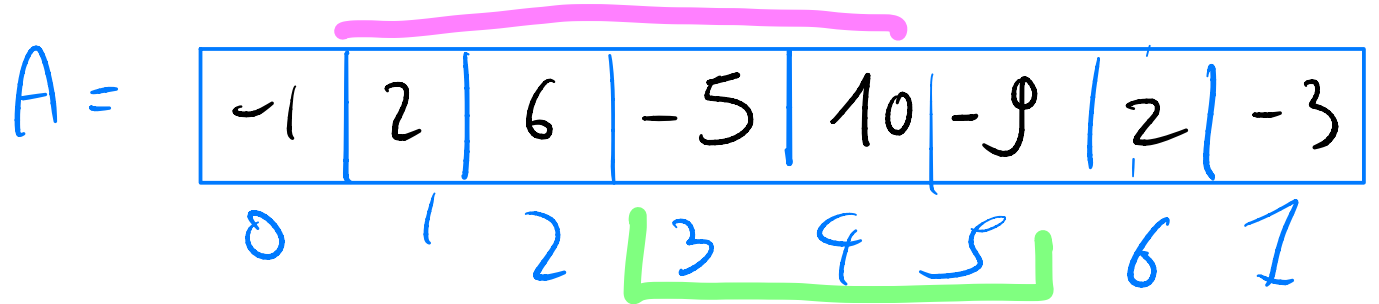
Segmento di somma massima

array/vector A di n elementi interi
alcuni negativi, altri positivi



segmento $A[i..j] = A[i]\,A[i+1] \dots A[j]$

$somma(i,j) = \sum_{k=i}^{j} A[k]$ es. somma(3,5) = -4

segmento di somma massima = $\max_{0 \le i \le j \le n-1} somma(i,j)$

BASELINE

```
1  SommaMassima1( a ):   ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
2    max = 0;
3    FOR (i = 0; i < n; i = i+1) {
4      FOR (j = i; j < n; j = j+1) {
5        somma = 0;
6        FOR (k = i; k <= j; k = k+1)
7          somma = somma + a[k];
8      IF (somma > max) max = somma;
9      }
10   }
11   RETURN max;
```

$0 \leq i \leq j \leq n-1$

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

$\in n^3$

prende la somma maggiore

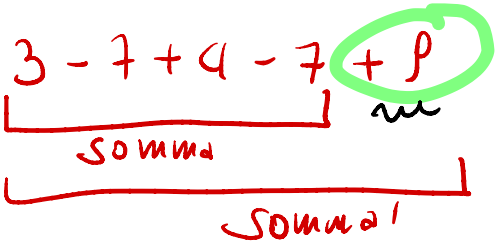
$somma(i,j+1) = somma(i,j) + A[j+1]$

$\sim n^2$ passi

```
1  SommaMassima2( a ): ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
2    max = 0;
3    FOR (i = 0; i < n; i = i+1) {
4      somma = 0;
5      FOR (j = i; j < n; j = j+1) {
6        somma = somma + a[j];
7        IF (somma > max) max = somma;
8      }
9    }
10   RETURN max;
```

somma ← somma(i, j)

```
1  SommaMassima1( a ):   ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
2    max = 0;
3    FOR (i = 0; i < n; i = i+1) {
4      FOR (j = i; j < n; j = j+1) {
5        somma = 0;
6        FOR (k = i; k <= j; k = k+1)
7          somma = somma + a[k];
8        IF (somma > max) max = somma;
9      }
10   }
11   RETURN max;
```



$\sim n$

```
1   SommaMassima3( a ):    ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
2     max = 0;
3     somma = max;
4     FOR (j = 0; j < n; j = j+1) {
5       IF (somma > 0) {
6         somma = somma + a[j];
7       } ELSE {
8         somma = a[j];
9       }
10      IF (somma > max) max = somma;
11    }
12    RETURN max;
```

Algoritmo A:

Complessità asintotica

- n = numero dei dati in ingresso
  ≠ dimensione del problema

- tempo di esecuzione
  ↳ spazio occupato in memoria

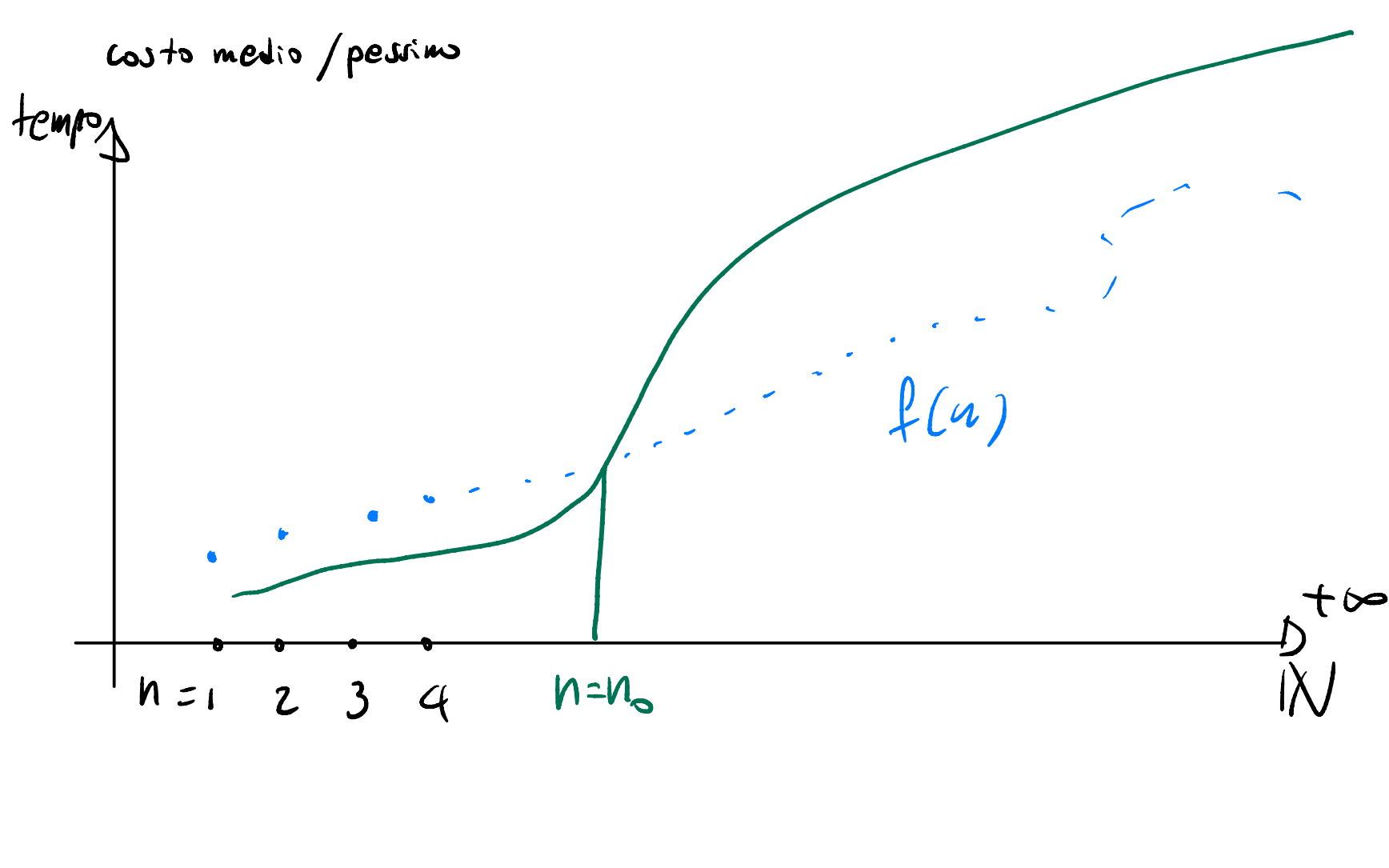
caso pessimo

caso medio

caso ottimo
(o min)

n → tutti gli input ammissibili di dimensione n

↳ tempo massimo


costo medio / pessimo
tempo
f(n)
+∞
n = 1
2
3
4
n = n₀
ℕ

LIM. SUPERIORE

$f(n) = O(g(n)) \iff \exists n_0, c > 0$ t.c. $\forall n \geq n_0 \quad f(n) \leq c \cdot g(n)$

LIM. INFERIORE

$f(n) = \Omega(g(n)) \iff \exists n_0, c > 0$ t.c. $\forall n \geq n_0 \quad f(n) \geq c \cdot g(n)$

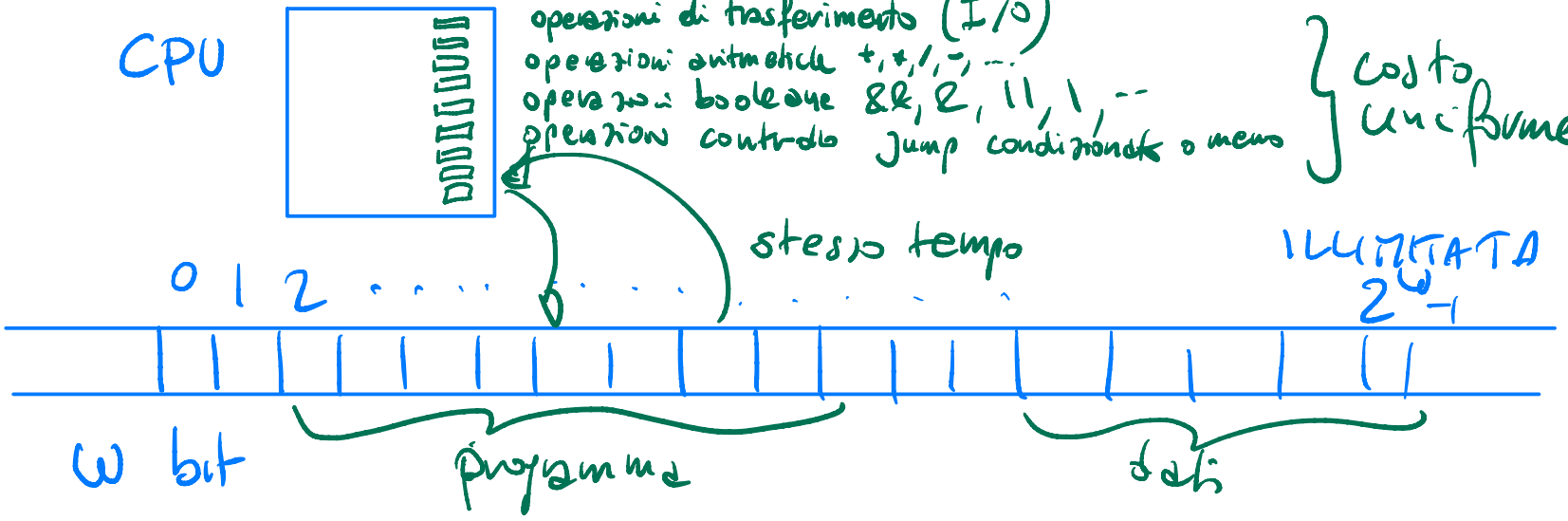
(altre definizioni)

$f(n) = \Theta(g(n))$ sse $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$

Criteri per la progettazione di algoritmi

- algoritmo deve essere CORRETTO : CORRETTEZZA
- algoritmo deve essere VELOCE/EFFICIENTE : COMPLESSITÀ COMPUTAZIONALE

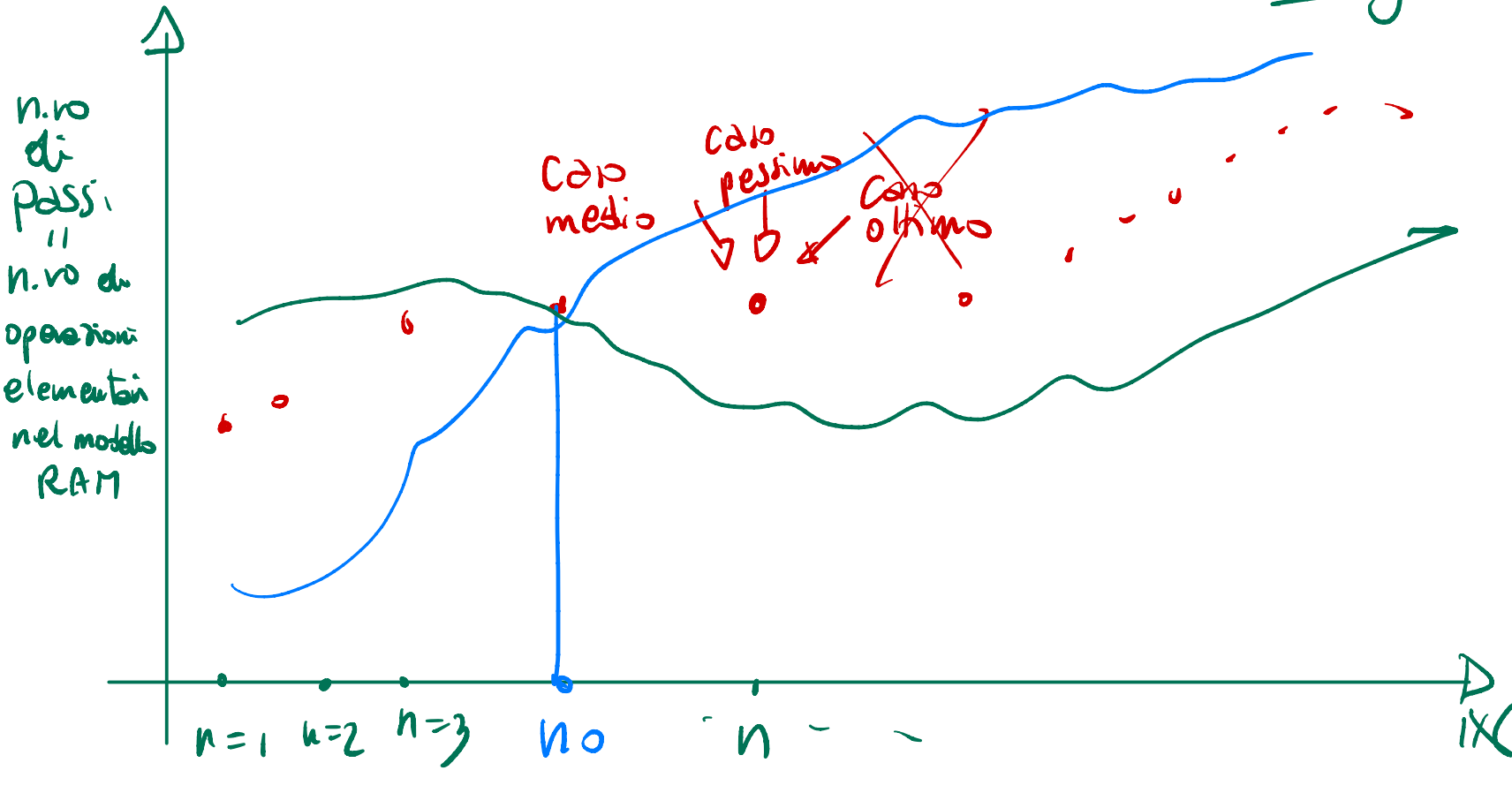
Modello di calcolo (semplificato) von Neumann (RAM = random access model)



Tempo di esecuzione

Complessità asintotica

A dp.



Costo computazionale

- tempo di esecuzione
- spazio occupato

(energia)

Analisi è asintotica

funzione del numero dei dati o dimensione dell'input

costanti moltiplicative e ordini inferiori: IGNORATI!

Confronto tra algoritmi è quantitativo: how fast

$f(n) = O(g(n))$ sse $\exists n_0, c > 0$ t.c. $f(n) \leq c\, g(n)$ $\forall n \geq n_0$

$f(n) = \Omega(g(n))$ sse $\exists n_0, c > 0$ t.c. $f(n) \geq c\, g(n)$ $\forall n \geq n_0$

$f(n) = \Theta(g(n))$ sse $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$

Algoritmo: derivare la sua complessità $f(n)$? Caso pessimo

- operazioni elementari del modello RAM : $O(1)$

  `x = x + 1;` $O(1)$

- costrutto condizionale

  `if (guardia) {blocco-then} else {blocco-else}`

  guardia: sempre valutata

  blocco-then / blocco-else: uno solo viene eseguito

Costo = costo(guardia) + max(costo(blocco-then), costo(blocco-else))

Iterazioni

for $(\underbrace{i=0}_{O(1)}; \underbrace{i<m}_{O(1)}; \underbrace{i++}_{O(1)})$ $\{\underbrace{\text{corpo}}_{t_i = \text{costo del corpo}}\}$

$$\text{costo} = O\left(m + \sum_{i=0}^{m-1} t_i\right)$$

while $(\underbrace{\text{guardia}}_{t'_i = \text{costo(guardia)}})$ $\{\underbrace{\text{corpo}}_{t_i = \text{costo(corpo)}}\}$

t'_i = costo (guardia) nell'i-esima iterazione

$0 \leq i \leq m$

m = # volte in cui la guardia è vera

$$\text{costo} = O\left(\sum_{i=0}^{m} t'_i + \sum_{i=0}^{m-1} t_i\right)$$

- chiamata a funzione non-ricorsiva

  $x = f(y_1, \ldots, y_k)$

  costo = costo del corpo della funzione

Nota Vedremo a parte come trattare le funzioni ricorsive

- blocco sequenziale di istruzioni

  $I_1;$
  $I_2;$
  $\vdots$
  $I_k;$

  $$\text{costo} = \sum_{i=1}^{k} \text{costo}(I_i)$$

esempio di analisi di complessità

```
1  SommaMassima3( a ):   ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
2    max = 0;
3    somma = max;
4    FOR (j = 0; j < n; j = j+1) {
5      IF (somma > 0) {
6        somma = somma + a[j];
7      } ELSE {
8        somma = a[j];
9      }
10     IF (somma > max) max = somma;
11   }
12   RETURN max;
```

O(1)
O(1)
O(1)
O(1)
O(1)
O(1)
O(1)
$t_j = O(1)$
O(1)

$$O\left(n + \sum_{j=0}^{n-1} t_j\right) = O(n)$$

```
1  SommaMassima2( a ): ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
2    max = 0;
3    FOR (i = 0; i < n; i = i+1) {
4      somma = 0;
5      FOR (j = i; j < n; j = j+1) {
6        somma = somma + a[j];
7        IF (somma > max) max = somma;
8      }
9    }
10   RETURN max;
```

$O(1)$

$O(1)$

$O(1)$ – t_j

$O(1)$

$t_i = \Theta(n-i)$

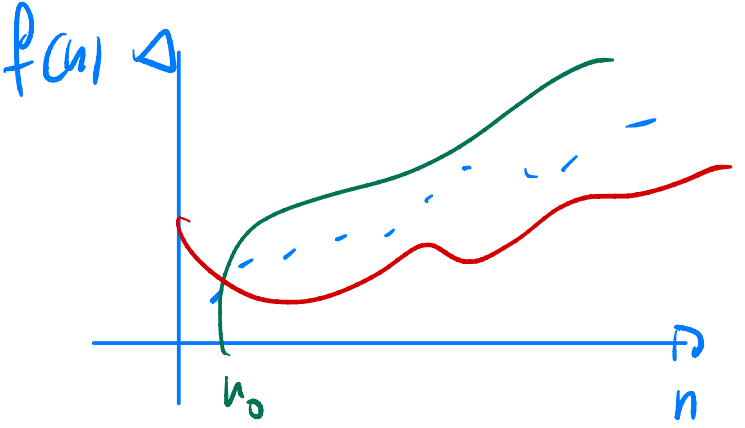
$\Theta\left(n-i+\sum_{j=i}^{n-1} O(1)\right)$

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

Complessità (costo) computazionale

ALGORITMO A

- ordine di grandezza del numero $f(n)$ di passi
- asintotico: al crescere di n = n. no (dim) dei dati
- caso pessimo / medio



- Problema computazionale $\Pi : IN \to OUT$

  $\Pi : \{0,1\}^* \to \{0,1\}^*$

  n = # bit in IN

$(\text{tight}) = \Theta(f(n)) = \Omega(f(n)) \cap O(f(n))$

- LIMITE SUPERIORE

  Π ha $O(f(n))$ come limite superiore se esiste un algoritmo la cui complessità è $O(f(n))$

- LIMITE INFERIORE

  Π ha $\Omega(f(n))$ come limite inferiore se ogni (presente, futuro) algoritmo ha complessità $\Omega(f(n))$

Limite inferiore semplice: fooling argument

"se non ci sono assunzioni sull'input, devi leggerlo tutto altrimenti non puoi risolvere correttamente il problema"

$\Omega(n)$ tempo per leggere gli n dati in input

→ (per esempio, l'input è già ordinato)

ORDINAMENTO

IN: array A di n elementi, su cui vale una relazione d'ordine

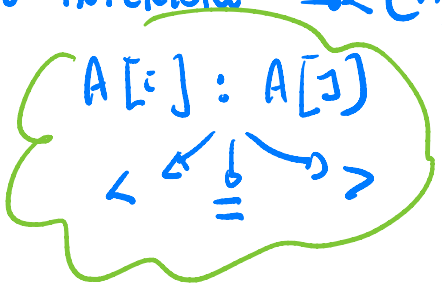
OUT: permutazione degli elementi di A in modo tale che risulti $A[0] < A[1] < \cdots < A[n-1]$

⇒ gli elementi di A possono essere solo confrontati a due a due o copiati

- LIM. INF. $\Omega(n \lg n)$ confronti
- LIM. SUP. $O(n^2)$ ~~→~~ $O(n \lg n)$

} complessità dell'ordinamento mediante confronti è $\Theta(n \lg n)$

LIMITE INFERIORE $\Omega(n \lg n)$ per confronti



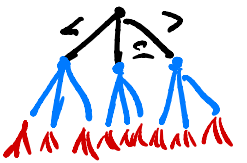
LED

ALG

BLACK BOX

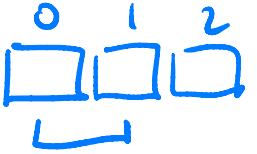
opera su A solo mediante confronti

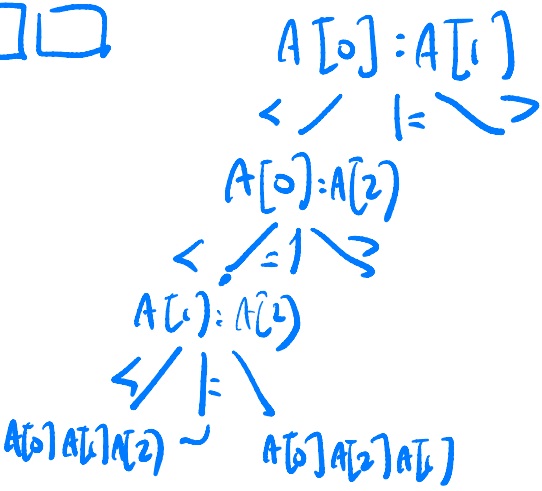
t	max n.ro di ordinamenti distinguibili con t confronti
0	1
1	3
2	3^2
3	3^3
⋮	
t	3^t



esempio di albero di decisione

$n = 3$ C, A, T

A 



ALG tra quante situazioni deve scegliere? Ci sono ∞ input di dimensione n

Input 1: C A T
Input 2: M E N
} 2 1 3

Due input sono equivalenti per ALG, se conducono alla stessa sequenza di confronti $\Rightarrow$ corrispondono alla stessa permutazione perché ALG usa solo l'ordine relativo tra coppie di elementi in A

ALG è corretto $\Rightarrow$ deve saper distinguere tra $n!$ permutazioni

$\Rightarrow$ se t = #confronti al termine dell'esecuzione di ALG,

$$3^t \geq n!$$

usare Stirling oppure $n! \geq \left(\frac{n}{2}\right)^{n/2}$ + log

$\Rightarrow$ $t = \Omega(n \lg n)$

LIMITE SUPERIORE

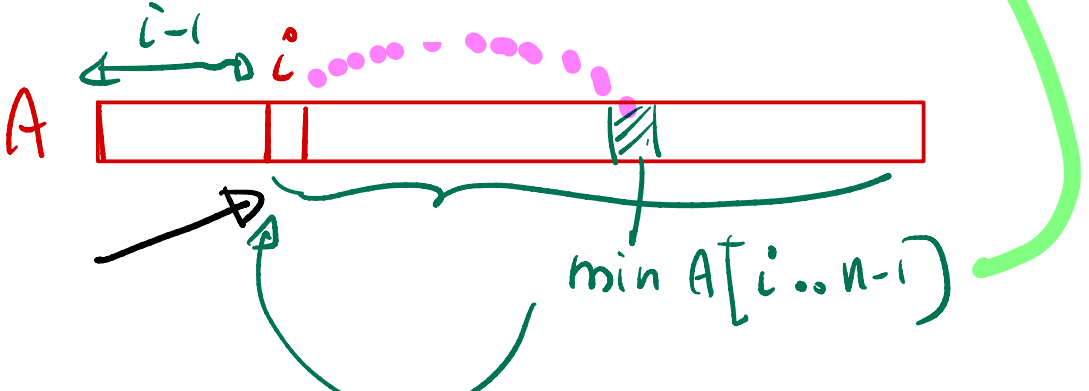
```
1   SelectionSort( a ):                          ⟨pre: la lunghezza di a è n⟩
2     FOR (i = 0; i < n-1; i = i+1) {
3   o(1)  minimo = a[i];
4   o(1)  indiceMinimo = i;
5       FOR (j = i+1; j < n; j = j+1) {
6     o(1)IF (a[j] < minimo) {
7     o(1)  minimo = a[j];
8     o(1)  indiceMinimo = j;
9         }
10      }
11  o(1)  a[indiceMinimo] = a[i];
12  o(1)  a[i] = minimo;
13    }
```

$\Theta(n + \sum_{i=0}^{n-1} t_i) = \Theta\left(\sum_{i=0}^{n-1}(n-i)\right) = \Theta(n^2)$

t_i

$t_i = \Theta\left(n - i + \sum_{j=i+1}^{n-1} \text{costante}\right)$

$= \Theta(n-i)$

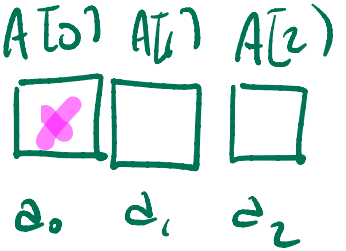


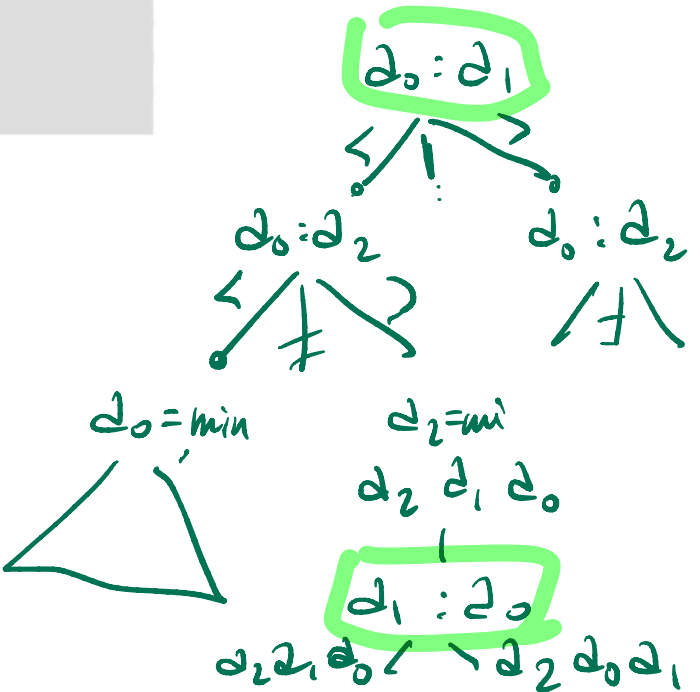
⇓

LIM. SUP.

$O(n^2)$

```
1  SelectionSort( a ):                          ⟨pre: la lunghezza di a è n⟩
2    FOR (i = 0; i < n-1; i = i+1) {
3      minimo = a[i];
4      indiceMinimo = i;
5      FOR (j = i+1; j < n; j = j+1) {
6        IF (a[j] < minimo) {
7          minimo = a[j];
8          indiceMinimo = j;
9        }
10     }
11     a[indiceMinimo] = a[i];
12     a[i] = minimo;
13   }
```

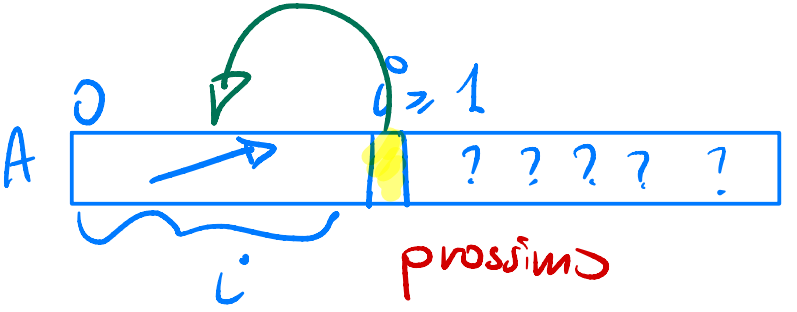




$\Theta(f(n))$ vs $O(f(n))$

selectionSort $\Theta(n^2)$

insertionSort $O(n^2)$



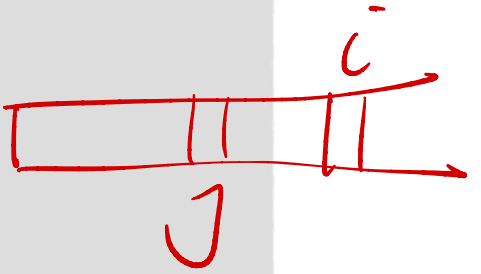

```
1  InsertionSort( a ):                          ⟨pre: la lunghezza di a è n⟩
2    FOR (i = 1; i < n; i = i+1) {
3      prossimo = a[i];
4      j = i;
5      WHILE ((j > 0) && (a[j-1] > prossimo)) {
6        a[j] = a[j-1];
7        j = j-1;
8      }
9      a[j] = prossimo;
10   }
```

$\sum_{i=1}^{n-1} O(m_i)$

$1 \le m \le i+1$

OK

$\Theta(m) = O(i)$



prossimo = A[i]

Esempio:

costo finale $O\left(n + \sum_{i=1}^{n-1} m_i\right)$ $\quad 1 \le m_i \le i+1$

caso ottimo (inutile?) $\Rightarrow m_i = O(1)$
$\Rightarrow O(n)$

già ordinato o quasi

caso medio $\Rightarrow m_i \sim \frac{i}{2}$
$\Rightarrow \sum_{i=1}^{n-1} \frac{i}{2} = \Theta(n^2)$

caso pessimo: $\Rightarrow m_i = i+1$
$\Rightarrow \sum_{i=1}^{n-1} (i+1) = \Theta(n^2)$

ordinato in modo inverso o quasi

Ordinamento

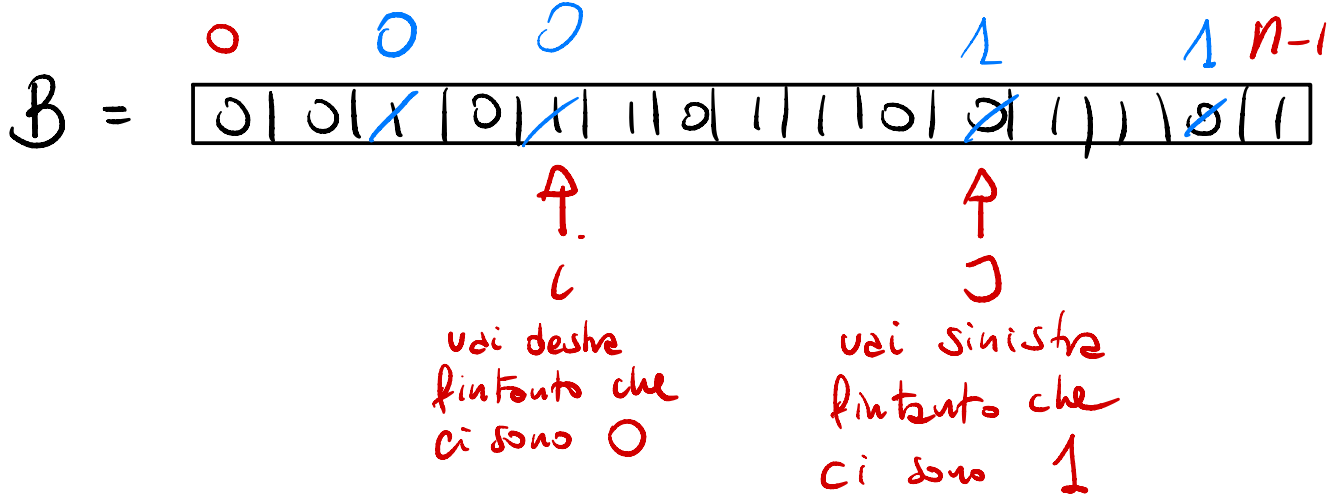
QUICKSORT → RICORSIVO
(metodologia Divide et impera)

- caso base : risolvi per un piccolo numero $n_0 = O(1)$ di dati
- passo induttivo : . dividi il problema in sottoproblemi della stessa forma ma con un numero di input inferiore
  - risolvi i sottoproblemi
  - combina le loro soluzioni

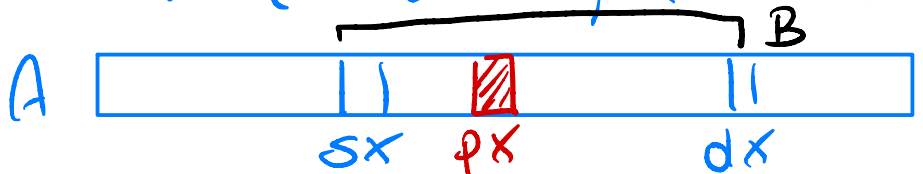
SOTTOPROBLEMA B = n bit $\in \{0,1\}$

Ordinare B mediante confronti e spostamenti

es



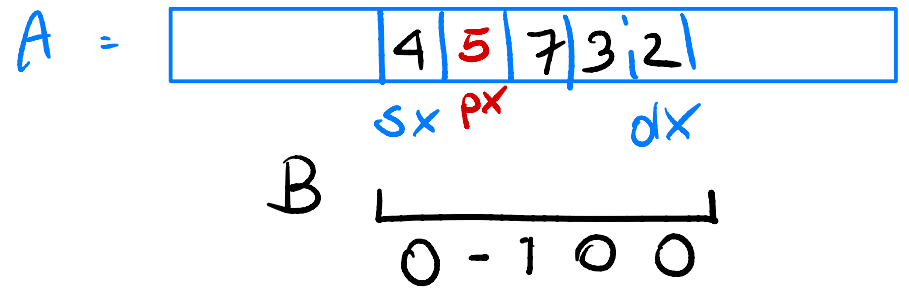
Nel nostro caso, QS deve ordinare in generale da sx a dx dentro A (inizialmente, sx=0 e dx=n-1)

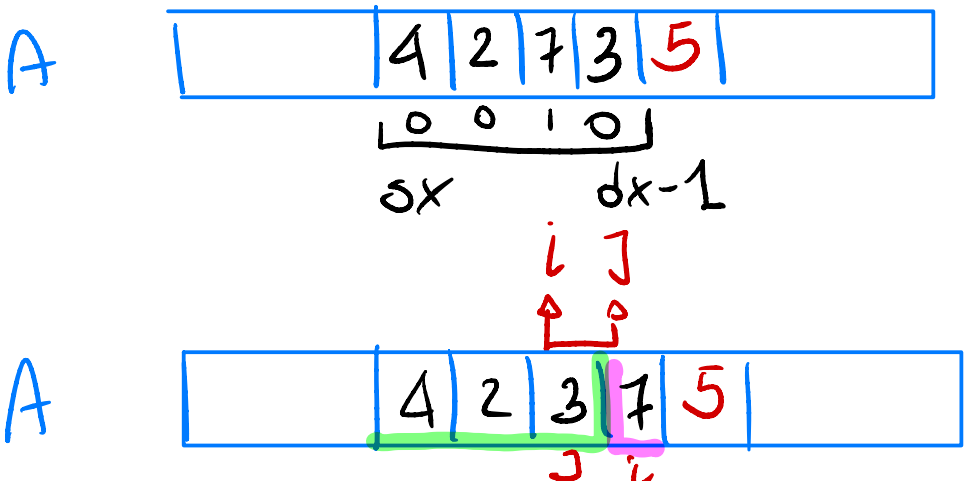


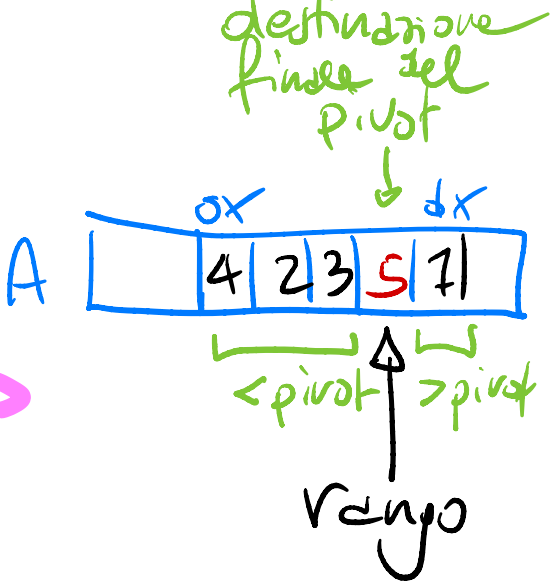
pivot: $sx \le px \le dx$
"A[px]

0: < pivot 1: > pivot

Esempio DISTRIBUZIONE o PARTIZIONE



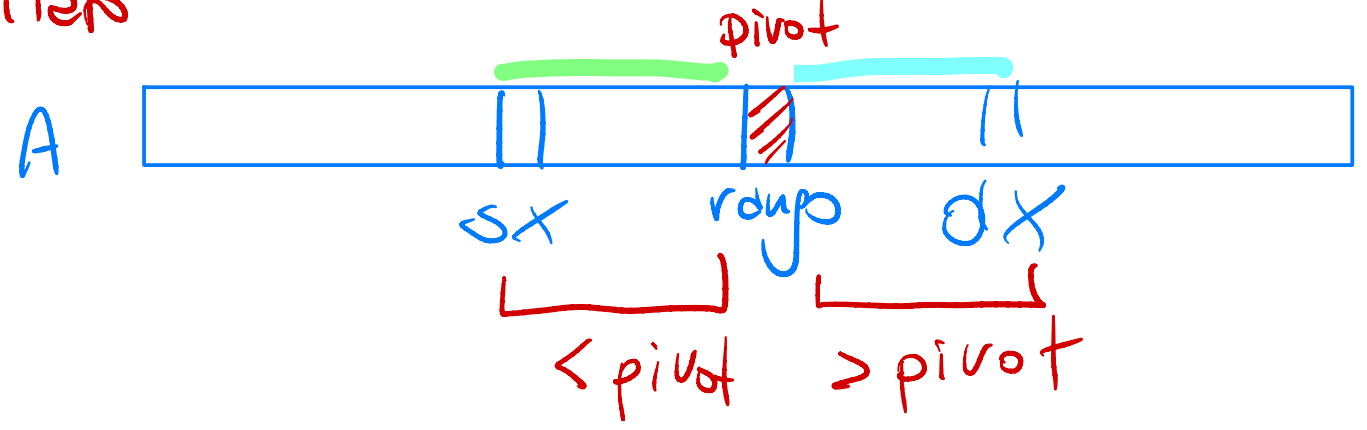




SCHEMA RICORSIVO DEL QUICKSORT

```
1  QuickSort( a, sinistra, destra ):
2       ⟨pre: 0 ≤ sinistra, destra ≤ n - 1⟩
3    IF (sinistra < destra) {
4      scegli pivot nell'intervallo [sinistra...destra];
5      rango = Distribuzione( a, sinistra, pivot, destra );
6      QuickSort( a, sinistra, rango-1 );
7      QuickSort( a, rango+1, destra );
8    }
```

risultato



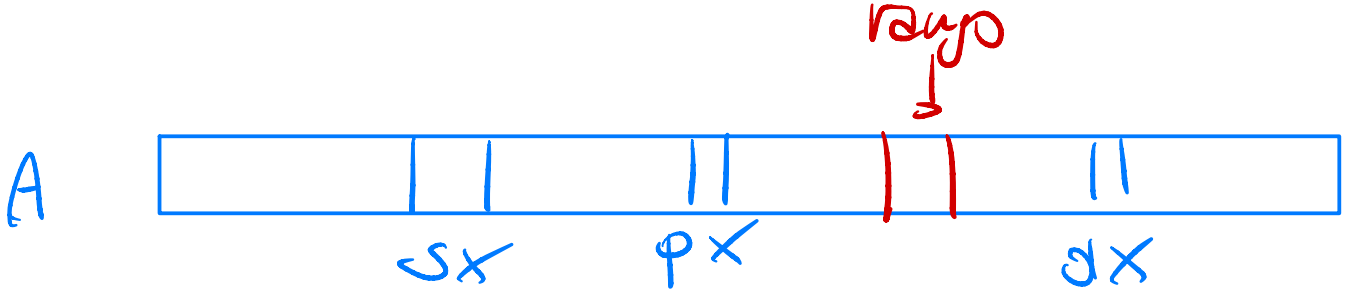
```
1   Distribuzione( a, sx, px, dx ):          ⟨pre: 0 ≤ sx ≤ px ≤ dx ≤ n - 1⟩
2     IF (px != dx) Scambia( px, dx );
3     i = sx;
4     j = dx-1;
5     WHILE (i <= j) {
6       WHILE ((i <= j) && (A[i] <= A[dx]))
7         i = i+1;
8       WHILE ((i <= j) && (A[j] >= A[dx]))
9         j = j-1;
10      IF (i < j) Scambia( i, j );
11    }
12    IF (i != dx) Scambia( i, dx );
13    RETURN i;
```

↳ scambia i contenuti di A[px] e A[dx]

0 ^ 0 = 0
1 ^ 0 = 1
0 ^ 1 = 1
1 ^ 1 = 0

A ^ B = C

(A ^ B) ^ B = A



Costo computazionale

```
1   Distribuzione( a, sx, px, dx ):          ⟨pre: 0 ≤ sx ≤ px ≤ dx ≤ n - 1⟩
2   IF (px != dx) Scambia( px, dx );
3   i = sx;
4   j = dx-1;
5   WHILE (i <= j) {
6     WHILE ((i <= j) && (A[i] <= A[dx]))
7   i = i+1;
8     WHILE ((i <= j) && (A[j] >= A[dx]))
9       j = j-1;
10    IF (i < j) Scambia( i, j );
11  }
12  IF (i != dx) Scambia( i, dx );
13  RETURN i;
```

O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1) O(1)

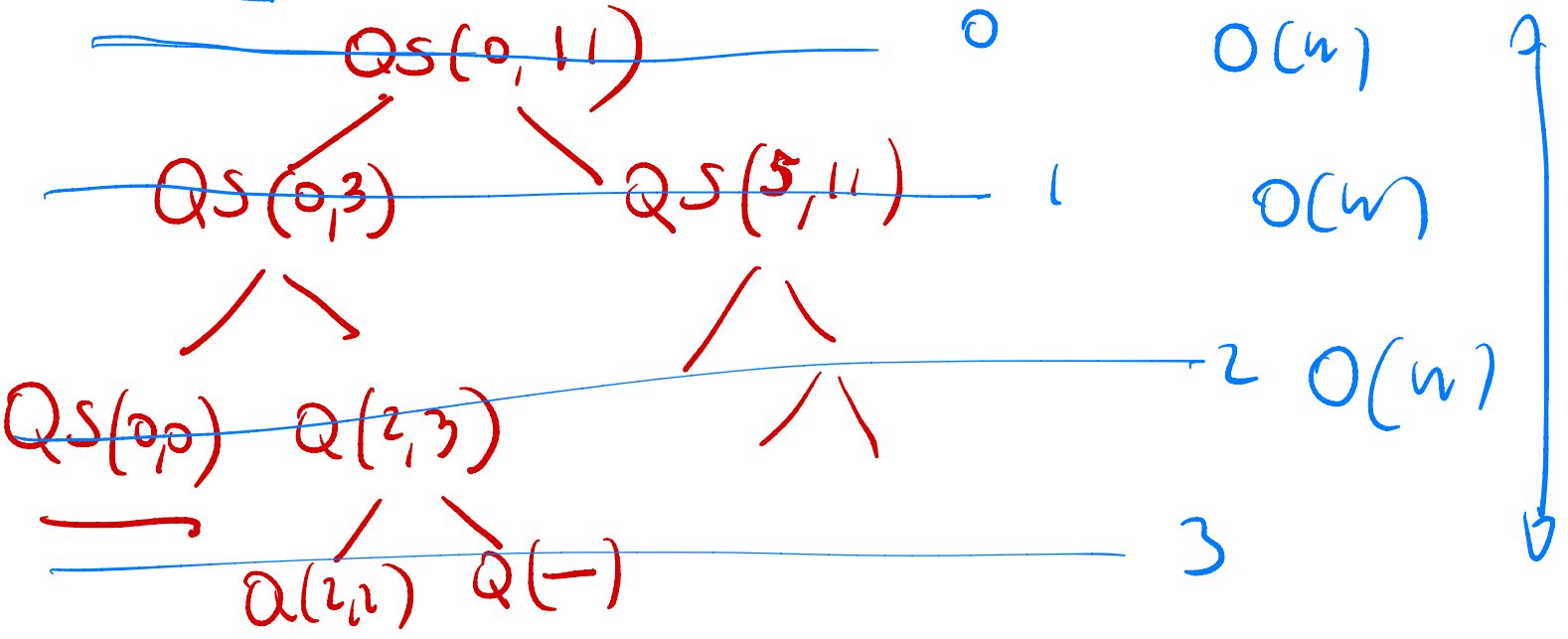
n_0 = # iterazioni del doppio while

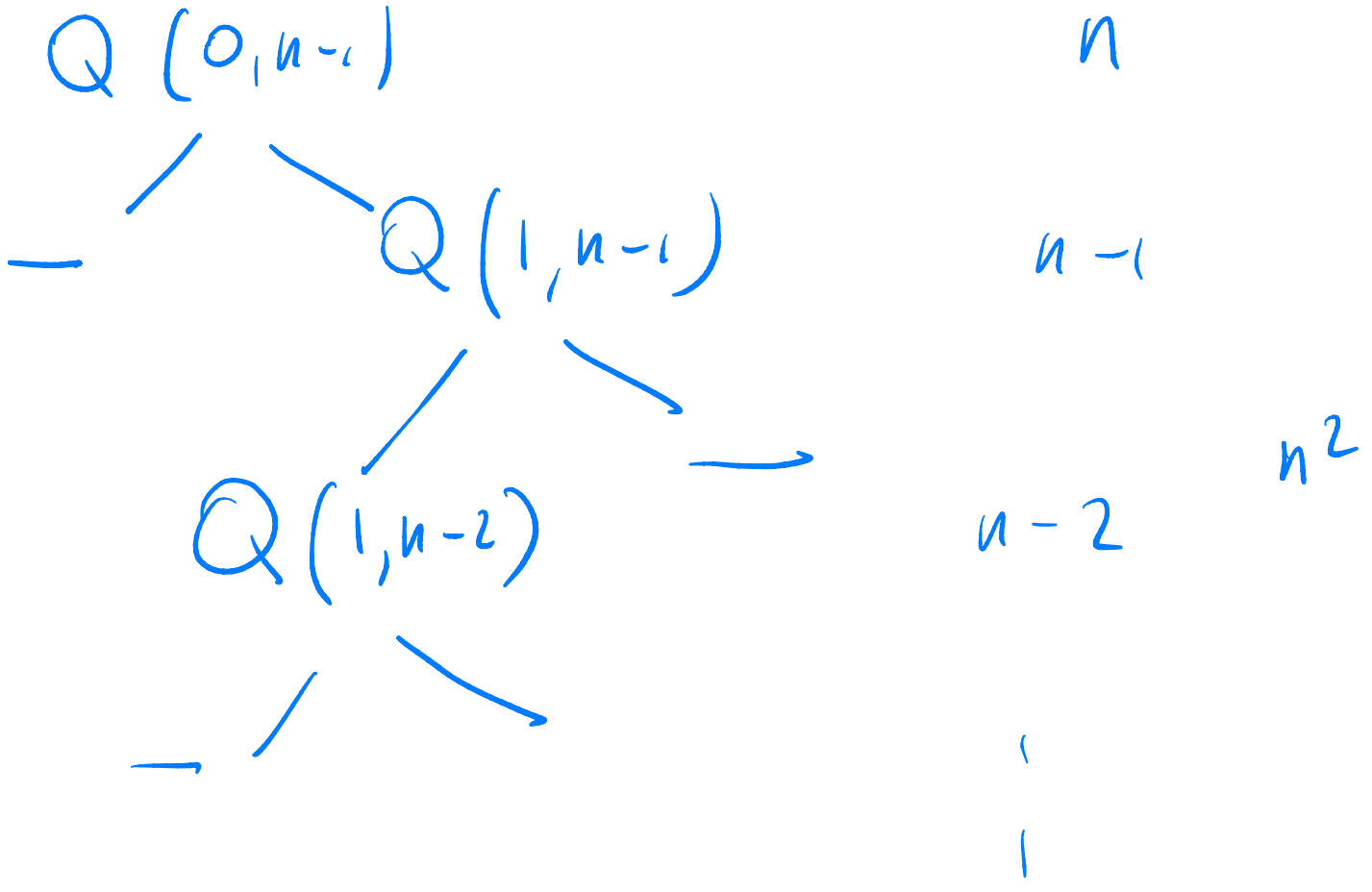
n_1 = " "

$n_0 + n_1 = O(n)$ ⇒ costo totale è $O(n)$

```
1  QuickSort( a, sinistra, destra ):
2       ⟨pre: 0 ≤ sinistra, destra ≤ n – 1⟩
3    IF (sinistra < destra) {
4      scegli pivot nell'intervallo [sinistra...destra];
5      rango = Distribuzione( a, sinistra, pivot, destra );
6      QuickSort( a, sinistra, rango-1 );
7      QuickSort( a, rango+1, destra );
8    }
```

Caso pessimo: albero delle chiamate ricorsive




Q(0,n-1)
Q(1,n-1)
Q(1,n-2)
n
n-1
n-2
1
1
n²

Caso pessimo è $O(n^2)$ perché ogni volta il pivot scelto è il minimo o il massimo del segmento $A[sx..dx)$ corrente

strategie deterministiche per scegliere il pivot
NON aiutano

ALGORITMO RANDOMIZZATO ←

- una funzione in più: random(a,b) → restituisce un intero nell'intervallo [a..b] con probabilità uniforme $\frac{1}{b-a+1}$ $\quad a \leq a \leq b$

QS randomizzato

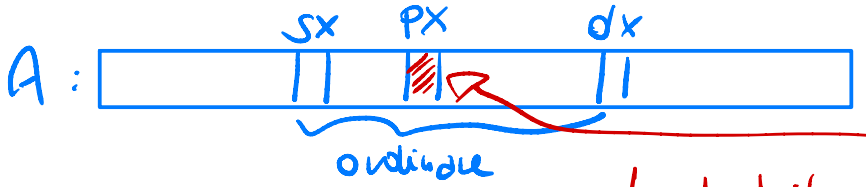
```
1  QuickSort( a, sinistra, destra ):
2        ⟨pre: 0 ≤ sinistra, destra ≤ n - 1⟩
3    IF (sinistra < destra) {
4      scegli pivot nell'intervallo [sinistra...destra];
5      rango = Distribuzione( a, sinistra, pivot, destra );
6      QuickSort( a, sinistra, rango-1 );
7      QuickSort( a, rango+1, destra );
8    }
```

pivot = random(sinistra, destra)

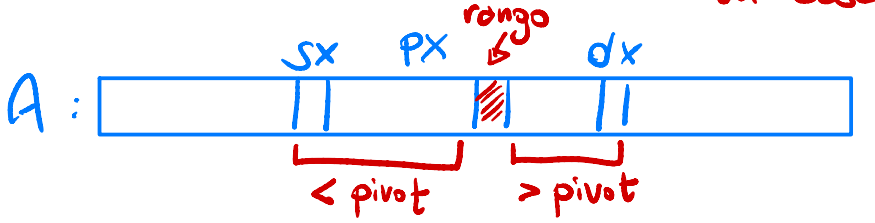
Caso medio

$O(n \lg n)$

QS randomizzato : px ← random(sx, dx)



distribuzione/partizione in base al pivot



tempo $O(dx - sx + 1)$

Parentesi su probabilità e randomization

Spazio di probabilità Ω discreto con prob. $P[\,]$

$$E[X] = \sum_{x \in \Omega} x \cdot Pr[X = x]$$

esempio: probabilità uniforme $= \frac{1}{5}$ 3, 1, 7, 2, 4

$$\Rightarrow \frac{3+1+7+2+4}{5}$$

$$E[aX + bY] = a\,E[X] + b\,E[Y]$$

non sono aleatorie

Variabile indicatrice

$$X = \begin{cases} 1 & \text{se un certo evento occorre, sia } p \text{ la sua probabilità} \\ 0 & \text{altrimenti} \end{cases}$$

→ lampadina a LED

$$E[X] = 0 \cdot (1-p) + 1 \cdot p = p$$

$X_1, X_2, \dots, X_n$

sono distribuite allo stesso modo

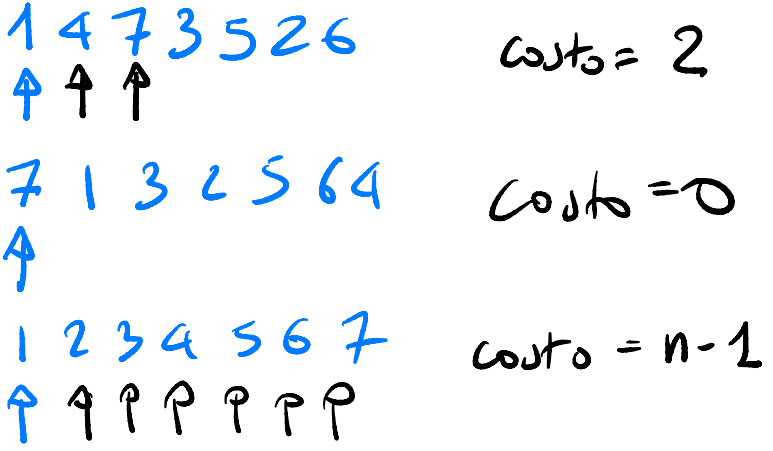
$$X = \sum_{i=1}^{n} X_i \quad \Rightarrow E[X] = \sum_{i=1}^{n} E[X_i] = np$$

($E[X_i] = p$)

Problema dello streaming

n negozi online ⇒ migliore servizio di streaming

Esempio



$0 \leq \text{costo} \leq n-1$

Idea: scegliere in modo casuale e uniforme il negozio da visitare

Usiamo variabili indicatrici

$$X_i = \begin{cases} 1 & \text{se compro lo streaming dal negozio } i \\ 0 & \text{altrimenti} \end{cases}$$

$X = \sum_{i=1}^{n} X_i$ è il numero di volte che ho comprato uno streaming

$$E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} Pr[X_i = 1] = \sum_{i=1}^{n} \frac{1}{i} \sim \ln n$$

$Pr[X_i = 1] =$

ex. 1 3 2 5 4 6 1

< 5 ; $i = 4$; non impatta su $Pr[X_i = 1]$

$i-1$ i

$< V$

non hanno impatto

$$\frac{\#\text{ permutazioni su } i \text{ elementi dove l'ultimo è il massimo}}{\#\text{ permutazioni su } i \text{ elementi}} = \frac{(i-1)!}{i!} = \frac{1}{i}$$

Esercizio: genera una permutazione di n elementi con prob. uniforme $\frac{1}{n!}$

```
for (i=0; i<n-1; i++) {
    j = random(i, n-1)
    scambia(i, j)
}
```

Usando le variabili indicatrici, dim. che otteniamo una qualunque permutazione con pr $= \frac{1}{n!}$

Analisi del QS randomizzato

```
1  QuickSort( a, sinistra, destra ):
2       ⟨pre: 0 ≤ sinistra, destra ≤ n - 1⟩
3    IF (sinistra < destra) {
4      scegli pivot nell'intervallo [sinistra...destra];
5      rango = Distribuzione( a, sinistra, pivot, destra );
6      QuickSort( a, sinistra, rango-1 );
7      QuickSort( a, rango+1, destra );
8    }
```

pivot = random (sinistra, destra)

X = # confronti costo = $O(n + X)$ tempo

costo medio = $O(n + E[X])$ tempo

scopo $X = \sum$ variabili indicatrici

① Dopo l'ordinamento, siano $z_1 < z_2 < \cdots < z_n$ gli elementi (ordinati)

SERVE SOLO AI FINI DELL'ANALISI

$$\forall i<j : X_{ij} = \begin{cases} 1 & \text{se } z_i \text{ e } z_j \text{ sono confrontate da QS rand.} \\ 0 & \text{altrimenti} \end{cases}$$

② $\forall i,j$ se z_i e z_j sono confrontate, una di loro è certamente il pivot dell'altro

③ $\forall i,j$ z_i e z_j sono confrontate al più una volta in tutto il QS rand.

④ $\forall i,j$: se z_i e z_j sono nella stessa partizione, allora anche $z_{i+1}, z_{i+2}, \ldots, z_{j-1}$ sono in tale partizione

① + ③ $\Rightarrow X = \sum_{i<j} X_{ij}$ è il numero totale di confronti operati dal QS rand.

$$E[X] = \sum_{i<j} E[X_{ij}] = \sum_{i<j} \underbrace{Pr[X_{ij}=1]}_{\text{② pr. } z_i \text{ o } z_j \text{ siano pivot una dell'altra}} \overset{④}{\leq} \sum_{i<j} \frac{2}{j-i+1}$$

$$\leq \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \sim \sum_{i} \underbrace{\sum_{k=2}^{n-i+1} \frac{2}{k}}_{\sim \ln n} = O(n \lg n)$$

SORTING:

selection sort $\Theta(n^2)$

insertion sort $O(n^2)$

quicksort $O(n^2) \rightsquigarrow O(n \lg n)$ caso medio
(o hybrid) randomized

limite inferiore $\Omega(n \lg n)$

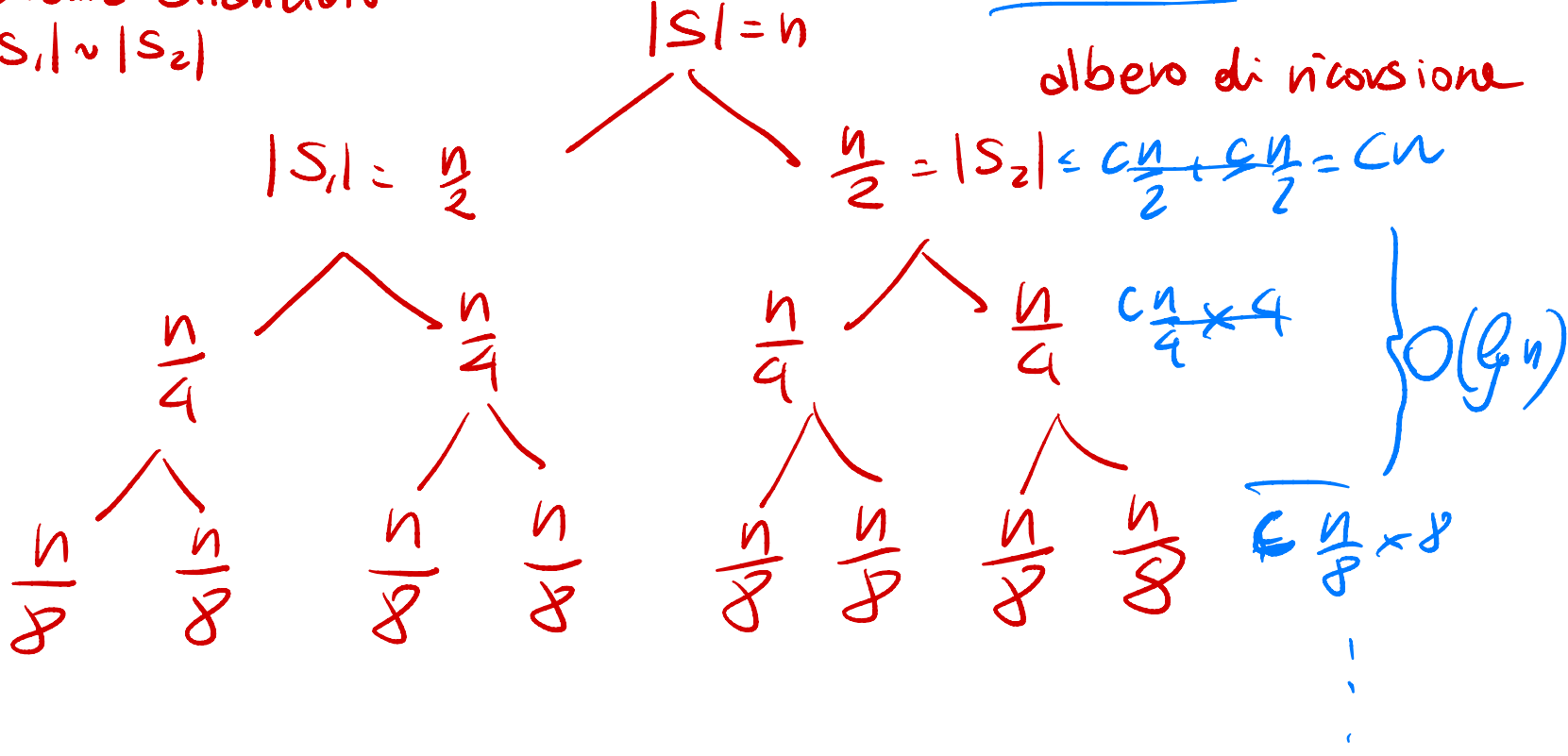
✓ mergesort $O(n \lg n)$

Lo illustrare il metodo divide et impera:

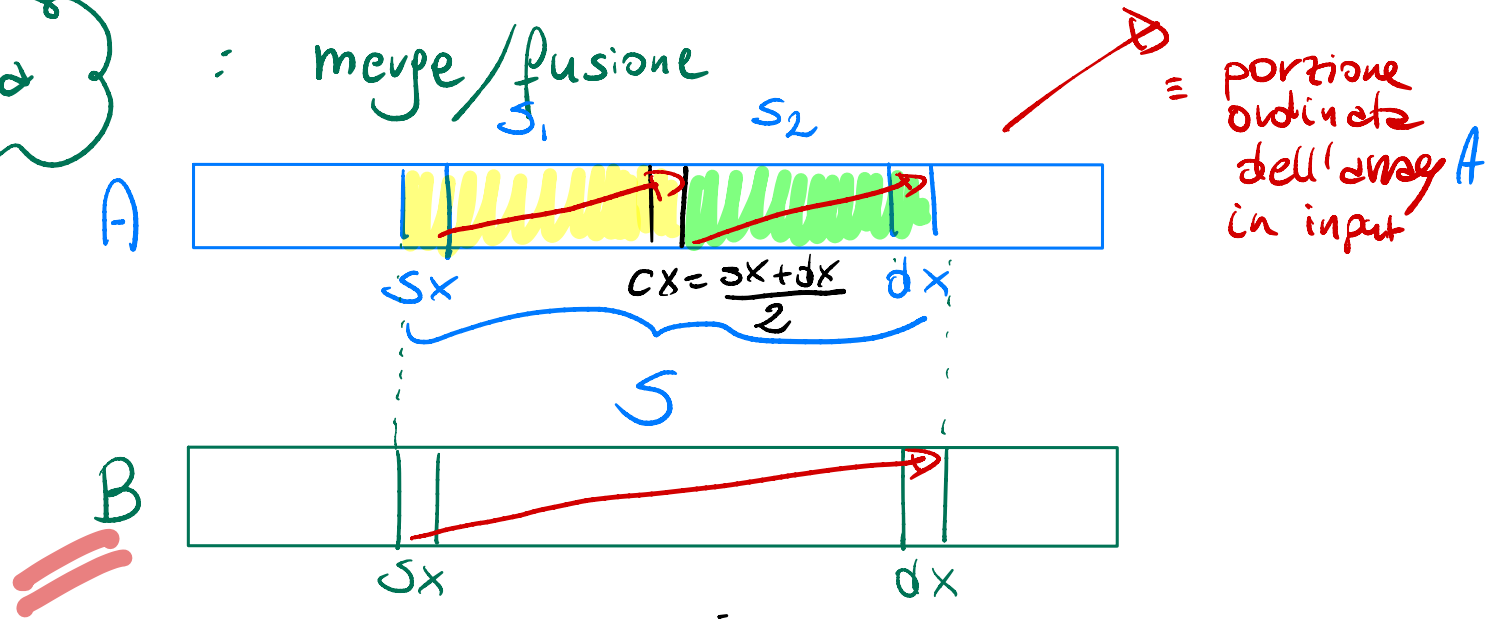
problema Π su un insieme S di dati in input

- BASE: $|S| \leq n_0$ costante $\Rightarrow$ soluzione $\Pi(S)$ per ispezione diretta
- INDUTTIVO/RICORSIONE
  - divide $S \rightarrow S_1 \cup S_2$ $\quad S_1 \cap S_2 = \phi$ $\quad S_1 \cup S_2 = S$
  - ricorsione (impera) $\Pi(S_1), \Pi(S_2)$
  - ricombinazione $\Pi(S) \leftarrow f(\Pi(S_1), \Pi(S_2))$

schema bilanciato
$|S_1| \sim |S_2|$



Idea : merge/fusione



≡ porzione ordinata dell'array A in input

i j

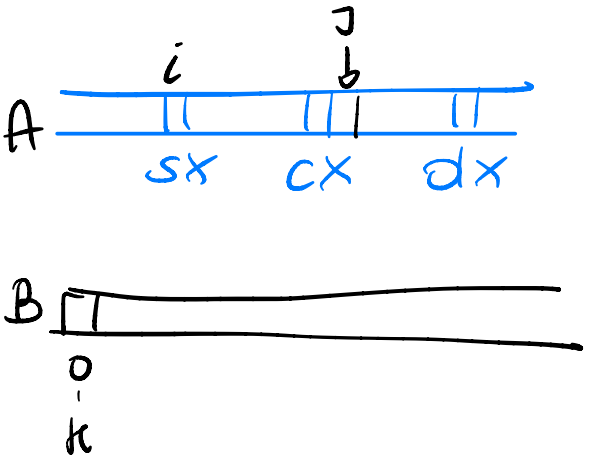
esempio | 3 7 18 21 4 20 25 30

S_1 S_2

3 4 7 18 20 21 25 30

```
1  Fusione( a, sx, cx, dx ):                    ⟨pre: 0 ≤ sx ≤ cx < dx ≤ n − 1⟩
2    i = sx;
3    j = cx+1;
4    k = 0;
5    WHILE ((i <= cx) && (j <= dx)) {
6      IF (a[i] <= a[j]) {
7        b[k] = a[i];
8        i = i+1;
9      } ELSE {
10       b[k] = a[j];
11       j = j+1;
12     }
13     k = k+1;
14   }
15   FOR ( ; i <= cx; i = i+1, k = k+1)
16     b[k] = a[i];
17   FOR ( ; j <= dx; j = j+1, k = k+1)
18     b[k] = a[j];
19   FOR (i = sx; i <= dx; i = i+1)
20     a[i] = b[i-sx];
```

ciascuno nella sua zona



copiare la metà non vuota

for(/**/ ; i <= cx ; i++)

$\Theta(n)$ tempo

n = destra − sinistra + 1 = numero di elementi da ordinare

```
1  MergeSort( a, sinistra, destra ):                    T(n)
2     /caso base/                  ⟨pre: 0 ≤ sinistra ≤ destra ≤ |A| − 1⟩
3    IF (sinistra < destra) {
4      centro = (sinistra+destra)/2;   ← divide          O(1) tempo
5      MergeSort( a, sinistra, centro );  ← T(n/2)       impera
6      MergeSort( a, centro+1, destra );  ← T(n/2)
7      Fusione( a, sinistra, centro, destra );
8    }                                  Θ(n) ricombina
```

$$\text{costo } T(n) \leq \begin{cases} \text{costante} & \text{se } n \leq 1 \\ 2T(n/2) + c \cdot n & \text{costante } c > 0 \end{cases}$$

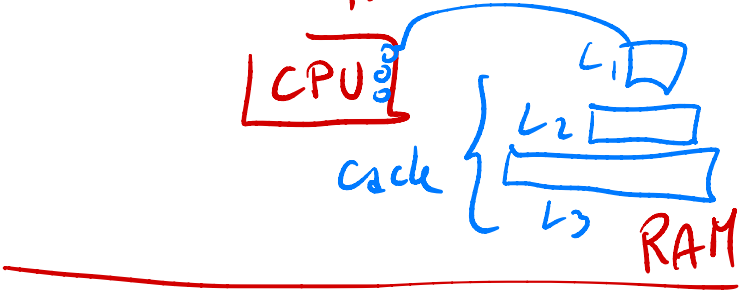
$2T(n/2)$: impera; $c \cdot n$: divide + ricombina

$$T(n) \leq 2T(n/2) + cn \Rightarrow T(n) = O(n \lg n)$$

usando l'albero della ricorsione

Per grandi quantità di dati:

- scansione sequenziale di A è molto efficiente
  - livelli di cache
  - memoria esterna



memoria interna

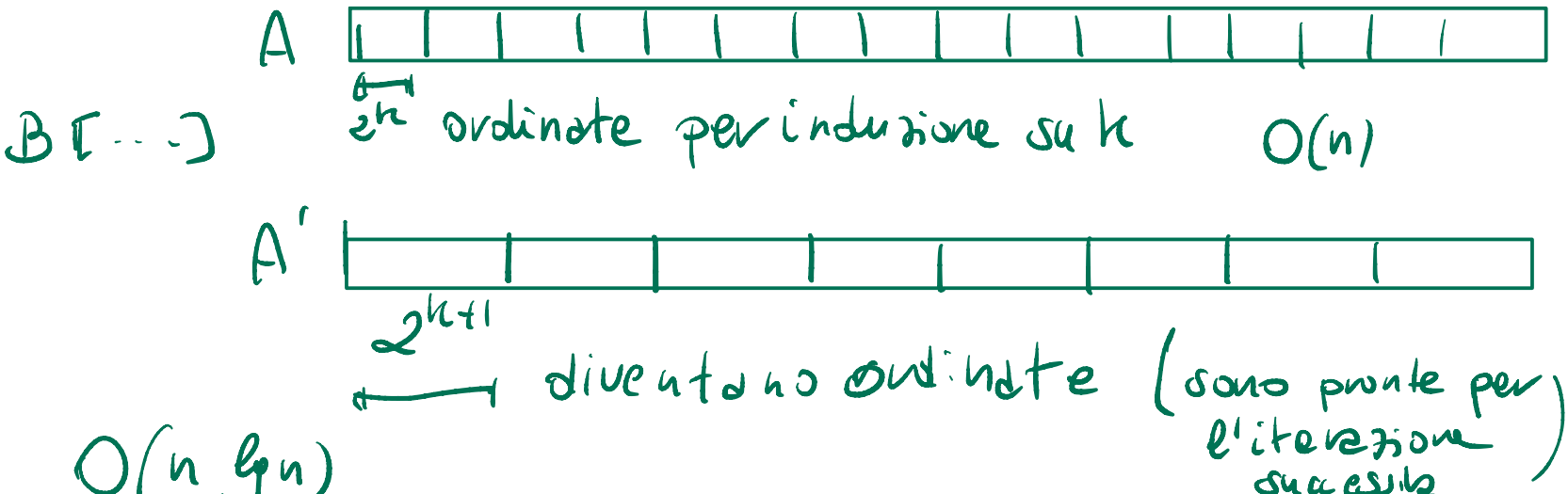
principio di località spaziale e temporale

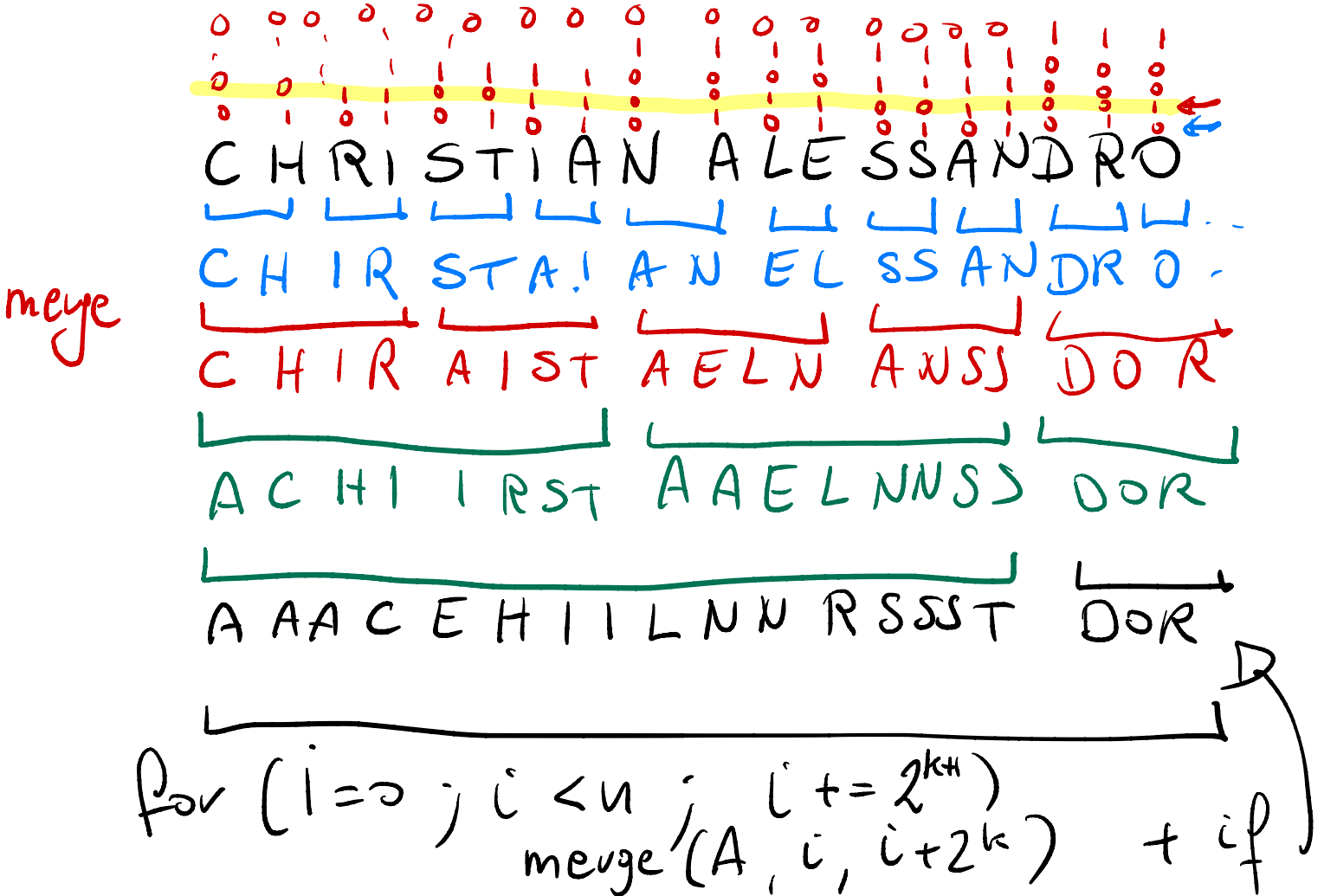
memoria esterna

Merge/fusione avviene tramite scansione i, j, k vanno solo in avanti

.ricorsione va sostituita con $\lg n$ scansioni

. for $(k=0;\ k \le \lg n;\ k++)$
scansione di tutto A con parametro k




CHRISTIAN ALESSANDRO
CHIRSTA!ANELSSANDRO
meyge
CHIRAISTAELNANSSDOR
ACHIIRSTAAELNNSSDOR
AAACEHIILNNRSSSTDOR
for (i=0 ; i<n ; i+=2^(k+1))
merge(A, i, i+2^k) + if

Divide et impera < Quicksort
< Mergesort $T(n) \leq 2T(\frac{n}{2}) + cn$ $c>0$ costante

$T(n) \leq \alpha T(\frac{n}{\beta}) + cf(n)$

- α: # chiamate ricorsive
- $\frac{n}{\beta}$: ogni chiamata ricorsiva è su $\frac{n}{\beta}$ elementi
- $f(n)$: tutto ciò che rimane (divide + ricombina)

Teorema 3.1 *Sia* $f(n)$ *una funzione non decrescente e siano* α, β, n_0, c_0 *e* c *delle costanti tali che* $\alpha \geq 1$, $\beta > 1$ *e* n_0, c_0, $c > 0$, *per la relazione di ricorrenza*

$$T(n) \leq \begin{cases} c_0 & \text{se } n \leq n_0 \\ \alpha T(n/\beta) + cf(n) & \text{altrimenti} \end{cases} \qquad (3.2)$$

(*dove* n/β *va interpretato come* $\lfloor n/\beta \rfloor$ *o* $\lceil n/\beta \rceil$). *Se esistono due costanti positive* γ e n'_0 tali che $\alpha f(n/\beta) \leq \gamma f(n)$ *per ogni* $n \geq n'_0$, *allora la relazione di ricorrenza ha i seguenti limiti superiori*:

1. $T(n) = O(f(n))$ *se* $\gamma < 1$;
2. $T(n) = O(f(n)\log_\beta n)$ *se* $\gamma = 1$;
3. $T(n) = O(n^{\log_\beta \alpha})$ *se* $\gamma > 1$ e $\alpha > 1$.

MASTER THEOREM

Nel MS: $a=2,\ b=2,\ f(n)=n \Rightarrow \gamma=1$

$$a f\left(\frac{n}{b}\right) = \gamma f(n)$$

$$2\,\frac{n}{2} = 1 \cdot n$$

$\Rightarrow$ caso 2 del master theorem

$$\Rightarrow O(f(n) \lg_b n) = O(n \lg n)$$

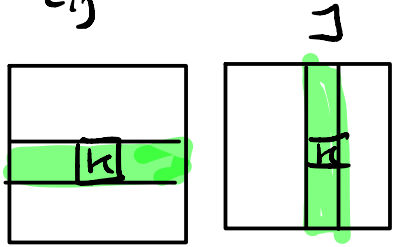
Q. conviene dividere in più di 2 parti nel mergesort? dividendo in $a=b=\text{cost} \geq 2$ parti, vale sempre le 2 e quindi cambia solo $\lg_b n = O(\lg n)$

MOLTIPLICAZIONE TRA MATRICI

A, B = matrici $n \times n$ $\rightarrow$ $C = A \times B$

a_{ij} b_{ij} c_{ij}

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$



$c_{ij} = \Theta(n)$ tempo

$C = \Theta(n^3)$ tempo

Strassen (anni '70)

$\times$ { $O(n^3)$ è un upper bound per il problema della MM
{ $\Omega(n^2)$ è un lower bound ovvio (fooling argument)

$+$ { $\Theta(n^2)$ tempo è la complessità asintotica per la somma

Congettura: la complessità della MM è $\Theta(n^\omega)$
per un qualche ω t.c. $2 < \omega \leq 2.371552$

Strassen's idea: usare divide-et-impera

hp. n = potenza del 2

Caso base: $n=1$ → prodotto tra due scalari $O(1)$ tempo

caso induttivo: $n \geq 2$

$$C_{ij} = A_{i1} \times B_{1j} + A_{i2} \times B_{2j}$$

$n/2 \times n/2$ $\quad n/2 \times n/2$ $\quad n/2 \times n/2$

$1 \leq i, j \leq 2$

$$\underset{C}{\begin{matrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{matrix}} = \underset{A}{\begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix}} \times \underset{B}{\begin{matrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{matrix}}$$

$n/2 \times n/2$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

4 somme
8 moltiplicazioni

Relazione di ricorrenza

$T(n)$ = costo di moltiplicazione di due matrici $n \times n$

$$T(n) \leq 8\,T\left(\frac{n}{2}\right) + c\,n^2 \qquad c > 0$$

Caso 3 del master theorem $\alpha = 8$ $\beta = 2$ $f(n) = n^2$

$$\alpha f\left(\frac{n}{\beta}\right) \leq \gamma f(n)$$

$$8\left(\frac{n}{2}\right)^2 \leq \gamma n^2$$

$$8\,\frac{n^2}{4} = 2n^2 \leq \gamma n^2 \Rightarrow \gamma = 2 > 1 \Rightarrow T(n) = O\left(n^{\log_\beta \alpha}\right) = O(n^3)$$

Strassen osservò che $q = 7$ moltiplicazioni sono possibili

$\Rightarrow O(n^{\log_2 q}) = O(n^{\log_2 7})$ dove $\log_2 7 < 3$

$$
\begin{aligned}
v_0 &= (b - d)(g + h) & v_4 &= a(f - h)\\
v_1 &= (a + d)(e + h) & v_5 &= d(g - e)\\
v_2 &= (a - c)(e + f) & v_6 &= (c + d)e\\
v_3 &= (a + b)h
\end{aligned}
$$

$$
\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix} \triangleq \begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} v_0 + v_1 - v_3 + v_5 & v_3 + v_4 \\ v_5 + v_6 & v_1 - v_2 + v_4 - v_6 \end{bmatrix}
$$

$v_5 + v_6 = dg - de + ce + de = ce + dg$

NEAREST NEIGHBOR (NN)

coppia di punti più vicina

IN: $S \subseteq \mathbb{R}^d$, $d(\,) : S \times S \to \mathbb{R}_0^+$

OUT: $x, y \in S$, $x \neq y$, t.c. $d(x,y) \leq d(x',y') \quad \forall\, x', y',\ x' \neq y'$
$x', y' \neq x, y$

(vector space model $\to$ d è enorme)

- Noi vediamo il caso $d = 2$ (piano Cartesiano)
  e $d(\,)$ = euclidea

BASELINE: $\forall$ coppie di punti, calcola la sua distanza
Ricorda una coppia con distanza minima

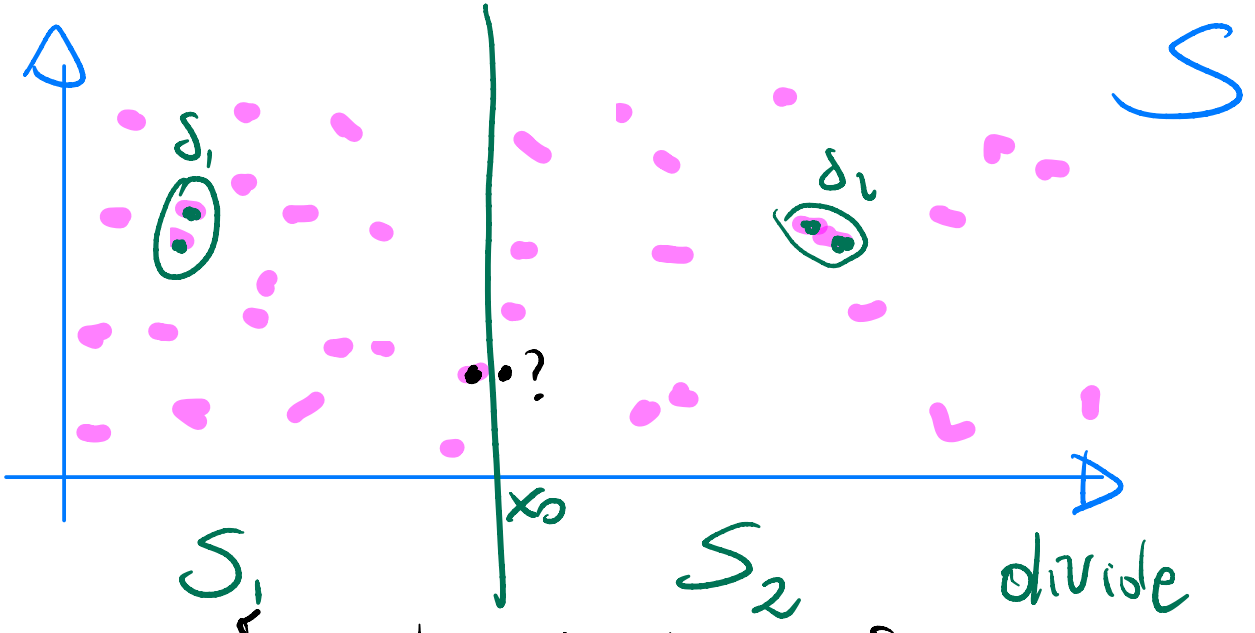
$\hookrightarrow n = |S| \rightarrow \binom{n}{2}$ possibili coppie, ciascuna distanza costa $O(1)$ tempo $\rightarrow \Theta(n^2)$ tempo

USIAMO DIVIDE ET IMPERA

- caso base = $n \leq 3$: ispezione diretta in $O(1)$
- passo induttivo :

$S \rightarrow S_1 \cup S_2$

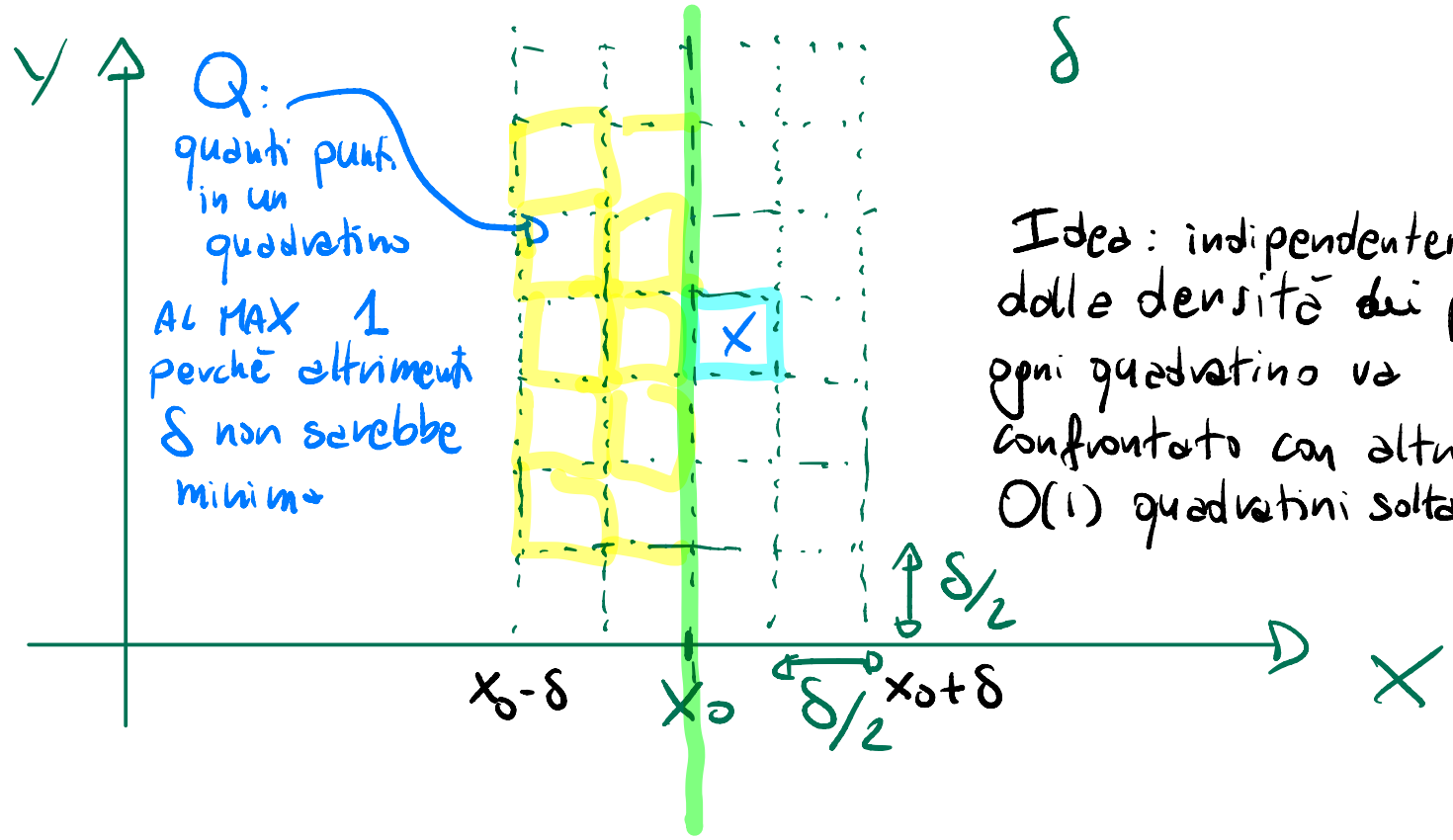
1) ordina S in base alla ascissa una volta per tutte $O(n \log n)$
   ossia $S' = S \Rightarrow$ otteniamo S' ordinato
2) Stessa cosa per l'ordinata (vedi dopo)



divide $O(n)$ tempo (vedi 1)

- conosciamo δ_1 per la coppia minima in S_1
- " δ_2 " " " " S_2
- manca la coppia minima p_1, p_2 dove $p_1 \in S_1$, $p_2 \in S_2$ impera

Prop cruciale: a noi interessano le coppie $p_1 \in S_1$ e $p_2 \in S_2$ t.c. $d(p_1, p_2) < \delta \triangleq \min\{\delta_1, \delta_2\}$



Idea: indipendentemente dalle densità dei punti ogni quadratino va confrontato con altri $O(1)$ quadratini soltanto

Selezioniamo $S_y \subseteq S$ t.c. $S_y = \{(x,y) \in S \mid |x - x_0| \leq \delta\}$

Nota: per 2), S ordinato $\Rightarrow$ S_y ordinato su y in $O(n)$ tempo

Scorriamo S_y in ordine e confrontiamo ciascun punto in S_y soltanto con i suoi successivi (n.o costante)

$O(n)$ tempo in totale

$T(n) \leq 2T(n/2) + cn$ (a cui si aggiunge il costo $O(n \lg n)$ di 1) e 2))

Indecidibilità e casualità

- Alan Turing '37 ← Teorema della fermata (Goedel)
- Andrey Kolmogorov '60, Chaitin

Algoritmo A, dati d'ingresso D → A(D) otteniamo un output

sono sequenze binarie

A(D) ma è "legale" A(A)

eseguo la seq. bin.

la stessa binaria è l'input

Q. (Turing '37): possiamo sempre stabilire se un programma termina?

HALTING PROBLEM

- esempio facile:

```
i = 0
while (i < 32) {
    i++
}
```

→ TERMINA

- esempio: congettura di Goldbach: $\forall n \geq 4 : n = p + q$ dove p, q primi (non necessariamente primi)

```
1  CongetturaGoldbach( ):
2    n = 2;
3    DO {
4      n = n + 2;
5      controesempio = TRUE;
6      FOR (p = 2; p <= n-2; p = p + 1) {
7        q = n - p;
8        IF (Primo(p) && Primo(q)) controesempio = FALSE
9      }
10   } WHILE (!controesempio);
11   RETURN n;
```

programma termina se la congettura è falsa

Turing ha dimostrato che non esiste un algoritmo / programma in grado di stabilire se A(D) termini o meno

Per assurdo: esiste algoritmo TERMINA(A,D) → SÌ se A(D) termina
→ NO altrimenti

- TERMINA(A,A) è "legale"
- nuovo programma PARADOSSO(X):
  while TERMINA(X,X) è "legale"
  j // istruzione vuota
- PARADOSSO(PARADOSSO) termina?
  - SÌ → TERMINA(PAR.,PAR.) = NO ⇒ PAR(PAR) non termina
  - NO ⇒ TERMINA(PAR,PAR.) = SÌ ⇒ PAR(PAR) termina

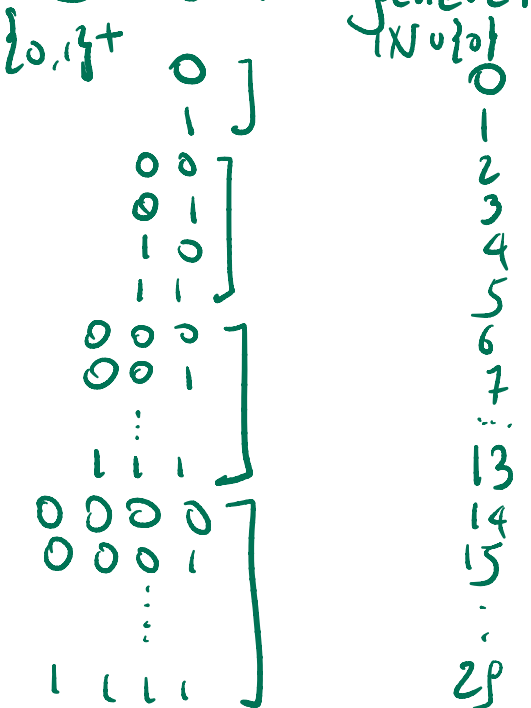
PAR(PAR) termina ⇔ PAR(PAR) non termina ... ASSURDO

Non tutti i problemi computazionali hanno un algoritmo/programma che li risolve

problema computazionale $\Pi: \{0,1\}^* \to \{0,1\}^*$

$\mathbb{N} \to \mathbb{N}$

BIGEZIONE: generazione lessicografica di $\{0,1\}^*$ // senza ε = seq. vuota



∃ algoritmo che genera tutte le seq. binarie in ordine lessicografico

Q1 $\#\Pi$? La cardinalità delle $f: \mathbb{N} \to \mathbb{N}$

Stessa cardinalità dei numeri reali

Q2. # alg./prg? La cardinalità delle sequenze binarie = cardinalità dei naturali $\Rightarrow$ stessa

Q1+Q2 → esistono problemi computazionali che non hanno algoritmi di risoluzione

Contributo di Turing: ha preso un problema reale, quello della fermata

Per esempio: equivalenza tra programmi è indecidibile

COS'È UNA SEQUENZA FINITA CASUALE?

KOLMOGOROV COMPLEXITY

$K(x)$

$x = 0101010101010101\,01\,01\,01\,01 \cdots 01$

```
for (i=0; i<n/2; i++)
    cout << '01';
cout << endl;
```

$\#bit = C + \log_2 n$

stiamo misurando quanto è lungo un programma

C = costante che dipende dal linguaggio di programmazione

n richiede $\log_2 n$ bit perché dobbiamo specificare la lunghezza della stringa

$K_L(x)$ = lunghezza (in bit) del programma nel linguaggio L che genera x (senza input)

$K_L(x) \geq c' + g|x|$ $K_L(x)$ e $K_{L'}(x)$ differiscono di una costante

$K(x) \geq |x| - c$ sse x è casuale

KOLMOGOROV COMPLEXITY

in altre parole, il modo più succinto di generare x è scriverla direttamente

Casuale $\leftrightarrow$ incomprimibile

- BUONA NOTIZIA: quasi tutte le sequenze binarie sono incomprimibili, quindi casuali
- CATTIVA NOTIZIA: data x, è indecidibile stabilire se $K(x) \geq |x| - c$

Fatto 1 $\forall n \quad \exists x \in \{0,1\}^n$ t.c. $K(x) \geq n$

$$S = \{ x \in \{0,1\}^n : K(n) < n \}$$

$$|S| = | A(y) = x \wedge |A(y)| < n |$$

$$0 \leq \quad \leq n-1$$

$$|S| = 2^0 + 2^1 + 2^2 + \cdots + 2^{n-1} = 2^n - 1$$

Fatto 2 $\forall n \; \forall c \geq 1$

$$\Pr_x(K(x) \geq n-c) > 1 - \frac{1}{2^c}$$

incomprimibile

$$S = \{ x \in \{0,1\}^n : K(n) < n-c \}$$

$$|S| = 2^0 + 2^1 + \cdots + 2^{n-c-1} = 2^{n-c} - 1$$

$$\frac{|S|}{\{0,1\}^n} = \frac{2^{n-c}-1}{2^n} = \frac{1}{2^c} - \frac{1}{2^n} < \frac{1}{2^c}$$

$$\Pr_x(K(x) \geq n-c) =$$

$$1 - \Pr_x(K(x) < n-c)$$

$$> 1 - \frac{1}{2^c}$$

E' indecidibile stabilire se $K(x) \geq |x| - c$

$R = \text{random} = \{x \in \{0,1\}^n : K(x) \geq |x| - c\}$

E' indecidibile l'appartenenza a R

Per assurdo, supponiamo esista A_R t.c. $A_R(x) = \text{SÌ}$ se $x \in R$

Costruiamo un algoritmo B che termina:

- genera $\{0,1\}^*$ in ordine lessicografico $<_L$
- per ogni x generata, se $A_R(x) = \text{SÌ}$ si ferma e stampa x

B_n è come B solo che genera $x \in \{0,1\}^n$

Sia B che B_n terminano con $x \in R$ per il FATTO 1

Sia x_n la stringa restituita da B_n

Equivalentemente x_n = è la $x \in R$, $|x| = n$, più piccola secondo $\leq_L$

$\emptyset$

$K(x_n) \geq |x_n| - c = n - c$

$x_n \in R$

$|B_n| \geq K(x_n)$ perché B_n genera x_n ma non è detto che sia il più corto

$c' + \log n = |B_n| \geq n - c$ ma ...

vera solo per un numero finito di valori di n

$\Rightarrow$ falsa asintoticamente per $n \to +\infty$

B_n è un algoritmo la cui codifica è $c' + \log_2 n$ per una qualche $c' > 0$

ASSURDO

A	2	-4	1	3	-7	-2	-1	2	-1	3	-2	-1	5
	0	1	2	3	4	5	6	7	8	9	10	11	12

$n = 13$

$A[i..j] = A[i]\, A[i+1] \cdots A[j]$ segmento

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

esempio $\text{somma}(3,6) = 3 - 7 - 2 - 1 = -7$

scopo: trovare $\max \text{somma}(i,j)$
$0 \le i \le j < n$

```
1  SommaMassima1( a ):   ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
2    max = 0;
3    FOR (i = 0; i < n; i = i+1) {
4      FOR (j = i; j < n; j = j+1) {
5        somma = 0;
6        FOR (k = i; k <= j; k = k+1)
7          somma = somma + a[k];
8        IF (somma > max) max = somma;
9      }
10   }
11   RETURN max;
```

$\sim n^3$ passi

$0 \le i \le j \le n-1$

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

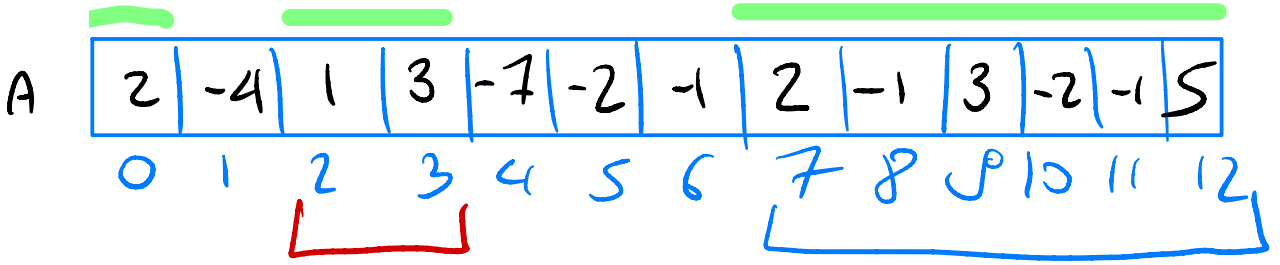
$somma(i,j+1) = somma(i,j) + A[j+1]$

BASELINE

```
1   SommaMassima2( a ): ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
2     max = 0;
3     FOR (i = 0; i < n; i = i+1) {
4       somma = 0;
5       FOR (j = i; j < n; j = j+1) {
6         somma = somma + a[j];
7         IF (somma > max) max = somma;
8       }
9     }
10    RETURN max;
```

$\sim n^2$ passi

← somma (i, j)



segmento massimale: non può essere esteso a sx o dx

$A[i,j]$ è massimale se $\nexists\ [i',j'] \supset [i,j) \wedge \underbrace{somma(i',j') > somma(i,j)}$

segmento massimo $\Rightarrow$ segmento massimale

Prop segmenti massimali sono disgiunti in A

$A[i,j)$ e $A[i',j')$ massimali $\Rightarrow [i,j) \cap [i',j') = \emptyset$

```
1  SommaMassima3( a ):   ⟨pre: a contiene n elementi di cui almeno uno positivo⟩
2    max = 0;
3    somma = max;
4    FOR (j = 0; j < n; j = j+1) {
5      IF (somma > 0) {
6        somma = somma + a[j];
7      } ELSE {
8        somma = a[j];
9      }
10     IF (somma > max) max = somma;
11   }
12   RETURN max;
```

somma = saldo

$\sim n$ passi

- correttezza
- efficienza

A = ordinato

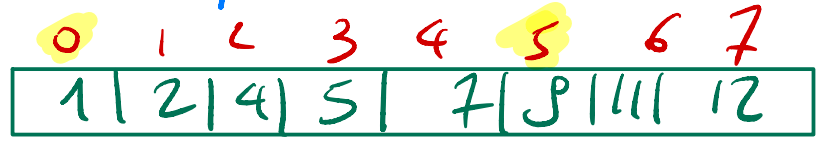
Scopo: dato un valore V, stabilire se esistono $i < j$ t.c. $A[i] + A[j] = V$

(istanza 3-SUM: $\exists\, i, j, k$ t.c. $A[i] + A[j] = A[k]$)

↳ non conosciamo un algoritmo subquadratico

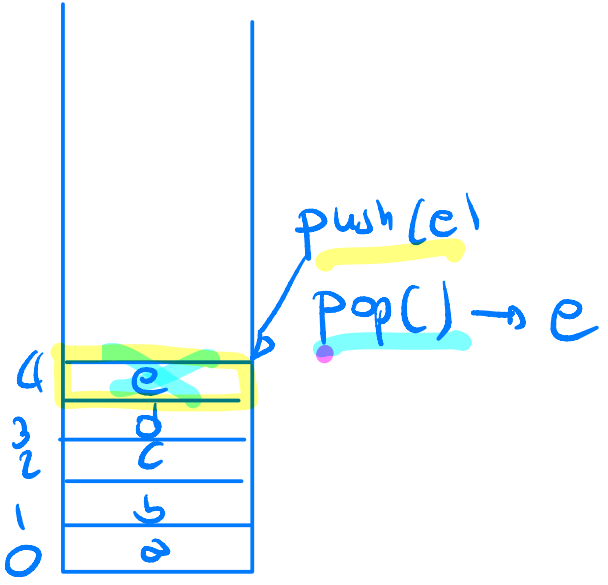
A non è ordinato

Esempio



V = 10

PILA / STACK : implementa la strategia LIFO = Last In First Out



push: a, b, c, d, e

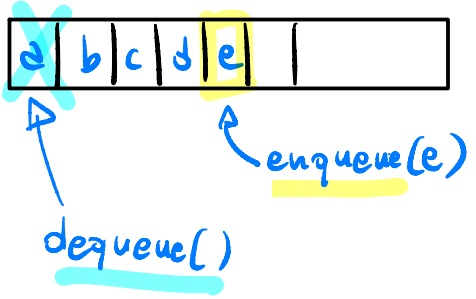
1

implementazioni:

- liste
- array
- vector ✔

- push(x): aggiunge x in cima alla pila
- pop(): restituisce l'elemento in cima alla pila e lo rimuove da essa

CODA / QUEUE

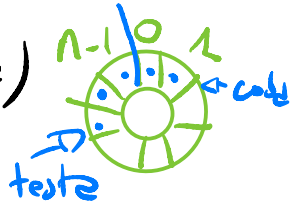


enqueue(x): mette x in fondo alla coda

dequeue(): restituisce l'elemento in testa alla coda e lo rimuove da essa

Implementazioni:

- liste
- array (coda circolare)
- vector ✔



CODA CON PRIORITA' (HEAP IMPLICITO)

- ogni elemento ha una priorità
- heap-min, heap-max
- enqueue, dequeue → miglior priorità

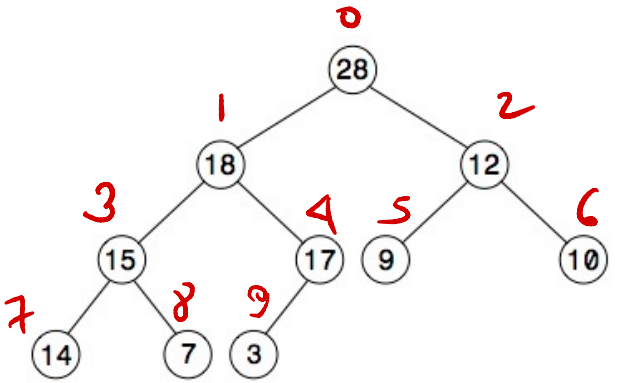
Per semplicità: heap-max

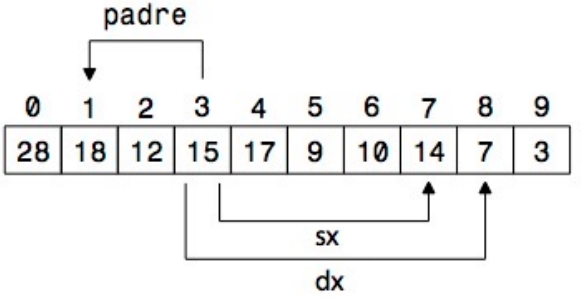
HEAP IMPLICITO: array "apparentemente" disordinato

HEAP IMPLICITO

vedi i=0

{ heap-max : padre > figli
{ heap-min : padre < figli





Nella nostra testa...
albero completo da sinistra
- ogni nodo interno ha due figli (tranne uno al più)
- le foglie sono su due livelli, di cui uno completo e l'altro con solo foglie tutte a sinistra

Nel computer è un array:

dato un nodo che corrisponde alla posizione i:

- figlio sx in posizione $2i+1$
- figlio dx in posizione $2i+1$
- $i > 0$: padre in posizione $\lfloor \frac{i-1}{2} \rfloor$

senza puntatori?

Spazio richiesto : n priorità + O(1) celle di memoria

altezza : lunghezza del percorso diretto radice-foglia più lungo

oss nodo i ha genitore posizione $\lfloor \frac{i-1}{2} \rfloor \leq \frac{i}{2}$

l'altezza è $O(\lg n)$ perchè la foglia n-1 dimezza il valore della posizione, ogni volta che "sale" verso la radice

max in heap-max
min in heap-min } posizione 0

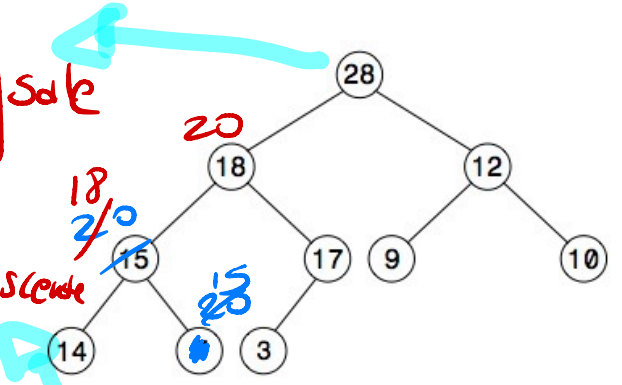
Posizione $\geq n$ → puntatore vuoto
posizione < 0

```
1  RiorganizzaHeap(i):   ⟨pre: heapArray è uno heap tranne che in posizione i⟩
2    WHILE (i>0 && heapArray[i].prio > heapArray[Padre(i)].prio) {
3      Scambia( i, Padre( i ) );
4      i = Padre( i );
5    }
6    WHILE (Sinistro(i) < heapSize && i != MigliorePadreFigli(i)) {
7      migliore = MigliorePadreFigli( i );
8      Scambia( i, migliore );
9      i = migliore;
10   }
```

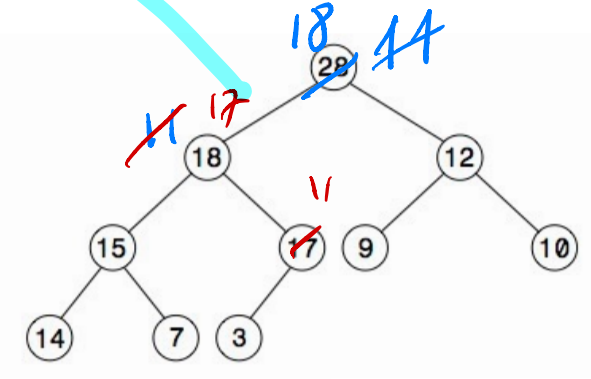
Sale

Scende

O(log n) tempo




```
1  MigliorePadreFigli(i):      ⟨pre: il nodo in posizione i ha almeno un figlio⟩
2    j = k = Sinistro(i);
3    IF (k+1 < heapSize) k = k+1;
4    IF (heapArray[k].prio > heapArray[j].prio) j = k;
5    IF (heapArray[i].prio >= heapArray[j].prio) j = i;
6    RETURN j;
```



Sinistro(i) = 2i+1

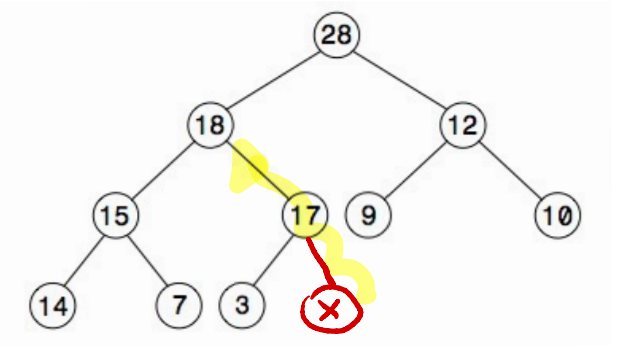
Destro(i) = 2i+1

Padre(i) = $\lfloor \frac{i-1}{2} \rfloor$

enqueue(x)

+

riorganizza Heap(n)



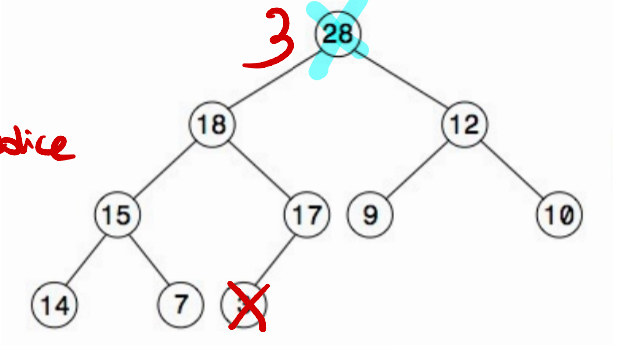
$O(\lg n)$

dequeue()

+

metti ultima foglia nella radice

riorganizza Heap(0)



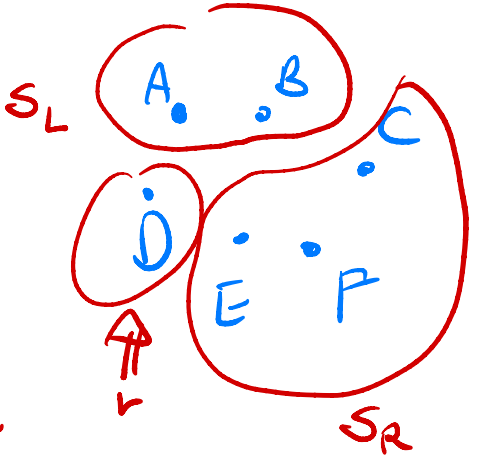
HEAP SORT: $O(n \log n)$ confronti / tempo ottimo

$O(1)$ spazio aggiuntivo ottimo

\\

Selection sort "furbo" dove il minimo di $A[i..n-1]$ viene trovato in tempo $\lg n$ invece che $n-i$

NATURA INDUTTIVA DEGLI ALBERI → BINARI



S = insieme di n elementi

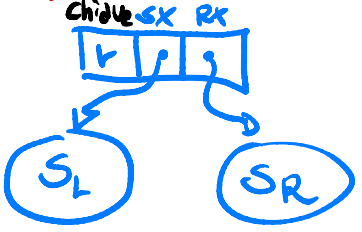
⇓

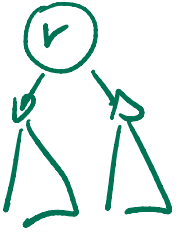
T = albero binario di n nodi

↳ 3-partizione → sicuramente c'è un singoletto chiamato radice

$S \to S_L \cup \{r\} \cup S_R$

T →





① $S = \phi \to T = NULL$ caso base

② S_L e/o S_R possono essere vuoti

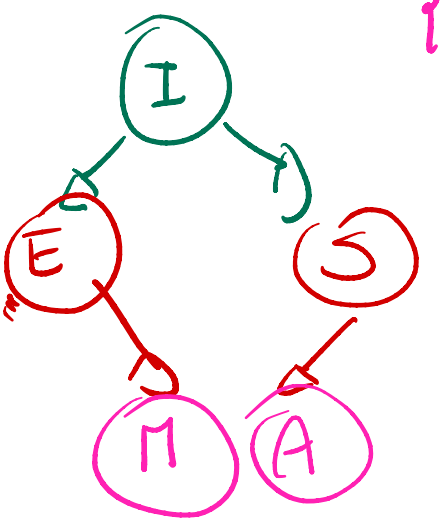
$S = \{E, S, A, M, I\} \rightarrow \{E, M\} \cup \{I\} \cup \{S, A\}$

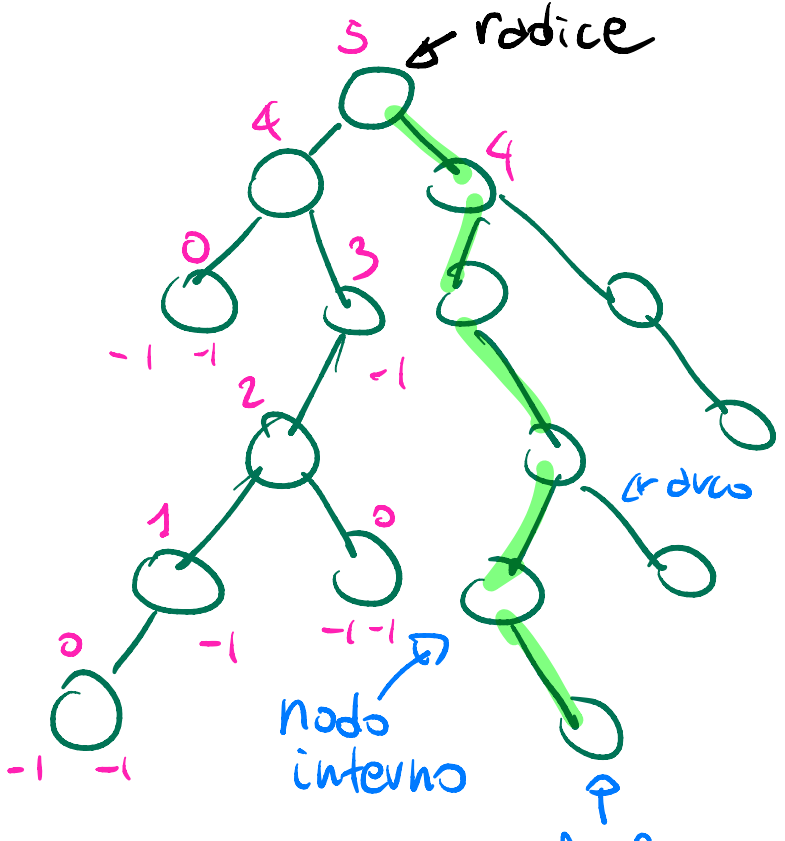
S_L S_R

$\{\} \{E\} \{M\}$ $\{A\} \{S\} \{\}$

$\{\} \{M\} \{\}$ $\{\} \{A\} \{\}$

$T =$

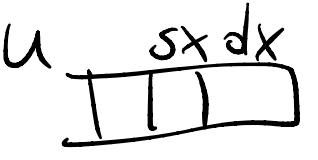




altezza = il massimo numero di archi attraversati in un percorso radice - foglia

```
altezza(u):
  if u = NULL then return -1
  return 1 + max(altezza(u.sx), altezza(u.dx))
```

foglia $\Leftrightarrow S_L = S_R = \emptyset$



```
1  Dimensione( u ):
2    IF (u == null) {
3      RETURN 0;
4    } ELSE {
5      dimensioneSX = Dimensione( u.sx );
6      dimensioneDX = Dimensione( u.dx );
7      RETURN dimensioneSX + dimensioneDX + 1;
8    }
```

divide-et-impera

lo problema decomponibile

NON DECIDIAMO LA DIVISIONE

← inizialmente u = radice

```
1  Decomponibile(u):
2    IF (u == null) {
3      RETURN Decomponibile(null);
4    } ELSE {
5      risultatoSX = Decomponibile(u.sx);
6      risultatoDx = Decomponibile(u.dx);
7      RETURN Ricombina(risultatoSX, risultatoDx);
8    }
```

caso base

schema "generico"

divide → sx, dx

impera

← ricombinazione

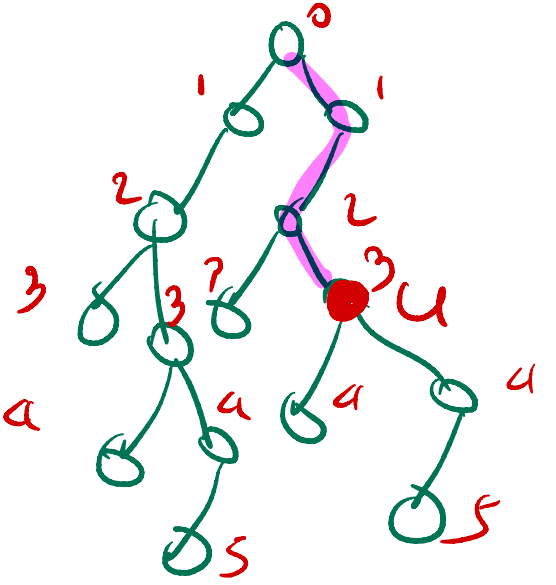
bottom-up

profondità(u) = # archi attraversati nel cammino dalla radice a u

top-down

traccia: aumentare # parametri nelle chiamate ricorsive

```
prof(u, d):        // inizialmente u = radice, d = 0
  if (u ≠ null) {
    u.profondità = d;   ← max - altezza
    prof(u.sx, d+1);
    prof(u.dx, d+1);
  }
```



livello = nodi con la stessa profondità

altezza(v) = $\max_u$ prof(u)

Relazione di ricorrenza

$$t(n) = \text{costante} + t(x) + t(n-x-1)$$

n nodi; $|S_L|$; $|S_R|$; non si conta la radice

Per induzione: $t(n) \leq c \cdot n$ per una certa costante

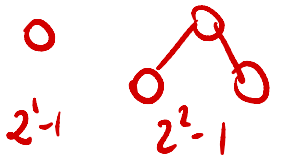
$\Rightarrow t(n) = O(n)$ tempo per i metodi visti finora

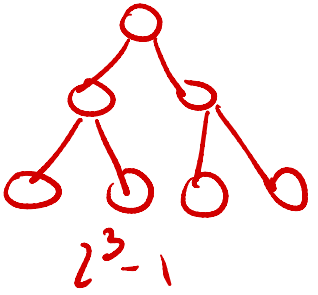
spazio aggiuntivo (stack size) $= O(\text{altezza}(v))$

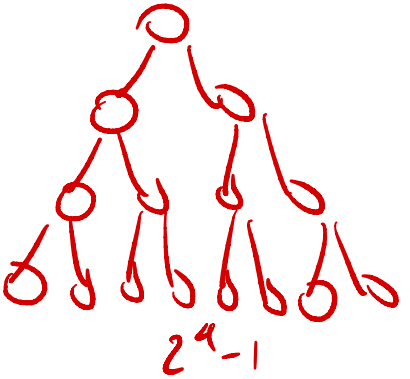
`ulimit -v` (Linux)
per togliere limiti allo stack

albero completamente bilanciato : massimo numero di nodi e foglie sullo stesso livello

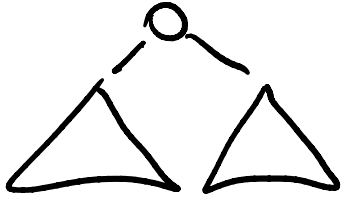
es.





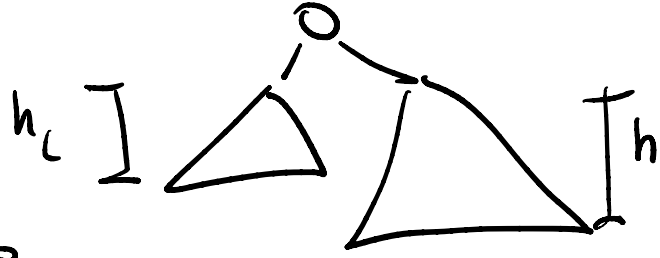


$n = 2^{h+1} - 1$



ok

booleano



ko !

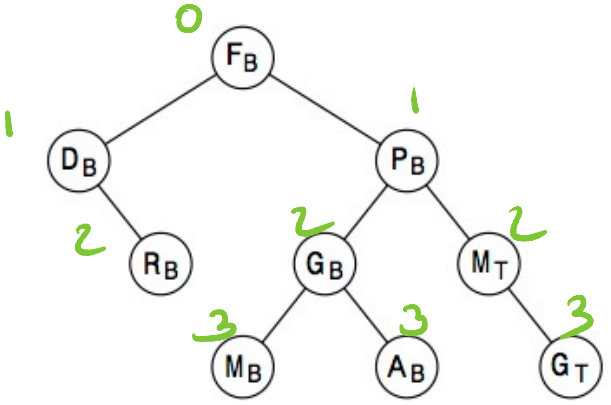
altezza

```
1  CompletamenteBilanciato( u ):
2    IF (u == null) {
3      RETURN <TRUE, -1>;
4    } ELSE {
5      <bilSX,altSX> = CompletamenteBilanciato( u.sx );
6      <bilDX,altDX> = CompletamenteBilanciato( u.dx );
7      completamenteBil = bilSX && bilDX && (altSX == altDX);
8      altezza = max(altSX, altDX) + 1;
9      RETURN <completamenteBil,altezza>;
10   }     ⟨post: restituisce TRUE come prima componente ⇔ T(u) è completamente
           bilanciato⟩
```

NON FARE 4 chiamate ricorsive!

non dimenticate queste

Regola: le informazioni ricevute dalle chiamate ricorsive nei figli, vanno aggiornate e propagate al padre



anticipata: chiamate ricorsive nell'albero di ricorsione

FB DB RB PB GB MB AB MT GT

simmetrica: chiavi ordinate in albero binario di ricerca

DB RB FB MB GB AB PB MT GT

posticipata: ordine di valutazione negli alberi sintattici

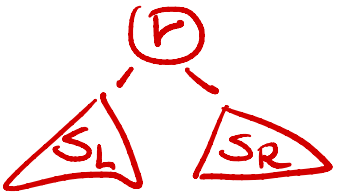
RB DB MB AB GB GT MT PB FB

ampiezza:

FB DB PB RB GB MT MB AB GT

0 1 2 3

si usa code: aspettiamo [ghosti]

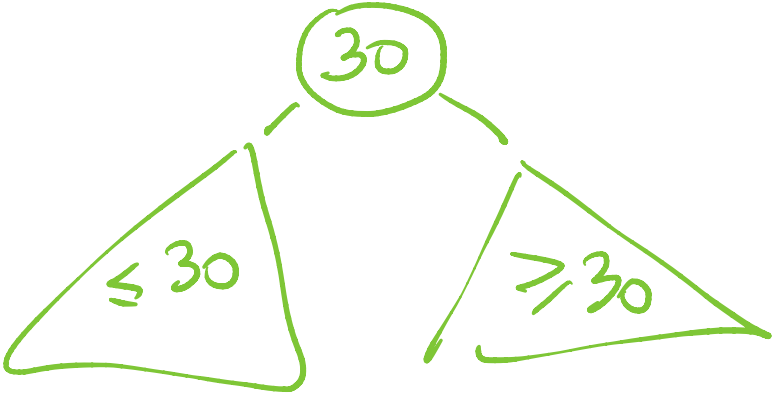



```
visita(u):
  if (u≠NULL) {
        cout << u.chiave
    visita(u.sx);
        cout << u.chiave
    visita(u.dx);
        cout << u.chiave
  }
```

ALBERO BINARIO DI RICERCA

visto finora

↳ chiavi (u.sx) ≤ chiave (u) ≤ chiavi (u.dx)



RICERCA BINARIA & ALBERO BINARIO DI RICERCA

- Array A ordinato
- elemento X

$\Big\}$ se X appare in A e in quale posizione

es. $X = G \Rightarrow$ posizione $= 5$

$X = H \Rightarrow$ non appare

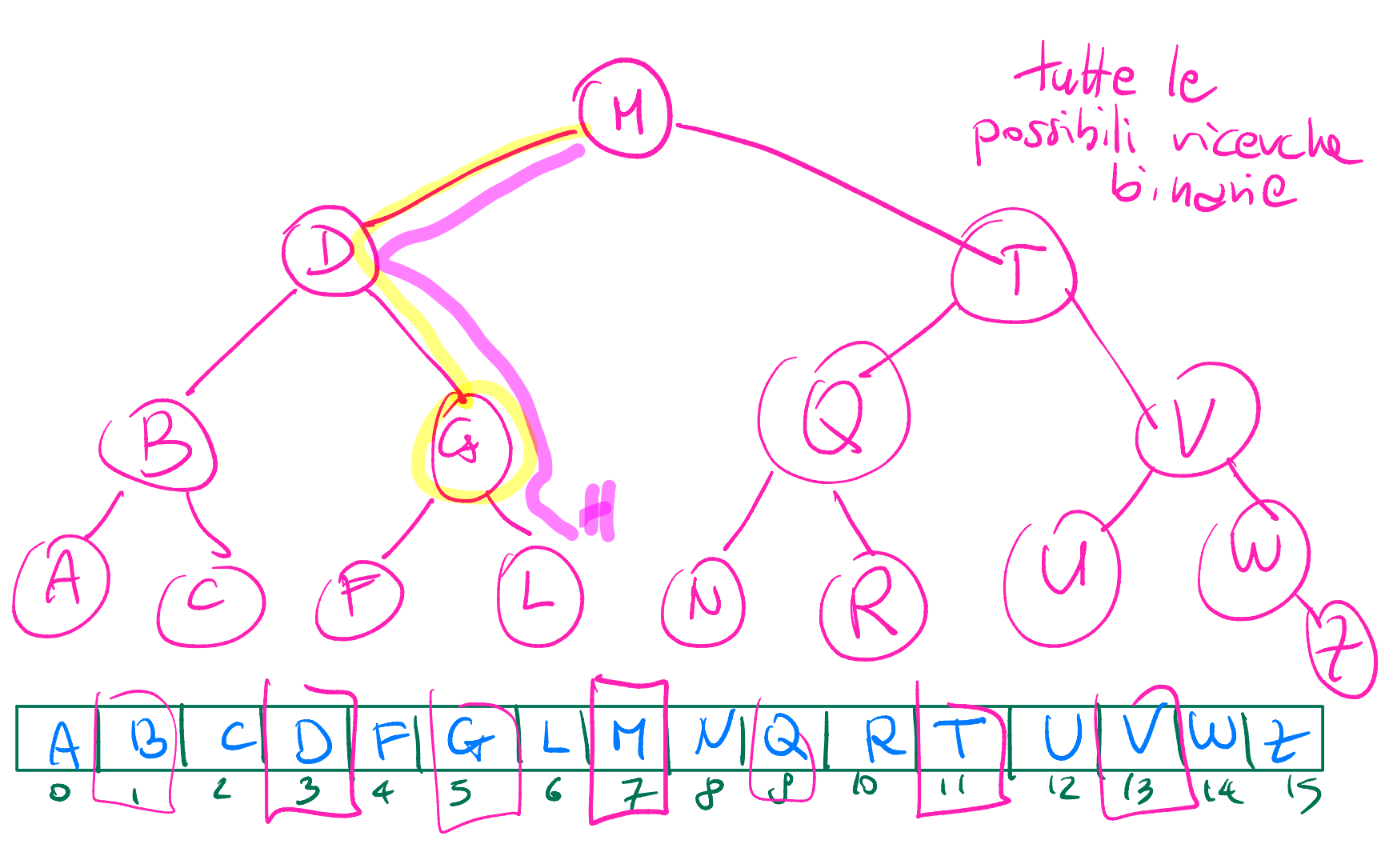
LIMITE SUPERIORE: prendendo l'elemento centrale dimezziamo $\Rightarrow O(\lg n)$ confronti

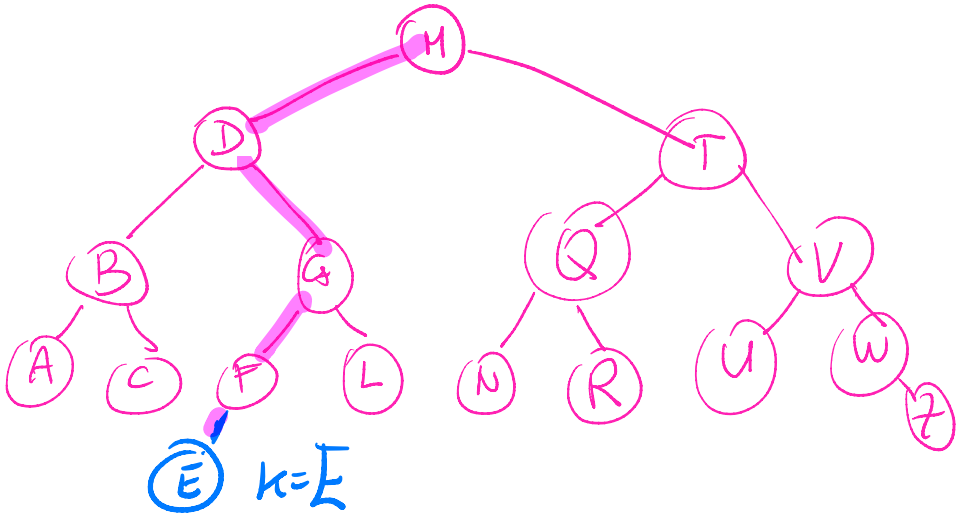
LIMITE INFERIORE

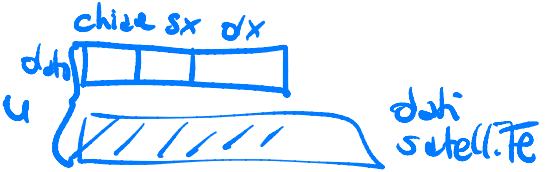
$3^t \geq n+1$ risposte

$\Rightarrow \Omega(\lg n)$ confronti

A	B	C	D	F	G	L	M	N	Q	R	T	U	V	W	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15


tutte le possibili ricerche binarie
M
D
T
B
G
Q
V
A
C
F
L
N
R
U
W
Z
A B C D F G L M N Q R T U V W Z
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15





ricorsione su u

cosa cerco

```
1  Ricerca( u, k ):
2    IF (u == null) RETURN null;
3    IF (k == u.dato.chiave) {
4      RETURN u.dato;
5    } ELSE IF (k < u.dato.chiave) {
6      RETURN Ricerca( u.sx, k );
7    } ELSE {
8      RETURN Ricerca( u.dx, k );
9    }
```

Caso BASE

trovato !

si può fare in maniera iterativa

vado a sx oppure a dx in base al confronto

O(h) tempo

h = altezza dell'albero

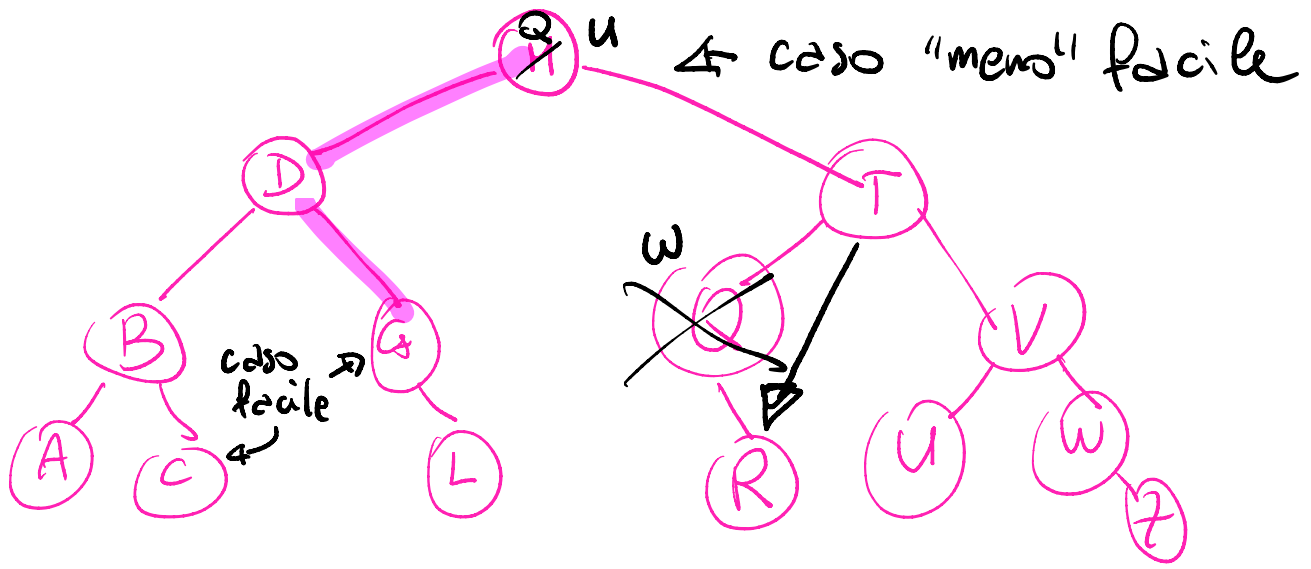
```
1  Ricerca( u, k ):
2    IF (u == null) RETURN null;
3    IF (k == u.dato.chiave) {
4      RETURN u.dato;
5    } ELSE IF (k < u.dato.chiave) {
6      RETURN Ricerca( u.sx, k );
7    } ELSE {
8      RETURN Ricerca( u.dx, k );
9    }
```

```
1  Inserisci( u, e ):        // non vengono inserite chiavi duplicate
2    IF (u == null) {
3      u = NuovoNodo();        new in C++  (malloc in C)
4      u.dato = e;
5      u.sx = u.dx = null;
6    } ELSE IF (e.chiave < u.dato.chiave) {
7      u.sx = Inserisci( u.sx, e );
8    } ELSE IF (e.chiave > u.dato.chiave) {
9      u.dx = Inserisci( u.dx, e );
10   }
11   RETURN u;                  ⟨post: se k appare già in u, non viene memorizzata⟩
```

Cancellazione di u

- caso facile: il nodo u è una foglia o ha un solo figlio
  È sostanzialmente simile alla cancellazione di un elemento di una lista

- caso "meno" facile: il nodo u ha due figli
  sia w = minimo del sottoalbero radicato in u.dx
  sostituisci la chiave di u con quella di w
  cancella fisicamente w perché è un caso facile
  costo ν $2h = O(h)$ tempo


Q
U
caso "meno" facile
D
T
B
G
caso facile
A
C
L
W
Q
V
R
U
W
Z

```
1  Cancella( u, k ):
2    IF (u != null) {
3      IF (u.dato.chiave == k) {
4        IF (u.sx == null) {
5          u = u.dx;
6        } ELSE IF (u.dx == null) {
7          u = u.sx;
8        } ELSE {
9          w = MinimoSottoAlbero( u.dx );
10         u.dato = w.dato; // w è adesso ridondante
11         u.dx = Cancella( u.dx, w.dato.chiave );
12       }
13     } ELSE IF (k < u.dato.chiave) {
14       u.sx = Cancella( u.sx, k );
15     } ELSE IF (k > u.dato.chiave) {
16       u.dx = Cancella( u.dx, k );
17     }
18   }
19   RETURN u;
```

casi facili

caso "meno" facile

questa chiamata ricorsiva sicuramente conduce a un caso facile

↳ simile all'inserimento

```
1  MinimoSottoAlbero( u ):
2    WHILE (u.sx != null)
3      u = u.sx;
4    RETURN u;
```

u ≠ NULL

⟨pre: u ≠ null⟩

non può avere 2 figli?

altezza h di un albero con n nodi

① $\lfloor \log_2 n \rfloor \leq h \leq n-1$ (essenzialmente una lista)

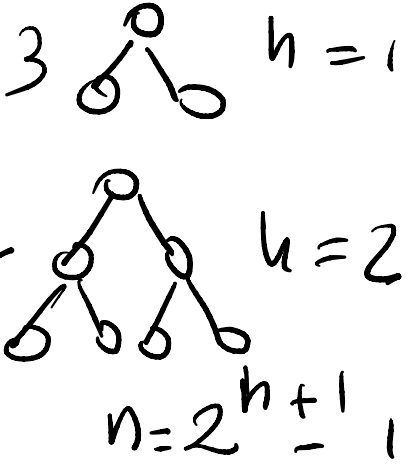
albero con n nodi e h minimo

$2^h \leq n < 2^{h+1}$

$\log_2 n < h+1$

$\lfloor \log_2 n \rfloor < h$

$\lfloor \log_2 n \rfloor \leq h$



Un albero binario è bilanciato se $h = O(\log n)$

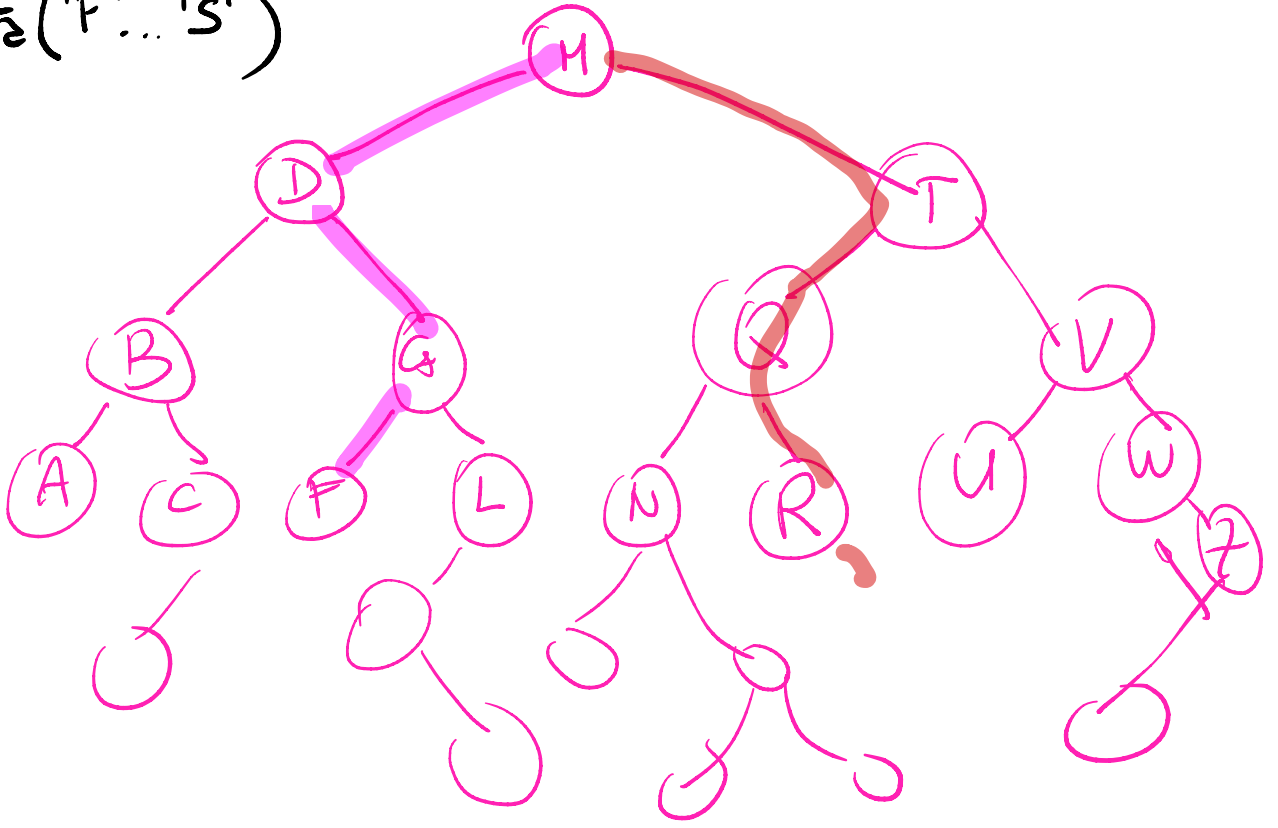
Spunto di riflessione:

Prendiamo un array permutato uniformemente a caso

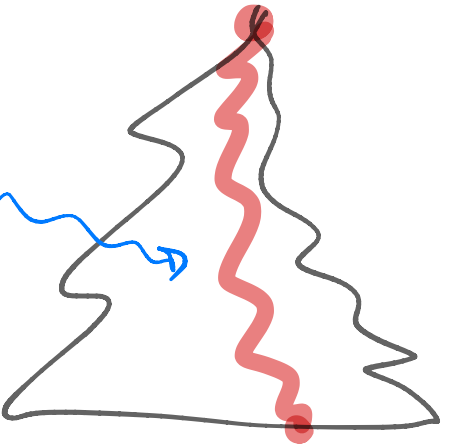
Riusando l'analisi del QS randomizzato, mostrare che l'albero binario di ricerca, in cui le chiavi sono appunto inserite nell'ordine della suddetta permutazione, ha altezza in media $O(\lg n)$

Prossimo lab...

- cerca('F')
- conta('F'...'S')



Alberi: costo computazionale è speso lineare nell'altezza h



Bilanciato: $h = O(\lg n)$

1) Caso medio randomizzato: se le chiavi sono inserite in un albero vuoto, in ordine casuale

collegamento con il QS randomizzato:

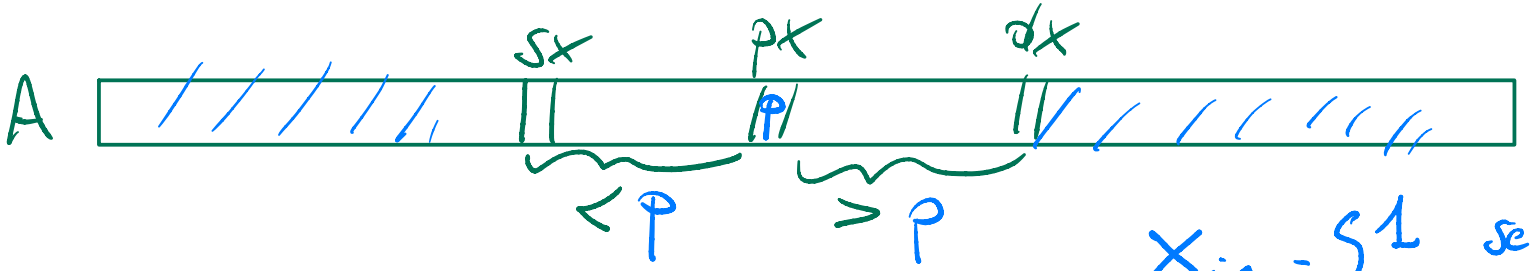
pivot → radice

chiavi < pivot → sottoalbero sx

chiavi > pivot → sottoalbero dx

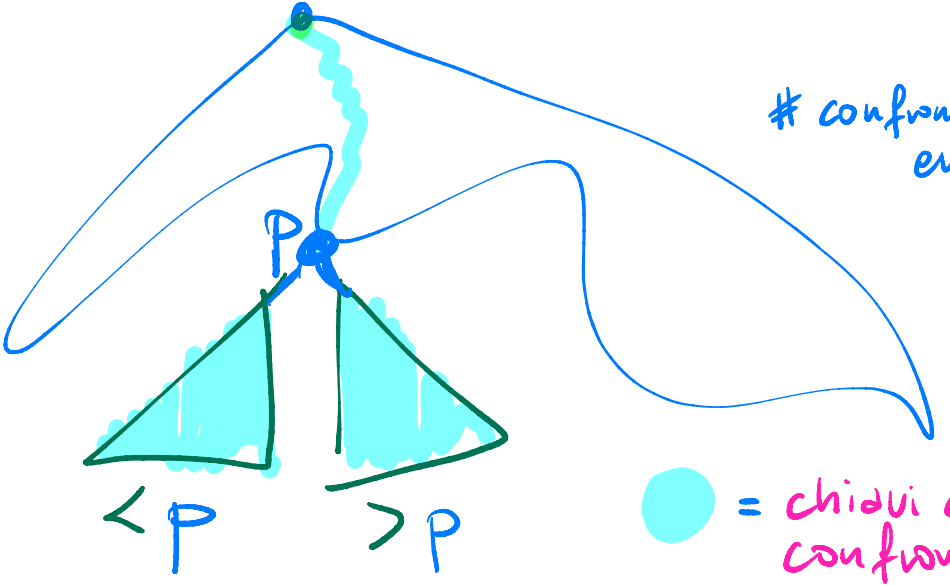
} ricorsivamente

QS randomizzato



$$X_{ij} = \begin{cases} 1 & \text{se } z_i : z_j \\ 0 & \text{altrimenti} \end{cases}$$

confronti nel QS
$$\text{ers} \sum_{i<j} X_{ij}$$



= chiavi con cui p è stato confrontato

Ricordiamo che $E[X_{ij}] = Pr[X_{ij} = 1] \leq \frac{2}{j-i+1}$

$z_1 < z_2 < \cdots < z_n$

Sia $z_i = p \Rightarrow Y = \sum_j X_{ij}$ indica esattamente il numero di chiavi con cui p è stato confrontato

Se p è una foglia, Y è la sua altezza nell'esperimento

$$E[\text{altezza}] = E[Y] = \sum_j E[X_{ij}] = \sum_j \frac{2}{|j-i+1|} = O(\ln n)$$

Cresce come serie armonica

2) $h = O(\log n)$ al caso pessimo → tante soluzioni

AVL

Def 1-bilanciato

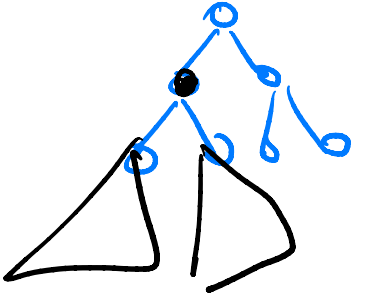
$h(x)$ = altezza del nodo x = max lunghezza di un cammino minimo da x a una delle sue foglie

$h(x) = \max(h(u.sx), h(u.dx)) + 1$

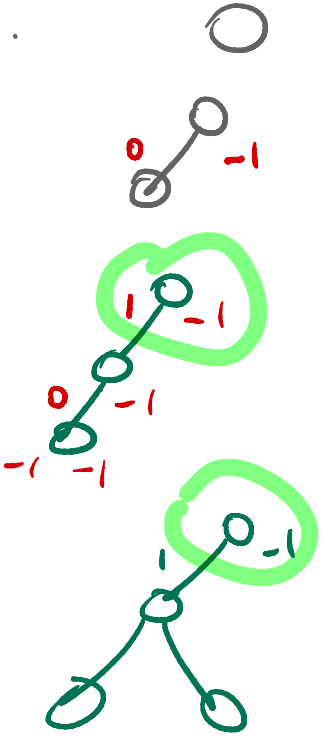
$h(NULL) = -1$

Albero è 1-bilanciato se

per ogni nodo u: $|h(u.sx) - h(u.dx)| \leq 1$



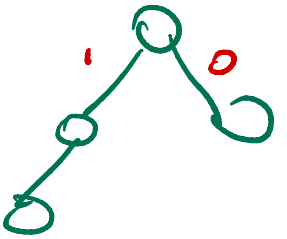
esempio .



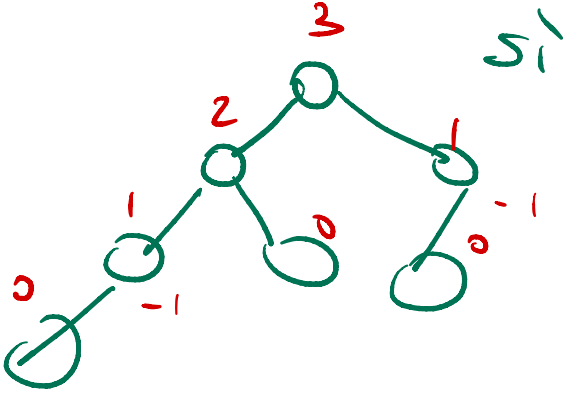
sì

sì

NO



sì



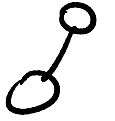
sì

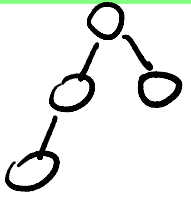
Prop Albero 1-bilanciato di n nodi $\Rightarrow h = O(\lg n)$

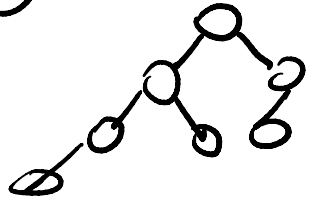
Alberi di Fibonacci (per induzione su h)

Fib_h

$h = 0$

$h = 1$ 

$h = 2$ 

$h = 3$ 

n_h

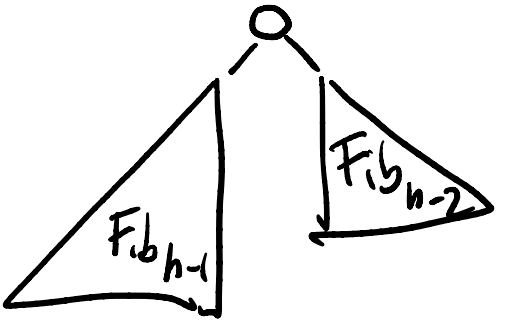
1

2

4

7

$h \geq 2$



$n_h = 1 + n_{h-1} + n_{h-2}$

Per costruzione e induzione, togliendo un nodo da Fib_h o si riduce l'altezza da h a $h-1$, oppure si viola la condizione di 1-bilanciato

↳ **Fatto** Dato $h \geq 0$, nessun albero 1-bilanciato di altezza h può avere meno di n_h nodi

h	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
n_h	1	2	4	7	12	20	33	54	88	143	232	376	609	986	1596
F_h	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377

numeri di Fibonacci

relazione tra n_h e F_h :

per induzione: $n_h = F_{h+3} - 1 \geq c^h$ per una qualche costante $c > 0$ collegata al rapporto aureo

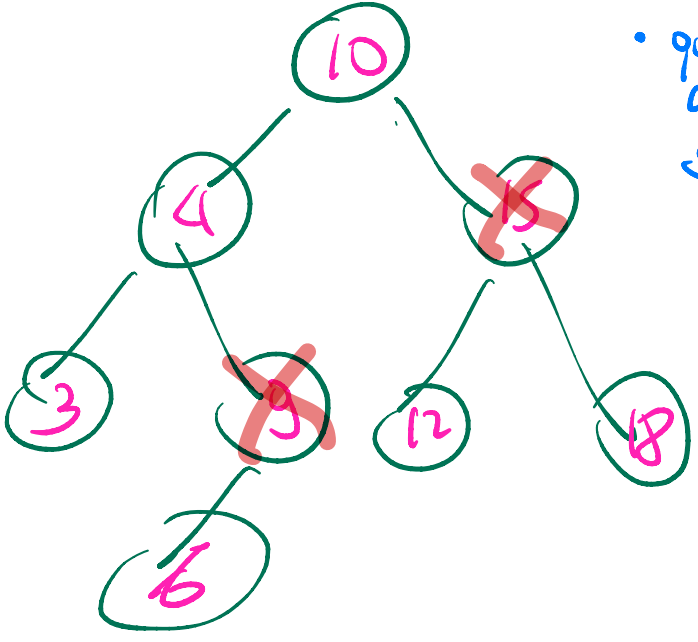
Prop Ogni albero 1-bilanciato con n nodi e altezza h soddisfa $h = O(\lg n)$

▶ $n \geq n_h \geq c^h \Rightarrow h = O(\lg n)$

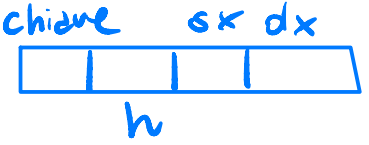
↑ per def. di n_h

Operazioni su AVL

- ricerca: stessa degli alberi binari di ricerca $O(h) = O(\lg n)$
- cancellazione: in modo logico
  - metti una marca di cancellazione sul nodo
  - quando il n.ro di marche è una frazione costante di n, ricostruisci l'albero dell'inizio solo con le chiavi non marcate

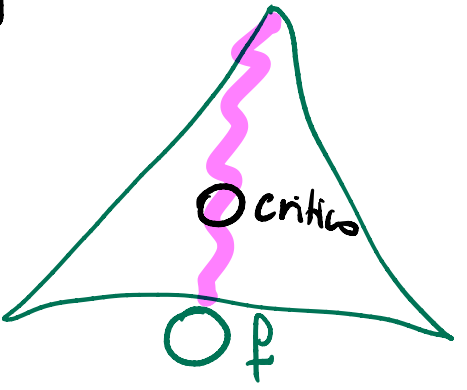


inserimento: crea sempre una nuova foglia



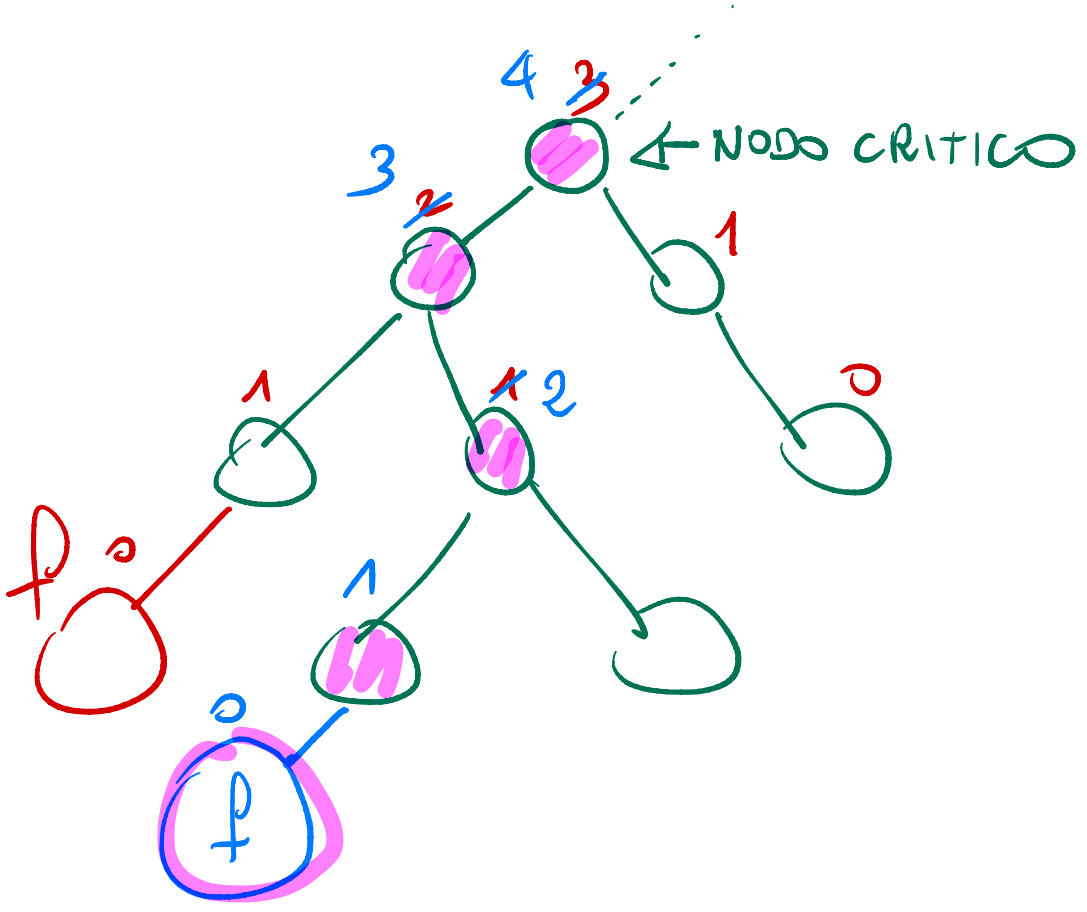

```
altezza(u):
  if (u = NULL) return -1;
  return u.h;
```

O(1) tempo



Def u = nodo critico se

1) è antenato di f
2) $|h(u.sx) - h(u.dx)| = 2$
3) è il nodo più vicino a f che soddisfa 1) + 2)


4
3
NODO CRITICO
3
2
1
1
1
2
0
f
0
1
0
f

```
1  Inserisci( u, e ):
2    IF (u == null) {
3      RETURN f = NuovaFoglia( e );
4    } ELSE IF (e.chiave < u.dato.chiave) {
5      u.sx = Inserisci( u.sx, e );
6    IF (Altezza(u.sx) - Altezza(u.dx) == 2) {
7      IF (e.chiave>u.sx.dato.chiave) u.sx=RuotaAntiOraria(u.sx);
8      u = RuotaOraria( u );
9      }
10   } ELSE IF (e.chiave > u.dato.chiave) {
11     u.dx = Inserisci( u.dx, e );
12     IF (Altezza(u.dx) - Altezza(u.sx) == 2) {
13     IF (e.chiave < u.dx.dato.chiave) u.dx = RuotaOraria(u.dx);
14     u = RuotaAntiOraria( u );
15     }
16   }
17   u.altezza = max( Altezza(u.sx), Altezza(u.dx) ) + 1;
18   RETURN u;
```

SD identiche a prima

Ss/sD

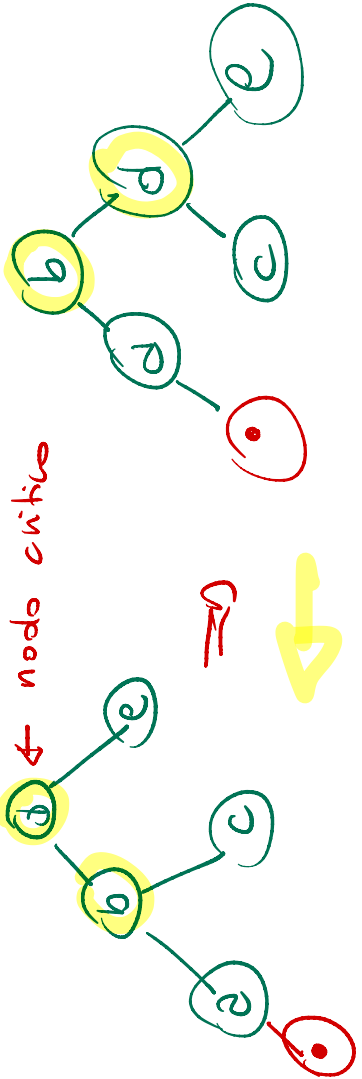
aggiorniamo l'altezza

```
1  Altezza( u ):
2    IF (u == null) {
3      RETURN -1;
4    } ELSE {
5      RETURN u.altezza;
6    }
```

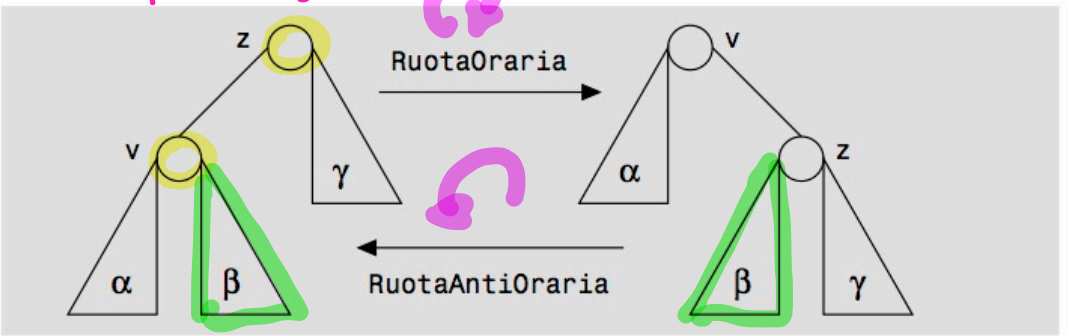
```
1  NuovaFoglia( e ):
2    u = NuovoNodo();
3    u.dato = e;
4    u.altezza = 0;
5    u.sx = u.dx = null;
6    RETURN u;
```

$O(h) = O(\log n)$

tempo



$\alpha < v < \beta < z < \gamma$

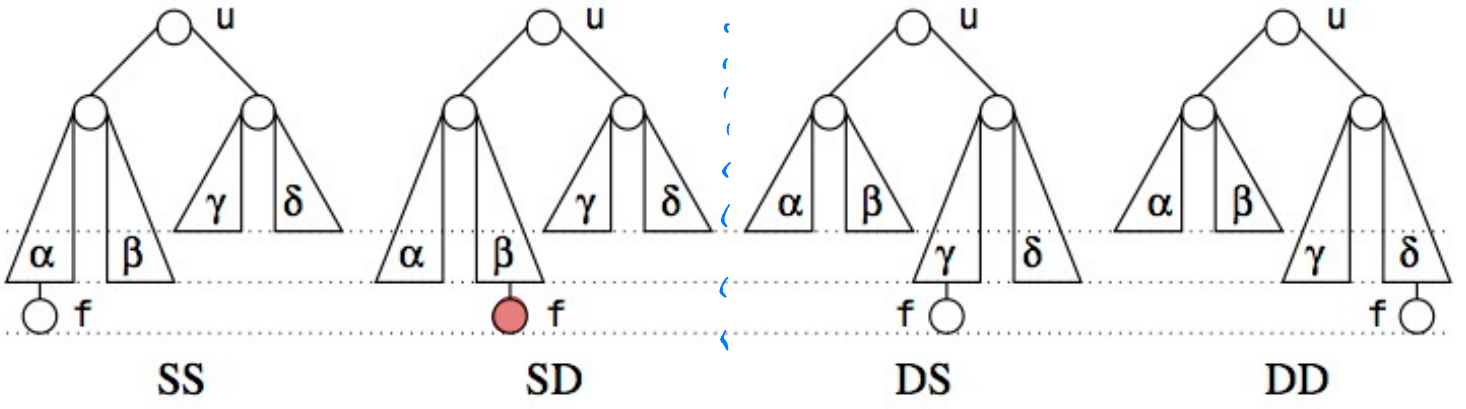


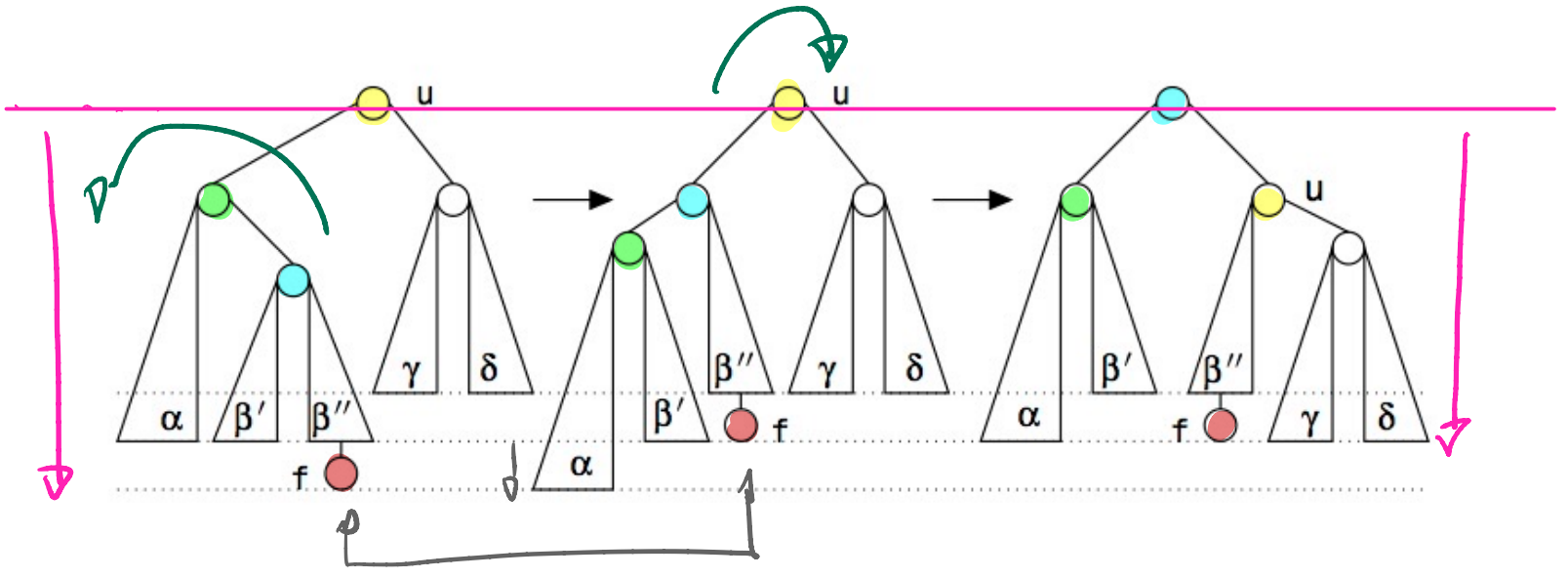

```
1  RuotaOraria( z ):
2    v = z.sx;
3    z.sx = v.dx;
4    v.dx = z;
5    z.altezza = max( Altezza(z.sx), Altezza(z.dx) ) + 1;
6    v.altezza = max( Altezza(v.sx), Altezza(v.dx) ) + 1;
7    RETURN v;
```

```
1  RuotaAntiOraria( v ):
2    z = v.dx;
3    v.dx = z.sx;
4    z.sx = v;
5    v.altezza = max( Altezza(v.sx), Altezza(v.dx) ) + 1;
6    z.altezza = max( Altezza(z.sx), Altezza(z.dx) ) + 1;
7    RETURN z;
```

$O(1)$ tempo

u = nodo critico f = foglia appena inserita





Caso SD: l'altezza del sottoalbero è la stessa di quella antecedente la doppia rotazione

PROBLEMA DEL DIZIONARIO

U = universo delle chiavi

$S \subseteq U$ insieme delle chiavi memorizzate nel computer

(classe C++: map)

- appartenenza (membership, find)
  $x \in U : x \stackrel{?}{\in} S$
- inserimento $S \leftarrow S \cup \{x\}$
  (insert/add)
- cancellazione $S \leftarrow S \setminus \{x\}$
  (delete/remove)

] STATICO

DINAMICO

Se U è anche ordinato:

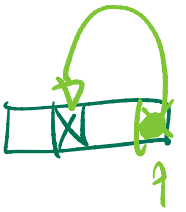
- $pred(x) = \max\{y \in S : y \leq x\}$
- $succ(x) = \min\{y \in S : y \geq x\}$

$x \in S \Rightarrow pred(x) = succ(x) = x$

- $intervallo(a,b) = \{y \in S : a \leq y \leq b\}$
- $count(a,b) = |intervallo(a,b)|$

Dizionario per $n = |S|$ chiavi

Struttura di dati	find	insert	delete
ARRAY NON ORDINATO	$O(n)$	$O(1)$	$O(1)$
ARRAY ORDINATO	$O(\lg n)$	$O(n)$	$O(n)$
LISTA NON ORDINATA	$O(n)$	$O(1)$	$O(1)$ / $O(n)$
LISTA ORDINATA	$O(n)$	$O(n)$	$O(1)$ / $O(n)$
HEAP	$O(n)$	$O(\lg n)$	$O(\lg n)$
A.B.R altezza $\lg n \le h < n$	$O(h)$	$O(h)$	$O(h)$
A.V.L	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
TABELLE HASH	$O(1)$ caso medio	$O(1)$ caso medio	$O(1)$ caso medio



in media $h = O(\lg n)$

HASH

parametro m = dimensione del codominio

funzione hash $h : U \to [m]$

osservazione ogni elemento di U è visto come una sequenza di bit, quindi un piuttosto numero (naturale) che viene "triturato" e quindi trasformato in un piccolo numero in $[m]$.

Pigeonhole $\Rightarrow$ poiché $|U| >> m$, $\exists$ sempre $k_1, k_2 \in U$

$k_1 \neq k_2$ ma $h(k_1) = h(k_2)$ COLLISIONE

OSSERVAZIONE Non si opera mediante confronti, ma si usa il risultato di h

Esempi di f. hash: MD5, SHA-1, SHA-256 per uso crittografico: offrono ulteriori garanzie che però non usiamo nei dizionari

Noi usiamo funzioni più semplici

$h(x) = x \,\%\, m$

Esempio più "moderno": HASH UNIVERSALE

p primo, $p \in [m+1, 2m]$, uniforme e casuale $a \in \mathbb{Z}_p^*$ e $b \in \mathbb{Z}_p$

$$h_{a,b}(x) = (a \cdot x + b) \,\%\, p \,\%\, m$$

prop $\Pr_{a,b}\left[h_{ab}(k_1) = h_{ab}(k_2)\right] = \frac{1}{m}$

CHIAVI LUNGHE → tecnica del ripiegamento (FOLDING)

$X \to x_1 \cdot x_2 \cdot \ldots \cdot x_h$, x_i ots in 64 bit

$$h(x) = \overset{h}{\underset{i=1}{\text{XOR}}}\ h'(x_i)$$

$h' : U' \to [m] \Rightarrow h : U \to [m]$

Nel caso particolare delle stringhe, spesso si usa

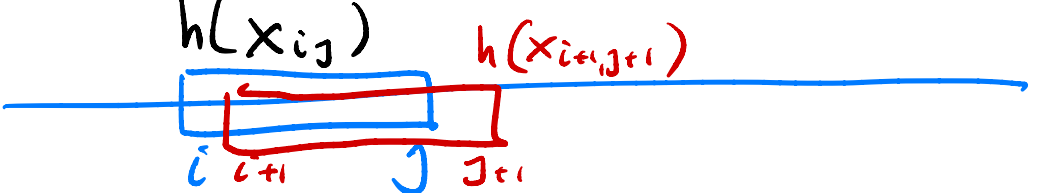
$X = d_0\, d_1 \ldots d_{n-1}$

$$h(x) = \sum_{i=0}^{n-1} (d_i \cdot \sigma^i) \,\%\, m$$

dove σ = la dimensione dell'alfabeto

(per ascii in pratica si usa $\sigma = 131$)

→ ROLLING



$$h(x_{i+1,j+1}) = \left(\left(h(x_{ij}) - a_i \alpha^{n-1} \right) \cdot \alpha + a_{j+1} \right) \% m$$

richiede $O(1)$ tempo calcolare l'hash successivo anche se le stringhe sono arbitrariamente lunghe (Karp-Rabin fingerprint, dove m = numero primo scelto casualmente)

TABELLA HASH : - funzione hash
- gestione delle collisioni

(nota fissato S, è possibile trovare una f. hash senza collisioni su S, sia chiama hash)

collisione → LISTA DI TRABOCCO
collisione → INDIRIZZAMENTO
collisione → ? cuckoo hashing

HASHING

Universal hash $h_{a,b}(x) = ((a \cdot x + b) \,\%\, p) \,\%\, m$

$a \in \mathbb{Z}_p^*$ $b \in \mathbb{Z}_p$ p primo $[m+1 \,..\, 2m-1]$

$U \to [m]$

$Pr(\text{collisione}) = \frac{1}{m}$

> Hash non mantiene l'ordine tra le chiavi

Tabella hash

- f. hash
- gestione collisioni
  - liste concatenate (liste di trabocco)
  - indirizzamento aperto
  - cuckoo hashing

x ∈ S	B	A	D	C	F	G	E	H	n = 8
h(x)	3	2	3	2	0	10	1	3	m = 11

LISTE CONCATENATE / TRABOCCO

insert(x):

```
T[h(x)].push_back(x)
```

find(x):

scan $T[h(x)]$

delete(x):

find i s.t. $T[h(x)][i] == x$

```
swap(T[h(x)][i], T[h(x)][last])
T[h(x)].pop_back();
```

Il metodo funziona bene se la f. hash distribuisce uniformemente le chiavi in $[m]$, cosa che avviene con h. universale

$\alpha = \frac{n}{m}$ = fattore di carico

α è la lunghezza media di una lista/vector

costo medio delle operazioni è $O(1+\alpha)$ tempo
(il caso pessimo è chiaramente $O(n)$)

$m \sim 2n \Rightarrow \alpha = \frac{1}{2} \Rightarrow O(1)$ medio

x ∈ S	B	A	D	C	F	G	~~E~~	H
h(x)	3	2	3	2	0	10	~~1~~	3

n = 8

m = 11

INDIRIZZAMENTO APERTO $m > n$

Scansione lineare (modulo m): i_{j+1}

T

0	1	2	3	4	5	6	7	8	9	10
F		A	B	D	C	H				G

cluster

cluster = unione di alcune liste di trabocco

inserimento: trova il primo posto libero (in modo circolare)

ricerca: parti dalla posizione h(x) in T e cerca in modo circolare finché: (a) trovi x oppure (b) raggiungi un posto libero

cancellazione: conviene marcare una chiave come cancellata

CANCELLAZIONE LOGICA

$\alpha = \frac{n}{m}$ costo $\left((1-\alpha)^{-1}\right)$ tempo $m \sim 2n \Rightarrow \alpha = \frac{1}{2}$

$\Rightarrow O(1)$ tempo

Poiché usiamo h. universale $p = \frac{n}{m}$ è la probabilità di trovare una posizione occupata

$\tilde{T}(n,m)$ = n.ro medio di posizioni esaminate per inserire una chiave in una tabella di m posizioni, di cui n sono occupate $\quad m > n$

$$\tilde{T}(n,m) = p\left(1 + \tilde{T}(n-1, m-1)\right) + (1-p)\cdot 1$$

$$\tilde{T}(0,m) = 1$$

$$\Rightarrow \tilde{T}(n,m) = 1 + \frac{n}{m} \cdot T(n-1, m-1)$$

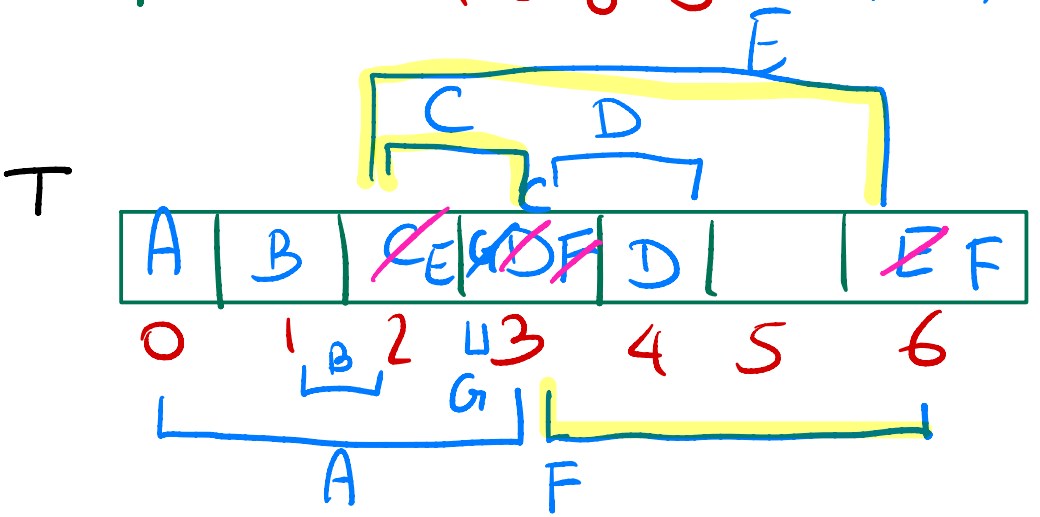
per sostituzione potete verificare che

$$\tilde{T}(n,m) \leq \frac{m}{m-n} = (1-\alpha)^{-1}$$

Cuckoo hashing: $O(1)$ tempo al caso pessimo { ricerca, cancellazione

$O(1)$ tempo medio ammortizzato per inserimento

$x \in U$	A	B	C	D	E	F	G	$n = 6$
$h_1(x)$	3	1	2	3	2	3	3	$m = 7$
$h_2(x)$	0	2	3	4	6	6	3	$m = 7$



Ricerca (x):

verifica se $T[h_1(x)] == x$ oppure $T[h_2(x)] == x$ $O(1)$ tempo caso pessimo

Cancellazione (x)

come ricerca, solo che sostituisce x con vuoto in T

Inserimento (x):

1. se $T[h_1(x)] == x$ oppure $T[h_2(x)] == x$, esci
2. se $T[h_1(x)] ==$ vuota oppure $T[h_2(x)] ==$ vuota, inserisci x, esci
3. $i = h_1(x)$ oppure $i = h_2(x)$

   while $T[i] \neq$ vuota do // ripeti al più n+1 volte → ciclo

   inserisci x in $T[i]$ e chiamiamo x la chiave "cacciata" da $T[i]$

   $i \leftarrow \{h_1(x), h_2(x)\} \setminus \{i\}$

   se while interrotto per n+1 iterazioni → REHASH

REHASH:

1. svuota la tabella T
2. sceglie in modo random h_1, h_2
3. riparte a inserire S con le nuove h_1 e h_2

Analisi dell'inserimento randomizzato

3 casi

① T ha una cella vuota $\Rightarrow O(1)$ tempo

② $T[h_1(x)] \neq$ vuota e attraversiamo un "cammino" di posizioni $h_1(x), i', i'', \ldots, j$

$T[j]$ era vuota prima di inserire x

lunghezza $\ell \Rightarrow \ell+1$ posizioni $O(1+\ell)$ tempo

③ come ②, solo che j non è mai vuota $\Rightarrow$ REHASH

(2) lunghezza ℓ $\quad m > 2n \cdot c$ per costante $c > 2$

Per induzione su ℓ: $\Pr\left(h_i(x) \underset{\ell}{\text{min}}\right] \leq \frac{1}{c^{\ell} m}$ (*)

lunghezza media: $\sum_{\ell \geq 1} \ell \cdot \frac{1}{c^{\ell} m} < \underbrace{\frac{c}{(c-1)^2}}_{O(1)} \cdot \frac{1}{m}$

Knuth, Concrete mathematics p. 33

costo medio (2) $O(1 + \text{lunghezza media}) = O(1)$

(3) REHASHING

Union bound

$$Pr(\text{REHASHING}) \le Pr(\text{esiste un ciclo}) \le \sum_{i=0}^{m-1} Pr(i \bmod j = i)$$

$$\le \sum_{i=0}^{m-1} \sum_{\ell \ge 1} Pr[i \bmod j^{\ell} = i] \le \frac{1}{m} \sum_{i=0}^{m-1} \sum_{\ell \ge 1} \frac{1}{c^{\ell}} < \frac{1}{m} \sum_{i=0}^{m-1} \frac{1}{c-1} = p$$

(*) $\le \frac{1}{c^{\ell} m}$

$\sum_{\ell \ge 1} \frac{1}{c^{\ell}} < \frac{1}{c-1}$

#REHASHING	
1	p
2	p^2
⋮	
t	p^t

#medio di rehashing

$$\sum_t t\, p^t = O(1)$$

$p < 1$ poiché $c > 2$

GRAFI = RELAZIONE BINARIA

$G = (V, E)$ V = insieme di nodi o vertici

$E \subseteq V \times V$ $n = |V|$ $m = |E|$

input.txt

```
8 11
0 1
0 6
...
6 7
```

n m

x y

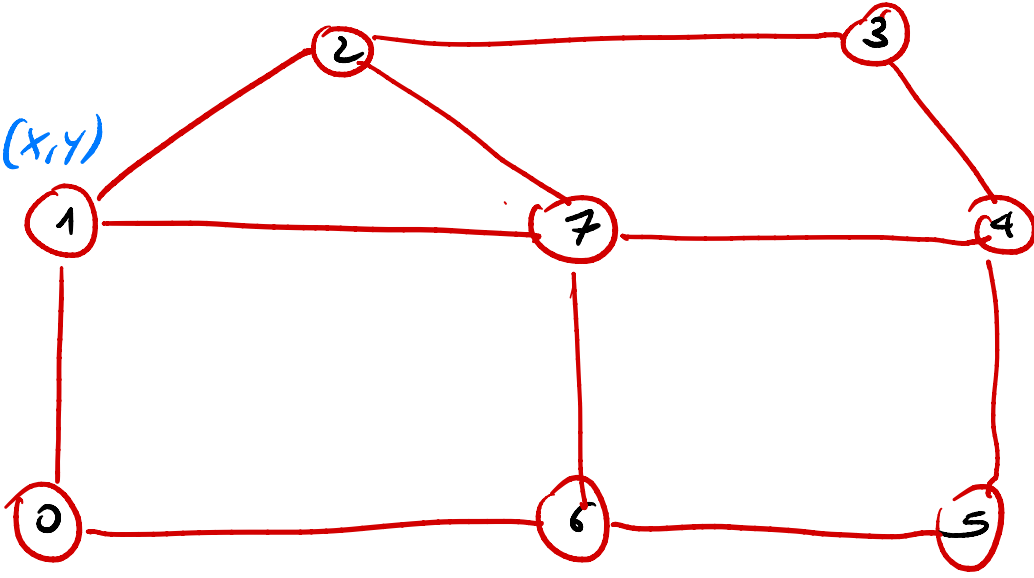
x y

x y

$\Rightarrow$ arco (x, y)

$0 \leq m \leq \binom{n}{2}$

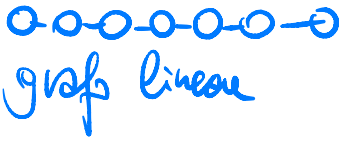
$dim(G) = n + m$



$V = \{0, 1, \dots, 7\}$ $E = \{(0,1), (0,6), (1,2), (1,7), (2,3), (2,7), (3,4), \dots, (6,7)\}$

Grafo: relazione binaria che ha come casi particolari

- rel. sequenziale $a_0 a_1 a_2 \dots a_{n-1}$

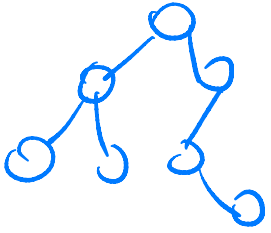


grafo lineare

(array, vector) $a_i < a_j$ sse $i < j$

- ordine parziale e gerarchie: $a_i < a_j$ sse

(alberi) i antenato di j

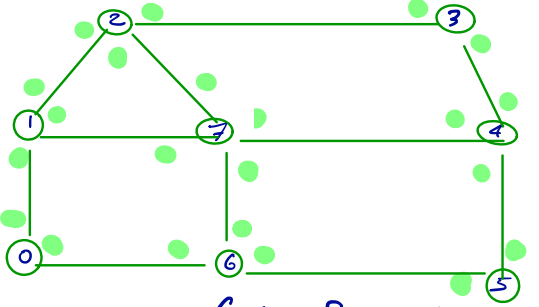


RAPPRESENTAZIONE DI GRAFI

- matrice di adiacenza

	0	1	2	3	4	5	6	7	grado
0		1					1		2
1	1		1					1	3
2		1		1				1	3
3			1		1				2
4				1		1		1	3
5					1		1		2
6	1					1		1	3
7		1	1		1		1		4

$\# \boxed{1} = 2m$



$vicini(u) = \{ x \in V : \{u,x\} \in E \}$

$N(u)$

$grado(u) = |N(u)|$

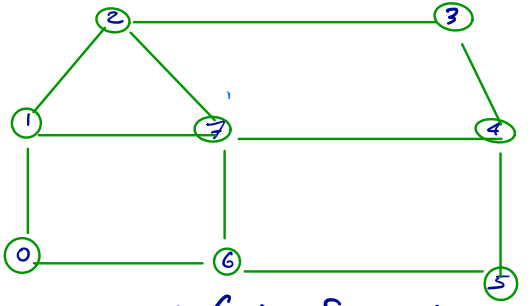
SPAZIO n^2

$somma\ gradi = \sum_{u \in V} grado(u) = 2m$

RAPPRESENTAZIONE DI GRAFI

- matrice di adiacenza

	0	1	2	3	4	5	6	7	grado
0		1					1		2
1	1		1					1	3
2		1		1				1	3
3			1		1				2
4				1		1		1	3
5					1		1		2
6	1					1		1	3
7		1	1		1		1		4



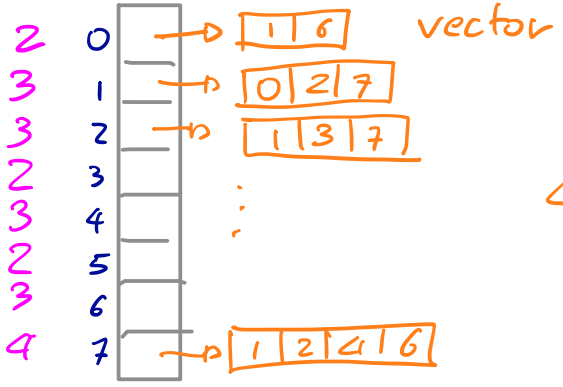
$vicini(u) = \{x \in V : \{u,x\} \in E\}$

$N(u)$

$grado(u) = |N(u)|$

- lista di adiacenza

grado



$0 \le m \le \binom{n}{2}$

spazio totale

$2m + n + n = O(m+n)$

lineare

GRAFI

$G = (V, E)$ $\quad E \subseteq V \times V$

V: nodi — E: archi

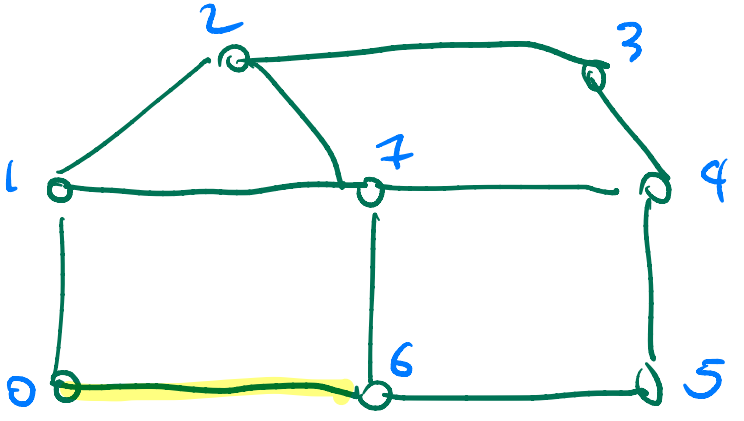
nodo $u \in V$

$N(u) = \{v \in V \mid uv \in E\}$

$N(7) = 1, 2, 4, 6$ lista di adiacenza

$d(u) = |N(u)|$

$d(7) = 4$



$(0,6)$ = $(6,0)$ relazione binaria non orientata undirected

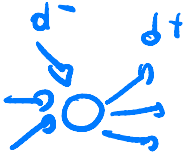
$(0,6)$ orientata directed

$(6,0)$

Handshaking lemma

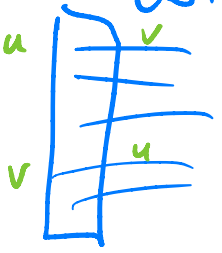
Grafo non orientato: $\sum_{u \in V} d(u) = 2|E|$ e quindi è pari il numero di nodi u t.c. $d(u)$ è dispari

Grafi orientati: $d^-(u)$, $d^+(u)$



$$\sum_{u \in V} d^-(u) = \sum_{u \in V} d^+(u) = |E|$$

Liste di adiacenza

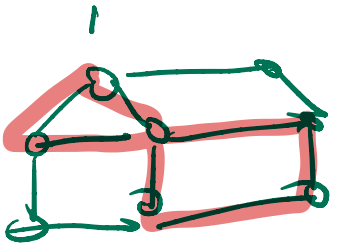


$|V| + \sum_{u \in V} d(u)$

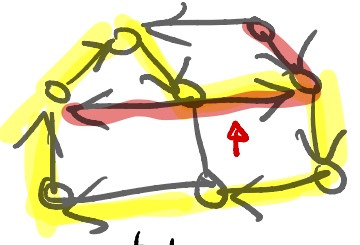
$|V| + \sum_{u \in V} d^+(u)$

Lineare: $|V| + |E|$

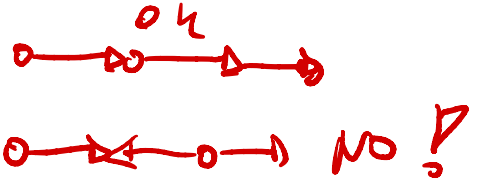
- Cammino $u_1 u_2 \dots u_k$ t.c. $u_i u_{i+1} \in E$ $(1 \le i < k)$
- lunghezza del cammino è $k-1$ archi
- ciclo: cammino in cui $u_1 = u_k$



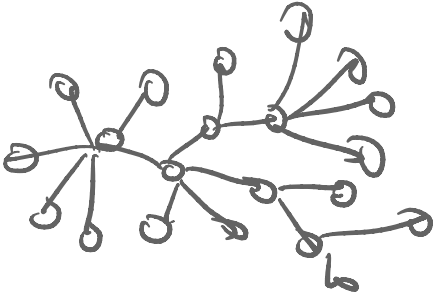
- G è <u>connesso</u> sse $\forall u, v \in V$ esiste un cammino
- G è <u>ciclico</u> sse $\exists$ un ciclo in G



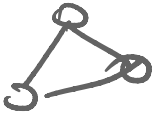
se G è orientato, si considera il cammino/ciclo orientato



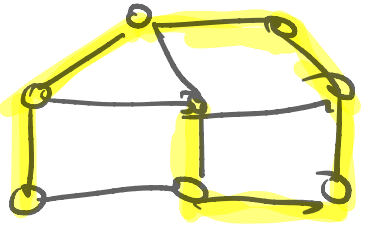
G è un albero se è connesso + aciclo



G è ciclico ? G è connesso ?



Your friend : DFS



Componenti Connesse

CC

```
1  Scansione( G ):
2    FOR (s = 0; s < n; s = s + 1)
3      raggiunto[s] = FALSE;
4    FOR (s = 0; s < n; s = s + 1) {
5      IF (!raggiunto[s]) DepthFirstSearchRicorsiva( s );
6    }
```

Nuova CC

```
1  DepthFirstSearchRicorsiva( u ):
2    raggiunto[u] = TRUE;
3    FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
4      v = x.dato;
5      IF (!raggiunto[v]) DepthFirstSearchRicorsiva(v);
6    }
```

GRAFI

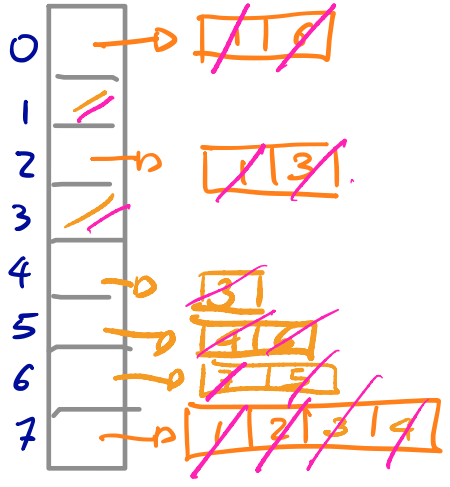
Connesso? Ciclico?

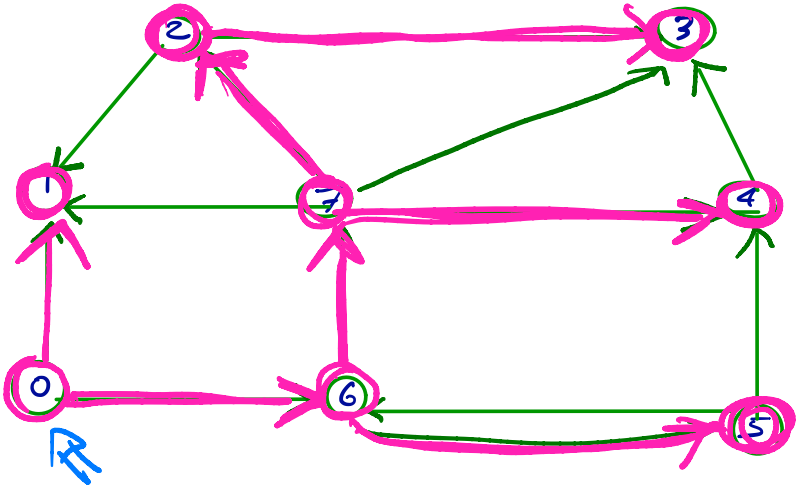
DFS = depth-first search = visita in profondità

```
1  Scansione( G ):
2    FOR (s = 0; s < n; s = s + 1)
3      raggiunto[s] = FALSE;
4    FOR (s = 0; s < n; s = s + 1) {
5      IF (!raggiunto[s]) DepthFirstSearchRicorsiva( s );
6    }
```

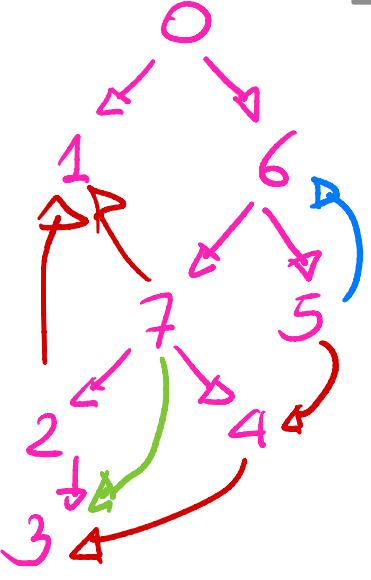
```
1  DepthFirstSearchRicorsiva( u ):
2    raggiunto[u] = TRUE;   ← ora è stato raggiunto
3    FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {   dato v : listaAdj[u]
4      v = x.dato;
5      IF (!raggiunto[v]) DepthFirstSearchRicorsiva(v);
6    }
     ← ora è stato completamente visitato
```

$O(n+m)$ tempo





DFSrec(0)



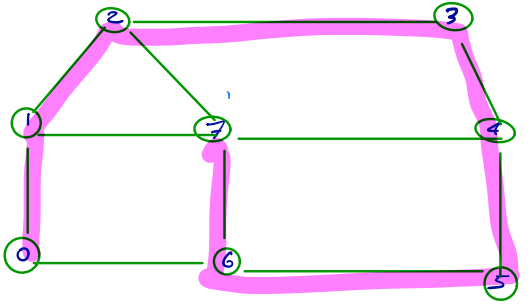
albero DFS (DFS tree)

- tree edge (archi dell'albero)
- forward edge (archi in avanti) — antenato → discendente (≠ genitore)
- back(ward) edge (archi all'indietro) — discendente → antenato
- cross edge (archi trasversali) — nodo visitato "dopo" → nodo visitato "prima"

oss Nei grafi non orientati gli archi UV forward e cross non esistono perché vengono attraversati a partire da V invece che da U

G è connesso?

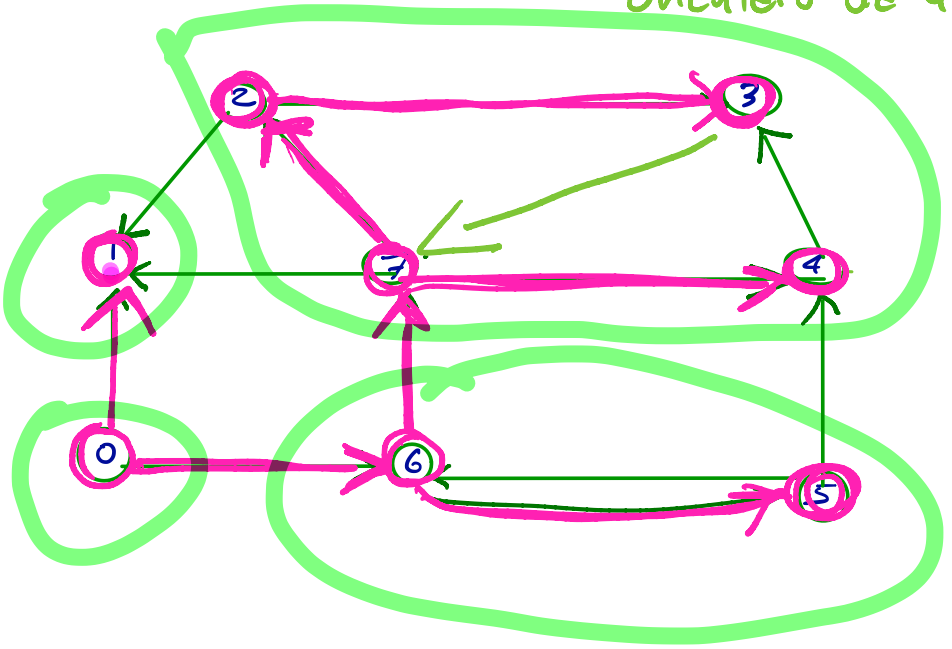
- G è non orientato: basta partire da un qualunque nodo u e verificare al termine della DFSric(u) che tutti i nodi siano stati raggiunti.

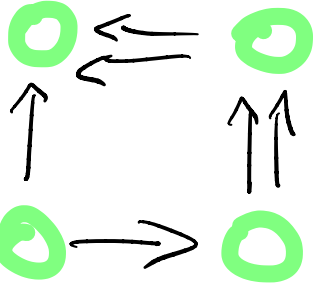


non è connesso

- G è orientato: esiste una nozione più forte che è quella di componente fortemente connesse (scc)
  ↑ strong

$\forall u, v \in SCC$: esiste cammino orientato da u a v





Le scc si trovano usando la DFS in modo più sofisticato, sempre $O(n+m)$

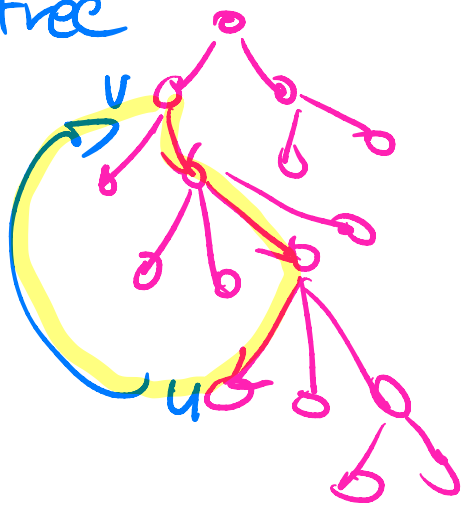
G è ciclico?

- G orientato / non orientato

Prop G è ciclico $\Rightarrow$ DFS $\exists$ arco back (u,v)

G è ciclico $\Leftarrow$ $\exists$ DFS con arco back (u,v)

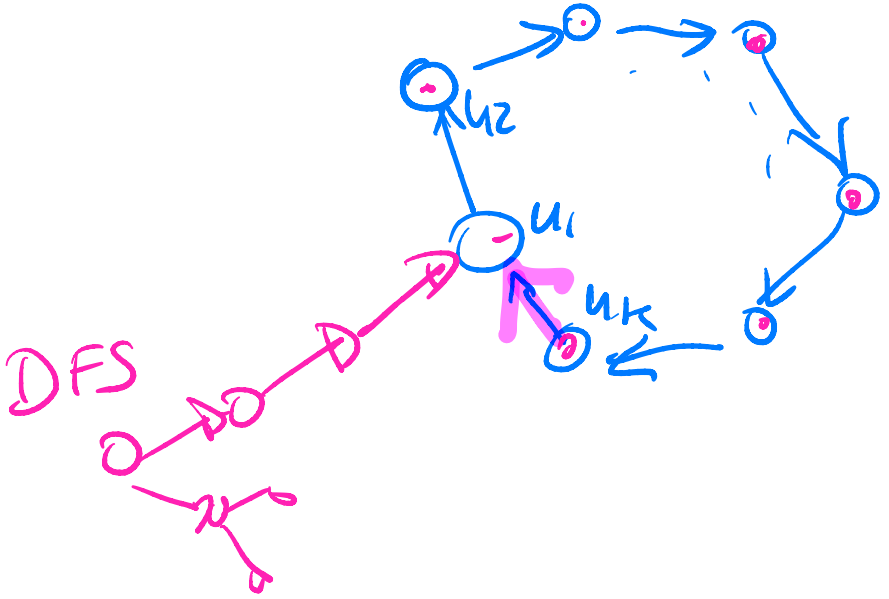
($\Leftarrow$) immediata DFS tree



ciclo va da v a u lungo il DFS tree e poi torna in v usando il back (u,v)

⇒ Prendiamo una qualunque DFS

Consideriamo un ciclo $u_1 u_2 \dots u_k u_1$ raggiunto dalla DFS dove u_1 è il "primo" nodo scoperto dalla DFS in tale ciclo



oss. siccome $u_2 \dots u_k$ sono raggiungibili da u_1, la DFS li scoprirà in un qualche ordine (non necessariamente $u_2, u_3 \dots, u_k$) —

⇒ u_k sarà discendente di u_1

⇒ (u_k, u_1) è back quindi $u = u_k$, $v = u_1$

```
1  Scansione( G ):
2    FOR (s = 0; s < n; s = s + 1)
3      raggiunto[s] = FALSE;
4    FOR (s = 0; s < n; s = s + 1) {
5      IF (!raggiunto[s]) DepthFirstSearchRicorsiva( s );
6    }
```

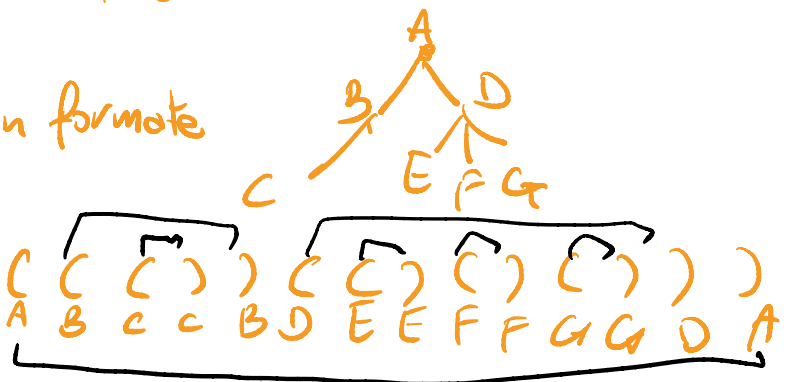
```
1  DepthFirstSearchRicorsiva( u ):
2    raggiunto[u] = TRUE;
3    FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
4      v = x.dato;
5      IF (!raggiunto[v]) DepthFirstSearchRicorsiva(v);
6    }
```

auto v: listaAdiacenza[u]

uv = tree edge

else uv = back edge se v è antenato di u

albero: parentesi ben formate



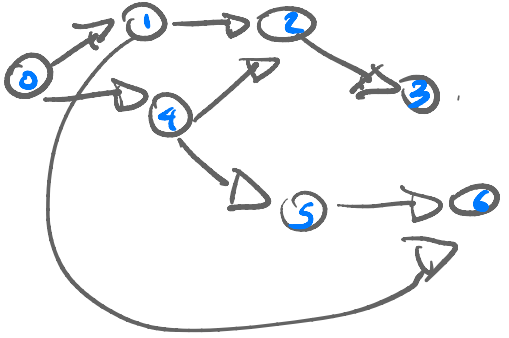
esercizio: trasformare C_u e $)_u$ in due vettori booleani
e usarli per classificare gli archi come visto prima

suggerimento: usare un contatore che sostituisce le parentesi

ORDINE TOPOLOGICO

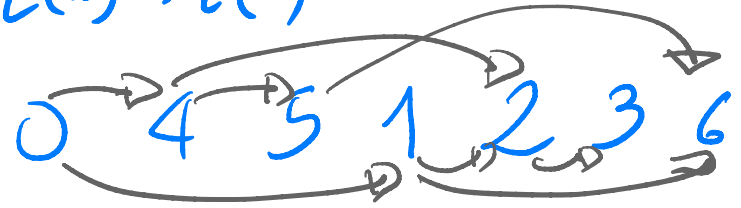
DAG = directed acyclic graph

$G = (V, E)$ orientato e aciclico



ordinamento topologico: $\eta : V \to [n]$, $n = |V|$

t.c. $(u,v) \in E \Rightarrow \eta(u) < \eta(v)$



DFS leggermente modificata

$O(n+m)$ tempo

```
1  OrdinamentoTopologico( ):
2    FOR (s = 0; s < n; s = s + 1)
3      raggiunto[s] = FALSE;
4    contatore = n - 1;
5    FOR (s = 0; s < n; s = s + 1) {
6      IF (!raggiunto[s]) DepthFirstSearchRicorsivaOrdina( s );
7    }
```

```
1  DepthFirstSearchRicorsivaOrdina( u ):
2    raggiunto[u] = TRUE;
3    FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
4      v = x.dato;
5      IF (!raggiunto[v]) DepthFirstSearchRicorsivaOrdina( v );
6    }
7    eta[u] = contatore;
8    contatore = contatore - 1;
```

$G = (V, E)$

$E \subseteq V \times V$

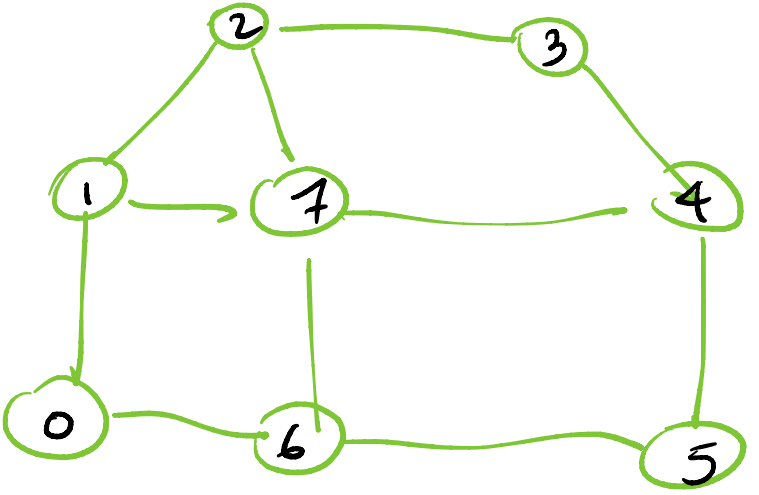
self-loop NO: (x, x)

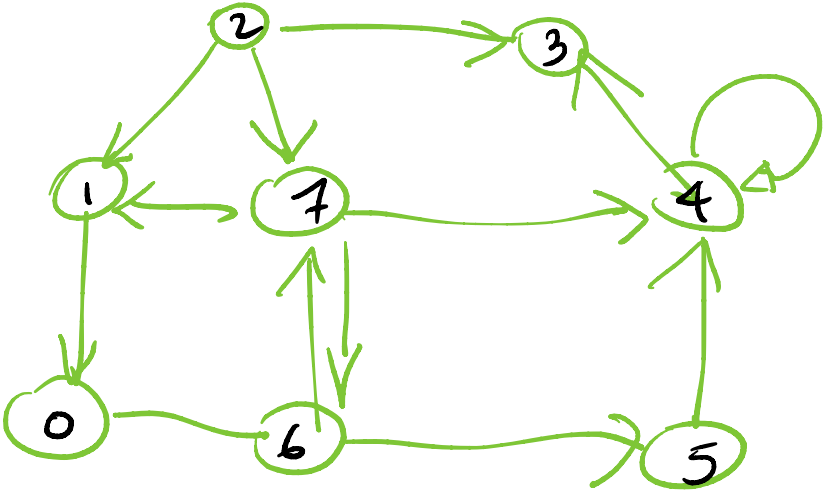
$G \leftrightarrow$ relazione binaria

$(u,v) \neq (v,u)$ orientato

$(u,v) = (v,u)$ non orientato
(rel. simmetrica)

$\{u,v\}$ oppure uv



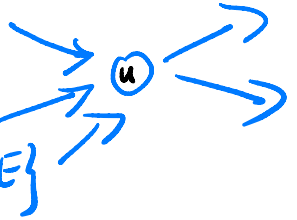


$d_u = \deg(u) = \text{grado}(u) = |N(u)|$

G non orientato

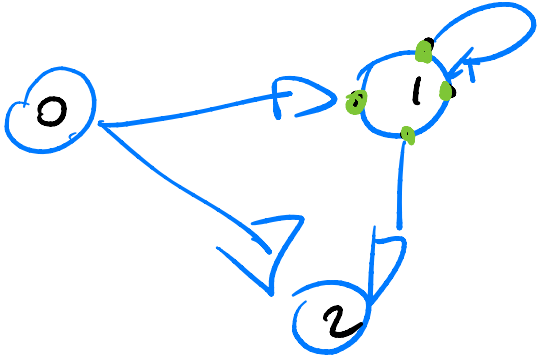
$N^-(u)$, $N^+(u)$

$\{v \in V : (v,u) \in E\}$ $\{v \in V : (u,v) \in E\}$



G orientato

$d_u^- = |N^-(u)|$, $d_u^+ = |N^+(u)|$



$d^-(1) = 2 = |\{(0,1),(1,1)\}|$

$d^+(1) = 2 = |\{(1,1),(1,2)\}|$

$d(1) = d^-(1) + d^+(1)$

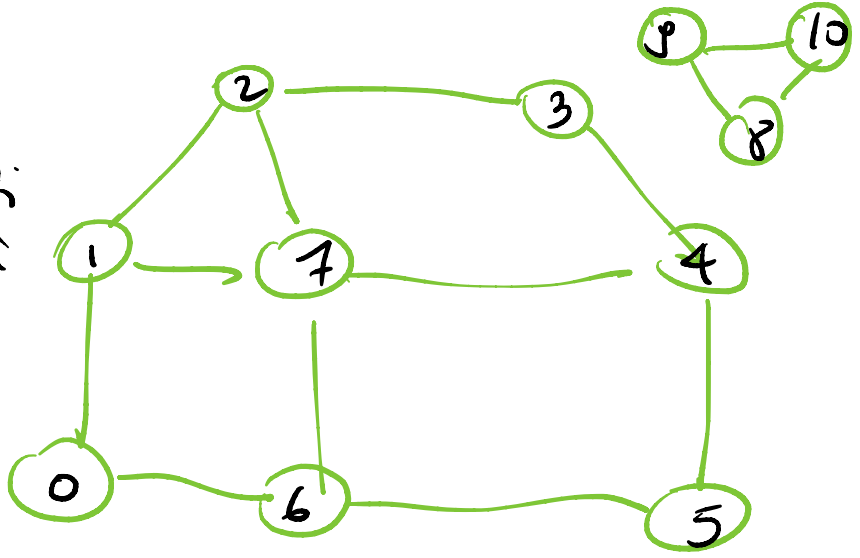
Handshaking lemma:

- G non orientato: $\sum_{u \in V} d_u = 2|E|$ e ci sono un numero pari di nodi u aventi d_u dispari

- G orientato: $\sum_{u \in V} d_u^- = \sum_{u \in V} d_u^+ = |E|$

- cammino [walk, trail, path]
  $u_1 u_2 \dots u_k$ sequenza di nodi in cui nodi adiacenti
  $(u_j, u_{j+1}) \in E$ sono collegati da archi

  $k-1$ = lunghezza cammino
  = # archi attraversati



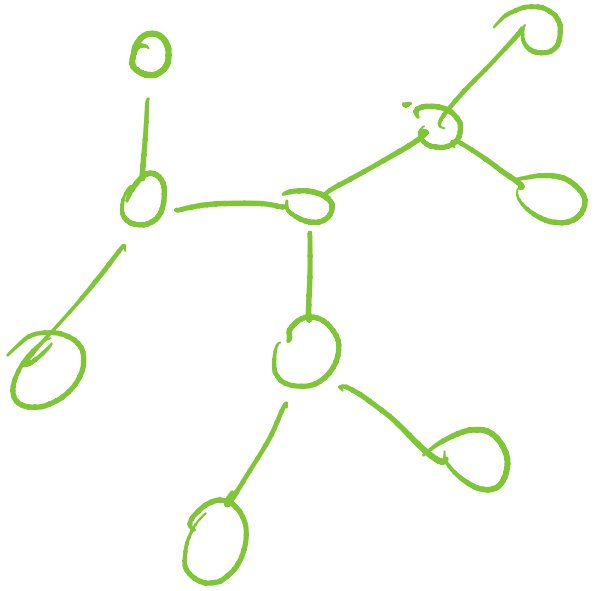
- ciclo quando $u_1 = u_k$ e $k > 1$

- G è **connesso** se esiste un cammino per ogni coppia di nodi

  (se non è connesso, prendiamo le sue componenti connesse)
  cc

- G è **ciclico** se contiene un ciclo; **aciclico** altrimenti

G è aciclico e connesso sse G è un albero



Domanda: come faccio a stabilire ① G connesso ?
② aciclico

VISITA IN PROFONDITA' : DFS depth-first search

```
1  Scansione( G ):
2    FOR (s = 0; s < n; s = s + 1)
3      raggiunto[s] = FALSE;
4    FOR (s = 0; s < n; s = s + 1) {
5      IF (!raggiunto[s]) DepthFirstSearchRicorsiva( s );
6    }
```

G è connesso sse istruzione 5 lancia DFS solo con s=0

```
1  DepthFirstSearchRicorsiva( u ):
2    raggiunto[u] = TRUE;
3    FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
4      v = x.dato;
5      IF (!raggiunto[v]) DepthFirstSearchRicorsiva(v);
6    }
```

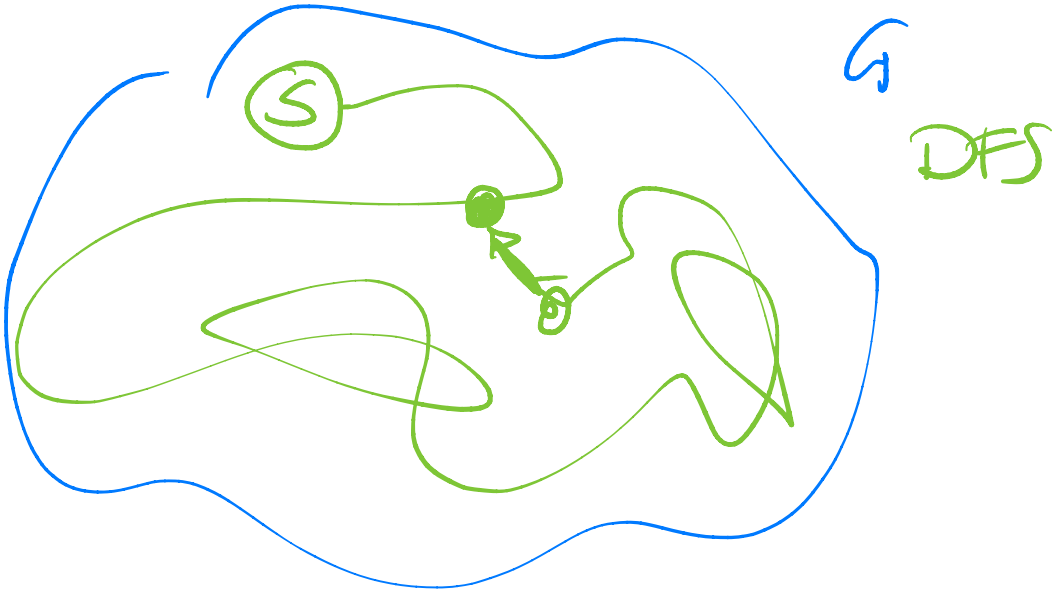
raggiunto ↔ visitato

for (auto v: adj[u])

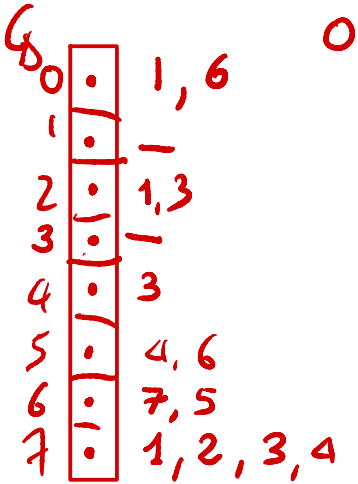
F completato = T

raggiunto[v] = TRUE ⇔ G ciclico
se
(u,v) arco back

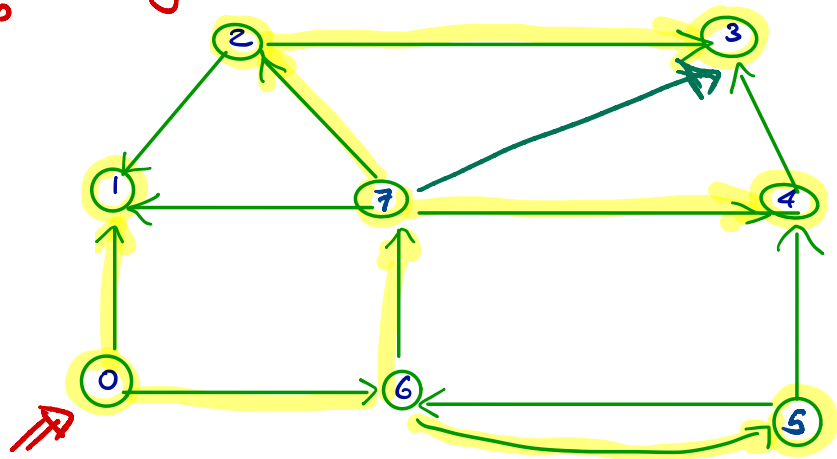
raggiunto[v] = TRUE
se
completato[v] = FALSE

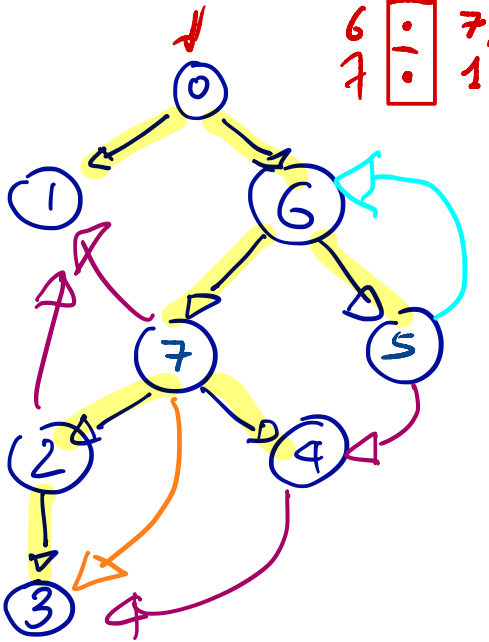

S
G
DFS

Esempio
(liste ordinate)



$O(m+n)$ tempo perché ogni arco viene attraversato 2 volte





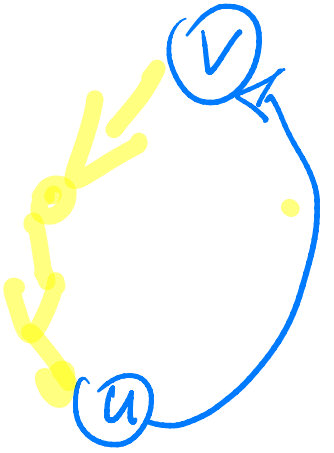
ALBERO DFS

- tree edge (archi dell'albero)
- forward edge (archi in avanti)
- backward edge (archi all'indietro)
- cross edge (archi trasversali)
  vanno da nodo scoperto "dopo" a nodo scoperto "prima"

G ciclico $\Leftrightarrow$ $\exists$ arco (u,v) back

($\Leftarrow$) $\exists$ back

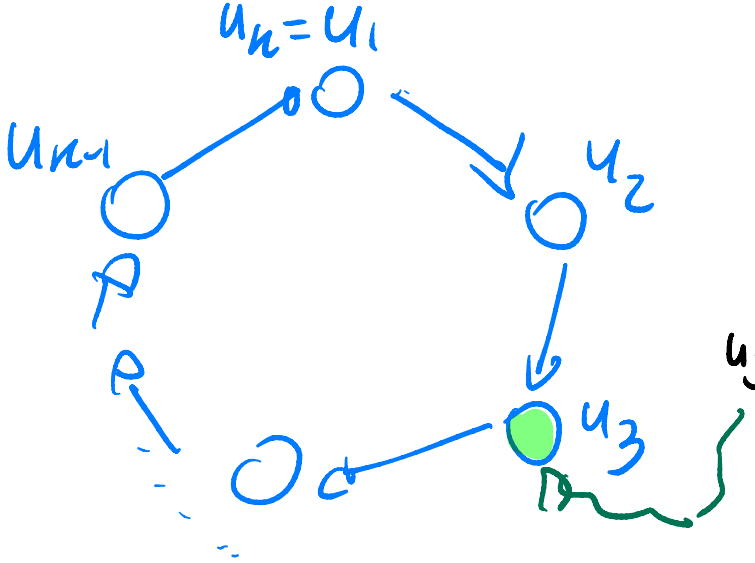
questo implica un cammino di tree edge da v a u: tale cammino diventa un ciclo estendolo con l'arco (u,v)



v antenato u nell'albero DFS

($\Rightarrow$) G ciclico

sia $u_1, u_2 \dots u_k = u_1$ un ciclo in G



Hp. almeno uno tra $u_1, \dots, u_{k-1}$ è il nodo raggiunto dalla DFS

DFS(u_j) scopre tutti gli altri nodi $u_1, \dots, u_{k-1}$ ($\neq u_j$) prima o poi in un certo ordine

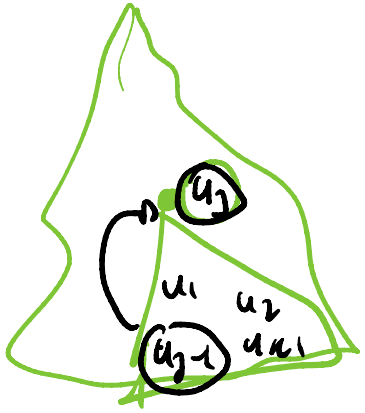
sono tutti quindi discendenti di u_j nell'albero DFS

ALBERO DFS

tra questi c'è anche u_{j-1}, predecessore di u_j nel ciclo $\left(u_j = u_1 \Rightarrow u_{j-1} = u_{k-1}\right)$

quindi (u_{j-1}, u_j) è un arco che collega un discendente a un suo antenato

$\Rightarrow$ è un arco back ∎

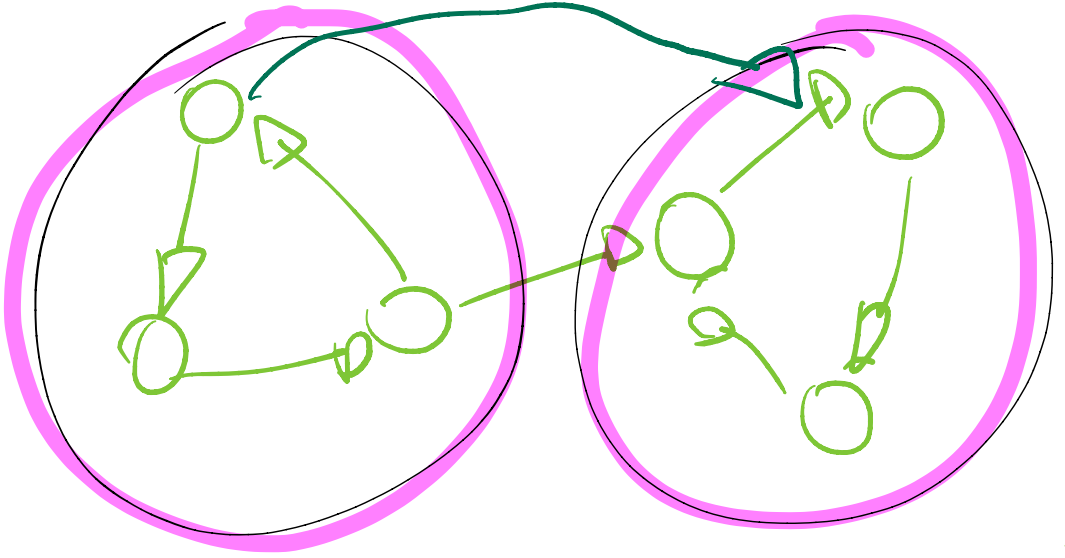


Su G non orientato: ∃ solo BACK

(u, v) back se collega u discendente proprio di v
(cioè v non è il padre di u nell'albero DFS)

G orientato : componente fortemente connessa

SCC

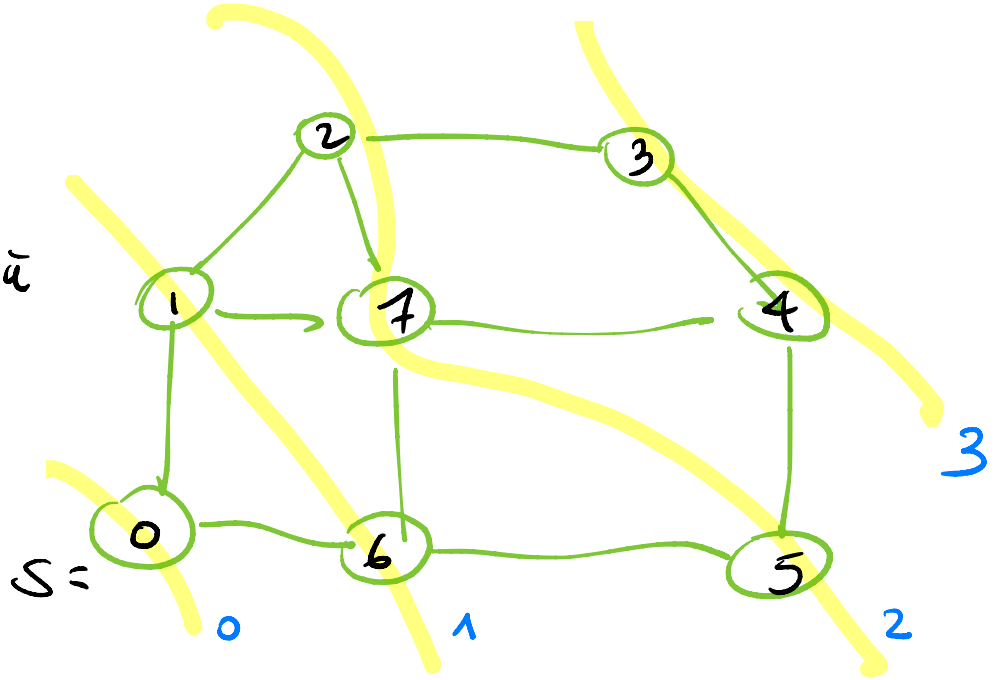


DAG
=
DIRECTED
ACYCLIC
GRAPH

sulle SCC

BFS = Breadth-first search
(visita in ampiezza)

$d(x,y)$ = lunghezza del cammino più breve tra x e y

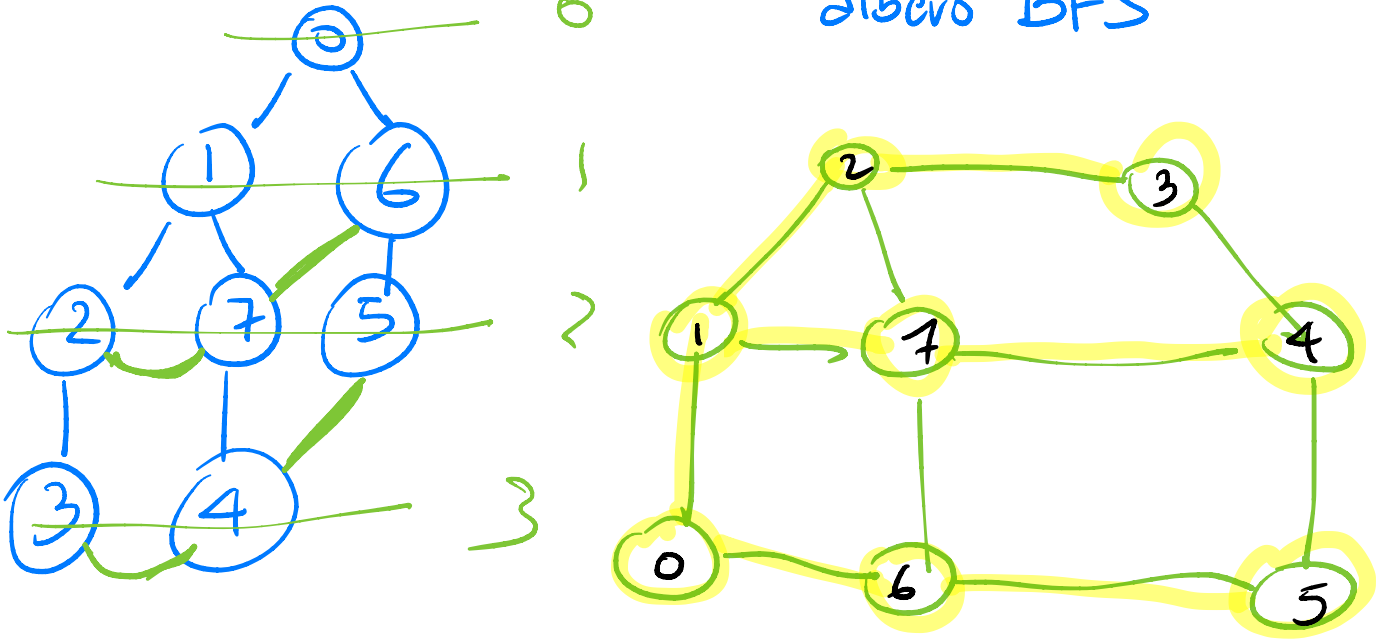


L'ordine di visita dei nodi è compatibile con l'ordine non-decrescente basato su $d(s,-)$

BFS(s):

```
1  for (u=0; u<n; u++) visitato[u] = false;
2  C.enqueue(s); visitato[s] = true;   d_s[s] = 0;
3  while (!C.empty()){
4        u = C.dequeue();        // d_s[u] è ben definito
5        for (auto v : adj[u]){
6              if (!visitato[v]) {C.enqueue(v); visitato[v] = true;}
   }
```

$d_s[v] = d_s[u] + 1$



$(u,v) \Rightarrow |d_s[u] - d_s[v]| \leq 1$

Complessità

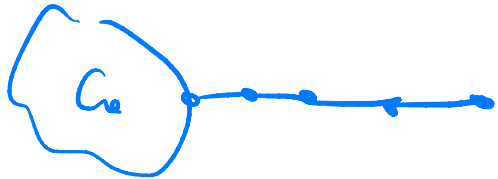
DFS e BFS richiedono tempo lineare $O(n+m)$

"ogni lista di adiacenza viene incrementalmente scandita solo in avanti, sempre nella stessa direzione"

distanza media $\sum_{u \neq v} \frac{d(u,v)}{n(n-1)}$

(esperimento di MILGRAM "6 gradi di separazione")

diametro $\max_{u \neq v} d(u,v)$

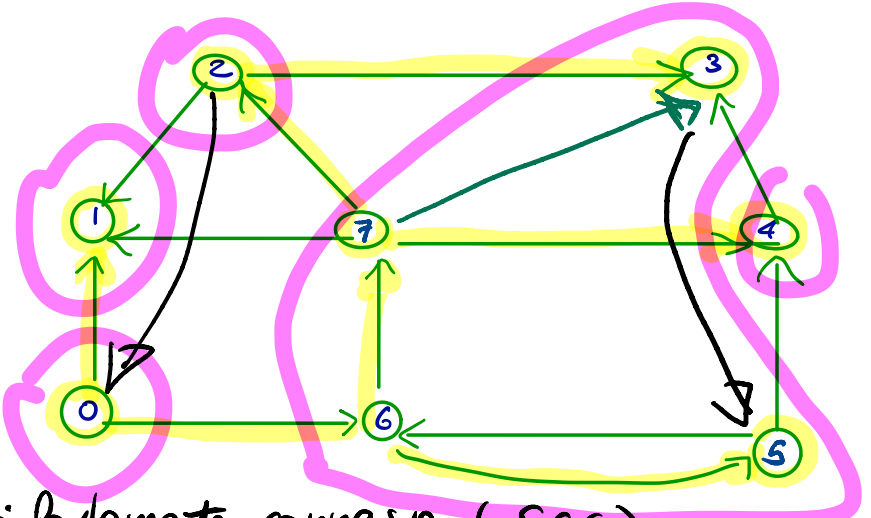


↳ $\max_{u \neq v} d(u,v) = \max_{s} \underbrace{\left(\max_{v} d(s,v) \right)}_{BFS(s)}$

$O(n \cdot BFS) = O(n^2 + nm)$ tempo

↳ se G è denso ⇒ metodi basati sulla moltiplicazione tra matrici

↳ se G è sparso ⇒ $O(n^2)$ tempo NON si può fare meno di così SETH



Componenti fortemente connesse (SCC)

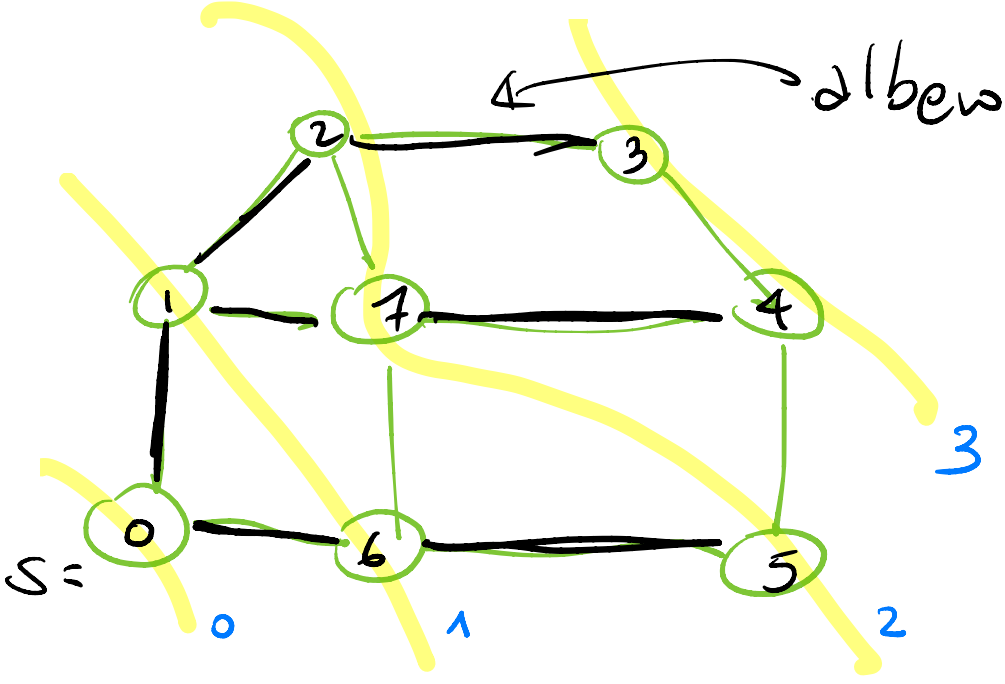
DFS aiuta a calcolare le SCC in $O(m+n)$ tempo

$\forall u,v \in SCC \Leftrightarrow \exists$ cammino $u \rightsquigarrow v$ e cammino $v \rightsquigarrow u$ (orientati $\forall$)

$\Leftrightarrow$ u, v sono su un ciclo orientato

C

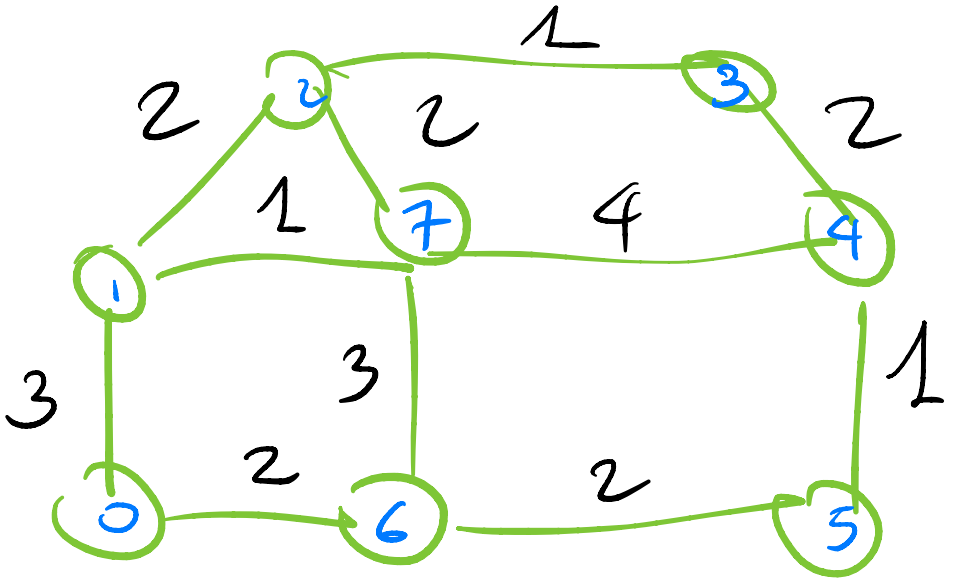
0	1	6	2	7	5	3	4
0	1	1	2	2	2	3	3



- tree edge
- non-tree edge
  - collegano nodi che sono
    - stesso livello oppure
    - il loro livello differisce di 1

GRAFI PESATI

$G = (V, E, W)$, $W : E \to \mathbb{R}$

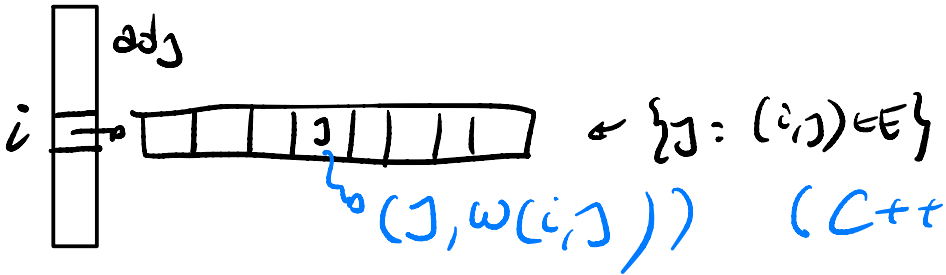


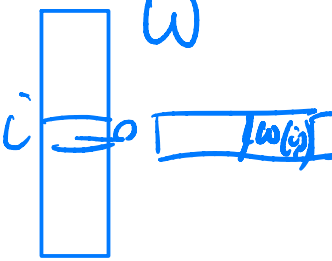
matrice adiacenza

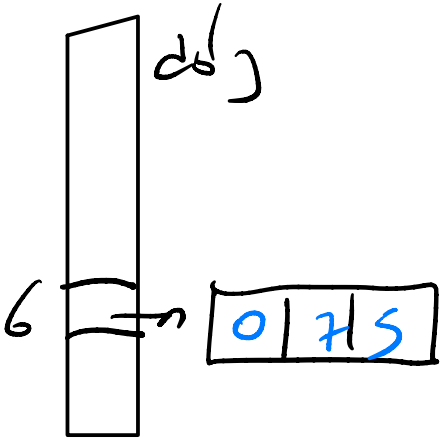
$$A_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & \text{altrimenti} \end{cases}$$

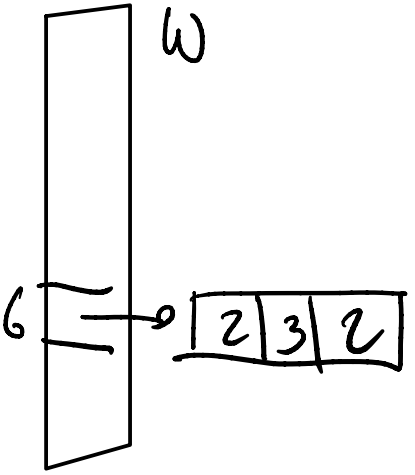
$$A_{ij} = \begin{cases} w(i,j) & (i,j) \in E \\ 0 & \text{altrimenti} \end{cases}$$

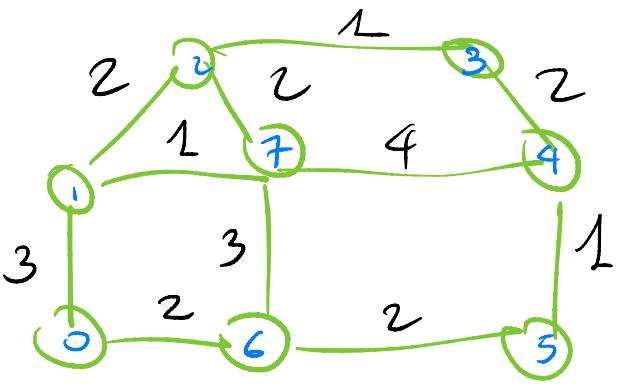
liste di adiacenza

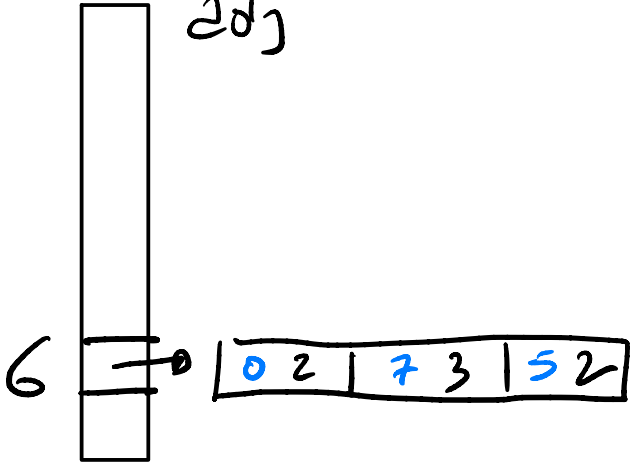












vector < pair<int, float> > adj [Nmax]

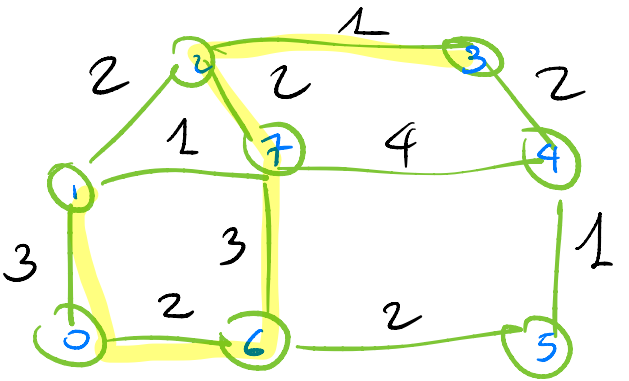
↑ ↑

j w(i,j)

cammini pesati

$u_1, u_2 \dots u_n$ ha peso $\sum_{i=1}^{k-1} w(u_i, u_{i+1})$

$d(u,v)$ = peso del cammino pesato minimo (in termini di peso)

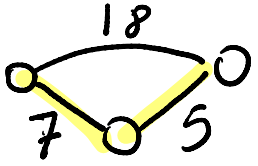


1 0 6 7 2 3

ha peso

3 + 2 + 3 + 2 + 1 = 11

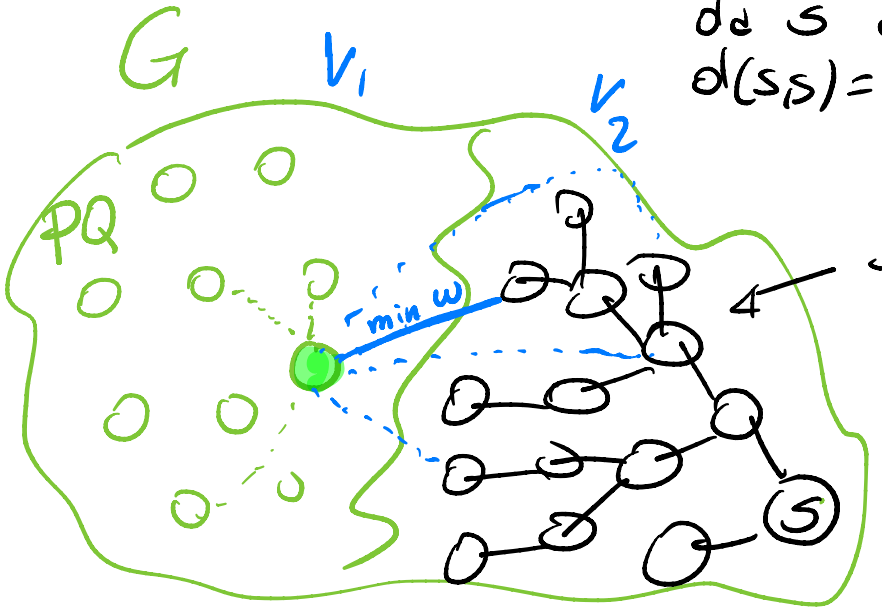
cammino pesato minimo potrebbe avere più archi del cammino minimo senza pesi:



Quindi la BFS non va bene per trovare i cammini minimi pesati

ALGORITMO DI DIJKSTRA per i CAMMINI MINIMI

- ~~BFS~~ ~~coda~~ → code di priorità PQ (heap min)
- Proprietà invariante: fissare un nodo s di partenza
  considerare i cammini pesati minimi
  da s a ogni altro nodo $u \neq s$
  $d(s,s) = 0$



sappiamo $d(s,u)$ per questi nodi

taglio (V_1, V_2) è una partizione di V

$E(V_1, V_2) = \{(u,v) \in E : \begin{matrix} u \in V_1 \\ v \in V_2 \end{matrix}\}$

V_1 → PQ coda di priorità

V_2 → albero dei cammini minimi scoperti finora

$\forall u \in V_1$: dist[u] = peso del miglior cammino trovato finora, usando nodi in V_2 e archi del taglio $E(V_1, V_2)$

pred[u] = arco $(u, pred[u])$ in $E(V_1, V_2)$

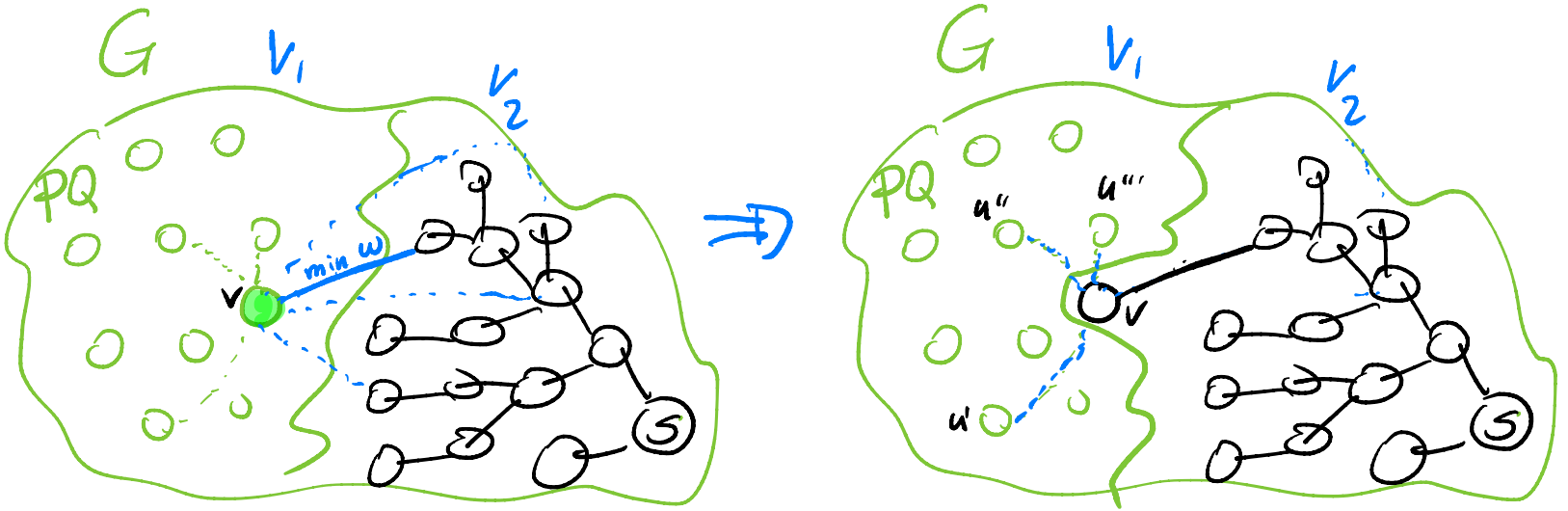
↳ vale solo quando $dist[u] \neq \infty$

INIT $V_1 = V - \{s\}$ $dist[u] = +\infty$ $dist[s] = 0$

$V_2 = \{s\}$ $pred[u] = -1$ $pred[s] = s$

V_1 → PQ

Passo generico:



Sia $v \in V_1$ il nodo con minima dist[v] usando PQ

Estrai v e poni $d(s,v) = dist[v]$ // funziona solo con pesi positivi

Aggiorna i vicini u di v che sono eventualmente ancora in V_1

ORA: $v \in V_2$

```
1  Dijkstra( s ):
2    FOR (u = 0; u < n; u = u + 1) {
3      pred[u] = -1;
4      dist[u] = +∞;
5    }
6    pred[s] = s;
7    dist[s] = 0;
8    FOR (u = 0; u < n; u = u + 1) {
9      elemento.peso = dist[u];
10     elemento.dato = u;
11     PQ.Enqueue( elemento );
12   }
13   WHILE (!PQ.Empty( )) {
14     e = PQ.Dequeue( );
15     v = e.dato;
16     FOR (x = listaAdiacenza[v].inizio; x != null; x = x.succ) {
17       u = x.dato;
18       IF (dist[u] > dist[v] + x.peso) {
19         dist[u] = dist[v] + x.peso;
20         pred[u] = v;
21         PQ.DecreaseKey( u, dist[u] );
22       }
23     }
24   }
```

INIT

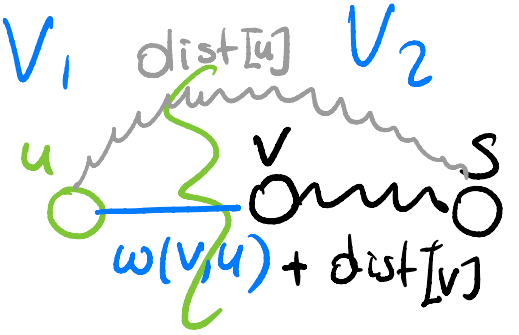
inizializza coda di priorità

estrarre v : HA DISTANZA dist[v] minima tra i nodi in V_1

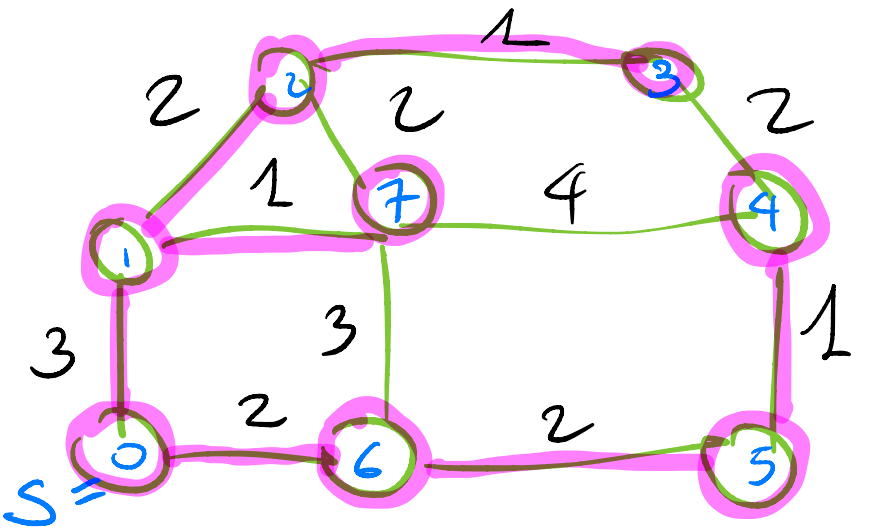
vicini di v

$w(v,u)$

aggiorna la priorità di u dist[u]



ESEMPIO



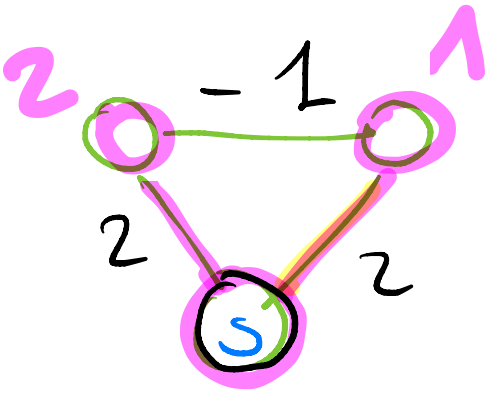
u	0	1	2	3	4	5	6	7
dist[u]	0	~~∞~~ 3	~~∞~~ 5	~~∞~~ 6	~~∞~~ ~~8~~ 5	6 4	~~∞~~ 2	~~∞~~ ~~5~~ 4
pred[u]	0	~~-1~~ 0	-1	~~-1~~ 2	~~-1~~ ~~7~~ 5	-1 6	~~-1~~ 0	~~-1~~ ~~5~~ 1

costo dominante: aggiornare PQ per ogni arco

$O(|E| \lg |V|)$ $O(m \lg n)$

tempo

Pesi negativi



dist[ ] = ~~2~~ 1

Dijkstra non funziona con pesi negativi

⇒ usare un altro algoritmo

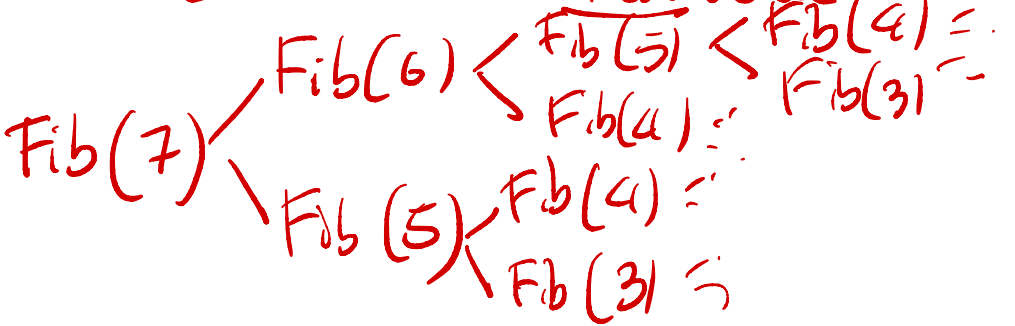
PROGRAMMAZIONE DINAMICA → ricorsione tabulata

esempio

Fibonacci $F_n = F_{n-1} + F_{n-2}$ $F_0 = 0$ $F_1 = 1$

```
int Fib(int n) { // n ≥ 0
  if (n ≤ 1) return n;
  return Fib(n-1) + Fib(n-2);
}
```

Problema: esplosione combinatoria delle chiamate ricorsive che calcolano lo stesso valore



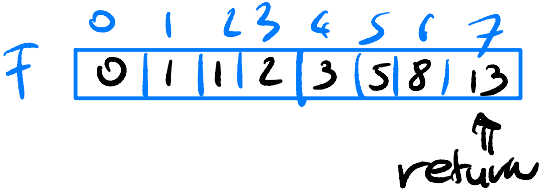
input n

circa $F_n \sim \Phi^n$ chiamate

dimensione $k = \log_2 n$

costo Φ^{2^k}

solitamente si usa una tabella




```
F[0] = 0
F[1] = 1
for(i = 2; i <= n; i++) F[i] = F[i-1] + F[i-2]
return F[n]
```

$O(n)$ tempo $O(2^k)$ tempo

semplice esempio di programmazione dinamica

- regola ricorsiva di risoluzione
- regola per riempire la tabella

$$x^n = \begin{cases} (x^{n/2})^2 & n \text{ pari} \\ (x^{\frac{n-1}{2}})^2 \cdot x & n \text{ dispari} \end{cases}$$

$T(n) \le T(\frac{n}{2}) + O(1)$ moltiplicazioni

$\Rightarrow T(n) = O(\lg n)$

oss calcolare F_n in $O(\lg n)$ tempo usando l'esponenziazione veloce A^n di una opportuna matrice binaria A 2×2

LCS = Longest common subsequence

stringhe/sequence

$A = a_0 a_1 a_2 \cdots a_{m-1}$

$B = b_0 b_1 b_2 \cdots b_{n-1}$

sottosequenza $(A) = a_{i_1} a_{i_2} \cdots a_{i_k}$ $\quad 0 \le i_1 < i_2 < \cdots < i_k \le m-1$

sottosequenza $(B) = b_{j_1} b_{j_2} \cdots$ $\quad 0 \le j_1 < j_2 < \cdots \le n-1$

X è sottosequenza comune se è sottosequenza sia di A che di B

esempio

A = a t c g a t a t c g a t

B = t t a t a t c c g a t c c

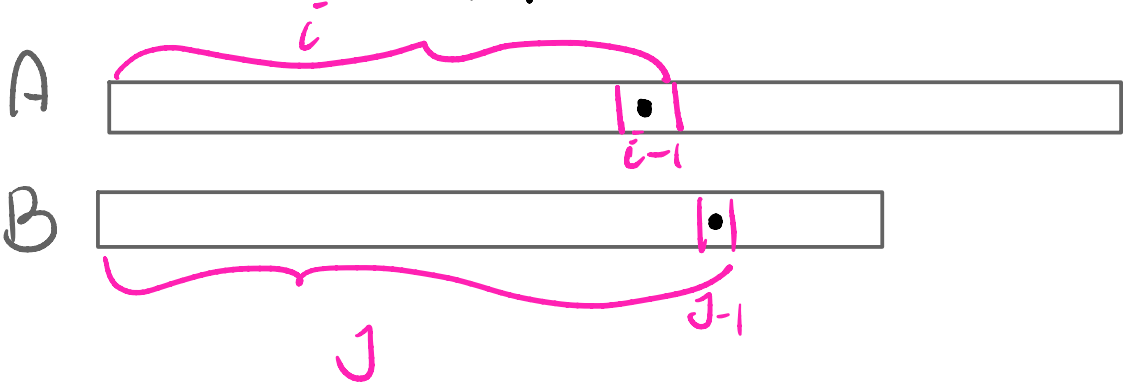
cerchiamo la lunghezza delle più lunghe

Problema: possono esserci un numero esponenziale di soluzioni / candidate

Soluzione: programmazione dinamica (DP = dynamic programming)

Regola ricorsiva:

- caso base: $A = \varepsilon$ oppure $B = \varepsilon \Rightarrow LCS(A, B) = 0$
- passo induttivo: $A, B \neq \varepsilon$



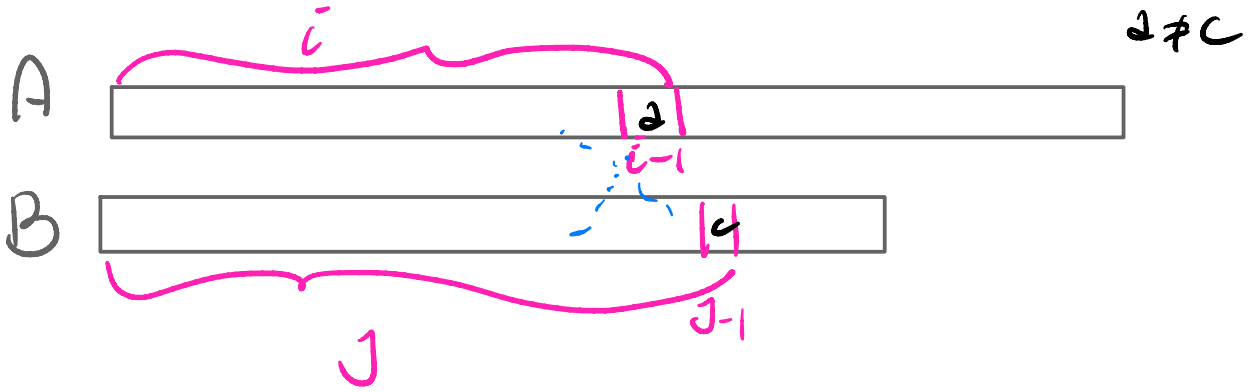
A[i-1]
vs
B[J-1]

1) $A[i-1] = B[j-1] \Rightarrow LCS(i,j) = 1 + LCS(i-1, j-1)$

$LCS(a_0 \cdots a_{i-1}, b_0 \cdots b_{j-1})$

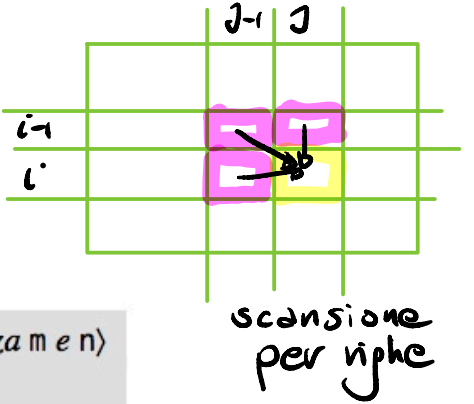
i $\quad$ j

2) $A[i-1] \neq B[j-1] \Rightarrow LCS(i,j) = \max\{LCS(i-1,j), LCS(i, j-1)\}$



$$L(i,j) = \begin{cases} 0 & \text{se } i=0 \text{ o } j=0 \\ L(i-1,j-1)+1 & \text{se } i,j>0 \text{ e } a[i-1]=b[j-1] \\ \max\{L(i,j-1), L(i-1,j)\} & \text{se } i,j>0 \text{ e } a[i-1]\neq b[j-1] \end{cases}$$

tabella 2-dim.




```
1   LCS( a, b ):                          ⟨pre: a e b sono di lunghezza m e n⟩
2     FOR (i = 0; i <= m; i = i+1)
3       lunghezza[i][0] = 0;
4     FOR (j = 0; j <= n; j = j+1)
5       lunghezza[0][j] = 0;
6     FOR (i = 1; i <= m; i = i+1)
7       FOR (j = 1; j <= n; j = j+1) {
8         IF (a[i-1] == b[j-1]) {
9           lunghezza[i][j] = lunghezza[i-1][j-1] + 1;
10        } ELSE IF (lunghezza[i][j-1] > lunghezza[i-1][j]) {
11          lunghezza[i][j] = lunghezza[i][j-1];
12        } ELSE {
13          lunghezza[i][j] = lunghezza[i-1][j];
14        }
15      }
16    RETURN lunghezza[m][n];
```

casi base riga 0 e colonna 0

passo induttivo

valore della LCS(A,B)

m n

```
1  StampaLCS( i, j ):                                  ⟨pre: 0 ≤ i ≤ m e 0 ≤ j ≤ n⟩
2    IF ((i > 0) && (j > 0)) {
3      <i', j'> = indice[i][j];
4      StampaLCS( i', j' );
5      IF ((i' == i-1) && (j' == j-1))) PRINT a[i-1];
6    }
```

lunghezza[][]

		B	A	A	B	D	C	D	C	A	A	C	A	C	B	A
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A 1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
D 2	0	0	1	1	1	2	2	2	2	2	2	2	2	2	2	2
C 3	0	0	1	1	1	2	3	3	3	3	3	3	3	3	3	3
A 4	0	0	1	2	2	2	3	3	3	4	4	4	4	4	4	4
A 5	0	0	1	2	2	2	3	3	3	4	5	5	5	5	5	5
B 6	0	1	1	2	3	3	3	3	3	4	5	5	5	5	6	6

KNAPSACK (ZAINO)

- n oggetti : $0 .. n-1$
- peso $p_0 \dots p_{n-1}$
- valore $v_0 \dots v_{n-1}$ $\Big\} \in \mathbb{N}$
- zaino capacità/possanza massima P

  ↳ condizione necessaria $I \subseteq \{0,1,\dots,n-1\} \Rightarrow \sum_{i \in I} p_i \leq P$

- obiettivo: $\max_I \sum_{i \in I} v_i$

DP in $O(nP)$ tempo

↳ pseudo polinomiale perché dipende dal valore di P

- regola ricorsiva : preso oggetto $i-1$ → scelgo → ricorrione
  → non lo scelgo → ricorrione

```
1  Bisaccia( peso, valore, possanza ):
2  ⟨pre: peso e valore sono array di n interi positivi, possanza è un intero positivo⟩
3    FOR (i = 0; i <= n; i = i+1) {
4      FOR (j = 0; j <= possanza; j = j+1) {
5        V[i][j] = 0;
6      }
7    }
8    FOR (i = 1; i <= n; i = i+1) {
9      FOR (j = 1; j <= possanza; j = j+1) {
10       V[i][j] = V[i-1][j];
11       IF (j >= peso[i-1])) {
12         m = V[i-1][j-peso[i-1]] + valore[i-1];
13         IF (m > V[i][j]) V[i][j] = m;
14       }
15     }
16   }
17   RETURN V[n][possanza];
```

P

V[i][j] = max valore in uno zaino di capacità j usando gli oggetti 0 ... i-1 (i)

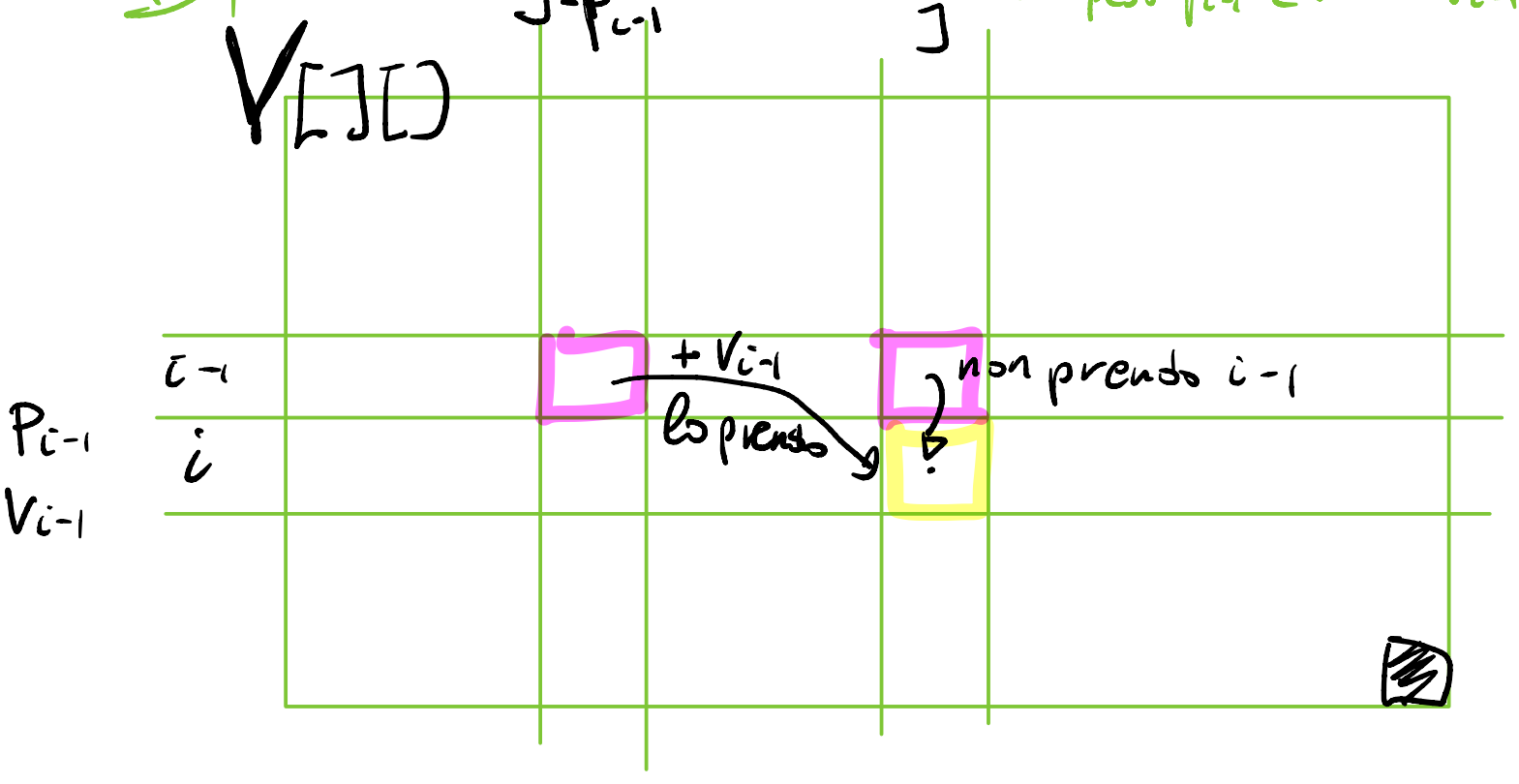
$$V[i][j] = \max\left(V[i-1][j],\ V[i-1][j-p_{i-1}] + v_{i-1}\right)$$

non prendo oggetto $i-1$

prendo oggetto $i-1$ con peso p_{i-1} e valore v_{i-1}

DP

$V[\ ][\ ]$



MST = Minimum (-cost) Spanning Tree

spanning tree = albero di ricoprimento

GREEDY vs DP

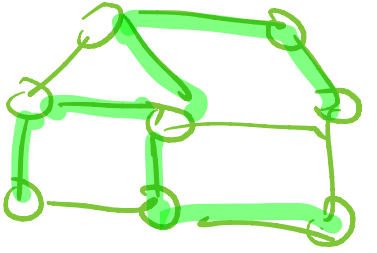
$G = (V, E, W)$ def $T \subseteq E$ è uno spanning tree se

① T è aciclico (albero)

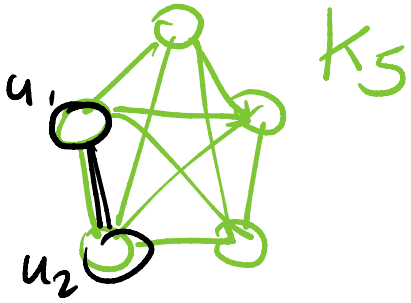
② $|T| = |V| - 1$ (tocca tutti i nodi)

Esempi visti in precedenza di spanning tree:

- DFS tree
- BFS tree — nei grafi non orientati
- Dijstra tree



oss il numero di spanning tree in un grafo può essere esponenziale



K_n = il grafo completo di n nodi (clique) a due a due collegati

$u_1 \to$ grado $n-1 \Rightarrow n-1$ scelte $\to u_2$

$u_2 \to n-2$ scelte per u_3 ...

ci sono $(n-1)!$ cammini senza cicli

un cammino è un caso particolare di spanning tree

$\Rightarrow$ almeno $(n-1)!$ spanning tree

(vedi polinomio di Tutte)

$T \subseteq E$

$G = (V, E, W)$

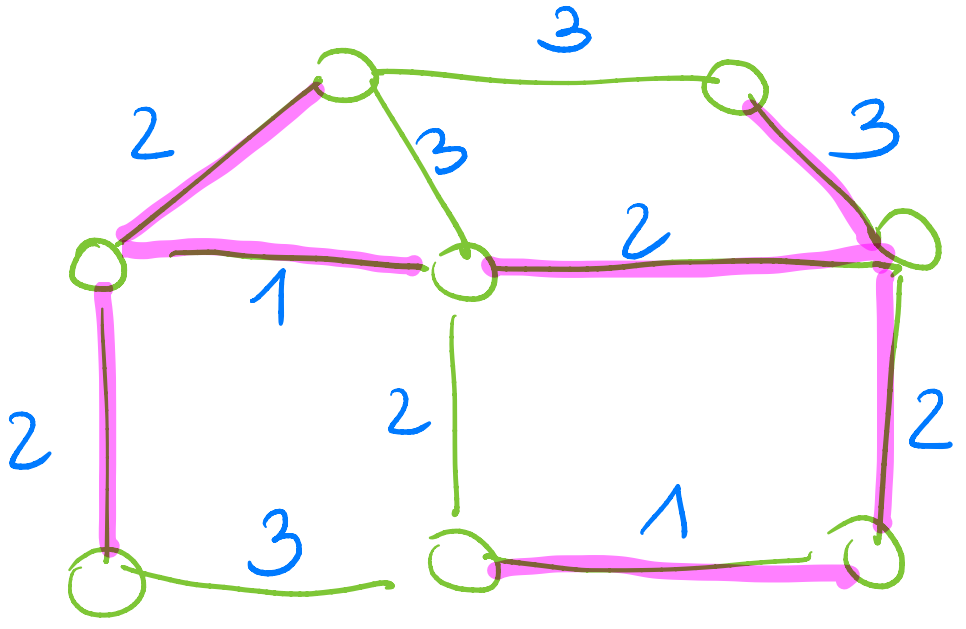
$T \rightarrow$ costo$(T) = \sum_{(i,j) \in T} W(i,j)$

spanning tree

MST T se $\forall$ spanning tree T': costo$(T) \leq$ costo(T')

scopo é trovare un MST:

- tempo quasi lineare pur essendoci tanti spanning tree
- esempio di greedy
- applicazione nei protocolli di rete
- uso di strutture di dati "ammortizzate"

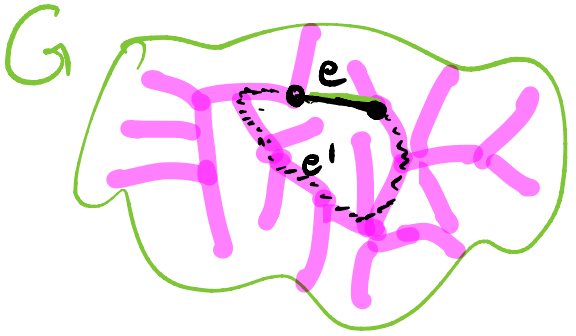


costo(T) = 13

REGOLA DEL CICLO

$T = MST(G)$

Per ogni arco $e \in E \setminus T$: $w(e) \geq w(e')$ per ogni arco $e' \neq e$ nel ciclo formatosi in $T \cup \{e\}$



$T' = T \cup \{e\} - \{e'\}$ con $w(e) < w(e')$ per assurdo

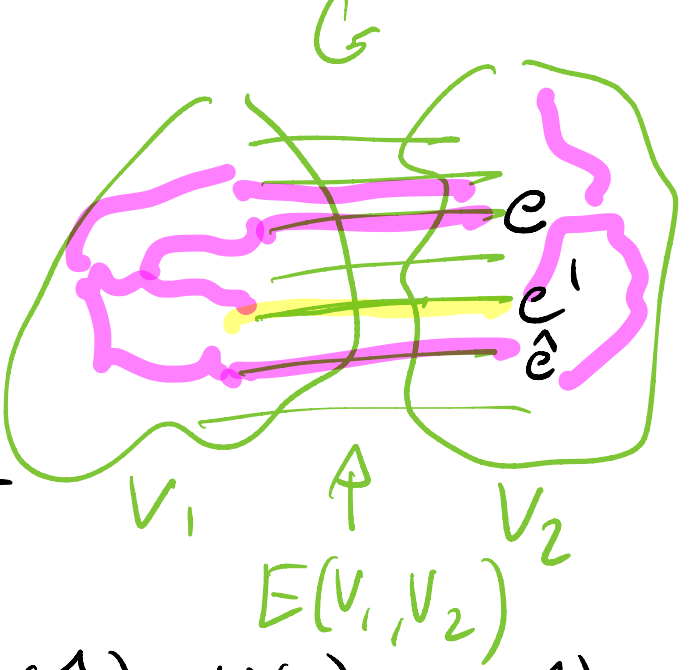
$costo(T') < costo(T)$: contraddizione

REGOLA DEL TAGLIO

$T = MST(G)$

Sia $E(V_1, V_2)$ un qualunque taglio $(V_1 \cup V_2 = V,\ V_1 \cap V_2 = \emptyset)$

$\{(u,v) \in E : u \in V_1,\ v \in V_2\}$



se $T \cap E(V_1, V_2) \neq \emptyset$

$\exists e \in T \cap E(V_1, V_2)$ t.c.

$\forall e' \in E(V_1, V_2) : w(e) \leq w(e')$

Per assurdo: $w(e') < w(e)$

Sia $\hat{e} \in T \cap E(V_1, V_2)$ l'arco che e nel ciclo creato in $T \cup \{e'\}$.

Nota: non è detto che $\hat{e} = e$, ma $w(\hat{e}) \geq w(e) \Rightarrow w(\hat{e}) > w(e')$

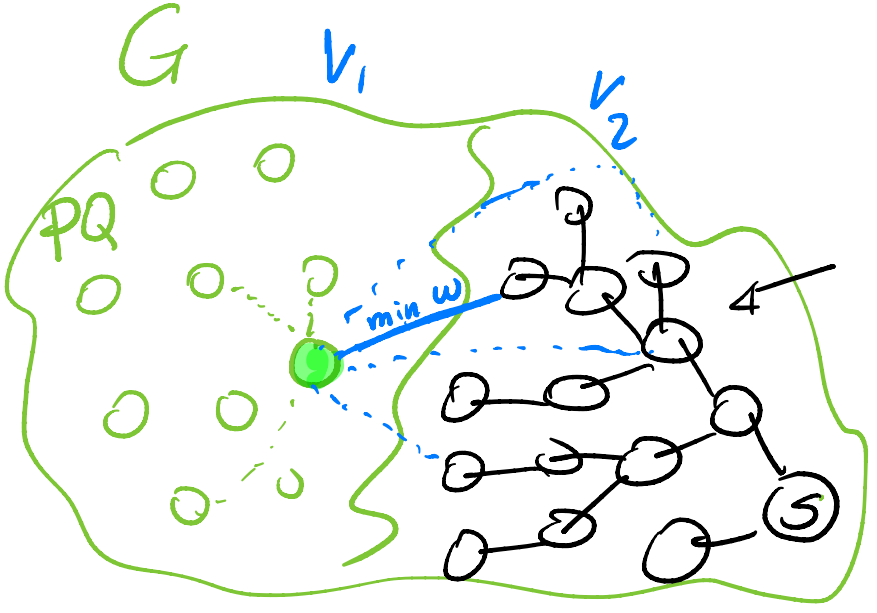
$T' = T \cup \{e'\} - \{\hat{e}\}$ ha costo strettamente minore di T!

Algoritmo Jarnik-Prim (molto simile a Dijkstra)

dist[u] = distanza da s a u
↓ peso[u] nella PQ

$$peso[u] = \min_{(u,v) \in E(V_1, V_2)} w(u,v)$$

V_1 = nodi nella PQ

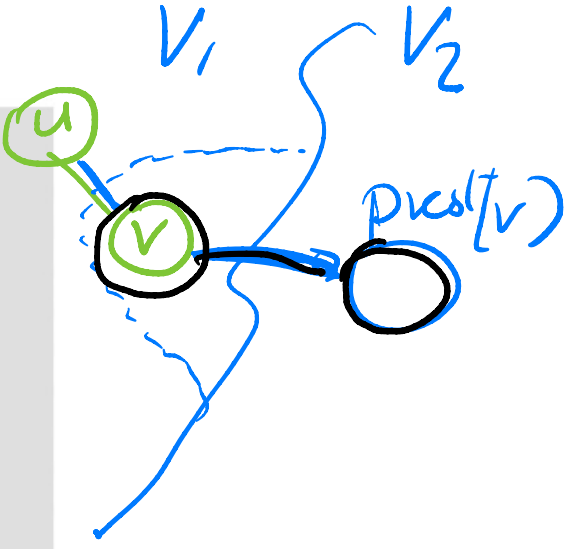


MST parzialmente costruito per questi nodi

taglio (V_1, V_2) è una partizione di V

$$E(V_1, V_2) = \{(u,v) \in E : u \in V_1, v \in V_2\}$$

```
1  Jarník-Prim( ):
2    FOR (u = 0; u < n; u = u + 1) {
3      incluso[u] = FALSE;
4      pred[u] = u;
5      elemento.peso = peso[u] = +∞;
6      elemento.dato = u;
7      PQ.Enqueue( elemento );
8    }
9    WHILE (!PQ.Empty( )) {
10     elemento = PQ.Dequeue( );
11     v = elemento.dato;
12     incluso[v] = TRUE;
13     mst.InserisciFondo( <pred[v], v> );
14     FOR (x = listaAdiacenza[v].inizio; x != null; x = x.succ) {
15       u = x.dato;
16       IF (!incluso[u] && x.peso < peso[u]) {
17         pred[u] = v;
18         peso[u] = x.peso;
19         PQ.DecreaseKey( u, peso[u] );
20       }
21     }
22   }
```



$\in$ MST

costruisce l'albero

$\langle +\infty, u \rangle$

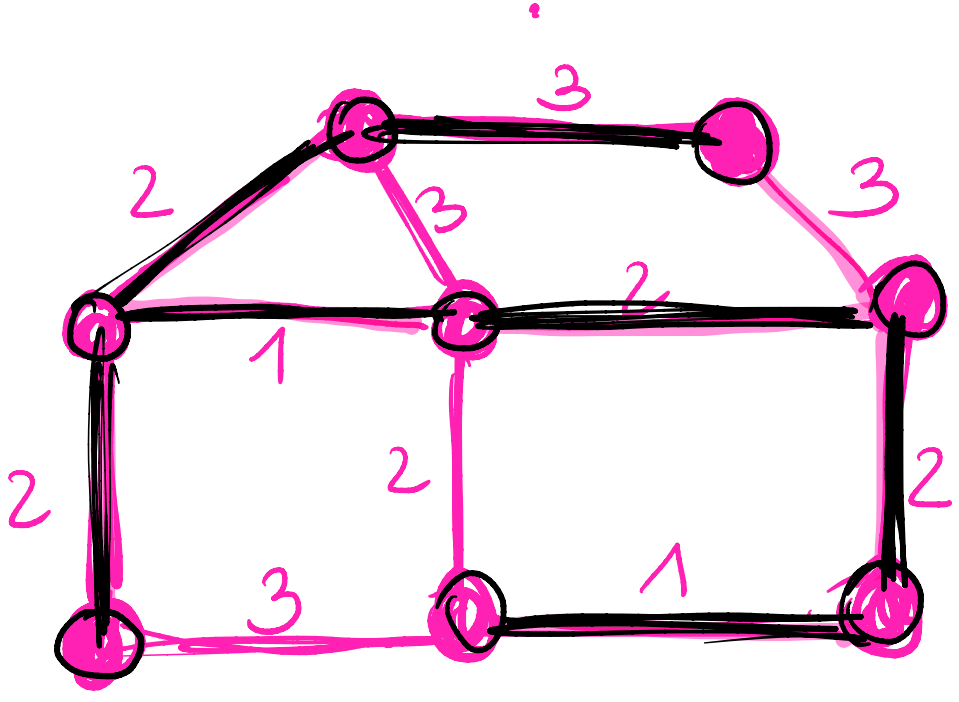
MIN

T come vector di archi

regola ciclo && regola taglio

$O(m \lg n)$ tempo

KRUSKAL

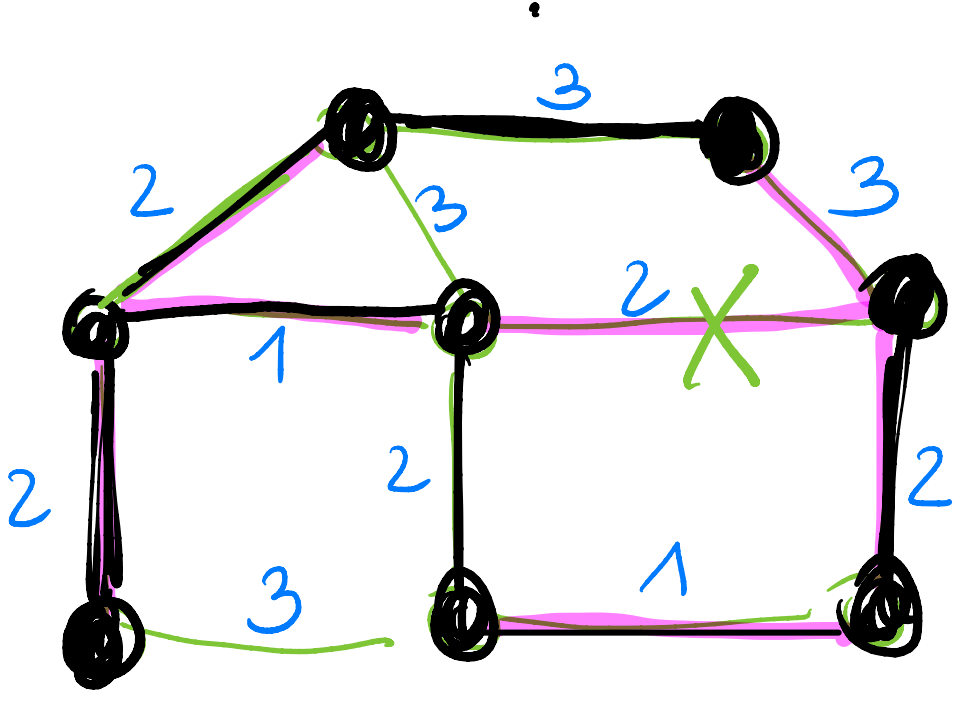


n nodi

n alberi

archi in ordine di peso non decrescente

KRUSKAL



n nodi

↓

n alberi

archi in ordine di peso non decrescente

```
1  Kruskal( ):
2    FOR (u = 0; u < n; u = u + 1) {
3      FOR (x = listaAdiacenza[u].inizio; x != null; x = x.succ) {
4        v = x.dato;
5        elemento.dato = <u, v>;
6        elemento.peso = x.peso;
7        PQ.Enqueue( elemento );
8      }
9      set[u] = NuovoNodo( );
10     Crea( set[u] );
11   }
12   WHILE (!PQ.Empty( )) {
13     elemento = PQ.Dequeue( );
14     <u,v> = elemento.dato;
15     IF (!Appartieni( set[u], set[v] )) {
16       Unisci( set[u], set[v] );
17       mst.InserisciFondo( <u,v> );
18     }
19   }
```

ordina gli archi

falso → ciclo
vero → taglio

UNION FIND su insiemi disgiunti

UNION-FIND

n insiemi disjunti, ciascuno un singoletto

$\{0\}, \{1\}, \{2\}, \ldots, \{n-1\}$

- Unisci (u, v) ≔ S_u = insieme contenente u

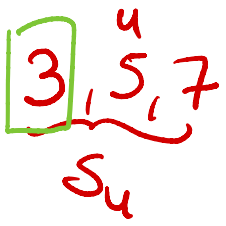
  S_v = " " v

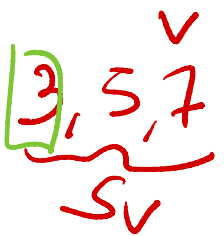
  unione disjunta distruttiva

  if $S_u \neq S_v$ then rimuovi sia S_u che S_v

  aggiungi $S_u \cup S_v$

- Appartieni: verifica se $S_u = S_v$





3 = rappresentante dell'insieme : primo del vettore

Rappresentazione:

set: = $\{u_1, u_2, \ldots, u_n\}$

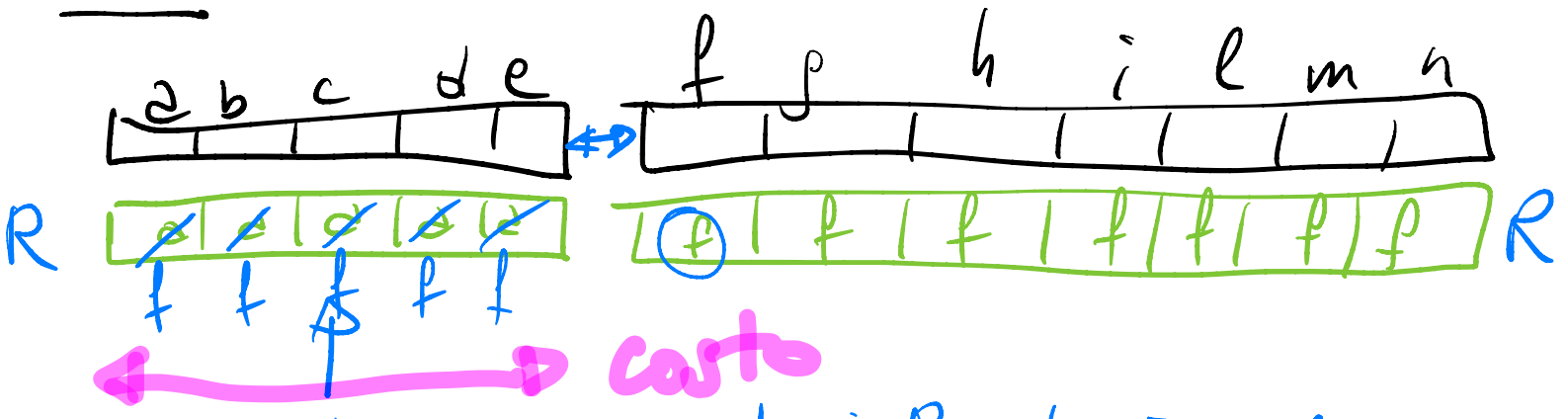
→

rappresentabile ∈

→

Appartenenza in $O(1)$ tempo

UNISCI



Cambio rappresentanti R nel più piccolo

$O(n)$ caso pessimo

$O(|\text{lista più corta}|)$ tempo

Prop Ci sono al più $n-1$ UNISCI e in totale richiedono $O(n \lg n)$ tempo

dim In ogni UNISCI, un elemento e di un insieme di taglia x, cambia appartenente se l'altro insieme ha taglia $\geq x$

e: passa da un insieme di taglia x a uno di taglia $\geq 2x$

Per induzione:
inizialmente la taglia 1 (singoletto)
dopo i cambiamenti di rappresentante in R
l'elemento sarà in un insieme di taglia $\geq 2^i$
Poiché ogni insieme ha taglia al più n,
segue che $2^i \leq n \Rightarrow i \leq \log_2 n$

MORALE: ogni elemento cambia rappresentante $O(\log n)$ volte
Costo $n-1$ UNISCI $= O($#totale di cambiamenti di rappresentante in R$)$
$= O(n \cdot \log n)$
↑ #elementi ↑ max # cambiamenti per elemento

KRUSKAL $O(|E| \log |V|)$ temp

int8_t uint_8

int_16_t

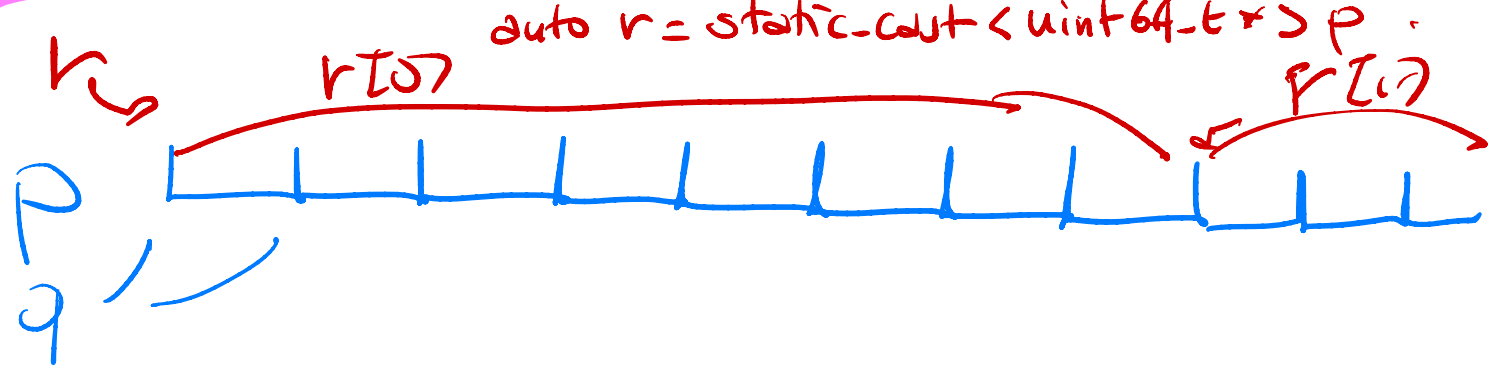
32

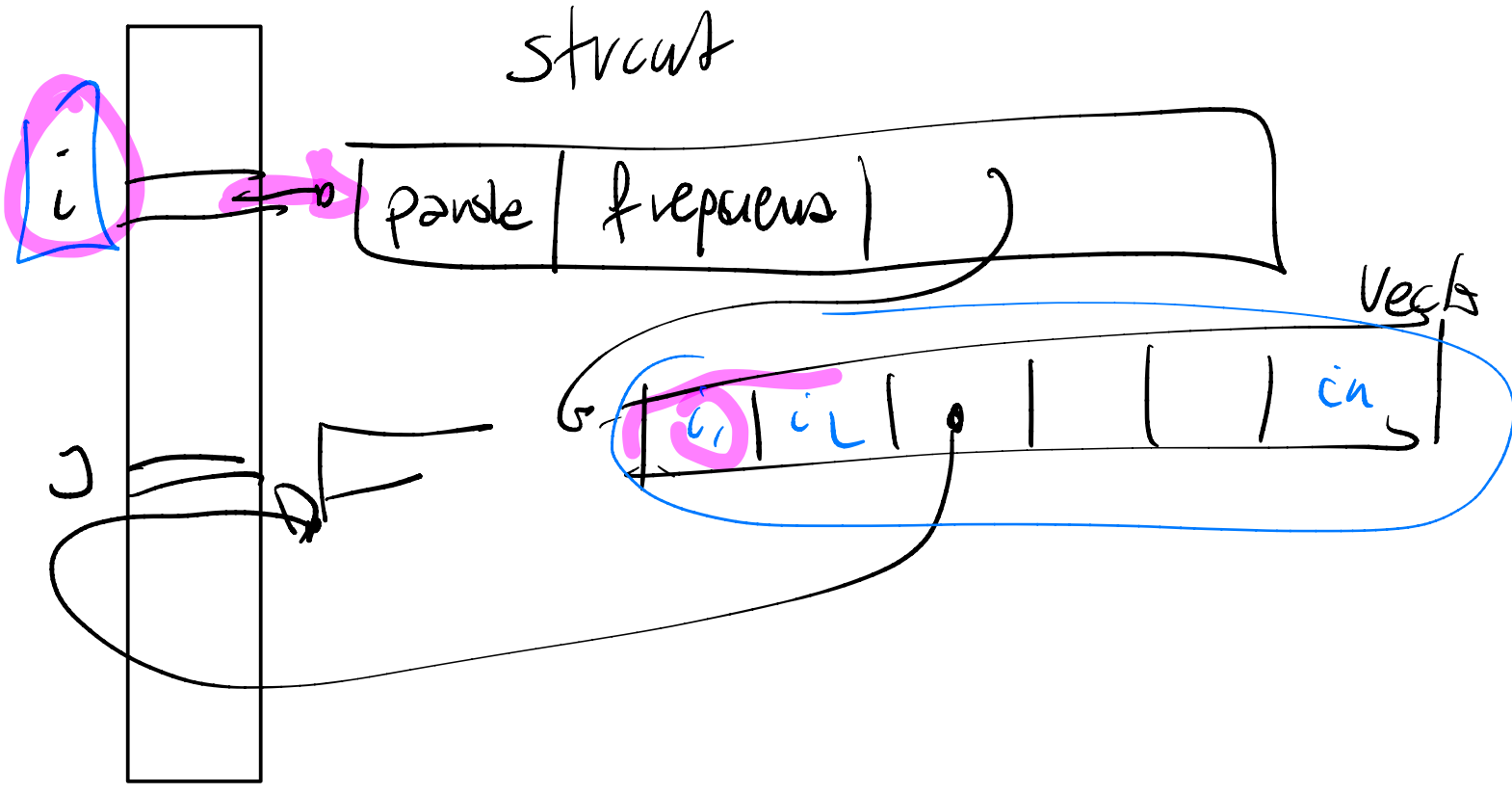
64

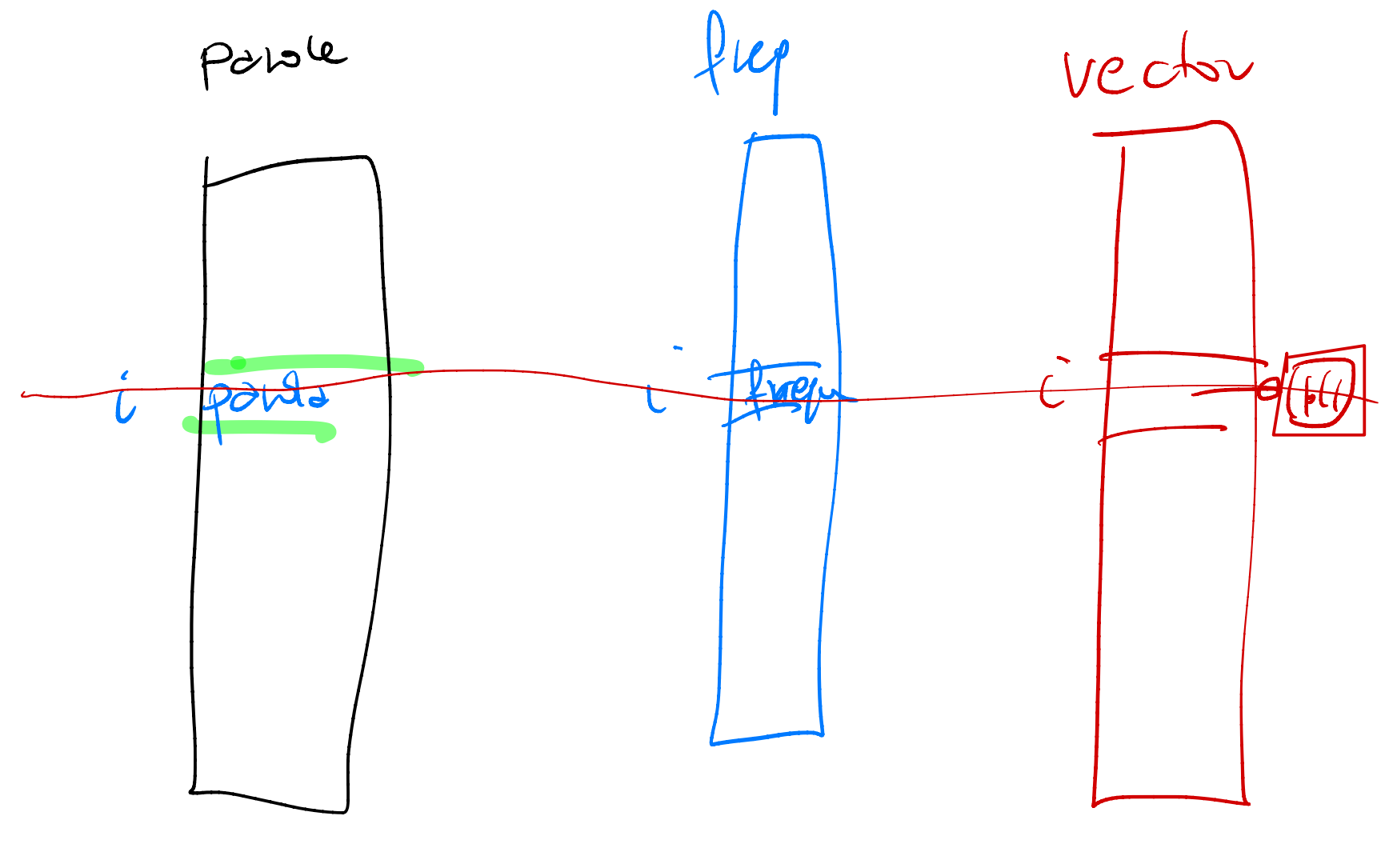
char * p;

if q[i] < 0

auto q = static_cast<int8_t *> p

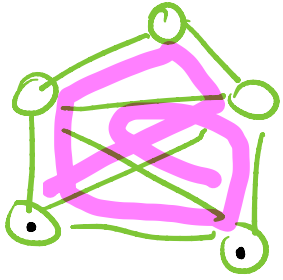



struct
i
parole
frequence
Vect
j
ci


parole
freq
vecteur
i
parole
i
freq
i

(INTRATTABILITÀ COMPUTAZIONALE

(trattabile = richiede tempo polinomiale)



ciclo euleriano
"
attraversa tutti gli archi una sola volta

$n = V$

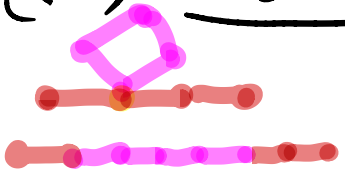
$G = (V, E)$ connesso

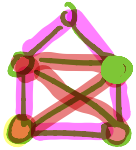
$O(poly(n))$
↳ $O(m+n)$

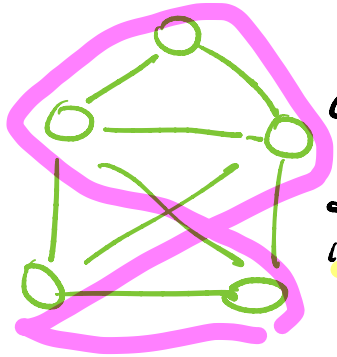
G euleriano sse tutti i nodi tranne due (eventualmente) sono di grado pari

(⇒) immediata

(⇐) accenno







ciclo hamiltoniano
"
attraverso tutti i nodi una sola volta

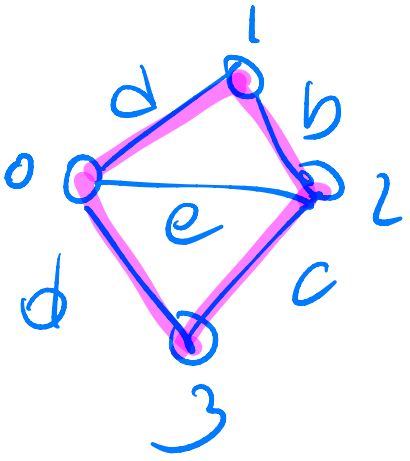
Genera tutte le $n!$ permutazioni dei nodi

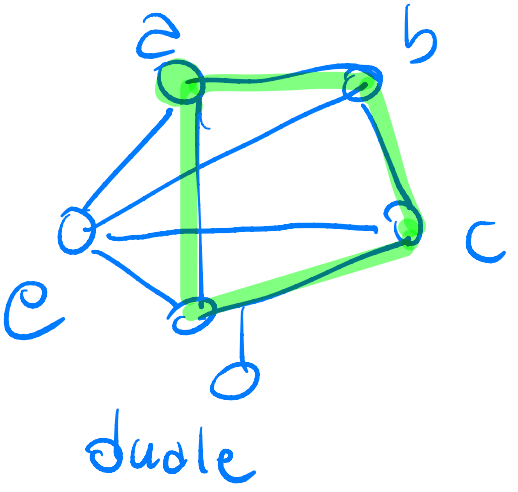
Verifica se esiste una permutazione $v_1 v_2 \dots v_n$ t.c. $(v_i, v_{i+1}) \in E$ ed (eventualmente) $(v_n, v_1) \in E$

costo esponenziale

D1: ma è esponenziale? boh

D2: ma allora è polinomiale? boh





dual e

Problemi decisionali → SI / NO

Ogni input può essere visto come una (opportuna) sequenza binaria in Σ^*, $\Sigma = \{0,1\}$

Problema decisionale $\Pi \subseteq \Sigma^*$

$$\Pi = \{x \in \Sigma^* \mid \Pi(x) = \text{sì}\}$$

Esempio: $G = (V,E)$ può essere visto come una sequenza binaria:
prendiamo la matrice di adiacenza di G e concateniamo le sue righe ⇒ sequenza di $|V|^2$ bit

HAM = problema di stabilire se G è hamiltoniano

$$HAM = \{x \in \Sigma^{n^2} \mid \text{il grafo ottenuto da } x \text{ come matrice d'adiacenza è hamiltoniano}\} \subseteq \Sigma^*$$

Classi P e NP

$\Pi \in P$ sse esiste un algoritmo deterministico polinomiale $|x|$ per stabilire se $x \in \Pi$

es. EULERIANO $\in P$, SORTING (decisionale) $\in P$, RICERCA BINARIA $\in P$

$\Pi \in NP$ sse esiste un verificatore V polinomiale per Π:

↑ non-deterministico polinomiale

$\forall x \in \Sigma^*$ se $x \in \Pi$ allora $\exists\, y \in \Sigma^*$, $|y| = poly(|x|)$ t.c. $V(x,y) = SI$ (certificato polinomiale)

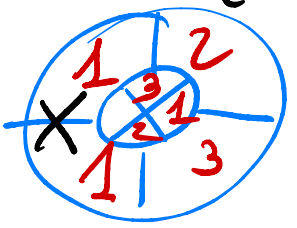
se $x \notin \Pi$ allora $\forall y \in \Sigma^* : V(x,y) = NO$

$P \subseteq NP$: un problema in P non solo si verifica in tempo poly ma si trova la soluzione in tempo poly

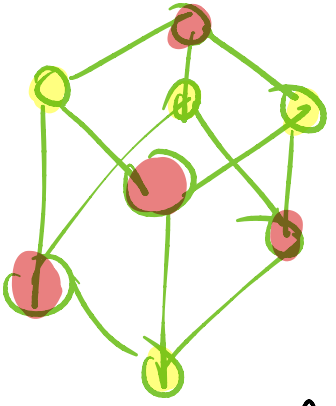
$P \supseteq NP$: BIG OPEN PROBLEM

Colorazione mappe planari ; colore diverso a paesi confinanti

K = # colori

$K=2$	$K=3$	$K \geq 4$
Sì/No	Sì/No	Sì
Paese = nodo Paese i confinante con paese j $\Leftrightarrow$ arco (i,j) polinomiale (pensateci!)	NP (completo)  Colori = $\{1,2,3\}$	Famoso teorema Appel-Haken '77 Ogni mappa planare è 4-colorabile poly

$k=2$



assegna colore opposto a quello del nodo corrente

L'assegnamento di colori fallisce se cerchiamo di assegnare due colori diversi allo stesso nodo

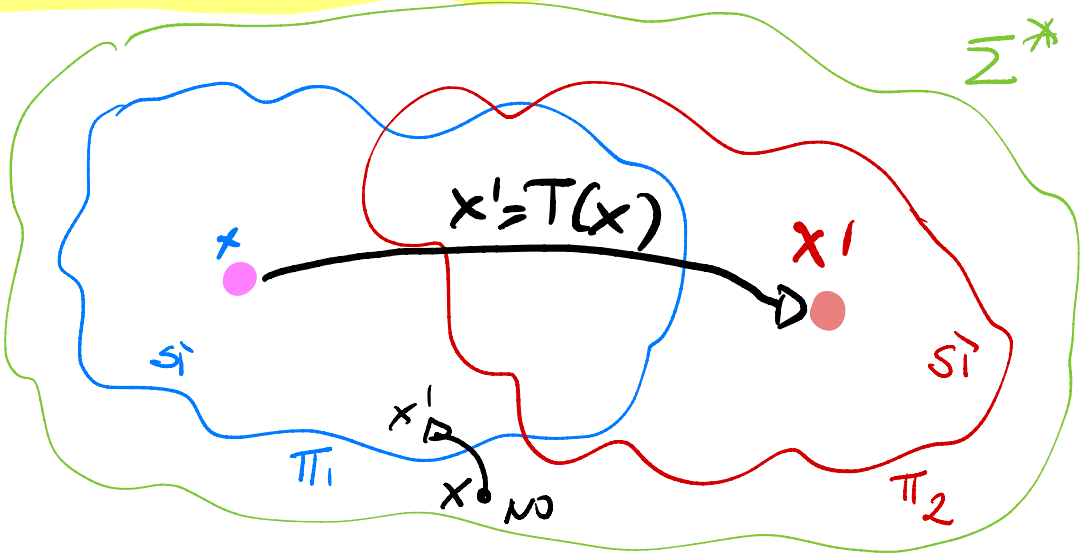
L'unica decisione è il colore del primo nodo da cui si parte, ma non cambia l'esito perché sono 2 colori!

RIDUZIONE / TRASFORMAZIONE POLINOMIALE

$\Pi_1, \Pi_2 \in NP$ $\qquad$ $\Pi_1 \propto \Pi_2$

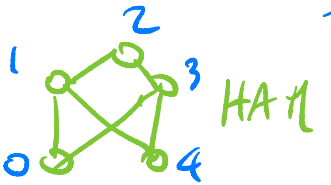
se esiste un algoritmo deterministico polinomiale $T(x)$

t.c. $\forall x \in \Sigma^*: \; x \in \Pi_1 \Leftrightarrow \underbrace{T(x)}_{x'} \in \Pi_2$ $\qquad$ (**)



Nota: Non è richiesto che sia iniettiva o suriettiva

Esempio (non usiamo direttamente Σ^* ma ciò che rappresentano le sue stringhe)



Π_1 = HAM (ciclo hamiltoniano)

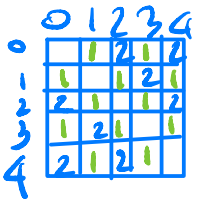
Π_2 = TSP (travel salesperson problem)

n città numerate da 1 a N

costo D_{ij} = costo per andare dalla città i alla città j

- distanza km
- pedaggio
- tempo di percorrenza
- costo del carburante

$n = 5$



tour = permutazione $c_1 c_2 \dots c_n$ delle città

$$\text{costo(tour)} = \sum_{i=1}^{n-1} D_{c_i c_{i+1}} + D_{c_n c_1}$$

$k = n = 5$

decisione: input n, D, k

output sì sse $\exists$ tour : costo(tour) $\leq k$

Trasformazione T per $\pi_1 \propto \pi_2$ NON RISOLVE!

1) Input $G = (V, E)$ (lo stesso di $\pi_1 = HAM$)

2) Output: (input per $\pi_2 = TSP$)

- $n = |V|$
- $D_{ij} = \begin{cases} 0 & \text{se } i=j \\ 1 & \text{se } ij \in E \\ 2 & \text{altrimenti} \end{cases}$
- $k = |V|$

T è un algoritmo deterministico polinomiale (*)

$O(|V|^2)$ tempo

Per vedere (*)

G hamiltoniano $\Leftrightarrow$ esiste tour di costo $\leq k = n$ in D

($\Rightarrow$) G hamiltoniano $\Rightarrow$ $\exists$ permutazione dei nodi $v_1 v_2 \dots v_n$ t.c. $\forall i \in [n-1]$: $v_i v_{i+1} \in E$, $v_n v_1 \in E$

$\Rightarrow$ $\exists$ tour $v_1 v_2 \dots v_n$ t.c. $D_{v_i v_{i+1}} = 1$ e $D_{v_n v_1} = 1$

$\Rightarrow$ costo(tour) $= n \leq k$ per costruzione

(un ciclo di n nodi usa n archi)

($\Leftarrow$) G non hamiltoniano $\Rightarrow$ $\forall$ permutazione $c_1 c_2 \dots c_n$ dei suoi nodi, esiste sempre una coppia ($c_i c_{i+1}$ oppure $c_n c_1$) non collegata da un arco : $\exists\, c_i c_{i+1} \notin E$ or

(altrimenti G sarebbe hamiltoniano!) $\exists\, c_n c_1 \notin E$

$\Rightarrow$ $\forall$ tour $c_1 c_2 \dots c_n$: ($\exists\, c_i c_{i+1}$ con $D_{c_i c_{i+1}} = 2$ or $\exists\, c_n c_1$ con $D_{c_n c_1} = 2$)

costo(tour) $\geq 2 + (n-1) \cdot 1 > n$ (poiché $D_{ij} \geq 1$ per $i \neq j$)

Riduzione α :

① Riflessiva : $\pi \alpha \pi$ $\quad (T = id)$

② Transitiva : $\pi_1 \alpha \pi_2 \alpha \pi_3$

$T_1 \quad T_2$

$T = T_2 \circ T_1$

$T(x) = T_2(T_1(x))$

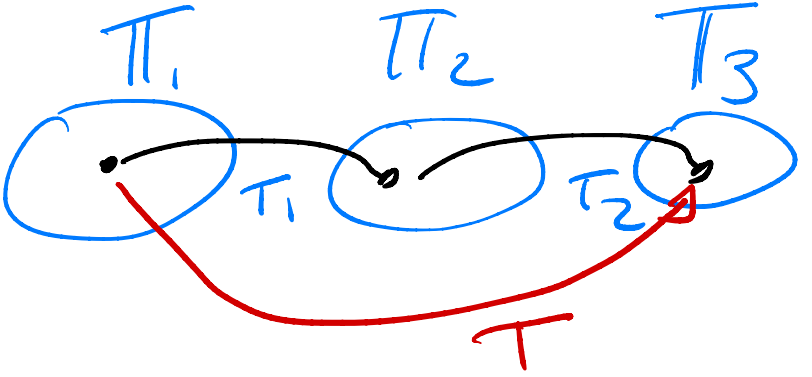
T det. polinomiale

$x \in \pi_1 \iff T(x) \in \pi_3$

$\Uparrow \qquad \Uparrow$

$T_1(x) \in \pi_2 \qquad T_2(T_1(x)) \in \pi_3$

$z = T_1(x)$



③ Simmetrica ?

$\pi_1 \alpha \pi_2$ BOH!

??? $\Rightarrow \pi_2 \alpha \pi_1$

Prop $\Pi_1 \propto \Pi_2 \quad : \quad \Pi_2 \in P \Rightarrow \Pi_1 \in P$

Sia A_2 l'alg. polinomiale per Π_2

costruiamo un algoritmo polinomiale per Π_1:

$\forall x \in \Sigma^*$ ① $x' = T(x)$ dove T è la trasform. polin. $\Pi_1 \propto \Pi_2$

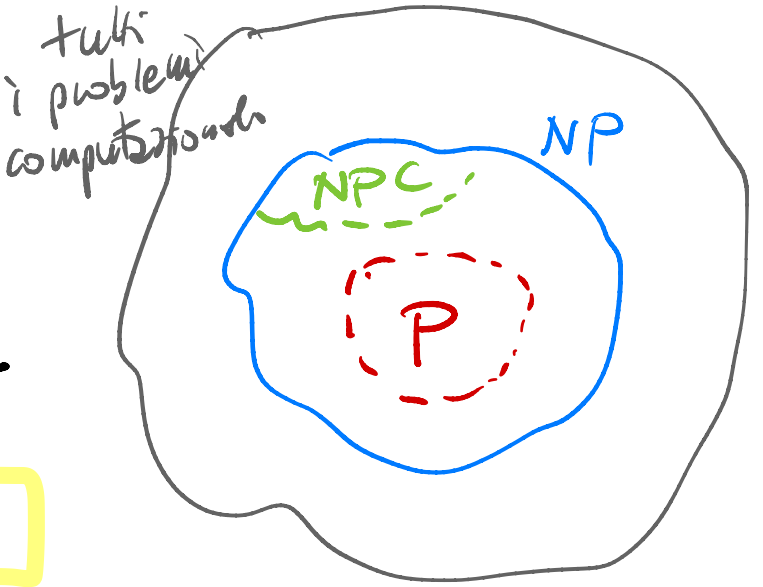
② esegui A_2 su x'

③ rispondi SÌ $\Leftrightarrow A_2(x') = \text{SÌ}$

Lettura interessante:

$\Pi_1 \notin P \Rightarrow \Pi_2 \notin P$

Cook '71, Levin '71



π è NP-completo: (NPC)

1) $\pi \in NP$
2) $\forall \pi' \in NP : \pi' \propto \pi$

Th Cook-Levin: $SAT \in NPC$

SAT: n variabili booleane $x_1 x_2 \dots x_n$, $x_i \in \{0,1\}$ (F T)

letterale: x_i, $\overline{x_i}$

clausola: $(x_i \vee \overline{x_j} \vee x_n)$ OR

formula CNF: AND di clausole: $(x_1 \vee \overline{x_4}) \wedge (x_2 \vee \overline{x_3} \vee x_4)$

$F(x_1, \dots, x_n)$ — C_1 — C_2

F è soddisfacibile se $\exists$ un assegnamento di verità $\tau : [n] \to \{0,1\}$

A.c. alla variabile x: viene assegnato il valore $\tau(i)$

$$F = (x_1 \vee x_2) \wedge (x_3 \vee \overline{x_4} \vee x_5) \wedge (\overline{x_1} \vee \overline{x_3} \vee \overline{x_5})$$

$$\tau : x_1 \leftarrow 0 \quad x_2 \leftarrow 1 \quad x_3 \leftarrow 0 \quad x_4 \leftarrow 0 \quad x_5 \leftarrow 1$$

SAT: F, n

stabilire se F è soddisfacibile

a sono 2^n possibili τ

Karp '72

$\pi \in NPC$:

1) $\pi \in NP$

2) $\exists \pi^* \in NPC$ t.c. $\pi^* \propto \pi$

↳ siccome vale la proprietà transitiva

$\forall \pi' \in NP : \pi' \propto \pi^*$ perché $\pi^* \in NPC$

$\pi^* \propto \pi \Rightarrow \forall \pi' \in NP : \pi' \propto \pi$

La proprietà simmetrica vale per i problemi in NPC

PROGETTO: Pangenome graph (bioinformatica)

BASEE {A, T, C, G}

t → ACCTCG
TGGAGC ←

ORIENTAZIONE

ALLINEAMENTO

segmento differisce

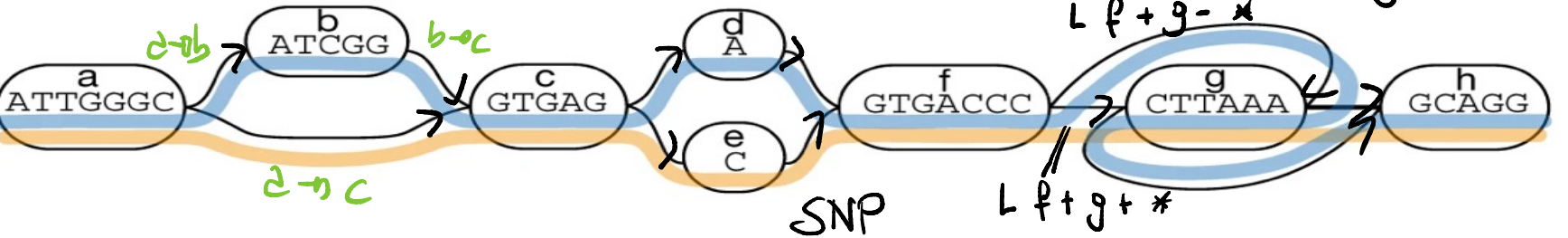
differisce

differisce

"differisce"

a

ATTGGGCATCGGGTGAGAGTGACCCTTTAAGGCAGG genoma 1
ATTGGGC-----GTGAGCGTGACCCCTTAAAGCAGG genoma 2



Rappresentazione in formato GFA: Graphical Fragment Assembly

```
H   VN:Z:1.0
S   s11   GAT
S   s12   T
S   s13   A
S   s14   CAG
S   unused      GAA
S   s15   A
S   s16   T
S   s17   TA
L   s11   +     s12   +     *
L   s11   +     s13   +     *
L   s12   +     s14   +     *
L   s13   +     s14   +     *
L   s14   +     s15   +     *
L   s14   +     s16   +     *
L   s15   +     s17   +     *
L   s16   +     s17   +     *
P   A     s11+,s12+,s14+,s15+,s17+    *,*,*,*
W   sample      1     A     0     10    >s11>s12>s14>s15>s17
W   sample      2     A     0     10    >s11>s13>s14>s16>s17
```

nome nodo

segmento associato

n=8 nodi

S=segment

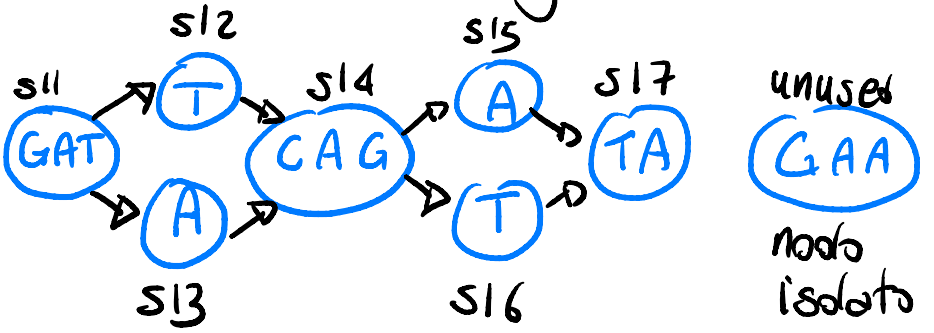
orientazione

*overlap non specificato (interpretato ØM → no overlap; xM → overlap di x basi)

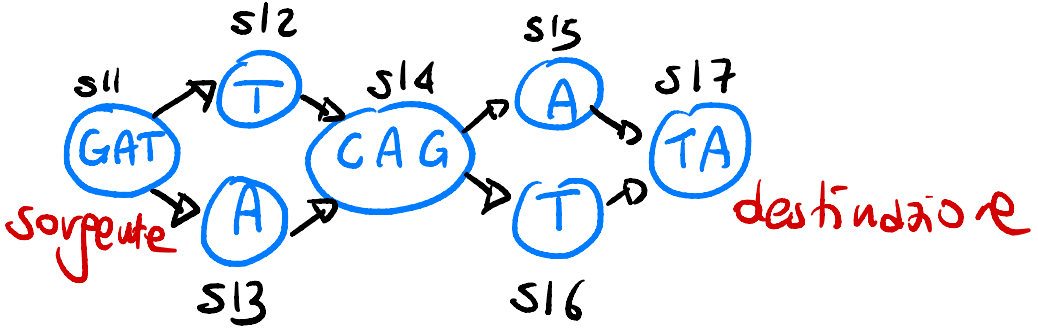
m=8 archi

archi

L=link



- formato testuale diviso in linee, con campi separati 'tab'
- prima linea inizia H = header
- le altre linee: S, L, P, W (J, ecc.)



- cammini da sorgenti (nodo con grado d'ingresso = 0) a destinazioni (con grado d'uscita = 0)
- per ogni cammino, la sequenza corrispondente si ottiene concatenando i segmenti dei nodi attraversati (tenendo conto di eventuali overlap)

- GATTCAGATA
- GATACAGATA

- GATTCAGTTA
- GATACAGTTA

il numero di sequenze può essere esponenziale nel numero di nodi

P A s11+,s12+,s14+,s15+,s17+ *,*,*,* → OVERLAP

W sample 1 A 0 10 >s11>s12>s14>s15>s17

W sample 2 A 0 10 >s11>s13>s14>s16>s17

nome

haplotype index

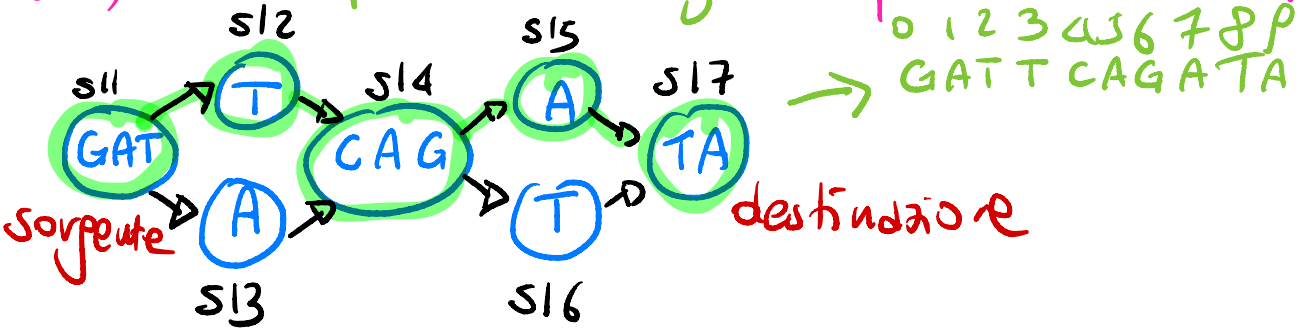
[start, end)

P = PATH

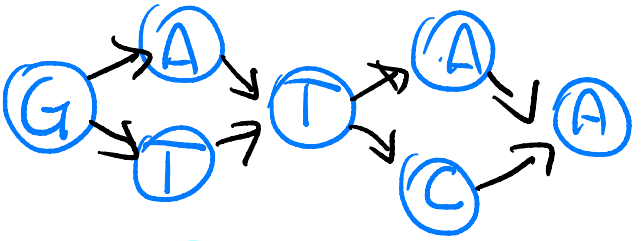
W = WALK "PATH senza overlap e jump"

da quale path si prende la stringa

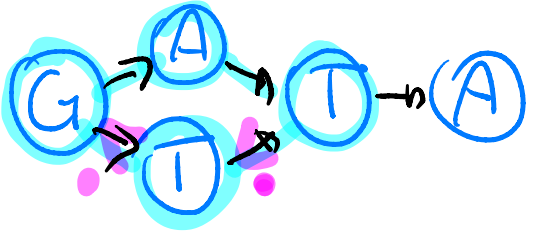
sequenza di nodi con l'orientazione dei rispettivi segmenti



```
H	VN:Z:1.0
S	11	G
S	12	A
S	13	T
S	14	T
S	15	A
S	16	C
S	17	A
S	21	G
S	22	A
S	23	T
S	24	T
S	25	A
L	11	+	12	+	*
L	11	+	13	+	*
L	12	+	14	+	*
L	13	+	14	+	*
L	14	+	15	+	*
L	14	+	16	+	*
L	15	+	17	+	*
L	16	+	17	+	*
L	21	+	22	+	*
L	21	+	23	+	*
L	22	+	24	+	*
L	23	+	24	-	*
L	24	+	25	+	*
P	A	11+,12+,14+,15+,17+	*,*,*,*
P	B	21+,22+,24+,25+	*,*,*
W	sample	1	A	0	5	>11>12>14>15>17
W	sample	2	A	0	5	>11>13>14>16>17
W	sample	1	B	0	5	>21>22>24<23<21
W	sample	2	B	0	4	>21>22>24>25
```



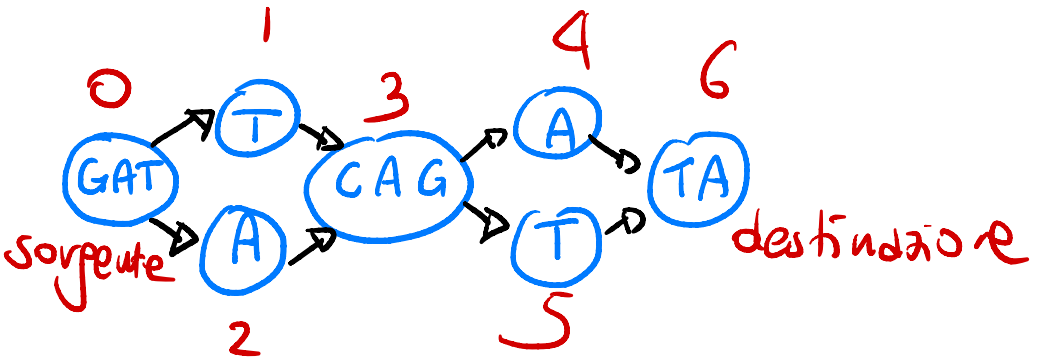
A



B

> attraversa l'arco nella sua direzione

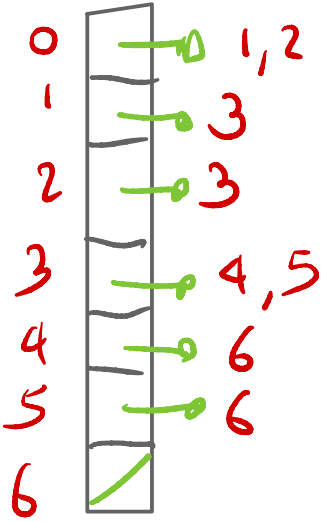
< " " nella direzione inversa



adj = array di vector

label = array di stringhe

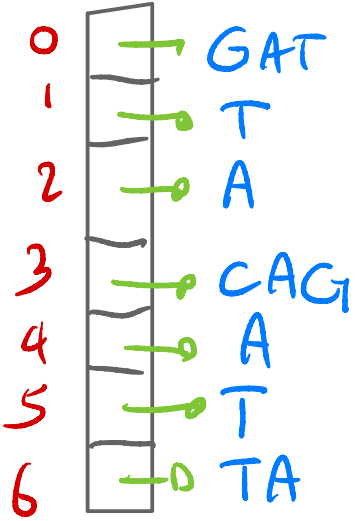
adj[] orientato



per codificare ±

usate interi negativi per −

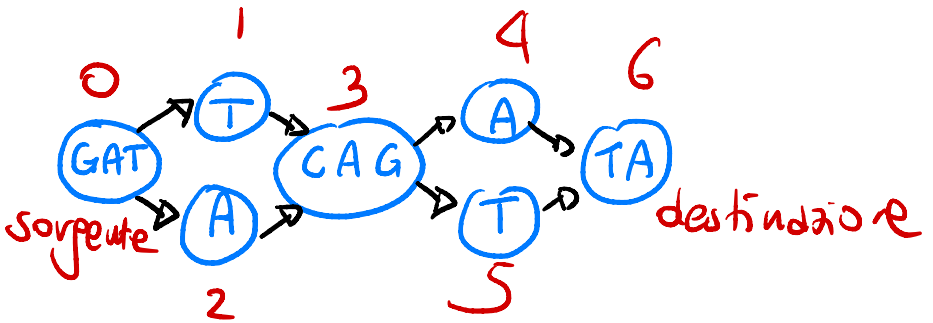
label[]



Progetto:

① leggere file .gfa e creare graf etichettato con "liste" di adiacenza
label[]
adj[]

② Considerare tutti i possibili cammini sorgente-destinazione senza materializzarli
ossia se ci sono più sorgenti e/o destinazioni scegliene una sola.



DFS sul graf non scopre tutti i cammini?

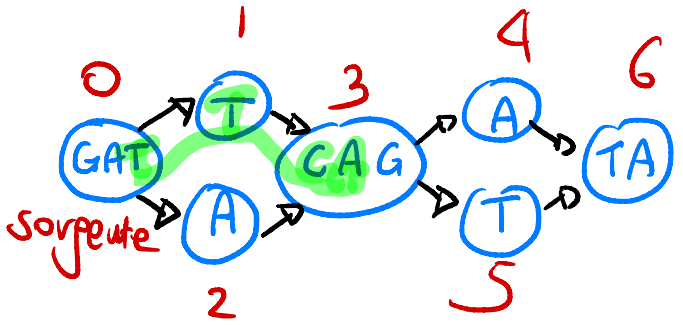
SUGGERIMENTO: siccome il graf è aciclico ignorate il boolean "visitato"

Usando un array di appoggio S, modificare la DFS(u) in modo che "appiunga/tolga" la stringa del nodo u (appiunge: inizio visita di u; toglie: fine visita di u)

Fatto interessante: quando u è destinazione (cioè grado uscita = 0), in S trovate la corrispondente sequenza.

(3) Data una sequenza pattern P, verificare che sia contenuta in una delle sequenze generate nel punto (2).

es. P = TTCA S = GATTCAGTTA



suggerimenti:

a) sfruttate l'array di appoggio S

b) sfruttate l'hash visto a lezione (rolling hash)

se $k = |P| = 4$, calcolate l'hash delle porzioni lunghe k di S e confrontatelo con l'hash (P)

Nota Se P occorre, è chiamato k-mero.
La sua frequenza è il numero di occorrenze.
Per esempio $P = ATA$ ha frequenza 2.

④ Dato k e un pangenome graph G, trovare i 10 k-meri più frequenti in G.