$T(n)=O(1)+\sum_{i=0}^{\log_2 n+1}\log(\frac{n}{2^i})\leq O(1)+\sum_{i=0}^{\log_2 n+1}\log n$ (è una serie decrescente quindi sostituisco con il primo term

→ $\in O(\log^2 n)$

Proviamo ad usare un altro metodo per risolvere la sommatoria così da cercare di risparmiare qualcosa

$$T(n)=O(1)+\sum_{i=0}^{\log_2 n+1}(\log_2 n-\log_2 i)=O(1)+\sum_{i=0}^{\log_2 n+1}\log_2 n-\sum_{i=0}^{\log_2 n+1} i=O(1)+(\log_2 n)^2-\log_2 n(\log_2 n-1)$$
$$\in O(\log n)^2$$

=> andava bene anche il metodo precedente poiché siamo arrivati allo stesso risultato

Esempio 6:

$T(n)=$ → $T(\sqrt{n})+1$ *per* $n\geq 2$
→ O(1) *per n*<2

posso scriverlo come $aT(\frac{n}{b})+1$ *con* $a\geq 1$ *e* $b>1$*?* NO

il costo sarà minore a log n perché scendo più rapidamente verso una costante rispetto a $\frac{n}{2}$

altezza : n n
|
$\sqrt{n}$ $n^{(\frac{1}{2})}$
|
$\sqrt{\sqrt{n}}$ $n^{(\frac{1}{4})}$

i-esimo passo => $n^{(\frac{1}{2^i})}$

=> $n^{(\frac{1}{2^i})}<2$ ➔ $i\geq\log_2(\log_2 n)\in O(\log n)$ poichè $\log^a n<n^\epsilon$ e $\log^a(\log n)<(\log n)^\epsilon$

Esempio 7:

$T(n)=$ → $T(\frac{n}{3})+T(\frac{2n}{3})+n$ *per* $n>1$
→ O(1) *per n*=1

non possiamo applicare il M.T.

Però osserviamo che $T(n)=T(\frac{n}{3})+T(\frac{2n}{3})+n\leq 2T(\frac{2n}{3})+n(=T'(n))$

$T'(n)$ ➔ $a=2$ $b=\frac{3}{2}$ $f(n)=n$ quindi si può applicare il M.T.

In pratica abbiamo maggiorato la nostra T per ricavare una nuova T' a cui potevamo applicare il M.T. , ma ciò rende il risultato che troviamo una stima, quindi andrà poi provata per induzione

#foglie = $n^{(\log_b a)}=n^{(\log_{\frac{3}{2}} 2)}$ $(\log_{\frac{3}{2}} 2>1)$ ➔ $n^{(1,\dots)}$

=> 1° caso con $\epsilon=-1+\log_{\frac{3}{2}} 2>0$

$T'(n)\in\Theta(n^{(\log_{\frac{3}{2}} 2)})$ ➔ $T(n)\in O(n^{(\log_{\frac{3}{2}} 2)})$

ora consideriamo $T(n)>T(\frac{2n}{3})+n(=T''(n))$

#foglie = $n^0=1$ => 3° caso
=> $T''(n)\in\Theta(n)$ ➔ $T(n)\in\Omega(n)$

facciamolo ora con l'albero di ricorsione, possiamo già notare a priori che non sarà mai un albero bilanciato

T(n) => n

/ \

$T(\frac{n}{3})$ $T(\frac{2n}{3})$ => $\frac{n}{3}+\frac{2n}{3}=n$

/ \ / \

$T(\frac{n}{9})$ $T(\frac{2n}{9})$ $T(\frac{4n}{9})$ => $\frac{n}{9}+\frac{2n}{9}+\frac{2n}{9}+\frac{4n}{9}=n$

. . .

=> pagheremo ad ogni livello n

Il nodo tutto a destra durerà di più (avrà cioè un altezza maggiore)
Cioè l'altezza massima non sarà un livello completo, ma quando il nodo più lungo sarà arrivato a T(1) allora sicuramente quella sarà l'altezza massima

=> quando $\frac{n}{(\frac{3}{2})^i} \leq 1$ → $i \geq \log_{\frac{3}{2}} n$

sono 2^i nodi per livello => $\lceil \text{tot nodi} \rceil = 2^{(\log_{\frac{3}{2}} n)} = (\text{tramite cambio di base}) = n^{(\log_{\frac{3}{2}} 2)}$

$T(n) = \#\text{foglie} + n \cdot \log_{\frac{3}{2}} n$ $\#\text{foglie} < n^{(\log_{\frac{3}{2}} 2)}$

→ $T(n) < n^{(\log_{\frac{3}{2}} 2)} + n \log_{\frac{3}{2}} n$

che è come scrivere $n^{(1+\beta)} + n^1 \cdot \log_{\frac{3}{2}} n$

=> la stima migliore che posso fare è $n \log n$

Vogliamo quindi provare che $T(n) \in O(n \log n)$ per induzione

$\exists c > 0, n_0 > 0 : \forall n \geq n_0$

- $T(\frac{n}{3}) \leq c \frac{n}{3} \log(\frac{n}{3})$
- $T(\frac{2n}{3}) \leq c \frac{2n}{3} \log(\frac{2n}{3})$ le assumiamo per la stessa costante c

cioè che $T(n) \leq c \cdot n \log n$

$T(n) \leq c \frac{n}{3} \log(\frac{n}{3}) + c \frac{2n}{3} \log(\frac{2n}{3}) + n \leq c \frac{n}{3} \log(\frac{2n}{3}) + c \frac{2n}{3} \log(\frac{2n}{3}) + n$

abbiamo sostituito $\frac{n}{3}$ con $\frac{2n}{3}$ maggiorando perché il logaritmo è una funzione monotona crescente

$= c \cdot n \cdot \log(\frac{2n}{3}) + n = c \cdot n \cdot \log n + c \; cdpt \; n \cdot \log(\frac{2n}{3}) + n \leq c \cdot n \cdot \log n \; ?$

si, solo se $c \cdot n \cdot \log(\frac{2n}{3}) + n \leq 0$ → (divido per n) $c \cdot \log(\frac{2n}{3}) \leq 0$ → $c > \frac{1}{-\log(\frac{2}{3})}$

posso farlo perché $\log(\frac{2}{3})$ è un numero negativo

Vogliamo dimostrare anche che $T(n) \in \Omega(n \log n)$

$\exists c > 0, n_0 > 0 : \forall n \geq n_0$

- $T(\frac{n}{3}) \geq c \frac{n}{3} \log(\frac{n}{3})$
- $T(\frac{2n}{3}) \geq c \frac{2n}{3} \log(\frac{2n}{3})$

$T(n) \geq c\ \frac{n}{3}\log(\frac{n}{3}) + c\ \frac{2n}{3}\log(\frac{2n}{3}) + n \geq c\ \frac{n}{3}\log(\frac{n}{3}) + c\ \frac{2n}{3}\log(\frac{n}{3}) + n = c \cdot n \cdot \log(\frac{n}{3}) + n = c \cdot n \cdot \log n - c \cdot n \cdot log3$

questo è $> n \cdot \log n$, solo se $c \cdot n \cdot log3 > 0$ → $c \cdot log3 < 0$ → $c < \frac{1}{log3}$ → $\in \Theta(n \log n)$

-Possiamo quindi considerare che se ci troviamo di fronte a una forma del tipo $T(\frac{a \cdot n}{b}) + T(\frac{c \cdot n}{b}) + n$ e $\frac{a}{b} + \frac{c}{b} = 1$
=> si può maggiorare e minorare la nostra T e risolvere con il M.T.
Rispetto a $T(\frac{\alpha \cdot n}{b})$ con $\alpha = max\{a\ ,\ c\}$

Esempio 8:

$T(n) = 3T(\frac{n}{2} + 2) + n$

per fare la stima ci avvaliamo del M.T. Dicendo che la nostra funzione è

$\simeq 3T(\frac{n}{2}) + n$

quindi avremo a = 3 , b = 2 e f (n) = n
$n^{(\log_2 3)} : n$ il lavoro delle foglie è maggiore => caso 1

$f(n) \in O(n^{(\log_2 3 - \epsilon)})$ con $\epsilon = -1 + \log_2 3$ → $T(n) \in O(n^{(\log_2 3)})$

per induzione vogliamo dimostrare che

$T(n) = 3T(\frac{n}{2} + 2) + n \in O(n^{(\log_2 3)})$

$\exists c > 0, n_0 > 0 : \forall n \geq n_0 \qquad T(\frac{n}{2} + 2) \leq c \cdot (\frac{n}{2} + 2)^{(\log_2 3)}$

$T(n) \leq 3c(\frac{n}{2} + 2)^{(\log_2 3)} + n \leq 3c[\quad] + n$

dovremmo trovare un numero minore a uno, ma non siamo riusciti a trovare tale valore per cui valga, anche se (la prof) è sicura che ci sia
=> proviamo a dimostrarlo con

$T(\frac{n}{2} + 2) \leq c \cdot (\frac{n}{2} + 2)^2$ cioè se $\in O(n^2)$

$T(n) \leq 3c(\frac{n}{2} + 2)^2 + n \leq 3c(\frac{n^2}{4} + 4 + 2n) + n \leq c\frac{3}{4}n^2 + 3c(4 + 2n) + n \leq c\frac{n^2}{4}$

→ $c\frac{n^2}{4} - 3c(4 + 2n) - n \geq 0$ → $c(\frac{n^2}{4} - 12 - 6n) \geq n$ → $c \geq \frac{4n}{n^2 - 48 - 24n}$

quando n cresce f è decrescente allora mi basta il primo valore che rende il tutto positivo (es: 100)

Esempio 9:

$T(n) =$ → $2T(\frac{n}{2} + 17) + n$ per $n > 4$

→ O(1) per $n \leq 4$

facciamo la stima con M.T. => $T(n) \in O(n \log n)$

verifichiamo per induzione:

$\exists c>0, n_0>0 : \forall n \geq n_0 \quad T(\frac{n}{2}+17) \leq c\cdot(\frac{n}{2}+17)\log(\frac{n}{2}+17)$

$T(n) \leq 2c(\frac{n}{2}+17)\log(\frac{n}{2}+17)+n$

sfruttiamo il fatto che il log è una f monotona crescente quindi

$\frac{n}{2}+17 \leq \frac{n}{2}+\frac{n}{4}=\frac{3n}{4}$ (vero per $n_0=17\cdot 4$) → $\leq 2c(\frac{n}{2}+17)\log(\frac{3n}{4})+n$

$\leq 2c(\frac{n}{2}+17)\log n+2c(\frac{n}{2}+17)\log\frac{3}{4}+n = cn\log n+34c\log n+2c(\frac{n}{2}+17)\log\frac{3}{4}+n$ è $\leq c\cdot n\cdot\log n$?

si se $34c\log n+cn\log\frac{3}{4}+34\log\frac{3}{4}+n<0$ (ci salva) $cn\log\frac{3}{4}$ perchè è negativo

$c(34\log n+n\log\frac{3}{4})+34\log\frac{3}{4}+n<0$ → $c>\frac{34\log\frac{3}{4}+n}{-34\log n-n\log\frac{3}{4}}$ basta il primo n (>64)

che renda c > 0

Esempio 10:

$T(n)=2T(\frac{n}{4})+\sqrt{n} \qquad a=2 \quad b=4 \quad f(n)=\sqrt{n}$

$n^{(\frac{1}{2})}=n^{(\log_4 2)} : n^{(\frac{1}{2})}$ **=>** 2° caso => $\Theta(\sqrt{n}\cdot\log n)$

Esempio 11:

$T(n)=3T(\frac{n}{3})+\log n \qquad a=3 \quad b=3 \quad f(n)=\log n$

$n^{(\log_3 3)}=n$ => caso 1

$\log n = O(n^{(1-\epsilon)})=\frac{n^1}{n^\epsilon}$ con $\epsilon=\frac{1}{2}$ → $\log n \in O(n^{(\frac{1}{2})})$ → $T(n)\in\Theta(n)$

Esempio 12:

$T(n)=4T(\frac{n}{2})+\frac{n^2}{\log n}$ è applicabile il M.T. ?

#foglie = $n^2 \qquad f(n)=\frac{n^2}{\log n}$

ai livelli pago meno che alle foglie => caso 1

ma dovrei dire che $f(n)\in O(n^{(2-\epsilon)})$ ma $f(n)$ è $\frac{n^2}{\log n}$ => io vorrei che $\frac{n^2}{\log n}\leq c\frac{n^2}{n^\epsilon}$

cioè che $\log n > n^\epsilon$ ma non è così => $f(n)\notin O(n^{(2-\epsilon)})$ => non si può applicare il M.T.

Quindi usiamo il metodo dell'albero di ricorsione, da cui ricaviamo che altezza: $\log_2 n$

i-esimo passo avremo $T(\frac{n}{2^i})$ con costo al livello di $2^i\cdot[\frac{\frac{n^2}{2^i}}{\log(\frac{n}{2^i})}]=\frac{n^2}{\log(\frac{n}{2^i})}$

$T(n)=n^2O(1)+\sum_{i=0}^{\log_2(n-1)}\left(\frac{n^2}{\log(\frac{n}{2^i})}\right)$ $\quad (O(1)\cdot n^2\in O(n^2))$ → $T(n)\le O(n^2)+\sum_{i=0}^{\log_2(n-1)} n^2\le n^2\log n+O(n^2)$

nella sommatoria ho sostituito i con il suo valore massimo quindi

$2^{(\log_2(n-1))}=\frac{2^{(\log_2 n)}}{2}=\frac{n}{2}$ → avrei $\log_2(\frac{n}{\frac{n}{2}})=\log_2 2=1$

Proviamo un modo più "fine":

$T(n)\le O(n^2)+\sum_{i=0}^{\log_2(n-1)}\left(\frac{n^2}{\log n-i}\right)\le O(n^2)+n^2\cdot\sum_{i=0}^{\log_2(n-1)}\left(\frac{1}{\log n-i}\right)$

faccio un cambio di variabile $t=\log_2(n-i)$ → $\sum_{i=0}^{\log_2(n-1)}\left(\frac{1}{\log n-i}\right)=\sum_{t=1}^{\log_2 n}\left(\frac{1}{t}\right)$

→ $\le O(n^2)+n^2\cdot\sum_{i=1}^{\log_2 n}\left(\frac{1}{i}\right)\le O(n^2)+n^2\cdot\log(\log n)$ → $O(n^2\cdot\log(\log n))$

che è minore rispetto al risultato trovato prima

NOTE:

1) $\sum_{i=0}^{\log_2(n-1)}\log(\frac{n}{2^i})=\sum_{i=0}^{\log_2(n-1)}(\log n-i)$ non ci si guadagna niente a spezzarlo, conviene solo se il logaritmo sta al denominatore, come nell'esercizio precedente

2) $\log(\log n)\in O(\log n)\in O(n)$

Esempio 13:

$T(n)=$ → 5 se $n\le 3$

→ $3T(\frac{n}{9})+n^2$ $\quad se\ n>3$

#foglie = $n^{(\frac{1}{2})}: n^2=f(n)$ => 3° caso

$3(\frac{n}{9})^2\le c\cdot n^2$ → $c=\frac{1}{27}$ → $T(n)\in\Theta(n^2)$

Dimostrazione per induzione

$T(n)\in O(n^2)$ Tesi: $\exists c>0, n_0>0:\forall n\ge n_0$ $\quad T(n)\le c\cdot n^2$

Hp: $T(\frac{n}{9})\le c\cdot(\frac{n}{9})^2$

$\le 3c(\frac{n}{9})^2+n^2$ → $\frac{1}{27}cn^2+n^2\le c\cdot n^2$?

$n^2\le\frac{26}{27}cn^2$ → $c\ge\frac{27}{26}$

Esempio 14:

Dimostriamo per induzione che: $T(n)=T(n-1)+n\in O(n^2)$

Tesi: $\exists c>0, n_0>0:\forall n\ge n_0$ $\quad T(n)\le c\cdot n^2$

Hp: $T(n-1)\le c\cdot(n-1)^2$

$T(n)=T(n-1)+n\le c(n-1)^2+n=c(n^2+1-2n)+n$ → $cn^2+c-2cn+n\le cn^2$?

si se $c(2n-1)-n\geq 0$ → $c\geq\frac{n}{2n-1}$ questa è una funzione decrescente allora basta scegliere una costante. Ad esempio per $n_0=4$ la mia funzione sarà sempre sotto $\frac{4}{7}$

Esempio 15:

$T(n)=$ → O(1) se $n\leq 2$

→ $9T(\frac{n}{3})+n$ se n > 2

#foglie = n^2

$$T(n)=O(n^2)+\sum_{i=0}^{\log_3 n-1}(\frac{9}{3})^i n=O(n^2)+n\cdot\sum_{i=0}^{\log_3 n-1}3^i=O(n^2)+\frac{3^{(\log_3 n)}-1}{2}n\leq O(n^2)+\frac{n^2}{2}\ \text{→}\ \in O(n^2)$$

Se lo risolvessimo con il M.T saremmo nel caso 1 => $T(n)\in\Theta(n^2)$

Dimostrazione per induzione

Tesi: $\exists c>0, n_0>0:\forall n\geq n_0 \quad T(n)\leq c\cdot n^2$

$T(n)=9T(\frac{n}{3})+n$ (sfruttando l'ipotesi induttiva) $\leq 9c(\frac{n}{3})^2+n$ → $=cn^2+n\leq cn^2$?

ciò non è vero, pensiamo che il problema sia nella nostra tesi, dovevamo scrivere $T(n)\leq c\cdot n^2+b\cdot n$ che è comunque un ordine di n^2

$T(n)=9[c(\frac{n}{3})^2+b\frac{n}{3}]+n$ → $cn^2+3bn+n\leq cn^2+bn$?

$2bn+n\leq 0$ → $b\leq\frac{-1}{2}$ e qualunque c > 0

Esempio 16:

$T(n)=$ → 6 se $n\leq 2$

→ $T(\frac{n}{2})+T(\frac{n}{3})+n^2$ se $n\geq 3$

Sfruttando il M.T possiamo dire che:

$*(1)=2T(\frac{n}{3})+n^2\leq T(\frac{n}{2})+T(\frac{n}{3})+n^2\leq 2T(\frac{n}{2})+n^2=*(2)$

*(1) #foglie = $n^{(\log_3 2)}$ => 3° caso => $\in\Omega(n^2)$

*(2) #foglie = n => 3° caso => $\in O(n^2)$

non possiamo dire che sono Θ poiché abbiamo maggiorato/minorato il T(n) iniziale cercando così una stima

Dimostriamo per induzione che $\in O(n^2)$, nell'induzione dobbiamo usare T(n) e non la funzione che abbiamo trovato maggiorando

Tesi: $\exists c>0, n_0>0:\forall n\geq n_0 \quad T(n)\leq c\cdot n^2$

Hp: $T(\frac{n}{2})\leq c(\frac{n}{2})^2$; $T(\frac{n}{3})\leq c(\frac{n}{3})^2$

$T(n)=T(\frac{n}{2})+T(\frac{n}{3})+n^2 \leq c(\frac{n}{2})^2+c(\frac{n}{3})^2+n^2 = cn^2(\frac{1}{4}+\frac{1}{9})+n^2$ → $cn^2\frac{13}{36}+n^2\leq c\cdot n^2$?

$n^2\leq cn^2\frac{23}{36}$ → $0\leq c\frac{23}{36}$ → $c\geq\frac{36}{23}$

Esempio 17:

$T(n)=$ → O(1) se $n\leq 2$

→ $T(\frac{n}{2})+T(\frac{n}{3})+n$ se $n\geq 3$

Sfruttando il M.T possiamo dire che:

$*(1)=2T(\frac{n}{3})+n\leq T(\frac{n}{2})+T(\frac{n}{3})+n\leq 2T(\frac{n}{2})+n=*(2)$

*(1) #foglie = $n^{(\log_3 2)}$: n => 3° caso => $\in \Omega(n^2)$

*(2) #foglie = n => 2° caso => $\in O(n\log n)$ che è un valore sicuramente troppo alto causato dalla maggiorazione

quindi controlliamo per induzione che $T(n)\in O(n)$

Tesi: $\exists c>0, n_0>0 : \forall n\geq n_0 \quad T(n)\leq c\cdot n$

$T(n)\leq c\frac{n}{2}+c\frac{n}{3}+n$ → $c\frac{5}{6}n+n\leq c\cdot n$?

$n\leq \frac{1}{6}c\,n$ → $c\geq 6$

Esempio 18:

$T(n)=$ → O(1) se $n\leq 4$

→ $2\mathrm{T}(\sqrt{n})+\log_2 n$ se $n\geq 4$

usiamo l'albero di ricorsione

T(n) => $\log_2 n$

/ \

$T(\sqrt{n})$ => $2\log_2\sqrt{n}=\log n$

/ \

$T(n^{(\frac{1}{4})})$ => $4\log_2 n^{(\frac{1}{4})}=\log n$

i-esimo passo avremo $T(n^{(\frac{1}{2^i})})$ con un costo al livello di $\log_2 n$

=> $n^{(\frac{1}{2^i})}<4$ → $2^2>2^{(\frac{\log_2 n}{2^i})}$ → $2>\frac{\log_2 n}{2^i}$ → $2^{(i+1)}>\log_2 n$ → $i>\log_2(\log_2 n)-1$

[$\log^2 n\neq \log^{(2)} n$ poichè sono rispettivamente uguali a $(\log n)(\log n)$ e $\log(\log n)$ **]**

$n^{(\log_2(\log_2 n)-1)}+\sum_{i=0}^{\log_2(\log_2 n)-1}\log n$ → $T(n)\in O(\log n\log\log n)$

Dimostrazione per induzione

Tesi: $\exists c>0, n_0>0 : \forall n\geq n_0 \quad T(n)\leq c\cdot\log n\log\log n$

$T(n)\leq 2c\log_2\sqrt{n}\cdot\log(\log\sqrt{n})+\log_2 n=c\log_2 n[\log_2(\frac{1}{2}\log_2 n)]+\log_2 n$

$c\log_2 n[\log_2\frac{1}{2}+\log_2\log_2 n]+\log_2 n$ → $c\log_2 n\log_2\log_2 n+c\log_2 n(-1)+\log_2 n\leq c\log n\log\log n$?

→ $-c+1\leq'$ → $c\geq 1$

Esempio 19:

$T(n)=T(n-1)+T(\frac{n}{2})+n \qquad n>1$

$n^2 < T(n-1)+n \le T(n) \le 2T(n-1)+n \ \ (=T'(n))$

```
          T'(n) =>    n
          /     \
    T '(n-1)   =>   2 (n - 1)
```

i-esimo passo avremo $T'(n-i)$ con un costo ai livelli di $2^i(n-i)$
altezza= $n-\alpha$ (con α=costante, in questo caso = 1)
#foglie = $2^{(n-1)}$

$T'(n)=2^{(n-1)}+\sum_{i=0}^{n-1}2^i(n-1) < 2^{(n-1)}+\sum_{i=0}^{n-1}2^i n \le 2^{(n-1)}+n\cdot\sum_{i=0}^{n-1}2^i \le 2^{(n-1)}+\frac{2^n-1}{1}n \ \rightarrow \in O(n\cdot 2^n)$

Prova per induzione:

$\exists c>0, n_0>0 : \forall n \ge n_0 \qquad T(n)\le c\cdot n\cdot 2^n$

$T(n)\le c(n-1)2^{(n-1)}+c\ \frac{n}{2}\ 2^{(\frac{n}{2})}+n \ \le \ c\cdot n\cdot 2^n \quad ?$

$c(n-1)\frac{2^n}{2}+c\ \frac{n}{2}\ 2^{(\frac{n}{2})}+n \ \le \ c\cdot n\cdot 2^n \ \rightarrow \ n\le c[n\cdot 2^n-\frac{n-1}{2}2^n-\frac{n}{2}\ 2^{(\frac{n}{2})}]$

$\rightarrow \ c\ge \frac{2n}{(2n-n+1)2^n-n\cdot 2^{(\frac{n}{2})}}$ è una funzione decrescente allora basta scegliere una costante per cui valga questa cosa, ad esempio: $n_0=4$

I seguenti esercizi sono alcune tipologie che potrebbero capitare all'esonero

1) Provate o confutate che se $g(n) \in \Theta(f(n)) \rightarrow f(n)-g(n)\in O(1)$
facciamo un controesempio ponendo $g(n)=n+\log n \ \rightarrow \ n<g(n)<2n$
$g(n)\in\Theta(n) \ \rightarrow \ f(n)=n$
$(n+\log n)-n=\log n \in O(1) \ ?$ no
altro controesempio:
$g(n)=2n^2 \quad g(n)\in\Theta(n^2) \quad f(n)=n^2 \quad g(n)-f(n)=n^2 \notin O(1)$

2) Provare che $\exists c>0 : 2^{(\Theta(\log_b n))}\in O(n^c)$
$c_2\log_b n\le\Theta(\log_b n)\le c_1\log_b n$ faccio un cambio di base e porto a potenza di due =>

$2^{(c_2\frac{\log_2 n}{\log_2 b})}\le 2^{(\Theta(\log n))}\le 2^{(c_1\frac{\log_2 n}{\log_2 b})} \ \rightarrow \ 2^{(\log_2 n\frac{c_2}{\log_2 b})}\le 2^{(\Theta(\log n))}\le 2^{(\log_2 n\frac{c_1}{\log_2 b})} \ \rightarrow \ n^{(\frac{c_2}{\log_2 b})}\le 2^{(\Theta(\log n))}\le n^{(\frac{c_1}{\log_2 b})} \ \rightarrow \ \exists c$

3) $2^{(\Theta(n))}\in O(2^n) \ ?$
cioè esistono c_1/c_2 tali che $c_2 2^n\le 2^{(\Theta(n))}\le c_1 2^n$
$\bar{c}_2 2^n\le 2^{(\Theta(n))}\le \bar{c}_1 2^n \ \rightarrow \ 2^n\le 2^{(\Theta(n))}\le 2^{(\bar{c}_1 n-n)}\cdot 2^n$
$\rightarrow \ \in\Omega(2^n)$ ma $2^{(\bar{c}_1 n-n)}\ne costante \ \rightarrow \notin O(2^n)$

4) ordinare le seguenti funzioni

$f_1(n)=\sum_{i=1}^{\log_2 n}\left(\frac{n}{2^i}\right) = n\cdot\sum_{i=1}^{\log_2 n}\left(\frac{1}{2}\right)^i \le 2n$

$f_2(n)=\sum_{i=1}^{n-1}\left(\frac{n^2}{i}\right) = n^2\cdot\sum_{i=1}^{n-1}\left(\frac{1}{i}\right) < n^2\log n$

$f_3(n)=2^{(8\log_4 n)} = 2^{\left(8\,\frac{\log_2 n}{\log_2 4}\right)}=(2^{(\log_2 n)})^4=n^4$ → $f_1 \le f_2 \le f_3$

5) Ordinare le seguenti funzioni

$\frac{n}{\log n}$; $n^{(\log_3 2)}$; $n^{(\log_2 3)}$; $n\log n$

=> $n^{(\log_3 2)}=n^{(1-\epsilon)}$ con $0<\epsilon<1$ → $=\frac{n}{n^\epsilon}<\frac{n}{\log n}$; $n^{(\log_2 3)}=n^{(1+\epsilon')}$ con $0<\epsilon<1$

[$\log^b n<n^a$ per quanto $b>a$ **]**

ricapitolando avremo $n^{(\log_3 2)}\le\frac{n}{\log n}\le n\log n\le n^{(\log_2 3)}$

Note: alcuni appunti sulle funzioni con log e log*

$\frac{n}{\log n}\le\frac{n}{\log(\log n)}\le n\log(\log n)\le n\log n$

$\log(\log^* n)\le\log^*(\log n)\le\log^* n$

$\log^* n = \log^*(\log n)+1$ (perchè è come se fossero $\log^{(i)}$ e $\log^{(i-1)}$)

$2^{(\log^* n)}$: $2^{(\log^*(\log n))}$ → $2^{(\log^*(\log n)+1)}$: $2^{(\log^*(\log n))}$ → $2^{(\log^*(\log n))}\cdot 2$: $2^{(\log^*(\log n))}$ → $2^{(\log^* n)}\in O(2^{(\log^*(\log n))})$

$2^{(\log^* n)}<\log^* n$? falso

---

6) $T(n)=aT(\frac{n}{b})+n\in\Theta(n^3)$?

per quali valori di a e b è vero ?
Per forza dobbiamo rientrare nel caso 1 del M.T.

$n^{(\log_b a)}=n^3$ → $\log_b a=3$ → $a=b^3$

7) Confrontare T e T' al variare dei parametri

$T'(n)=T(\frac{n}{b})+\log n$

$T'(n)$ #foglie=1 3° caso →n^0 : $n^{(0-\epsilon)}=\frac{1}{n^\epsilon}$

$a\log(\frac{n}{b})\le c\log n$; $c<1$

$(\log n-\log b)\le c\log n$ → $(c-1)\log n\le-\log b$ quindi non vale la condizione perché mi ritroverei con $\log b\ge\log n$ che non è possibile perché $\log n$ cresce più velocemente di una costante

=> usiamo l'albero

$T(n)$ => $\log n$

|

$T(\frac{n}{b})$ => $\log \frac{n}{b}$

|

. . .

$T(\frac{n}{b^i})$ => $\log \frac{n}{b^i}$

Costo complessivo:

$$\sum_{i=0}^{\log_b n-1} \log \frac{n}{b^i} = \sum_{i=0}^{\log_b n-1} (\log n - i) \leq \log_b^2 n - . \frac{(\log_b^{2n})}{2} \leq \frac{1}{2} \log_b^2 n \rightarrow T'(n) \in O(\log^2 n)$$

=> qualunque sia il valore di a e b T '(n) < T(n)

8) Scrivere una procedura ricorsiva Dummy la cui complessità sia T(n) $\in O(n^2)$

T(n) = T(n-1) + n

```
Dummy (  int n ) {
      if (n ≤ 3) return 7;
      else {
            Dummy ← Dummy (n -1) + 3n;
            for ( i =1; i < n; i++ ) { a ← 0; }
      }
}
```

se la funzione richiesta fosse stata non ricorsiva bastavano due for uno dentro l'altro

9)

```
 Pippo ( int : n ) : integer {
      int k ← n;
      int a ← 0;
      while ( k ≥ 4 ) do {
            for( i ← 1; i ≤⌊(log2 k / log2(log2 k))⌋ ; i++) { a++; }
            k ←  √k ;
      }
}
```

k	t
n	0
$\sqrt{n}$	1
$n^{(\frac{1}{4})}$	2
$n^{(\frac{1}{2^i})}$	i

$n^{(\frac{1}{2^i})} \geq 4 \rightarrow \frac{1}{2^i}\log_2 \geq 2 \rightarrow \log_2 n \geq 2^{(i+1)} \rightarrow \log\log n \geq i+1 \rightarrow i \leq \log\log n - 1$

$$\sum_{t=0}^{\log\log n-1}\Big(\sum_{i=1}^{\lfloor(\frac{\log n^{(\frac{1}{2^t})}}{\log(\log n^{(\frac{1}{2^t})})}\rfloor} O(1)\Big) \leq \sum_{t=0}^{\log\log n-1}\Big(\frac{\log n^{(\frac{1}{2^t})}}{\log(\log n^{(\frac{1}{2^t})})}\Big) = \sum_{t=0}^{\log\log n-1}\Big(\frac{\frac{1}{2^t}\log n}{\log(\frac{1}{2^t}\log n)}\Big) = \sum_{t=0}^{\log\log n-1}\Big(\frac{\frac{1}{2^t}\log n}{\log\frac{1}{2^t}+\log\log n}\Big)$$

$$= \sum_{t=0}^{\log\log n-1}\Big(\frac{\frac{1}{2^t}\log n}{\log\log n - t}\Big) \leq \log n \sum_{t=0}^{\log\log n-1}\Big(\frac{\frac{1}{2^t}}{\log\log n - t}\Big)$$

$\log\log n$ => t = 0 log log n
=> t = log log n = 1 = t'

=> t' = log log n – t

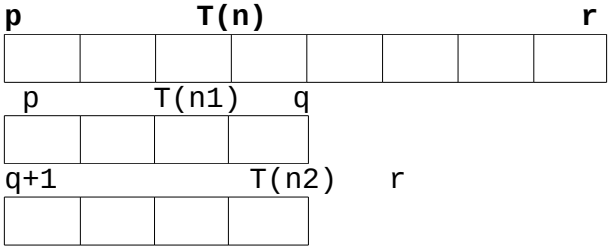
$$\Rightarrow \quad \log n \sum_{t'=1}^{\log\log n}\Big(\frac{1}{\frac{2^{(-t'+\log\log n)}}{t'}}\Big) = \log n \sum_{t=0}^{\log\log n-1}\Big(\frac{\frac{2^{(t')}}{\log n}}{t'}\Big)$$

$$\Rightarrow \quad \frac{\sum_{t'=1}^{\log\log n} 2^{(t')}}{t'} < \sum_{t'=1}^{\log\log n} 2^{(t')} < \sum_{t'=1}^{\log\log n} 2^{(\log\log n)} = \sum_{t'=1}^{\log\log n} \log n = \log\log n \log n$$

il risultato è corretto ma maggiorato
possiamo fare una cosa più fine

$$\Rightarrow \quad < \frac{2^{(\log\log n+1)}-1}{1} = 2\log n \quad \Rightarrow \text{Pippo} \quad \in O(\log n)$$

Torniamo a considerare gli Algoritmi di Ordinamento, abbiamo visto nelle lezioni precedenti
1) Insertion-Sort
2) Ricerca massima ripetuta
entrambi nel worst-case avevano $O(n^2)$
Oggi introduciamo il **Merge-Sort,** si basa sulla tecnica di DIVIDE-ET-IMPERA e vedremo che $\in O(n\log_2 n)$



Una volta che sono ordinati questi due array minori voglio ricombinarli in modo da risolvere il mio problema originario

Il nostro caso base sarà n=1 perché questo insieme sarà già ordinato, se è più grande dividilo e ordinalo.

Dati due array ordinati come faccio a ricostruire un insieme ordinato?

p	T(n1)		q
8	9	15	27

q+1	T(n2)		r
1	3	7	18

L'osservazione è che nessuno degli elementi sarà più piccolo del primo elemento di ognuno dei due array, quindi scorro il puntatore sull'array che ha il minimo tra i due.
Un trucco è aggiungere alla fine dell'array un $+\infty$ così che quando esaurisco uno dei due array passeranno solo gli altri elementi senza dover controllare la dimensione dell'array

A = array di partenza ; q ci serve per partizionare in due parti
Merge (A, p, q, r)

T_{MS} (n) = → O(1) se n = 1
→ O(1) + T_{MS} (n1) + T_{MS} (n2) + $\Theta(n)$ se n > 1
l' ”O(1)” nella seconda parte è il costo della scelta dell'intero q

Potrei scegliere n1 e n2 a piacere ma con n1 = n2 avrò:

$2T(\frac{n}{2})+c_1 n$ → $\in\Theta(n\log n)$

quindi q = (punto di mezzo) = $\lfloor\frac{p+r}{2}\rfloor$

$q=p+\frac{n}{2}$ → $q=p+\lfloor\frac{r-p+1}{2}\rfloor$

Note: Se voglio dividere l'array in 3 partizionare

$q=p+\lfloor\frac{n}{3}\rfloor-1$ ma $q\neq\lfloor\frac{p+r}{3}\rfloor$

quindi la seconda formula vale solo se si divide a metà

Merge-Sort la complessità ottima si ottiene solo se si divide a metà

Codice:

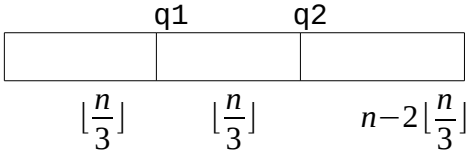
```
MergeSort ( A, p, r ){
     if ( p < r ) {   // analogo a dire n > 1
          q ← ⌊(p+r)/2⌋ ;
          MergeSort ( A, p, q );
          MergeSort ( A, q+1 , r );
          Merge ( A, p, q, r )  //ricombina i 2 array
     }
}

Merge ( A, p, q, r ){
     n1 = q – p + 1;
     n2 = ( r – p + 1 ) - n1;          //oppure r - q
    //L[1 … n1 + 1] ← A[p … q, +∞ ] creiamo un nuovo array che viene
inizializzato con la prima metà
     for i ← 1 to n1 do
          L[i] ← A[p + i – 1];
```

```
      L[n1 + 1] ←  +∞ ;

      //R[1 … n2 + 1] ← A[q + 1 … r, +∞ ]
      for i ← 1 to n2 do
            R[i] ← A[q + i];
      R[n2 + 1] ←  +∞ ;
      i ← 1; j ← 1;   //puntatori ai primi elementi dei array metà
      for k ← p to r do {
            if ( L[i] < R[j] ) {
                  A[k] ← L[i];
                  i++;
            }
            else {
                  A[k] ← R[j];
                  j++;
            }
      } // fine for
} //fine Merge
```

Se usiamo il Merge-Sort dividendo l'array in 3 parti bilanciate devo trovare due valori



Codice:

```
MergeSort ( A, p, r ){
      if ( p < r ) {
            q1 ←   p+⌊(r−p+1)/3⌋−1 ;

            q2  ←   p+2⌊(r−p+1)/3⌋−1
            MergeSort ( A, p, q1 );
            MergeSort ( A, q1 + 1 , q2);
            MergeSort ( A, q2 + 1 , r );
            Merge ( A, p, q1, q2 , r )
      }
}
```

$T(n) = \rightarrow$ $O(1)$ se $n = 1$

$\rightarrow$ $3T(\frac{n}{3})+O(1)+$ quanto vale il Merge se $n > 1$

```
Merge ( A, p, q1, q2 , r ){
      n1 =
      n2 =
      n3 =
 //L[1 … n1 + 1] ← A[p … q1, +∞ ]
 //M[1 … n2 + 1] ← A[q1 + 1 … q2, +∞ ]
 //R[1 … n3 + 1] ← A[q2 + 1 … r, +∞ ]
      i ← 1;     j ← 1; w ← 1;
```

```
      for k ← p to r do {
            if ( L[i] < R[w] ) and ( L[i] < M[j] ) {
                  A[k] ← L[i] ;          i++;
            }
            if ( R[w] < L[i] ) and ( R[w] < M[j] ) {
                  A[k] ← R[w] ;          w++;
            }

            if ( M[j] < R[w] ) and ( M[j] < L[i] ) {
                  A[k] ←M[j] ;           j++;
            }
      }
}
```

T_{MS} = → O(1) se n = 1

→ $3T(\frac{n}{3})+c \cdot n$ se n > 1

semplicemente avremo una c più grande => $\in O(n \log n)$
anche se ha una base del log più grande, cioè un costo più piccolo di prima, si ha una "c" più grande quindi non cambia

$k\,T(\frac{n}{k})+n\,k$ finché k è una costante ok, ma in caso k = n avremmo

$T(n)=n\,T(1)+n^2$ quindi non si ha più convenienza

Osserviamo che un algoritmo che deve contare ad esempio le occorrenze di un evento è molto diverso da quello che per esempio deve stamparle, poiché questo secondo avrà un costo di almeno O(1) per il numero delle occorrenze (T(n) = O(1) #occorrenze)

Esempio 1:
evento inversione
i < j A[i] > A[j]
insieme ordinato crescente #inversioni = 0
insieme ordinato decrescente #inversioni = $\frac{n(n-1)}{2}=O(n^2)$
(i , j) è inversione
L'algoritmo che elenca le inversioni e le <u>conta</u> : |A| = n

```
counter = 0
for i ← 1 to n do
      for j ← i + 1 to n do
            if ( A[i] > A[j] ) {
                  stampa ( i , j );          counter++;
            }
```

complessità: $\sum_{i=1}^{n}(\sum_{j=i+1}^{n} O(1))=\sum_{i=1}^{n} O(n-i-1+1)=O(n^2)$
quindi posso contare le inversioni in $O(n^2)$

Quale è lo spazio delle soluzioni ?

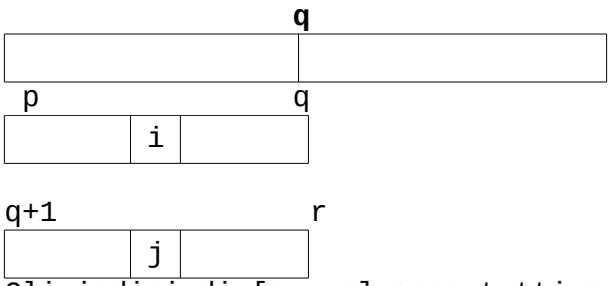
1) elenca inversioni => |S| $\subseteq |\{(i,j): i<j\}|$ => |S| = $\frac{n(n-1)}{2}$

2) conta inversioni => è un numero "c" : $0 \le c \le \frac{n(n-1)}{2}$

1) $\forall$ inversione ➔ *operazione* (in questo caso stampare)
complessità intrinseca del problema (cioè quella che non dipende dall'algoritmo che si adotta)

è pari a (cardinalità di S nel worst-case =>) |S| → $\frac{n(n-1)}{2}$ operazioni

2) per contare il numero di inversioni posso fare meglio, useremo quindi un algoritmo derivato
dal Merge-Sort



Gli indici di [p … q] sono tutti minori di [q+1 … r]

Supponiamo che si verifichi, per i < j, R[j] < L[i] => (i , j) è inversione
L è ordinato => so senza confrontarli che R[j] < L[i+1]
poiché L[i] < L[i+1]< L[i+2] < … < L[q]
noto che quindi c'è anche un altra inversione => (i+1 , j) … ma anche (i+2 , j)
=> esistono altre n1 – i inversioni oltre la prima per un totale di (n1 – i) + 1
n1 = |L|

Codice:

```
containv ( A, p, r ) : integer {      // <= tornerò quindi il numero delle inversioni
     if ( p < r ) {
          q ← ⌊(p+r)/2⌋ ;
          containv ←. containv(A, p, q) + containv(A, q+1, r) + Mergeinv(A, p, q, r);
     }
}

Mergeinv (A, p, q, r) : integer {
     n1 = q – p + 1;       n2 = r – p + 1 – n1;
      //L[1 … n1 + 1] ← A[p … q1, +∞ ]
      //R[1 … n2 + 1] ← A[q1 + 1 … r, +∞ ]
     counter = 0;    i = 1;     j = 1;
     for k ← p to r do {
          if ( L[i] ≤ R[j] ) {
               A[k] ← L[i];    i++;
          }
```

```
            else {
                  A[k] ← R[j];    j++;  counter ← n1 – i + 1 + counter;
            }
      }
      return counter;
}
```

$T(n)=2T(\frac{n}{2})$è $O(n)$ → $\in O(n\log n)$

Esempio 2:
Moltiplicazione di 2 interi (x e y) rappresentati ciascuno su n bit

$x=(x_1,x_0)$ $x_1=\frac{n}{2}$ bit più "pesanti" $x_0=\frac{n}{2}$ bit meno significativi

=> $x_1 2^{(\frac{n}{2})}+x_0 2^0$

$y=(y_1,y_0)$ $y_1 2^{(\frac{n}{2})}+y_0 2^0$

x = 23 (010111) y = 36 (100100)

raggruppando n/2 bit avrei $x=2\cdot 8+7$ dati dalla divisione della rappresentazione binaria

010 = 2 con valore 8 essendo più a sinistra e 111 = 7 con valore 1

y sarà => $y=4\cdot 8+4$ essendo 100 = 4

$$x\,y=(x_1 2^{(\frac{n}{2})}+x_0)(y_1 2^{(\frac{n}{2})}+y_0)$$
$$=x_1 y_1 2^{(\frac{n}{2})}+x_1 y_0 2^{(\frac{n}{2})}+x_0 y_1 2^{(\frac{n}{2})}+x_0 y_0$$

quindi la moltiplicazione x y si può ottenere con 4 moltiplicazioni di n/2 bit

=> $T(n)=4T(\frac{n}{2})+O(1)\in O(n^2)$

Posso cercare di fare meglio di così, quindi considero

$\bar{x}=x_1+x_0$ e $\bar{y}=y_1+y_0$

p = $\bar{x}\,\bar{y}=x_1 y_1+x_1 y_0+x_0 y_1+x_0 y_0$ osservo che $p-x_1 y_1-x_0 y_0=x_1 y_0+x_0 y_1$

=> $x\,y=x_1 y_1 2^n+(x_1 y_0+x_0 y_1)2^{(\frac{n}{2})}+x_0 y_0$ = $x_1 y_1 2^n+(p-x_1 y_1+x_0 y_0)2^{(\frac{n}{2})}+x_0 y_0$

$\bar{x}=\frac{n}{2}$ $\bar{y}=\frac{n}{2}$ => basterà fare solo 3 moltiplicazioni, cioè :

$x_1 y_1$, $x_0 y_0$, $p=\bar{x}\,\bar{y}$

quindi $T(n)=3T(\frac{n}{2})+O(1)\in O(n^{(\log_2 3)})$ che è migliore del precedente

Esempio 3:
Moltiplicazione di due matrici
P = A x B matrici quadrate (n x n)
a_{ik} b_{kj} ← prodotto scalare
numero di operazioni = numero di prodotti scalari
#(prodotti scalari) per P = A x B

p_{ij} = $\sum_{k=1}^{n}$ a_{ik} b_{kj} ← n prodotti scalari

n^2 p_{ij} => n^3

posso dividere una matrice in 4 matrici di $(\frac{n}{2} \text{ x } \frac{n}{2})$

avremo quindi $\begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix} \quad \begin{matrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{matrix}$

P $\begin{matrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{matrix}$

$P_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$
$P_{12} = A_{11} \cdot B_{21} + A_{12} \cdot B_{22}$
$P_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$
$P_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$

ho 8 moltiplicazioni di matrici $(\frac{n}{2} \text{ x } \frac{n}{2})$

posso dire quindi che $T(n) = 8T(\frac{n}{2}) + O(1)$

T(n) = #(prod scalari) (moltiplicazione matrici di (n x n))

$T(\frac{n}{2})$= #(prod scalari) (moltiplicazione matrici di $(\frac{n}{2} \text{ x } \frac{n}{2})$)

$\in O(n^3)$

Possiamo fare di meglio però ed ottenere un $T(n) = 7T(\frac{n}{2}) + O(n^2)$, avremmo quindi un costo di $O(n^{(\log_2 7)}) \leq O(n^3)$, per fare ciò usiamo una tecnica simile a quella di prima

NB: qui sugli appunti viene descritto un accenno e basta, per maggiori dettagli guardare il libro

Non posso fare meno di n^2 poiché devo minimo riempire la matrice P (n x n)
=> mi creo 10 matrici di appoggio
S1, S2, … , S10
del tipo:
S1 = $B_{12} - B_{22}$ S2 = $A_{11} - A_{12}$ S3 = $A_{21} - A_{22}$ …

sono tutte matrici $(\frac{n}{2} \text{ x } \frac{n}{2})$

creo altre 7 matrici di appoggio
C1, C2, … , C7
del tipo:
C5 = S5 x S6 C4 = A_{22} x S4 C2 = S2 x B_{22} C6 = S7 x S8

C(i) è il prodotto di 2 matrici $(\frac{n}{2} \text{ x } \frac{n}{2})$

P_{11} = C5 + C4 – C2 + C6 cioè si possono fare come combinazione lineare dei C(i)
Algoritmo di Strassen

Esempi:
Array ordinato di elementi distinti
$\forall i\, A[i] \in \mathbb{N}$
vogliamo trovare la posizione del primo intero non negativo.
determinare j tale che: $j : A[j] \geq 0$
$j-1 : A[j-1] < 0$
Spazio delle soluzioni: (cioè i possibili valori che possono essere dati in output da questo algoritmo)
{ 1, 2, … , n } $\cup \nexists j \leftarrow \forall i\, A[i] < 0$
|spazio delle soluzioni| = n + 1

metodi:
scansione => è una soluzione ma non sfrutta l'ipotesi che è ordinato => O(n)

```
cerca ( A, p, r ) : integer {
     if ( p < r ) {
          q ← ⌊(p+r)/2⌋ ;
          if ( A[q] ≥ 0 )
               cerca ← cerca ( A, p , q );
          else
               cerca ← cerca ( A, q+1, r )
     }
     else {     //cioè p = r
          if ( A[p]  ≥ 0 )
               return p;
          else
               //non esiste un indice, quindi tutti gli elementi sono < 0
     }
}
```

T(n) = T(n1) + O(1)
con n1 = max { T(q-p+1); T(r-q+1) }

per avere la complessità ottima => $T(n) = T(\frac{n}{2}) + O(1) \in O(\log n)$

Se l'array non è ordinato conviene comunque scorrerlo tutto, proviamo comunque ad usare il metodo del DIVIDE-ET-IMPERA

```
cerca ( A, p, r ) : integer {
     if ( p < r ) {
          q ← ⌊(p+r)/2⌋ ;
          cerca ← cerca ( A, p, q );
          if ( cerca != -1 )
               cerca ← cerca ( A, q+1, r );
          else {
               if ( A[p]  ≥ 0 )
                    return p;
               else
                    return -1; //-1 indica che non esiste un indice
          }
     }
```

} => $T(n) = 2T(\frac{n}{2}) + O(1) \in O(n)$

Se spezzo sempre in " if (p < r) { [p , q] e [q+1 , r] }"
avrò sempre un caso terminale con dimensione = 1.
Bisogna stare attenti al caso terminale poiché nella scelta degli indici c'è il rischio di scrivere un algoritmo che non rientri mai nel caso base e quindi si entri in un loop

Esempio 2:
Trovare numero mancante
A array ordinato crescente A[i] $\in$ [1 … n+1] senza ripetizioni
[Es: |A| = 5 A[1,2,3,4,5] A[i] $\in$ [1 … 6] numero mancante = 6]
spazio delle soluzioni = { 1, 2, … n +1 }
|spazio delle soluzioni| = n+1
metodi:
=> "brute force" => scansione => O(n) ma non sfrutto l'ordinamento
osserviamo che nell'indice i io avrò A[i], se manca un numero avrò invece A[i]+1

codice:
```
cerca ( A, p, r ) : integer {
     if ( p < r ) {
          q ← ⌊(p+r)/2⌋ ;
          if ( A[q] = q )
               cerca ← cerca ( A, q+1, r );
          else
               cerca ← cerca ( A, p, q )
     }
     else {
          if ( A[p] = q+1 )
               return p;
          else
               return //n+1
     }
}
```
$\in O(\log n)$

Quick-Sort
Algoritmo basato sul DIVIDE-ET-IMPERA

p	i	r

A[r] è il perno
Dopo la procedura di partition (di separazione)
Tutti gli elementi { A[p] … A[i - 1] } $\leq$ A[r] , ma non sono in ordine
Tutti gli elementi { A[i] … A[r - 1] }> A[r]
Poi scambio A[r] con A[i] , A[r] → pivot
ottengo che { A[p] … A[i] } $\leq$ A[r] e { A[i + 1] … A[r] } > A[r]

Esempio:

p						r
7	3	2	8	10	11	6

A [r] = 6 → pivot
i = p – 1 confronto A[j] con A[r] con $p \leq j \leq r$
Se A [j] > A[r] non faccio niente, se è più piccolo incremento i e scambio A[j] con A[i]

3	7	2	8	10	11	6

Con i che punta a 3
incremento j, controllo A[j] = 2 < 6 =>

3	2	7	8	10	11	6

E i ora punta a 2. Controllo i confronti finché j = r senza fare più niente essendo gli altri elementi tutti più grandi del pivot. A questo punto scambio i con j = A[r] =>

3	2	6	8	10	11	7

Codice:

```
Partition ( A, p, r) : integer {
      pivot ← A[ r ];
      i ← p – 1;
      for j ← p to r - 1  do {
            if ( A[ j ] ≤ pivot ){
                  i++;
                  //scambia ( A[ i ] , A[ j ] )
            }
      }
      i++;
      //scambia ( A[ r ] , A[ i ] )
      return i ;
}

Quick-sort ( A, p, r ){
      if ( r > p ){
            q ← partition ( A, p , r );// O(n)
            quick-sort ( A, p, q - 1 );// T( q - 1 - p + 1 ) = T( q - p )
            quick-sort ( A, q + 1, r );//T( r – (q + 1) + 1 ) = T( r - q )
      }
}
```

n = r – p + 1
le 2 chiamate ricorsive valgono insieme q – p + r – q = r – p => n – 1
esistono svariati modi di fare la funzione partition, questa scritta da noi ha il vantaggio di non usare memoria aggiuntiva => ordinamento in place

La complessità dipende dalla scelta del pivot
nel caso pessimo pivot = max

```
          T(n)  => O(n)
          /    \
       T(n-1)      O(1)    => O( n – 1 )
         /     \
     T(n-2)     O(1)       => O( n – 2 )
```

se il nuovo pivot è di nuovo il massimo
h = n => $O(n^2)$
caso in cui l'array è già ordinato in senso crescente

Succede anche se l'array è ordinato in senso decrescente, verrà un albero sbilanciato dalla parte opposta a quello precedente.

Altro caso peggiore in cui i pivot si alterna con il massimo e il minimo dell'array ogni volta.
Però non sono tantissimi i casi in cui si abbia il caso pessimo $T(n)=O(n^2)$, fortunatamente questo capita poche volte, in questi casi l'algoritmo non è "quick" però nel caso medio migliora di molto la sua complessità.

Average case (caso medio)
$T(n)=O(n)+T(n_1)+T(n-1-n_1)$
nel codice sopra scritto risulta: $n_1=q-p$ rango del pivot nell'insieme A[p] … A[q]
Ora possiamo considerare il nostro n_1 = rango del pivot nell'insieme A[1] … A[n] meno 1
$1 \le \text{rango pivot} \le n$ => se rango pivot $=1 \Rightarrow 0 \le n_1 \le n-1$ <= se rango pivot = n

Nell'average case si considera che il pivot abbia pari probabilità di avere
rango 1 = Prob(rango = 2) = … = $\frac{1}{n}$
E (T(n)) = $\sum_{i=1}^{n} (\text{prob(pivot abbia rango i)}[T(i-1)+T(n-(i-1)-1)+O(n)])$
ps: la prof ha usato una E sbarrata verticalmente comunque indica la media
[…] ← indica il tempo di esecuzione al variare del rango

= $\frac{1}{n} \sum_{i=1}^{n} [T(i-1)+T(n-(i-1)-1)+O(n)]$
il $\frac{1}{n}$ iniziale indica il fatto che hanno tutti la stessa probabilità
faccio un cambio di variabile
= $\frac{1}{n} \sum_{n_1=0}^{n-1} [T(n_1)+T(n-n_1-1)+O(n)]$

ora devo vedere cosa succede al variare di n_1

	$T(n_1)$	$T(n-n_1-1)$
0	T(0)	T(n-1)
1	T(1)	T(n-2)
2	T(2)	T(n-3)
n-2	T(n-2)	T(1)
n-1	T(n-1)	T(0)

Notiamo che si ripetono sempre 2 volte al variare di n_1
=> = $\frac{1}{n}\sum_{n_1=0}^{n-1}(2T(n_1)+O(n))=\frac{1}{n}\sum_{n_1=0}^{n-1}2T(n_1)+\frac{1}{n}\sum_{n_1=0}^{n-1}O(n)=\frac{1}{n}\sum_{n_1=0}^{n-1}2T(n_1)+\frac{1}{n}O(n)n$
il mio valore medio sarà $\frac{2}{n}\sum_{n_1=0}^{n-1}T(n_1)+O(n)$
per induzione faccio una stima
$T(n) \in O(n\log n-\frac{1}{4}n)$ in cui la seconda parte è il termine di ordine

inferiore utile all'induzione
Sostituiamo:

$$T(n) \leq c(n\log n - \frac{1}{4}n) \leq \frac{2}{n}\sum_{n_1=0}^{n-1} c(n_1 \log n_1 - \frac{1}{4}n_1) + a\,n$$

spezzo la sommatoria =>

$$\leq \frac{2c}{n}[\sum_{n_1=0}^{\frac{n}{2}} n_1 \log n_1 - \frac{n_1}{4}] + \frac{2c}{n}[\sum_{n_1=\frac{n}{2}+1}^{n-1} n_1 \log n_1 - \frac{n_1}{4}] + a\,n$$

$$\leq \frac{2c}{n}\log\frac{n}{2}\sum_{n_1=0}^{\frac{n}{2}} n_1 - \frac{2c}{4n}\frac{\frac{n}{2}(\frac{n}{2}+1)}{2} + \frac{2c}{n}\log n \sum_{\frac{n}{2}+1}^{n-1} n_1 - \frac{2c}{n}\sum_{\frac{n}{2}+1}^{n-1}(\frac{n_1}{4})$$

$$\leq c\,n\log n - \frac{cn}{4} + a\,n - \frac{cn}{4} + c\log n$$ ← questa riga non sono sicuro che è giusta

dopo diversi calcoli risulterà $O(n\log n)$

Complessità nel Worst-Case, nel caso in cui si abbiano gli input che portano ad usare il maggior numero di passi.

Casi visti:
- n^2 ← Insertion-Sort
- n^2 ← Quick-Sort
- $n\log n$ ← Merge-Sort

Average-Case, la media dei tempi di esecuzione dell'algoritmo, per ogni input la sua probabilità → E(T(n)) = $\sum \text{prob(i)}\,T(n,i)$ prob(i) è uniforme

Abbiamo visto 2 casi:
- n^2 ← Insertion-Sort (risulta O(n^2) anche nel worst-case)
- $n\log n$ ← Quick-Sort

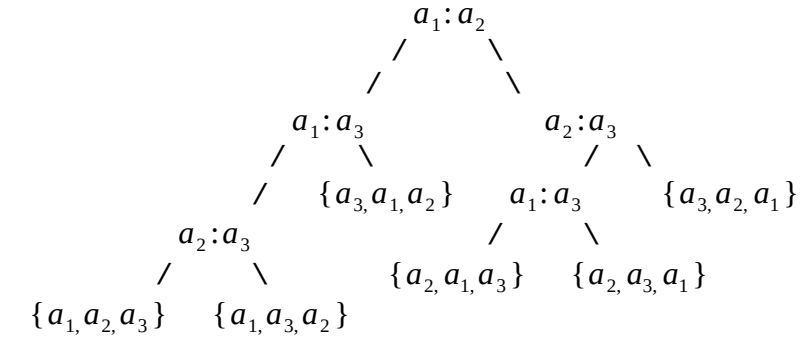
vogliamo vedere la complessità intrinseca del problema, cioè che esula dall'algoritmo usato.

Albero delle decisioni
- è una descrizione di algoritmi basati sui confronti
- ogni nodo rappresenta un confronto

dopo il confronto si hanno due possibilità:

a : b
O
/ \
≤ >

supponiamo di voler descrivere l'algoritmo che ordina { a_1, a_2, a_3 }
con l'albero delle decisioni:



Proviamo con 1, 8, 2:

1 : 8
/ (1 ≤ 8)
1 : 2
/ (1 ≤ 2)
8 : 2
(8 > 2) \
{ 1, 2, 8 }

e 8, 5, 11

8 : 5
\
5 : 11
/
8 : 11
/
{ 5, 8, 11 }

L'esecuzione è un cammino.
Nel worst-case quanti passi faremo ? Esattamente l'altezza dell'albero nel nostro caso h = 3
#foglie = 3! , e devono essere sempre 3!, nel caso di ordinamento di 3 numeri, il caso contrario significa che avrei una sequenza che non saprei ordinare

L'albero con altezza minima sarà quello con complessità minima ma dovrò avere per forza 3! foglie

Infatti provando a scrivere un albero diverso notiamo che con un livello e basta potrei avere al massimo 2 foglie, con 2 livelli al massimo 4, sono quindi obbligato ad avere 3 livelli.
Allora la complessità intrinseca dell'ordinamento di 3 numeri è 3.

Dato un problema noi guardiamo lo spazio delle soluzioni, in particolare la cardinalità di questo (|spazio soluzioni|), possiamo quindi dire che ogni albero deve avere #foglie = |spazio soluzioni|
e avrà altezza = $\log_2 |\text{spazio soluzioni}|$

Quindi:
n → n! Tra tutti gli alberi si deve avere l'albero binario completo per almeno n! Foglie. Con h = $\log_2 n!$
=> #confronti = h = $\log_2 n!$
Quindi non ci può essere nessun algoritmo basato sui confronti che faccia meno di $\log_2 n!$
Studiamo questo dato:

$\frac{2^n}{2} < n! < n^n$ => $n-1 < \log n! < n\log n$

n! = 1 · 2 · … · n ho quindi n/2 termini più grandi di n/2 =>

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

=> $(\frac{n}{2})^{\frac{n}{2}}$ < n! < n^n => $\frac{n}{2}\log\frac{n}{2} \leq \log n! \leq n\log n$

=> a partire da un certo n_0 $\log\frac{n}{2} = \log n - 1 \geq \frac{\log n}{2}$

$\frac{n}{4}\log n \leq \log n! \leq n\log n$ => $\log n! \in \Theta(n\log n)$

$h \geq \log n! \geq n\log n$ ← complessità intrinseca del problema di ordinamento basato su confronti
=> il nostro "leader" è il MERGE-SORT che anche nel caso pessimo è $O(n\log n)$

Un algoritmo è ottimo quando la sua complessità nel worst-case coincide con la complessità intrinseca del problema.
Chiamiamo quindi Ω la complessità intrinseca del problema

Esempio 1:
1) Array A ordinato crescente, se $k \in A$ => ritorna posizione
se $k \notin A$ => ∄
Ci basiamo sull'idea del DIVIDE-ET-IMPERA

=> $T(n) = T(\frac{n}{2}) + O(1) = O(\log n)$ è un algoritmo ottimo ?

=> devo scoprire quale è lo spazio delle soluzioni:
spazio soluzioni = { 1, 2, …, |A|, ∄ } avrò quindi n+1 foglie
=> $\Omega(\log_2(n+1)) = \Omega(\log_2(n))$
=> il nostro algoritmo è già ottimo non possiamo cercare un algoritmo che farà meglio di così

Esempio 2:
Supponiamo ora lo stesso problema ma con un array non ordinato
lo spazio delle soluzioni è lo stesso
=> T(n) = T(n-1) + O(1) = O(n)
=> $\Omega(\log_2(n+1)) = \Omega(\log_2(n))$
=> possiamo migliorare l'algoritmo o dobbiamo fare delle considerazioni Ω
posso ignorare di leggere un input ? NO
poiché l'array non è ordinato
entra in gioco ADVERSARY MODE o ORACLE
Nel nostro caso
Dimensione input/output
il problema ha almeno una dimensione pari all'input non ho nessuna considerazione che non leggendo un dato "l'avversario" mi mostri che era lì

il mio k
=> $\Omega(|A|)$ => $\Omega(n)$
il bound dell'output è “stupido”
un caso in cui si ha output maggiore era per esempio il problema delle inversioni in cui |output| = #inversioni
es: elenca le inversioni, l'evento contabile coincide con la dimensione dell'output

Esempio 3:
Calcolare massimo in un insieme S, non ordinato |S| = n
La soluzione banale è scandire l'insieme, con un costo O(n)
Calcoliamo la complessità intrinseca del problema
Ω = → $\log_2|S|$ dato da |spazio soluzioni|=|{x : $x \in S$ }| = |S|
→ |S| Dimensione dell'input, devo controllarli tutti
→ 1 Dimensione dell'output, devo restituire un solo valore

Devo considerare il maggiore => $\Omega(n)$
=> la soluzione sopra proposta è ottima => $\Theta(n)$

Esempio 4:
Calcolare contemporaneamente massimo e minimo in S, non ordinato
$\Omega(n)$ |S| = n
O(n) = (n – 1) + (n – 2)
n-1 è dato dalla ricerca del massimo poiché devo scorrere tutto l'array
n-2 è dato dalla ricerca del minimo scorrendo tutto l'array meno il massimo
Nel caso in cui faccio i confronti insieme ho 2(n – 1) + 1
Sono entrambi algoritmi ottimi in cui l'unica differenza è data dalla costante c. Posso quindi provare ad abbassarla.
Si può fare in $\frac{3}{2}n$

1	7	3	6	11	18	21	5

Confronto gli elementi a coppie, (1,7) , (3,6) e così via
e pongo in due array i risultati del confronto

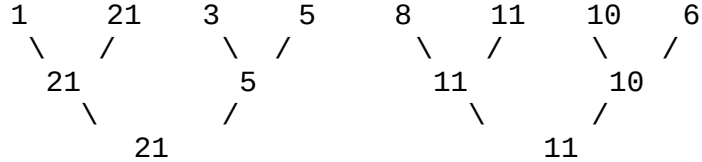
7	6	18	21

1	3	11	5

Calcolo quindi ora il massimo a sinistra e il minimo a destra
avrò quindi costo $(\frac{n}{2}-1)2+\frac{n}{2}$ ← questo ultimo per la divisione

Esempio 5:
Dato S non ordinato, determinare x,y tale che rango(x) = n e rango(y) = n – 1
[rango è la posizione di un numero dell'insieme ordinato, nell'esempio di prima avrei rango(3)=2, rango(18)=7 **]**
quindi “rango(x) = n e rango(y) = n – 1” significa cercare il termine massimo e il precedente al massimo, cioè il penultimo.
$\Omega(n)$
Prima cerco il massimo in n elementi(cioè tutto S), poi cerco il massimo in n-1 elementi => O(n) => l'algoritmo è ottimo

Potrei anche usare un diverso metodo che vedremo con il seguente esempio



avremo quindi un ultimo confronto (21,11) in cui vincerà 21, ed ecco trovato il massimo, il massimo – 21 lo troveremo confrontando il massimo tra gli elementi che hanno perso il confronto con il massimo, cioè (1, 5, 11) e vedremo che è proprio l'11

con questo avrò un costo (n+1)+ log n – 1 che è comunque un O(n)

Esempio 6:

Array A, $A[i] \in \mathbb{R} \quad \forall 1 \le i \le n$

Definiamo sottoarray come una porzione consecutiva di A

A[i … j] è la porzione di A dall'indice i all'indice j

somma S(i,j) = $\sum_{t=i}^{j} A[t]$

Trovare il sottoarray A* tale che la coppia di indici i* j* dia

S* = S(i*,j*)= max{S(i,j)} $\quad \forall (i,j) \in$ coppie di indici

Codice:

```
S* ← 0;
for i ← 1 to n do {
     for j ← i to n do {
          s ← 0;
          for t ← i to j do
               S ← S + A[t];
     }
     S* ← max {S, S*};
}
```

$$T(n)=\sum_{i=1}^{n}(\sum_{j=1}^{n}(\sum_{t=i}^{j}O(1)+O(1)))=\sum_{i=1}^{n}(\sum_{j=1}^{n}(\sum_{t=i}^{j}O(1)))+\sum_{i=1}^{n}(\sum_{j=1}^{n}O(1))$$

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

$$\le O(1)\sum_{i=1}^{n}(\frac{nn}{2})+O(n^2)\le O(n^3)+O(n^2) \rightarrow \in O(n^3)$$

Questo lo definiamo un "Brute force Algorithm", cioè risolve il problema ma non è un algoritmo ottimo

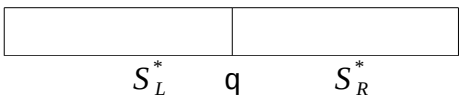
Spazio delle soluzioni = (i,j) : $1 \le i \le j \le n$

$|\text{Spazio soluzioni}| \ge \frac{n^2}{2}$ albero delle decisioni => $\log_2 \frac{n^2}{2}$

Dimensione input = n Dimensione output = 1 => $\Omega(n)$

Proviamo ad usare il metodo DIVIDE-ET-IMPERA

$2T(\frac{n}{2})+O(n)$ => $O(n \log n)$



Divido cioè l'array e cerco il sottoarray nei mezzi array creati
Se la soluzione fosse in mezzo ?
Consideriamo un i* nell'array sinistro, un j* in quello destro
allora a noi interessano le coppie che iniziano in i* e finiscono in q,
cioè che potrebbero attaccarsi nell'altra metà dell'array.
Scandisco quindi da q verso sinistra cercando il sottoarray A[i* … q]
maggiore, faccio la stessa cosa dall'altra parte da q+1 a j* e avrò quindi
il sottoarray A[i* … j*] che sarà in concorrenza che la ricerca del
sottoarray maggiore.

Codice:
FMS = Find-Max-SubArray
FMCS = Find-Max-Crossing-SubArray

```
FMS (A, p, r) : (indice, fine, S) {
     if( p = r)
          return (p, r, A[p]);
     else {
          q ← ⌊(p+r)/2⌋ ;
            i_sx, f_sx, V_sx ← FMS(A, p, q);
            i_dx, f_dx, V_dx ← FMS(A, q+1, r);
            i_cr, f_cr, V_cr ← FMCS (A, p, q, r);
            ī, f̄, val̄ ← max( V_sx, V_dx, V_cr );    // ī, f̄ ← inizio e fine di
 val̄
     }
}

FMCS (A, p, q, r) {
     left-sum ← −∞ ; sum ← 0;
     for i ← q down p do {
          sum ← sum + A[i];
          if ( sum > left-sum){
               left-sum ← sum;
               max-left ← i;
          }
     }
     right-sum ← −∞ ; sum ← 0;
     for j ← q+1 to r do {
          sum ← sum + A[j];
          if (sum > right-sum){
               right-sum ← sum;
               max-right ← j;
          }
     }
     return (max-left, max-right, left-sum + right-sum);
}
```

Max-Sum sottovettore

$T(n)=2T(\frac{n}{2})+O(n)=O(n\log n)$ algoritmo DIVIDE-ET-IMPERA

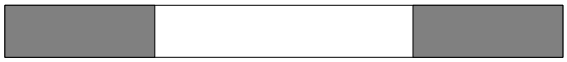
$\Omega(n)$ dovuto alla dimensione dei dati

Quale dei due va aggiornato ?
Cioè vorremmo cercare O(n), avendo $\Omega(n)$ per avere un algoritmo ottimo

=> $T(n)=2T(\frac{n}{2})+O(1)$ cioè un algoritmo DIVIDE-ET-IMPERA che nella ripartizione dei risultati abbia un costo costante

Per fare ciò dovrei "portarmi dietro" nella ricorsione sia la somma del sottovettore che parte da p, sia quella che arriva in q

p q



poiché non so se quando verrà richiamata la funzione, questo sottovettore sarà ricollegato a destra o a sinistra.

Es:
13,-3,-25,20,-3,-16,-23,18,20,-7,-5,-22,15,-4,7
Considerando due divisioni in sottovettori avrei
13,-3,-25,20 | -3,-16,-23,18 | 20,-7,-5,-22,15,-4,7
Quindi nel secondo ad esempio dovrei tener conto sia di -3 che di 18 poiché sono i sottovettori con somma massima che partono dagli estremi

=> Per ogni porzione devo portarmi avanti
SL sotto vettore di lunghezza max sinistra
SR sotto vettore di lunghezza max destra
SELL somma massima partendo dall'estremo sinistro verso destra
SELR somma massima partendo da q verso sinistra
SERL somma massima partendo da q verso destra
SERR somma massima partendo dall'estremo destro verso sinistra

p q r

SELL SELR SERL SERR
Più all'interno SL e SR
Ognuno di questi sono segnati da "i" e "f" che sono rispettivamente l'indice di inizio e fine

SL i=f=?
SR i=f=?
SELL i=p f=?
SELR i=? f=q
SERL f=q i=?
SERR i=? f=r

SL o SR potrebbero coincidere con le sequenze partite dagli estremi

Se avessimo questo informazioni potremmo trovare la somma-massima in tempo costante ?

=> alla fine la soluzione potrà essere: max { SL, SR, SELR+SERL }
Ponendo nel caso finale:

SEL SER
avremo che: SEL=SELL se $i_{SELR} \neq p$

Troviamo quindi la soluzione in O(n) poiché abbiamo un costo constante per trovare il massimo

Codice:
S = valore somma massima centrale, i e f di s i suoi indici di inizio e fine

```
OttDIV(A,p,r,S, i_s , f_s ,SEL, i_SEL , f_SEL ,SER, i_SER , f_SER ){
     if( p < r ){
          q ← ⌊(p+r)/2⌋ ;
          OttDIV(A,p,q,SL, i_SL , f_SL ,SELL, i_SELL , f_SELL ,SELR, i_SELR , f_SELR
);
          OttDIV(A,q+1,r,SR, i_SR , f_SR ,SERL, i_SERL , f_SERL ,SERR, i_SERR ,
 f_SERR );

          if(max{SELR+SERL,SR,SL}=SELR+SERL){
               S ← SELR+ SERL;
                 i_s ← i_SELR ; f_s ← f_SERL ;
          }
          if(max{SELR+SERL,SR,SL}=SL){
               S ← SL;
                 i_s ← i_SL ; f_s ← f_SL ;
          } else {
               S ← SR;  i_s ← i_SR ; f_s ← f_SL ;
          }
          if( i_SELR == p && SELR + SERL > SELL){
               SEL ← SELR + SERL;
                 i_SEL ← p; f_SEL ← f_SERL ;
          } else {
               SEL ← SELL;  i_SEL ← i_SELL ; f_SEL ← f_SELL ;
          }
          if( f_SERL == r && SELR + SERL > SERR){
               SER ← SELR + SERL;
                 i_SER ← i_SERR ; f_SER ← r;
          } else {
               SER ← SERR;
                 i_SER ← i_SERR ; f_SER ← f_SERR ;
          }
     } //fine if (p < r)
     else {     // if (p = r), cioè ho un solo elemento
          S ← A[p];          i_s ← f_s ← p;
          SEL ← A[p];        i_SEL ← f_SEL ← p;
          SER ← A[p];        i_SER ← f_SER ← r;      //r = p
     }
}
```

Il codice sopra indicato non è del tutto corretto poiché in certi casi non prende l'effettivo range a soma maggiore. La correzione proposta per determinare i corretti SEL e SER è l'aggiunta dei seguenti casi:

SEL = max { SELL, SELL + SERL(se f_{SELL} =q), SELR + SERL(se i_{SELR} =p), SELL + SERR(se f_{SELL} =q && i_{SERR} =q+1) }

SER = max { SERR, SELR + SERL(se f_{SERL} =r), SELR + SERR(se f_{SERR} =q+1), SELL + SERR (se f_{SELL} =q && i_{SERR} = q+1)}

Esempio:
13,-3,-25,20,-3,-16,-23,18,20,-7,-5,-22,15,-4,7
Dividendo ricorsivamente in sottovettori arriverò ad avere elementi singoli dell'array. Essendo il caso base avremo evidentemente SEL=SER e S=elemento stesso, esempio con elemento 13:

```
     13     |   -3
  13    13   -3   -3
    \  /       \  /
     13          -3
       \       /
          13
```

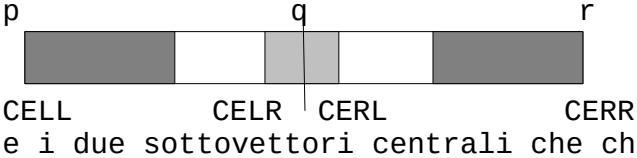
adesso avremo però SEL=13, SER=10, si continua così riunendo i vari sottovettori fino a trovare il massimo.
C'è una versione più facile di questo algoritmo che ci mette O(n), questo è un algoritmo iterativo lineare, che deriva dalla seguente osservazione:
Dato A[1 … j+1] possiamo dire che:
A[j+1] può appartenere o meno alla soluzione. Quindi:
- se A[j+1] $\in$ Sol => Sol(A[i … j]) + A[j+1]
cioè dice che la soluzione è la soluzione precedente, fino a j, a partire da un certo i, più l'ultimo elemento j+1, questo solo se A[j+1]>0
- se A[j+1] $\notin$ Sol => Sol(A[1 … j])
cioè se l'ultimo elemento non appartiene alla soluzione allora possiamo dire che la soluzione esiste nella parte di array fino a j

Un altro esempio di un algoritmo simile, “ma più facile”, è quello di, dato un insieme, trovare il sottovettore crescente di lunghezza massima.
Questo problema ha $\Omega(n)$
Esempio:

1 2 3	1 10 11 13 18	5
3	5	1

Una soluzione in O(n) iterativa è quella di scorrere finché non si trova un valore minore a quello precedente, in questo caso confronto il massimo prima trovato con quello attuale e mi salvo il nuovo massimo

Si può fare anche con DIVIDE-ET-IMPERA
$T(n)=2T(\frac{n}{2})+O(1)=O(n)$ ma come nel problema precedente bisognerà portarsi dietro:



e i due sottovettori centrali che chiameremo Csx e Cdx

Consideriamo ora il problema della ricerca binaria su una lista ordinata
Quanto costa?

$$T(n)=T(\frac{n}{2})+O(n)=O(n)$$

adesso determinare il valore q non costa più O(1) poiché la lista non è ad accesso diretto.
Allora un algoritmo di ordinamento di una lista richiede minimo $\Omega(n\log n)$ dato dall'albero delle decisioni.

Consideriamo di avere da ordinare un vettore di n elementi
considerando che: (min) $m\leq A[i]\leq M$ (max) $\forall i\in[1\dots n],\quad A[i]\in\mathbb{N}$

esempio: $A[i]\in[2\dots4]$ A = 4 3 3 2 3 3 4
Come prima cosa, con una scansione conto progressivamente le occorrenze dei valori:

$C_2=$0 1
$C_3=$0 ~~1~~ ~~2~~ ~~3~~ 4
$C_3=$0 ~~1~~ 2

Lo spazio è quindi: n + (M – m + 1)
Ora se voglio ordinare sfrutto le informazioni che mi sono portato dietro, e riesco a farlo in O(n + (M - m + 1) + n)
(il secondo n è il tempo che mi serve a riscrivere)
pongo k = M – m + 1 , il rango in cui mi muovo
=> diventa O(n + k)

L'Algoritmo del **Counting-Sort** è una versione più complessa di quello appena visto, esso preserva l'ordine relativo delle chiavi uguali
nell'esempio di prima quindi: $4_1\ 3_1\ 3_2\ 2\ 3_3\ 3_4\ 4_2$ → $2\ 3_1\ 3_2\ 3_3\ 3_4\ 4_1\ 4_2$
abbiamo quindi bisogno di conoscere il rango (k)

```
Counting-Sort (A, p, r, k) {
      C: array locale
      for i ← 1 to k
            C[i] ← 0;
      for j ← p to r
            C[ A[j] ]++;
      for i ← 2 to k        //costo O(k)
            C[i] ← C[i] + C[i-1];      //Prefix – Sum
      for j ← r down p {
            B[ C[ A[j] ] ] ← A[j];
            C[ A[j] ]--;
      }
}
```

Nell'esempio di prima abbiamo che la posizione dell'ultimo 4 è 7 = 1 + 4 + 2
Quindi trovo la posizione con la somma di chi lo precede
Posizione ultimo 3 è 5 = 1 + 4
Posizione ultimo 2 è 1
=> C' = [1, 5, 7]

C[i-1] = $\sum_{j=1}^{i-1}$ occ(j) occ(j) = occorrenze di j

C[i] = occ(i) => C[i] + C[i-1] = $\sum_{j=1}^{i} \text{occ(j)}$

[7 | 8 | 9 | 3 | 2 | 1] => C [7 | 15 | 24 | 27 | 29 | 30]

Es:
A [4 | 3 | 2 | 3 | 1 | 1 | 6 | 5]

=> C [2 | 1 | 2 | 1 | 1 | 1]
C[1] = 2 indica che il valore 1 nell'array A è stato trovato 2 volte
Dopo il Prefix-Sum C diverrà = [2 | 3 | 5 | 6 | 7 | 8]
quindi C[2]=3 indica che l'ultima occorrenza di 3 nell'array A dovrà trovarsi in posizione 3

inizio a scandire da indice 7 (l'ultimo dell'array A)
Poichè trovo il valore 5 in A e il valore 7 in C
=> pongo 5 in B[7] => C diventerà [2 | 3 | 5 | 6 | 6 | 8]
Scorro e vedo 6, B[8]=6 => C = [2 | 3 | 5 | 6 | 6 | 7]
Scorro e vedo 1, B[2]=1 => C = [1 | 3 | 5 | 6 | 6 | 7]
. . .
fino ad arrivare a:
B [1 | 1 | 2 | 3 | 3 | 4 | 5 | 6]
C [0 | 2 | 3 | 5 | 6 | 7]

Facciamo un altro esempio ma questa volta con un elemento mancante:
M = 3 , m = 1 e manco 2
A [1 | 3 | 1]
C [0 | 0 | 0] → creo l'array C e lo azzero
C [2 | 0 | 1] → conto le occorrenze
C [2 | 2 | 3] → dopo Prefix-Sum
e infine avrò B [1 | 1 | 3] e C [1 | 2 | 2]
Quindi mi posso accorgere se ho elementi mancanti se in C dopo il prefix-sum ho due numeri consecutivi uguali.
Il counting-sort non è un algoritmo in-place poiché uso 2 array aggiuntivi.
Il 3° for di solito viene fatto da 2 a k, in caso però dovrebbe essere:
for i ← (m + 1) to M

Viene definito algoritmo stable (stabile) se le chiavi mantengono lo stesso ordine relativo prima e dopo l'ordinamento.

Ora vedremo a cosa possono servire gli algoritmi stable

Radix – Sort

numeri rappresentati su 'd' cifre

es: con d = 3, per numeri in base 10, quindi ciascuna cifra sta in $0 \leq \ \leq 9$

10^2	10^1	10^0
3	2	7
9	1	3
3	7	6
3	5	1
4	3	2

sistema posizionale pesato

$327 = 3 * 10^2 + 2 * 10^1 + 7 * 10^0$

=> i numeri sono compresi tra $0 \leq \ \leq B^d - 1$ con B = base

dato dal fatto che si avrà

$(B-1)B^{d-1} + \ \ + (B-1)B^0$

$\leq (B-1)\sum_{j=0}^{d-1} B^j = (B-1)\frac{B^d-1}{B-1} = B^d - 1$

Se ordinassi i numeri sopra scritti con counting-sort avrei O(n + k)

con $0 \leq k \leq B^{d+1} - 1$ quindi non so a priori chi "governa" tra n e k

dipenderà dalla grandezza di n e dal range

Possiamo fare di meglio ?

Consideriamo la cifra a destra l'handle e i restanti numeri come dei satelliti di esso. Allora facciamo counting-sort sulla 3a colonna

3	5	1	poi lo facciamo	9	1	3
4	3	2	sulla 2a colonna	3	2	7
9	1	3		4	3	2
3	7	6		3	5	1
3	2	7		3	7	6

infine lo facciamo sulla 3a colonna, notiamo che così facendo abbiamo mantenuto l'ordine relativo

3	2	7	
3	5	1	e infine l'insieme risulta ordinato
3	7	6	
4	3	2	
9	1	3	

L'ordine della colonna da quale si parte a ordinare è importante, infatti notiamo nel seguente esempio come partendo dalla 1a colonna non ordina l'insieme

3	9	1		3	9	1		4	3	1		4	3	1
3	7	6	=>	3	7	6	=>	3	7	6	=>	3	9	1
4	3	1		4	3	1		3	9	1		3	7	6

E' importante quindi partire sempre dalla cifra meno significativa

i	i-1	0	
a_i	s_i	s_0	
b_i	t_i	t_0	$s_i \leq t_i$

se $a_i<b_i$ $(a_i,s_i)<(b_i,t_i)$
se $a_i>b_i$ $(a_i,s_i)>(b_i,t_i)$
se $a_i=b_i$ domina l'ordine tra s_i e t_i
=> il costo del radix-sort diventa $T_{R-S}=d(n+B)$, con B = base

Dati n interi, e valida $0\leq A[i]\leq n^2=M$
per ordinare possiamo dire che:
1) Algoritmi basati su confronti abbiamo alla meglio $O(n\log n)$ (con Merge-Sort)
2) Counting-Sort lo possiamo usare perché il rango è fisso avendo un $O(n+K)$ ma $k\in n^2$ => $O(n^2)$, quindi non è conveniente
3) Radix-Sort dobbiamo scrivere quante cifre ci servono e di conseguenza sapere quanto sia la base.

Cerchiamo ora la relazione che c'è tra il massimo valore rappresentabile(M) e il numero di cifre:

$M=B^d-1$ #cifre = d

cerco quindi $d=f(M,B)$
ricavo da M, d => $d=\log_B(M+1)$ => $d=\lceil\log_B(M+1)\rceil$

es: per rappresentare 7 in base 2 ci occorrono 3 cifre, infatti $d=\log_2(7+1)=\log_2 8=3$

$d=\lceil\log_B(M+1)\rceil\in\Theta(\log_B M)$ → $\Theta(\frac{\log_2 M}{\log_2 B})$
notiamo quindi che più la base è grande più diminuisce il numero

$d=\Theta(\log_B M)$ facciamo ora counting-sort con un costo di: $\log_B M(n+B)$
da qui ci si divide in due casi:
→ se B = 2 => $\log_2 M(n)$ → $O(2n\log n)$ → $O(n\log n)$
→ se $B=2^r$ => $\frac{\log_2 M}{\log_2 2^r}=\frac{\log_2 M}{r}(n+2^r)$ → $\frac{2\log_2 n}{r}(n+2^r)$
con r = $\log_2 n$ => $\frac{2\log_2 n}{\log_2 n}(n+n)=O(n)$
visto che ho $\Omega(n)$ => è un algoritmo ottimo

=> Radix-Sort con $B=2^r$ e r = $\log_2 n$ ho l'algoritmo ottimo (O(n)) da usare
Es:
n = 4 => $M=n^2=16$
7 0111
13 1101
15 1111 r = $\log_2 n$ => $r=\log_2 4=2$
1 0001

con counting-sort sulla prima colonna

```
11 01                              00 01
00 01      counting-sort    =>     01 11
01 11      2° colonna              11 01
11 11                              11 11
```

=> con n interi $0 \le A[i] \le n^c = M \quad c \ge 0$

qualsiasi c scelga avrò $\frac{c \log n}{r}(n+2^r)$ avendo quindi O(n)

se avevamo $0 \le A[i] \le n^{\log_2 n} \quad d = \log_2 n^{\log_2 n} = (\log_2 n)^2$ se $B = 2$

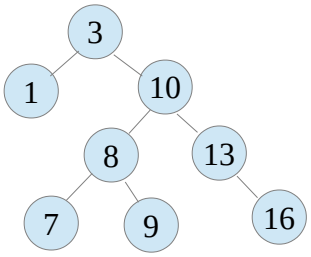
$(\log_2 n)^2 (n+2^r)$ ➔ $r = \log n$ ➔ $O(n \log n)$

Alberi binari di ricerca

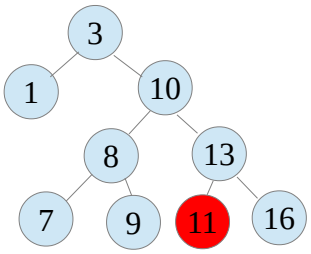
E' una struttura dinamica, essendo un albero binario ogni nodo ha al più 2 figli

Tutte le chiavi che sono memorizzate nel sottoalbero sinistro sono minori o uguali della chiave della radice, quelle del sottoalbero destro saranno maggiori

Es:



proviamo ad inserire la chiave 11, creo un nuovo nodo, confronto con la radice,
11 > 3 => vado nel sottoalbero destro, 11 > 10 => vado ancora a destra, 11 < 13 allora vado a sinistra, è vuoto => mi “attacco”



che relazione c'è tra l'altezza e il numero dei nodi ?

Sappiamo che ho n nodi, quindi avrò $\lceil \frac{n}{2} \rceil$ foglie ed avrò $\text{h minima} \ge \log_2 \lceil \frac{n}{2} \rceil$

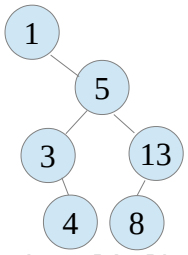
e l' h massima = n

L'albero binario di ricerca può servire ad ordinare un insieme, se faccio una visita in-order avrei infatti tutte le chiavi in senso crescente

Odinamento by means of binary search tree
- se l'albero è già dato devo fare una visita in-order => O(n)
- se invece si ha l'insieme posso prima costruire l'albero e poi visitarlo

quindi dato l'insieme A avrei una funzione costruisci(T, A) e poi una visita-in-order(T)

Es: A = { 1 ,5, 13, 8, 3, 4 }



visita in order mi darebbe quindi:
1 ,3, 4, 5, 8, 13

E' possibile costruire l'albero in O(n) ?
se fosse possibile riusciremmo ad ordinare (con costruzione più visita) in O(n) senza dare nessuna indicazione sulle chiavi

Però noi sappiamo che ogni algoritmo basato sul confronto ha un costo minimo di $n \log n$ => non è possibile costruire l'albero in O(n)
il costo vero è almeno $O(n \log n)$

Quanto costa costruire un albero binario di ricerca?

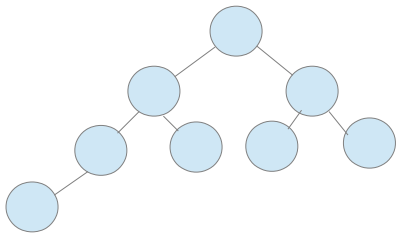
$\sum_{i=1}^{n} i \rightarrow O(n^2)$ Nel worst-case cioè un albero che si riduce ad un cammino

Quindi possiamo affermare i seguenti punti:
- la forma degli alberi binari di ricerca è qualsiasi
- la proprietà delle chiavi non si può verificare localmente
- la costruzione costa $\Omega(n \log n)$

Binary Heap
E' un albero binario quasi completo, completo fino al penultimo livello, e nell'ultimo le foglie sono addossate a sinistra

Es: heap con 4 livelli

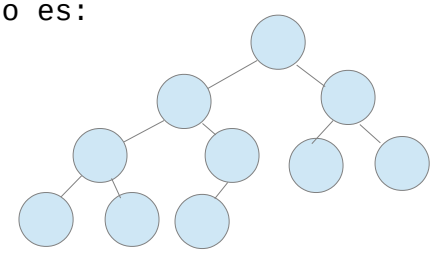
livello: 0
1
2
3



#nodi = 8

questo è l'heap più piccolo che si può avere con 4 livelli

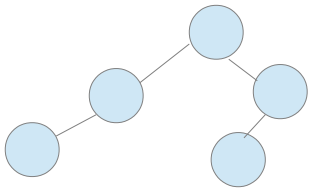
altro es:



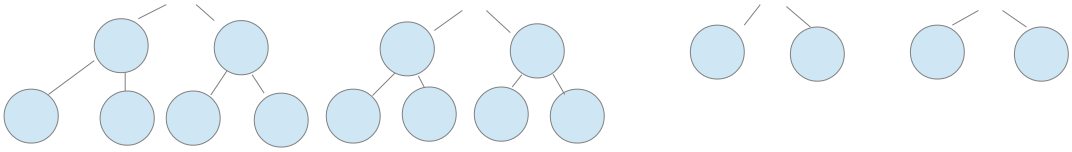
#nodi = 10

se ho h livelli (da 0 a h-1) fino al livello h-2 è un albero binario completo, nel livello h-1 le foglie sono a sinistra, può ovviamente essere anche tutto pieno.

Es: il seguente albero non è uno heap, poiché nell'ultimo livello le foglie non sono addossate a sinistra

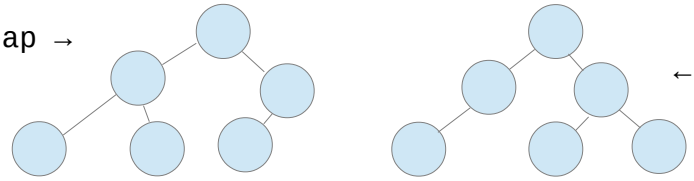


Se prendo #nodi = 21 avrò obbligatoriamente i primi 4 livelli completi e l'ultimo così composto:



Es: #nodi = 6

heap →

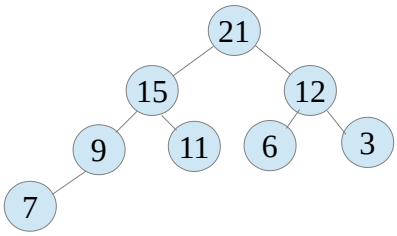


← non heap

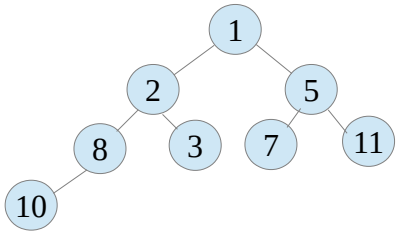
Dato il numero dei nodi la forma dello heap è fissata (ciò non succede per esempio con gli alberi binari di ricerca)

Proprietà:
- forma ben definita
- uno heap può essere max-heap (*1) o un min-heap (*2)

*1) ogni sottoalbero ha come chiave della radice il massimo delle chiavi di esso
Es:



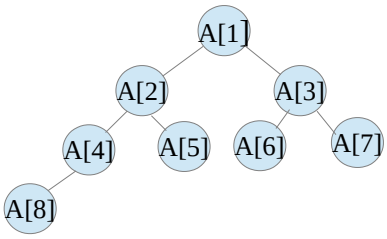
*2) è l'opposto di max-heap
Es:



Questa struttura dati si studia sotto due aspetti:
- Ci permette di fare un algoritmo di ordinamento sul posto, in cui quello ottimo ha costo $O(n \log n)$
- Essa facilita alcune operazioni, con la “coda di priorità” permette di calcolare il massimo e il minimo con un costo inferiore

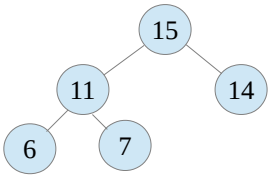
Questa forma dello heap ci permette di rappresentare l'albero in modo implicito in un array

A[1] A[2] A[3] …
in cui A[1] sarà la radice e A[2] e A[3] i suoi figli
Es:




```
        i
      /   \
  2·i       2·i + 1
```

Es: max-heap

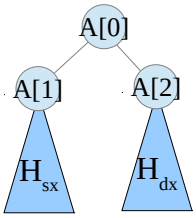


A [15 | 11 | 14 | 6 | 7]

left(i) = 2·i
right(i) = 2·i + 1
parent(i) = $\lfloor \frac{i}{2} \rfloor$

Max-Heap	Min-Heap
MAX(H) = A[0]	MIN(H) = A[0]

Come si verifica dato un albero che sia uno heap?

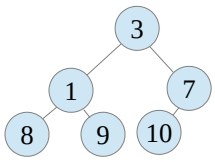


Determinato che il sottoalbero destro e sinistro sono max-heap, basterà controllare che A[0] ≥ max{A[1], A[2]}

di conseguenza A[0] = MAX(A)

A differenza di un albero binario di ricerca si può “lavorare” localmente
Es: se l'albero precedente fosse un albero di ricerca dovrei controllare che
$\min(H_{dx}) \leq \text{root.(key)} \leq \max(H_{sx})$, mentre come visto prima per uno heap basta controllare le chiavi stesse dei due figli e non tutto il sottoalbero

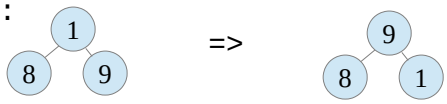
Es: A [3 | 1 | 7 | 8 | 9 | 10] è un max-heap?
Dal punto di vista della forma è uno heap poiché avrei il seguente albero:



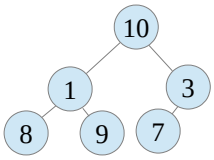
le foglie sono heap poiché sono massimi locali, e quindi superano il test sopra descritto. Però $1 \geq max(8,9)$ è falso => non è un max-heap

Possiamo però risistemare l'array per far si che diventi uno heap
basta prendere il massimo tra i due figli e “farlo saltare” scambiandolo con la radice.

Es:

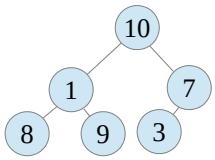


Nel nostro caso, partendo dall'ultima foglia, finita la salita avremo:



però attenzione: 3 – 7 non è più un max-heap, allora devo ricontrollare:

=>



NB: le foglie si trovano sempre tra $\lfloor \frac{n}{2} \rfloor + 1, \ldots, n$, su queste non bisogna far niente.

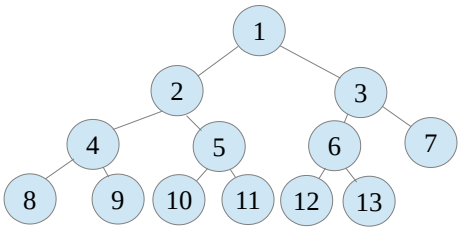
Codice:

```
Build-max-heap (Array A, int n) {
      for i ← ⌊n/2⌋ down to 1
            maxheapify(A, i, n);
}
```

Chiamata la seguente funzione so già che i sottoalberi di 2·i e 2·i+1 sono già max-heap, l'unico nodo che sta violando la proprietà dello heap è i

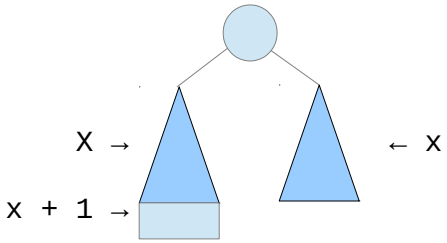
```
maxheapify ( A, i, n) {
      largest ← A[i]; t ← i;
      if ( 2·i≤n ) && (A[i] < A[2·i]){
            t ← 2·i;
            largest ← A[2·i];
      }
      if ( 2·i+1≤n ) && (A[2·i + 1] > largest) {
            t ← 2·i + 1;
            largest ← A[2·i + 1]; //questa riga è inutile ai fini del
problema
      }
      if ( i≠t ) {
            scambia(A[t], A[i]);
            maxheapify(A, t, n);
      }
}
```

Es: il primo nodo non foglia è il padre di 13



quindi parto dalla posizione 6 e chiamo maxheapify, poi decrementerò 6 fino ad 1, cioè la radice.
Studiamo la complessità:
maxheapify ha complessità massima se i = 1, il costo è O(1) + la chiamata ricorsiva, in cui nel caso pessimo va nell'albero sinistro (poiché l'albero più pesante è l'albero sinistro)

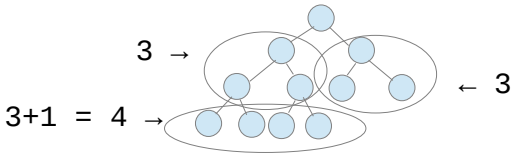
Se abbiamo un albero completo i suoi 2 sottoalberi hanno $\frac{n-1}{2}$ nodi, in uno heap può capitare nel caso pessimo che l'albero sinistro sia un albero completo cioè ha un livello in più pieno.



$n = 1 + x + (x + x + 1) = 3x + 2$ ➔ $x \simeq \frac{n}{3}$

=> $T(n) \leq O(1) + T(\frac{2n}{3})$ che è una maggiorazione di $2(\frac{n-2}{3}) + 1 = \frac{2n}{3} - \frac{1}{3}$

Es:



“Questo albero è un po' sbilanciato, come i miei alberi di natale” cit. Pinotti

=> costo = $\frac{2^i}{3^i} n = 1$ ➔ $n = \frac{3^i}{2^i}$ ➔ $i = \log_{\frac{3}{2}} n = \frac{\log_2 n}{\log_2 \frac{3}{2}}$

=> $T(n) \leq O(1) + T(\frac{2n}{3}) = O(\log n)$

Costo Build-max-heap sarebbe $\frac{n}{2} \log n$ ➔ $\in O(n \log n)$
possiamo fare un analisi più precisa, maxheapify lavora sull'altezza dell'albero e , nel worst-case, al crescere pago di più

Es: albero completo con #nodi = $2^5 - 1$ e h = 5
l'ultimo livello paga: $2^3 \cdot 0$
il 3° livello paga: $\frac{2^3}{2} \cdot 1$
il 2° livello paga: $\frac{2^3}{4} \cdot 2$
il 1° livello paga: $\frac{2^3}{8} \cdot 3$

generalizzando abbiamo $\frac{n}{2} \cdot \frac{1}{2^i} \cdot i$

il 2 elevato alla i viene messo poiché ad ogni salita di livello divido per 2 poiché il numero dei nodi diminuisce

Allora avrei $\sum_{i=1}^{h} (\frac{n}{2^{i+1}} i) = \frac{n}{2} \sum_{i=1}^{h} \frac{1}{2^i}$

poiché, considerando x < 1, abbiamo che:

$\sum_{k=1} k\,x^k = x \sum_{k=1} \frac{d}{dx}(x^k)$ **[** $\frac{d}{dx}(x^k) = k\,x^{k-1}$ **]**

$= x \frac{d}{dx}(\frac{1}{1-x}) = \frac{x}{(1-x)^2}$ non ci interessa a cosa va la sommatoria perché comunque converge a $\frac{1}{(1-x)}$

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

=> la somma di tutti quei lavori è $O(n)$ e non $O(n \log n)$, poiché si hanno tanti nodi in cui si paga poco, più cresco più pago, ma più cresco meno nodi trovo, quindi posso costruire in $O(n)$

Vogliamo ora ordinare in senso crescente
Vorremmo quindi arrivare a un array ordinato in senso crescente, partendo dal fatto che noi sappiamo che il massimo è nella prima posizione

```
//Es: [ 18 | 8 | 16 | 5 | 2 | 14 | 7 | 4 | 3 | 1 ]

Heap-Sort (A , n){
      build-max-heap (A, n); //O(n)
      for i ← n down to 2 {
            scambia (A[1], i);    //mette il massimo nell'ultima posizione
                  // [ 1 | . . . | 18 ]
      /*
      ora ci restringiamo ad uno heap più piccolo, cioè togliamo l'ultimo
            elemento (che non è altro che l'ultima foglia), che nell'esempio
è 18.
      Ora dobbiamo considerare che l'unica cosa fuori posto potrà essere
            solo la radice
      */
            maxheapify(A, 1, i – 1);
            // l'array ora sarà: [ 16 | 8 | 14 | 5 | 2 | 1 | 7 | 4 | 3 |
18 ]

            // quando ripartirà il for dopo lo scambia avrò:
            //  [ 3 | 8 | 14 | 5 | 2 | 1 | 7 | 4 | 16 | 18 ]
      }
}
```

Al 3° passaggio avrò: [14 | 8 | 7 | 5 | 2 | 1 | 3 | 4 | 16 18]
ora è 14 l'elemento massimo tra 1 e n-2
4° : [4 | 8 | 7 | 5 | 2 | 1 | 3 | 14 16 18]
5°: [3 | 5 | 7 | 4 | 2 | 1 | 8 14 16 18]
il for finirà quando avrò un solo nodo da sistemare

maxheapify avrebbe un costo di $\sum_{i=2}^{h-1} \log_{\frac{3}{2}} i$

=> $T(n) = O(n) + \sum_{i=2}^{h-1} \log_{\frac{3}{2}} i \leq O(n) + \sum_{i=2}^{h-1} \log_{\frac{3}{2}} n \leq O(n) + O(n \log n) \leq O(n \log n)$

è il classico algoritmo in cui cerco il massimo, quindi dovrebbe costare n^2 ma con il fatto che è uno heap la ricerca del massimo costa solo $n \log n$

$T(n) \in \Theta(n \log n)$ poichè ci sono almeno $\frac{n}{2}$ termini che costano $\log n$

$c_1 n \log n \leq \log n + \log(n-1) + \log(n-2) + \ldots + \log 1 \leq c_2 n \log n$

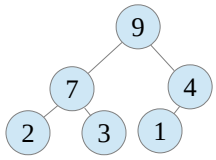
è un algoritmo in-place ma non stable

Lo heap è un interessante struttura dati poiché ho un insieme parzialmente ordinato che mi permette di trovare il massimo ad esempio in O(1) a nella stessa non varrebbe la stessa cosa per il minimo.
Ma non esisterà mai una struttura dati ottima per tutte le operazioni.

Viene definito coda di priorità poiché nel caso di uno heap binario è come se fosse una coda che serve per primo quello con chiave più alta

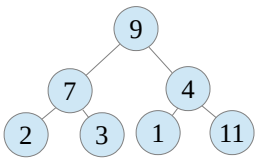
Altre operazioni sullo heap:
- Insertion-key $O(\log_2 n)$ → proporzionale all'altezza dell'albero

Es: (#nodi) n = 6

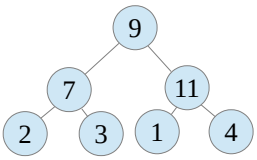


chiave da aggiungere = 11

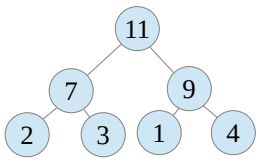
aggiungendo un nuovo nodo il numero di nodi diverrà 7 (n' = 7), come sappiamo sullo heap la forma è fissata rispetto al numero di questi.
Il nuovo nodo andrà quindi messo come figlio destro di 4, però perderà la proprietà dello heap, quindi devo riconfrontare per risistemare l'albero



=>



=>



Codice: su max-heap

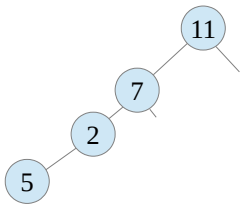
```
Insertion-key ( A, n, k) {
      n ← n + 1;
      A[n] ← k; i ← n;
      while ( A[⌊i/2⌋] < A[i]  &&  ⌊i/2⌋≥1 ){
            scambia(A[i], A[⌊i/2⌋] );
            i ←  ⌊i/2⌋ ;
      }
}
```

Ogni cammino di uno heap è una sottosequenza dell'array ordinata decrescente

$A[1] \geq A[2] \geq A[4] \geq A[8] \ldots$

altezza heap → h = $\Theta(\log_2 n)$ => costo dell'insertion-key è $O(\log_2 n)$
il logaritmo ha base 2 perchè è uno heap binario.
Quindi inserire una nuova chiave in uno heap è come ordinare l'array del cammimo

Es:

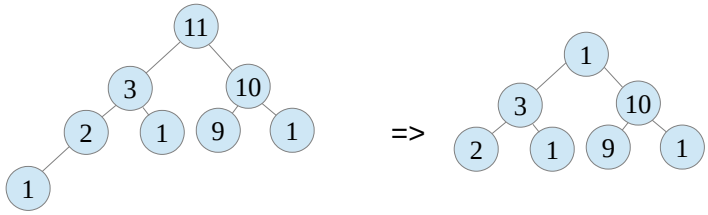


quindi inserire 5
è come ordinare l'array [11 | 7 | 2 | 5]

- Extract-max O(log n) → #figli · h → $\sum_{i=0}^{h}$ #figli

Estrarre il massimo significa ricostruire lo heap tolta la radice, avrà quindi come nuova dimensione n' = n – 1, il nuovo albero avrà quindi una foglia in meno
=> prendo l'ultima foglia e la metto al posto della radice tolta

Es:



=> invoco maxheapify sulla radice e risistemo l'albero

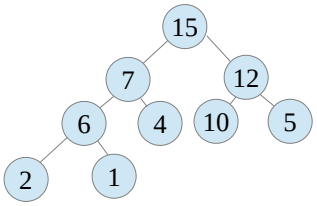
Codice:

```
extract-max ( A, n ) : key {
      max ← A[1];
      A[1] ← A[n];
      n ← n – 1;
      maxheapify(A, 1, n);
      return max;
}
```

il costo è il costo di maxheapify => $O(\log n)$

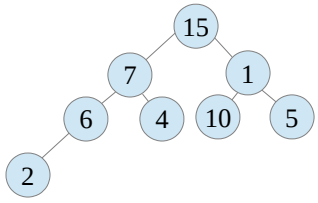
- Extract-Key
avrà come parametri (A, n, “posizione key nello heap”)
Il concetto alla base di questo algoritmo è simile a quello dell'extract-max, cioè prenderò l'ultima foglia dello heap e la porrò al posto dell'elemento estratto e poi risistemerò lo heap con maxheapify

Es: voglio fare extract-key(A, 9, 3)



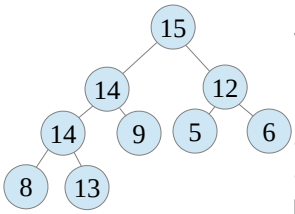
voglio cioè estrarre l'elemento 12

mi copio la chiave, e sposto il nodo con elemento 1 in posizione 3 e chiamo maxheapify su 1



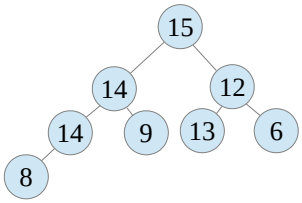
Purtroppo non basta controllare se il nostro albero rimane uno heap controllando solo i figli, come vediamo nell'esempio seguente:

Es: extract-key(A, 9, 6)



vogliamo cioè estrarre l'elemento con chiave 5

avrei quindi questo albero in cui per maxheapify è uno heap
anche se come possiamo notare è falso



Codice:

```
Extract-key ( A, n, index) : key {
     temp ← A[index]; A[index] ← A[n];
     temp2 ← A[n]; n ← n – 1;
     maxheapify(A, index, n);
     if (A[index] = temp2 ){ // cioè maxheapify non ha avuto effetto e forse
          i ← index;              // devo risalire
          while ( A[⌊i/2⌋] < A[i]  &&  ⌊i/2⌋≥1 ) {
               scambia( A[⌊i/2⌋] , A[i])
               i ←  ⌊i/2⌋ ;
          }
     }
}
```

Il costo è O(log n), nel caso di uno heap binario

nel δ-heap , avendo index, io so che sono circa nel livello log(index)

quindi il costo sarà max{ $\delta[\log_\delta n - \log_\delta(index)], \log_\delta(index)$ }

l'insert-key guadagna se il numero dei figli aumenta, mentre per l'extract funziona al contrario

Es:

$$\delta = 4$$

$$\text{Insert-key} = \log_4 n = \frac{\log_2 n}{2}$$

$$\text{Extract} = 4 \cdot \log_4 n = 2\log_2 n$$

Consideriamo come si memorizza uno heap con δ-figli

ad esempio se abbiamo un 4-heap e vogliamo il j-esimo figlio del nodo i considerando $1 \le j \le 4$ avremo:
4-heap (i, j)
4 (i – 1) + j + 1

Es: 4-heap(3,2) => 4 (3-1) + 2 + 1 = 11

Questa formula è generata, quindi funziona con ogni heap, anche uno binario

δ-heap $1 \le j \le \delta$

$\delta(i-1)+j+1$

per sapere il padre si avrà: parent(i) = $\lfloor (\frac{i-2}{\delta}+1) \rfloor$

Es: parent(11) = $\frac{11-2}{4}+1 = 3$

- Change-key
ricerca di una data chiave in uno heap, l'algoritmo è molto semplice, ogni volta leggo una chiave, se essa è maggiore di quella che sto cercando devo salire, altrimenti devo scendere

Tabella riassuntiva

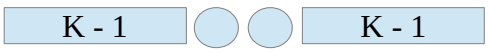
Tipo struttura	Find-max	Insert-key	Extract
2-heap	O(1)	$O(\log_2 n)$	$O(\log_2 n)$
δ-heap	O(1)	$O(\log_\delta n)$	$O(\delta \log_\delta n)$
Array qualsiasi	O(n)	O(1)	O(n) * (find-max)

Ricerca della mediana
mediana: elemento che nell'insieme ordinato si trova in posizione centrale

n = 2k + 1

La mediana è l'elemento di rango = $\lceil \frac{n}{2} \rceil = \lceil \frac{2k+1}{2} \rceil = k+1$

n = 2k



I due cerchi al centro indicano rispettivamente la “mediana inferiore”, in posizione $\frac{n}{2}$, e la mediana superiore in posizione $\lceil\frac{n}{2}\rceil+1$
Nel caso di insieme pari si prende come mediana di riferimento quella inferiore

Codice: dato un array A, trovare la mediana

```
straightforwardmedian ( A, n ) : integer {
     merge-sort( A, n );
     if ( n % 2 = 0 ) //cioè la lunghezza dell'array è pari
          return A[n/2] ;
     else
          return  A[⌈n/2⌉] ;
}
```

Il costo è $O(n\log n)$ dovuto al merge-sort

Es: A [10 | 3 | 7]
dopo merge-sort → [3 | 7 | 10] => return $A[\lceil\frac{3}{2}\rceil]=7$

E' un algoritmo ottimo per la ricerca della mediana ?
L'albero delle decisioni ci dice di guardare lo spazio delle soluzioni

|spazio soluzioni| = |{ A[1], . . . , A[n] }| = n

L'albero di altezza minima che allora n foglie è l'albero completo di h = $\log n$ => $\Omega(\log n)$
Ma tutti i dati devono essere controllati => $\Omega(n)$
Esistono due tipi di algoritmo per la ricerca della mediana, noi studiamo solo quello deterministico

Dati n numeri, li disponiamo come una matrice di 5 colonne

Es: n = 25
A ={ 10, 3, 7, 9, 1 | 4, 8, 15, 20, 21 | 23, 6, 2, 11, 14 | 28, 31, 29, 33, 34 | 1, 7, 8, 6, 3}
Dovrei quindi restituire l'elemento di rango 13

$\lceil\frac{n}{5}\rceil$ = numero di righe

10	3	7	9	1	Ordiniamo ciascuna riga della matrice	=>	1	3	7	9	10
4	8	15	20	21			4	8	15	20	21
23	6	2	11	14			2	6	11	14	23
28	31	29	33	34			28	29	31	33	34
1	7	8	6	3			1	3	6	7	8

In ciascuna riga della seconda matrice la mediana è quella in posizione 3
Calcoliamo quindi la mediana sulla colonna 3, questa la chiameremo
Mediana delle mediane (MM)= 11

Ordino rispetto a MM, mettendo quindi MM al centro, sopra quelli con chiave più piccola, e sotto quelli con chiave più grande

1	3	7	9	10	
1	3	6	7	8	< 11
2	6	11	14	23	
28	29	31	33	34	> 11
4	8	15	20	21	

Considerando che a destra e sotto la MM abbiamo elementi sicuramente maggiori, e sopra e a sinistra sicuramente minori possiamo affermare che MM non è sicuramente ne il massimo ne il minimo dell'array.
Questo è importante perché così siamo sicuri che, per la proprietà transitiva non potrà avere nel passaggio seguente uno dei due insiemi vuoto.
Questo non può essere affermato nel tipo di algoritmo probabilistico.

Quindi ora si usa l'MM per partizionare, ponendo a sinistra gli elementi più piccoli e a destra quelli più grandi

1	3	7	9	10	1	3	6	7	8	2	6	4	8
							11					
14	23	28	29	31	33	34	15	20	21			

14 elementi	MM	10 elementi

Rango(11) = 15

Gli elementi a destra non sono sicuramente la mediana perché hanno rango superiore a 13, devo andare quindi a cercarla tra gli elementi a sinistra, cioè devo andare a cercare l'elemento di rango 13 tra gli elementi a sinistra

Riassumendo l'algoritmo abbiamo che:
dato k = elemento da cercare
Selection (A , n , k) esegue i seguenti punti:

1) raggruppa gli n interi in $\lceil \frac{n}{5} \rceil$ gruppi
2) ordina ciascun gruppo
3) Determina MM, cioè la mediana della 3° colonna
   questa sarà una chiamata ricorsiva,
   diventando Selection (A[*,3], $\lceil \frac{n}{5} \rceil$, $\lceil (\lceil \frac{n}{5} \rceil / 2) \rceil$)
4) Partiziona gli elementi di A rispetto a MM
   cioè partiziona A in:
   LA = { x : x < MM }
   RA = { x : x $\geq$ MM }
   nota: rango(MM) = cardinalità degli elementi più piccoli + 1
   cioè rango(MM) = |{ x : x < MM }| + 1
5) Questa sarà una chiamata ricorsiva con il seguente codice:

```
if k = rango(MM)
      return MM
else if k < rango(MM)
      Selection ( LA, |LA|, k )
else if k > rango(MM)
      Selection ( RA, |RA|, k - rango(MM) )
```

$|\mathrm{LA}| \simeq \frac{n}{4} \simeq |\mathrm{RA}|$, così quando vado a ricorrere non si cade nel caso pessimo in cui MM è il minimo

1) 2) 3) 4) 5)

$T(n)=O(n)+O(1)\lceil\frac{n}{5}\rceil+T(\lceil\frac{n}{5}\rceil)+O(n)+\max\{\,T(|\mathrm{LA}|),T(|\mathrm{RA}|)\,\}$

2) = O(n), anche se uso un algoritmo non ottimo per ordinare il fatto che l'insieme è limitato (solo 5 elementi per gruppo) rende il costo un O(1)

Il numero di elementi sicuramente più grandi di MM sono almeno:

$3(\lceil(\lceil\frac{n}{5}\rceil/2)\rceil-2)$

in cui il '3' è dato da $\lceil\frac{5}{2}\rceil$, il '-2' invece è dovuto al fatto che la riga della MM ha solo 2 elementi più grandi e perché non si può sapere se l'ultima riga dell'array è completa

=> $|\mathrm{RA}|\geq 3(\lceil(\lceil\frac{n}{5}\rceil/2)\rceil-2)=3\lfloor\frac{n}{10}\rfloor+6$

se si vuol ricorrere sulla parte sinistra =>

$T(|\mathrm{LA}|)=T(n)-|\mathrm{RA}|\leq T(n)-3\lfloor\frac{n}{10}\rfloor+6\leq T(\frac{7n}{10}+6)$

per simmetria sarà uguale anche dall'altra parte

T(n) = → $O(n)+T(\frac{n}{5}+1)+T(\frac{7n}{10}+6)$ se $n>5$

→ $O(1)$ se $n\leq 5$

la somma dei due argomenti = $\frac{9}{10}n+7<n$ => $\in O(n)$

“Sveliamo l'arcano del 5” cit. Pinotti
Questo algoritmo funziona anche se i gruppi hanno un numero pari o diverso da 5
ad esempio se scegliessimo una matrice a 7 colonne avremmo:

$4(\lceil(\lceil\frac{n}{7}\rceil/2)\rceil-2)$, in cui il 4 è dato da: $\lceil\frac{7}{2}\rceil$

riscriviamo quindi l'equazione

$T(n)\leq O(n)+T(\lceil\frac{n}{7}\rceil+1)+T(n-4(\frac{n}{14}-2))=O(n)+T(\lceil\frac{n}{7}\rceil+1)+T(\frac{5n}{7}+8)$

la somma dei due argomenti è ancora più piccola di n quindi il costo rimane O(n)

Però le costanti sono più grandi poiché ordinare gruppi di 7 ha un costo maggiore. Dovrei allora prendere il numero dispari più piccolo.

Quindi perché non si prende il 3 ?

$\lceil\frac{3}{2}\rceil(\lceil(\lceil\frac{n}{3}\rceil/2)\rceil-2)$→ $T(n)\leq O(n)+T(\frac{n}{3}+1)+T(n-2(\frac{n}{6}-2))\leq O(n)+T(\frac{n}{3}+1)+T(\frac{2n}{3}+4)$

la somma dei argomenti questa volta è n più qualcosa => con gruppi di 3 non ho un costo O(n)

Mentre se si provasse con un numero pari si sbilancerebbero le ripartizioni

Esercizi:

1) Dato $V[i]$ con $1 \le i \le n$ ordinato
Si ha double gap se $V[i+1]-V[i] \le 2$
verificare se esiste e trovarlo

esempio: n = 5 A [6 , 7 , 9 , 10 , 11]

```
verifica( V , p , r ) {
     if ( 1 + p < r ) {    // = r – p + 1 > 2 , cioè se ho solo 2 elementi
          q ← ⌊(p+r)/2⌋ ;
          if ( V[q]−V[p] ≥ q−p+1 )
               return verifica( V , p , q );
          else if ( V[r]−V[q] ≥ r−q+1 )
               return verifica( V , q , r);
          else
               return  −∞ ;   //cioè non esiste
     }
     else {
          if ( V[r] ≥ V[p]+2 )
               return p;
          else
               return  −∞ ;
     }
}
```

Il caso base su questo esercizio è " 1 + p < r" cioè quando sono rimasto con un sottoarray di soli 2 elementi, ciò deriva dal fatto che nella ricorsione, sia che ricorro a destra o a sinistra porto dietro in entrambi i 2 sottoarray l'indice q, così da essere sicuro di non perdere nessuna coppia. Quindi è importante fermarsi ad una coppia di elementi, perché se usassimo il solito
"p < r" si rischierebbe di entrare in un loop quando si hanno 3 elementi.
Infatti da 3 elementi creerei 2 sottoarray di 2 elementi da cui poi si formerebbe uno di uno (che terminerebbe) e uno di 2 che continuerebbe il ciclo.

$T(n)=T(\frac{n}{2})+O(1)$ Più chiamate ricorsive ma mutualmente esclusive => $\in O(\log n)$

è ottimo ?
=> spazio soluzioni = 1 , 2 , … , n , $-\infty$
$\log|\text{sp sol}|=\log n$ → $\Omega(\log n)$ → è ottimo

2) variante dell'esercizio precedente
$V[i]<V[i+1] \ \forall\, i \in 1 \le i \le n-1$
Si ha un punto fisso se $\exists\, i : V[i]=i$

Soluzione immediata: scansione $O(n)$
Controlliamo se è ottimo:
$\log|\text{sp sol}|=\log(n+1)$ → $\Omega(\log n)$

quale è un equazione di ricorrenza che mi dia $\Omega(\log n)$?

$T(n)=2T(\frac{n}{2})+O(1) \in O(n)$

invece $T(n)=T(\frac{n}{2})+O(1) \in O(\log n)$

quindi cercheremo un algoritmo basato sul DIVIDE-ET-IMPERA

Osserviamo che:
$V[q]>q \rightarrow V[t]>t \ \forall t>q \quad V[t]\geq V[q]+t-q$
ma essendo $V[q]-q>0 \rightarrow V[t]\geq V[q]+t-q>0$
è inutile quindi controllare gli elementi a destra di q

```
puntofisso( V , p , r ) {
     if ( r > p ) {
          q ← ⌊(p+r)/2⌋ ;
          if ( V[q]=q )
               return q;
          else if ( V[q]>q )
               return puntofisso( V , p , q – 1 );
          else // V[q]<q
               return puntofisso( V , q + 1 , r );
               /*
                 t<q  V[t]≤V[q]−q+t  V[q+1]≤V[q]−1
                          V[q]−q<0 → V[t]<t
               */
     }
     else
          if ( V[r]=r )
               return r;
          else
               return //non esiste
}
```

3) def: a è un elemento dominante in V se:

occ(a) → $\lfloor \frac{n}{2} \rfloor+1=\frac{n}{2}+1$ se n è pari

→ $\lceil \frac{n}{2} \rceil$ se n è dispari

esempio: n = 6 V= { 3 , 7 , 7 , 8 , 7, 7 }7 numero dominante

Ordiniamo V con merge-sort in $O(n\log n)$
V' = V ordinato
n pari:

$V'[\frac{n}{2}+1]=V'[n]$ elemento dominante $V'[n]$

oppure $V'[1]=V'[\frac{n}{2}]$ elemento dominante $V'[1]$

n dispari:

$$V'[\lceil \frac{n}{2} \rceil] = V'[n]$$

$$V'[\lceil \frac{n}{2} \rceil] = V'[1]$$

$T(n) = O(n\log n) + O(1) + O(n)$ → $\in O(n\log n)$
l'ultimo $O(n)$ serve a controllare che l'elemento trovato sia effettivamente un elemento dominante

Però possiamo fare meglio, possiamo usare la "selection" (con un costo $O(n)$)
per la ricerca della mediana più $O(n)$ per il check

4) Ordinare n numeri nel range $[1 \dots n^2]$
- $T_{merge} = T_{heap} = O(n\log n)$
- $T_{Q-S} = O(n^2) = T_{insert} = T_{count}$
- T_{R-S} con base = 2 → $2\log(n+2) \in O(n\log n)$
  con base = 2^r $\frac{2\log n}{r}(n+2^r)$ ma con $r = \log n$ → $2(n+n)$ → $\in O(n)$

5) Trovare k-1 punti che dividono l'insieme in k parti uguali
cioè k-1 valori tali che si hanno k-intervalli di uguale ampiezza

esempio: k = 4
A → 3 , 8 , 7 , 5 , 10 , 11 , 9 , 1
1 , 3 | 5 , 7 | 8 , 9 | 10 , 11
l'11 è dato per default perché è il massimo
cerco quindi elementi in posizione:
$\frac{n}{k}$ → $2\frac{n}{k}$ → $3\frac{n}{k}$ → … → $(k-1\frac{n}{k})$ → $k\frac{n}{k}$
quindi gli elementi di posizione $i\frac{n}{k}$ con $1 \le i \le k-1$

se k non divide n avrò insiemi uguali non devono differire al più di uno

esempio: n = 10 k = 4
A → 3 , 8 , 7 , 5 , 10 , 11 , 9 , 1 , 16 , 13
avrò 4 gruppi , 2 di cardinalità 3 e 2 di cardinalità 2
1 , 3 , 5 | 7 , 8 , 9 | 10 , 11 | 13 , 16
quindi si prendono $\lfloor \frac{n}{k} \rfloor$ elementi e ne rimangono $|n|_k$

esempio: $\lfloor \frac{10}{4} \rfloor = 2$ e rimangono $|10|_4 = 2$

quindi $|n|_k$ gruppi $\lfloor \frac{n}{k} \rfloor + 1$ e $k - |n|_k$ di $\lfloor \frac{n}{k} \rfloor$

Problema: Trovare k – 1 statistiche d'ordine

1a soluzione:
con ordinamento => ordina(A) // $O(n\log n)$

estrai → $A[\, i\,(\lfloor\frac{n}{k}\rfloor+1)\,]$ $\quad 1\le i\le |n|_k$

→ $A[\, i\,(\lfloor\frac{n}{k}\rfloor) + |n|_k\cdot(\lfloor\frac{n}{k}\rfloor+1)\,]$ $\quad 1\le i\le k-|n|_k$

2a soluzione:
possiamo fare meglio usando la selection
se invocassimo la selection k – 1 volte la selection avremmo
però $O(nk)$ che è migliore se e solo se $k\le\log n$

for i ← 1 to $|n|_k$
selection(A , $i(\lfloor\frac{n}{k}\rfloor+1)$);
for i ← 1 to $k-|n|_k$
selection(A , $|n|_k(\lfloor\frac{n}{k}\rfloor+1)+i\lfloor\frac{n}{k}\rfloor$);

esempio: 3 , 8 , 7 , 5 , 10 , 11 , 9 , 1
- faccio selection su rango(2) = 3
  1 , <u>3</u> | 7 , 5 , 10 , 11 , 9 , 8
  dimensione del 2o sottoarray è n - $|n|_k$
- richiamo la selection sul sottoarray destro
  1 , 3 | 5 , <u>7</u> | 10 , 11 , 9 , 8
  dimensione: $n+(n-\lfloor\frac{n}{k}\rfloor)+(n-2\lfloor\frac{n}{k}\rfloor)$ => $kn-\lfloor\frac{n}{k}\rfloor\sum_1^{|n|_k} i$ non è ancora buono

- proviamo a bilanciare
facciamo selection su rango centrale
3 , 5 , 1 , <u>7</u> | 10 , 11 , 9 , 8 $O(n)$
ora se si richiamerà selection a destra e a sinistra, cioè su array di dimensione $\frac{n}{2}$ che costerà in totale $O(n)$
quindi parto dall'indice più centrale e pago $O(n)$
sui 2 sottoarray ricercherò il loro centrale, sempre in $O(n)$
=> $\sum_{t=0}^{l} 2^i = k$ l = livello in cui mi devo fermare
$2^{l+1}-1=k$ → $l=\log(k+1)-1$ => l'algoritmo costa $l\cdot n \in O(n\log k)$

Considerazioni su min-heap
find-min $O(1)$
extract-min $O(\log n)$

quanto costa d-ary (cioè in un d-heap)
se d = 3 extract costa $O(3\log_3 n)=O(\log n)$
se d = $f(n)$ extract $O(f(n)\log_{f(n)} n)$

esempio: d = $\sqrt{(n)}$ $\log_{\sqrt{(n)}} n=2$ ma è comunque $O(\sqrt{(n)}\cdot 2)\in O(\sqrt{(n)})$

insert-key → d = 2 $O(\log_2 n)$
d = 3 $O(\log_3 n)$
non c'è la costante davanti perché mi confronto esclusivamente con il padre

d = $f(n)$ $O(\log_{f(n)} n)$

Esercizio:
1) Dato un array di elementi compresi tra [1 , 2 , 3]
trovare il numero con maggior numero di occorrenze

lower bound è $O(n)$
=> anche un algoritmo di scansione con 3 contatori va bene perché costa $O(n)$

Se fossero ordinati basta cercare l'ultima occorrenza di 1 e l'ultima di 2 così conosco le occorrenze di tutti e 3
n_1 = occorrenze di 1
n_2 = occorrenze di 2
max (n_1 , n_2 - n_1 , n - n_2)
il maggiore sarà quello con numero di occorrenze maggiori

2) Data una matrice M di k-righe e n-colonne
le righe sono ordinate in senso crescente
Problema: restituire un unico array S ordinato in senso crescente

1a soluzione: faccio merge su tutte le righe in $O((nk)\log(nk))$

non è la soluzione migliore però so che devo pagare almeno nk poiché devo scorrere tutti i valori per creare l'array unico

Osserviamo che il candidato ad essere il minimo può essere solo nella prima colonna.

2a soluzione: cerco il minimo nella prima colonna, una trovato lo inserisco in S poi sostituisco l'elemento appena preso con il suo successivo nella riga e ricerco il minimo.
Pago però $O(nk^2)$ poiché la ricerca del minimo costa O(k) e devo farla per ogni elemento

Osserviamo che è inutile ricercare il minimo se io ho sostituito un unico elemento nel mio insieme di minimi

3a soluzione:
=> posso creare un min-heap della prima colonna $O(k)$ ed estraggo il minimo (cioè la radice) in $O(1)$ e ci inserisco il suo successore e ricostruisco lo heap
in $O(\log k)$ => $T(n)=O(k)+O((nk)\log k)$ poiché devo fare per ogni elemento lo stesso processo, ma un unico build-min-heap

4a soluzione:
un altro modo di generalizzare il merge

posso fare il merge di 2 righe passando da k a $\frac{k}{2}$ array ordinati

$\frac{k}{2}2n = kn$ ← costo

pago kn ad ogni passo

$i: \frac{k}{2^i} = 1$ → $i = \log k$ => $O(kn \log k)$

3) Dato un array ordinato un shift di t-posizioni si ha quando sposto ogni elemento di t posizioni reinserendole all'inizio dell'array

1 | 8 | 9 | 10 | 18 | 26 | 31 | 33
t = 3
26 | 31 | 33 | 1 | 8 | 9 | 10 | 18

problema: dato un array shiftato determinare t
se c'è stato il punto di rottura il 1° elemento non è il minore di tutto l'array

se $A[q] < A[r]$ o $A[p] < A[q]$ è ordinato

```
trovashift ( A , p , r ) {
     if ( r=p+1 ) && ( A[p]>A[r] )
          return p;
     else
          return 0;  //se non c'è stato shift
     else {
          q ← ⌊(p+r)/2⌋ ;
          if ( A[p]<A[q] )
               return trovashift( A , q , r );
          else
               return trovashift( A , p , q );
     }
}
```

Oggi introduciamo la **programmazione dinamica**
è una tecnica che risolve una classe ampia di problemi, ed ha alcune cose in comune con la tecnica DIVIDE-ET-IMPERA, e viene utilizzata per risolvere problemi di ottimizzazione.
"programmazione" intende il fatto che memorizzeremo i risultati parziali in una matrice.

Elementi in comune con DIVIDE-ET-IMPERA:
- DIVIDE-ET-IMPERA "spezzetava" in sottoproblemi e ricombinava, ma i sottoproblemi erano disgiunti (es: merge-sort, non succede mai che 2 sottoproblemi lavorano sulla stessa porzione di array)
- La prog. dinamica invece ha sottoproblemi condivisi

Es: diciamo che in un array c'è una porzione A che va risolta più volte, invece che ripetere l'operazione memorizzeremo il risultato in una tabella(matrice)
poiché se dovessi risolverlo più volte aumenterebbe la mia complessità, invece così rimane polinomiale

esempio:
Supponiamo di avere un asta di acciaio di lunghezza n e volete dividerla in porzioni più piccole, a seconda della lunghezza si avranno profitti diversi

i	1	2	3	4	5
P_i	1	5	8	9	10

Supponiamo di avere n = 5,
quale è il modo migliore per avere il maggior profitto ?

Varie soluzioni ammissibili:
S=(1,1,1,1,1) = 5 ; S=(1,2,2) = 11 ; S=(5) = 10 ; S=(4,1) = 10

cioè enumerare tutte le possibili soluzioni, ma si avrebbe un costo esponenziale, basandoci su "taglia un pezzettino di lunghezza i e risolvi il problema di dimensione n-i", cioè dare una definizione ricorsiva del problema che sarebbe =>

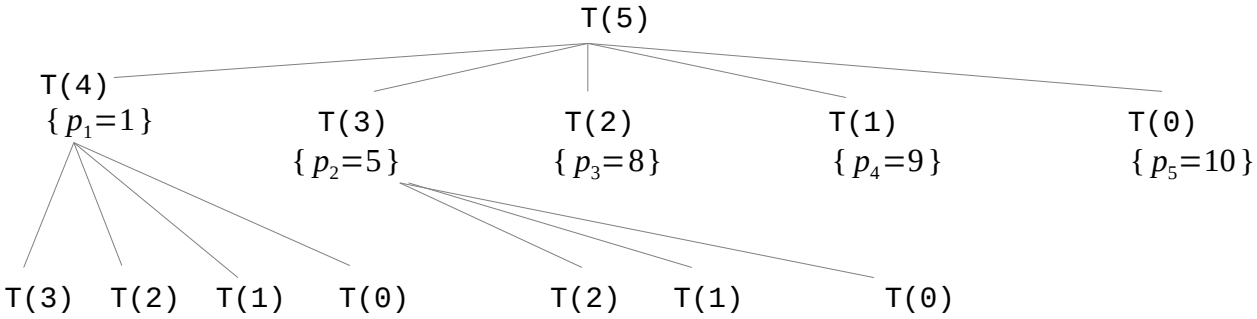
Sol(n) = $\rightarrow \quad max\{p_i + sol(n-i)\} \quad (1 \le i \le n)$ se $n \ge 1$

$\rightarrow \quad 0 \quad$ se $n = 0$

con n = 5 avremmo quindi:

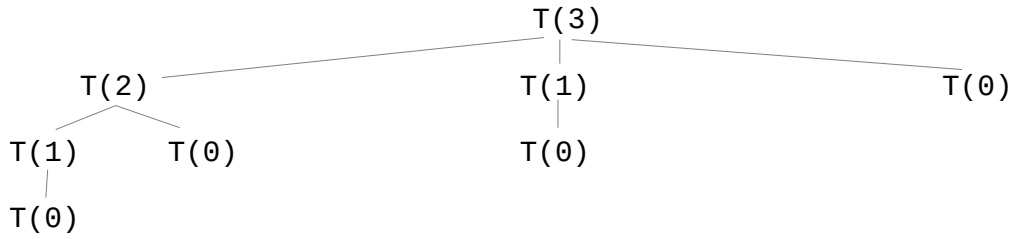
$p_1 + sol(4)$ oppure $p_2 + sol(3)$ oppure … oppure $p_5 + sol(0) (= p_5)$

ci serve una base della ricorsione => $sol(0) = 0$



notiamo quindi che alcuni T(i) li ripetiamo più volte

Concentriamoci un attimo sul sottoproblema T(3)



=> le soluzioni con n = 3 sono { (1,1,1) , (2,1) , (3) }
= 3 = 6 = 8

memorizzo il massimo valore e al prossimo T(3) mi ricorderò che vale 8

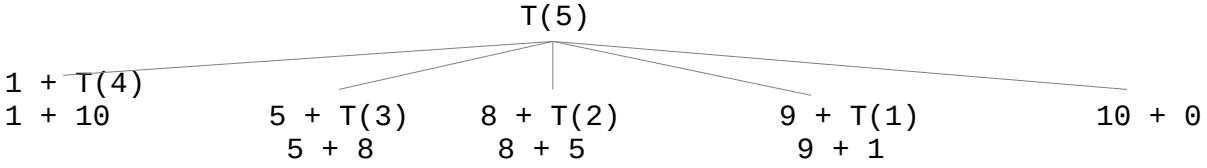
Perché è interessante avere una matrice per ricordare i parziali ?
Perché se costruiamo l'albero avremo 2^n nodi => $\Omega(2^n)$ => complessità esponenziale

mentre se faccio la matrice:

=> M[i] = sol(i)

quindi per esempio per risolvere T(3) non l'ho più fatto con un albero di 8 nodi ma con uno di 3

Ritorniamo al problema principale e avremo:



Solo 5 nodi invece che 2^5

=> risolvere T(n), se conosco i risultati di T(n-i) fino a T(1)

M[n] = O(n)

però dovrò fare M[0] M[1] M[i] M[n] con un costo O(i)

=> $\sum_{i=0}^{n} i = O(n^2)$ = complessità in tempo

complessità in spazio sarà comunque O(n) , introduciamo quindi il costo in memoria poiché ci serve lo spazio per salvare i dati

Si ha comunque bisogno di una definizione ricorsiva per risolvere un problema in programmazione dinamica, potrei risolverlo con un approccio top-down con l'albero (complessità esponenziale).
Mentre con la programmazione dinamica userò un approccio bottom-up per risolvere prima i problemi più piccoli e utilizzarli poi per risolvere i problemi più grandi.

Notiamo però che dalla tabella io ho il valore della soluzione (massimo profitto = 13) ma non ho la soluzione (S=(2,3))

adesso devo cercare di capire quale è la soluzione

Ogni volta che trovo il massimo mi devo ricordare quale è il taglio che me lo ha dato (se ne ho più di uno è a mia scelta)
Oltre all'array per il profitto ne avrò quindi un altro (di ugual lunghezza) in cui segnerò quale è il taglio che mi ha dato l'ottimo.

R	0	1	2	3	4	5
	0	1	2	3	2	2

Serve quindi a ricostruire la soluzione

io so che M[5] = 13
=> R[5] = 2 => 5 – 2 = 3 => R[3] = 2 cioè S=(R[5], R[5-R[5]])=(3,2)

esempio se avessi avuto n = 4
sol = (R[4], R[4–R[4]])

posso avere più elementi ma R[(4–R[4]) – R[4–R[4]]] = 0

M[6] = max { $p_i + M[n-i]$ } $1 \le i \le 6$
= { 1 + M[5], 5 + M[4], . . . , 10 + M[1] }
= { 14, 15, 16 , 14 , 11, $-\infty$ + M[0] }
=> max = 16 mi devo ricordare che ho tagliato il pezzetto di lunghezza 3

					5	6
=> R					2	3

M[6] = 16 => sol(3,3)

Tutto questo si basa sulla proprietà (con un termine improprio) della sub-ottimalità
cioè si deve cercare la soluzione ottima per soluzioni più piccole (“sub-”)

Esempio 2: knapsack
zaino di capacità $M \in \mathbb{N}$, n oggetti $x_1, x_2, \ldots, x_n$
ciascun oggetto ha un (“ingombro”) volume

volume: W_1 W_n $W_i \in \mathbb{N}$

profitto: p_1 p_n $p_i \in \mathbb{R}$

M = 10 4 oggetti

x	1	2	3	4
W	1	3	2	5
p	1	5	8	6

Max{ $\sum_{i=1}^{n} y_i \cdot p_i$ } y_i = → 0 se $x_i \notin sol$

$\sum_{i=1}^{n} y_i \cdot w_i \leq M$ → 1 se $x_i \in sol$

quindi avrò un vettore booleano che mi indicherà se prendo o no un oggetto

S=(1,0,0,1) indica che ho preso il 1° e l'ultimo oggetto che hanno
$W_1 + W_n = 6 \leq 10$ e un $P_{tot}=7$
devo cercare quella con profitto massimo

S=(1,1,1,1) non è una soluzione ammissibile poiché $W_{tot}=11>10$

spazio soluzioni = { Tutti i possibili sottoinsiemi ammissibili = 2^n }
=> con metodo enumerabile avrei $\Omega(2^n)$

definizione ricorsiva ?

Pensando top-down si avrà alla fine un ingombro $m \leq M$

prendendo l'oggetto i-esimo si hanno 2 casi:
1) l'oggetto i non sta nello zaino, avrò quindi $m - W_i < 0$
2) $m - W_i > 0$

considero un ordine degli oggetti e guardo l'oggetto x_n

(Z) zaino = 7
S[m,n] → $x_n \notin Z$ Z = { $x_1,\ldots,x_{n-1}$ } => m
→ $x_n \in Z$ Z = { $x_1,\ldots,x_{n-1}$ } U { x_n } con $x_1,\ldots,x_{n-1} = m - W_n$

nel secondo caso ho sicuramente p_n e ho in totale il miglior zaino che potevo avere con x_{n-1} oggetti
=>
S[m,n] → $S[m,n-1]$
→ $S[m-W_n,n-1]+p_n$
con $m-W_n$ ← capacità zaino rimasta
n-1 ← quali oggetti ci ho messo dentro, in questo caso memorizzo il più grande

mi ricorderò poi il massimo

S[m,n] = max{ $S[m,n-1], S[m-W_n,n-1]+p_n$ } con $n>0, m>0$
S[0,n] = 0 quindi con $m \leq 0$
S[m,0] = 0 quindi con $n \leq 0$

nota: Non è detto che la soluzione ottima riempa completamente lo zaino

pseudocodice della definizione ricorsiva:

```
zaino ( m : integer , n : integer) : real {
      if(m = 0) return 0;
      if(n = 0) return 0;
      else {
            q ← max( p_n + zaino( m−W_n , n-1), zaino(m, n-1));
            return q;
      }
}
```

Tuttavia questo codice ha una complessità esponenziale ($\Omega(2^n)$)
Con la programmazione dinamica la nostra tabella di memorizzazione dovrà essere M(n,m) = sol(m,n)

N \ m	0	...	m'	...	M
0	0	0	0	0	0
...	0				
n'	0				
...	0				
n	0				

significa 0 oggetti

Considero il profitto massimo che posso avere con n' oggetti e capacità uguale a m'
M(n',m') = sol(m',n')

es:

	W	p
x_1	1	10
x_2	2	5
x_3	3	1
x_4	4	8

N | m	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	10	15	15	15	15
3	0	10	10	15	15	15	16
4	0	10	10	15	15	18	18

In M[1,1] posso scrivere o il valore precedente M[0,1] o il primo oggetto $x_1(W_1 \leq m)$ => 10 + M[0,0](= 0) = 10, max{0,10} = 10

il meglio che possiamo fare sulla 1ª riga è per forza 10 perché posso usare un solo oggetto

es:
M[1,2] = max{ M[0,2] , M[0,1] + 10 }
M[2,3] = max{ M[1,3] , M[1,3-2] + 5 }
(3-2 ← il 2 è l'ingombro (w_2) dell'oggetto che sto considerando)
M[2,1] = max{ M[1,1] , M[1,1-2] + 5 } = max { 10, 5 + 0 }

```
pseudocodice:
zaino ( M : integer , n : integer ) : real {
     for i ← 0 to n do
          M[i,0]  ← 0
     for j ← 0 to M do
          M[0,j]  ← 0
     for i ← 1 to n do
          for j ← 1 to M do
               M[i,j] ← M[i-1,j]
               if(  W_i ≤ j  )
                    if(M[i,j] < M[i-1, j- w_i ] + p_i )      // j − W_i ≥ 0
                         M[i,j] ← M[i-1,j- W_i ] +  p_i
}
```

quanto è costato trovare M[n,m] ?
(n + 1) + (m + 1) + (n + 1) (m + 1) O(1) => O(M·n)

$W_i \in \mathbb{N}^+ \quad M \in \mathbb{N}^+$
questa soluzione vale esclusivamente se loro appartengono ai naturali, se appartenessero ai reali dovrei risolvere gli zaini con qualunque capacità reale, cioè infiniti

Potrei avere capacità di uno zaino (M) molto grande, il che rende il costo meglio dell'esponenziale ma “non proprio polinomiale”

Un problema è caratterizzato da 2 cose:
number(P) ← il numero di bit che servono a rappresentare i numeri del mio problema (p_i , W_i , M) es: $\log_2 p_i \, \forall i$
size(P) ← sono il numero di oggetti matematici coinvolti nel problema (si riferisce ai n oggetti, 1 zaino)

Quindi se vado a guardare l'istanza del mio problema (es: $x_1,\ldots,x_n$) posso andare a contare i bit che mi serviranno.

Un problema è definito polinomiale in tempo se quando richiede un tempo si ha T(n) = max{ $\text{number(P)}^k, \text{size(P)}^k$ } k = O(1), cioè elevati ad una costante

quindi per essere polinomiale dovrei avere n^k per il size o $n\log_2(max\{\ \})$ poiché sono limitati dal più grande

es: il numero più grande che noi rappresentiamo nel nostro esempio precedente è 10 => $\#\text{bit} \le (2n+1)\lceil \log_2 10 \rceil$
2n ← perché devo scriverli tutti (es: la nostra tabella contiene 8 elementi infatti n = 4) per un massimo di $\log_2 10$ bit, poiché appunto 10 è il più grande non ci sarà nessun numero che dovrà essere rappresentato con più bit

possiamo scrivere la nostra complessità come number(P) elevato a una costante ? (usiamo number perché è sicuramente più grande di size nel nostro caso)

$M = 2^{\log_2 M} \le 2^{number(P)}$
M non è legato a number in modo polinomiale ma esponenziale quindi la nostra complessità è **pseudopolinomiale,** questa complessità cresce con la

grandezza del numero (cioè più sarà grande lo zaino più crescerà la complessità)

Nel caso di un ordinamento basato sui confronti ci si basa solo sul size(P), non ci interessa (a parte algoritmi come counting-sort) della grandezza dei numeri
differente è il caso dello zaino

ritorniamo al nostro problema:
abbiamo bisogno di un altra struttura d'appoggio per segnarci per ogni valore da quali oggetti è composto
R

N \ m	0	1	2	...	6
0	0	0	0	0	0
1	0		1		
2	0		0		
3	0		0		
4	0				1

Questa sarà una matrice booleana che segnerà 1 se l'oggetto fa parte di quelli utilizzati per avere valore corrispondente nella matrice M

M[4,6] = max{ M[3,6] , 8 + M[3,2] }
$y_4=0$ $y_4=1$

R[n,m] = → 1 (ho preso l'oggetto) se M[n,m] = p_n + M[n-1,m- W_n]
→ 0 se M[n,m] = M[n-1,m]

quanto costa scrivere R ?
ogni volta visito una cella sola diminuendo il numero di oggetti, visiterò allora al massimo n oggetti

se devo soltanto calcolare il valore della funzione obiettivo (matrice M) non devo costruire tutta la matrice, potrei lavorare semplicemente con 2 righe, risparmiando spazio in memoria

Punti cruciali:
- definizione ricorsiva
- definizione top-down
- trasformazione a bottom-up
- creare la matrice
- deve valere la sub-ottimalità

di cui 2 fasi principali → calcolare valore funzione obiettivo
→ trovare la soluzione

Zaino con ripetizioni
nella definizione ricorsiva non vado più a prendere "n-1"
ma n poiché devo riconsiderare l'oggetto stesso avendo più istanze di esso

nella matrice quando avrò finito di scrivere la riga "n" devo copiarla nella riga "n+1" quindi vale la seguente regola:

$M[\,n\,,\,m\,] = max\{\; p_n + M[\,n\,,\,m - V_n\,]\,,\, M[\,n-1\,,\,m\,]\,\}$
quando l'oggetto n-esimo inizia a starci si cercherà di replicarlo fino alla fine

Es: 2 oggetti
1 $p_1=5$ $V_1=2$
2 $p_2=2$ $V_2=1$
M = 4 ← capacità zaino

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	5	5	10
2	0	2	5	7	10

M[1,4] ← 2 oggetti di tipo 1

Il resto del problema sarà analogo allo zaino semplice, in quello con ripetizioni l'unica cosa che cambia è la definizione ricorsiva

Esempio:
Supponiamo di avere n matrici e di voler fare la loro moltiplicazione

$A_1 \cdot A_2 \;\ldots\; A_n$ $A_i = p_{i-1}\; x\; p_i$ ← dimensione delle matrici
es: $A_1 \cdot A_2$ ← $(p_0 x p_1)(p_1 x p_2)$

Quanti sono i prodotti scalari che devo fare ?

Nell'esempio qui sopra io devo riempire una matrice che sarà di dimensione $p_0 x p_2$, il numero dei prodotti quindi è dato dalla dimensione delle matrici => $p_0 \cdot p_1 \cdot p_2$

se devo fare il prodotto tra 3 matrici
$A_1 \cdot A_2 \cdot A_3$, il prodotto finale è uguale rispetto a quale matrice moltiplico per prima, ma sarà diverso il numero di prodotti scalari

$(A_1 \cdot A_2) \cdot A_3$ $C_{p_0 x p_2} \cdot A_{3\; p_2 x p_3}$
$p_0 x p_2$ $p_2 x p_3$ => $p_0 p_1 p_2 + p_0 p_2 p_3$

Se faccio invece:
$A_1 \cdot (A_2 \cdot A_3)$ $A_{1\; p_0 x p_1} \cdot C_{p_1 x p_3}$ => $p_0 p_1 p_3 + p_1 p_2 p_3$

e $p_0 p_1 p_2 + p_0 p_2 p_3 \neq p_0 p_1 p_3 + p_1 p_2 p_3$
il problema quindi più che di ottimizzazione diventa un problema di minimizzazione

$\prod_{i=1}^{n} A_i = A_1 A_2 \dots A_n$ cerchiamo quindi di minimizzare il numero di prodotti scalari da fare

es: con 4 matrici ho 5 possibilità di parentesizzare le matrici per diversi costi finali

$A_1 A_2 A_3 A_4$
$A_1 | (A_2 A_3 A_4)$
$(A_1 A_2) | (A_3 A_4)$
$(A_1 A_2 A_3) | A_4$

$1 \le k \le n-1$ con k il punto in cui posso mettere la parentesi

Quindi se faccio la definizione ricorsiva sarà:
- Il caso base è quando ho 1 o 2 matrici soltanto
- M[i,j] = numero <u>minimo</u> di prodotti scalari che servono per moltiplicare le matrici $A_i A_{i+1} \dots A_j$ con $j \ge i$

$$M[i,j] = \begin{cases} \to 0 & i=j \quad // \text{ una sola matrice} \\ \to p_{i-1}\, p_i\, p_{i+1} & i=j-1 \quad // \text{ o } j=i+1 \text{ cioè solo 2 matrici} \\ \to \min\{ M[i,k]+M[k+1,j]+p_{i-1}\, p_k\, p_j \} & // \ i<j-1 \\ \qquad 1 \le k \le j-1 \end{cases}$$

es:
$A_1 | (A_2 A_3 A_4) = M[1,1]+M[2,4]+p_0\, p_1\, p_4$
// $p_0\, p_1\, p_4$ ← è l'ultima moltiplicazione tra A_1 e il risultato
// della parentesi

$(A_1 A_2) | (A_3 A_4) = M[1,2]+M[3,4]+p_0\, p_2\, p_4$
$(A_1 A_2 A_3) | A_4 = M[1,3]+M[1,1]+p_0\, p_3\, p_4$

Se usassimo l'albero per risolvere la definizione ricorsiva sopra espressa ci verrebbe $\Omega(2^n)$

quindi per poter trasformare il nostro costo in polinomiale dobbiamo sfruttare la memorizzazione

Esistono sottoproblemi ripetuti, quindi se si memorizza la soluzione si risparmia perché non devo risolvere di nuovo operazioni già fatte ma solo leggerne il risultato

Tutto questo funziona perché vale la sub-ottimalità, cioè risolvo problemi più piccoli con valore ottimo

Cioè se avessi, posto per assurdo, $M'[i,\bar{k}]$ con costo maggiore di $M[i,\bar{k}]$, con $\bar{k}$ la posizione ottima per porre la parentesi, e M' dico che è ottima anche se richiede più prodotti scalari.
Notiamo però che non cambia $p_{i-1}\, p_{\bar{k}}\, p_j$ in entrambi i casi, quindi M' non è ottima perché $M[i,\bar{k}]$ era ottima (perché usava meno prodotti scalari), quindi devo rimpiazzarlo (tecnica del taglia e incolla), tolgo la soluzione non ottima però i costi al livello non devono cambiare (come in questo caso), sennò non potrei confrontare.

La matrice d'appoggio va riempita in un modo intelligente, cioè devo riempirla in ordine perché per scriverla uso valori interni alla matrice stessa scritti in precedenza

M n x n

	1	…	J	…	n
1					
...					
i					
...					
n					

Minimo numero di prodotti scalari per moltiplicare le matrici da i a j

- sarà una matrice triangolare superiore perché non ha senso per i > j
- sulla diagonale saranno tutti 0

Es:

	1	2	3	4
1	0	V		
2	-	0	V	
3	-	-	0	V
4	-	-	-	0

Anche M[1,2] è un caso base , quindi indicheremo con V tutti i casi base

M[1,3] = $min\{M[1,1]+M[2,3]+p_0 p_1 p_3 , M[1,2]+M[3,3]+p_0 p_2 p_3\}$
abbiamo tutti e 4 i campi richiesti => posso riempire quella cella

non posso invece, per esempio, riempire subito la cella M[1,4] poiché non avrò M[2,4] => devo per forza riempirla per diagonali

Codice:

```
//inizializzo
for i ← 1 to n
     M[i,i] ← 0
for l ← 1 to n-1 {
     for i ← 1 to n-l {
          M[i, i+l+1] ← +∞
          for k ← i to i+l
               M[i,i+l+1] ← min{M[i,i+l+1], M[i,k]+ M[k+1, i+l+1]+
                                                          p_{i-1} p_k p_{i+l+1} }
     }
```

es:
M[1,2] = M[1,1]+ M[2,2] + $p_0 p_1 p_2$
     che è come fosse $p_0 p_1 p_2$ poiché nella diagonale ci sono solo 0
M[1,3] = M[1,1],M[2,3], $p_0 p_1 p_3$
     poi dovrò fare k = 2 => M[1,2],M[3,3], $p_0 p_2 p_3$

Calcoliamo il costo:
$\Omega(n^2)$ per forza perché devo almeno riempire la matrice, la riempio per metà ma $\frac{n^2}{2} \in \Theta(n^2)$. Il costo per riempire una cella è il "for" più interno (con l+1 tentativi), "l" va al più a "n-1" => per riempire una cella faccio al più "n" tentativi => $n^2 \cdot n \in O(n^3)$

Avremo un altra tabella della stessa dimensione in cui dovremo ricordarci il valore di k che ci dà il taglio minimo
S[i, i+l+1] ← arg{M[i,i+l+1]} //questa riga va dentro l'ultimo for

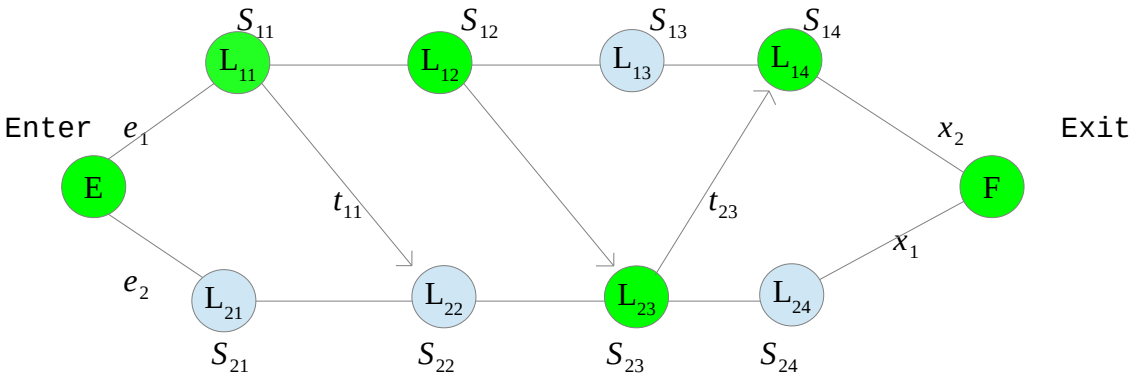
La matrice S verrà inizializzata a "-1" che è una marca qualsiasi che verrà sicuramente sovra scritta

es:
1) se in S[1,4] = 2 significa che per arrivare alla soluzione migliore ho messo la parentesi dopo la 2ª matrice($(A_1\ A_2)(A_2\ A_3)$)

2) in S[1,5] = 3 => $(A_1\ A_2\ A_3)(A_4\ A_5)$
per il secondo taglio guarderò S[1,3] = 2 => $((A_1\ A_2)\ A_3)(A_4\ A_5)$
in S[1,2] = 1 per forza perché sono solo 2 matrici
=> avrò alla fine $(((A_1)(A_2))(A_3))((A_4)(A_5))$

Lo scorrimento di S non costerà più di n

Altro esempio di programmazione dinamica
Assemblare oggetto, in diversi stadi su 2 catene di montaggio



All'inizio l'oggetto deve scegliere quale catena prendere, ma anche durante può cambiare catena (es: pallini verdi).

Associamo quindi ad ogni stadio S_{1i} = tempo trascorso nello stadio "i" della 1ª catena, (viceversa per la catena 2)

e segniamo come t_{1i} = il tempo di attraversamento dalla catena 1 alla 2 dopo aver eseguito lo stadio "i" sulla catena 1 (viceversa per la catena 2)

e_1 e_2 = tempo di entrata
x_1 x_2 = tempi di uscita

L_{1i} = il nodo che rappresenta lo stadio "i" della catena 1

quindi l'esempio dei pallini verdi sarà: $[\, E\, ,\, L_{11}\, ,\, L_{12}\, ,\, L_{23}\, ,\, L_{14}\, ,\, F\,]$
e il tempo di esecuzione totale sarà: $e_1+S_{11}+S_{12}+t_{12}+S_{23}+t_{23}+S_{14}+x_1$

Il problema quindi è cercare il percorso migliore

definiamo $f_1(1)$ = tempo di esecuzione minimo fino allo stadio "n" dato che lo stadio n è eseguito sulla catena 1 //analogo per $f_2(1)$

chiamo $f^* = min\{\, f_1(n)+x_1\, ,\, f_2(n)+x_2\}$

la mia soluzione finale non può essere composta da soluzioni non ottime

$f_1(n)=min\{\, f_1(n-1)+S_{1n}\, ,\, f_2(n-1)+t_{2,(n-1)}+S_{1n}\}$, non sapendo su che catena ero prima pongo i due casi, il secondo è quello che comprende il costo dello spostamento dalla catena 2 alla 1
//analogo per $f_2(n)=min\{\, f_1(n-1)+t_{1,(n-1)}+S_{2n}\, ,\, f_2(n-1)+S_{2n}\}$

=> → $f_1(1)=e_1+S_{11}\ ;\ f_2(1)=e_2+S_{21}$
→ $f_1(j)=min\{\, f_1(j-1)+S_{1j}\, ,\, f_2(j-1)+t_{2,(j-1)}+S_{1j}\}$ $\quad j\geq 2$
→ $f_2(j)=min\{\, f_1(j-1)+t_{1,(j-1)}+S_{2j}\, ,\, f_2(j-1)+S_{2j}\}$ $\quad j\geq 2$

"Se io proseguissi senza salvarmi nulla … probabilmente esploderei" cit Pinotti

$r_i(j)$ = # di occorrenze del sottoproblema $f_i(j)$
quanto è $r_1(j-1)$? Cioè quante volte viene chiamato il problema $f_1(j-1)$?

$r_1(n)=1 \qquad r_2(n)=1$
$r_1(j)=r_1(j+1)+r_2(j+1)$ //uguale per r_2

Per induzione possiamo dimostrare che $r_i(j)=2^{n-j}$
base dell'induzione: j = n => $2^{n-n}=1$
supponiamo per ip indutt di sapere che $r_1(j+1)=2^{n-(j+1)}$,analogo per $r_2(j+1)$

$r_1(j)=r_1(j+1)+r_2(j+1)=2\cdot 2^{n-j-1}=2^{n-j}$

il che, se volessi risolvere la ricorsione direttamente, mi porterebbe ad un costo di $\Omega(2^n)$

Vediamo di risolvere il problema dal basso verso l'alto
per risolvere $f_1(j)$ abbiamo bisogno di $f_2(j)$

f	1	2		j-1	j		n
1					.		
2							

La matrice F andrà quindi riempita colonna per colonna

la complessità è quindi O(n) perché devo riempire O(n) celle ma con un costo costante perché devo controllare solo 2 valori

Come facciamo a costruire la soluzione? Devo capire dove ho eseguito i vari stadi, creo allora la un altra matrice S (della dimensione precedente) che verrà riempita così:

```
if( f1(j)=f1(j−1)+S1j )
      S[1,j] ← 1
else
      S[1,j] ← 2
```

supponiamo che il minore sia in F[1,n]
=> l'ultimo stadio l'ho fatto in catena 1, vado in S[1,n], in cui ci sarà scritto dove ho fatto lo stadio precedente, e così via fino a tornare al primo stadio

Codice:

```
print_stadio( S , n ){ //codice per costruire la soluzione
      if( f* = f1(n)+x1 )
            i ← 1           //ultimo stadio era in catena 1

      else
            i ← 2           //ultimo stadio era in catena 1
      //print stadio n su catena i
      for j ← n down to 2 {
            i ← S[i,j]
            //print stadio j-1 su catena i
      }
}
```

Altro esempio programmazione dinamica: palindrome
Data una stringa di caratteri voglio sapere quanti caratteri aggiungere per renderla palindrome

quanto è il minimo numero di caratteri da aggiungere ?
Se la mia stringa ha un solo carattere è già palindroma, se ne ha 2 lo è se i caratteri sono uguali, o la rendo palindrome aggiungendo 1 dei due caratteri dalla parte opposta

es: “o” → 0
“os” → “oso” // if $(c_1=c_2)$ => 0 // es “oo”
→ “sos” // else => 1

$f(c_1c_2)$ = → min{ $1+f(c_2), f(c_1)+1$ } se $c_1 \neq c_2$
→ 0 se $c_1=c_2$

consideriamo ora una stringa di lunghezza 3
es: “oso” ← è già palindroma
“oss” ← non lo è => devo sistemare => “o[ss]o” o “s[os]s” e poi devo rendere palindrome la parte centrale

$f(c_1c_2c_3)$ = → 0 (= $f(c_2)$) se $c_1=c_3$
→ min{ $1+f(c_2c_3), 1+f(c_1c_2)$ } se $c_1 \neq c_3$

nel nostro esempio il minimo sarebbe “osso” poiché c_2c_3 è già palindromo
mentre nell'altro caso avrei avuto $s[<\begin{matrix}oso\\sos\end{matrix}>]s$

es: “ast” → a[st]a 1 + f(st)
→ t[as]t 1 + f(as)

non crescono le possibilità di scelta al crescere della stringa, infatti anche con una stringa lunga 4 avrei comunque:

$f(c_1c_2c_3c_4)$ = → $f(c_2c_3)$ se $c_1=c_4$
→ min{ $1+f(c_2c_3c_4), 1+f(c_1c_2c_3)$ } se $c_1 \neq c_4$

$f(c_i \dots c_{i+l-1})$ //stringa f di lunghezza l
la nostra stringa la possiamo rappresentare come un array A[1 … l]

=> f(1 … l) => f[l,1] = f^*

f[l,i] = numero <u>minimo</u> di caratteri da aggiungere per rendere palindrome la sottostringa che inizia in posizione “i” e ha lunghezza “l”

f[l,i] = → f[l-2,i+1] se A[i] = A[i+l-1] //1° e ultimo carattere
→ min{ 1+f[l-1,i+1], 1+f[l-1,i] } //se fallisce il controllo sopra

caso base: f[1,i] = 0
f[2,i] = → 1 se A[i] $\neq$ A[i+1]
→ 2 se A[i] = A[i+1]

es: A = “casee” , creiamo la matrice F

l\i	1	2	3	4	5
1	0	0	0	0	0
2	1	1	1	0	-
3	2	2	1	-	-
4	3	2	-	-	-
5	3	-	-	-	-

F[2,2] => “a” $\neq$ “s” => = 1
F[3,1] => “cas” “c” $\neq$ “s” => min{1 + F[2,2], 1 + F[2,1]} = min{2,2} = 2

=> complessità in tempo $O(l^2)$
complessità in spazio: mi basterebbero solo 3 righe invece che l'intera matrice => $O(l)$

Esercizi lasciati dalla prof.ssa:
1) insieme di interi I

ci chiediamo se esiste $I' \subset I$ t.c. $T' = \sum_{i \in I'} a_i = \lfloor (\frac{\sum_{i \in I} a_i}{2}) \rfloor$

Es:
I = { 1 , 3 ,2 , 4 } T = 10 T' = 5
I' = { 1 ,5 } o { 3 , 2 }

soluzione a pagina 130

2) problema panini di Poldo
ci sono vari panini di diverse dimensioni , una volta scelto un panino si possono prendere solo panini più piccoli o uguali

Dato un insieme di interi cerchiamo quindi la massima sottosequenza non crescente

es: 5 , 7 , 3 , 2 , 8
=> sceglie 5 => può prendere solo 3 e 2 poi
(si può scegliere solo in ordine, quindi se ad es si sceglie 8 che è l'ultimo non si può prendere più nessun panino)

note: bruteforce $\Omega(n^3)$

3) problema del resto
ho vari tipi di monete
$d_k > d_{k-1} > \ldots > d_1 = 1$
voglio formare il resto R con il minimo numero di monete

note: si può fare anche con la progr. Greedy

Algoritmi Greedy

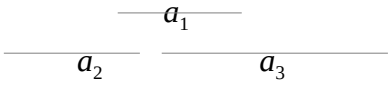
Consideriamo il seguente problema
Ho una stampante (risorsa condivisa) e "n" attività

attività: $a_1 \dots a_n$
ad ogni a_i associamo i valori s_i tempo d'inizio e f_i tempo di fine

2 attività sono compatibili se il tempo di inizio della 2ª è maggiore del tempo di fine della 1ª $(s_j > f_1)$
dati (a_i, a_j) definiamo che: $s_i \leq s_j$

=> il problema è selezionare un sottoinsieme di attività $(I \subset A)$ tale che sono compatibili tra di loro (I compatibile $\forall (i,j) \in I$) e che sia massimo il sottoinsieme (|I| max , $|I| \geq 1$)

poiché devo trovare la cardinalità massima potrei scegliere attività più corte, però questa scelta non porta all'ottimo
es:



$La_1 = f_1 - s_1$
$La_1 < La_2 < La_3$

l = durata

sceglierei a_1 ma sia a_2 che a_3 sarebbero incompatibili perché si intersecano => il massimo sottoinsieme che potrei avere sarebbe I = { a_1 }
ma la soluzione ottima sarebbe invece I' = { a_2 , a_3 }

un altra soluzione potrebbe essere quella di fare partire le attività che partono per prime, le ordiniamo per tempo di inizio minimo e mando la prima

es:

se scelgo a_1 non potrò fare nessun altra attività, e anche in questo caso non avrei quindi la soluzione ottima

=> come potremmo definire il nostro problema top-down ?

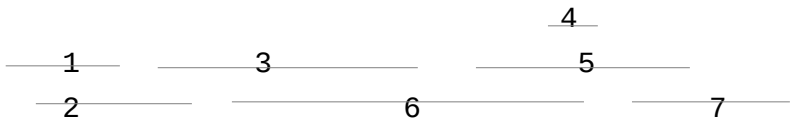
L'attività a_k può o non può appartenere al nostro insieme
se a_k è incluso si escludono tutte le attività a lui intersecate
potrei però combinare tale soluzione con la migliore precedentemente fatta

$1 \leq i \; j \leq 2n$

=> C[i,j] = massimo numero di attività compatibili fra le attività che iniziano al tempo “i” o ad un tempo successivo e finiscano al tempo “j” o precedente
//stiamo quindi considerando tutto ciò che è incluso tra f_1 e s_j

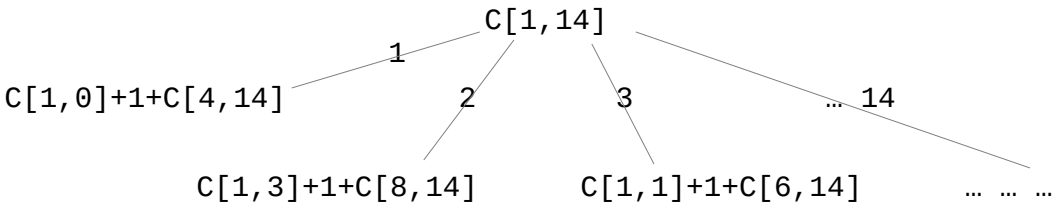
C[i,j] = C[i,r-1] + 1(= a_k) + C[t+1, j]
r = tempo d'inizio dell'attività a_k , t = tempo di fine della stessa
r e t non necessariamente consecutivi

C[i,r] sono tutte le attività compatibili con a_k che quindi iniziano e finiscono prima



si hanno 2n istanti di tempo in cui un attività si può aprire o chiudere (non consideriamo istanti uguali)

nell'esempio sopra gli istanti saranno quindi 14



siamo quindi nell'ipotesi di poter far valere la programmazione dinamica, avrei però una matrice $M^{2n \; x \; 2n}$ triangolare superiore

però dentro agli intervalli non mi serve portarmi dietro tutto ma basta prendere la 1^{a} soluzione compatibile che finisce prima

in C[6,14] posso prendere {4},{7} o {5} o {6},{7}
scelgo 4 perché finisce prima

l'osservazione è C[i,j] esiste soluzione ottima che contiene l'attività a_k tale che $S_k \geq i$ e $f_k \leq j$ e f_k è il minimo tempo di fine fra {i,j} a patto che S_k è maggiore di “i” (come detto prima)

non posso sfruttare direttamente questa proprietà senza sfruttare tutta la programmazione dinamica ?
=> la regola quindi è prendere l'attività compatibile che finisce prima
nel nostro esempio di prima quindi: I = a_1, a_3, a_4, a_7

=> questa è la nostra regola greedy
=> ordino le attività per tempo di fine, quindi l'attività "i" ha tempo di fine f_i che nell'insieme ordinato ha rango "i"

I conterrà quindi sempre l'attività a_1
I = { 1 | j } $s_j \geq f_1; f_j = min\{j\}$

la complessità di questo algoritmo sarà $O(n \log n) + O(n)$ //il primo per l'ordinamento, il secondo per la scansione

Supponiamo di avere un altra soluzione S ottima ottenuta senza utilizzare la regola greedy, vediamo se con scambi successivi possiamo trasformarla in quella greedy.

S → $\in 1$ se S non contiene 1 => conterrà un altra attività che
→ $\notin 1 (\rightarrow \in j)$ avrà f maggiore o uguale a f_2
=> S' = S – j + 1 questa soluzione è ancora ottima e compatibile
=> continuando così la soluzione diventa la greedy

es: G = { a_2, a_4, a_7 } S = { a_3, a_4, a_7 }
S non ha scelto a_2 ma sappiamo che $f_3 > f_2$ => S' ={ a_3, a_4, a_7 } U { a_2 }
ed è sicuramente compatibile S' = G => sono uguali

Altro classico esempio di algoritmo greedy: zaino frazionario

n oggetti caratterizzati da un volume e un profitto

1	2	…	n
v_1	v_2	…	v_n
p_1	p_2	…	p_n

zaino di capacità M

posso prendere un oggetto per una parte del suo volume

es:

	1	2	3
v	3	5	7
p	30	60	21
P u	10	12	3

P u = profitto unitario

=> se prendo $v_1=3 \rightarrow p_1=30$ o $v_1=2 \rightarrow p_1=20$
consideriamo quindi di dividerli in numeri naturali

$1 \leq k_i \leq v_i \quad k, v \in \mathbb{N} \qquad k_i \cdot \frac{p_i}{v_i}$

se $\sum_{i=0}^{n} v_i \geq M$ => posso riempire lo zaino fino all'orlo
la soluzione ottima riempe tutte le M posizioni dello zaino (non scontato nel caso dello zaino normale)

es: M = 90
$v_1=80 \quad p_1=160$
$v_2=15 \quad p_2=10$
$v_3=75 \quad p_3=60$
nello zaino intero posso prendere solo v_1 come soluzione ottima, oppure v_2+v_3 per riempire fino all'orlo, ma non sarebbe la soluzione ottima

ordino per profitto specifico (= $\frac{p_i}{v_i}$) decrescente

$\frac{p_1}{v_1} \geq \frac{p_2}{v_2} \geq \ldots \geq \frac{p_n}{v_n}$

scelta 1 $\in$ zaino => ho riempito $v_1 \cdot \frac{p_1}{v_1} = p_1$
se avessi scelto l'oggetto j (con j > 1)
=> $v_1 \cdot \frac{p_j}{v_j} \leq v_1 \cdot \frac{p_1}{v_1}$
se l'oggetto “1” ci sta tutto => avrò M - v_1
=> sceglierò l'oggetto con profitto specifico maggiore dopo v_1

Codice:

```
capacità_residua ← M
valore_zaino ← 0
while( capacità_residua > 0 ) {
      if( capacità_residua ≥ v_j ){
            valore_zaino ← valore_zaino + p_j
            capacità_residua ← capacità_residua - v_j
            j++
      } else {
            valore_zaino ← valore_zaino + capacità_residua · p_j / v_j
            capacità_residua ← 0
      }
}
```

la regola greedy è: $v_i \cdot \frac{p_i}{v_i} \geq v_i \cdot \frac{p_j}{v_j}$ con j > i

nello zaino intero #oggetti = $\sum_{i=1}^{n} v_i \rightarrow \frac{p_i}{v_i}$

es: v p
1 → 3 9
2 → 2 5
posso considerare 3 oggetti di tipo “1” distinti con profitto uguale a 3 e 2 oggetti di tipo “2” con profitto uguale a 2,5

però la complessità sarà uguale $M\sum_{i=1}^{n} v_i$
mentre nello zaino frazionario è $O(n\log n)+O(n)$

NB: la regola Greedy non vale in generale, ma vale sicuramente se ho ingombri multipli tra di loro

Altro esempio Greedy: problema della compressione → Codici di Huffman

problema: abbiamo un testo e vorremmo utilizzare il numero minimo di bit per scrivere quei caratteri

ASCII → “lossy” → 5 x 8 bit ((#caratteri) · (lunghezza codifica))

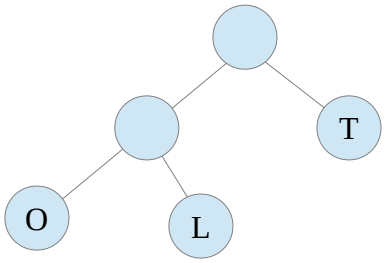
supponiamo che i caratteri siano all'interno di un certo alfabeto
(char $\in$ alfabeto) ed un testo (TEXT) tale che
$\forall\, c\,(char) \in$ TEXT si ha f(c) = #occorrenze di c in TEXT
//f(c) = frequenza

es: con la parola “lossy” avrei: f(l) = f(o) = f(y) = 1 ; f(s) = 2

io so che il numero minimo di caratteri distinti nel testo mi da un lower-bound ($\geq \log_2(\#\, char\ distinti \in TEXT)$)

dati 2 caratteri c_1 e c_2 vogliamo che:
1 – se $f(c_1) > f(c_2)$ voglio che la codifica di c_1 abbia codifica minore o uguale a c_2 ($cod(c_1) \leq cod(c_2)$)

2 – creiamo un albero binario in cui salvarci la nostra stringa, le foglie dell'albero saranno i caratteri, ogni volta che andrò a sinistra metto “0”, se vado a destra “1”



“O” = 00
“L” = 01
“T” = 1

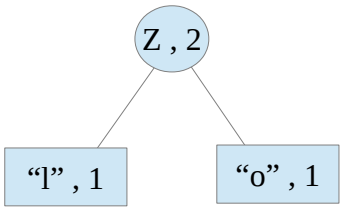
non esiste una codifica che sia prefisso di un altra
es: “W” = 001 e “O” = 00 non è possibile

Inseriamo i caratteri in una coda di priorità

```
Codice:
n ← #char distinti nel testo
Q ← {(c,f(c)) : ∀ c ∈ TEXT }     //Q = min-queue su f(c)
for i ← 1 to n-1 {
      allocate a node z
      z->left ← x ← extract-min(Q)
      z->right ← y ← extract-min(Q)
      f(z) ← f(x) + f(y)
      insert Q ← (z , f(z))
}
```

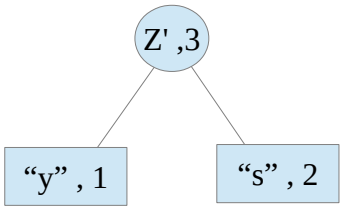
es: “lossy”

1) Q = | (“l”, 1) | (“o”, 1) | (“s”, 2) | (“y”, 1) | ← vettore che scandisco alla ricerca del minimo quindi non ordinato in partenza



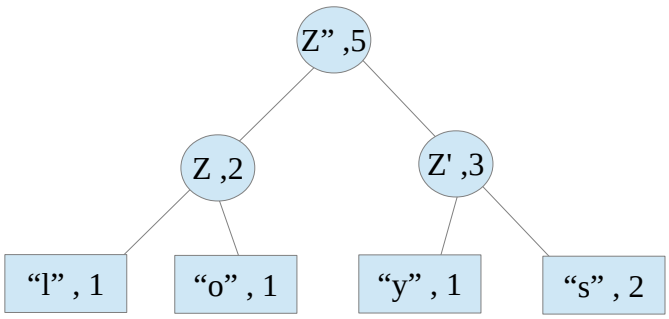
inseriamo Z nella coda => Q = | (Z, 2) | (“s”, 2) | (“y”, 1) |

2)



Q = | (Z, 2) | (Z', 3) |

3)



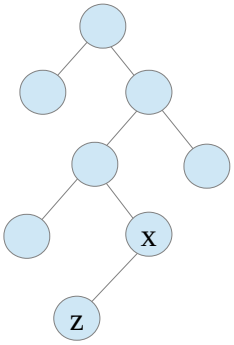
Q = | (Z”, 5) |

codifica =>
“l” = 00 “o” = 01
“y” = 10 “s” = 11
=> il costo in bit è 2 per ogni char

al passaggio “2)” avrei anche potuto prendere Z al posto di “s” poiché entrambi valevano 2, sarebbe stata anche quella una soluzione accettabile, in quel caso la codifica sarebbe stata:
“l” = 110 “o” = 111
“y” = 10 “s” = 0
è diversa ma il costo finale in numero di bit rimane uguale (10)

P = x,y frequenza minima nel testo
cod(x) cod(y) hanno la stessa lunghezza (perché sono 2 foglie figlie dello stesso nodo) e differiscono di un bit soltanto
=> non posso avere una soluzione ottima che non soddisfa il fatto che x,y non sono alla massima profondità

Ci devono essere sempre 2 foglie alla massima profondità (cioè non un unico figlio)



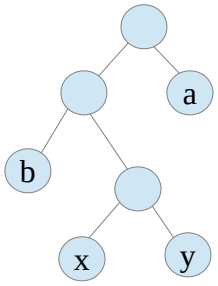
← questa è una situazione che non può esserci, poiché basterebbe cancellare “x” e porci “z”

così da avere gli stessi caratteri (perché solo le foglie hanno caratteri) ma con meno bit, infatti nell'albero sopra cod(z) = 1010 , dopo lo scambio varrebbe 101

---

se avessi un T” ottimo non realizzato con la soluzione sopra descritta, cioè che non rispetta la proprietà che i caratteri a minor frequenza sono a maggior profondità, possiamo dimostrare che il costo (nell'albero con la proprietà) è minore o uguale a un qualsiasi altro

T”



provo a scambiare x e y con a e b
quanto costa la codifica ?

$$Costo_T(TEXT) = \sum_{c \in TEXT} f(c) \cdot cod_T(c)$$
$$= \sum_{c-\{a,b,x,y\}} f(c)\, cod_T + f(a)\, cod_T(a) + f(b)\, cod_T(b) + f(x)\, cod_T(x) + f(y)\, cod_T(y)$$
$$Costo_{T''}(TEXT) = \sum_{c \in TEXT} f(c) \cdot cod_{T''}(c)$$
$$= \sum_{c-\{a,b,x,y\}} f(c)\, cod_{T''} + f(a)\, cod_{T''}(a) + f(b)\, cod_{T''}(b) + f(x)\, cod_{T''}(x) + f(y)\, cod_{T''}(y)$$

$Costo_{T''}(TEXT) - Costo_T(TEXT)$ // $f(c)\, cod_T(c) = f(c)\, cod_{T''}(c)$ => si semplifica
$= f(a)\, cod_{T''}(a) + \ldots + f(y)\, cod_{T''}(y) - f(a)\, cod_T(a) - \ldots - f(y)\, cod_T(y)$

sappiamo che:
$cod_{T''}(a) = cod_T(x)$ $cod_{T''}(b) = cod_T(y)$ $cod_{T''}(x) = cod_T(a)$ $cod_{T''}(y) = cod_T(b)$

=> sostituiamo

$$f(a)\, cod_T(x) + f(b)\, cod_T(y) + f(x)\, cod_T(a) + f(y)\, cod_T(b) - f(a)\, cod_T(a) - \ldots - f(y)\, cod_T(y)$$

metto in evidenza
$$f(a)\,(cod_T(x) - cod_T(a)) + f(x)\,(cod_T(a) - cod_T(x)) + f(b)\,(cod_T(y) - cod_T(b)) + f(y)\,(cod_T(b) - cod_T(y))$$

mi accorgo che ho la stessa quantità a segno diverso => raccolgo il segno
$$f(a)\,(cod_T(x) - cod_T(a)) - f(x)\,(cod_T(x) - cod_T(a)) + f(b)\,(cod_T(y) - cod_T(b)) - f(y)\,(cod_T(y) - cod_T(b))$$

raccolgo
$$(cod_T(x) - cod_T(a))(f(a) - f(x)) \;+\; (cod_T(y) - cod_T(b))(f(b) - f(y)) \;\leq 0$$
$$< 0 \qquad \geq 0 \qquad < 0 \qquad \geq 0$$

=> $Costo_{T''}(TEXT) \leq Costo_T(TEXT)$

Vale la sub-ottimalità, infatti supponiamo un albero come il precedente,
lo chiamiamo T , definiamo un altro albero T' = T – {x,y}
se non vale la sub-ottimalità => esiste T” che costa meno di T'

supponiamo per assurdo che: $Costo_{T'}(TEXT) \geq Costo_{T''}(TEXT)$
T” avrà l – {x,y} U (Z) tale che Z = f(x) + f(y)

=> posso avere $\overline{T}$ cui aggiungo x e y (T = $\overline{T}$) troverò che
Costo di T” è minore di quello di T' allora anche quello di $\overline{T}$ sarà minore

a T, in pratica ho T che è ottimo che è fatto da una soluzione sub-ottima (T') però suppongo che T” = T' ha un costo minore, il punto è che anche per la greedy vale la sub-ottimalità

Grafi

G(V,E) E = $(u,v) \in V$

può essere → diretto $(u,v) \neq (v,u)$

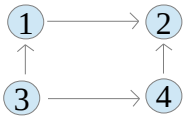
→ non diretto $(u,v) = (v,u)$

funzione peso $w: E \rightarrow R$ o $w: V \rightarrow R$

implementazione:

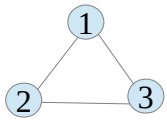
a) con matrice delle adiacenze G = V x V

es:



G	1	2	3	4
1	0	1	0	0
2	0	0	0	0
3	1	0	0	1
4	0	1	0	0

Grafo non orientato



con la matrice delle adiacenze è come avere un grafo orientato e da ogni vertice entra e esce un arco diretto allo stesso vertice

G	1	2	3
1	0	1	1
2	1	0	1
3	1	1	0

Per $w: V \rightarrow R$ dovrei avere un vettore aggiuntivo

con questa rappresentazione funziona bene l'inserzione di archi ($O(1)$), però si ha un alta occupazione di memoria (V^2)

b) M : V x E
matrice in cui ogni colonna rappresenta un arco, le righe i vertici (segno "+1" per arco uscente, "-1" per entrante)

c) lista delle adiacenze
un vettore per i vertici, a cui sono collegate delle liste che indicano i vertici a cui è collegato

l'occupazione di spazio in questo caso è uguale a $\Theta(V+E)$
l'aggiunta di un arco può avere un costo costante (inserimento in testa), ma non sempre si può fare

d) vettore delle adiacenze
ho un vettore per "nodo" (che è il vettore dei vertici) di cardinalità |V|+1 e un vettore "arco" di cardinalità |E|, cioè tante celle quanti sono gli archi

$\forall\, v \in Adj(u)$ [u,arco(nodo(u))] … [u,arco(nodo(v+1))-1]
adj() = adiacenti

questa sarebbe la rappresentazione migliore in costi di spazio e tempo

Algoritmi per la visita di un grafo
1) visita in larghezza (Breadth first search) BFS
2) visita in profondità (Depth first search) DFS

1) G = (V,E) vale sia per grafi orientati che non
Abbiamo un vertice da cui partiamo $s \in V$

utilizzeremo per marcare i nodi dei colori per capire se siamo già passati o meno da un nodo

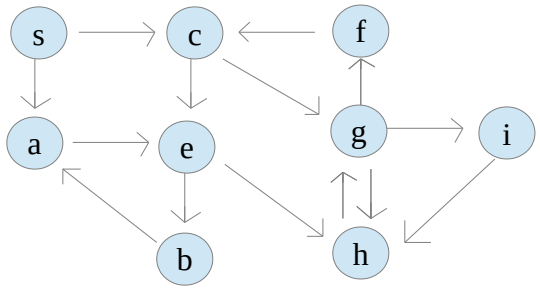
$\forall\, v \in V$ definiamo un colore:
color(v) = → "white" , se non l'ho mai visitato
→ "grey" , ho messo il vertice in coda
→ "black" , se ho finito la visita

sono grafi non pesati allora è come dire che $\exists\, w : E \to 1$, cioè che ha peso costante

La BFS $\forall v$ calcola la distanza da “s” a “v” , d(s,v) o d(v) , perché s è parametro implicito

ed è il minimo numero di archi che devo attraversare per raggiungere “v” da “s”

d(v) = la somma degli archi che attraverso, visto che tutti gli archi hanno peso unitario



d(s,h) = d(h)
abbiamo vari cammini
- p1 = (s,c)(c,g)(g,i)(i,h) l(p1) = 4
- p2 = (s,a)(a,e)(e,h) l(p2) = 3
- p3 = (s,c)(c,g)(g,h) l(p3) = 3

=> d(h) = 3

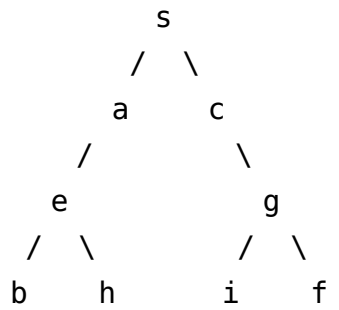
d(s) = 0
facendo una visita si impone un albero di copertura, cioè un sottoinsieme del grafo
per ogni vertice avremo: $\pi : V \rightarrow V$ ← che è il puntatore al padre che quindi ci darà l'albero di copertura, $\pi(s)=nil$ poiché sarà la radice
La prima volta che si esegue il colore di tutti i nodi sarà “white” tranne che color(s) = grey

Codice:

```
BFS( G=(V,E) , s ){
     //inizializzazione
      ∀ v ∈ V−{s} {
          color(v) ← white
          d(v) ← +∞  ;  π(v) ← nil
     }
     d(s) ← 0  ;  π(s) ← nil  ;  color(s) ← grey
     enqueue(Q,s)    //inserimento in coda del vertice s
```

```
      while ( Q ≠ 0 ){
            u ← dequeue(Q)  //estrazione dalla coda
             ∀ v ∈ adj(u) {
                  if( color(v) = white ){
                        //cioè non ho ancora mai visitato questo nodo
                        enqueue(Q , v)
                        d(v) ← d(u) + 1
                         π(v) ← u
                        color(v) ← grey
                  }
            }
            color(u) ← black
      }
}
```

avremo quindi alla fine del codice un albero di copertura T
|T| = |V| - 1



complessità BFS
numero di dequeue = |V|
poi devo visitare gli archi uscenti => |E| = $\sum_{v \in V} adj(v)$
estrazione e inserimento in coda lo consideriamo costo costante
=> O(V + E)

- nel caso si rappresenti il grafo con la matrice delle adiacenze |E|= V^2
- nella lista delle adiacenze invece rimane |E|

in V + E domina sempre E se il grafo è connesso
L'algoritmo BFS ha la caratteristica di calcolare la distanza dalla sorgente
Q è una coda FIFO però se la guardiamo rispetto all'etichetta “d” (distanza) diventa una min-priority-queue

Nel caso in cui il nostro grafo sia non orientato, noi lo consideriamo come un grafo orientato e ogni arco diventa 2 archi, uno uscente e uno entrante per gli stessi nodi, quindi avremo che |E| sarà raddoppiata

In generale nella BFS potrei usare d(v) come marca per capire se un nodo è già stato visitato, poiché in caso sarebbe $+\infty$

La BFS visita solo i nodi raggiungibili dalla sorgente “s” (quindi nel caso di un grafo non connesso avrei una parte di grafo che rimarrebbe non visitata)

2) DFS funziona sia per grafi diretti che non
anche in questo algoritmo utilizziamo color(v) come per la BFS

avremo altre due etichette
d[v] ← tempo in cui si inizia la visita
f[v] ← tempo di fine visita

anche DFS costruisce l'albero di copertura ed usa una struttura dati che è la pila data dalla ricorsione

Codice:

```
DFS( G=(V,E) ){
     //inizializzazione
      ∀ v ∈ V {
          color(v) ← white
          d(v) = f(v) ← +∞
     }
     time ← 0   // 0 ≤ time ≤ 2|V|
      ∀ v ∈ V {
          if( color(v) = white ){
               DFS-visit( G , v )
               π(v) ← nil
          }
     }
}
```

```
DFS-visit ( G=(V,E) , u ){
      color(u) ← grey
      time++
      d(u) ← time
        ∀ v ∈ adj(u)  {
            if( color(v) =  white ){
                  π(v) ← u
                DFS-visit( G , v )
            }
      }
      color(u) ← black
      time++
      f(u) ← time
}
```

$|V| = V_1 \cup V_2$ |V| = # alberi //cioè numero di alberi che estraggo

quando il grafo è non-diretto un albero è l'insieme dei vertici raggiungibili dalla sua radice e viceversa
L'albero trovato coincide con la componente connessa

in un grafo diretto non vale il discorso della simmetria quindi non si può fare lo stesso ragionamento

in base ai valori d(u) e f(u) possiamo catalogare gli archi

gli archi possono essere classificati in “backward” se vanno da un nodo verso un suo antenato nell'albero
in “forward” se vanno verso un suo discendente
in ”cross” questi possono essere anche archi tra vari alberi, o vertici allo stesso livello

il numero di alberi che si formano dipende da quale vertice si inizia l'algoritmo

gli archi “cross” hanno intervalli d( ) f( ) distinti , non vale lo stesso per gli altri due tipi di archi

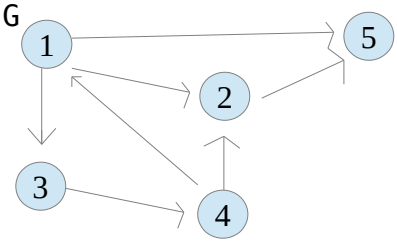
In un grafo diretto con la procedura DFS possiamo catalogare vari tipi di arco
Tutti gli archi che estraggo con la procedura sono in T
u → v $(u,v) \in adj(u)$
se color(v) = white => $(u,v) \in T$

Poi a seguito avrò 3 tipologie di arco:
- Forward (F)
u → v color(v) = black
$d(u)<d(v)<f(v)$

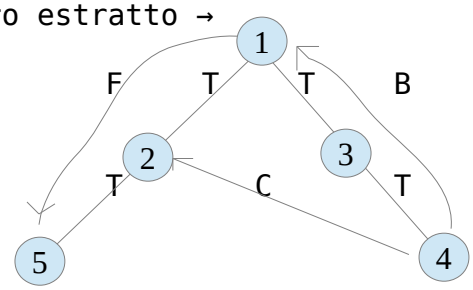
- Backward (B)
u → v color(v) = grey => $(u,v) \in B$

- Cross (C)
u → v color(v) = black
$d(v)<f(v)<d(u)$

Es:
G



Albero estratto →



F: (1,5) → color(5) = black => $f(s) \neq +\infty \quad d(1)<d(5)<f(5)$
C: (4,2) → color(2) = black => $d(2)<f(2)<d(4)$
B: (4,1) → color(1) = grey

un grafo G è <u>ciclico</u> (cioè contiene almeno un ciclo) se e solo se contiene un arco di tipo B (es: Vertici 1 ,3, 4)

Consideriamo ora un grafo non diretto (undirected)
DFS-Visit(G , $v \in V$), ti permette di visitare tutti i vertici “u” per cui esiste un cammino da “v” a “u”, esiste un cammino u → v allora restituisce la componente connessa in G a cui appartiene v

quindi se si hanno più componenti connesse bisogna chiamare DFS-visit tante volte quante sono le componenti

tenendo un contatore possiamo quindi anche contare il numero di queste, posso creare un vettore in cui segnarmi (tramite il contatore) quali nodi appartengono a quale componente, in questo caso dovrò avere DFS-visit(G, i, cc) al cui interno avrò “nome-cc ← cc”
(nome-cc = vettore di lunghezza |V|).

Es: consideriamo un grafo di 8 vertici con 2 componenti connesse entrambe da 4 vertici ciascuna allora avremo:
nome-cc | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |

I tipi di arco nel grafo non orientato sono 2:
- archi appartenenti a T
- archi di tipo backward

Tutte le volte che trovo un nodo bianco questo sarà un arco in T

Codice:
```
DFS-visit ( G ,v, padre ){
      color(v) ← grey
      time++
      d(v) ← time
       ∀ u ∈ adj(v)  {
            if( color(u) = white ){
                  π(u) ← v
                  DFS-visit ( G, u, v )
            } else {
                  if( color(u) = grey  AND  u ≠ padre )
                        (v,u) ∈ B
            }
      }
      time++
      f(v) ← time
      color(v) ← black
}
```

L'arco di tipo forward non esiste perché sarebbe l'arco verso un vertice nero, ma se arrivo a questo significa che ho già classificato tutti gli archi
Rimane il fatto che esiste almeno un backward c'è almeno un ciclo

In entrambi i casi (grafi diretti e non) il numero di archi backward non indica il numero di cicli

Grafi Bipartiti
$V_1 \cup V_2 = V \quad V_1 \cap V_2 = \emptyset \quad \forall\, c=(u,v) \leftarrow E \quad u \in V_1 \; v \in V_2$
in pratica l'insieme dei vertici è diviso in due gruppi distinti, ogni arco può esistere solo se unisce 2 vertici di gruppi differenti

un albero è un grafo bipartito

Un grafo è bipartito se e solo se non ha cicli di lunghezza dispari
Un grafo è bipartito se e solo se è 2-colorabile

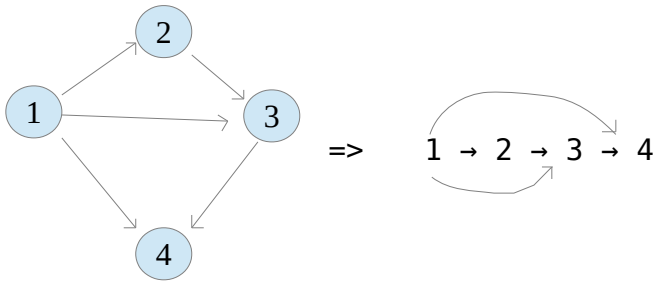
```
BFS-color( G=(V,E) , s ∈ V  ){
      //inizializzazione
       ∀ v ∈ V
            d(v) ← +∞
      d(s) ← 0
      insert-queue( Q, S ); color(s) ← grey; parte(s) ← 1
      while ( Q≥0 ) {
            u ← extract-min(Q)
```

```
            ∀ v ∈ adj(u)  {
                if( color(v) = white ){
                      color(v) ← grey;        π(v) ← u
                      d(v) ← d(u) +1   insert-queue( Q, v )
                      parte(v) ←( parte(u) + 1 )mod 2
                } else {
                      if( parte(v) = parte(u) )
                            // => G non è bipartito
                }
            }
      }
}
```

Sort-Topologico // o bilinearizzazione di un grafo
si applica a grafi diretti e aciclici (DAG = Directed Acyclic Graph)

Linearizzare significa fare in modo tale che tutti gli archi vadano da sinistra a destra

es:



In generale non esiste un unica linearizzazione
Non è possibile linearizzare un ciclo, poiché si avrebbe sempre un arco che torna verso sinistra

Osserviamo che il vertice più alla destra è un <u>pozzo</u> (= vertice che ha solo archi entranti)

Per costruire la soluzione ci basiamo sulla DFS e quando un nodo diventa black lo mettiamo in una lista, inserendo in testa, alla fine si avrà la nostra linearizzazione
“insert-head(v, S)” con “S” puntatore alla lista e “v” vertice da inserire

Essendo un DAG si avranno solo archi di tipo T, F, C; poiché non avendo cicli non si possono avere archi di tipo B

Il costo dell'algoritmo sarà quello di una DFS

Componenti fortemente connesse
in un grafo diretto vogliamo trovare un insieme di nodi tale che se esiste un cammino da “u” a “v” deve esserci il viceversa

Calcoliamo la DFS(G) poi calcoliamo G^T (cioè se $\exists (u,v) \in G \rightarrow (v,u) \in G^T$, quindi tutti gli archi non presenti in G saranno presenti nel suo trasposto, possiamo farlo in tempo V+E se stiamo lavorando con una lista delle adiacenze)

poi si fa la DFS(G^T) , se raggiungiamo tutti i nodi allora G è un grafo fortemente connesso

Passiamo ora a lavorare su **grafi pesati**
$w : E \rightarrow \mathbb{R}$
il fatto di avere un peso ci introduce un discorso di ottimizzazione

Dato G, non diretto e connesso (un unica componente connessa) vogliamo estrarre un albero di copertura la cui somma degli archi sia minima
MST → Minimum Spanning Tree

per risolvere questo problema applicheremo una soluzione di tipo Greedy

|T| = |V| - 1 ← numero di archi da estrarre

min{ $\sum_{e \in T} w(e)$ } // e = archi dell'albero

Metodo di soluzione (non da i dettagli ma da delle regole)
- Regola BLUE
  considera un taglio e colora di blue l'arco di costo minimo
  (partizione dei vertici e si guarda gli archi che hanno gli estremi nelle 2 partizioni)
  regola greedy per l'inclusione
- Regola RED
  considera un ciclo nel grafo e colora di rosso l'arco di costo massimo
  regola greedy per l'esclusione

questo è un metodo non deterministico

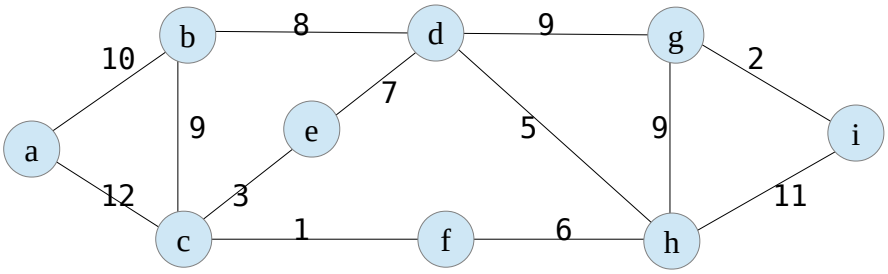
Quanto tutti gli archi sono colorati l'insieme degli archi blue formano l'albero MST

supponiamo di non aver colorato un arco
u → v
- o "u" e "v" appartengono già all'albero => è un ciclo => (u,v) verrà colorato di rosso
- o non sono connessi => c'è sicuramente un taglio => applicherò la regola blue

supponiamo di aumentare il peso
$\forall\, e \in G \;\; w(e)+\Delta = \hat{w}(e)$ => $\forall\, (spanning\, tree)\; T \rightarrow \hat{w}(T)=w(t)+(|V|-1)\Delta$
$w(T)=\sum_{e \in T} w(e)$
$\sum_{e \in T} \hat{w}(e)=\sum_{e \in T}(w(e)+\Delta)=\sum_{e \in T} w(e)+\sum_{e \in T}\Delta=\sum_{e \in T} w(e)+(|V|-1)\Delta$
se vale per ogni albero vale anche per il MST, quindi non devo cercare un nuovo albero

Quindi non ha importanza se ho pesi negativi o positivi nel mio grafo di partenza poiché basterebbe sommare a tutti un valore tale che porti tutto a pesi positivi e avrei comunque l'MST giusto

Trasformiamo ora il metodo in un algoritmo
E' = ordina gli archi in senso non decrescente
$\forall\ e=(u,v) \in E'$
colora(e) = blue se $u,v \neq T'$ //se puoi applicare la regola blue, cioè //se non c'è ancora nessun cammino che mi // lega “u” con “v”
colora(e) = red se $u,v \in T'$



E' = (c,f) , (g,i) , (c,e) , (e,f) , (d,h) , (f,h) , (e,d) , (b,d) , (d,g) , (c,b) , (g,h) , (a,b) , (h,i) , (a,c)

partendo da (c,f) controllo quindi se appartiene o meno a T, in caso negativo lo coloro di blue in caso contrario di rosso

Per sapere se u,v appartengono allo stesso albero utilizzeremo un array indicizzato con i nomi dei vertici, poi per fare il controllo leggeremo l'etichetta di “u” e “v”, se sono uguali sono nello stesso albero

L'inizializzazione di questo array considera come se avessimo una foresta in cui ogni vertice è un albero a se, alla fine avremo che tutte le etichette avranno lo stesso valore

es:
(c,f) avranno rispettivamente 3 e 6
find(c) != find(f) => T ← T ∪ {e}; union(u,v); //considerando T inizializzato ad albero vuoto, e l'arco che si sta controllando e “find()” la funzione che legge l'etichetta nell'array
union ← significa prendere la minima etichetta (per convenzione) e riscrivila nell'altra etichetta del vertice e in tutte le etichette che erano uguali
//nel nostro esempio (c,f) quindi dovrei cercare tutte le etichette con 6 e scriverci 3

Codice:
//inizializzazione
$\forall\ v \in V$
label(v) ← v //setto l'etichetta
T ← 0

$\forall\ e=(u,v) \in E'$ {
if(find(u) != find(v)){
T ← T ∪ {e}
union(u,v);
}

```
      else
            color(e) ← red
}
```

Costi:
find() → O(1)
union() → O(|V|)

Codice union:

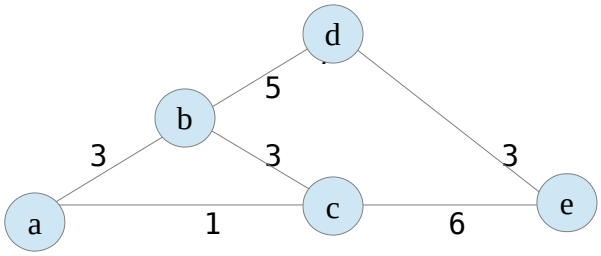
```
      e1 ← find(u)
      e2 ← find(v)
      lmin ← min(e1, e2);   lmax ← max(e1, e2)
      for i ← 1 to |V|
            if( label(i) = lmax )
                  label(i) ← lmin
```

a differenza del tipo di struttura dati che si utilizza questo algoritmo avrò costi diversi

implementazione diretta dell'algoritmo di Kruskal
$T_{MST}(G)=|E|\log|E| + 2|E|\, T_{FIND} + (|V|-1)T_{\text{UNION}}$

usando la rappresentazione degli insiemi allora il nostro costo diventa
$|E|\log|V| + 2|E| + (|V|^2)$ ($|E|<|V|^2 \rightarrow \log|E|=2\log|V|$)

Esiste una versione di Kruskal che si chiama reverse-Kruskal che ordina in senso non crescente e considera il grafo un albero, toglie poi gli archi massimi (senza però isolare i vertici)



Algoritmo di Prim (Prim-Algorithm per MST)
utilizza la regola blue (se ho un taglio inserisco l'arco di costo minimo)
Questo algoritmo estende un albero a partire da un vertice scelto

Es:
x = { a } y = { b , c , d , e } cut = { (a,b) , (a,c) }
inserisco allora nell'albero l'arco (a,c), perché di costo minore

=> x = { a , c } y = { b , d , e } cut = { (a,b) , (c,b) , (c,e) }
(a,b) e (b,c) hanno stesso peso, prendiamo (a,b) e notiamo che:
a differenza della scelta avremmo avuto MST diversi, però possiamo anche dire che a differenza della radice che si prende l'MST non cambierà

=> x = { a , c , b } y = { d , e } cut = { (b,d) , (c,e) }
prendiamo (c,e)

=> x = { a , c , b. e } y = { d } cut = { (b,d) , (d,e) }
prendiamo (d,e)
e abbiamo così finito il nostro MST

L'arco di costo massimo è sempre escluso dall'MST ?
Ovviamente no, se l'arco fosse l'unico ponte che scollega un sottografo non potremmo fare a meno di prenderlo

Per il nostro algoritmo dovremo per ogni vertice ricordarci il padre ed avere un etichetta d(v) che sarà il costo dell'arco per arrivare a quel vertice

Codice:

```
Prim ( G=(V,E), r ∈ V ){
      ∀ v ∈ V−{r} {
                  π(u) ← nil
                 d(u) ← +∞
     }
      π(r) ← nil;    d(r) ← −∞
     Q ← V      //coda inizializzata con tutti i vertici
     while ( Q ≠ 0 ){
          u ← extract-min(Q)   //rispetto all'etichetta, quindi la prima
                               //volta sarà r
           ∀ v ∈ Adj(u) AND v ∈ Q {
                relax( u , v , w(u,v) )
          }
     }
}

relax( u , v , w(u,v) ){
     if ( d(v) > w(u,v) ){
          d(v) ← w(u,v)
           π(v) ← u
     }
}
```

La nostra pila sarà un vettore indicizzato con il nome dei vertici

	a	b	c	d	e
Q	−∞	+∞	+∞	+∞	+∞

La prima relax sarà (a , b , w(a,b) = 3) => +∞ > 3 => d(b) = 3

Complessità:

$T_{PRIM} = O(|V|) + T_Q + |V|\, T_{EX-MIN} + |E|\, T_{RELAX}$

con $O(|V|)$ per l'inizializzazione e T_Q uguale al costo per inizializzare la coda
A seconda di come è implementata la coda avrò costi diversi, nel nostro caso con Q array avrò:

$T_{PRIM} = O(|V|) + O(|V|^2) + O(|E|)\, O(1) = O(|V|^2 + |E|) = O(|V|^2)$

Possiamo invece rappresentare il nostro Q come uno HEAP binario

$=> \; T = O(|V|) + O(|V|) + |V|\, O(\log|V|) + |E|\, \log|V| = O(|V|) + (\,|V| + |E|\,)\log|V|$

$= O(\,(\,|V| + |E|\,)\, \log\,|V|\,)$

con T_{RELAX} che diventa la funzione decrease-key dello heap
Per questa rappresentazione si utilizza anche un vettore d'appoggio in cui salvarsi dove si trovano le etichette nello heap

Da notare però che se si ha un grafo denso si avrà $O(|V|^2 \log|V|)$ quindi un costo maggiore anche all'algoritmo che utilizza una coda-vettore
=> finchè $|E| < \frac{|V|^2}{\log|V|}$ $\quad T_{PRIM-HEAP} < T_{PRIM-ARRAY}$

Se avessimo uno heap p-ario
$T = O(|V|) + O(|V|) + |V|\, O(p\, \log_p|V|) + |E| O(\log_p|V|)$
se $p = 2 + \frac{|E|}{|V|}$
=> poniamo $m=|E|$ e $n=|V|$
avremo che $T = O(n) + n\,(2 + \frac{m}{n})\log_{2+\frac{m}{n}} n + n\log_{2+\frac{m}{n}} n$

$= O(n) + O(m\, \log_{2+\frac{m}{n}} n)$

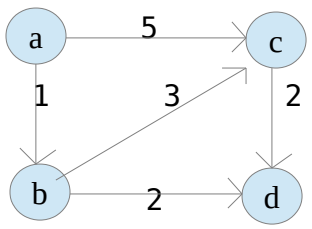
scelgo m perché è sicuramente in ordine di grandezza più grande di n

se $m=\Omega(n^{1+\epsilon})$ → con $\frac{m}{n}$ → $\frac{n^{1+\epsilon}}{n}$ → $n^{1+\epsilon-1} = n^{\epsilon}$ avrò che:
$T = O(m \log_{2+n^{\epsilon}} n) = O(m\, \frac{\log_2 n}{\log_2(2+n^{\epsilon})}) \le c \cdot m\, \frac{\log_2 n}{\log_2 n^{\epsilon}} \le \frac{c\, m}{\epsilon}$

es: $m=n^{\frac{3}{2}}=n^{1+\frac{1}{2}}$ → $p=2+\frac{m}{n}=2+\sqrt{n}$

Si è trovata una struttura ad hoc basata su lo Heap di Fibonacci (che a sua volta si basa sui heap binomiali), con questa struttura
$T_{relax} \in O(1)$ $\quad T_{EX-MIN} \in O(\log n)$
=> $T = |V|\log|V| + |E| \quad \forall\, m \ge n$

Cammini di costo minimo da sorgente singola
Grafo diretto $w : E \to \mathbb{R}$
Sorgente è un vertice $s \in V$
cammino p da "s" a "t" ((s,a) . . . (u,t))
$c(p) = \sum_{e \in p} w(e)$



sorgente = a
$\forall\, v \in V$ a → v (cammino costo minimo)

Es:
a → c 1) (a,c) => 5
2) (a,b),(b,c) => 4

a → b 1) (a,b) => 1

a → d 1) (a,b),(b,d),(c,d) => 6
2) (a,b),(b,d) => 3
3) (a,c),(c,d) => 7

Cammino composto da sottocammini che sono a loro volta cammini ottimi => sub-ottimalità
=> programmazione dinamica

Se i pesi sono positivi posso risolvere il problema anche con un algoritmo greedy

Quando ho pesi negativi posso avere cicli di costo negativo quindi può non esistere il cammino minimo

Quindi si avranno 3 casi:
1 – Se esistono cicli di costo negativo => non esiste soluzione
2 - $w : E \rightarrow \mathbb{R}$ => risolvo con algoritmo di Bellman Ford
3 - $w : E \rightarrow \mathbb{R}^{+} \cup \{0\}$ => algoritmo greedy (Dijkstra)

Codice:

```
Dijkstra ( G=(V,E), r ∈ V ){
      ∀ u ∈ V−{r} {
            π(u) ← nil
            d(u) ← +∞
      }
      π(r) ← nil;     d(r) ←0
      Q ← V      //coda inizializzata con tutti i vertici
      while ( Q ≠ 0 ){
            u ← extract-min(Q)    //rispetto all'etichetta, quindi la prima
                                  //volta sarà r
            ∀ v ∈ Adj(u) AND v ∈ Q {
                  relax( u , v , w(u,v) )
            }
      }
}

relax( u , v , w(u,v) ){
      if ( d(v) > d(u) + w(u,v) ){
            d(v) ← d(u) + w(u,v)
            π(v) ← u
      }
}
```

Per il costo di questo algoritmo vale tutto ciò che è stato detto per l'algoritmo di Prim

Quando estraggo avrò $d(v) = \delta(s,v)$
Sono sempre garantito che trovo un albero di copertura poiché per la sub-ottimalità non è possibile che ci siano cicli => quello che si estrae è un albero di copertura

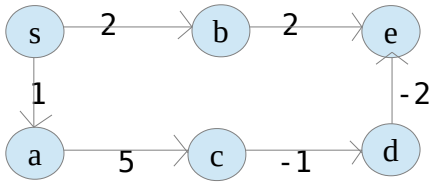
Correttezza (dim per induzione):
vogliamo dimostrare che per ogni vertice l'etichetta è quella con costo minimo

base: $d(s)=0 \quad \delta(s,s)=0 \quad \delta$ ← cammino minimo
supponiamo che $\exists u : d(u) \neq \delta(s,u)$ //primo vertice che non soddisfa la // proprietà, quindi per cui l'algoritmo fallisce
sappiamo che $d(u) \geq \delta(s,u)$

Supponiamo che esista un altro cammino (quello con costo minimo) che vada da "s" a "u", alcuni dei vertici facenti parte di questo cammino li ho già estratti => soddisfacevano la proprietà

supponiamo ora di avere $x \in$ p(s → u) con $d(u)=\delta(s,x)$
=> avrò un $y \in Adj(x)$ con l'arco (x,y) "rilassato"
=> ho messo in coda $d(y)\,[=\delta(s,y)\,] \in Q$
i costi degli archi sono non negativi => $l(y,u) \geq 0$ // $l(\,)$ ← lunghezza
=>
→ $d(u) \leq d(y)$
→ $d(u) \geq \delta(s,u) = \delta(s,y) + l(y,u) \geq \delta(s,y) = d(y)$
//(d(u) maggiore del cammino poiché è la prima volta che il cammino fallisce)
=> $d(u)=d(y)$ => l'algoritmo estrae sempre il costo ottimo

Controesempio con costi negativi in cui l'algoritmo fallisce



sorgente = s

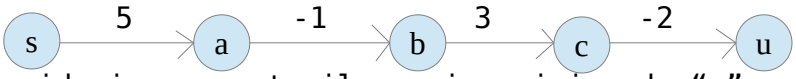
s → $d(s)=0=\delta(s,s)$
$d(a)=1 \quad d(b)=2$
$\pi(a)=s \quad \pi(b)=s$
estraiamo "a"
$d(c)=6 \quad \pi(c)=a$
estraiamo b
$d(e)=4 \quad \pi(e)=b$
estraiamo e
$d(d)=5 \quad \pi(d)=c$

visitando l'arco avrei un costo minore, ma avendo già estratto "e" non me ne posso accorgere

Consideriamo il caso in cui il costo degli archi sia costante ($w : E \rightarrow c \quad c>0$) Dijkstra funziona, però diventa uguale all'algoritmo BFS, ma costando quest'ultimo (BFS) di meno sarà quello conveniente da utilizzare

C'è un solo caso in cui Dijkstra funziona con archi negativi, cioè solo quando gli archi con peso negativo sono archi uscenti dalla sorgente, questo perché sono i primi archi che rilasso allora vengono estratti subito come minimo non creando errori nel controlli successivi

Bellman-Ford $O(|V||E|)$



consideriamo questo il cammino minimo da “s” a “u”, se rilasso gli archi avrò quindi il costo di tale cammino [In generale però io non so quale sia il cammino minimo]

L'osservazione è: si prendono tutti gli archi e si definisce un ordine
E = lista ordinata di archi// ordine qualsiasi
e poi rilasso tutti gli archi nella lista

Codice:
//inizializzazione
$\forall v - \{s\}$ {
d(v) ← +∞ ; $\pi(v)$ ← nil
}
d(s) ← 0; $\pi(s)$ ← nil

for j ← 1 to $|V|$
for i ← 1 to $|E|$
relax($e_i = (u_i, v_i), w(u_i, v_i)$)

Es:
E = (s,a) , (c,u) , (a,b) , (b,c)
- d(a) ← 5
- d(u) ← d(c) + d(u) = +∞ − 2 = +∞
- d(b) ← 4
- d(c) ← 7

riparto quindi con il ciclo (rispetto a j) e questa volta avrò d(u) ← 5

Cambiamo l'ordine degli archi
E = (c,u) , (a,b) , (b,c) , (s,a)
nel primo giro farò soltanto: d(a) ← 5

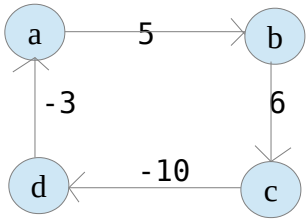
notiamo quindi che gli archi del cammino minimo vengono fatti in ordine

come struttura dati conviene tenere un semplice array che costi O(1) in accesso visto che non devo fare l'extract

per ogni vertice “v” il numero di archi sul cammino minimo da “s” a “v” è al più $|V|-1$, quindi possiamo cambiare il primo ciclo facendogli fare un iterazione in meno

notiamo che è inutile ordinare la lista di archi poiché aumenterei il costo anziché diminuirlo

Vediamo ora il caso in cui c'è possibilità che ci siano cicli di costo negativo



sorgente = c

E = (c,d) , (d,a) , (a,b) , (b,c)
Se togliamo (b,c) ho un solo cammino che è anche l'ottimo, ma se ho questo arco come faccio ad accorgermi che esiste un ciclo ?

Notiamo che un cammino di lunghezza n contiene sicuramente un ciclo

Se facendo l' n-esima iterata c'è un nodo in cui fare relax => c'è un ciclo ed è di costo negativo

Codice:

```
// la prima parte del codice è quella di Bellman-Ford
ciclonegativo ← false
while( i ≤ |E| AND !ciclonegativo){
      if( d(v_i) < d(u_i)+w(u_i,v_i) )
            ciclonegativo ← true
      i++
}
return ciclonegativo
```

Con un costo uguale a Bellman-Ford

E' possibile trovare un ciclo anche prima dell'ultima iterata, si aspetta la fine per controllare perché si è sicuri che in quel caso se c'è lo trovo

Supponiamo di risolvere il problema di cammini minimi da sorgente singola su un DAG pesato
Useremo Bellman-Ford per risolvere questo problema

Consideriamo che esiste un “ordinamento” degli archi per cui la complessità diminuisce
Infatti se si fa prima un sort-topologico sul grafo, si avrà che con una sola iterazione si riusciranno a visitare tutti gli archi correttamente.
Quindi togliendo il primo for il costo viene $O(|E|)$

Cammini minimi tra tutte le coppie

G = (V , E) pesato

$\forall u, v \in V \times V$

$$\delta(u,v) = \sum_{e \in p^*} w(e)$$

Vari metodi:
1 – Metodo analogo alla moltiplicazione fra matrici (prodotto tropicale)
2 – Algoritmo Floyd-Warshall
3 – Algoritmo Johnson

1 -
In questo algoritmo vale la sub-ottimalità
quindi, se conosco il cammino ottimo da "u" a "v" e conosco il padre di "v"

$p^*(u,v) : u \rightarrow \ldots \rightarrow \pi(v) \rightarrow v$

=> so che $p^*(u,\pi(v))$ è ottimo

$p^*(u,v) : p^*(u,*) + p^*(*,v)$

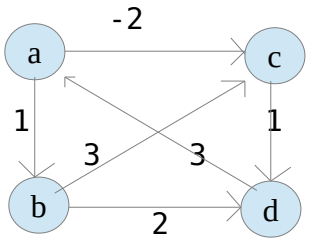
Io non conosco però i vertici esatti per cui devo passare quindi considero qualsiasi possibile vertice del grafo, e pongo gli archi che non esistono a $+\infty$

$$\delta(u,v) = min\{\ \delta(u,j) + w(j,v)\ \} \quad 1 \le j \le n$$

Utilizzeremo una matrice dei pesi così composta:

$W : V \times V \rightarrow \mathbb{R}$ =
- $\rightarrow 0$ se $u = v$
- $\rightarrow w(u,v)$ se $u,v \in E$
- $\rightarrow +\infty$ se $u,v \notin E$

Es:



W	a	b	c	d
a	0	1	-2	$+\infty$
b	$+\infty$	0	3	2
c	$+\infty$	$+\infty$	0	1
d	3	$+\infty$	$+\infty$	0

$\delta^{(1)}(u,v) = W =$ Costo cammino minimo da "u" a "v" attraversando al più un arco

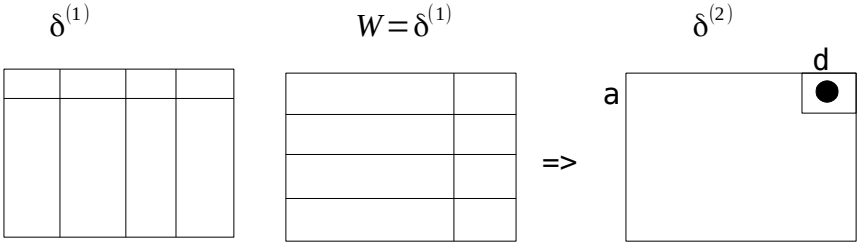
=> $\delta^{(2)}(u,v) = min\{\ \delta^{(1)}(u,j) + w(j,v)\ \}$

=> $\delta^{(l)}(u,v) = min\{\ \delta^{(l-1)}(u,j) + w(j,v)\ \}$

Es:
$a \le j \le d \quad \delta^{(2)}(a,d) = min\{\ \delta^{(1)}(a,j) + w(j,d)\ \} = min\{\ 0+\infty\ ;\ 1+2\ ;\ -2+1\ ;\ \infty+0\ \}$
con j=a e j=d è come se stessi considerando un solo arco, invece con j=b e j=c ne consideriamo 2
$\delta^{(2)}(a,d) = -1$ quello con j=c

mi interessa anche ricordarmi $\pi^{(2)}(a,d) = c$ => a → c → d

Riguardiamo l'operazione a livello matriciale



come 1ª operazione abbiamo la somma di $\delta^{(1)}(a,a) + w(a,d)$
quindi notiamo che lavoriamo sulla riga "a" di $\delta^{(1)}$ e sulla colonna "d" di W

$\delta^{(l)} = \delta^{(l-1)} \odot W$ prodotto tropicale

$\delta^{(2)}$	a	b	c	d
a	0	1	-2	-1
b	5	0	3	2
c	4	+∞	0	1
d	3	4	1	0

L'ordine con cui si riempe la matrice non ha importanza

Se compare un numero diverso da 0 nella diagonale so che avrò un ciclo di costo negativo e ciò invalida tutto l'algoritmo

Serve anche una matrice dei padri che mi ricorda l'indice del padre o "nil" se ho +∞

$\delta^{(3)}$	a	b	c	d
a	0	1	-2	-1
b	5	0	3	2
c	4	5	0	1
d	3	4	1	0

Per un grafo qualsiasi mi fermerò a $|V| - 1$ passate
#passate = n- 1 //se il costo degli archi è $w : E \to \mathbb{R}^+$ se invece ci sono anche costi negativi devo farne n

Oppure la prima volta che una matrice non si modifica ci possiamo fermare, ma il costo in complessità non cambia

Se siamo su un DAG la diagonale rimarrà sempre 0 quindi posso fermarmi a $n-1$

Complessità in tempo:
$n(n^2 \cdot n)$ → n^4 di cui il primo n è perché devo costruire tante matrici quanti sono i vertici

Ricostruire la soluzione è al massimo n-1, cioè ordine della lunghezza del cammino

da $O(n^4)$ si può scendere a $O(n^3 \log n)$
perché supponiamo:
$\delta^{(4)}(u,v) = min\{\ \delta^{(2)}(u,j) + \delta^{(2)}(j,v)\ \}$
$\delta^{(2)}(u,v) = min\{\ \delta^{(1)}(u,j) + \delta^{(1)}(j,v)\ \} = W \cdot W$
quindi se consideriamo $n = |V| = 2^k$ => in k passi vado da W a $\delta^{(k)}$ => $\log_2 n$

Se ho $n \neq 2^k$ posso comunque dire che $2^{t-1} < n \leq 2^t$

es: se ho n = 6
$\delta^{(4)} \cdot \delta^{(4)}$ vabene comunque perché non possono esistere cammini minimi di lunghezza più di 5
=> $\lceil \log_2 n \rceil$

Complessità in spazio:
$2n(n^2) \in O(n^3)$ se devo solo segnare il costo del cammino e non ricostruire la soluzione mi bastano 3 matrici=> $O(n^2)$

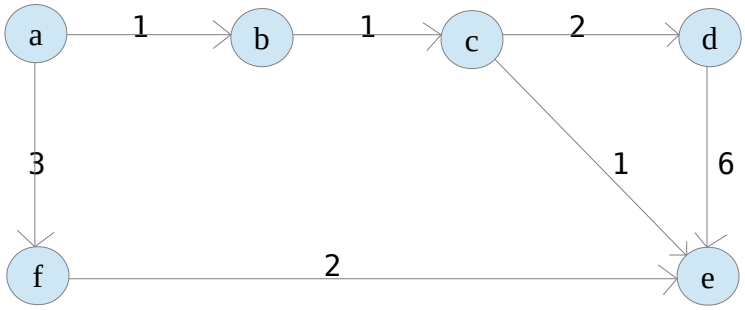
Potenza $G^2 = G\ x\ G = (v, E')$ $\qquad E' = \{\ (u,v) : d_G(u,v) \leq 2\ \}$
$G^t = G^{t-1}\ x\ G = (V, E^{(t)}) = \{\ (u,v) : d_G(u,v) \leq t\ \}$

Questo algoritmo modificato può essere utilizzato per fare la potenza di un grafo usando “OR” invece di “min”

$E^{(l)}(u,v) = \{\ E^{(l-1)}(u,v)\ \text{OR}\ (\ E^{(l-1)}(u,j)\ \text{AND}\ W(j,v)\ )\ \}$
con W = matrice delle adiacenze W = V x V con $a \in \{\ 0,1\ \}$ elemento di W
$= \{\ E^{(l-1)}(u,j)\ \text{AND}\ W(j,v)\ \}$ con $W(j,j) = 1$

$\delta^{(2^t)}(u,v) = min\{\ \delta^{(2^{t-1})}(u,j) + \delta^{(2^{t-1})}(j,v)\ \}$
$\pi(2^t) = arg\{\ min\ \delta(2^t)(u,v)\ \}$
Le matrici dei padri le devo andare a leggere 2 volte per poter ricostruire la soluzione

es:



da “a” ad “e” ho a → b → c → e con costo 3
in $\delta^{(4)}$ troverò già questo cammino
$\delta^{(4)}(a,e)=min\{\ \delta^{(2)}(a,j)+\delta^{(2)}(j,e)\ \}\quad a \le j \le f$

j = a => $(\delta^{(2)}(a,a)=)\ 0+5\ (=\delta^{(2)}(a,e))$
perché con 2 passi posso fare solo a → f → e

j = b => $(\delta^{(2)}(a,b)=)\ 1+2\ (=\delta^{(2)}(b,e))$

j = c => $(\delta^{(2)}(a,c)=)\ 2+1\ (=\delta^{(2)}(c,e))$

j = d => $(\delta^{(2)}(a,d)=)\ +\infty+6\ (=\delta^{(2)}(d,e))$

j = e => $(\delta^{(2)}(a,e)=)\ 5+0\ (=\delta^{(2)}(e,e))$

j = f => $(\delta^{(2)}(a,f)=)\ 3+2\ (=\delta^{(2)}(f,e))$

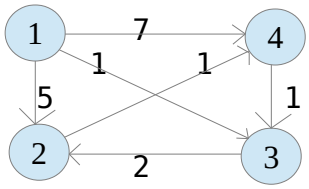
per ricostruire la soluzione con j = b andrò a leggere due $\pi^{(2)}$ sia di (a,b) che di (b,e)

2) Floyd-Warshall

Parte dalla seguente osservazione:
$\delta^{(0)}(i,j)$ = cammino da “i” a “j” di costo minimo senza attraversare nessun nodo intermedio
= costo dell'arco (i,j)
= W = → 0 $se\ i=j$
→ $w(i,j)$ $se\ (i,j)\in E$
→ $+\infty$ $se\ (i,j)\notin E$

Poi si considera $\delta^{(1)}(i,j)$, cioè il cammino di costo minimo da “i” a “j” passando per al più un nodo intermedio (il nodo 1)
$\delta^{(1)}(i,j)=min\{\ \delta^{(0)}(i,j)\ ;\ \delta^{(0)}(i,1)+\delta^{(0)}(1,j)\ \}$

Es:



$\delta^{(0)}$	1	2	3	4
1	0	5	1	7
2	+∞	0	+∞	1
3	+∞	2	0	+∞
4	+∞	+∞	1	0

$\delta^{(2)}$	1	2	3	4
1	0	5	1	6
2	+∞	0	+∞	1
3	+∞	2	0	3
4	+∞	+∞	1	0

$\delta^{(0)}=\delta^{(1)}$ Perché non ci sono archi entranti

$\delta^{(2)}(i,j)=$ cammino costo minimo da "i" a "j" passando per un sottoinsieme dei vertici {1,2}

=> $\delta^{(k)}(i,j)=$ cammino costo minimo da "i" a "j" passando per un sottoinsieme dei vertici {1, … , k}

$= \min\{ \delta^{(k-1)}(i,j); \delta^{(k-1)}(i,k) + \delta^{(k-1)}(k,j) \}$

notiamo che si lavora sempre sulla matrice precedente, quindi si possono sovrascrivere le matrici più vecchie

$\delta^{(3)}$	1	2	3	4
1	0	3	1	4
2	+∞	0	+∞	1
3	+∞	2	0	3
4	+∞	3	1	0

è un algoritmo di programmazione dinamica
vale il fatto che se sulla diagonale trovo un numero diverso da 0 allora esiste un ciclo negativo

Complessità in tempo:
$n (n^2) \rightarrow \in O(n^3)$

Complessità in spazio:
$\in O(n^2)$ si potrebbe ottimizzare ancora di più riscrivendo sulla stessa matrice con piccole modifiche alla definizione ricorsiva

la matrice dei padri sarà cosi composta:
$\pi^{(k)}(i,j)$ = → *true* *se* $\delta^{(k)}(i,j)= \delta^{(k-1)}(i,k) + \delta^{(k-1)}(k,j)$
→ *false* *se* $\delta^{(k)}(i,j)= \delta^{(k-1)}(i,j)$

3) Johnson
Dijkstra $O(m+n\log n)$
se applicato n volte (per ogni vertice) (<=> $w : E \to \mathbb{R}^+$)
=> $n(m+n\log n)$

se il grafo non è denso Johnson risulta essere anche meglio di n^3

Algoritmo:
1 – costruisci $G'=V \cup s\,;\, E \cup \{ (s,v)\ \forall\, v \in V \}$ $w(s,v)=0$
in pratica creo un nuovo vertice "s" e lo collego a tutti gli altri vertici con un arco di costo 0

- applico Bellman-Ford su G'
=> $\forall\, v \in V$ calcolo $\delta(s,v) = d(v)$

2 – ricalcolo i pesi $\forall\, E \in G$
s → v → u
$\delta(s,v)+w(v,u) \geq \delta(s,u)$ => $\hat{w}=d(v)+w(u,v)-d(u) \geq 0$
ora ho ottenuto un grafo con tutti i costi non negativi

3 – $\forall\, v \in V$ applica Dijkstra

osservo che:
u → u_1 → u_2 → v
costo cammino
C(u,v) = $\hat{w}(u,u_1)+\hat{w}(u_1,u_2)+\hat{w}(u_2,v)$
= $d(u)+w(u,u_1)-d(u_1)+d(u_1)+w(u_1,u_2)-d(u_2)+d(u_2)+w(u_2,v)+w(u_2,v)-d(v)$
= $d(u)+(w(u_1)+w(u_1,u_2)+w(u_2,v))-d(v)$

4) per ritrovare i costi originali
$\forall\, u,v$ $\delta(u,v)=C(u,v)-d(u)+d(v)$

Il cammino minimo trovato da Dijkstra è lo stesso al cammino minimo senza il cambio dei pesi poiché tutti i possibili cammini che trova sono alterati dalla stessa quantità $(d(u)-d(v))$

Complessità:
1 – $n \cdot m$
2 – m
3 – 4 – $n(m+n\log n)$
=> $O(nm+n^2\log n)$

"Abbiate il dubbio che non è tutto calcolabile" cit Pinotti

Esercitazione 8-05
Riepilogo BFS/DFS

L'algoritmo DFS, a seguito del suo procedimento considera 4 tipi di archi di cui un grafo orientato ne avrà tutti e 4 (pag. 109), un grafo non diretto invece ne potrà avere solo di due tipi (T, B).
Nei grafi non diretti per controllare che effettivamente un arco sia in B, si dovrà usare una DFS-visit modificata, che dovrà ricordarsi il padre DFS-visit(G, v, padre) (pag. 110).

La complessità di BFS e DFS è O(V + E), in particolare questa cambia rispetto a come è stato implementato il grafo:
- matrice delle adiacenze => $O(V^2)$
- lista delle adiacenze => O(V + E)

In un grafo non pesato esiste un almeno un ciclo se trovo un arco in B

Alcune tipologie di esercizio:
1) trovare il vertice “v” a massima distanza dal vertice “u” in G=(V, E)

si utilizzerà BFS
si ricorda che:
d(u) = 0 d(v) = $\delta(u,v)$ = #archi per arrivare ad “u”
$\delta(u,v)$ = minimo #archi da attraversare per raggiungere “v” a partire da “u”

Quindi una volta applicata BFS si scorre il vettore contenente d( ) e si cerca tra questi il massimo, quello sarà il vertice “v”
Costo: O(V + E) + V

2) Esistenza di cicli di lunghezza dispari su grafi non orientati

Ci basiamo sulla proprietà dei grafi bipartiti per cui un grafo è bipartito <=> non esistono cicli di lunghezza dispari

Allora applicheremo BFS-color (pag. 110)

3) Trovare le componenti connesse di un grafo non orientato

Basta applicare semplicemente DFS

4) Trovare la componente connessa di cardinalità massima

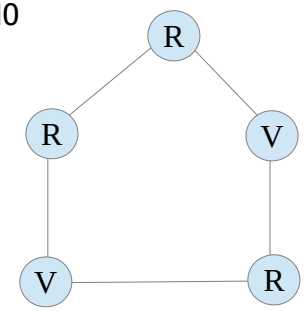
DFS tenendo il conteggio di quanti archi si visitano

5) Verificare se esiste una componente connessa di soli nodi “rossi"
G = (V , E) $\forall\, v \in V \quad info(v) = \{ rosso , verde \}$

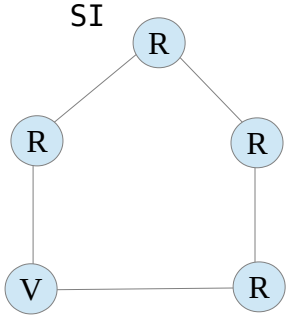
Si applicherà DFS controllando il campo info con un “if” e in caso si interrompe la visita

6) Dato un grafo G per cui $\forall\, v \in V \quad info(v) = \{\, rosso\ ,\ verde\ \}$
Dire se i nodi $R \subset V$: info(v) = rosso sono connessi ?
Scrivere una procedura che controlla se si ha una componente connessa di soli nodi rossi

NO



SI



perché posso raggiungere tutti i rossi senza passare per nodi verdi

Utilizzerò la DFS visitando solo i nodi raggiungibili marcati di rosso
Appena fallisce, cioè quando è stato trovato un nodo verde, controllerò se rimangono nodi rossi che non sono stati toccati (color = white), se si, allora non sono connessi

7) Testare, dato un grafo orientato, se un nodo è un pozzo universale
"v" lo è se $\forall\, u \in V - \{v\} \quad \exists\,(u,v)\ ;\ \nexists$ archi uscenti da v

Nella matrice delle adiacenze deve avere la riga del vertice con tutti zeri e la colonna di 1 (escluso se stesso)
Costo: O(V)

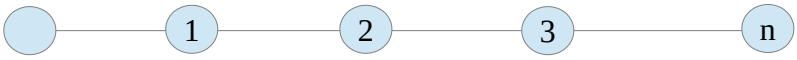
Sulla lista delle adiacenze devo controllare che da "v" la lista è "nil", ma su tutti gli altri vertici devo cercare il nodo "v"
Costo: O(E)

8) Verificare se un grafo è un DAG

Non ci devono essere cicli (nei grafi diretti)

Programmazione dinamica e Greedy

1) Problema delle automobili elettriche di autonomia di k chilometri
Si ha un percorso con varie distanze tra le stazioni di rifornimenti



$d(i+1, i) \leq k \quad \forall 0 \leq i \leq n-1$

Voglio minimizzare il numero dei rifornimenti

es:
k = 40

idea Greedy

```
r ← 0      // #numero di rifornimenti che faccio
i ← 0      // stazioni di rifornimento
p ← 0      // chilometraggio percorso
while ( i < n ){
     while ( p + d(i+1 , i) ≤ k  AND i < n ){
          i ←i + 1
          p ← p + d( i + 1 , i )
     }
     if( p + d( i + 1 , i ) > k ) {   // cioè devo rifornire
          r++;  p ← 0
     }
}
```

2) n file di lunghezza l1, l2, . . . , ln
devo memorizzarli in una USB-KEY di capacità D

Cerco S con massima cardinalità tale che $\sum_{i \in S} l_i \leq D$
scelta greedy: metti per primo il file più corto
ordina i file per lunghezza non decrescente

```
d ← 0;          i ← 1
while( d + l_i ≤ D  AND  i<n  ) {
     i++;  d ← d + l(i)
}
return i;
```

Dimostrazione ottimalità:
l[1] < l[2] < … < l[i]

OPT   l[1] … => devo dimostrare l'ottimalità nel resto
          l[1] ≠ OPT[1]
           => OPT[1] è sicuramente più lungo di l[1]
           => l[1] < OPT[1]
           la somma dei file seguenti a OPT[1] + l[1] è sicuramente minore a OPT[1] + i
           successivi
Costo: $O(n \log n + n)$

3) SUBSET-SUM
Esercizio 1 pag. 93

Problema zaino intero

M[i , m] = → true se $\exists \bar{s} \ \sum_{x \in S} x = m$
→ false otherwise

if($m \geq a_i$){
    M[i , m] = { (M[i-1, m] = true) OR (M[i-1 , $m - a_i$] = true) }
}
else
    M[i , m] ← M[i-1 , m]

Esercizi su grafi
Grafo con funzione peso

- Si potrebbe avere un costo dei pesi appartenente ad un insieme finito
es: w : E → { a , … , b }
=> con Kruskal si abbassa il costo di ordinamento (con il Counting-Sort, pag. 54)
    a (E + (b − a + 1))

- Dato un G e un MST si aggiunge un vertice
G' = V ∪ {new}; E ∪ { (new, v) ; $\forall v \in V$ }
cioè il nuovo vertice si unisce a tutti o ad un sottoinsieme

Voglio calcolare il nuovo MST
entra sicuramente l'arco di costo minore dei nuovi, ora devo controllare che è effettivamente un MST applicando la regola rossa ai cicli

Posso fare la ricerca del ciclo e verificare i pesi e togliere il massimo ma posso $O(V^2)$

Ma posso anche fare la seguente considerazione:
aggiungendo il vertice e i nuovi archi a G avremo che
G'' è così composto:
| V'' | = |V| + 1 e | E'' | ≤ |V − 1| + |V|
=> applico Prim (pag. 114) a G'', avrò complessità $V \log V + E$, con Fibonacci o $O((V+E)\log V)$ con lo heap

=> V log V dato da | E'' |

se il numero di archi che ho inserito è basso conviene utilizzare l'altra procedura