

Algoritmi

Un algoritmo è un procedimento che descrive una sequenza di passi ben definiti finalizzato a risolvere un dato problema (computazionale).

- **Differenza tra algoritmo e programma:**

- Un programma è la codifica di un algoritmo in un linguaggio di programmazione specifico.
- Un algoritmo è un concetto astratto indipendente dal linguaggio di programmazione, ambiente di sviluppo o sistema operativo.

L'obiettivo del corso è insegnare ad analizzare e progettare "buoni" algoritmi, ovvero:

1. **Corretti:** producono correttamente il risultato desiderato.
2. **Efficienti:** usano poche risorse di calcolo, come tempo e memoria.

Concetti principali

1. **Complessità temporale:** misura il tempo impiegato dall'algoritmo per risolvere un problema.
2. **Algoritmo:** sequenza di passi per risolvere un problema.
3. **Problema:** astrazione di un compito da svolgere.
4. **Istanza:** specifica del problema con dati concreti.
5. **Modello di calcolo:** astrazione delle risorse a disposizione per risolvere il problema.
6. **Efficienza:** rapporto tra la correttezza e la complessità di un algoritmo.
7. **Dimensione dell'istanza:** numero di elementi che caratterizza un'istanza.
8. **Caso peggiore:** la situazione più sfavorevole per l'algoritmo.
9. **Correttezza:** garanzia che l'algoritmo produce il risultato desiderato per qualsiasi istanza.

Esempio: Trovare una moneta falsa

Problema: Individuare una moneta falsa tra n monete identiche, con una bilancia a due piatti.

- **Istanza:** n monete specifiche, una delle quali è falsa.
- **Dimensione dell'istanza:** il valore di n.
- **Modello di calcolo:** bilancia a due piatti.
- **Algoritmo:** strategia di pesatura.
- **Correttezza:** l'algoritmo deve trovare la moneta falsa per qualsiasi istanza.
- **Complessità temporale:** numero di pesate necessarie per trovare la moneta falsa.

- **Complessità temporale nel caso peggiore:** numero massimo di pesate necessarie per qualsiasi istanza di dimensione n .
- **Efficienza:** l'algoritmo dovrebbe usare poche pesate.

Alg3(X)

1. **if** ($|X|=1$) **then return** unica moneta in X
2. dividi X in due gruppi X_1 e X_2 di (uguale) dimensione $k = \lfloor |X|/2 \rfloor$ e se $|X|$ è dispari una ulteriore moneta y
3. **if** peso(X_1) = peso(X_2) **then return** y
4. **if** peso(X_1) > peso(X_2) **then return** Alg3(X_1)
else return Alg3(X_2)

Corretto? sì!

pesate nel caso peggiore?

$\lfloor \log_2 n \rfloor$
(da argomentare)

efficiente? ...boh?!

*però meglio
di Alg2
☺*

Alg3: analisi della complessità

$P(n)$: # pesate che Alg3 esegue nel caso peggiore su un'istanza di dimensione n

$$P(n) = P(\lfloor n/2 \rfloor) + 1 \quad P(1) = 0$$

Oss.: $P(x)$ è una funzione non decrescente in x

$$\begin{aligned} P(n) &= P(\lfloor n/2 \rfloor) + 1 \\ &= P(\lfloor (1/2)\lfloor n/2 \rfloor \rfloor) + 2 \\ &\leq P(\lfloor n/4 \rfloor) + 2 \\ &\leq P(\lfloor n/8 \rfloor) + 3 \\ &\leq P(\lfloor n/2^i \rfloor) + i \\ &\leq P(1) + \lfloor \log_2 n \rfloor = \lfloor \log_2 n \rfloor \end{aligned}$$

Oss.: vale
 $\lfloor (1/2)\lfloor n/2 \rfloor \rfloor \leq \lfloor (1/2) n/2 \rfloor \leq \lfloor n/4 \rfloor$

quando $\lfloor n/2^i \rfloor = 1$?

per $i = \lfloor \log_2 n \rfloor$

Alg4 (X)

1. **if** ($|X|=1$) **then return** unica moneta in X
2. dividi X in tre gruppi X_1, X_2, X_3 di dimensione bilanciata
siano X_1 e X_2 i gruppi che hanno la stessa dimensione (ci sono sempre)
3. **if** peso(X_1) = peso(X_2) **then return** Alg4(X_3)
4. **if** peso(X_1) > peso(X_2) **then return** Alg4(X_1)
else return Alg4(X_2)

Corretto? sì!

pesate nel caso peggiore?

efficiente? ...boh?!

$\lceil \log_3 n \rceil$
(da argomentare)

però meglio
di Alg3
☺

Alg4: analisi della complessità

$P(n)$: # pesate che Alg4 esegue nel caso peggiore su un'istanza di dimensione n

$$P(n) = P(\lceil n/3 \rceil) + 1 \quad P(1) = 0$$

Oss.: $P(x)$ è una funzione non decrescente in x

sia k il più piccolo intero tale che $3^k \geq n$ $n' = 3^k$

$$\xrightarrow{k \geq \log_3 n} \xrightarrow{k \text{ intero}} k = \lceil \log_3 n \rceil$$

$$P(n) \leq P(n') = k = \lceil \log_3 n \rceil$$

$$\begin{aligned} P(n') &= P(n'/3) + 1 \\ &= P(n'/9) + 2 \end{aligned}$$

$$\begin{aligned} &= P(n'/3^i) + i \\ &= P(1) + k = k \quad \text{per } i=k \end{aligned}$$

Lower bound

Teorema: qualsiasi algoritmo che individua correttamente la moneta falsa tra n monete deve effettuare nel caso peggiore almeno $\lceil \log_3 n \rceil$ pesate.

La dimostrazione usa argomentazioni matematiche per mostrare che un qualsiasi algoritmo corretto deve avere una certa complessità temporale nel caso peggiore. La dimostrazione utilizza la tecnica dell'albero di decisione.

Nota: L'algoritmo 4 è un algoritmo ottimo per questo problema.

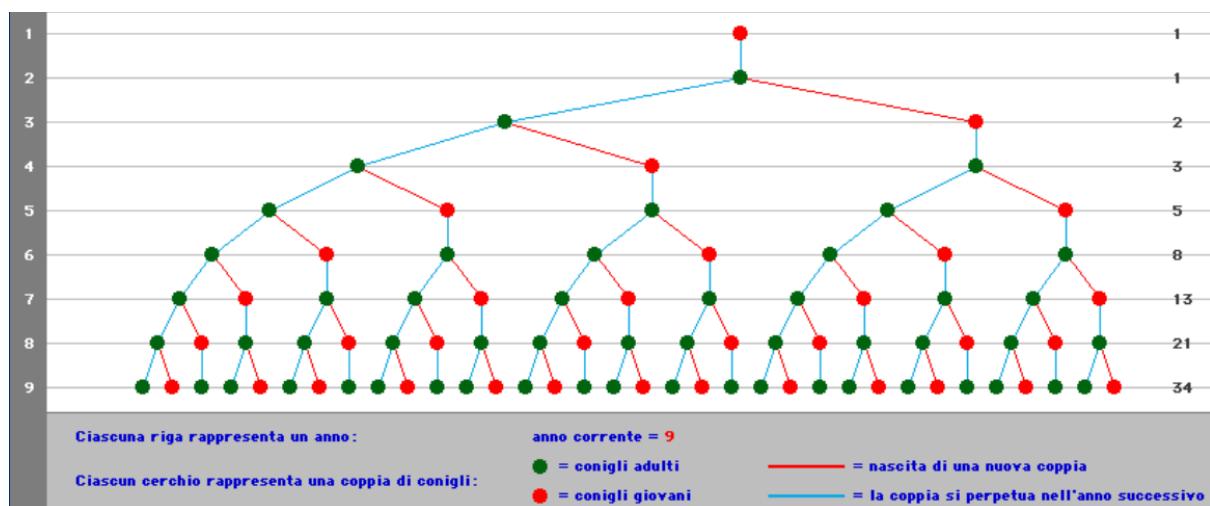
I numeri di Fibonacci

Scenario

Immagina una popolazione di conigli in condizioni ideali: una coppia di conigli si riproduce ogni anno, generando due nuovi conigli (un maschio e una femmina) ogni volta. I conigli non muoiono mai e la gestazione dura solo un anno. In questo scenario, quanto velocemente si espanderebbe la popolazione?

L'albero dei conigli

Per visualizzare la crescita della popolazione, possiamo utilizzare un albero genealogico:



- Ogni nodo rappresenta una coppia di conigli.
 - I figli di un nodo rappresentano le coppie generate da quella coppia nell'anno successivo.

La regola di espansione

Ogni anno, il numero di coppie di conigli aumenta in base al numero di coppie presenti negli anni precedenti. Possiamo descrivere questa crescita con una relazione di ricorrenza:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{se } n \geq 3 \\ 1 & \text{se } n = 1, 2 \end{cases}$$

dove:

- F_n rappresenta il numero di coppie di conigli nell'anno n
- F_{n-1} rappresenta il numero di coppie di conigli nell'anno $n-1$
- F_{n-2} rappresenta il numero di coppie di conigli nell'anno $n-2$

Calcolare F_n

Esistono due approcci per calcolare il valore di F_n

Approccio numerico

Possiamo utilizzare una funzione matematica per calcolare direttamente i numeri di Fibonacci:

$$F_n = \frac{1}{\sqrt{5}} \cdot (\phi^n - \hat{\phi}^n)$$

dove:

- $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$
- $\hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.618$

Questo metodo è efficiente per calcolare singoli valori di F_n , ma diventa meno efficiente quando si desidera calcolare una serie di valori consecutivi.

Algoritmo Fibonacci 1

```
algoritmo fibonacci1(intero n) → intero
    return  $\frac{1}{\sqrt{5}} \left( \phi^n - \hat{\phi}^n \right)$ 
```

Correttezza

L'algoritmo Fibonacci 1 è basato sulla formulazione matematica dei numeri di Fibonacci utilizzando i numeri aurei ϕ e ϕ^{\wedge} . Tuttavia, per ottenere un risultato corretto, la precisione di ϕ e ϕ^{\wedge} è cruciale. Utilizzando solo 3 cifre decimali, come nell'esempio fornito:

$$\phi \approx 1.618 \quad \phi^{\wedge} \approx -0.618$$

si otterranno discrepanze con il valore reale di F_n per valori di n sufficientemente grandi a causa dell'arrotondamento.

Algoritmo Fibonacci 2

```
algoritmo fibonacci2(intero n) → intero
  if (n≤2) then return 1
  else return fibonacci2(n-1) + fibonacci2(n-2)
```

Correttezza

L'algoritmo Fibonacci 2 implementa la definizione ricorsiva dei numeri di Fibonacci è corretto, in quanto segue la definizione esatta dei numeri di Fibonacci.

Efficienza

Tuttavia, l'algoritmo Fibonacci 2 non è efficiente. Il tempo di esecuzione, misurato come il numero di linee di codice eseguite, cresce in modo esponenziale con il valore di n .

Per calcolare il tempo di esecuzione, consideriamo un modello di calcolo rudimentale in cui ogni linea di codice costa un'unità di tempo.

- Se $n \leq 2$, l'algoritmo esegue una sola linea di codice.
- Se $n = 3$, l'algoritmo esegue 4 linee di codice: 2 per la chiamata `Fibonacci2(3)`, 1 per `Fibonacci2(2)` e 1 per `Fibonacci2(1)`.

In generale, la relazione di ricorrenza per il tempo di esecuzione è:

$$T(n) = 2 + T(n-1) + T(n-2)$$

L'analisi di questa relazione di ricorrenza dimostra che il tempo di esecuzione cresce in modo esponenziale con n . Ad esempio, per $n=45$, l'algoritmo impiega circa 3.4 miliardi di linee di codice.

Algoritmo Fibonacci 3

L'algoritmo Fibonacci 3 utilizza la tecnica della programmazione dinamica per memorizzare i risultati delle chiamate ricorsive in un array:

```
algoritmo fibonacci3(intero n) → intero
    sia Fib un array di n interi
    Fib[1] ← 1; Fib[2] ← 1
    for i = 3 to n do
        Fib[i] ← Fib[i-1] + Fib[i-2]
    return Fib[n]
```

L'algoritmo è corretto, in quanto memorizza e utilizza i valori calcolati in precedenza per evitare ricalcoli superflui.

Efficienza

L'algoritmo Fibonacci 3 è più efficiente dell'algoritmo Fibonacci 2. Il tempo di esecuzione è lineare con il valore di n .

- Le righe 1, 2 e 5 vengono eseguite una sola volta.
- Le righe 3 e 4 vengono eseguite al massimo n volte.

Pertanto, il tempo di esecuzione è:

$$T(n) \leq n + n + 3 = 2n + 3$$

Per $n=45$, l'algoritmo impiega circa 93 linee di codice, un notevole miglioramento rispetto all'algoritmo Fibonacci 2.

Algoritmo Fibonacci 4

L'algoritmo Fibonacci 4 ottimizza ulteriormente l'algoritmo Fibonacci 3 evitando di memorizzare tutti i valori intermedi:

```

algoritmo fibonacci4(intero n) → intero
  a ← 1, b ← 1
  for i = 3 to n do       $T(n) \leq 4n + 2$ 
    c ← a+b
    a ← b
    b ← c
  return c

```

L'algoritmo mantiene solo gli ultimi due valori di Fibonacci, *a* e *b*, e li aggiorna iterativamente.

Algoritmo Fibonacci 5

L'algoritmo Fibonacci 5 sfrutta le proprietà delle matrici per calcolare i numeri di Fibonacci in modo esponenziale:

```

algoritmo fibonacci5(intero n) → intero
1.    $M \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ 
2.   for i = 1 to n - 1 do
3.      $M \leftarrow M \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ 
4.   return M[0][0]

```

Efficienza

L'algoritmo Fibonacci 5 è il più efficiente tra quelli presentati. Il tempo di esecuzione è ancora $O(n)$.

Possiamo calcolare la *n*-esima potenza elevando al quadrato la $(n/2)$ -esima potenza

- Se *n* è dispari eseguiamo una ulteriore moltiplicazione

Algoritmo fibonacci 6

```

algoritmo fibonacci6(intero n) → intero
1.    $A \leftarrow \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ 
2.    $M \leftarrow \text{potenzaDiMatrice}(A, n - 1)$ 
3.   return M[0][0]

funzione potenzaDiMatrice(matrice A, intero k) → matrice
4.   if (k = 0) return  $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ 
5.   else  $M \leftarrow \text{potenzaDiMatrice}(A, \lfloor k/2 \rfloor)$ 
6.      $M \leftarrow M \cdot M$ 
7.   if (k è dispari) then  $M \leftarrow M \cdot A$ 
8.   return M

```

L'analisi del tempo di esecuzione si basa sulla funzione `PotenzaDiMatrice` utilizzata per calcolare la matrice M_{n-1} . La relazione di ricorrenza per il tempo di esecuzione è: $T(n) \leq T(\lfloor 2n \rfloor) + c$ dove c rappresenta il tempo impiegato per una singola moltiplicazione di matrici.

Utilizzando il metodo dell'iterazione, si ottiene: $T(n) \leq c + \lfloor \log_2 n \rfloor + T(1) = O(\log_2 n)$

Pertanto, il tempo di esecuzione dell'algoritmo Fibonacci 6 cresce logaritmicamente con n , offrendo un notevole miglioramento rispetto agli algoritmi precedenti.

	Tempo di esecuzione	Occupazione di memoria
<code>fibonacci2</code>	$O(\phi^n)$	$O(n)$
<code>fibonacci3</code>	$O(n)$	$O(n)$
<code>fibonacci4</code>	$O(n)$	$O(1)$
<code>fibonacci5</code>	$O(n)$	$O(1)$
<code>fibonacci6</code>	$O(\log_2 n)$	$O(\log_2 n)$

Modelli di Calcolo

Macchina di Turing

- **Pro:**
 - Modello astratto per la calcolabilità.
- **Contro:**
 - Troppo di basso livello per i calcolatori reali.
 - Poco adatto per l'analisi di efficienza.

Macchina a Registri (RAM)

- **Caratteristiche:**
 - Programma finito.
 - Nastro di ingresso e uno di uscita.
 - Memoria strutturata come un array.
 - CELLE: contengono valori interi/reali.

- CPU per eseguire istruzioni.
- **Pro:**
 - Astrazione dell'architettura di von Neumann.
 - Più realistica della macchina di Turing.
- **Contro:**
 - Ancora un'astrazione.

Analisi di Complessità

Passi Elementari

- Unità di base per misurare la complessità di un algoritmo.
- Esempi di passi elementari su una RAM:
 - Istruzione di I/O (accesso ai nastri).
 - Operazione aritmetico/logica.
 - Accesso/modifica della memoria.

Criteri di Costo

- **Costo uniforme:**
 - Tutte le operazioni hanno lo stesso costo.
 - Complessità temporale misurata come numero di passi elementari.
- **Costo logaritmico:**
 - Il costo dipende dalla dimensione degli operandi.
 - Operazione su un operando di valore x ha costo $\log x$.
 - Modella meglio la complessità di algoritmi "numerici".
 - Meno usato rispetto al costo uniforme.

Caso Peggio e Caso Medio

Caso Peggio

- $T_{\text{worst}}(n) = \max_{\text{istanze } I \text{ di dimensione } n} \{\text{tempo}(I)\}$
- Rappresenta il tempo di esecuzione sulla "peggiore" istanza di dimensione n.
- Garantisce un limite superiore per il tempo di esecuzione.

Caso Medio

- $T_{\text{avg}}(n) = \sum_{\text{istanze } I \text{ di dimensione } n} \{P(I) \text{ tempo}(I)\}$
- Rappresenta il tempo di esecuzione "tipico" per le istanze di dimensione n.
- Difficile da calcolare senza conoscere la distribuzione di probabilità $P(I)$.
- Spesso si fanno assunzioni (talvolta non realistiche) su $P(I)$.

Notazione Asintotica: Intuizioni

Complessità computazionale

La complessità computazionale di un algoritmo viene spesso espressa con una funzione $T(n)$, che rappresenta il numero di passi elementari eseguiti su una RAM nel caso peggiore su un'istanza di dimensione n .

Idea della Notazione Asintotica

L'idea della notazione asintotica è di descrivere $T(n)$ in modo **qualitativo**, pur mantenendo l'informazione essenziale sul comportamento dell'algoritmo. In altre parole, si vuole **perdere un po' di precisione per guadagnare in semplicità**.

Esempio

Consideriamo la funzione:

```
T(n) = {  
    71 n^2 + 100 [n/4] + 7 se n è pari;  
    70 n^2 + 150 [(n+1)/4] + 5 se n è dispari  
}
```

Usando la notazione asintotica, possiamo scrivere:

$T(n) = \Theta(n^2)$

Cosa significa questo?

Intuitivamente, significa che $T(n)$ è **proporzionale a n^2** . In altre parole:

- Ignoriamo le costanti moltiplicative (come 71 o 70).
- Ignoriamo i termini di ordine inferiore (come $[n/4]$ o $[(n+1)/4]$), che crescono più lentamente di n^2 .

Nota Importante

L'utilizzo della notazione asintotica implica un'assunzione implicita: **ci interessa il comportamento dell'algoritmo su istanze grandi**. Per istanze piccole, il valore preciso di $T(n)$ potrebbe essere significativamente diverso dalla sua approssimazione asintotica.

Vantaggi della Notazione Asintotica

- Semplifica la descrizione e la comparazione di algoritmi.
- Permette di identificare i "termini dominanti" della complessità computazionale.
- È utile per capire il comportamento asintotico dell'algoritmo, ovvero come si comporta per grandi valori di n .

Esempi di Notazione Asintotica Comune

- $O(n)$: L'algoritmo è proporzionale a n .
- $\Theta(n)$: L'algoritmo è "circa" proporzionale a n (considerando solo i termini dominanti).
- $\Omega(n)$: L'algoritmo è almeno proporzionale a n .
- $O(\log n)$: L'algoritmo cresce logaritmicamente con n .
- $O(n^k)$: L'algoritmo cresce polinomialmente con n (con esponente k).

Conclusione

La notazione asintotica è uno strumento prezioso per l'analisi della complessità computazionale degli algoritmi. Permette di descrivere il comportamento degli algoritmi in modo semplice e intuitivo, pur mantenendo l'informazione essenziale sul loro comportamento asintotico.

Notazione Asintotica: O , Ω e Θ

Definizioni

Notazione O

Una funzione $f(n)$ è in $O(g(n))$ se e solo se esistono due costanti positive c e n_0 tali che:

$$0 \leq f(n) \leq c \cdot g(n) \text{ per ogni } n \geq n_0$$

Questo significa che per n sufficientemente grande, $f(n)$ è sempre **limitato superiormente** da un multiplo costante di $g(n)$.

Notazione Ω

Una funzione $f(n)$ è in $\Omega(g(n))$ se e solo se esiste una costante positiva c e un valore n_0 tale che:

$$0 \leq c \cdot g(n) \leq f(n) \text{ per ogni } n \geq n_0$$

Questo significa che per n sufficientemente grande, $f(n)$ è sempre **maggiore o uguale** a un multiplo costante di $g(n)$.

Notazione Θ

Una funzione $f(n)$ è in $\Theta(g(n))$ se e solo se è sia in $\mathcal{O}(g(n))$ che in $\mathcal{\Omega}(g(n))$. In altre parole:

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ per ogni } n \geq n_0$$

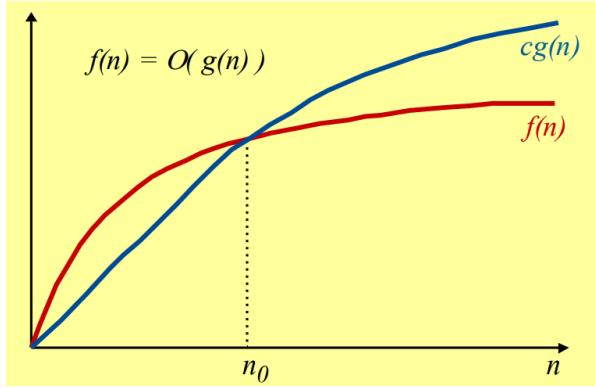
dove c_1 e c_2 sono due costanti positive e n_0 è un valore tale che le diseguaglianze valgono per tutti i valori di n superiori a n_0 .

Esempi

Funzione $f(n) = 2n^2 + 3n$

Notazione O:

- $f(n) = \mathcal{O}(n^3)$ (con $c=1$ e $n_0=3$), perché per $n \geq 3$: $0 \leq 2n^2 + 3n \leq 1 \cdot n^3$
- $f(n) = \mathcal{O}(n^2)$ (con $c=3$ e $n_0=3$), perché per $n \geq 3$: $0 \leq 2n^2 + 3n \leq 3 \cdot n^2$
- $f(n) \neq \mathcal{O}(n)$, perché per $n=1$: $f(1) = 5 > 1 \cdot 1$



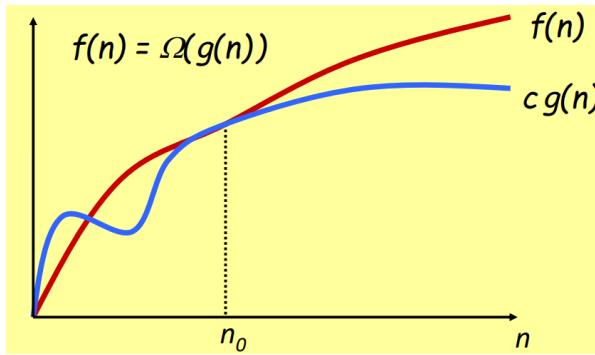
$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = O(g(n))$$

$$f(n) = O(g(n)) \not\Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = O(g(n)) \Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \text{ (se esiste)} < \infty$$

Notazione Ω :

- $f(n) = \Omega(n)$ (con $c=1$ e $n_0=2$), perché per $n \geq 2$: $0 \leq 1 \cdot n \leq 2n^2 + 3n$
- $f(n) = \Omega(n^2)$ (con $c=1$ e $n_0=3$), perché per $n \geq 3$: $0 \leq 1 \cdot n^2 \leq 2n^2 + 3n$
- $f(n) \neq \Omega(n^3)$, perché per $n=1$: $f(1)=5 < 1 \cdot 1^3$



$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \Omega(g(n))$$

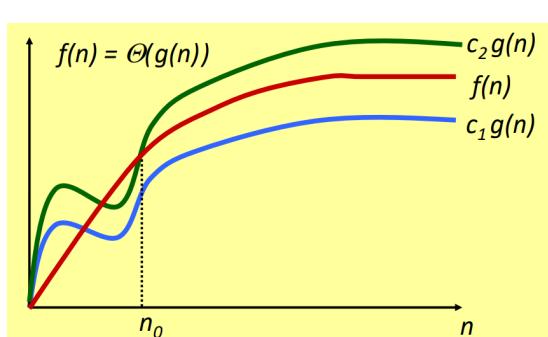
$$f(n) = \Omega(g(n)) \not\Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

$$f(n) = \Omega(g(n)) \Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \text{ (se esiste)} > 0$$

Notazione Θ :

- $f(n) = \Theta(n^2)$ (con $c_1=1$ e $c_2=2$ e $n_0=3$), perché: $1 \cdot n^2 \leq 2n^2 + 3n \leq 2 \cdot n^2$ per ogni $n \geq 3$

- $f(n) \neq \Theta(g(n))$, perché per $n = 1$: $f(1) = 5 > 1 \cdot 1$
- $f(n) \neq \Theta(n^3)$, perché per $n = 1$: $f(1) = 5 < 1 \cdot 1^3$



$$\begin{array}{ll} f(n) = \Theta(g(n)) & \Rightarrow f(n) = O(g(n)) \\ f(n) = O(g(n)) & \not\Rightarrow f(n) = \Theta(g(n)) \end{array}$$

$$\begin{array}{ll} f(n) = \Theta(g(n)) & \Rightarrow f(n) = \Omega(g(n)) \\ f(n) = \Omega(g(n)) & \not\Rightarrow f(n) = \Theta(g(n)) \end{array}$$

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \Omega(g(n)) \text{ e } f(n) = O(g(n))$$

Notazione Asintotica: o e ω

Notazione $o(g(n))$

La notazione $o(g(n))$ rappresenta l'insieme di tutte le funzioni $f(n)$ che **crescono più lentamente** di $g(n)$ **all'infinito**. In altre parole:

Definizione:

- $f(n) = o(g(n))$ se e solo se per ogni costante positiva c esiste un valore n_0 tale che per tutti i valori di n maggiori o uguali a n_0 :
- $$0 \leq f(n) < c g(n)$$

Proprietà:

- $o(g(n)) \subseteq o(g(n))$ per qualsiasi funzione $g(n)$

Definizione alternativa:

- $f(n) = o(g(n)) \iff \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$

Notazione $\omega(g(n))$

La notazione $\omega(g(n))$ rappresenta l'insieme di tutte le funzioni $f(n)$ che **crescono più velocemente** di $g(n)$ **all'infinito**. In altre parole:

Definizione:

- $f(n) = \omega(g(n))$ se e solo se per ogni costante positiva c esiste un valore n_0 tale che per tutti i valori di n maggiori o uguali a n_0 :

$$0 < c g(n) < f(n)$$

Proprietà:

- $\omega(g(n)) \subseteq O(g(n))$ per qualsiasi funzione $g(n)$

Definizione alternativa:

- $f(n) = \omega(g(n))$ se e solo se il limite del rapporto $f(n)/g(n)$ tende a infinito all'infinito:

$$\bullet \quad f(n) = \omega(g(n)) \iff \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

Riassumendo

$$f(n) = \Theta(g(n)) \iff 0 < c_1 \leq \frac{f(n)}{g(n)} \leq c_2 < \infty \quad \text{asintoticamente}$$

$$f(n) = O(g(n)) \iff \frac{f(n)}{g(n)} \leq c_2 < \infty \quad \text{asintoticamente}$$

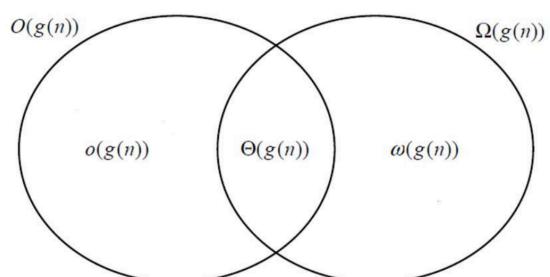
$$f(n) = \Omega(g(n)) \iff 0 < c_1 \leq \frac{f(n)}{g(n)} \quad \text{asintoticamente}$$

$$f(n) = o(g(n)) \iff \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = \omega(g(n)) \iff \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

Graficamente

O	Ω	Θ	O	ω
\leq	\geq	$=$	$<$	$>$



Proprietà della notazione asintotica

Transitività

$$\begin{array}{lllll} f(n) = \Theta(g(n)) & e & g(n) = \Theta(h(n)) & \Rightarrow & f(n) = \Theta(h(n)) \\ f(n) = O(g(n)) & e & g(n) = O(h(n)) & \Rightarrow & f(n) = O(h(n)) \\ f(n) = \Omega(g(n)) & e & g(n) = \Omega(h(n)) & \Rightarrow & f(n) = \Omega(h(n)) \\ f(n) = o(g(n)) & e & g(n) = o(h(n)) & \Rightarrow & f(n) = o(h(n)) \\ f(n) = \omega(g(n)) & e & g(n) = \omega(h(n)) & \Rightarrow & f(n) = \omega(h(n)) \end{array}$$

Riflessività

$$\begin{array}{l} f(n) = \Theta(f(n)) \\ f(n) = O(f(n)) \\ f(n) = \Omega(f(n)) \end{array}$$

Simmetria

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

Simmetria trasposta

$$\begin{array}{lll} f(n) = O(g(n)) & \Leftrightarrow & g(n) = \Omega(f(n)) \\ f(n) = o(g(n)) & \Leftrightarrow & g(n) = \omega(f(n)) \end{array}$$

Polinomi

$$P(n) = a_d n^d + a_{d-1} n^{d-1} + \dots + a_0 \quad \xrightarrow{\qquad} \quad \begin{array}{l} P(n) = \Theta(n^d) \\ P(n) = O(n^d) \\ P(n) = \Omega(n^d) \end{array}$$

Esponenziali

$$\begin{array}{ll} f(n) = a^n & \xrightarrow{\qquad} \quad a^n = \omega(n^d) \\ a > 1 & \quad \quad \quad a^n = \Omega(n^d) \\ \lim_{n \rightarrow \infty} \frac{a^n}{n^d} = \infty & \end{array}$$

Logaritmi

$$\begin{array}{ll} f(n) = \log_b(n) \quad b > 1 & \xrightarrow{\qquad} \quad [\log_b(n)]^c = o(n^d) \\ \lim_{n \rightarrow \infty} \frac{[\log_b(n)]^c}{n^d} = 0, \forall c, d > 0 & [\log_b(n)]^c = O(n^d) \end{array}$$

Fattoriali

$$f(n) = n! = n * (n-1) * \dots * 2 * 1 \quad \xrightarrow{\qquad} \quad \begin{array}{l} n! = o(n^n) \\ n! = \omega(a^n) \end{array}$$

Analisi complessità fibonacci3: un Upper Bound

algoritmo fibonacci3(intero n) → intero

```
1 sia Fib un array di n interi
2 Fib[1] ← Fib[2] ← 1
3 for i = 3 to n do
4   Fib[i] ← Fib[i-1] + Fib[i-2]
5 return Fib[n]
```

$T(n)$: complessità computazionale nel caso peggiore con input n

c_j : #passi elementari eseguiti su una RAM quando è eseguita la linea di codice j

- linea 1, 2 e 5 eseguite una volta
- linee 3 e 4: eseguite al più n volte

$$T(n) \leq c_1 + c_2 + c_5 + (c_3 + c_4)n = \Theta(n)$$

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

Analisi complessità fibonacci3: un Lower Bound

algoritmo fibonacci3(intero n) → intero

```
1 sia Fib un array di n interi
2 Fib[1] ← Fib[2] ← 1
3 for i = 3 to n do
4   Fib[i] ← Fib[i-1] + Fib[i-2]
5 return Fib[n]
```

Nota: poiché ogni istruzione di alto livello esegue un #costante di passi elementari posso contare # di istruzioni

$T(n)$: complessità computazionale nel caso peggiore con input n

c_j : #passi elementari eseguiti su una RAM quando è eseguita la linea di codice j

la linea 4 è eseguita almeno $n-3$ volte

$$T(n) \geq c_4(n-3) = c_4n - 3c_4 = \Theta(n)$$

$$\rightarrow T(n) = \Omega(n)$$

$$\rightarrow T(n) = \Theta(n)$$

Vantaggi della Notazione Asintotica

La notazione asintotica offre diversi vantaggi per l'analisi degli algoritmi:

1. Indipendenza dall'implementazione e dalla macchina:

- La notazione asintotica descrive la complessità di un algoritmo in modo indipendente dalla sua implementazione specifica o dalla macchina su cui viene eseguito.
- Questo la rende una misura universale per confrontare le prestazioni di diversi algoritmi, indipendentemente dai dettagli di come vengono messi in pratica.

2. Ignorare i "dettagli" irrilevanti:

- Per n sufficientemente grande, le costanti moltiplicative e i termini di ordine inferiore nelle funzioni tendono a diventare insignificanti rispetto al termine dominante.
- La notazione asintotica ci permette di ignorare questi "dettagli" e concentrarci sul comportamento a lungo termine dell'algoritmo.

3. Semplicità e chiarezza:

- L'analisi dettagliata del numero di passi realmente eseguiti da un algoritmo può essere difficile, noiosa e fornire informazioni poco utili.
- La notazione asintotica offre una descrizione concisa e chiara della complessità computazionale, facilitando la comprensione e il confronto tra algoritmi.

4. Impossibilità di conoscere i costi reali:

- In molti casi, è impossibile conoscere con precisione i costi reali di un'operazione, come ad esempio il costo di un'istruzione di alto livello.
- La notazione asintotica fornisce una stima approssimativa del comportamento dell'algoritmo, pur non richiedendo informazioni precise sui costi reali.

5. Efficacia nella pratica:

- È stato dimostrato che la notazione asintotica descrive con accuratezza il comportamento degli algoritmi in pratica.
- Offre quindi un modo affidabile per valutare le prestazioni degli algoritmi e scegliere quello più efficiente per un determinato problema.

Ricerca di un elemento in un array/lista non ordinata

Algoritmo Ricerca Sequenziale

L'algoritmo di ricerca sequenziale, noto anche come ricerca lineare, è un metodo semplice per cercare un elemento in un array o una lista non ordinata. Ecco lo pseudocodice:

```
algoritmo RicercaSequenziale(array  $L$ , elem  $x$ ) → intero
1.  $n =$  lunghezza di  $L$ 
2.  $i=1$ 
3. for  $i=1$  to  $n$  do
4.   if ( $L[i]=x$ ) then return  $i$  \\\trovato
5. return -1 \\\non trovato
```

Analisi di complessità:

- **Tempo di esecuzione ($T(n)$):** Il numero di elementi acceduti (linea 4) su un array di dimensione n .
 - **Caso peggiore (Tworst(n)):** n (si verifica quando l'elemento da cercare non è presente nell'array).
 - **Caso medio (Tavg(n)):** $(n+1)/2$.
- **Operazioni RAM:** Il numero di operazioni RAM su un array di dimensione n .
 - **Caso peggiore (Tworst(n)):** $\Theta(n)$.
 - **Caso medio (Tavg(n)):** $\Theta(n)$.

Una variante: ricerca di un elemento in un array ordinato

L'algoritmo di ricerca binaria è un metodo molto più efficiente per cercare un elemento in un array ordinato. Ecco lo pseudocodice:

```
algoritmo RicercaBinariaRic(array L, elem x, int i, int j) --> intero
1. if (i>j) then return -1
2. m=└ (i+j)/2 ┘
3. if (L[m]=x) then return m
4. if (L[m]>x) then return RicercaBinariaRic(L, x, i, m-1)
5. else return RicercaBinariaRic(L, x, m+1,j)
```

Analisi di complessità:

- **Tempo di esecuzione ($T(n)$):** La relazione di ricorsione $T(n) = T(n/2) + O(1)$ porta a $T(n) = O(\log n)$.
- **Operazioni RAM:** $\Theta(\log n)$, sia nel caso peggiore che nel caso medio.

Equazioni di ricorrenza per la complessità computazionale

Introduzione

La complessità computazionale di un algoritmo ricorsivo può essere espressa in modo naturale attraverso una equazione di ricorrenza. Un'equazione di ricorrenza definisce il costo computazionale di una funzione in termini di se stessa e di sottoproblemi più piccoli.

Metodo dell'iterazione

Un metodo per risolvere le equazioni di ricorrenza è quello di utilizzare l'**iterazione**. L'idea è di "srotolare" la ricorsione, ottenendo una sommatoria dipendente solo dalla dimensione n del problema iniziale.

Esempi

Esempio 1: $T(n) = c + T(n/2)$

Consideriamo l'equazione di ricorrenza:

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

dove c è una costante. Applicando il metodo dell'iterazione, otteniamo:

$$\begin{aligned} T(n) &= c + T(n/2) = 2c + T(n/4) = 2c + c + T(n/8) = 3c + \\ T(n/8) &= \dots = ic + T(n/2^i) \end{aligned}$$

Per $i = \log_2(n)$, si ottiene:

$$T(n) = c \log_2(n) + T(1) = \Theta(\log n)$$

Esempio 2: $T(n) = T(n-1) + 1$

Consideriamo l'equazione di ricorrenza:

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

Applicando il metodo dell'iterazione, si ottiene:

$$\begin{aligned} T(n) &= T(n-1) + 1 = T(n-2) + 1 + 1 = T(n-2) + 2 = T(n-3) + 1 + \\ &2 = T(n-3) + 3 = \dots = T(n-i) + i \end{aligned}$$

Per $i = n-1$, si ottiene: $T(n) = T(1) + n-1 = \Theta(n)$

Esempio 3: $T(n) = 2T(n-1) + 1$

Consideriamo l'equazione di ricorrenza:

$$T(n) = 2T(n-1) + 1$$

Applicando il metodo dell'iterazione, si ottiene:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 = 2(2T(n-2) + 1) + 1 = 4T(n-2) + 2 + 1 = \\ &4(2T(n-3) + 1) + 2 + 1 = 8T(n-3) + 4 + 2 + 1 = \dots = 2^i T(n-i) + \sum_{j=0}^i 2^j \end{aligned}$$

Per $i = n-1$, si ottiene:

$$T(n) = 2^{n-1}T(1) + \sum_{j=0}^{n-2} 2^j = \Theta(2^n)$$

Esempio 4: $T(n) = T(n-1) + T(n-2) + 1$

Consideriamo l'equazione di ricorrenza:

$$T(n) = T(n-1) + T(n-2) + 1$$

Applicando il metodo dell'iterazione, si ottiene:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 = T(n-2) + T(n-3) + 1 + T(n-3) + T(n-4) + 1 + 1 = \\ &= T(n-2) + 2T(n-3) + T(n-4) + 3 = \\ &= T(n-3) + T(n-4) + 1 + 2(T(n-4) + T(n-5) + 1) + T(n-5) + T(n-6) + 1 + 3 = \dots \end{aligned}$$

Questo processo può essere continuato per ottenere una soluzione in termini di n e di costanti.

Analisi dell'albero della ricorsione

Introduzione

L'analisi dell'albero della ricorsione è un metodo per stimare la complessità computazionale di un algoritmo ricorsivo. L'idea è di rappresentare le chiamate ricorsive come un albero e di analizzare il costo computazionale di ogni nodo per ottenere una stima del costo complessivo dell'algoritmo.

Metodi

Esistono due metodi principali per analizzare l'albero della ricorsione:

1. Analisi per nodi:

- **Stimare il tempo speso da ogni nodo:** Se il tempo speso da ogni nodo è costante, la complessità computazionale totale ($T(n)$) è proporzionale al numero di nodi nell'albero.
- **Sommare il tempo speso da ogni nodo:** Per ottenere una stima più precisa, si può stimare il tempo speso da ogni tipo di nodo nell'albero e sommarlo per ottenere il costo complessivo.

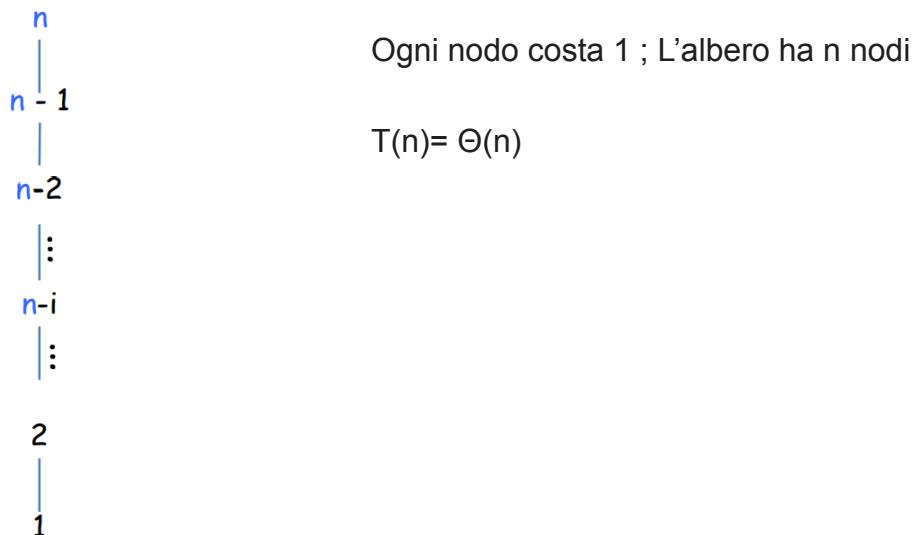
2. Analisi per livelli:

- **Analizzare il tempo speso su ogni livello:** Si stima il costo computazionale massimo per ogni livello dell'albero.
- **Stimare il numero di livelli:** Si determina il numero massimo di livelli nell'albero.
- **Moltiplicare il costo per il numero di livelli:** Il costo complessivo è approssimativamente il prodotto del costo massimo per livello e del numero di livelli.

Esempio: $T(n) = T(n-1) + 1$, $T(1) = 1$

Consideriamo l'equazione di ricorrenza:

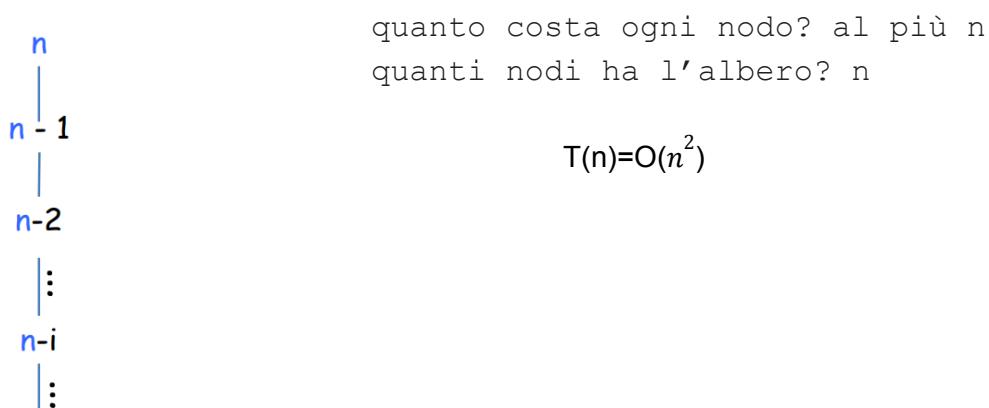
$$T(n) = T(n-1) + 1$$

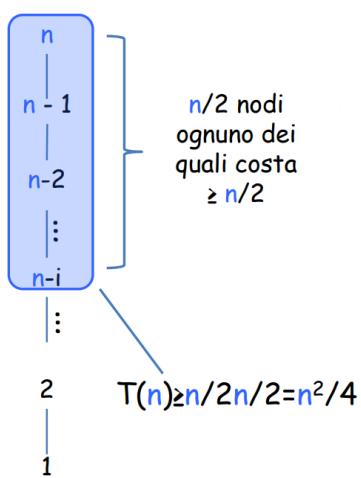


Esempio: $T(n) = T(n-1) + n$, $T(1) = 1$

Consideriamo l'equazione di ricorrenza:

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





$$\rightarrow T(n)=\Omega(n^2)$$

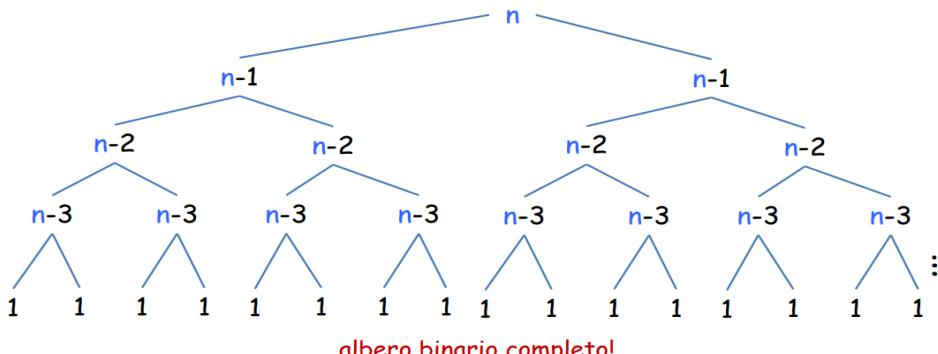
Da questa analisi, sapendo che, Un Θ si verifica nel momento in cui è sia O che Ω possiamo dire:

$$T(n)=\Theta(n^2)$$

Esempio: $T(n) = 2T(n-1) + 1$, $T(1) = 1$

Consideriamo l'equazione di ricorrenza:

$$T(n) = 2T(n-1) + 1$$



albero binario completo!

quanto costa ogni nodo?

...uno!

quanto è alto l'albero?

... $n-1$!

quanti nodi ha un albero
binario completo di altezza h ?

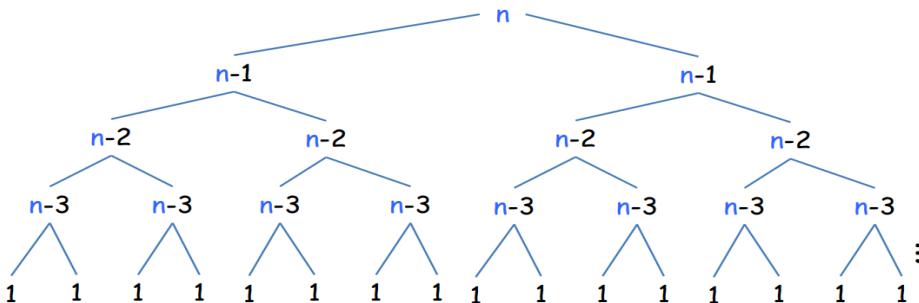
$$\sum_{i=0}^h 2^i = 2^{h+1} - 1$$

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

Esempio: $T(n) = 2T(n-1) + n$, $T(1) = 1$

Consideriamo l'equazione di ricorrenza:

$$T(n) = 2T(n-1) + n$$



albero binario completo!

quanto costa ogni nodo?

...al più n

quanto è alto l'albero?

... $n-1$

quanti nodi ha un albero
binario completo di altezza h ?

$$\sum_{i=0}^h 2^i = 2^{h+1} - 1$$

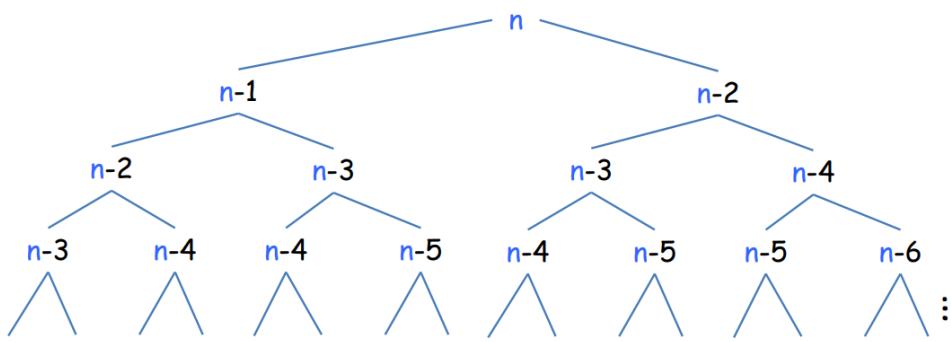
$$T(n) \leq n2^n = \Theta(n2^n)$$

$$T(n) = O(n2^n)$$

Esempio: $T(n) = T(n-1) + T(n-2) + 1$, $T(1) = 1$ [Fibonacci 2]

Consideriamo l'equazione di ricorrenza:

$$T(n) = T(n-1) + T(n-2) + 1$$



albero chiamate ricorsive dell'algoritmo Fibonacci!

quanto costa ogni nodo? ...uno

quanti nodi ha l'albero? $\Theta(\phi^n)$



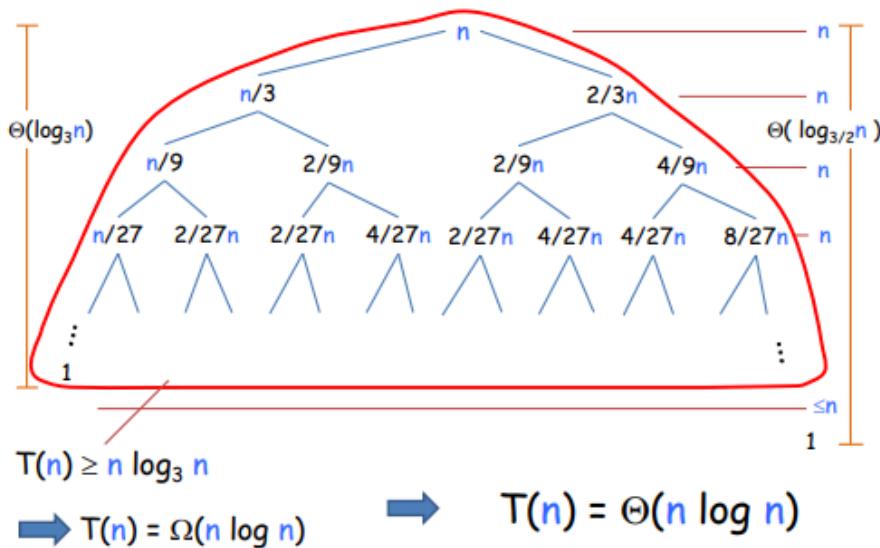
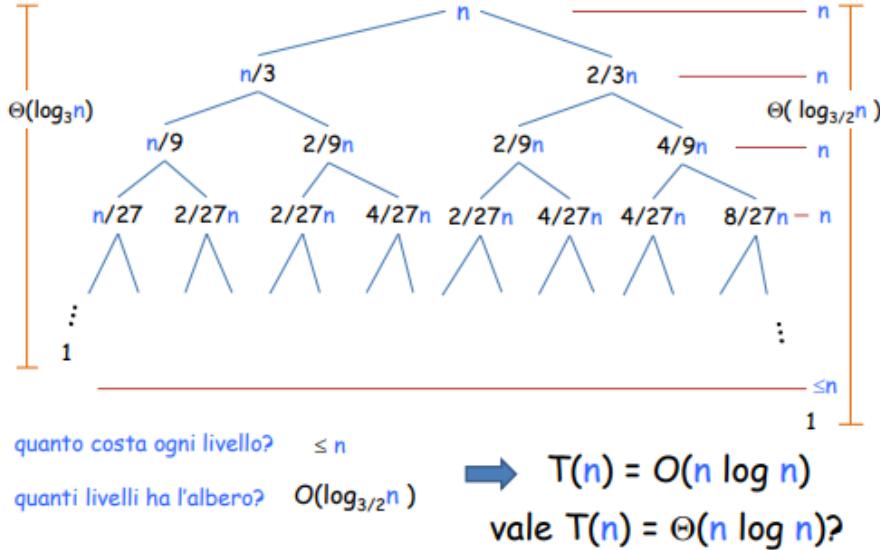
$$T(n) = \Theta(\phi^n)$$

$[T(n) = o(2^n)]$

Esempio: $T(n) = T(n/3) + T(2n/3) + n$

Consideriamo l'equazione di ricorrenza:

$$T(n) = T(n/3) + T(2n/3) + n$$



Metodo della sostituzione per l'analisi della complessità computazionale

Introduzione

Il metodo della sostituzione è una tecnica per dimostrare la correttezza di una stima della complessità computazionale di un algoritmo. Il metodo si basa su tre passaggi:

1. **Individuare una (forma della) soluzione:** Si propone una possibile forma per la soluzione, in genere una funzione con parametri da definire.
2. **Dimostrare la soluzione con induzione matematica:** Si utilizza l'induzione matematica per dimostrare che la soluzione proposta è corretta.
3. **Risolvere per le costanti:** Si determinano i valori delle costanti presenti nella soluzione proposta.

Esempio: $T(n) = n + T(n/2)$, $T(1) = 1$

Consideriamo l'algoritmo con equazione di ricorrenza:

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

e caso base:

$$T(1) = 1$$

Vogliamo dimostrare che la seguente stima è corretta:

$$T(n) \leq 2n$$

Dimostrazione con induzione matematica

Passo base:

Per $n = 1$, si ha:

$$T(1) = 1 \leq 2 * 1$$

Quindi, la stima è valida per il caso base.

Passo induttivo:

Ipotizziamo che la stima sia valida per un valore generico $k < n$, ovvero:

$$T(k) \leq 2k$$

Dimostriamo che la stima è valida anche per n .

Dall'equazione di ricorrenza, si ha:

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

Sostituendo l'ipotesi induttiva per $T(n/2)$, si ottiene:

$$T(n) \leq n + 2(n/2) = (c/2 + 1)n$$

Per dimostrare che la stima è valida per n , dobbiamo dimostrare che:

$$(c/2 + 1) \leq 2$$

Sviluppando la diseguaglianza, si ottiene:

$$c/2 \leq 1$$

Da cui segue:

$$c \geq 2$$

Quindi, la stima è valida per ogni $n > 1$ se $c \geq 2$.

Conclusione

Poiché la stima è valida per il caso base e per il passo induttivo, e le costanti soddisfano la condizione $c \geq 2$, possiamo concludere che la stima:

$$T(n) \leq 2n$$

è corretta. Di conseguenza, la complessità computazionale dell'algoritmo è:

$$T(n) = O(n)$$

Algoritmi basati sul divide et impera

Gli algoritmi basati sulla tecnica del divide et impera si basano su tre passaggi fondamentali:

1. **Dividi:** Il problema di dimensione n viene diviso in a sottoproblemi di dimensione n/b .
2. **Conquista:** I sottoproblemi vengono risolti ricorsivamente.
3. **Combina:** Le soluzioni dei sottoproblemi vengono combinate per ottenere la soluzione del problema originale.

Relazione di ricorrenza

Sia $f(n)$ il tempo impiegato per dividere e ricombinare istanze di dimensione n . La relazione di ricorrenza che descrive la complessità computazionale di un algoritmo basato sul divide et impera è:

$$T(n) = \begin{cases} a T(n/b) + f(n) & \text{se } n > 1 \\ \Theta(1) & \text{se } n = 1 \end{cases}$$

Esempio:

Algoritmo Fibonacci

```

algoritmo fibonacci6(intero n) → intero
1.    $A \leftarrow \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ 
2.    $M \leftarrow \text{potenzaDiMatrice}(A, n - 1)$ 
3.   return M[0][0]

funzione potenzaDiMatrice(matrice A, intero k) → matrice
4.   if (k ≤ 1) then M ←  $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ 
5.   else M ← potenzaDiMatrice(A, [k/2])
6.       M ← M · M
7.   if (k è dispari) then M ← M · A
8.   return M

```

$$a=1, b=2, f(n)=O(1)$$

Teorema Master

Il teorema Master fornisce un'analisi asintotica della complessità computazionale degli algoritmi basati sul divide et impera. Il teorema si basa sul confronto tra due funzioni:

$n^{\log_b a}$ vs $f(n)$

- $n^{\log_b a}$: rappresenta la crescita esponenziale del numero di sottoproblemi.
- $f(n)$: rappresenta il tempo impiegato per dividere e ricombinare le istanze.

Enunciato informale:

- Se $n^{\log_b a}$ e $f(n)$ hanno lo stesso ordine asintotico, la complessità computazionale dell'algoritmo è $\Theta(f(n) \log n)$.

- Se una delle due funzioni è "polynomialmente" più veloce dell'altra, la complessità computazionale dell'algoritmo è dominata dalla funzione più veloce.

Soluzioni della relazione di ricorrenza

La relazione di ricorrenza $T(n) = aT(n/b) + f(n)$ ha tre possibili soluzioni:

$$1. T(n) = \Theta(n^{\log_b(a)})$$

Se $f(n) = O(n^{\log_b(a) - \varepsilon})$ per un ε positivo, la complessità computazionale è dominata dal termine esponenziale del numero di sottoproblemi.

$$2. T(n) = \Theta((n^{\log_b(a)} \log n))$$

Se $f(n) = \Theta(n^{\log_b(a)})$, la complessità computazionale include un logaritmo in base al numero di sottoproblemi.

$$3. T(n) = \Theta(f(n))$$

Se $f(n) = \Omega(n^{\log_b(a) - \varepsilon})$ per un ε positivo e $f(n/b) \leq c f(n)$ per una costante c minore di 1 e n sufficientemente grande, la complessità computazionale è dominata dalla funzione $f(n)$.

Esempi

$$1) T(n) = n + 2T(n/2)$$

$$a=2, b=2, f(n)=n=\Theta(n^{\log_2 2}) \rightarrow T(n)=\Theta(n \log n)$$

(caso 2 del teorema master)

$$2) T(n) = c + 3T(n/9)$$

$$a=3, b=9, f(n)=c=O(n^{\log_9 3 - \varepsilon}) \rightarrow T(n)=\Theta(\sqrt[n]{n})$$

(caso 1 del teorema master, es: $\varepsilon=0.1$)

$$3) T(n) = n + 3T(n/9)$$

$$a=3, b=9, f(n)=n=\Omega(n^{\log_9 3 + \varepsilon})$$

$3(n/9) \leq c n$ per $c=1/3$

(caso 3 del teorema master, es: $\varepsilon=0.1$)

Conclusione

Il teorema Master fornisce un metodo per analizzare la complessità computazionale degli algoritmi basati sul divide et impera. Il teorema si basa sul confronto tra il tasso di crescita del numero di sottoproblemi e il tempo impiegato per dividere e ricombinare le istanze. In base al risultato del confronto, la complessità computazionale dell'algoritmo può essere espressa in termini di una delle tre funzioni presentate.

Gestione di collezioni di oggetti

Tipo di dato: – Specifica una collezione di oggetti e delle operazioni di interesse su tale collezione (es. inserisci, cancella, cerca)

Struttura dati: – Organizzazione dei dati che permette di memorizzare la collezione e supportare le operazioni di un tipo di dato usando meno risorse di calcolo possibile.

Tipo dato DIZIONARIO:

tipo Dizionario:

dati:

un insieme S di coppie $(elem, chiave)$.

operazioni:

`insert(elem e, chiave k)`

aggiunge a S una nuova coppia (e, k) .

`delete(chiave k)`

cancella da S la coppia con chiave k .

`search(chiave k) → elem`

se la chiave k è presente in S restituisce l'elemento e ad essa associato, e null altrimenti.

Tipo dato PILA:

tipo Pila:

dati:

una sequenza S di n elementi.

operazioni:

`isEmpty() → result`

restituisce `true` se S è vuota, e `false` altrimenti.

`push(elem e)`

aggiunge e come ultimo elemento di S .

`pop() → elem`

toglie da S l'ultimo elemento e lo restituisce.

`top() → elem`

restituisce l'ultimo elemento di S (senza toglierlo da S).

Tipo dato CODA:

tipo Coda:

dati:

una sequenza S di n elementi.

operazioni:

`isEmpty() → result`

restituisce `true` se S è vuota, e `false` altrimenti.

`enqueue(elem e)`

aggiunge e come ultimo elemento di S .

`dequeue() → elem`

toglie da S il primo elemento e lo restituisce.

`first() → elem`

restituisce il primo elemento di S (senza toglierlo da S).

Rappresentazioni indicizzate e collegate

Rappresentazioni indicizzate

Le rappresentazioni indicizzate, tipicamente rappresentate dagli array, si basano su una collezione di celle numerate che contengono elementi di un tipo prestabilito.

Proprietà:

- **Forte:**

- Gli indici delle celle sono numeri consecutivi.
- L'accesso a un elemento avviene tramite il suo indice, garantendo un accesso diretto e veloce.

- **Debole:**

- Non è possibile aggiungere nuove celle ad un array una volta creato.
- La dimensione dell'array è fissa e deve essere specificata in anticipo.

Rappresentazioni collegate

Le rappresentazioni collegate si basano su una collezione di record, ovvero strutture dati che contengono i dati e un riferimento (puntatore) ad altri record.

Proprietà:

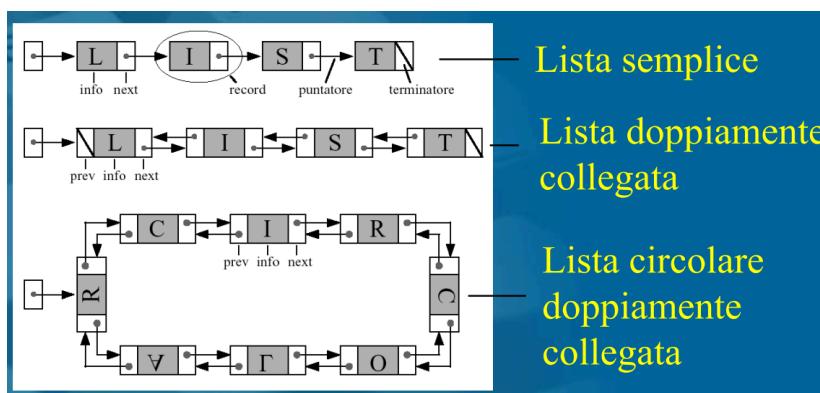
- **Forte:**
 - È possibile aggiungere o togliere record dinamicamente alla struttura.
 - La struttura può adattarsi a variazioni del numero di elementi.
- **Debole:**
 - Gli indirizzi dei record non sono necessariamente consecutivi.
 - L'accesso a un elemento richiede la traversata della struttura seguendo i puntatori, il che può essere meno efficiente rispetto all'accesso diretto negli array.

Pro e contro:

Rappresentazioni indicizzate: – Pro: accesso diretto ai dati mediante indici – Contro: dimensione fissa (riallocazione array richiede tempo lineare)

Rappresentazioni collegate: – Pro: dimensione variabile (aggiunta e rimozione record in tempo costante) – Contro: accesso sequenziale ai dati

Esempi di strutture collegate



Realizzazione di un dizionario

Metodi di implementazione

Esistono diversi modi per implementare un dizionario, ognuno con i suoi pro e contro. Di seguito, analizziamo alcuni metodi comuni:

1. Array non ordinato (sovradimensionato):

- **Insert: costa O(1)** - L'inserimento di un nuovo elemento è molto efficiente, in quanto basta inserirlo dopo l'ultimo elemento dell'array.
- **Search: costa O(n)** - Per trovare un elemento, è necessario scorrere l'intero array nel caso peggiore, con un costo proporzionale al numero di elementi.
- **Delete: costa O(n)** - L'eliminazione di un elemento richiede la ricerca dell'elemento e il successivo spostamento di tutti gli elementi successivi, per un costo proporzionale al numero di elementi.

2. Array ordinato:

- **Search: costa O(log(n))** - La ricerca binaria può essere utilizzata per trovare un elemento in un array ordinato, con un costo logaritmico rispetto al numero di elementi.
- **Insert: costa O(n)** - L'inserimento di un nuovo elemento richiede il mantenimento dell'ordinamento dell'array, il che comporta:
 - **O(log(n)) confronti per trovare la giusta posizione da inserire.**
 - **O(n) spostamenti di elementi per mantenere l'ordinamento.**
- **Delete: costa O(n)** - L'eliminazione di un elemento richiede il mantenimento dell'ordinamento dell'array, con un costo simile all'inserimento.

3. Lista non ordinata:

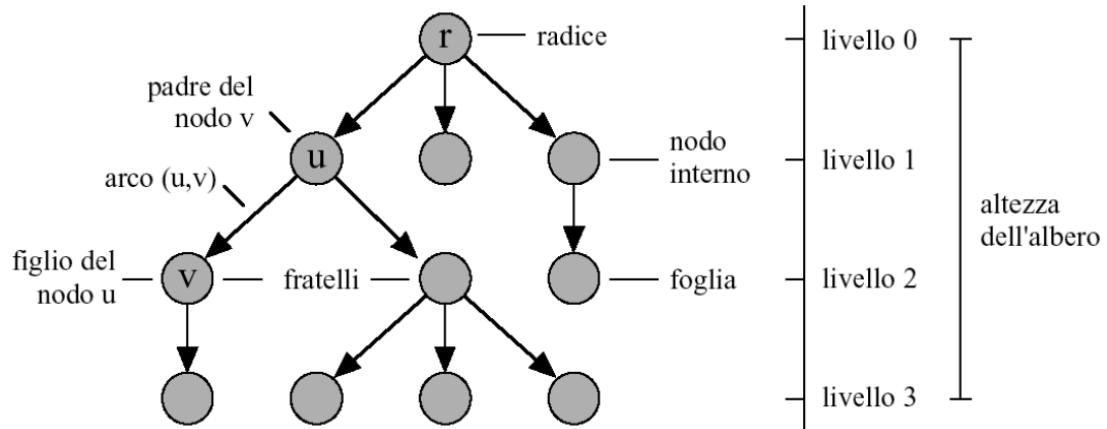
- **Insert: costa O(1)** - L'inserimento di un nuovo elemento è molto efficiente, in quanto basta inserirlo in fondo alla lista.
- **Search: costa O(n)** - Per trovare un elemento, è necessario scorrere l'intera lista nel caso peggiore, con un costo proporzionale al numero di elementi.
- **Delete: costa O(n)** - L'eliminazione di un elemento richiede la ricerca dell'elemento e il successivo spostamento di tutti gli elementi successivi, per un costo proporzionale al numero di elementi.

4. Lista ordinata:

- **Insert: costa O(n)** - L'inserimento di un nuovo elemento richiede il mantenimento dell'ordinamento della lista, con un costo proporzionale al numero di elementi.

- **Search: costa $O(n)$** - Non è possibile utilizzare la ricerca binaria per la ricerca in una lista ordinata, in quanto gli elementi non sono indicizzati.
- **Delete: costa $O(n)$** - L'eliminazione di un elemento richiede il mantenimento dell'ordinamento della lista, con un costo proporzionale al numero di elementi.

Alberi:



Dati contenuti nei **nodi**, relazioni gerarchiche definite dagli **archi** che li collegano.

grado di un nodo: numero dei suoi figli

albero d-ario, albero d-ario completo

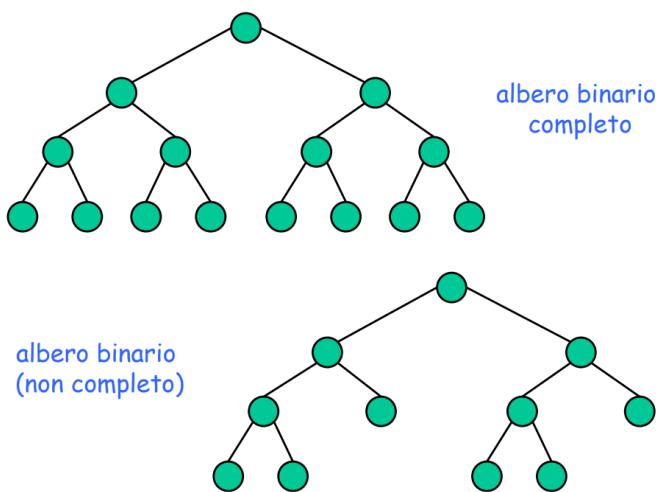
albero d-ario: albero in cui tutti i nodi interni hanno (al più) d figli

d=2 → albero binario

un albero d-ario è completo: se tutti nodi interni hanno esattamente d figli e le foglie sono tutte allo stesso livello

u **antenato** di v se u è raggiungibile da v risalendo di padre in padre

v **discendente** di u se u è un antenato di v.

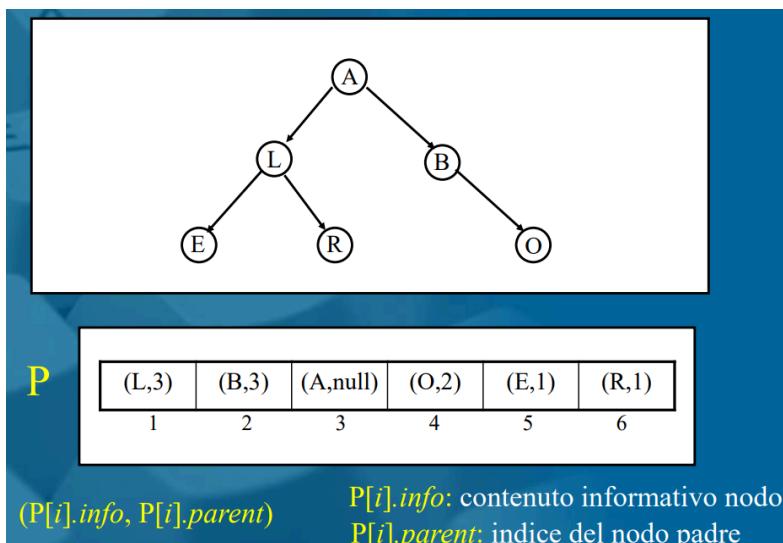


Rappresentazioni indicizzate di alberi

Vettore dei padri

Per rappresentare un albero con n nodi usando un **vettore dei padri**, si utilizza un array P di dimensione n (o almeno n). Ogni cella i del vettore contiene una coppia:

- $info$: il contenuto informativo del nodo i .
- $parent$: l'indice (nell'array) del nodo padre di i .



Osservazioni:

- # arbitrario di figli
- tempo per individuare il padre di un nodo: $O(1)$
- tempo per individuare uno o più figli di un nodo: $O(n)$

Vettore posizionale per alberi d-ari (quasi) completi

Struttura

Un vettore posizionale rappresenta un albero d-ario (quasi) completo organizzando i nodi "per livelli" all'interno di un array.

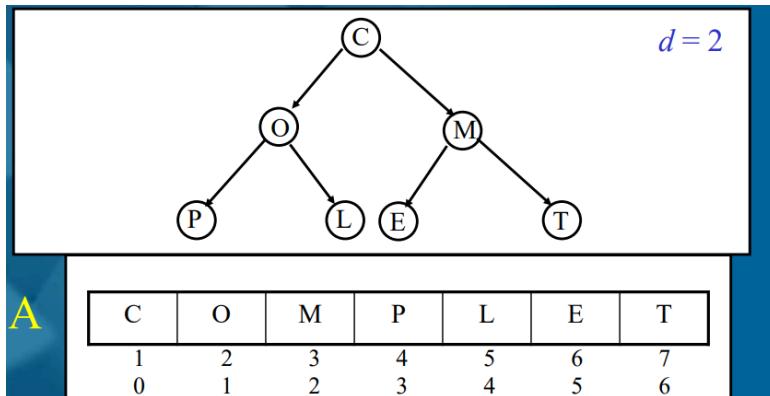
Due modi per indicizzare i nodi:

1. Indici a partire da 0:

- Il j -esimo figlio (con $j \in \{1, \dots, d\}$) di un nodo i si trova in posizione $d * i + j$.
- Il padre di un nodo i si trova in posizione $\lfloor (i - 1) / d \rfloor$.

2. Indici a partire da 1:

- Il j -esimo figlio (con $j \in \{1, \dots, d\}$) di un nodo i si trova in posizione $d * (i - 1) + j + 1$.
- Il padre di un nodo i si trova in posizione $\lfloor (i - 2) / d \rfloor + 1$.



Osservazioni:

- # di figli esattamente d
- solo per alberi completi o quasi completi
- tempo per individuare il padre di un nodo: $O(1)$
- tempo per individuare uno specifico figlio di un nodo: $O(1)$

Algoritmi di visita per alberi

Introduzione

Gli algoritmi di visita permettono di accedere in modo sistematico ai nodi e agli archi di un albero. Si distinguono in base al particolare ordine di accesso ai nodi.

Algoritmo di visita in profondità (DFS)

L'algoritmo di visita in profondità (DFS) parte da un nodo radice r e procede visitando i nodi di figlio in figlio fino a raggiungere una foglia. In questo punto, l'algoritmo "retrocede" al primo antenato che ha ancora figli non visitati (se esiste) e ripete il procedimento a partire da uno di quei figli.

Proprietà:

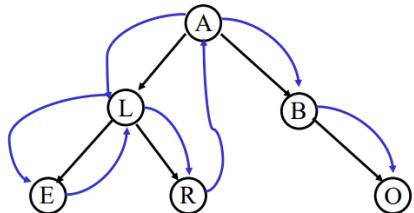
- Ogni nodo viene inserito e estratto dalla pila una sola volta.
- Il tempo speso per ogni nodo è $O(1)$ (se è possibile individuare i figli di un nodo in tempo costante).
- Il numero di nodi nulli inseriti/estratti è $O(n)$.
- Complessità temporale: $O(n)$.

Pseudocodice Per alberi binari:

```
algoritmo visitaDFS(nodo r)
    Pila S
    S.push(r)
    mentre (not S.isEmpty()) do
        u  $\leftarrow$  S.pop()
        se (u  $\neq$  null) allora
            visita il nodo u
            S.push(figlio destro di u)
            S.push(figlio sinistro di u)
```

Esempio:

Consideriamo il seguente albero:



L'esecuzione del DFS con nodo radice A produce la seguente sequenza di visite:

A, L, E, R, B, O

versione ricorsiva (Per alberi binari):

```
algoritmo visitaDFSRicorsiva(nodo r)
1.   se (r  $\neq$  null) allora
2.       visita il nodo r
3.       visitaDFSRicorsiva(figlio sinistro di r)
4.       visitaDFSRicorsiva(figlio destro di r)
```

Algoritmo di visita in ampiezza (BFS)

L'algoritmo di visita in ampiezza (BFS) parte da un nodo radice r e procede visitando i nodi per livelli successivi. Un nodo sul livello i può essere visitato solo se tutti i nodi sul livello $i-1$ sono stati visitati.

Proprietà:

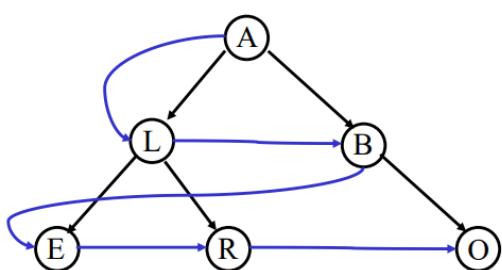
- Ogni nodo viene inserito e estratto dalla coda una sola volta.
- Il tempo speso per ogni nodo è $O(1)$ (se è possibile individuare i figli di un nodo in tempo costante).
- Il numero di nodi nulli inseriti/estratti è $O(n)$.
- Complessità temporale: $O(n)$.

Versione iterativa (per alberi binari):

```
algoritmo visitaBFS(nodo r)
    Coda C
    C.enqueue(r)
    while (not C.isEmpty()) do
        u  $\leftarrow$  C.dequeue()
        if (u  $\neq$  null) then
            visita il nodo u
            C.enqueue(figlio sinistro di u)
            C.enqueue(figlio destro di u)
```

Esempio:

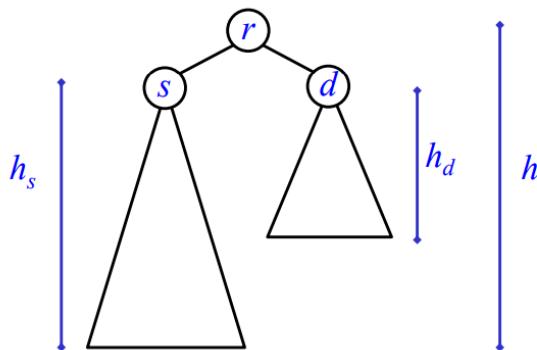
Consideriamo il seguente albero:



L'esecuzione del BFS con nodo radice A produce la seguente sequenza di visite:

A, L, B, E, R, O

Esempi Pratici della Visita : Calcolo dell'altezza



CalcolaAltezza (nodo r)

1. **if** ($r = \text{null}$) **then return** -1
2. $sin = \text{CalcolaAltezza}(\text{figlio sinistro di } r)$
3. $des = \text{CalcolaAltezza}(\text{figlio destro di } r)$
4. **return** $1 + \max\{sin, des\}$

COMPLESSITÀ TEMPORALE = $O(n)$

$$h = 1 + \max\{h_s, h_d\}$$

PROBLEMA 3.6

Si scrivano varianti dell'algoritmo per:

1. calcolare il numero di foglie di un albero;
2. calcolare il grado medio dei nodi dell'albero (numero medio di figli di un nodo non foglia);
3. verificare se esiste un nodo dell'albero che abbia un dato contenuto informativo.

CalcolaNumFoglie (nodo r)

1. **if** ($r = \text{null}$) **then return** 0
2. **if** (r è una foglia) **then return** 1
3. $sin = \text{CalcolaNumFoglie}(\text{figlio sinistro di } r)$
4. $des = \text{CalcolaNumFoglie}(\text{figlio destro di } r)$
5. **return** ($sin + des$)

CalcolaGradoMedio (nodo r)

1. $n = \text{numero nodi dell'albero}$
2. $nfoglie = \text{CalcolaNumFoglie} (\text{nodo } r)$
3. **if** ($r \neq \text{null}$) **return** ($\text{SommaGradi}(r)/(n-nfoglie)$)

SommaGradi(nodo r)

1. **if** ($r = \text{null}$) **return** 0
2. **if** (r è una foglia) **return** 0
3. $S = \text{numero figli di } r + \text{SommaGradi}(\text{figlio sinistro di } r) + \text{SommaGradi}(\text{figlio destro di } r)$
4. **return** S

CercaElemento (nodo r , chiave k)

1. **if** ($r = \text{null}$) **then return** null
2. **if** ($\text{chiave}(r) = k$) **then return** r
3. $sin = \text{CercaElemento}(\text{figlio sinistro di } r, k)$
4. **if** ($sin \neq \text{null}$) **then return** sin
5. **return** $\text{CercaElemento}(\text{figlio destro di } r, k)$

Ordinamento

Introduzione

Il problema dell'ordinamento consiste nel dato un insieme S di n oggetti presi da un dominio totalmente ordinato, ordinare S . In altre parole, si vuole trovare una permutazione (riarrangiamento) della sequenza di input tale che ogni elemento sia nella sua posizione corretta rispetto agli altri.

Algoritmi di ordinamento

Esistono diversi algoritmi di ordinamento, ognuno con le sue caratteristiche e complessità computazionale. Tra i più comuni:

1. Selection Sort:

- **Descrizione:** L'algoritmo Selection Sort utilizza un approccio incrementale per ordinare una sequenza di elementi. Ad ogni passo, individua il minimo degli elementi non ancora ordinati e lo colloca nella sua posizione corretta.

- **Pseudocodice:**

SelectionSort (A)

1. **for** k=0 **to** n-2 **do**
2. m = k+1
3. **for** j=k+2 **to** n **do**
4. **if** (A[j] < A[m]) **then** m=j
5. scambia A[m] con A[k+1]

- **Correttezza:**

- L'algoritmo mantiene due invarianti:

- Dopo ogni passo k (con $k = 0, \dots, n-2$), i primi $k+1$ elementi sono ordinati.
 - I primi $k+1$ elementi sono i $k+1$ elementi più piccoli dell'array.

- **Complessità temporale:**

- Analisi del tempo medio:

- Ogni linea di codice costa tempo $O(1)$.
 - Il numero massimo di operazioni è $n(n-1)/2$, che equivale a $\Theta(n^2)$.
 - Pertanto, la complessità temporale media è $O(n^2)$.

- Analisi del tempo peggiore:

- Il numero minimo di operazioni è $n(n-1)/2$, che equivale a $\Theta(n^2)$.
 - Pertanto, la complessità temporale peggiore è $\Omega(n^2)$.

- **Conclusione:**

- La complessità temporale dell'algoritmo Selection Sort è $\Theta(n^2)$.

Insertion Sort

Descrizione

L'algoritmo Insertion Sort utilizza un approccio incrementale per ordinare una sequenza di elementi. Ad ogni passo, considera un nuovo elemento e lo inserisce nella sua posizione corretta all'interno della sequenza già ordinata.

Implementazione (pseudocodice):

```
algoritmo insertionSort(array A)
1.   for k = 1 to n - 1 do
2.       x ← A[k + 1]
3.       for j = k downto 0 do
4.           if (j > 0 ∧ A[j] ≤ x) then break
5.       if (j < k) then
6.           for t = k downto j + 1 do A[t + 1] ← A[t]
7.           A[j + 1] ← x
```

Correttezza

L'algoritmo mantiene l'invariante che dopo ogni passo k (con $k = 1, \dots, n-1$), i primi k elementi sono ordinati.

Complessità temporale

- La complessità temporale dell'algoritmo Insertion Sort è $\Theta(n^2)$.

Bubble Sort

Descrizione

L'algoritmo Bubble Sort utilizza un approccio incrementale per ordinare una sequenza di elementi. Ad ogni scansione, confronta coppie di elementi adiacenti e li scambia se non sono nell'ordine corretto. Il processo si ripete fino a che non ci sono più scambi.

Implementazione (pseudocodice):

```
algoritmo bubbleSort(array A)
1.   for i = 1 to (n - 1)
2.       scambi ← false
3.       for j = 2 to (n - i + 1)
4.           if (A[j - 1] > A[j]) then scambia A[j - 1] e A[j]; scambi ← true
5.       if (scambi) then break
```

Correttezza

L'algoritmo mantiene l'invariante che dopo ogni scansione, i primi $i+1$ elementi sono ordinati.

Complessità temporale

- La complessità temporale dell'algoritmo Bubble Sort è $\Theta(n^2)$.

Merge Sort

Descrizione

Il Merge Sort è un algoritmo di ordinamento che utilizza la tecnica del "divide et impera" per ordinare una sequenza di elementi. L'algoritmo si basa su tre fasi:

1. Divide: L'array viene diviso in due sottoarray di uguale dimensione (o quasi uguale se la dimensione è dispari).
2. Risolvi: I due sottoarray vengono ordinati ricorsivamente utilizzando lo stesso algoritmo Merge Sort.
3. Impera: I due sottoarray ordinati vengono fusi in un unico array ordinato.

Implementazione

L'implementazione del Merge Sort si basa su :

- `merge_sort(array, start, end)`: divide l'array in due sottoarray e li ordina ricorsivamente, quindi li fonde.

Pseudocodice:

MergeSort (A, i, f)

1. **if** ($i < f$) **then**
2. $m = \lfloor (i+f)/2 \rfloor$
3. MergeSort(A,i,m)
4. MergeSort(A,m+1,f)
5. Merge(A,i,m,f)

Correttezza

- La funzione `merge_sort` divide correttamente l'array in due sottoarray e li ordina ricorsivamente.
- La funzione `merge` fonde correttamente due sottoarray ordinati in un unico array ordinato.

Complessità temporale

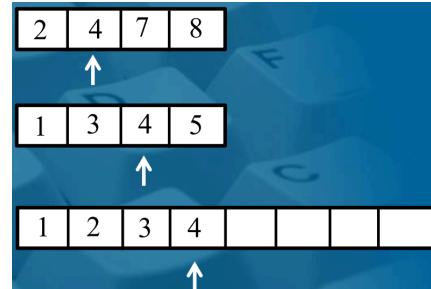
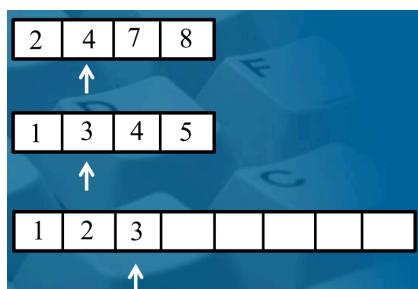
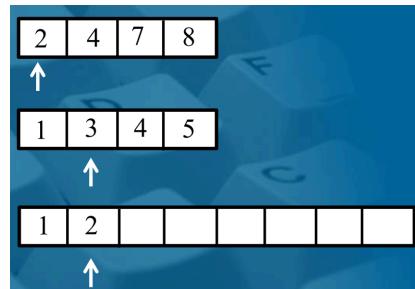
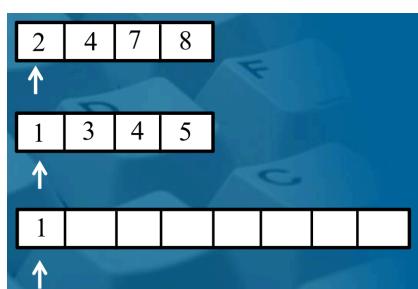
- Il tempo di esecuzione del Merge Sort è descritto dalla seguente relazione di ricorrenza: $T(n) = 2T(n/2) + O(n)$.
- Applicando il Teorema Master, si ottiene che la complessità temporale del Merge Sort è $O(n \log n)$.

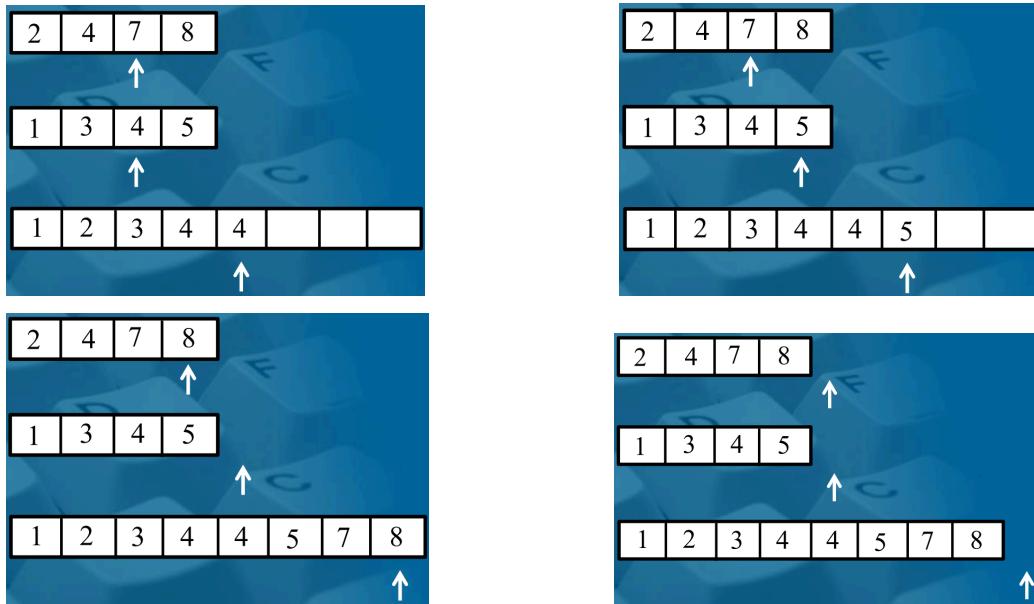
Memoria

- La complessità spaziale del Merge Sort è $\Theta(n)$.
- La procedura `merge` usa memoria ausiliaria pari alla dimensione della porzione da fondere.
- Il Merge Sort non ordina "in loco", ovvero richiede memoria ausiliaria pari a $\Theta(n)$ (oltre all'input) per funzionare.

Considerazioni

- Il Merge Sort è un algoritmo di ordinamento efficiente con una complessità temporale media e peggiore di $O(n \log n)$.
- Non è un algoritmo di ordinamento "in loco", ovvero richiede memoria ausiliaria per funzionare.





QuickSort

Descrizione

Il QuickSort è un algoritmo di ordinamento che utilizza la tecnica del "divide et impera" per ordinare una sequenza di elementi. L'algoritmo si basa su tre fasi:

1. Divide:

- Si sceglie un elemento x della sequenza come "perno".
- Si esegue la partizione della sequenza in due sottoarray:
 - Elementi minori o uguali al perno ($A[i:m-1]$).
 - Elementi maggiori del perno ($A[m+1:f]$).

2. Risolvi:

- I due sottoarray vengono ordinati ricorsivamente utilizzando lo stesso algoritmo QuickSort.

3. Impera:

- Si concatena la sottosequenza ordinata a sinistra ($A[i:m-1]$) con il perno ($A[m]$) e la sottosequenza ordinata a destra ($A[m+1:f]$), ottenendo l'array ordinato completo.

Differenze con il MergeSort

Rispetto al MergeSort, il QuickSort si differenzia per:

- **Divide:** la fase di "divide" è più complessa nel QuickSort, in quanto richiede la partizione dell'array.
- **Impera:** la fase di "impera" è più semplice nel QuickSort, in quanto si limita alla concatenazione delle sottosequenze ordinate.

Partizione

La partizione è un'operazione fondamentale nel QuickSort. Viene eseguita "in loco", ovvero senza richiedere memoria ausiliaria, e si basa su due scansioni simultanee:

- Scansione da sinistra: si cerca il primo elemento maggiore del perno.
- Scansione da destra: si cerca il primo elemento minore del perno.

Gli elementi trovati vengono scambiati, e le scansioni vengono ripetute fino a quando non ci sono più scambi da fare.

Partition (A, i, f)

1. $x = A[i]$
2. $inf = i$
3. $sup = f + 1$
4. **while** (true) **do**
5. **do** ($inf = inf + 1$) **while** ($inf \leq f$ e $A[inf] \leq x$)
6. **do** ($sup = sup - 1$) **while** ($A[sup] > x$)
7. **if** ($inf < sup$) **then** scambia $A[inf]$ e $A[sup]$
8. **else break**
9. scambia $A[i]$ e $A[sup]$
10. **return sup**

Proprietà (invariante)

L'invariante del QuickSort garantisce che:

- In ogni istante, gli elementi $A[i], \dots, A[inf-1]$ sono minori o uguali al perno.
- Gli elementi $A[sup+1], \dots, A[f]$ sono maggiori del perno.

Pseudocodice:

QuickSort (A, i, f)

1. **if** ($i < f$) **then**
2. m=Partition(A,i,f)
3. QuickSort(A,i,m-1)
4. QuickSort(A, m +1,f)

Tempo di esecuzione

Il tempo di esecuzione del QuickSort dipende dalla scelta del perno:

Caso medio:

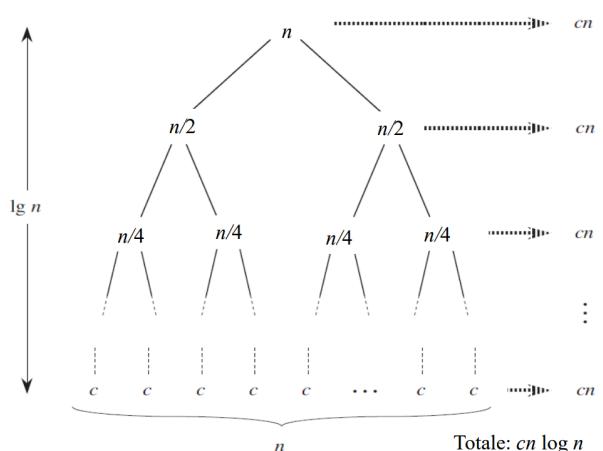
- Se la partizione produce sottoarray di dimensioni simili (circa metà della dimensione originale), il tempo di esecuzione medio è $O(n \log n)$.

Caso peggiore:

- Se la partizione produce sottoarray di dimensioni molto diverse (ad esempio, un elemento e tutto il resto), il tempo di esecuzione peggiore è $O(n^2)$.

Caso migliore

Il caso migliore del QuickSort si verifica quando la partizione produce sempre sottoarray di dimensioni uguali. In questo caso, la complessità temporale è $O(n \log n)$.



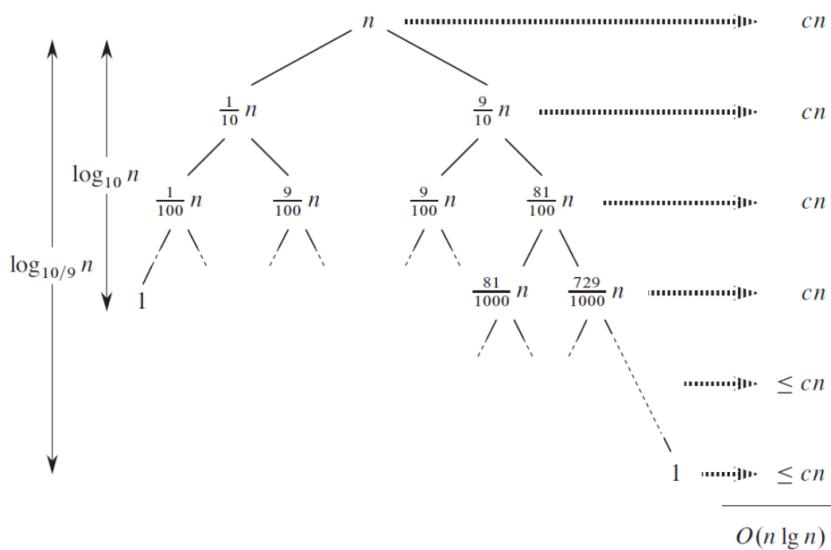
Caso medio: intuizioni

Sebbene il caso medio del QuickSort sia difficile da dimostrare matematicamente, possiamo ragionare per intuizioni:

- La probabilità che la partizione produca sempre il caso peggiore è molto bassa.
- Per partizioni non troppo sbilanciate, l'algoritmo è generalmente veloce.
- Se la partizione producesse sempre sottoarray con un rapporto di 9 a 1, la complessità rimarrebbe comunque $O(n \log n)$.
- Anche con un rapporto di 99 a 1, la complessità sarebbe ancora $O(n \log n)$.

Caso medio: considerazioni sulle istanze

Tuttavia, se le istanze non sono equiprobabili (ad esempio, un array ordinato in ordine inverso), il caso medio potrebbe non essere valido.



Versione randomizzata

Per ovviare al problema della dipendenza dalla scelta del perno, si può utilizzare una versione randomizzata del QuickSort:

- Si sceglie il perno a caso tra gli elementi da ordinare.

Con questa modifica:

- Il tempo di esecuzione medio è sempre $O(n \log n)$.

- Il tempo di esecuzione peggiore è comunque $O(n^2)$, ma si verifica solo in casi estremamente improbabili.

Conclusione

Il QuickSort è un algoritmo di ordinamento efficiente e versatile, con un tempo di esecuzione medio $O(n \log n)$ e una versione randomizzata con buone garanzie di performance. Tuttavia, è importante considerare il caso peggiore e la dipendenza dalla scelta del perno per valutare la sua applicabilità in scenari specifici.

HeapSort

Idea

L'algoritmo HeapSort utilizza un approccio incrementale simile al SelectionSort, ma sfrutta una struttura dati efficiente per selezionare gli elementi dal più grande al più piccolo.

Differenze con il SelectionSort:

- Selezione del massimo: Invece di cercare il minimo ad ogni passo, l'HeapSort trova e rimuove il massimo dalla struttura dati.
- Struttura dati: L'utilizzo di una struttura dati chiamata "heap" permette di estrarre il massimo in tempo logaritmico, migliorando significativamente l'efficienza rispetto al SelectionSort.

Tipo di dato:

- Un tipo di dato definisce un insieme di oggetti e le operazioni che è possibile eseguire su di essi.
- Esempi di tipi di dati: array, liste, alberi, grafi, dizionari.

Struttura dati:

- Una struttura dati è un'organizzazione dei dati che permette di memorizzare la collezione e supportare le operazioni del tipo di dato in modo efficiente.
- L'obiettivo è minimizzare l'utilizzo di risorse di calcolo (memoria, tempo di esecuzione).

Cruciale:

- Progettare una struttura dati chiamata "heap" che permetta di:
 - Generare velocemente un heap a partire da un array.
 - Trovare il massimo elemento (il più grande) in tempo logaritmico.
 - Rimuovere il massimo elemento in tempo logaritmico.

HeapSort: implementazione

L'algoritmo HeapSort si basa sui seguenti passi:

1. Costruzione dell'heap: Dato un array A , si costruisce un heap a partire da esso.
2. Estrazione del massimo: Si estrae il massimo elemento dall'heap e lo si inserisce nella posizione corretta nell'array ordinato.
3. Ripetizione: Si ripetono i passi 2 e 3 finché l'heap non è vuoto.

Tipo di dato associato: coda con priorità

L'heap è una struttura dati associata ad un tipo di dato chiamato "coda con priorità".

Proprietà:

- Gli elementi sono organizzati in base a una "chiave".
- L'elemento con la priorità più alta (il più grande) è sempre accessibile in tempo logaritmico.
- Le operazioni di inserimento e rimozione richiedono tempo logaritmico (in media).

Struttura dati heap

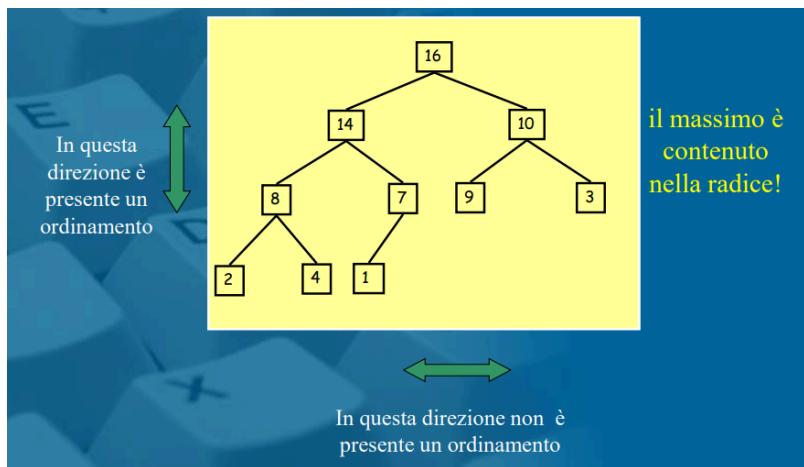
Proprietà:

1. Albero binario radicato: L'heap è rappresentato come un albero binario radicato.
2. Completezza: L'albero è completo fino al penultimo livello.
3. Struttura rafforzata: Le foglie sull'ultimo livello sono compattate a sinistra.
4. Proprietà di heap:
 - Ogni nodo contiene un elemento (chiave).
 - La chiave del padre è sempre maggiore o uguale alla chiave del figlio (tranne per la radice).

Proprietà salienti:

1. **Massimo nella radice:** Il massimo elemento è sempre contenuto nella radice.
2. **Altezza:** L'altezza di un heap con n nodi è $O(\log n)$.
3. **Rappresentazione in array:** Gli heap con struttura rafforzata possono essere rappresentati in un array di dimensione n .

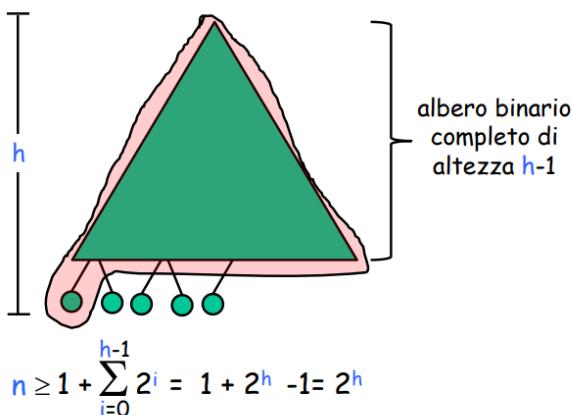
1)



2)

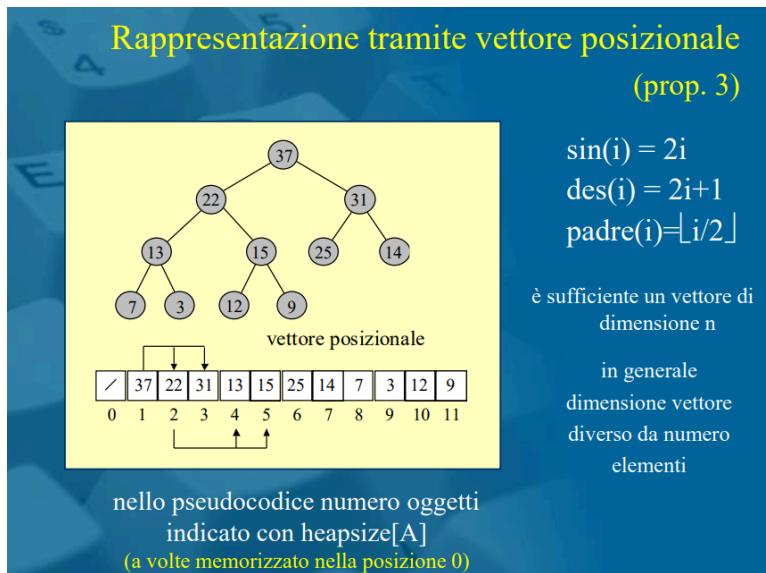
Altezza di un heap (prop. 2)

Sia H un heap di n nodi e altezza h .



$$\Rightarrow h \leq \log_2 n$$

3)



Procedura fixHeap

Descrizione

La procedura **fixHeap** viene utilizzata per ripristinare la proprietà di heap in un albero binario dopo che un valore nella radice è stato modificato o spostato. La procedura funziona come segue:

1. **Controllo dei figli:** Si controlla se la radice v ha figli.
2. **Trova il figlio massimo:** Se v ha figli, si identifica il figlio u con la chiave massima tra i due figli.
3. **Confronto con la radice:** Se la chiave di v è minore della chiave di u , si scambiano le chiavi di v e u .
4. **Ricorsione:** Si applica ricorsivamente la procedura **fixHeap** al figlio u per assicurarsi che anche il suo sottoalbero sia un heap.

Tempo di esecuzione

L'analisi del tempo di esecuzione della procedura **fixHeap** si basa sui seguenti punti:

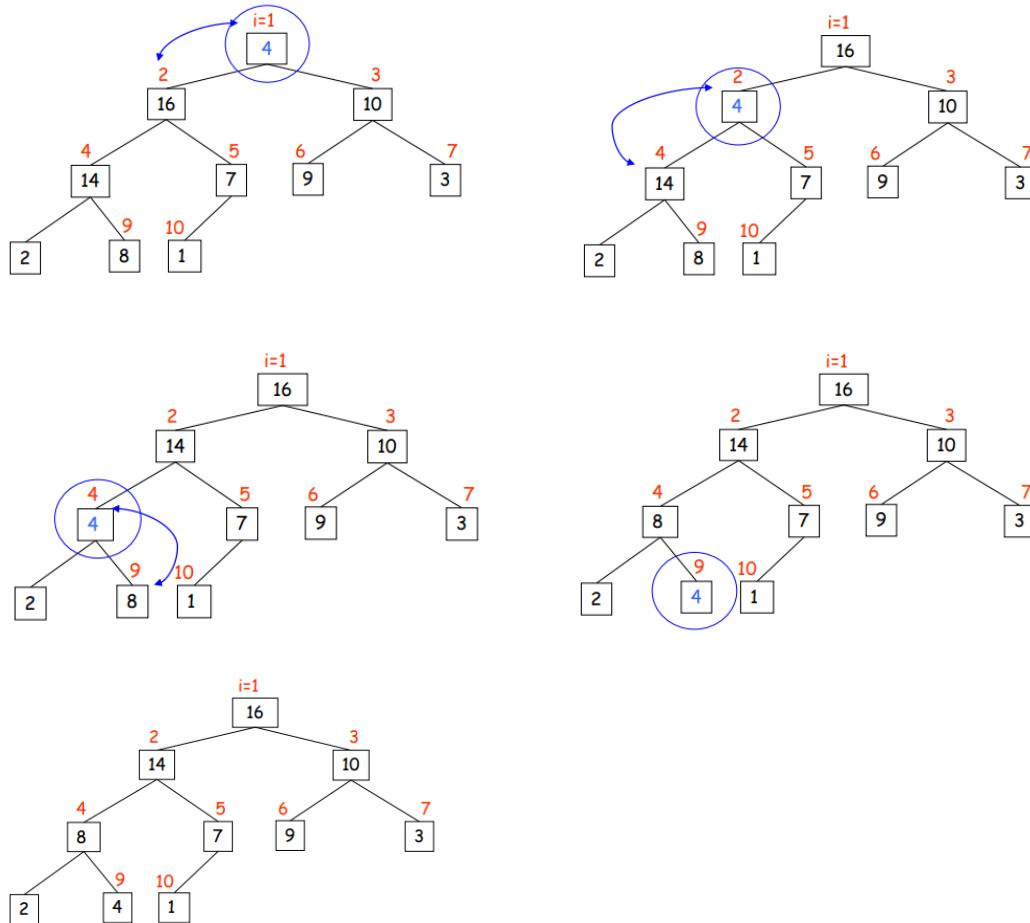
- L'unica operazione che richiede tempo è il confronto tra chiavi.
- Il numero massimo di confronti per ogni chiamata di **fixHeap** è 1 (confronto tra v e u).
- L'altezza massima dell'albero è $\log n$, dove n è il numero di nodi.
- Il numero massimo di chiamate ricorsive è $\log n$.

Pertanto, la complessità temporale della procedura `fixHeap` è:

$$T(n) = O(1) + T(n/2) = O(\log n)$$

Considerazioni

- La procedura `fixHeap` è efficiente e ripristina la proprietà di heap in tempo logaritmico.
- La procedura è utilizzata per implementare diverse operazioni sugli heap, come l'inserimento e la cancellazione di elementi.
- L'efficienza della procedura dipende dalla corretta implementazione della struttura dati heap.



Complessità: $O(\log n)$

Pseudocodice:

```
fixHeap (i,A)
1.   s=sin(i)
2.   d=des(i)
3.   if (s ≤ heapsize[A] e A[s] >A[i])
4.     then massimo=s
5.   else massimo=i
6.   if (d ≤ heapsize[A] e A[d] >A[massimo])
7.     then massimo=d
8.   if (massimo≠i)
9.     then scambia A[i] e A[massimo]
10.    fixHeap(massimo,A)
```

Estrazione del massimo

Descrizione

L'estrazione del massimo è un'operazione fondamentale sugli heap. Essa consiste nel rimuovere il massimo elemento dall'heap e restituirlo. L'algoritmo per estrarre il massimo si basa sui seguenti passi:

1. Copia il valore:

- Si copia la chiave contenuta nella foglia più a destra dell'ultimo livello nella radice dell'heap.
- In altre parole, si sposta l'elemento in posizione `heap-size` nella radice.

2. Rimuovi la foglia:

- Si rimuove la foglia più a destra dell'ultimo livello.
- Nella rappresentazione con array posizionale, ciò significa semplicemente decrementare il valore di `heap-size`.

3. Ripristina la proprietà di heap:

- Si applica la procedura `fixHeap` sulla radice per ripristinare la proprietà di heap sull'albero.

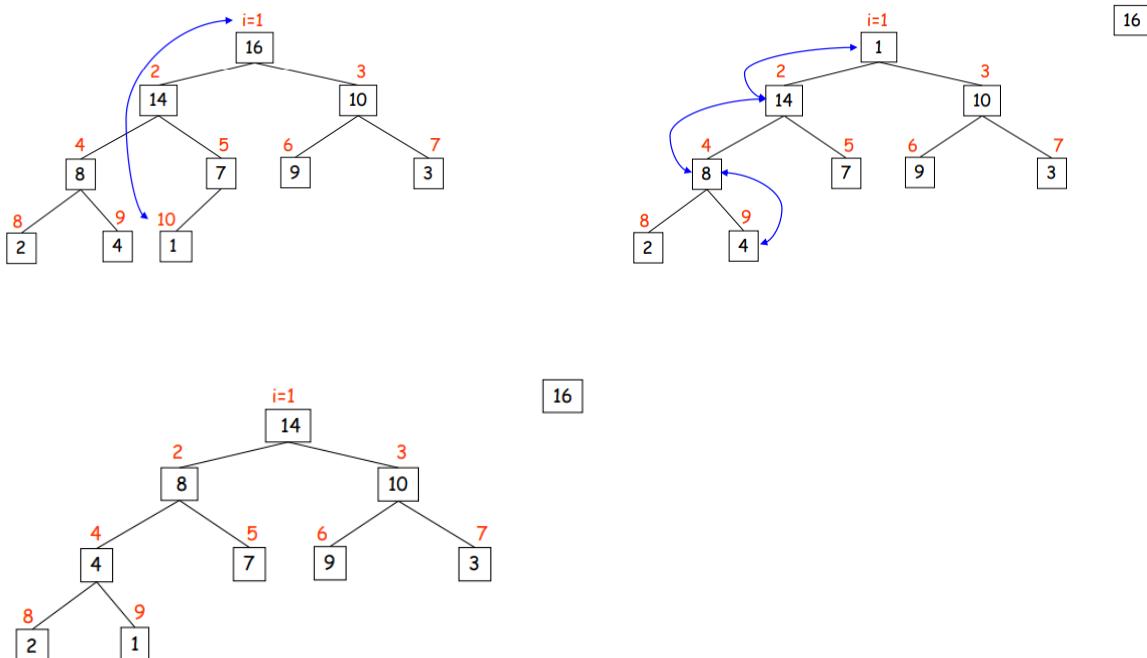
Tempo di esecuzione

L'analisi del tempo di esecuzione dell'estrazione del massimo si basa sui seguenti punti:

- L'operazione di copia richiede tempo costante.
- L'operazione di rimozione della foglia richiede tempo costante.
- La procedura `fixHeap` ha complessità temporale $O(\log n)$.

Pertanto, la complessità temporale dell'estrazione del massimo è:

$$T(n) = O(1) + O(1) + O(\log n) = O(\log n)$$



Costruzione dell'heap

Algoritmo ricorsivo

L'algoritmo per costruire un heap da un array di elementi si basa sulla tecnica del "divide et impera" e utilizza la procedura `heapify`. La procedura `heapify` funziona come segue:

1. Controllo:

- Se l'heap H è vuoto, non c'è nulla da fare.

2. Heapify dei sottoalberi:

- Si applica ricorsivamente la procedura `heapify` al sottoalbero sinistro di H .
- Si applica ricorsivamente la procedura `heapify` al sottoalbero destro di H .

3. Ripristino della proprietà di heap:

- Si applica la procedura `fixHeap` alla radice di H per assicurarsi che la proprietà di heap sia valida per l'intero albero.

```
heapify(heap H)
    if (H non è vuoto) then
        heapify(sottoalbero sinistro di H)
        heapify(sottoalbero destro di H)
        fixHeap(radice di H, H)
```

Complessità di `heapify`

Per analizzare la complessità di `heapify`, consideriamo un heap con n elementi e sia h la sua altezza. Scegliamo un intero n' tale che:

- n' è maggiore o uguale a n .
- Un heap con n' elementi ha:
 - Altezza h .
 - Struttura completa fino all'ultimo livello.

Inoltre, vale la seguente proprietà:

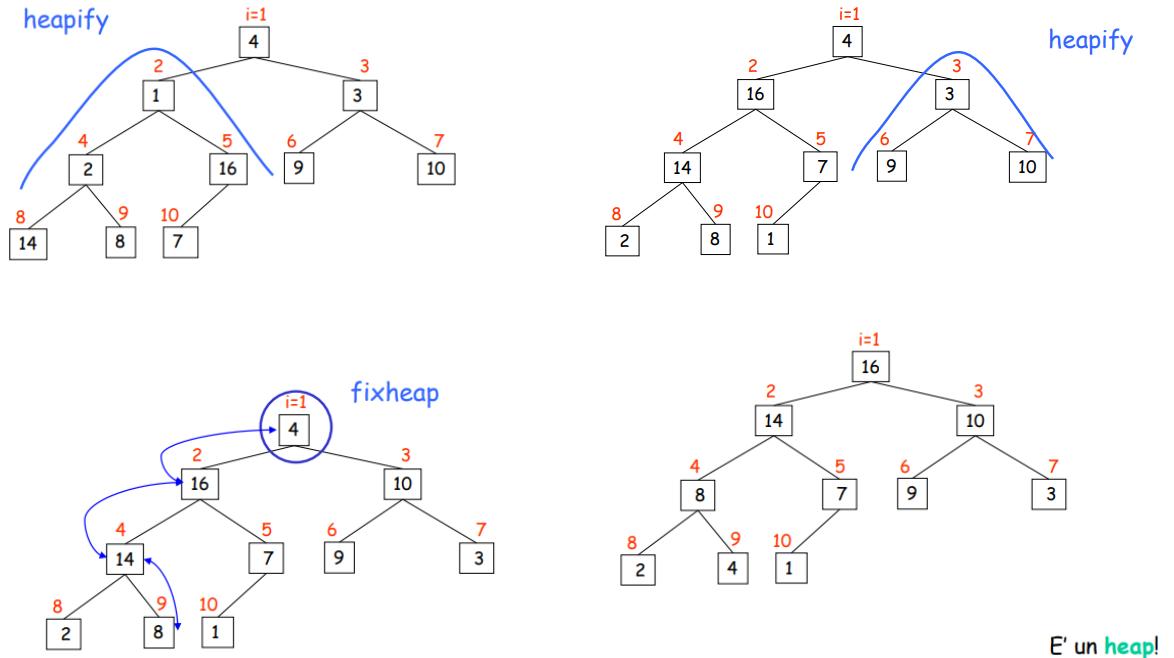
- $T(n)$ (complessità di `heapify` su un heap con n elementi) è minore o uguale a $T(n')$ (complessità di `heapify` su un heap con n' elementi).
- n' è minore o uguale a $2n$ (raddoppio del numero di elementi).

Applicando il Teorema Master, si ottiene:

- $T(n') = 2 T((n'-1)/2) + O(\log n')$
- $T(n') \leq 2 T(n'/2) + O(\log n')$
- $T(n') = O(n')$

Pertanto, la complessità di `heapify` su un heap con n elementi è:

- $T(n) \leq T(n') = O(n') = O(2n) = O(n)$



Max-Heap e Min-Heap

Un heap può essere implementato come un Max-Heap o un Min-Heap:

- **Max-Heap:** La chiave di un padre è sempre maggiore o uguale alla chiave del figlio (tranne per la radice). In questo caso, l'estrazione del massimo restituisce l'elemento con il valore maggiore.
- **Min-Heap:** La chiave di un padre è sempre minore o uguale alla chiave del figlio (tranne per la radice). In questo caso, l'estrazione del minimo restituisce l'elemento con il valore minore.

Per ottenere un Min-Heap, è sufficiente invertire la proprietà di ordinamento delle chiavi:

- **Min-Heap:** `chiave(padre(v)) <= chiave(v)` per ogni nodo v diverso dalla radice.
- L'uso del max-heap (implementato con un vettore posizionale) ci permette di usare solo memoria ausiliare costante pertanto useremo un Min-Heap.

L'algoritmo HeapSort

- Costruisce un heap tramite heapify
- Estraie ripetutamente il massimo per $n-1$ volte – ad ogni estrazione memorizza il massimo nella posizione dell'array che si è appena liberata.

heapSort (A)

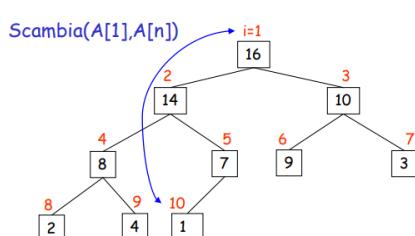
1. Heapify(A)
2. Heapsize[A]=n
3. **for** $i=n$ **down to** 2 **do**
4. scambia $A[1]$ e $A[i]$
5. Heapsize[A] = Heapsize[A] - 1
6. fixHeap(1,A)

ordina in loco in tempo $O(n \log n)$

Teorema

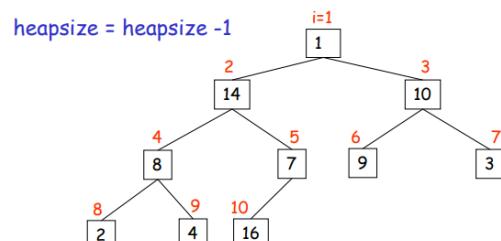
L'algoritmo HeapSort ordina in loco un array di lunghezza n in tempo $O(n \log n)$ nel caso peggiore.

Input: $A = <4, 1, 3, 2, 16, 9, 10, 14, 8, 7>$
 Heapify(A) $\rightarrow A_0 = <16, 14, 10, 8, 7, 9, 3, 2, 4, 1>$

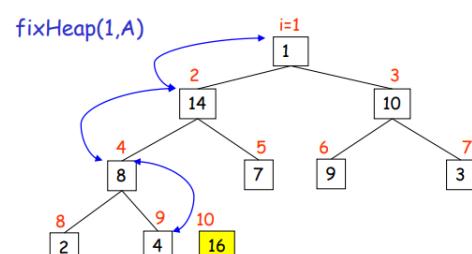
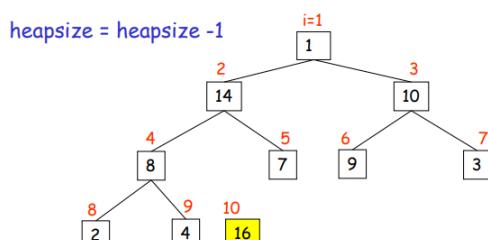


Input: $A = <4, 1, 3, 2, 16, 9, 10, 14, 8, 7>$
 Heapify(A) $\rightarrow A_0 = <16, 14, 10, 8, 7, 9, 3, 2, 4, 1>$

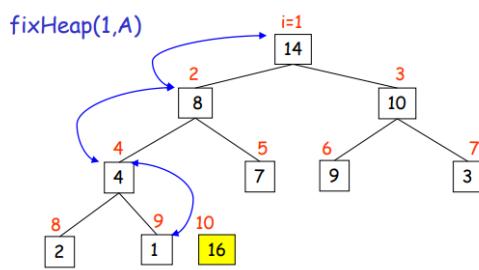
Input: $A = <4, 1, 3, 2, 16, 9, 10, 14, 8, 7>$
 Heapify(A) $\rightarrow A_0 = <16, 14, 10, 8, 7, 9, 3, 2, 4, 1>$



Input: $A = <4, 1, 3, 2, 16, 9, 10, 14, 8, 7>$
 Heapify(A) $\rightarrow A_0 = <16, 14, 10, 8, 7, 9, 3, 2, 4, 1>$

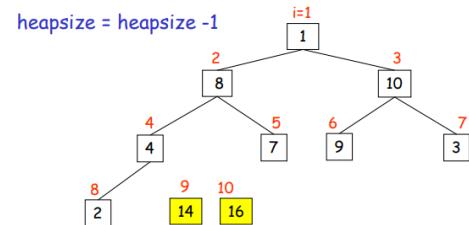
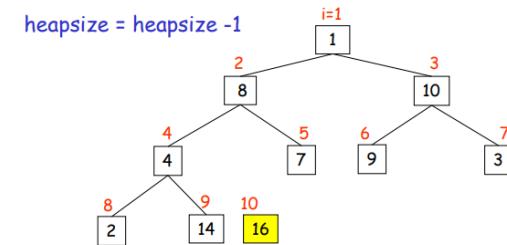
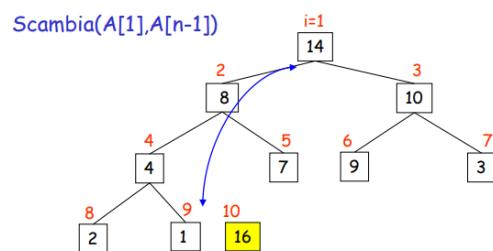


Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$
 $\text{Heapify}(A) \rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$



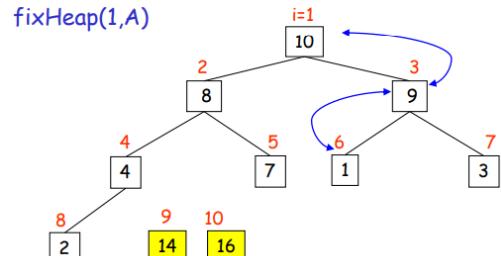
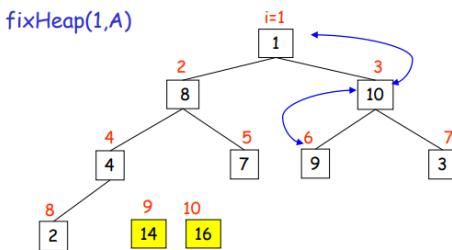
Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$
 $\text{Heapify}(A) \rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$

Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$
 $\text{Heapify}(A) \rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$



Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$
 $\text{Heapify}(A) \rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$

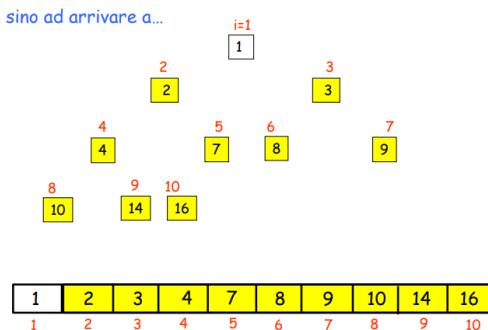
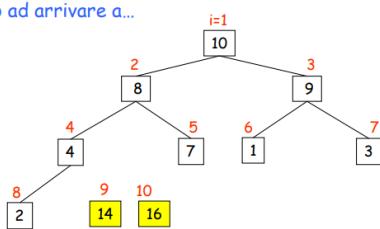
Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$
 $\text{Heapify}(A) \rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$



Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$
 $\text{Heapify}(A) \rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$

Input: $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$
 $\text{Heapify}(A) \rightarrow A_0 = \langle 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 \rangle$

E così via, sino ad arrivare a...



Complessità di un algoritmo: delimitazioni superiore e inferiore

Definizioni

Delimitazione superiore (Upper Bound):

Un algoritmo A ha complessità (costo di esecuzione) $O(f(n))$ rispetto ad una certa risorsa di calcolo se la quantità $r(n)$ di risorsa utilizzata da A nel caso peggiore su istanze di dimensione n verifica la relazione:

$$r(n) = O(f(n))$$

Delimitazione inferiore (Lower Bound):

Un algoritmo A ha complessità (costo di esecuzione) $\Omega(f(n))$ rispetto ad una certa risorsa di calcolo se la quantità $r(n)$ di risorsa utilizzata da A nel caso peggiore su istanze di dimensione n verifica la relazione:

$$r(n) = \Omega(f(n))$$

Complessità di un problema:

Delimitazione superiore:

Un problema P ha una complessità $O(f(n))$ rispetto ad una risorsa di calcolo se esiste un algoritmo che risolve P il cui costo di esecuzione rispetto a quella risorsa è $O(f(n))$.

Delimitazione inferiore:

Un problema P ha una complessità $\Omega(f(n))$ rispetto ad una risorsa di calcolo se ogni algoritmo che risolve P ha costo di esecuzione nel caso peggiore $\Omega(f(n))$ rispetto a quella risorsa.

Ottimalità di un algoritmo

Definizione:

Dato un problema P con complessità $\Omega(f(n))$ rispetto ad una risorsa di calcolo, un algoritmo che risolve P è (asintoticamente) ottimo se ha costo di esecuzione $O(f(n))$ rispetto a quella risorsa.

complessità temporale del problema dell'ordinamento

- Upper bound: $O(n^2)$
 - Insertion Sort, Selection Sort, Quick Sort, Bubble Sort
- Un upper bound migliore: $O(n \log n)$
 - Merge Sort, Heap Sort
- Lower bound: $\Omega(n)$
 - banale: ogni algoritmo che ordina n elementi li deve almeno leggere tutti

Abbiamo un **gap di log n** tra upper bound e lower bound!

Sui limiti della velocità: una delimitazione inferiore (lower bound) alla complessità del problema:

Teorema: Ogni algoritmo basato su confronti che ordina n elementi deve fare nel caso peggiore $\Omega(n \log n)$ confronti.

Nota: il #di confronti che un algoritmo esegue è un **lower bound** al #di passi elementari che esegue

Albero di decisione per gli algoritmi di ordinamento per confronto

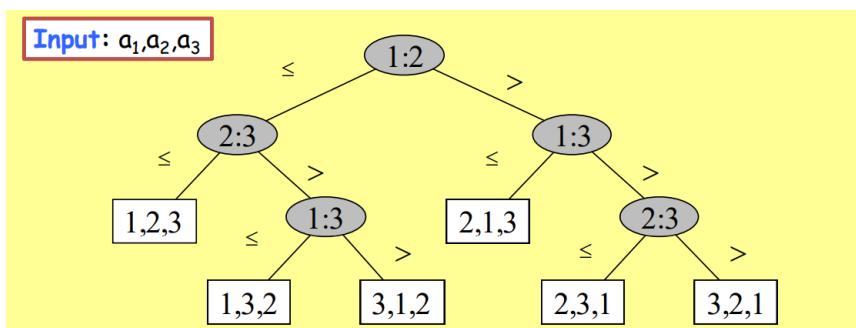
Introduzione

Gli algoritmi di ordinamento per confronto, come il Bubble Sort, il Selection Sort e il Merge Sort, possono essere descritti in modo astratto utilizzando gli alberi di decisione. Un albero di decisione rappresenta le diverse sequenze di

confronti che un algoritmo può eseguire su un input di una determinata dimensione.

Struttura di un albero di decisione

- **Nodo interno (non foglia):** Rappresenta un confronto tra due elementi a_i e a_j . L'etichetta del nodo indica il confronto da eseguire (ad esempio $a_i \leq a_j$).
- **Arco:** Rappresenta l'esito del confronto. Un arco che discende da un nodo indica che il confronto è vero, mentre un arco che sale indica che il confronto è falso.
- **Nodo foglia:** Rappresenta una possibile permutazione degli elementi, ovvero l'output dell'algoritmo su un'istanza specifica.



Proprietà

- L'albero di decisione non è associato a un problema specifico, ma a un algoritmo e a una dimensione dell'istanza.
- L'albero di decisione descrive le diverse sequenze di confronti che un algoritmo può eseguire su istanze di una data dimensione.
- L'albero di decisione rappresenta una descrizione alternativa dell'algoritmo, personalizzata per istanze di una specifica dimensione.

Vantaggi degli alberi di decisione

- Gli alberi di decisione forniscono una rappresentazione visiva compatta del comportamento di un algoritmo di ordinamento per confronto.
- Permettono di analizzare il numero massimo di confronti che un algoritmo può eseguire su un'istanza di data dimensione.
- Possono essere utilizzati per confrontare l'efficienza di diversi algoritmi di ordinamento.

Limiti degli alberi di decisione

- La rappresentazione degli alberi di decisione diventa rapidamente complessa per istanze di grandi dimensioni.

- Non forniscono informazioni sul tempo di esecuzione esatto dell'algoritmo.

Relazione tra algoritmo e albero di decisione

- **Cammino radice-foglia:** Per una specifica istanza, i confronti eseguiti dall'algoritmo durante l'ordinamento corrispondono a un preciso cammino dall'inizio (radice) dell'albero di decisione fino a una foglia.
- **Diversi cammini:** L'algoritmo segue un cammino diverso a seconda delle caratteristiche dell'istanza da ordinare.
- **Caso peggiore:** Il cammino più lungo dell'albero di decisione rappresenta il caso peggiore, ovvero la sequenza di confronti che richiede il maggior numero di operazioni per ordinare l'istanza.
- **Numero di confronti nel caso peggiore:** Il numero di confronti nel caso peggiore è pari all'altezza dell'albero di decisione.

Proprietà generali

- **Foglie e permutazioni:** Un albero di decisione di un algoritmo (corretto) che risolve il problema dell'ordinamento di n elementi deve avere necessariamente almeno $n!$ foglie. Ogni foglia rappresenta una possibile permutazione ordinata degli n elementi.

Proprietà di un albero binario (LEMMA)

- **Altezza di un albero binario con k foglie:** Un albero binario T con k foglie ha altezza:

$$h = \log_2(k)$$

dim (per induzione su k)

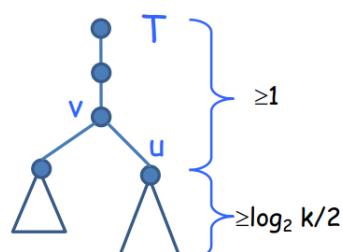
caso base: $k=1$ altezza almeno $\log_2 1=0$

caso induttivo: $k>1$

considera il nodo interno v più vicino alla radice che ha due figli (v potrebbe essere la radice). nota che v deve esistere perché $k>1$.

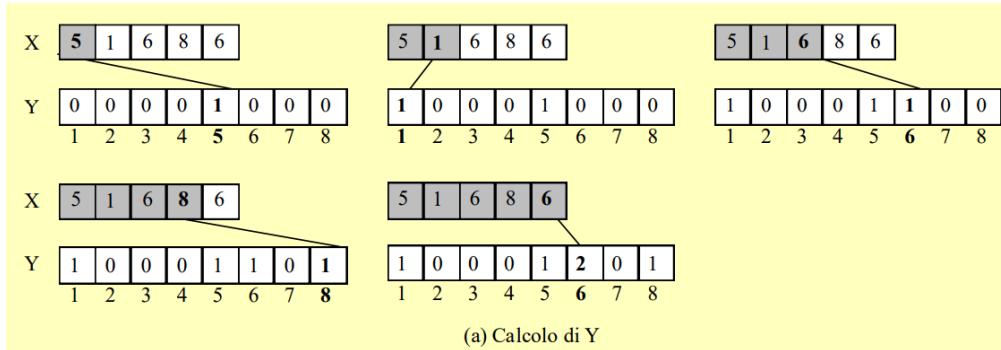
v ha almeno un figlio u che è radice di un (sotto)albero che ha almeno $k/2$ foglie e $< k$ foglie.

T ha altezza almeno
 $1 + \log_2 k/2 = 1 + \log_2 k - \log_2 2 = \log_2 k$

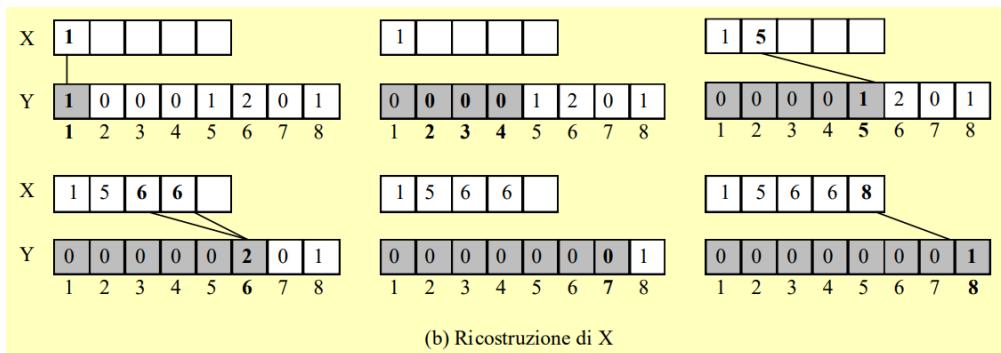


IntegerSort:

- 1) Per ordinare n interi con valori in [1,k] Mantiene un array Y di k contatori tale che $Y[x] = \text{numero di volte che il valore } x \text{ compare in } X$



- 2) Scorre Y da sinistra verso destra e, se $Y[x]=k$, scrive in X il valore x per k volte.



Pseudocodice:

IntegerSort (X, k)

1. Sia Y un array di dimensione k } $O(1)$ - tempo costante
 2. **for** $i=1$ **to** k **do** $Y[i]=0$ } $O(k)$
 3. **for** $i=1$ **to** n **do** incrementa $Y[X[i]]$ } $O(n)$
 4. $j=1$ } $O(1)$
 5. **for** $i=1$ **to** k **do** } $O(k)$
 6. **while** ($Y[i] > 0$) **do** } per i fissato
 7. $X[j]=i$ #volte eseguite è al più $1+Y[i]$
 8. incrementa j
 9. decrementa $Y[i]$
- $\Rightarrow O(k+n)$

$$\sum_{i=1}^k (1+Y[i]) = \sum_{i=1}^k 1 + \sum_{i=1}^k Y[i] = k + n$$

Complessità temporale totale:

- $O(k) + O(n) = O(n + k)$

Bucket Sort

Descrizione

Il Bucket Sort è un algoritmo di ordinamento che sfrutta la conoscenza del range di valori delle chiavi per distribuire i record in "cestini" o "bucket". L'algoritmo si divide in due fasi:

Fase 1: Distribuzione nei bucket:

1. Creare un array di k liste vuote, dove k è il range di valori delle chiavi.
2. Percorrere l'array di record e inserire ogni record nella lista corrispondente alla sua chiave.

Fase 2: Concatenamento delle liste:

1. Concatenare le liste in ordine crescente delle chiavi, ottenendo l'array ordinato.

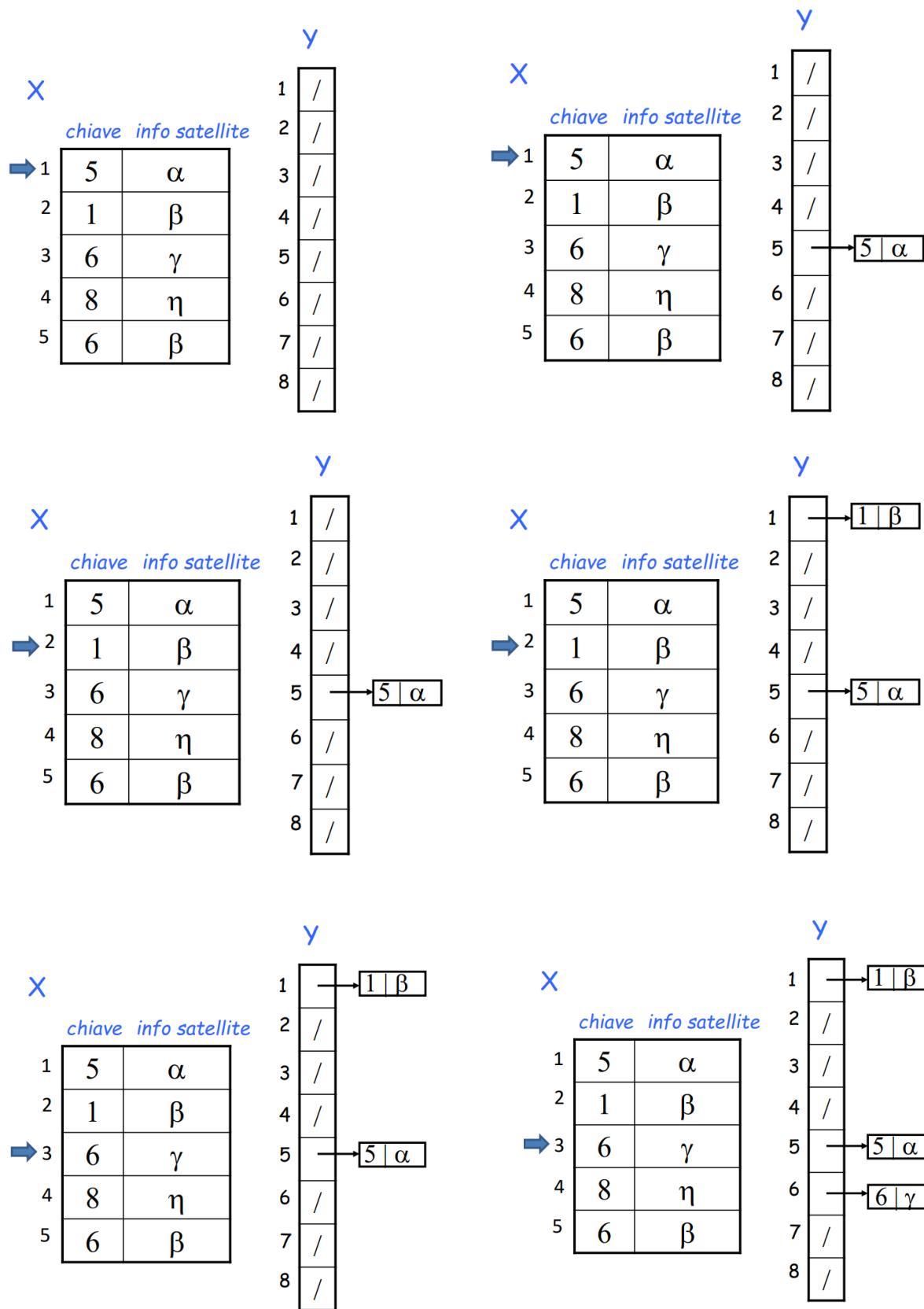
Complessità temporale totale:

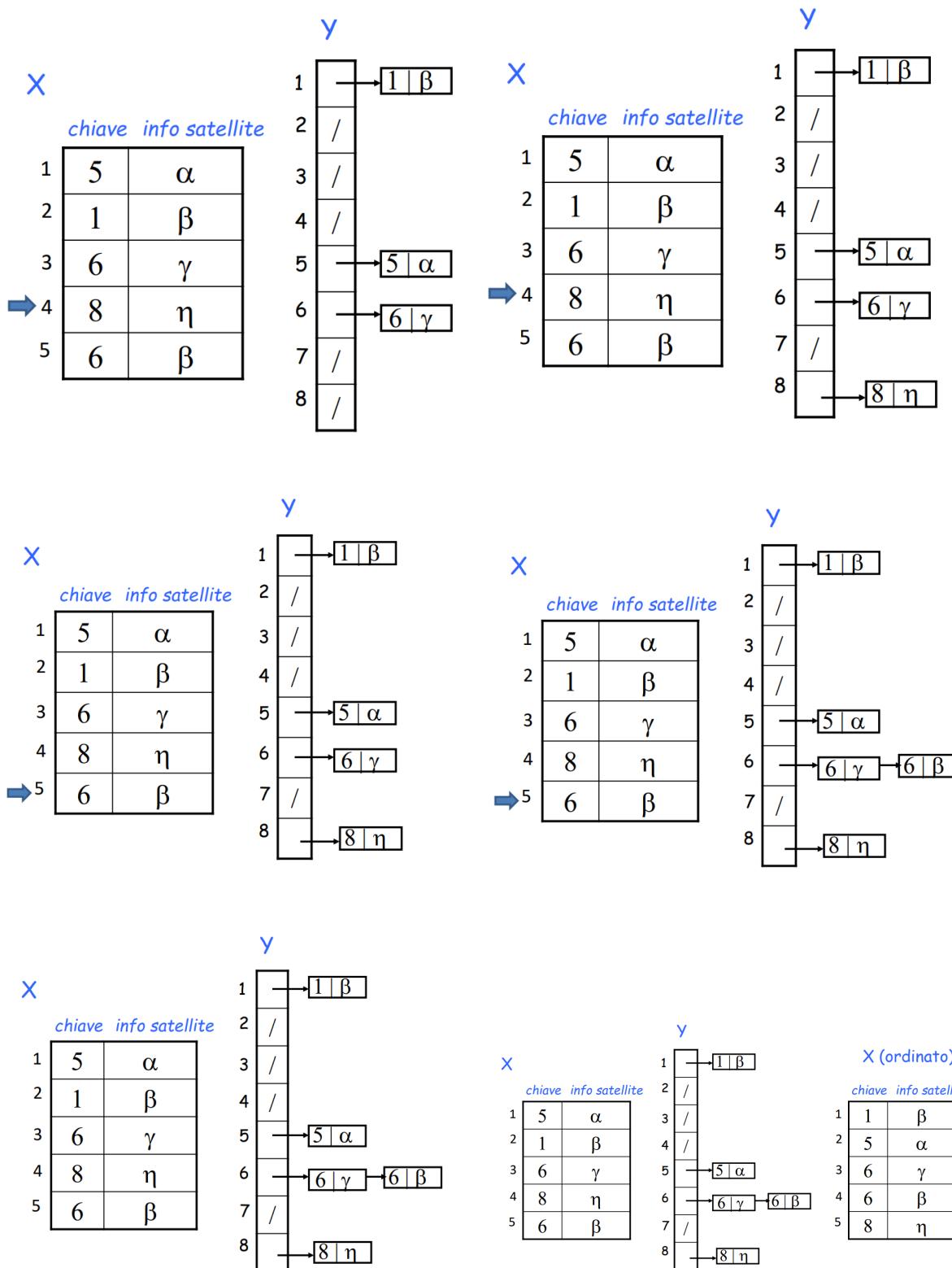
- $O(k) + O(n) = O(n + k)$

Pseudocodice:

BucketSort (X, k)

1. Sia Y un array di dimensione k
2. **for** $i=1$ **to** k **do** $Y[i]=$ lista vuota
3. **for** $i=1$ **to** n **do**
4. appendi il record $X[i]$ alla lista $Y[\text{chiave}(X[i])]$
5. **for** $i=1$ **to** k **do**
6. copia ordinatamente in X gli elementi della lista $Y[i]$





Stabilità

Un algoritmo di ordinamento è stabile se preserva l'ordine iniziale tra elementi con la stessa chiave.

Il Bucket Sort è stabile se si appende ogni elemento di x in coda alla opportuna lista $y[i]$.

In questo modo, gli elementi con la stessa chiave saranno mantenuti nell'ordine originale all'interno del rispettivo bucket.

Radix Sort

Descrizione

Il Radix Sort è un algoritmo di ordinamento che sfrutta la rappresentazione posizionale dei numeri per ordinarli in base alle loro cifre. L'algoritmo si basa su ripetute applicazioni del Bucket Sort, ordinando i numeri per ogni cifra a partire dalla meno significativa fino alla più significativa.

Implementazione

1. Rappresentazione in base b : Rappresentare gli n interi da ordinare in base b (ad esempio, base 10 per numeri decimali).
2. Ciclo sulle cifre: Per ogni cifra t da 0 a numero di cifre del valore massimo (escluso):
 - Estrazione della cifra: Estrarre la t -esima cifra di ogni numero e usarla come chiave per il Bucket Sort.
 - Bucket Sort: Applicare il Bucket Sort ordinando i numeri in base alla t -esima cifra estratta.
3. Output ordinato: L'array ottenuto dopo l'ultima passata del Bucket Sort rappresenta l'array ordinato.

Correttezza

Lemma:

- Se x e y hanno una diversa t -esima cifra, la t -esima passata di Bucket Sort li ordina correttamente.
- Se x e y hanno la stessa t -esima cifra, la proprietà di stabilità del Bucket Sort li mantiene ordinati correttamente.

Dimostrazione:

- Diversi valori della t -esima cifra: Se la t -esima cifra di x è diversa da quella di y , il Bucket Sort li inserirà in bucket diversi, ordinandoli correttamente.

- Stessi valori della t -esima cifra: La stabilità del Bucket Sort garantisce che se x precede y nell'array prima della t -esima passata, lo precederà anche dopo, mantenendo l'ordine originale.

Tempo di esecuzione

Numero di passate:

- Il numero di cifre per rappresentare il valore massimo k in base b è $O(\log_b(k))$.

Tempo per ogni passata:

- Ciascuna passata richiede tempo $O(n + b)$, poiché la chiave in ogni passata è un intero in $[0, b-1]$.

Tempo di esecuzione totale:

- $O((n + b) \log_b(k))$

Considerazioni su b :

- Se $b = \Theta(n)$, si ha $O((n + b) \log_b(k)) = O(n \log_b(k)) = O(n (\log k / \log n))$.
- In questo caso, il tempo di esecuzione diventa lineare se $k = O(n^c)$, dove c è una costante.

PROBLEMA ORACOLO:

Problema 4.10

Dato un vettore X di n interi in $[1, k]$, costruire in tempo $O(n+k)$ una struttura dati ([oracolo](#)) che sappia rispondere a domande ([query](#)) in tempo $O(1)$ del tipo: “quanti interi in X cadono nell’intervallo $[a, b]$?””, per ogni a e b .

SOLUZIONE MIGLIORE:

Idea: Costruire in tempo $O(n+k)$ un array Y di dimensione k dove $Y[i]$ è il numero di elementi di X che sono $\leq i$

CostruisciOracolo (X, k)

1. Sia Y un array di dimensione k
2. **for** $i=1$ **to** k **do** $Y[i]=0$
3. **for** $i=1$ **to** n **do** incrementa $Y[X[i]]$
4. **for** $i=2$ **to** k **do** $Y[i]=Y[i]+Y[i-1]$
5. **return** Y

InterrogaOracolo (Y, k, a, b)

1. **if** $b > k$ **then** $b=k$
 2. **if** $a \leq 1$ **then return** $Y[b]$
- else return** $(Y[b]-Y[a-1])$

Come implementare efficientemente un dizionario

Introduzione

Un dizionario è una struttura dati che associa chiavi a valori. Le operazioni principali su un dizionario includono:

- **Inserimento:** Inserire una nuova coppia chiave-valore nel dizionario.
- **Ricerca:** Trovare il valore associato a una data chiave nel dizionario.
- **Eliminazione:** Rimuovere una coppia chiave-valore dal dizionario.

L’efficienza di un dizionario è valutata in base al tempo di esecuzione di queste operazioni fondamentali. In questo articolo, esploreremo come implementare un dizionario in modo da garantire che tutte le operazioni abbiano un tempo di esecuzione di $O(\log n)$, dove n è il numero di elementi nel dizionario.

Due approcci

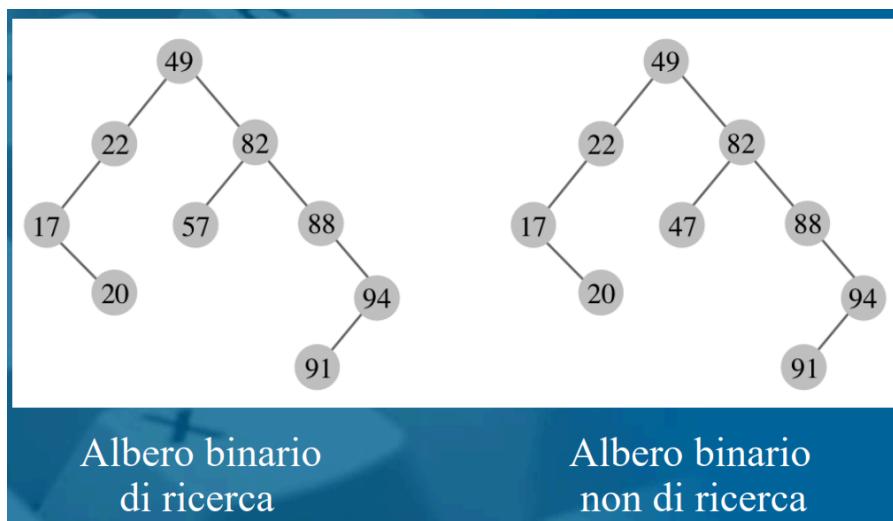
Esistono due approcci principali per implementare un dizionario efficiente:

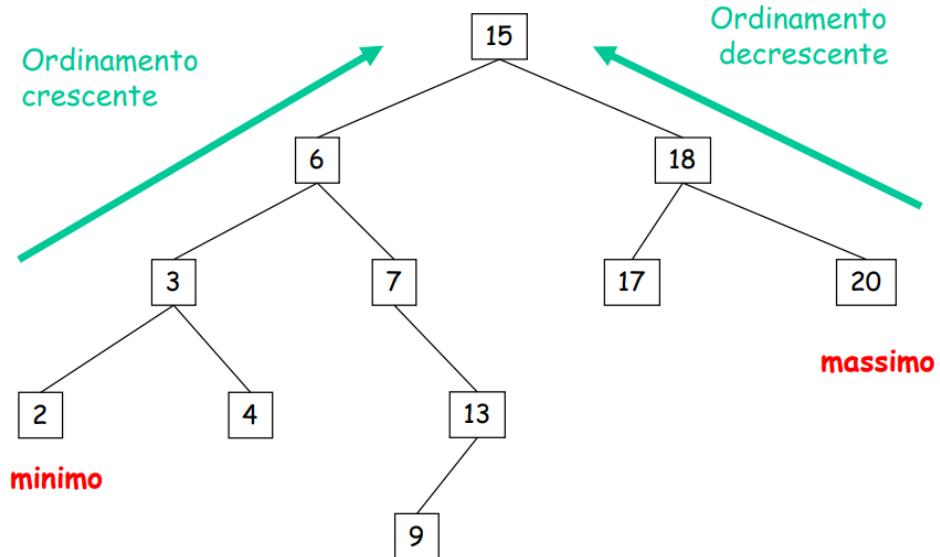
1. Utilizzare un albero (binario):
 - o Definire un albero tale che ogni operazione richieda tempo proporzionale all'altezza dell'albero. $O(\text{altezza albero})$
2. Mantenere l'altezza dell'albero sempre logaritmica utilizzando strutture come gli alberi AVL. $O(\log n)$

Alberi binari di ricerca (BST)

Un albero binario di ricerca (BST) è un albero binario che soddisfa le seguenti proprietà:

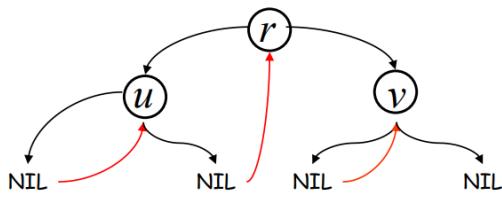
- Ogni nodo v contiene un elemento $\text{elem}(v)$ associato a una chiave $\text{chiave}(v)$ presa da un dominio totalmente ordinato.
- Per ogni nodo v vale che:
 - o Le chiavi nel sottoalbero sinistro di v sono minori o uguali a $\text{chiave}(v)$.
 - o Le chiavi nel sottoalbero destro di v sono maggiori a $\text{chiave}(v)$.



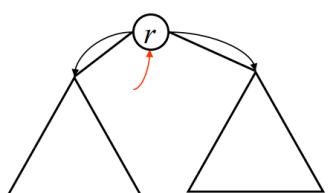


Verifica di correttezza:

Indichiamo con h l'altezza dell'albero. Vogliamo mostrare che la visita in ordine simmetrico restituisce la sequenza ordinata Per induzione sull'altezza dell'ABR: $h=1$ (mostriamolo senza perdita di generalità quando l'albero è completo.)



$$\rightarrow \text{chiave}(u) \leq \text{chiave}(r) \leq \text{chiave}(v)$$



Albero di altezza $\leq h-1$.
Tutti i suoi elementi sono minori o uguali della radice

Albero di altezza $\leq h-1$.
Tutti i suoi elementi sono maggiori o uguali della radice

$h = \text{generico}$ (ipotizzo che la procedura sia corretta per altezza $< h$)

Implementazione della funzione `search(chiave k, elem e)`

Traccia un cammino nell'albero partendo dalla radice: su ogni nodo, usa la proprietà di ricerca per decidere se proseguire nel sottoalbero sinistro o destro

```
algoritmo search(chiave k) → elem
1.   v ← radice di T
2.   while (v ≠ null) do
3.     if (k = chiave(v)) then return elem(v)
4.     else if (k < chiave(v)) then v ← figlio sinistro di v
5.     else v ← figlio destro di v
6.   return null
```

Implementazione della funzione `insert(elem e, chiave k)`

Descrizione

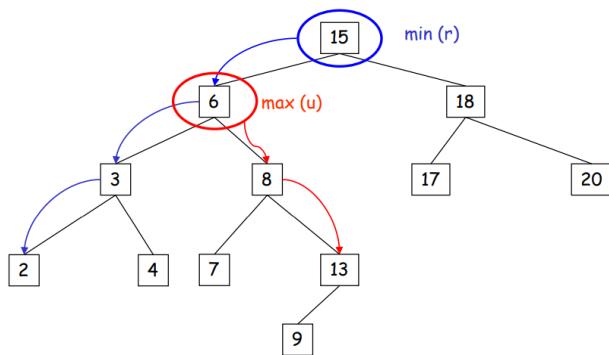
La funzione `insert(elem e, chiave k)` inserisce un nuovo elemento *e* con chiave *k* in un albero binario di ricerca (BST). L'algoritmo si basa sull'idea di simulare una ricerca per identificare la posizione corretta per il nuovo nodo foglia.

Ricerca del massimo:

algoritmo max(*nodo u*) → *nodo*

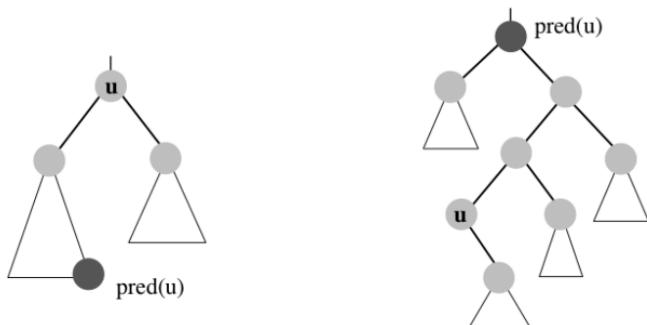
1. $v \leftarrow u$
2. **while** (figlio destro di $v \neq \text{null}$) **do**
3. $v \leftarrow \text{figlio destro di } v$
4. **return** v

Nota: è possibile definire una procedura min(*nodo u*) in maniera del tutto analoga.



predecessore e successore

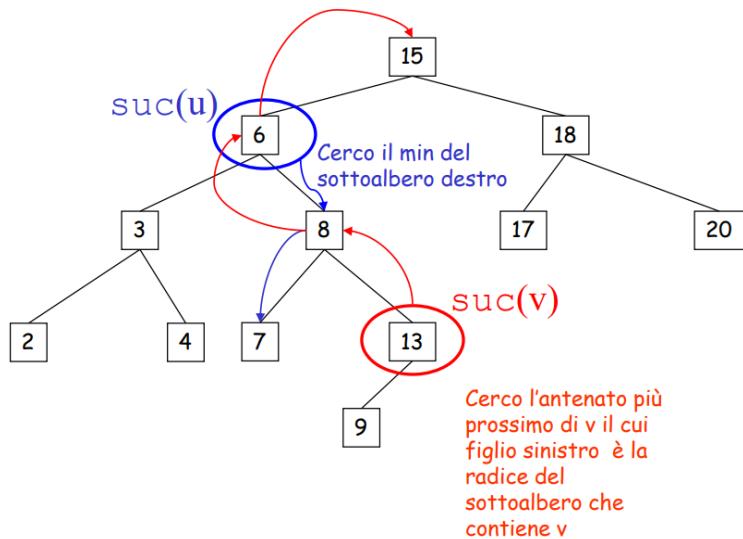
- il predecessore di un nodo u in un BST è il nodo v nell'albero avente massima chiave $\leq \text{chiave}(u)$
- il successore di un nodo u in un BST è il nodo v nell'albero avente minima chiave $\geq \text{chiave}(u)$.



algoritmo pred(*nodo u*) → *nodo*

1. **if** (u ha figlio sinistro $sin(u)$) **then**
2. **return** max($sin(u)$)
3. **while** ($parent(u) \neq \text{null}$ e u è figlio sinistro di suo padre) **do**
4. $u \leftarrow parent(u)$
5. **return** $parent(u)$

Nota: la ricerca del successore di un nodo è simmetrica

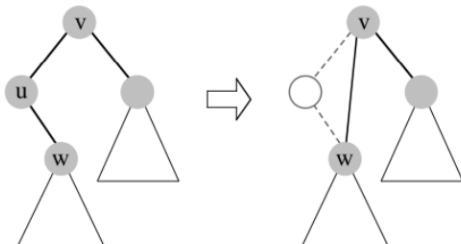


Implementazione della funzione `delete (elem e)`

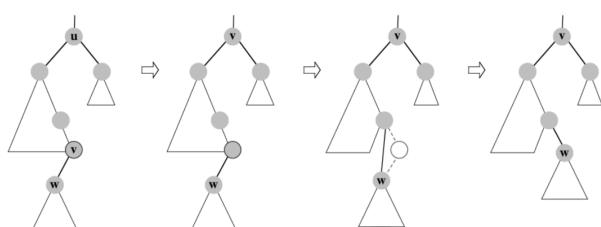
Descrizione

La funzione `delete (elem e)` elimina un elemento con valore `e` da un albero binario di ricerca (BST). L'algoritmo si basa su tre casi principali:

1. **Nodo foglia:** Se il nodo da eliminare è una foglia, viene semplicemente rimosso.
2. **Nodo con un solo figlio:** Se il nodo da eliminare ha un solo figlio, questo figlio diventa il figlio del padre del nodo eliminato.



3. **Nodo con due figli:** Se il nodo da eliminare ha due figli, viene sostituito con il suo predecessore (o successore) e il predecessore (o successore) viene eliminato.



Costo delle operazioni

- Tutte le operazioni hanno costo $O(h)$ dove h è l'altezza dell'albero.
- $O(n)$ nel caso peggiore (alberi molto sbilanciati e profondi).

Alberi AVL (Adel'son-Vel'skii e Landis, 1962)

Definizioni

Fattore di bilanciamento ($\beta(v)$) di un nodo v :

- La differenza tra l'altezza del sottoalbero sinistro di v e l'altezza del sottoalbero destro di v .

Albero bilanciato in altezza:

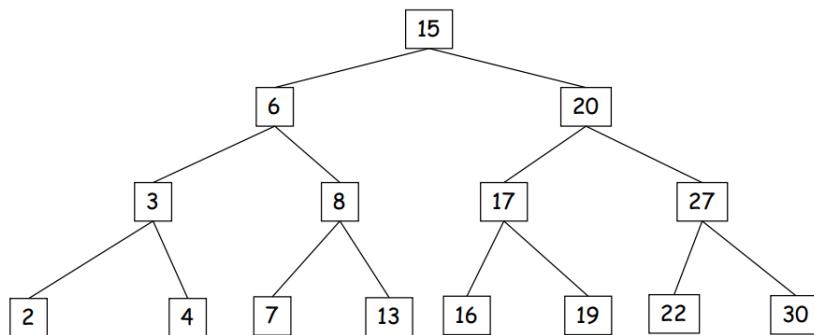
- Un albero in cui ogni nodo v ha un fattore di bilanciamento $\beta(v)$ in valore assoluto minore o uguale a 1.

Alberi AVL:

- Alberi binari di ricerca bilanciati in altezza.

Memorizzazione del fattore di bilanciamento:

- Il valore di $\beta(v)$ viene generalmente memorizzato come informazione aggiuntiva nel record associato al nodo v .



Dimostrazione che l'altezza di un albero AVL è $O(\log n)$

Introduzione

In questa sezione dimostreremo che un albero AVL con n nodi ha un'altezza massima di $O(\log n)$.

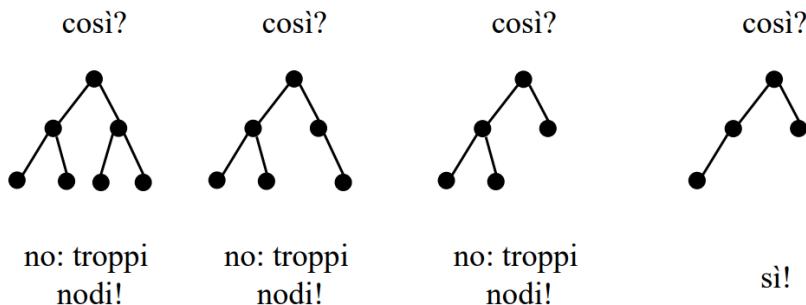
Idea della dimostrazione: considerare, tra tutti gli AVL, i più sbilanciati

Albero di Fibonacci di altezza h : albero AVL di altezza h con il minimo numero di nodi n_h

$$\begin{array}{ccc} \text{minimizzare \# nodi} & \equiv & \text{massimizzare altezza} \\ \text{fissata l'altezza} & & \text{fissato \#nodi} \end{array}$$

Intuizione: se gli alberi di Fibonacci hanno altezza $O(\log n)$, allora tutti gli alberi AVL hanno altezza $O(\log n)$

Come è fatto un albero di Fibonacci di altezza 2?:



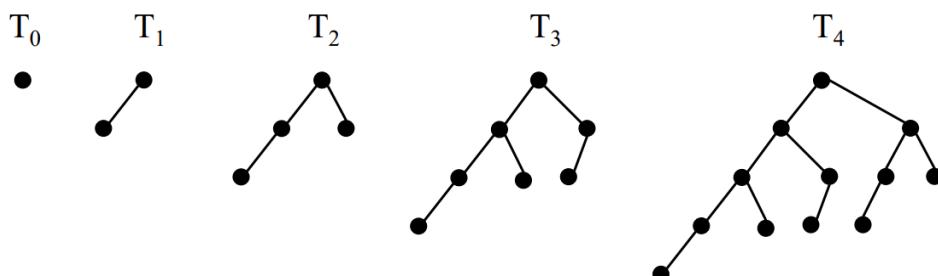
Infatti: se togliamo

ancora un nodo, o diventa sbilanciato, o cambia la sua altezza.

Nota: ogni nodo (non foglia) ha fattore di bilanciamento pari (in valore assoluto) a 1.

Alberi di Fibonacci (generici) piccoli valori h :

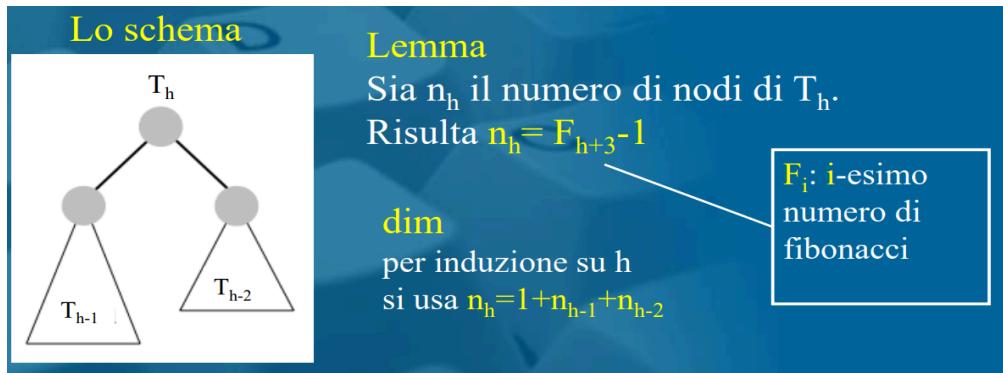
T_i : albero di Fibonacci di altezza i (albero AVL di altezza i con il minimo numero di nodi):



Nota che: se a T_i tolgo un nodo, o diventa sbilanciato, o cambia la sua altezza.

Inoltre: ogni nodo (non foglia) ha fattore di bilanciamento pari (in valore assoluto) a 1.

Lemma e schema :



Proprietà degli alberi AVL

- Proprietà di ricerca: La ricerca in un albero AVL ha tempo medio di esecuzione di $O(\log n)$, dove n è il numero di nodi.
- Proprietà di inserimento: L'inserimento di un nuovo elemento in un albero AVL richiede tempo medio di esecuzione di $O(\log n)$.
- Proprietà di eliminazione: L'eliminazione di un elemento da un albero AVL richiede tempo medio di esecuzione di $O(\log n)$.

Mantenimento del bilanciamento

Gli alberi AVL mantengono il bilanciamento inserendo e rimuovendo nodi in modo da garantire che il fattore di bilanciamento di ogni nodo sia sempre compreso tra -1 e 1.

Esistono due tipi principali di operazioni di bilanciamento:

- Rotazioni: Vengono utilizzate per modificare la struttura dell'albero in modo da ripristinare il bilanciamento dopo un'inserimento o un'eliminazione.
- Inserimenti speciali: In alcuni casi specifici di inserimento, è possibile inserire un nuovo nodo in modo da mantenere il bilanciamento senza dover eseguire rotazioni.

Vantaggi degli alberi AVL

- Efficienza di ricerca, inserimento ed eliminazione con tempo medio di esecuzione di $O(\log n)$.
- Bilanciamento automatico che garantisce prestazioni efficienti indipendentemente dalla sequenza di inserimenti e rimozioni.

- Semplicità di implementazione rispetto ad altre strutture dati bilanciate come gli alberi B.

Svantaggi degli alberi AVL

- Maggiore complessità di implementazione rispetto agli alberi binari di ricerca standard.
- Necessità di memorizzare il fattore di bilanciamento per ogni nodo.
- Possibili degradazioni delle prestazioni in caso di implementazioni non ottimali.

Classe AlberoAVL:

classe AlberoAVL estende AlberoBinarioDiRicerca:

dati:

albero binario di ricerca T ereditato, più il fattore di bilanciamento di ogni nodo.

$$S(n) = O(n)$$

operazioni:

search(chiave k) → elem $T(n) = O(\log n)$
ereditata.

insert(elem e, chiave k) $T(n) = O(\log n)$
chiama **insert()** ereditata, poi ricalcola i fattori di bilanciamento ed eventualmente ribilancia tramite $O(1)$ rotazioni.

delete(elem e) $T(n) = O(\log n)$
chiama **delete()** ereditata, poi ricalcola i fattori di bilanciamento ed eventualmente ribilancia tramite $O(\log n)$ rotazioni.

TIPO di Dato: CodaPriorità

Pseudocodice:

tipo CodaPriorita:

dati:

un insieme S di n elementi di tipo $elem$ a cui sono associate chiavi di tipo $chiave$ prese da un universo totalmente ordinato.

operazioni:

findMin() → elem
restituisce l'elemento in S con la chiave minima.

insert(elem e, chiave k)
aggiunge a S un nuovo elemento e con chiave k .

delete(elem e) Suppongo sempre che mi venga dato un riferimento diretto all'elemento da cancellare
cancella da S l'elemento e .

deleteMin()
cancella da S l'elemento con chiave minima.

Operazioni aggiuntive:

increaseKey(*elem e, chiave d*)
incrementa della quantità *d* la chiave dell'elemento *e* in *S*.

decreaseKey(*elem e, chiave d*)
decrementa della quantità *d* la chiave dell'elemento *e* in *S*.

merge(CodaPriorita *c₁*, CodaPriorita *c₂*) → CodaPriorita
restituisce una nuova coda con priorità *c₃* = *c₁* ∪ *c₂*.

Una coda di priorità è una struttura dati che associa a ciascun elemento un valore di priorità. Le operazioni fondamentali su una coda di priorità includono:

- **Insert:** Inserisce un nuovo elemento con la sua priorità nella coda.
- **FindMin:** Trova l'elemento con la massima priorità nella coda.
- **DeleteMin:** Rimuove l'elemento con la massima priorità dalla coda.
- **Delete:** Rimuove un elemento specifico dalla coda.

L'efficienza di una coda di priorità è valutata in base al tempo di esecuzione di queste operazioni. In questo articolo, esamineremo quattro implementazioni elementari di una coda di priorità e analizzeremo le loro prestazioni per le operazioni fondamentali.

Implementazioni elementari

1. Array non ordinato

- **Dimensione:** L'array viene dimensionato sufficientemente grande per ospitare tutti gli elementi.
- **Numero di elementi:** Il numero di elementi nella coda viene tenuto traccia in una variabile *n*.
- **FindMin:** Richiede tempo $\Theta(n)$ (bisogna controllare tutti gli elementi).
- **Insert:** Richiede tempo $O(1)$ (inserimento in fondo all'array).
- **Delete:** Richiede tempo $O(1)$ (rimozione dell'ultimo elemento).
- **DeleteMin:** Richiede tempo $\Theta(n)$ (bisogna prima trovare il minimo e poi rimuoverlo).

2. Array ordinato

- **Dimensione:** L'array viene dimensionato sufficientemente grande per ospitare tutti gli elementi.
- **Ordinamento:** L'array viene mantenuto ordinato in ordine decrescente per priorità.

- **Numero di elementi:** Il numero di elementi nella coda viene tenuto traccia in una variabile n .
- **FindMin:** Richiede tempo $O(1)$ (il minimo è in fondo all'array).
- **Insert:** Richiede tempo $O(n)$ (bisogna trovare la posizione corretta e spostare gli elementi successivi).
- **Delete:** Richiede tempo $O(n)$ (bisogna spostare gli elementi successivi dopo la rimozione).
- **DeleteMin:** Richiede tempo $O(1)$ (rimozione dell'ultimo elemento).

3. Lista non ordinata

- **Lista bidirezionale:** La lista viene implementata come una lista bidirezionale per facilitare l'inserimento e l'eliminazione in entrambi i lati.
- **FindMin:** Richiede tempo $\Theta(n)$ (bisogna controllare tutti gli elementi).
- **Insert:** Richiede tempo $O(1)$ (inserimento in coda o in testa).
- **Delete:** Richiede tempo $O(1)$ (rimozione dell'elemento con riferimento diretto).
- **DeleteMin:** Richiede tempo $\Theta(n)$ (bisogna prima trovare il minimo e poi rimuoverlo).

4. Lista ordinata

- **Lista bidirezionale ordinata:** La lista viene mantenuta ordinata in ordine crescente per priorità.
- **FindMin:** Richiede tempo $O(1)$ (il minimo è in testa alla lista).
- **Insert:** Richiede tempo $O(n)$ (bisogna trovare la posizione corretta nella lista ordinata).
- **Delete:** Richiede tempo $O(1)$ (aggiornamento dei puntatori dopo la rimozione).
- **DeleteMin:** Richiede tempo $O(1)$ (aggiornamento del puntatore alla testa).

Riepilogo implementazioni elementari

	FindMin	Insert	Delete	DeleteMin
Array non ord.	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$
Array ordinato	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Lista non ordinata	$\Theta(n)$	$O(1)$	$O(1)$	$\Theta(n)$
Lista ordinata	$O(1)$	$O(n)$	$O(1)$	$O(1)$

d-heap

Definizione

Un d-heap è un albero radicato d-ario con le seguenti proprietà:

1. Struttura:

- È completo almeno fino al penultimo livello.
- Tutte le foglie sull'ultimo livello sono compattate verso sinistra.

2. Contenuto informativo:

- Ogni nodo v contiene un elemento $\text{elem}(v)$ e una chiave $\text{chiave}(v)$ presa da un dominio totalmente ordinato.

3. Ordinamento parziale (inverso) dell'heap (minheap):

- Per ogni nodo v diverso dalla radice, vale $\text{chiave}(v) \geq \text{chiave}(\text{parent}(v))$.

Proprietà

1. Altezza:

- Un d-heap con n nodi ha altezza $\Theta(\log_d n)$.

2. Radice:

- La radice contiene l'elemento con chiave minima (per via della proprietà di ordinamento a heap).

3. Rappresentazione implicita:

- Un d-heap può essere rappresentato implicitamente tramite un vettore posizionale grazie alla proprietà di struttura.

Procedure ausiliarie

Le procedure ausiliarie servono per ripristinare la proprietà di ordinamento a heap su un nodo v che non la soddisfi.

Esistono due tipi principali di procedure ausiliarie:

1. Percolation up:

- Sposta un nodo v verso l'alto nell'albero finché la proprietà di ordinamento a heap non è soddisfatta.

2. Percolation down:

- Sposta un nodo v verso il basso nell'albero finché la proprietà di ordinamento a heap non è soddisfatta.

```
T(n)=O(logd n)
procedura muoviAlto(v)
    while ( v ≠ radice(T) and chiave(v) < chiave(padre(v)) ) do
        scambia di posto v e padre(v) in T
    il vecchio amico FixHeap!
procedura muoviBasso(v)
    repeat
        sia u il figlio di v con la minima chiave(u), se esiste
        if ( v non ha figli o chiave(v) ≤ chiave(u) ) break
        scambia di posto v e u in T
```

$$T(n) = O(1)$$

Implementazione

Un d-heap può essere implementato utilizzando diverse strutture dati, come array o array dinamici. La scelta della struttura dati dipende dalle esigenze specifiche dell'applicazione.

Operazioni di base

Le operazioni di base su un d-heap includono:

- **Insert: $T(n)=O(\log d n)$ per l'esecuzione di muoviAlto**

crea un nuovo nodo v con elemento e e chiave k , in modo che diventi una foglia sull'ultimo livello di T . La proprietà dell'ordinamento a heap viene poi ripristinata spingendo il nodo v verso l'alto tramite ripetuti scambi di nodi.

- **Delete & DeleteMin: $T(n)= O(\log d n)$ o $O(d \log d n)$ per l'esecuzione di muoviAlto o muoviBasso Può essere usata anche per implementare la cancellazione del minimo, con costo $O(d \log d n)$**

scambia il nodo v contenente l'elemento e con una qualunque foglia u sull'ultimo livello di T , e poi elimina v . Ripristina infine la proprietà dell'ordinamento a heap spingendo il nodo u verso la sua posizione corretta scambiandolo ripetutamente con il proprio padre o con il proprio figlio contenente la chiave più piccola

- **Findmin: $T(n) = O(1)$**

findMin() → elem
restituisce l'elemento nella radice di T .

- **Modifica della chiave:**

DecreaseKey $T(n)=O(\log d n)$ per l'esecuzione di muoviAlto

decrementa il valore della chiave nel nodo v contenente l'elemento e della quantità richiesta d . Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso l'alto tramite ripetuti scambi di nodi.

increaseKey $T(n)=O(d \log d n)$ per l'esecuzione di muoviBasso:

aumenta il valore della chiave nel nodo contenente l'elemento e della quantità richiesta d . Ripristina poi la proprietà dell'ordinamento a heap spingendo il nodo v verso il basso tramite ripetuti scambi di nodi.

Funzione merge (CodaPriorità c1, CodaPriorità c2)

Esistono due modi principali per combinare due code di priorità c1 e c2:

1. Costruzione da zero

Idea:

- Generare un nuovo heap d-ario contenente tutti gli elementi di c1 e c2.

Come:

- Generalizzare la procedura `heapify`.
- Rendere i d sottoalberi della radice heap ricorsivamente.
- Chiamare `muoviBasso` sulla radice.

Complessità (con d costante):

- $T(n) = d * T(n/d) + O(d * \log_d n)$
- $T(n) = \Theta(n)$

2. Inserimenti ripetuti

Idea:

- Inserire ad uno ad uno gli elementi della coda più piccola c1 nella coda più grande c2.

Complessità:

- $O(k * \log n)$, dove $n = |c1| + |c2|$

Vantaggi:

- Semplice da implementare.
- Efficiente per $k * \log n = o(n)$, ovvero per $k = o(n/\log n)$.

Osservazione: Nel caso peggiore, entrambe le operazioni hanno un costo di $\Omega(n)$.

Riepilogo

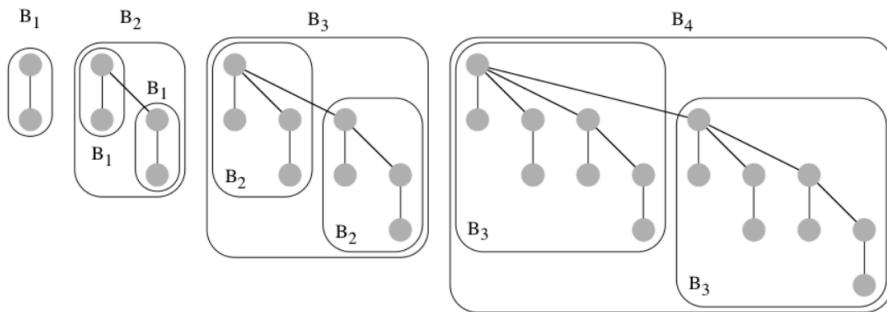
	Find Min	Insert	Delete	DelMin	Incr. Key	Decr. Key	merge
Array non ord.	$\Theta(n)$	O(1)	O(1)	$\Theta(n)$	O(1)	O(1)	O(n)
Array ordinato	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Lista non ordinata	$\Theta(n)$	O(1)	O(1)	$\Theta(n)$	O(1)	O(1)	O(1)
Lista ordinata	O(1)	O(n)	O(1)	O(1)	O(n)	O(n)	O(n)
d-Heap	O(1)	$O(\log_d n)$	$O(d \log_d n)$	$O(d \log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(n)$

Heap Binomiali

Definizione di albero binomiale

Un albero binomiale B_i è definito ricorsivamente come segue:

1. **B_0 consiste di un unico nodo.**
2. **Per ogni $i > 0$, B_{i+1} è ottenuto fondendo due alberi binomiali B_i ponendo la radice dell'uno come figlia della radice dell'altro.**



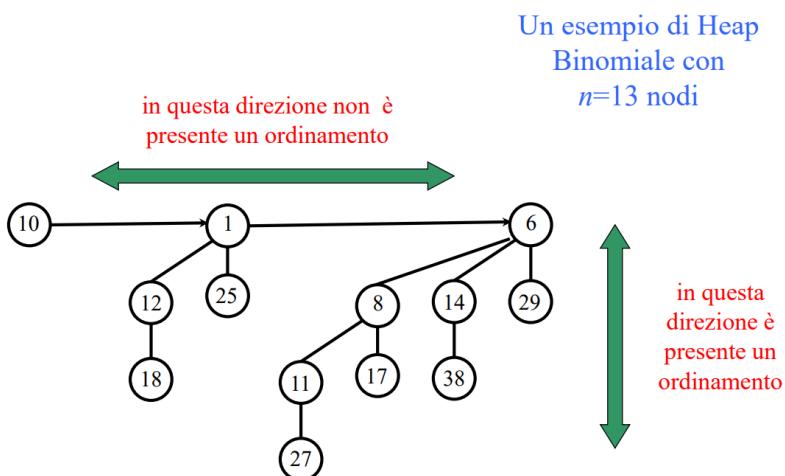
Un albero binomiale B_h gode delle seguenti proprietà:

1. Numero di nodi ($|B_h|$): $n = 2^h$.
2. Grado della radice: $D(n) = \log_2 n$
3. Altezza: $H(n) = h = \log_2 n$.
4. Figli della radice: i sottoalberi radicati nei figli della radice di B_h sono B_0, B_1, \dots, B_{h-1} .

Definizione di heap binomiale

Un heap binomiale è una foresta di alberi binomiali che soddisfa le seguenti proprietà:

1. **Unicità:** Per ogni intero $i \geq 0$, esiste al più un B_i nella foresta.
2. **Contenuto informativo:** Ogni nodo v contiene un elemento $\text{elem}(v)$ e una chiave $\text{chiave}(v)$ presa da un dominio totalmente ordinato.
3. **Ordinamento a heap:** $\text{chiave}(v) \geq \text{chiave}(\text{parent}(v))$ per ogni nodo v diverso da una delle radici.



Proprietà topologiche

Dalla proprietà di unicità degli alberi binomiali che lo costituiscono, si può dedurre che:

- Un heap binomiale di n elementi è formato dagli alberi binomiali $B_{i_0}, B_{i_1}, \dots, B_{i_h}$, dove i_0, i_1, \dots, i_h corrispondono alle posizioni degli 1 nella rappresentazione in base 2 di n .
- In un heap binomiale con n nodi, vi sono al più $\lfloor \log n \rfloor$ alberi binomiali, ciascuno con grado ed altezza di $O(\log n)$.

Conseguenze

Le proprietà topologiche implicano che:

- Un heap binomiale con n nodi ha altezza $O(\log n)$.
- Le operazioni di ricerca, inserimento e rimozione in un heap binomiale hanno tempo di esecuzione medio di $O(\log n)$.

Vantaggi degli heap binomiali

- Efficienza delle operazioni fondamentali (ricerca, inserimento, rimozione) con tempo medio di esecuzione di $O(\log n)$.
- Struttura semplice e intuitiva.
- Facilità di implementazione rispetto ad altre strutture dati con proprietà simili.

Svantaggi degli heap binomiali

- Necessità di mantenere la proprietà di heap durante le operazioni di modifica.
- Possibili degradazioni delle prestazioni in caso di implementazioni non ottimali.

Procedura ausiliaria: Ristruttura

Utile per ripristinare la proprietà di unicità in un heap binomiale
(ipotizziamo di scorrere la lista delle radici da sinistra verso destra, in
ordine crescente rispetto all'indice degli alberi binomiali).

T(n): lineare nel numero di alberi binomiali in input

procedura ristruttura()

$i = 0$

while (esistono ancora due B_i) **do**

si fondono i due B_i per formare un albero B_{i+1} , ponendo la radice con
chiave più piccola come genitore della radice con chiave più grande

$i = i + 1$

Implementazione HeapBinomiale Pseudocodice:

classe HeapBinomiale **implementa** CodaPriorita:

dati:

una foresta H con n nodi, ciascuno contenente un
elemento di tipo $elem$ e una chiave di tipo $chiave$ presa da
un universo totalmente ordinato.

operazioni:

findMin() $\rightarrow elem$

scorre le radici in H e restituisce l'elemento a chiave
minima.

insert($elem e, chiave k$)

aggiunge ad H un nuovo B_0 con dati e e k . Ripristina poi
la proprietà di unicità in H mediante fusioni successive
dei doppioni B_i .

deleteMin()

trova l'albero T_h con radice a chiave minima. Togliendo
la radice a T_h , esso si spezza in h alberi binomiali
 T_0, \dots, T_{h-1} , che vengono aggiunti ad H . Ripristina poi
la proprietà di unicità in H mediante fusioni successive
dei doppioni B_i .

decreaseKey($elem e, chiave d$)

decrementa di d la chiave nel nodo v contenente l'elemento e . Ripristina poi la proprietà dell'ordinamento
a heap spingendo il nodo v verso l'alto tramite ripetuti
scambi di nodi.

delete($elem e$)

richiama **decreaseKey($e, -\infty$)** e poi **deleteMin()**.

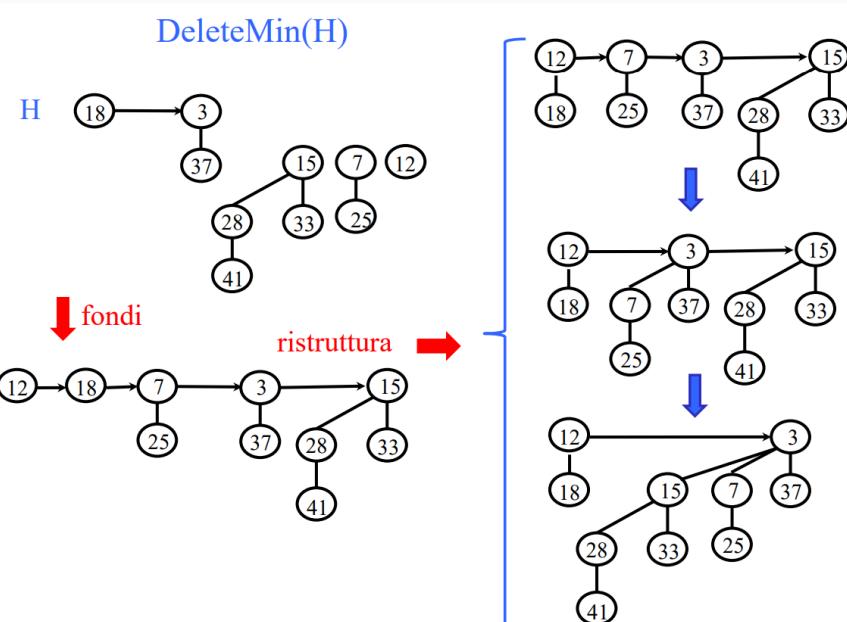
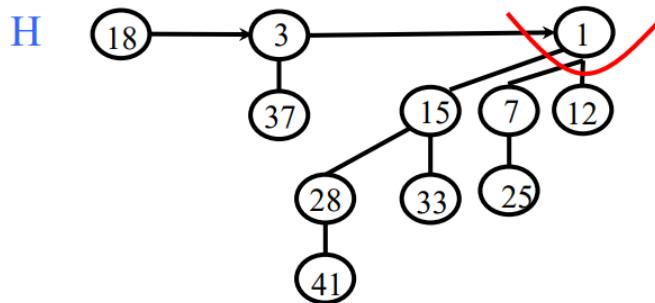
increaseKey(elem e , chiave d)
 richiama **delete**(e) e poi **insert**($elem, k + d$), dove
 k è la chiave associata all'elemento e .

merge(CodaPri. c_1 , CodaPri. c_2) \rightarrow CodaPri.
 unisce gli alberi in c_1 e c_2 in un nuovo heap binomiale c_3 .
 Ripristina poi la proprietà di unicità nell'heap binomiale
 c_3 mediante fusioni successive dei doppioni B_i .

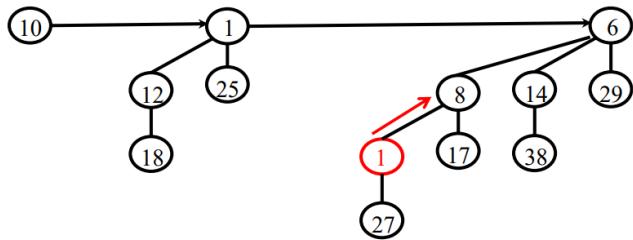
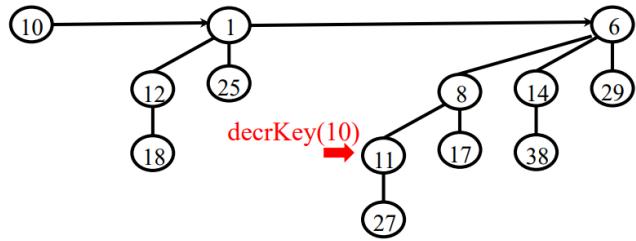
Tutte le operazioni richiedono tempo $T(n) = O(\log n)$

Esempio grafico:

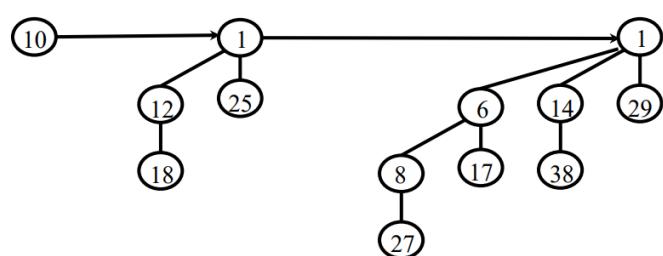
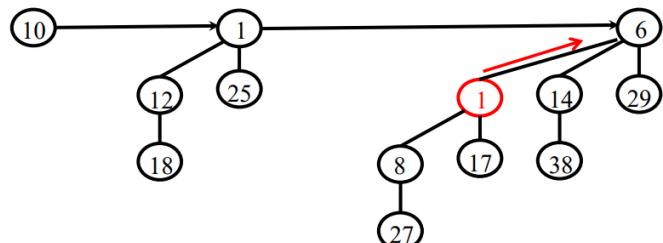
DeleteMin(H)

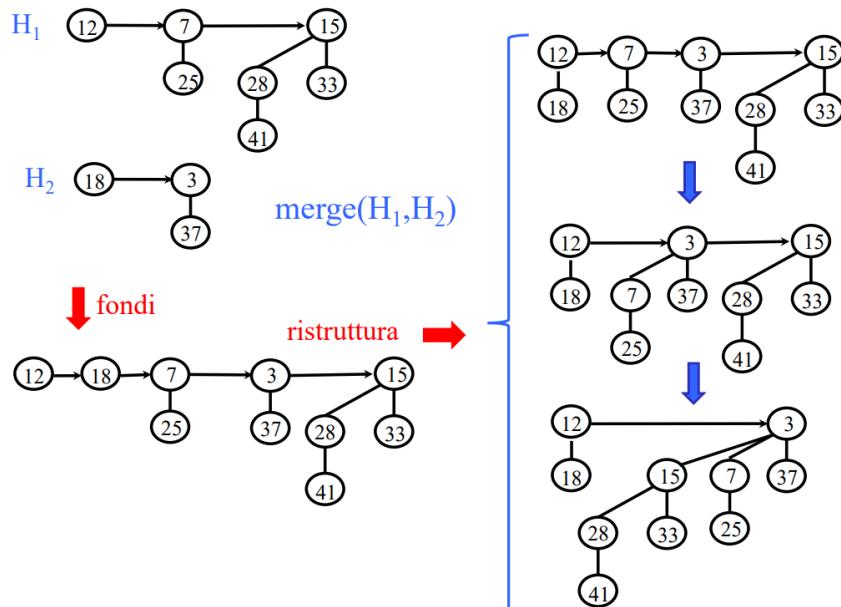


...decremento di una chiave



...decremento di una chiave





Heap di Fibonacci (Fredman, Tarjan, 1987)

Heap di Fibonacci come heap binomiale rilassato

Un heap di Fibonacci può essere visto come un heap binomiale "rilassato" che presenta due caratteristiche principali:

1. Rilassamento della proprietà di unicità
2. Atteggiamento "pigro" durante l'inserimento

Heap di Fibonacci come generalizzazione

Un heap di Fibonacci può essere visto anche come una generalizzazione degli heap binomiali, in cui la rigida struttura degli alberi binomiali viene ulteriormente allentata:

- I nodi non sono necessariamente disposti a formare alberi binomiali completi.
- Possono esserci nodi "pendenti" (con un solo figlio) che non appartengono ad alcun albero binomiale.

Analisi ammortizzata

L'analisi degli heap di Fibonacci si basa sul concetto di tempo di esecuzione ammortizzato:

- Si considera il costo complessivo di una sequenza di operazioni e lo si divide per il numero di operazioni stesse.
- In questo modo, si ottiene un costo medio per operazione che tiene conto di eventuali "ammortamenti" del costo in alcune operazioni per compensare il costo elevato in altre.

	FindMin	Insert	Delete	DelMin	IncKey	DecKey	merge
d-Heap (d cost.)	O(1)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
Heap Binom.	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Heap Fibon.	O(1)	O(1)	O(log n)*	O(log n)*	O(log n)*	O(1)*	O(1)

L'analisi per d-Heap e Heap Binomiali è nel caso peggiore, mentre quella per gli Heap di Fibonacci è ammortizzata (per le operazioni asteriscate)

Definizione di Grafo (Non Orientato)

Un grafo non orientato

$G=(V,E)$ è costituito da:

1. Insieme di vertici (o nodi):

- Rappresentato da
- V , è un insieme che contiene elementi distinti chiamati vertici o nodi.
- I vertici rappresentano le entità principali del grafo.

2. Insieme di archi:

- Rappresentato da E , è un insieme di coppie ordinate di vertici, chiamate archi.
- Gli archi collegano i vertici e rappresentano le relazioni tra di essi.
- A differenza dei grafi orientati, gli archi in un grafo non orientato non hanno una direzione specifica.

Esempio:

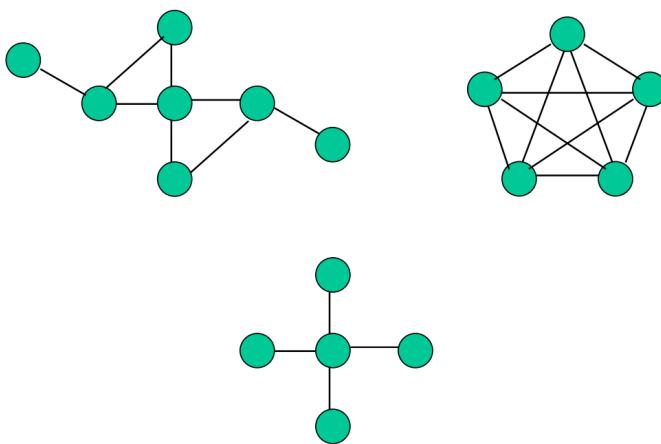
Consideriamo il grafo di Eulero associato alla città di Königsberg:

- $V = \{A, B, C, D\}$: l'insieme dei vertici rappresenta le quattro città di Königsberg (A, B, C, D).
- $E = \{(A, B), (A, B), (A, D), (B, C), (B, C), (B, D), (C, D)\}$: l'insieme degli archi rappresenta i sette ponti che collegano le città.

Nota:

Come correttamente indicato, il grafo di Königsberg descritto è un multigrafo, in quanto presenta archi paralleli tra alcuni vertici (ad esempio, due archi tra A e B). Un multigrafo è un tipo specifico di grafo non orientato che ammette archi multipli tra le stesse coppie di vertici.

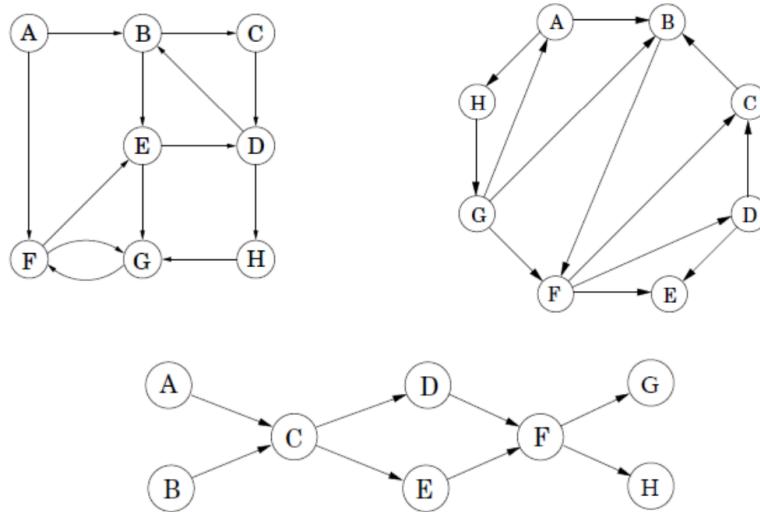
Un grafo semplice, invece, è un grafo non orientato che non contiene archi multipli.



Grafo diretto

Un grafo diretto $D=(V,A)$ consiste in:

- un insieme V di vertici (o nodi);
- un insieme A di coppie ordinate di vertici, detti archi diretti.



Terminologia per i Grafi Non Orientati

Un grafo non orientato

$G=(V,E)$ è caratterizzato da alcuni termini chiave:

1. Vertici (o nodi):

- Rappresentati da
- V , è un insieme che contiene elementi distinti chiamati vertici o nodi.
- Il numero di vertici è indicato da
- $n=|V|$.
- I vertici rappresentano le entità principali del grafo.

2. Archi:

- Rappresentati da
- E , è un insieme di coppie ordinate di vertici.
- Il numero di archi è indicato da
- $m=|E|$.
- Gli archi collegano i vertici e rappresentano le relazioni tra di essi.
- A differenza dei grafi orientati, gli archi non hanno una direzione specifica.

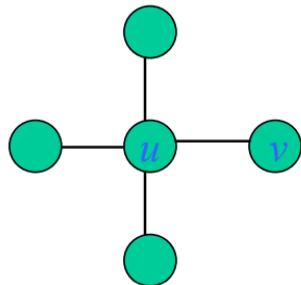
3. Adiacenza:

- Due vertici
- u e
- v sono detti adiacenti (o vicini) se esiste un arco che li collega.
- Un arco
- (u,v) è incidente a entrambi i vertici
- u e
- v , che vengono definiti estremi dell'arco.

4. Grado di un vertice:

- Il grado di un vertice
- u , indicato con
- $d(u)$, è il numero di archi incidenti a
- u .
- Il grado del grafo è il valore massimo del grado di tutti i vertici, ovvero:

$$\text{grado del grafo} = \max\{d(v) \mid v \in V\}$$



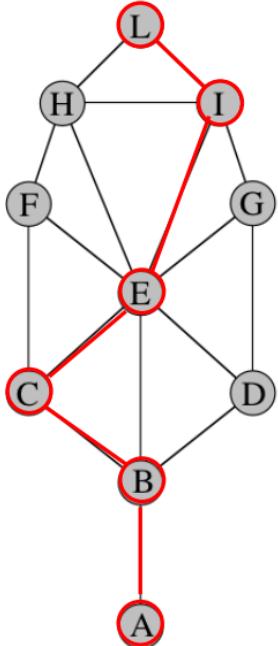
Somma dei gradi di ogni nodo:

$$\sum_{v \in V} \delta(v) = 2m$$

somma il grado uscente/entrante di tutti i nodi:

$$\sum_{v \in V} \delta_{\text{out}}(v) = \sum_{v \in V} \delta_{\text{in}}(v) = m$$

- cammino: sequenza di nodi connessi da archi .

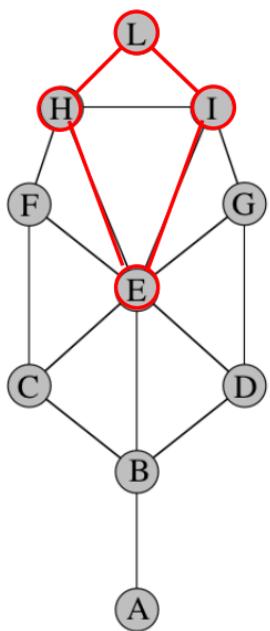


- lunghezza di un cammino: #archi del cammino .

- distanza: La lunghezza del più corto cammino tra due vertici si dice distanza tra i due vertici

distanza fra L e A: 4

in un grafo orientato, il cammino deve rispettare il verso di orientamento degli archi .



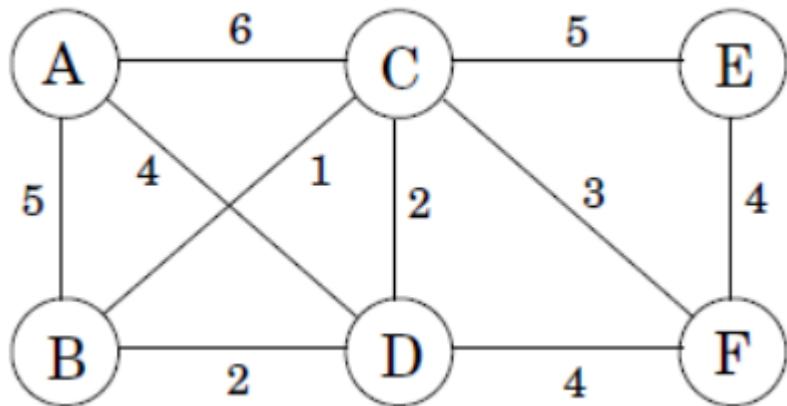
- G è connesso se esiste un cammino per ogni coppia di vertici.

- ciclo: un cammino chiuso, ovvero un cammino da un vertice a se stesso.

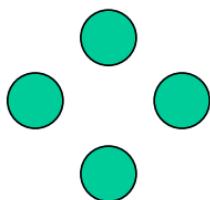
- il diametro è la massima distanza fra due nodi.

- $\max_{u,v \in V} \text{dist}(u,v)$ • il diametro di un grafo non connesso è ∞

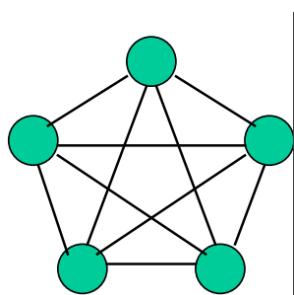
- **Grafo pesato:** è un grafo $G=(V,E,w)$ in cui ad ogni arco viene associato un valore definito dalla funzione peso w (definita su un opportuno insieme, di solito i reali).



Grafo totalmente sconnesso: è un grafo $G=(V,E)$ tale che $V \neq \emptyset$ ed $E = \emptyset$.

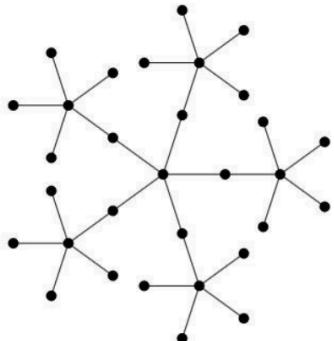


Grafo completo: per ogni coppia di nodi esiste un arco che li congiunge.
Il grafo completo con n vertici verrà indicato con K_n $m=|E|=n \cdot (n-1)/2$



N.B : un grafo (senza cappi o archi paralleli) può avere un numero di archi m compreso tra 0 e $n(n-1)/2 = \Theta(n^2)$.

Un albero è un grafo connesso ed aciclico.



Teorema

Sia $T=(V,E)$ un albero; allora $|E|=|V|-1$.

dim. (per induzione su $|V|$)

caso base: $|V|=1$ T $|E|=0=|V|-1$

caso induttivo: $|V|>1$

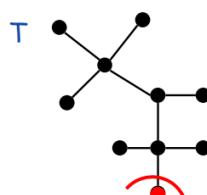
Sia n il numero di nodi di T

poiché T è connesso e aciclico ha almeno una foglia (nodo con grado 1)

se tutti i nodi avessero grado
almeno 2 ci sarebbe un ciclo
(riuscite a vedere perché?)

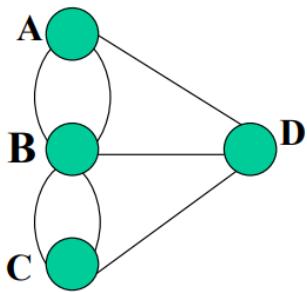
rimuovendo tale foglia si ottiene grafo
connesso e aciclico con $n-1$ nodi che per
ipotesi induttiva ha $n-2$ archi

→ T ha $n-1$ archi



Ciclo Euleriano: Dato un grafo G , un ciclo (rispettivamente un cammino) Euleriano è un ciclo (rispettivamente un cammino non chiuso) di G che passa per tutti gli archi di G una e una sola volta.

Teorema (di Eulero): Un grafo G ammette un ciclo Euleriano se e solo se tutti i nodi hanno grado pari. Inoltre, ammette un cammino Euleriano se e solo se tutti i nodi hanno grado pari tranne due (i due nodi di grado dispari sono gli estremi del cammino).



Soluzione al problema dei 7 ponti.

Matrice di adiacenza vs Liste di adiacenza

Matrice di adiacenza:

- Rappresenta un grafo con una matrice
- $n \times n$, dove
- n è il numero di vertici.
- Ogni cella della matrice contiene un 1 se esiste un arco tra i vertici corrispondenti, e uno 0 altrimenti.

Vantaggi:

- Semplice da implementare.
- Facile da controllare se esiste un arco tra due vertici (tempo costante: $O(1)$).

Svantaggi:

- Richiede uno spazio di memoria di
- $O(n^2)$ che può essere elevato per grafi con molti vertici.
- Inefficiente per grafi sparzi (con pochi archi rispetto al numero di vertici).

Liste di adiacenza:

- Rappresenta un grafo con un array di liste, dove ogni lista contiene i vertici adiacenti a un vertice specifico.

Vantaggi:

- Richiede uno spazio di memoria di $O(m+n)$, dove m è il numero di archi.

Svantaggi:

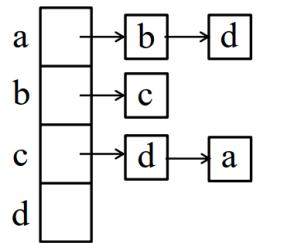
- Implementazione leggermente più complessa rispetto alla matrice di adiacenza.
- Richiede tempo di ricerca lineare ($O(\delta(v))$) per trovare un arco specifico in un vertice con grado elevato ($\delta(v)$ indica il numero di archi incidenti al vertice).

Grafi diretti:

	a	b	c	d
a	0	1	0	1
b	0	0	1	0
c	1	0	0	1
d	0	0	0	0

Matrice di adiacenza

$O(n^2)$



liste di adiacenza

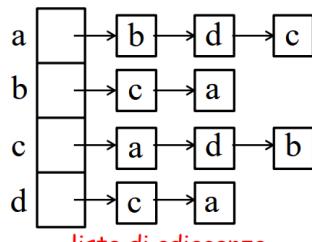
$O(m + n)$

Grafi non diretti:

	a	b	c	d
a	0	1	1	1
b	1	0	1	0
c	1	1	0	1
d	1	0	1	0

Matrice di adiacenza

$O(n^2)$



liste di adiacenza

$O(m + n)$

Scopo e tipi di visita di un grafo

Introduzione

Una visita di un grafo

G permette di esaminare i nodi e gli archi di G in modo sistematico, ovvero in un ordine ben definito. Questo processo è fondamentale per diverse applicazioni, come la ricerca di percorsi, l'identificazione di componenti connesse e la valutazione di proprietà del grafo.

Durante una visita, si genera un albero di visita, che rappresenta l'ordine in cui i nodi sono stati esplorati. Esistono diversi algoritmi per eseguire una visita di un grafo, ognuno con caratteristiche e proprietà specifiche.

Tipi di visita

Esistono due tipi principali di visita di un grafo:

1. Visita in ampiezza (BFS - Breadth First Search):

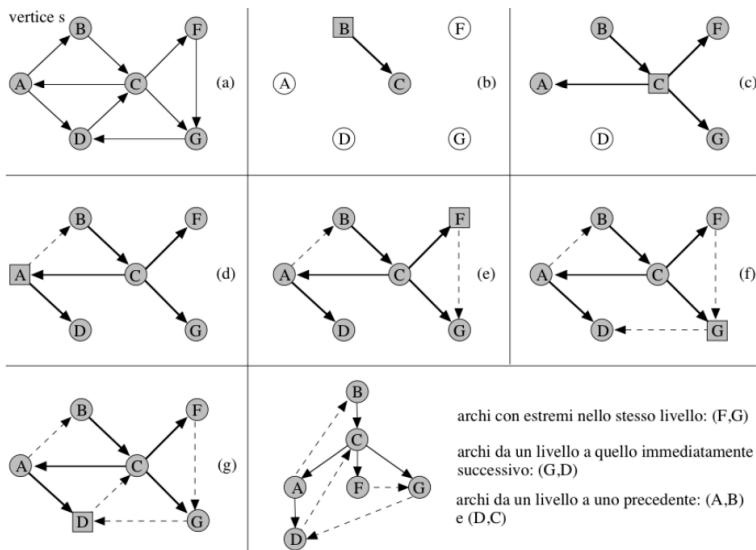
- L'algoritmo esplora il grafo livello per livello, partendo dal nodo sorgente
- *s* e visitando tutti i suoi nodi adiacenti prima di procedere al livello successivo.
- I nodi vengono visitati in ordine di distanza dal nodo sorgente.
- La visita in ampiezza garantisce che il primo percorso trovato tra il nodo sorgente e qualsiasi altro nodo sia il cammino minimo.

algoritmo visitaBFS(*vertice s*) → *albero*

1. rendi tutti i vertici non marcati
 2. $T \leftarrow$ albero formato da un solo nodo *s*
 3. Coda *F*
 4. marca il vertice *s*
 5. *F.enqueue(s)*
 6. **while (not *F.isempty()*) do**
 7. $u \leftarrow F.dequeue()$
 8. **for each (arco (u, v) in *G*) do**
 9. **if (*v* non è ancora marcato) then**
 10. *F.enqueue(v)*
 11. marca il vertice *v*
 12. rendi *u* padre di *v* in *T*
 13. **return T**
- Costo: Il tempo di esecuzione dipende dalla struttura dati usata per rappresentare il grafo (e dalla connettività o meno del grafo rispetto ad *s*):
 - Liste di adiacenza: $O(m+n)$
 - Matrice di adiacenza: $O(n^2)$

Teorema

Per ogni nodo *v*, il livello di *v* nell'albero BFS è pari alla distanza di *v* dalla sorgente *s* (sia per grafi orientati che non orientati)



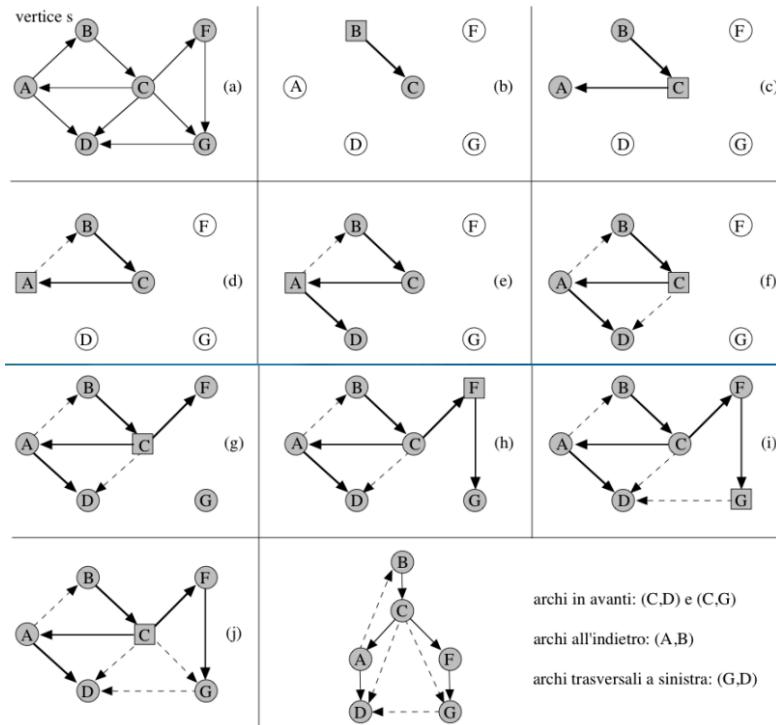
2. Visita in profondità (DFS - Depth First Search):

- L'algoritmo esplora il grafo in modo ricorsivo, seguendo un percorso il più possibile profondo fino a quando non incontra un nodo foglia (un nodo senza nodi adiacenti non ancora visitati).
- I nodi vengono visitati in ordine di profondità nell'albero di visita.
- La visita in profondità è utile per trovare cicli e per esplorare l'intera struttura del grafo.

```
procedura visitaDFSRicorsiva(vertice v, albero T)
1.   marca e visita il vertice v
2.   for each ( arco  $(v, w)$  ) do
3.     if ( w non è marcato ) then
4.       aggiungi l'arco  $(v, w)$  all'albero T
5.       visitaDFSRicorsiva(w, T)
```

```
algoritmo visitaDFS(vertice s)  $\rightarrow$  albero
6.   T  $\leftarrow$  albero vuoto
7.   visitaDFSRicorsiva(s, T)
8.   return T
```

- **Costo della visita in profondità**
Il tempo di esecuzione dipende dalla struttura dati usata per rappresentare il grafo (e dalla connettività o meno del grafo rispetto ad s):
 - Liste di adiacenza: $O(m+n)$
 - Matrice di adiacenza: $O(n^2)$



- Proprietà dell'albero DFS radicato in s :
 - Se il grafo è non orientato, per ogni arco (u,v) si ha:
 - (u,v) è un arco dell'albero DFS, oppure
 - i nodi u e v sono l'uno discendente/antenato dell'altro
 - Se il grafo è orientato, per ogni arco (u,v) si ha:
 - (u,v) è un arco dell'albero DFS, oppure
 - i nodi u e v sono l'uno discendente/antenato dell'altro, oppure
 - (u,v) è un arco trasversale a sinistra, ovvero il vertice v è in un sottoalbero visitato precedentemente ad u .

DFS (quando non tutti i nodi sono raggiungibili dal punto di partenza):

VisitaDFS (grafo G)

1. **for each** nodo v **do** imposta v come *non marcato*
2. $clock=1$
3. $F \leftarrow$ foresta vuota
4. **for each** nodo v **do**
5. **if** (v è *non marcato*) **then**
6. $T \leftarrow$ albero vuoto
7. visitaDSFRicorsiva(v, T)
8. aggiungi T ad F
9. **return** F

proprietà:

- per ogni coppia di nodi u e v , gli intervalli $[pre(u),post(u)]$ e $[pre(v),post(v)]$ o sono disgiunti o l'uno è contenuto nell'altro.
- u è antenato di v nell'albero DFS, se $pre(u) < pre(v) < post(v) < post(u)$ condizione che rappresentiamo così:

pre/post per l'arco (u,v) tipo di arco

$\begin{bmatrix} & & & \end{bmatrix}$	in avanti			
$\begin{bmatrix} & & & \end{bmatrix}$	all'indietro			
$\begin{bmatrix} & & & \end{bmatrix}$	trasversali			

Riconoscere la presenza di un ciclo in un grafo diretto

Algoritmo DFS e archi all'indietro

Per determinare se un grafo diretto

G contiene un ciclo, è possibile utilizzare l'algoritmo di visita DFS (Depth-First Search) e controllare la presenza di archi all'indietro.

Proprietà:

- Un grafo diretto
- G ha un ciclo se e solo se la visita DFS rivela un arco all'indietro.

Dimostrazione:

1. \Rightarrow (se): Se c'è un arco all'indietro, per definizione, un ciclo esiste.

2. \Leftarrow (solo se): Se c'è un ciclo, sia v_i il primo nodo scoperto nella visita DFS che appartiene al ciclo.

- Poiché v_{i-1} è raggiungibile da v_i , la visita DFS di v_{i-1} deve avvenire prima di terminare la visita di v_i . Di conseguenza, l'arco (v_{i-1}, v_i) è un arco all'indietro, rilevato durante la visita DFS.

Definizioni

1. **Grafo diretto aciclico (DAG):** Un grafo diretto G è definito aciclico se non contiene cicli diretti.

2. **Ordinamento topologico:** Un ordinamento topologico di un grafo diretto

$G=(V,E)$ è una funzione biunivoca

$\sigma: V \rightarrow \{1, 2, \dots, n\}$ tale che per ogni arco $(u, v) \in E$, $\sigma(u) < \sigma(v)$.

3. **Pozzo e sorgente:**

- Un pozzo è un nodo in un grafo diretto che ha solo archi entranti.
- Una sorgente è un nodo in un grafo diretto che ha solo archi uscenti.

4. **Teorema:**

Un grafo diretto G ammette un ordinamento topologico se e solo se G è un DAG.

dim
 \Rightarrow

per assurdo: sia σ un ordinamento topologico di G

e sia $\langle v_0, v_1, \dots, v_k = v_0 \rangle$ un ciclo

allora $\sigma(v_0) < \sigma(v_1) < \dots < \sigma(v_{k-1}) < \sigma(v_k) = \sigma(v_0)$

\Leftarrow : ...adesso diamo un algoritmo costruttivo.

Relazione tra cicli e ordinamento topologico

L'esistenza di un ciclo in un grafo diretto impedisce la costruzione di un ordinamento topologico valido. Infatti, se un ciclo è presente, non è possibile ordinare i vertici in modo tale che ogni arco sia diretto verso un vertice con indice maggiore.

Viceversa, se un grafo ammette un ordinamento topologico, è garantito che il grafo sia aciclico. L'ordinamento topologico fornisce un ordine in cui i vertici possono essere visitati senza dover tornare indietro su archi già percorsi, il che implicherebbe l'esistenza di un ciclo.

calcolare ordinamento topologico :

Algoritmo: fai una visita DFS e restituisci i nodi in ordine decrescente rispetto ai tempi di fine visita $\text{post}(v)$.

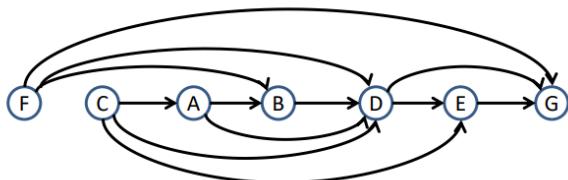
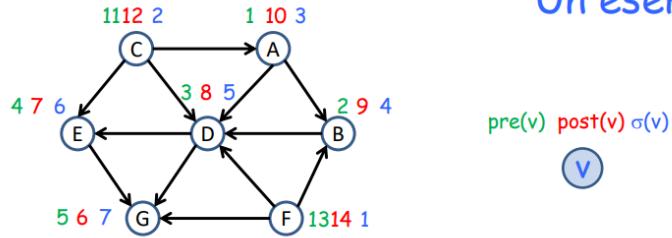
Complessità temporale: se G è rappresentato con liste di adiacenza $\Theta(n+m)$.

Pseudocodice:

OrdinamentoTopologico (grafo G)

1. $\text{top}=n; L \leftarrow$ lista vuota;
2. chiama visita DFS ma:
 1. quando hai finito di visitare un nodo v (quando imposta $\text{post}(v)$):
 2. $\sigma(v)=\text{top}; \text{top}=\text{top}-1;$
 3. aggiungi v in testa alla lista L
3. **return** L e σ

Un esempio



correttezza:

per ogni coppia di nodi u e v , gli intervalli $[\text{pre}(u), \text{post}(u)]$ e $[\text{pre}(v), \text{post}(v)]$ o sono disgiunti o l'uno è contenuto nell'altro:

pre/post per l'arco (u,v) tipo di arco

$[$	$[$	$]$	$]$	in avanti
u	v	v	u	

all'indietro

$[$	$]$	$[$	$]$	trasversali
v	v	u	u	

non ci possono essere archi all'indietro

Un algoritmo alternativo:

algoritmo ordinamentoTopologico(*grafo G*) \rightarrow *lista*

$\widehat{G} \leftarrow G$

ord \leftarrow lista vuota di vertici

while (esiste un vertice *u* senza archi entranti in \widehat{G}) **do**

 appendi *u* come ultimo elemento di *ord*

 rimuovi da \widehat{G} il vertice *u* e tutti i suoi archi uscenti

(*) **if** (\widehat{G} non è diventato vuoto) **then errore** il grafo *G* non è aciclico
return *ord*

(*) perché altrimenti in \widehat{G} ogni vertice deve avere almeno un arco entrante, e quindi posso trovare un ciclo percorrendo archi entranti a ritroso, e quindi *G* non può essere aciclico)

Tempo di esecuzione (con liste di adiacenza): $\Theta(n+m)$

Componenti Fortemente Connesse in un Grafo Diretto

Definizione

Una componente fortemente connessa di un grafo diretto $G=(V,E)$ è un insieme massimale di vertici $C \subseteq V$ tale che per ogni coppia di vertici u e v in C , esiste un percorso diretto da u a v e viceversa.

- **Massimale:** Se si aggiunge un qualsiasi vertice a C , la proprietà non è più vera.

Proprietà del grafo delle componenti

Il grafo delle componenti fortemente connesse di G è sempre un DAG (Directed Acyclic Graph), ovvero un grafo diretto aciclico.

Algoritmo per calcolare le componenti

Esistono diversi algoritmi per calcolare le componenti fortemente connesse di un grafo diretto. Uno dei più efficienti è basato sulla visita DFS (Depth-First Search) ricorsiva.

Idea dell'algoritmo:

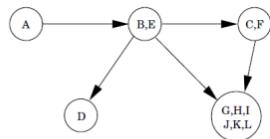
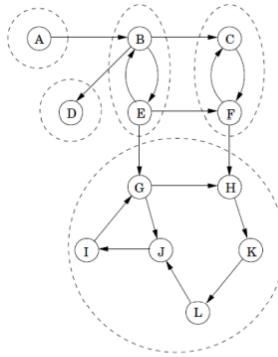
1. Eseguire una visita DFS a partire da un nodo u .
2. "Eliminare" la componente trovata (i vertici visitati) dal grafo originale.
3. Ripetere i passi 1 e 2 da un nodo non ancora visitato finché tutti i nodi sono stati visitati.

Proprietà utili:

Proprietà 1:

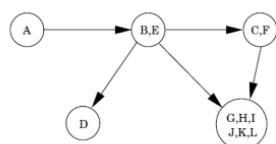
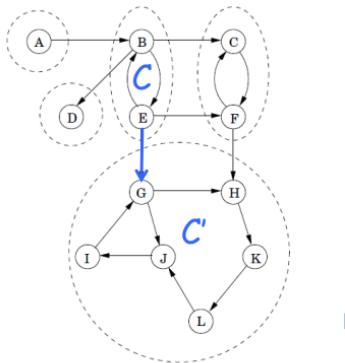
- Se si esegue la procedura di visita DFS ricorsiva a partire da un nodo u , la procedura termina dopo che tutti i nodi raggiungibili da u sono stati

visitati.



Proprietà 2:

- Se C e C' sono due componenti distinte e c'è un arco da un nodo in C verso un nodo in C' , allora il valore di post più alto in C è maggiore del valore di post più alto in C' .



Dimostrazione:

- Se la visita DFS visita prima C' di C , il risultato è banale.

- Se la visita DFS visita prima C , allora si ferma dopo aver raggiunto tutti i nodi di C e C' e termina su un nodo di C . In questo caso, il valore di post per i nodi di C sarà maggiore di quello per i nodi di C' .

Proprietà 3:

- Il nodo che riceve dalla visita DFS il valore di post più grande appartiene a una componente sorgente.

Dimostrazione:

- Invertiamo gli archi del grafo. Il nodo con il valore di post più grande nella visita DFS del grafo invertito sarà una sorgente nel grafo originale.

VisitaDFS (grafo G)

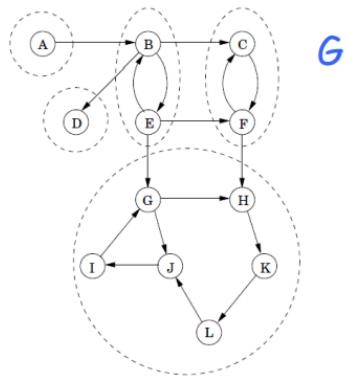
1. calcola G^R
2. esegui $\text{DFS}(G^R)$ per trovare valori $\text{post}(v)$
3. **return** CompConnesse(G)

Complessità temporale:
se G è rappresentato
con liste di adiacenza
 $\Theta(n+m)$

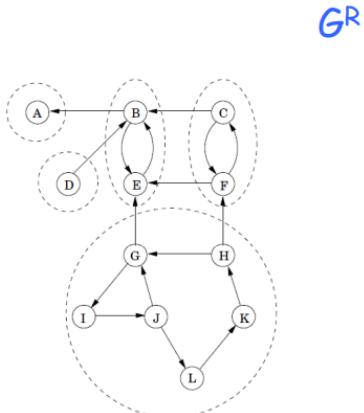
CompConnesse (grafo G)

1. **for each** nodo v **do** imposta v come *non marcato*
2. $Comp \leftarrow \emptyset$
3. **for each** nodo v in ordine decrescente di $\text{post}(v)$ **do**
4. **if** (v è *non marcato*) **then**
5. $T \leftarrow$ albero vuoto
6. visitaDSFRicorsiva(v, T)
7. aggiungi T a $Comp$
8. **return** $Comp$

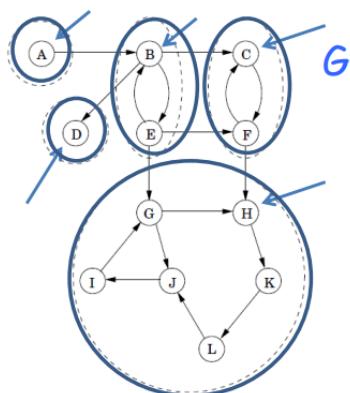
Esempio:



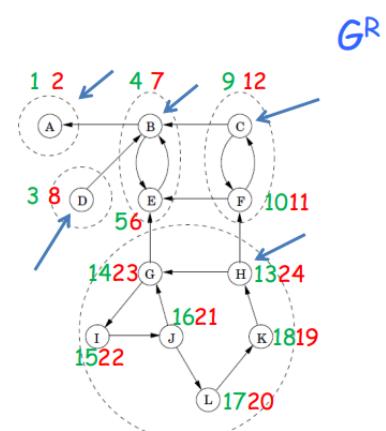
G



G^R



G



G^R

Cammini minimi in grafi pesati

Definizione

Un grafo $G=(V,E,w)$ è costituito da:

Un insieme V di vertici (o nodi).

Un insieme E di archi.

Una funzione peso w che assegna un peso reale $w(u,v)$ a ciascun arco $(u,v) \in E$.

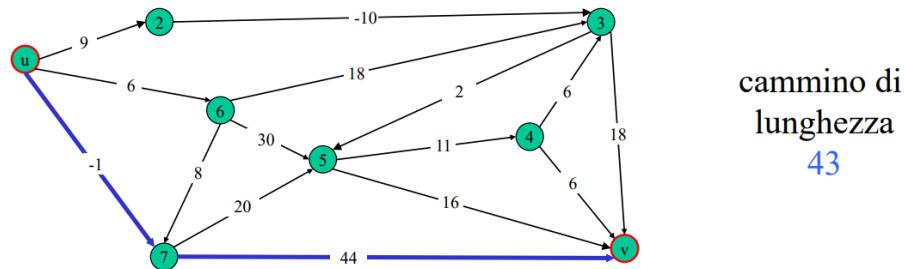
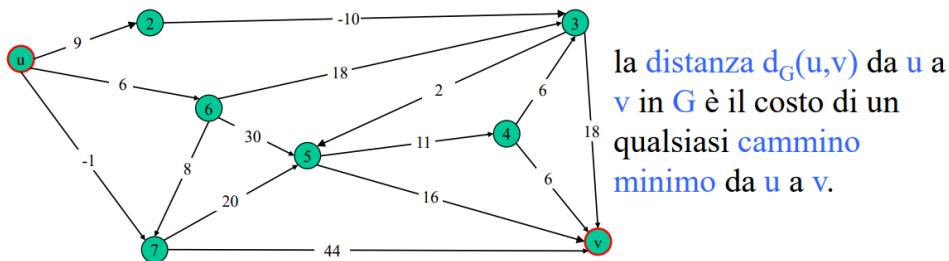
Il costo o lunghezza di un cammino $\pi=(v_0, v_1, \dots, v_k)$ in G è definito come la somma dei pesi degli archi che compongono il cammino:

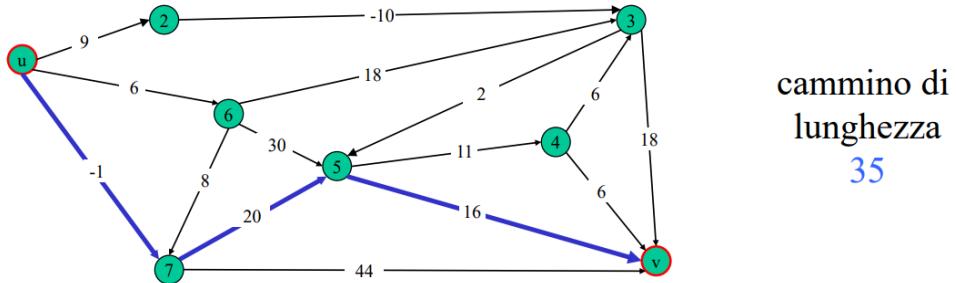
$$w(\pi) = \sum_{i=1}^k w(v_{i-1}, v_i)$$

Un cammino minimo tra una coppia di vertici x e y è un cammino che collega x e y e ha un costo minore o uguale a quello di qualsiasi altro cammino tra gli stessi vertici.

Nota: Un cammino minimo non è necessariamente unico. Possono esistere diversi cammini con il costo minimo tra x e y .

Esempio Cammino minimo grafo non pesato:





- se non esiste nessun cammino da u a v – $d(u,v)=+\infty$
- se c'è un cammino che contiene un ciclo (raggiungibile) il cui costo è negativo – $d(u,v) = -\infty$

Oss: se G non contiene cicli negativi, esistono cammini minimi che sono cammini semplici (non contiene nodi ripetuti).

Ogni sottocammino di un cammino minimo è un cammino minimo.

disuguaglianza triangolare: per ogni $u, v, x \in V$, vale: $d(u,v) \leq d(u,x) + d(x,v)$.

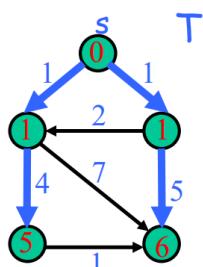
Problema del calcolo dei cammini minimi a singola sorgente:

- 1) Dato $G=(V,E,w)$, $s \in V$, calcola le distanze di tutti i nodi da s , ovvero, $d_G(s,v)$ per ogni $v \in V$
- 2) Dato $G=(V,E,w)$, $s \in V$, calcola l'albero dei cammini minimi di G radicato in s .

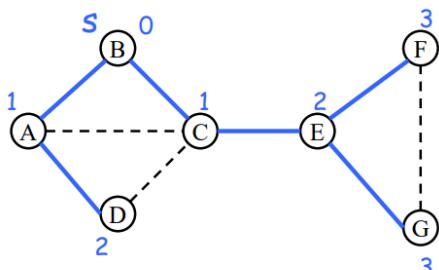
Albero dei cammini minimi (o Shortest Path Tree - SPT)

T è un albero dei cammini minimi con sorgente s di un grafo $G=(V,E,w)$ se:

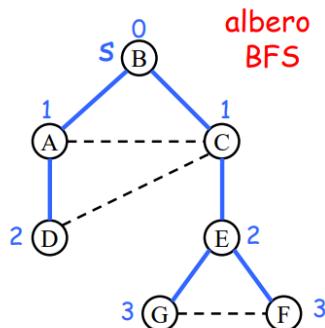
- T è un albero radicato in s
- per ogni $v \in V$, vale:
 $d_T(s,v) = d_G(s,v)$



Albero dei cammini minimi (o Shortest Path Tree - SPT)



per grafi non pesati:
SPT radicato in s
=
Albero BFS radicato in s



Algoritmo di Dijkstra per trovare il cammino minimo in un grafo

Assunzioni e idea intuitiva

L'algoritmo di Dijkstra si basa su due assunzioni:

1. Tutti gli archi hanno peso non negativo: Il peso di ogni arco (u,v) nel grafo è uguale o superiore a 0, ovvero $w(u,v) \geq 0$.
2. Esiste un cammino tra ogni coppia di vertici: Il grafo è connesso, ovvero per ogni coppia di vertici u e v esiste un cammino che li collega.

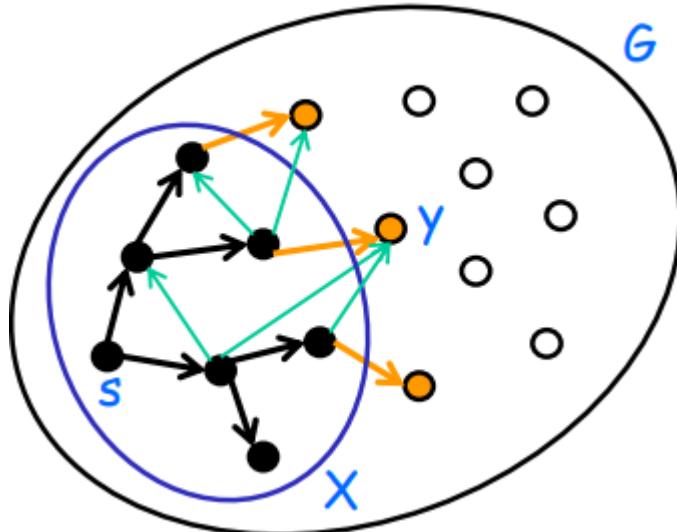
L'idea intuitiva dell'algoritmo può essere immaginata come "pompare acqua" nella sorgente:

- Immaginiamo gli archi come tubi e i pesi degli archi come la lunghezza dei tubi.
- L'acqua scorre a velocità costante attraverso i tubi.

L'obiettivo è determinare la quantità di acqua che raggiunge ogni nodo del grafo e il percorso che l'acqua ha seguito per arrivarci.

Implementazione: approccio greedy

L'algoritmo di Dijkstra utilizza un approccio greedy (goloso) per trovare il cammino minimo:



1. Inizializzazione:

- Viene mantenuta per ogni nodo v una stima (per eccesso) $D(v)$ della distanza $d(s,v)$ dalla sorgente s . Inizialmente, l'unica stima finita è $D(s)=0$.
- Viene mantenuto un insieme X di nodi per cui le stime sono esatte e un albero T dei cammini minimi verso i nodi in X (albero "nero"). Inizialmente, $X=s$ e T non ha archi.

2. Iterazione:

- Ad ogni passo, l'algoritmo aggiunge a X il nodo u in $V-X$ con la stima minima. Viene aggiunto a T uno specifico arco (arancione) entrante in u .
- Le stime per i nodi adiacenti a u vengono aggiornate.

3. Aggiornamento delle stime:

- I nodi da aggiungere progressivamente a X (e quindi a T) sono mantenuti in una coda di priorità, associati a un unico arco (arco arancione) che li connette a T .
- La stima per un nodo y in $V-X$ è:

$$D(y) = \min\{D(x) + w(x, y) : (x, y) \in E, x \in X\}$$

- L'arco che fornisce il minimo è l'arco arancione.
- Se y è in coda con l'arco (x,y) associato e se dopo aver aggiunto u a T si trova un arco (u,y) tale che $D(u)+w(u,y) < D(x)+w(x,y)$, allora l'arco (x,y) viene rimpiazzato con (u,y) e la stima $D(y)$ viene aggiornata.

4. Terminazione:

- L'algoritmo termina quando tutti i nodi sono stati aggiunti a X .

Proprietà e complessità

L'algoritmo di Dijkstra ha le seguenti proprietà:

- Trova il cammino minimo tra la sorgente
- s e qualsiasi altro nodo del grafo.
- La complessità computazionale è di
- $O(V \log V)$ per grafi con coda di priorità basata su heap, dove
- V è il numero di vertici.

Applicazioni

L'algoritmo di Dijkstra ha diverse applicazioni, tra cui:

- Reti di trasporto: trovare il percorso più breve tra due città.
- Reti di computer: trovare il percorso più efficiente per trasmettere dati.
- Problemi di pianificazione: trovare la sequenza di attività da eseguire per minimizzare il tempo o il costo.

Pseudocodice:

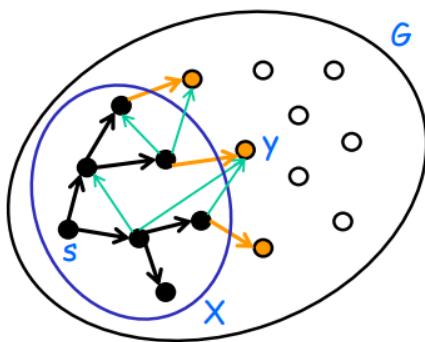
```

algoritmo Dijkstra(grafo G, vertice s) → albero
  for each (vertice u in G) do  $D_{su} \leftarrow +\infty$ 
   $\hat{T} \leftarrow$  albero formato dal solo nodo s;  $X \leftarrow \emptyset$ 
  CodaPriorita S
   $D_{ss} \leftarrow 0$ 
  S.insert(s, 0)
  while (not S.isEmpty()) do
    u ← S.deleteMin();  $X \leftarrow X \cup \{u\}$ 
    for each (arco (u, v) in G) do
      if ( $D_{sv} = +\infty$ ) then
        S.insert(v, Dsu + w(u, v))
         $D_{sv} \leftarrow D_{su} + w(u, v)$ 
        rendi u padre di v in  $\hat{T}$ 
      else if ( $D_{su} + w(u, v) < D_{sv}$ ) then
        S.decreaseKey(v, Dsv - Dsu - w(u, v))
         $D_{sv} \leftarrow D_{su} + w(u, v)$ 
        rendi u nuovo padre di v in  $\hat{T}$ 
  return  $\hat{T}$ 

```

Costo(Escludendo la coda priorità): tempo O(m+n)

Nota: T^* è un albero che contiene tutti i nodi in *X* più i nodi correntemente contenuti nella coda di priorità (nodi arancioni); è composto cioè dagli archi di *T* (albero dei cammini minimi ristretto ai nodi in *X*) più gli archi arancioni (potenziali archi da aggiungere a *T*).



- nodi per i quali non è stato "scoperto" nessun cammino; $\text{stima} = +\infty$
- nodi "scoperti"; hanno $\text{stima} < +\infty$ sono mantenuti in una coda con priorità insieme al "miglior" arco entrante (arancione)

Tempo di esecuzione: implementazioni elementari

Supponendo che il grafo G sia rappresentato tramite liste di adiacenza, e supponendo che tutti i nodi siano connessi ad s , avremo n insert, n deleteMin e al più m decreaseKey nella coda di priorità, al costo di:

	Insert	DelMin	DecKey
Array non ord.	$O(1)$	$O(n)$	$O(1)$
Array ordinato	$O(n)$	$O(1)$	$O(n)$
Lista non ord.	$O(1)$	$O(n)$	$O(1)$
Lista ordinata	$O(n)$	$O(1)$	$O(n)$

- $n \cdot O(1) + n \cdot O(n) + O(m) \cdot O(1) = O(n^2)$ con array non ordinati
- $n \cdot O(n) + n \cdot O(1) + O(m) \cdot O(n) = O(m \cdot n)$ con array ordinati
- $n \cdot O(1) + n \cdot O(n) + O(m) \cdot O(1) = O(n^2)$ con liste non ordinate
- $n \cdot O(n) + n \cdot O(1) + O(m) \cdot O(n) = O(m \cdot n)$ con liste ordinate

Tempo di esecuzione: implementazioni efficienti

Supponendo che il grafo G sia rappresentato tramite liste di adiacenza, e supponendo che tutti i nodi siano connessi ad s , avremo n insert, n deleteMin e al più m decreaseKey nella coda di priorità, al costo di:

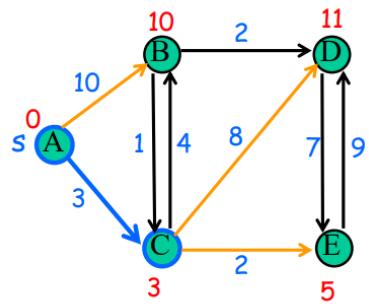
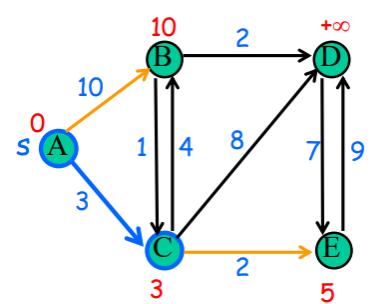
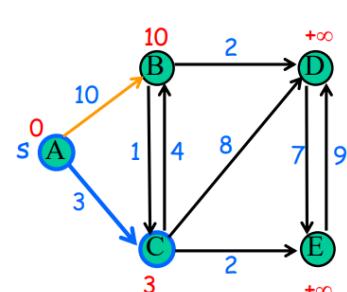
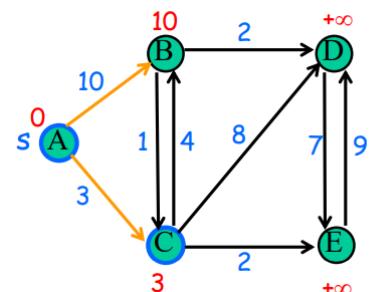
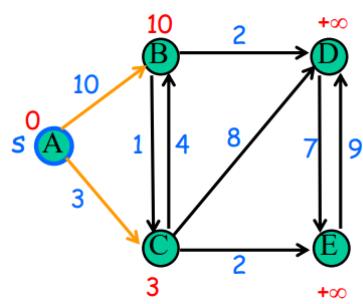
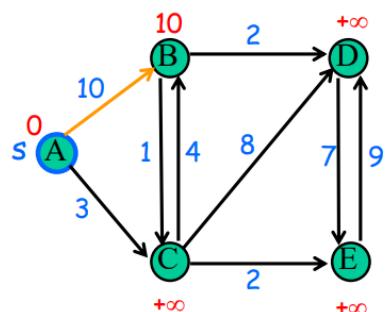
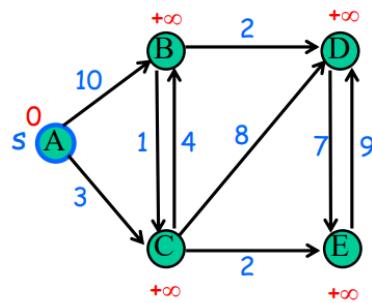
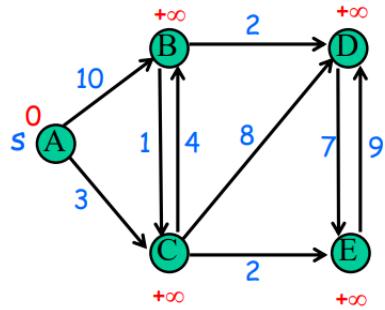
	Insert	DelMin	DecKey
Heap binario	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap Binom.	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap Fibon.	$O(1)$	$O(\log n)^*$ (ammortizzata)	$O(1)^*$ (ammortizzata)

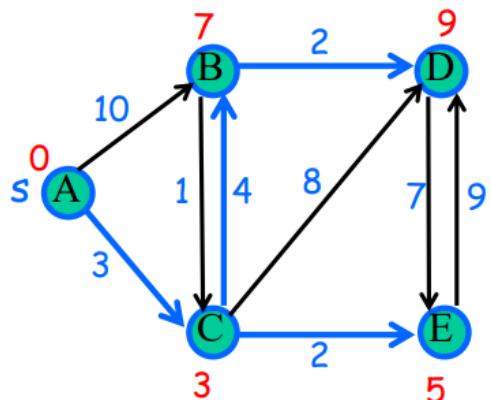
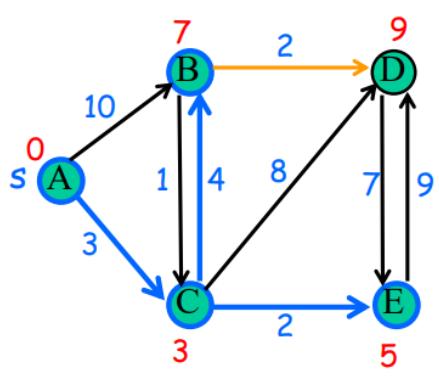
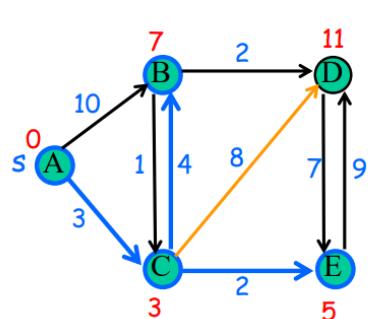
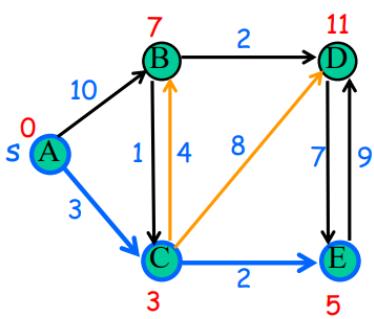
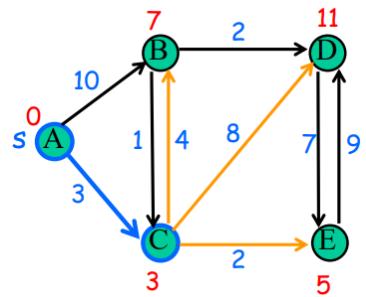
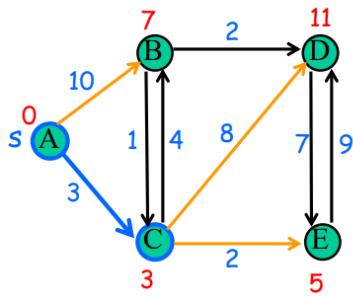
- $n \cdot O(\log n) + n \cdot O(\log n) + O(m) \cdot O(\log n) = O(m \cdot \log n)$ utilizzando heap binari o binomiali
- $n \cdot O(1) + n \cdot O(\log n)^* + O(m) \cdot O(1)^* = O(m + n \cdot \log n)$ utilizzando heap di Fibonacci

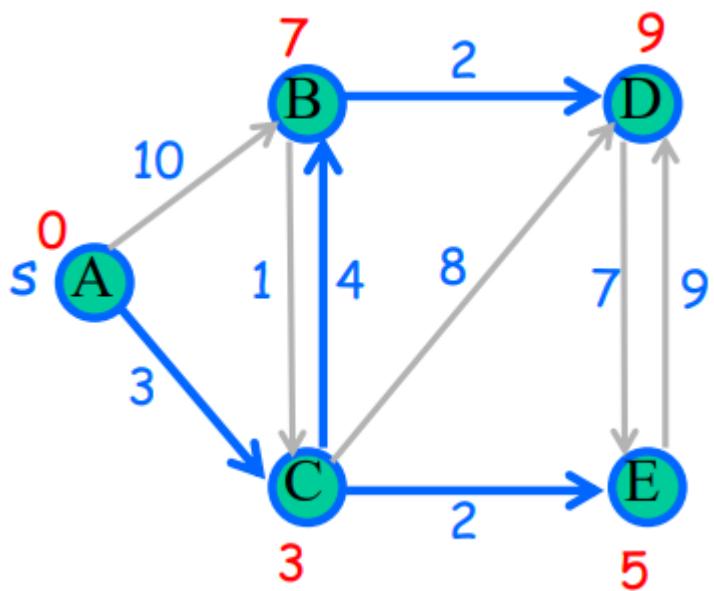
soluzione migliore: mai peggiore,
a volte meglio delle altre

Costo migliore per le code di priorità : tempo $O(m+n \log n)$

Esempio Dijkstra:







CORRETTEZZA:

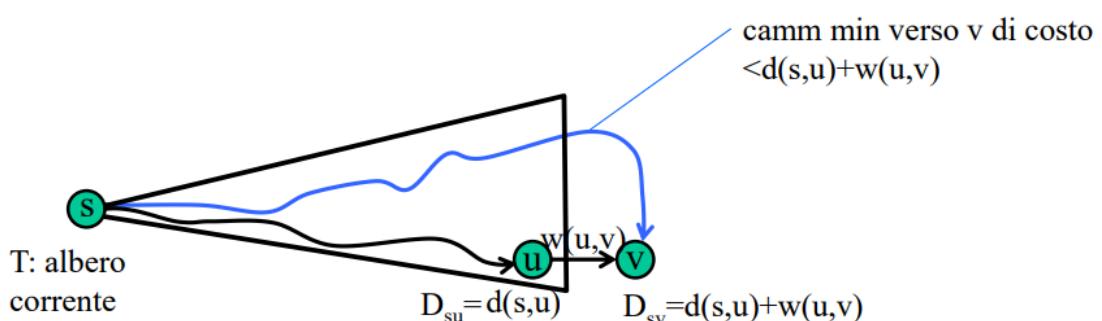
Lemma

Quando il nodo v viene estratto dalla coda con priorità vale:

- $D_{sv} = d(s, v)$ (stima esatta)
- il cammino da s a v nell'albero corrente ha costo $d(s, v)$ (camm. min in G)

dim (per assurdo)

Sia v il primo nodo per cui l'alg sbaglia
sia (u, v) l'arco aggiunto all'albero corrente (arco arancione)



Lemma

Quando il nodo v viene estratto dalla coda con priorità vale:

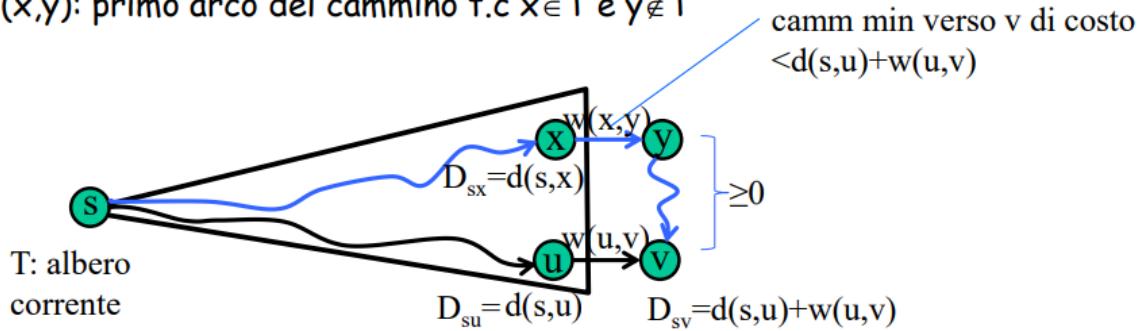
- $D_{sv} = d(s,v)$ (stima esatta)
- il cammino da s a v nell'albero corrente ha costo $d(s,v)$ (camm. min in G)

dim (per assurdo)

Sia v il primo nodo per cui l'alg sbaglia

sia (u,v) l'arco aggiunto all'albero corrente (arco arancione)

(x,y) : primo arco del cammino t.c $x \in T$ e $y \notin T$



Lemma

Quando il nodo v viene estratto dalla coda con priorità vale:

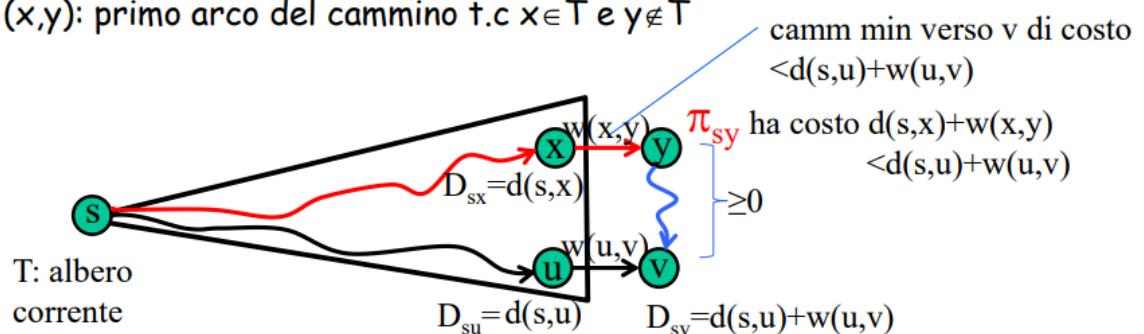
- $D_{sv} = d(s,v)$ (stima esatta)
- il cammino da s a v nell'albero corrente ha costo $d(s,v)$ (camm. min in G)

dim (per assurdo)

Sia v il primo nodo per cui l'alg sbaglia

sia (u,v) l'arco aggiunto all'albero corrente (arco arancione)

(x,y) : primo arco del cammino t.c $x \in T$ e $y \notin T$



$$D_{sy} \leq d(s,x) + w(x,y) < d(s,u) + w(u,v)$$

assurdo: l'alg avrebbe estratto y e non v

(se $y=v$, v avrebbe avuto una stima più piccola)

