Algoritmi e Strutture Dati - A.A. 2018/2019

Roberto Frenna

July 6, 2020

Contents

1 Prefazione

Questi appunti sono stati scritti in tempo reale mentre seguivo le lezioni frontali di Benerecetti. Sfortunatamente, non ho mai trovato l'occasione di integrarli propriamente con le videolezioni (o altre registrazioni) dello stesso anno scolastico, pertanto ci sono svariate problematiche:

- Manca un filo logico e una coesione in alcune spiegazioni, dovute al fatto che cercavo di formalizzare un linguaggio parlato che non sempre ha un risultato ottimale.
- È capitato diverse volte che mi assentassi durante le lezioni o che arrivassi in ritardo, quindi alcuni argomenti potrebbero mancare o essere incompleti. In genere, tali occorrenze sono annotate.
- Ci sono un po' di easter egg sparse che potrebbero non instillare molta fiducia.
- Imparavo nuove funzionalità di LaTeX man mano che scrivevo gli appunti – la qualità, quindi, non è costante.
- La parte iniziale sulle notazioni asintotiche è assente.
- Gli argomenti non sono ordinati dal punto di vista "logico", sono ordinati seguendo l'ordine scelto dal professore durante la spiegazione.

Tra ogni lezione dovrebbe essere presente un indicatore con la data del giorno. Nel caso sia necessario, dovrebbe quindi risultare semplice integrare ciò che manca con il resto che è stato spiegato.

Viste le considerazioni, consiglio *caldamente* di utilizzare e integrare questi appunti con gli altri appunti noti, che sono di qualità superiore (oltre ad essere scritti offline con più cura, calma e dedizione).

2 Algoritmi di ordinamento

2.1 Insertion Sort

L'algoritmo Insertion Sort è efficiente soltanto per sequenze piccole, ma è utile in quanto è un algoritmo di ordinamento "sul posto". Alla base del suo funzionamento vi è un indice i, inizialmente impostato al primo elemento, alla cui sinistra la sequenza è considerata ordinata. A quel punto si seleziona il primo elemento successivo ad i nella sottosequenza non ordinata scegliendo $j = i + 1$ e si cerca il posto per j all'interno della sequenza ordinata. Infine, si incrementa i e si ritorna alla selezione di j se la sequenza non è terminata.

Listing 1: Implementazione di "insertion sort"

```
1 InsertionSort(A)
2     for j = 2 to Length(A) do
3         key = A[j]
4         i = j - 1
5         while i > 0 && A[i] > key do
6             A[i + 1] = A[i]
7             i = i - 1
8         A[i + 1] = key
9     return A
```

Nel caso migliore la funzione ha complessità $\Omega(n)$, nel caso peggiore $O(n^2)$ e nel caso medio sempre $O(n^2)$.

2.2 Ordinare una sequenza con l'approccio "divide et impera"

Data una sottosequenza *7, 3, 2, 10, 15, 1, 8, 2*, si può dividere in due sottosequenze distinte:

$$7, 3, 20, 10 \; e \; 15, 1, 8, 2$$

Dividendo ulteriormente la prima sottosequenza, potrò ad esempio ottenere *7, 3* che ordinato è *3, 7* e *20, 10* che ordinata è *10, 20*.

$$7, 3, 20, 10 \rightarrow 3, 7, 10, 20$$

$$15, 1, 8, 2 \rightarrow 1, 2, 8, 15$$

Si procede a scegliere gli elementi più piccoli per costruire la sequenza definitiva ordinata:

$$1, 2, 3, 7, 8, 10, 15, 20$$

2.3 Merge Sort

Input: *una sottosequenza da ordinare che inizia nella posizione p e termina nella posizione r*. Si vuole individuare la posizione q che sta a metà tra p e r, se la dimensione è sufficientemente grande.

Listing 2: Implementazione del "merge sort"

```
1 MergeSort(A, p, r)
2     if p < r then
3         q = (p + r) / 2
4         MergeSort(A, p, q)
5         MergeSort(A, q + 1, r)
6         Merge(A, p, q, r)
```

La linea 2 determina se la sequenza di ingresso è un caso base. Le linee 4 e 5 rappresentano la parte *impera* dell'approccio *divide et impera*. Al termine delle due chiamate in linea 4 e 5, gli elementi che stanno tra p e q e tra $q+1$ ed r sono ordinati. Servirà quindi una chiamata successiva per unire le due sottosequenze affinché l'intera sequenza sia ordinata.

2.3.1 Termine della funzione ricorsiva

La funzione termina se ogni chiamata ricorsiva raggiunge il caso base. Come si fa a garantirlo? Per assicurarsi che l'algoritmo termini bisogna essere sicuri che $[p, q] < [p, r]$ e che $[q + 1, r] < [p, r]$, ovvero che il numero di elementi tra p e q è strettamente minore del numero di elementi che stanno tra p ed r.

$$p < r \Rightarrow p + p < p + r \Rightarrow 2p < p + r$$

$$p \leq q < r \Rightarrow p \leq \frac{p + r}{2} < r$$

La precedente disuguaglianza è verificata? E' banale dimostrarlo moltiplicando entrambi i membri per 2:

$$2p \leq p + r < 2r$$

Ripetiamo il procedimento aggiungendo r invece di p:

$$p < r \Rightarrow p + r < r + r \Rightarrow p + r < 2r$$

Attenzione, non è scontato che la seguente sia vera:

$$2 \times \frac{p+r}{2} = p + r$$

Se infatti il numero non è esattamente divisibile per 2, ciò risulta falso ad esempio per

$$\frac{p+r}{2} = \frac{5}{2} = 2$$

Se $p + r$ è *pari*, la seguente è verificata:

$$2p < p + r < 2r \Rightarrow p < \frac{p+r}{2} < r$$

Se $p + r$ è *dispari*:

$$p \leq \frac{p+r}{2} < r$$

Poiché in entrambi i casi la condizione iniziale è verificata, possiamo garantire che la funzione avrà fine.

2.3.2 Complessità

Trattandosi di una funzione ricorsiva, i metodi usati fino ad ora per misurare la complessità di una funzione non risultano immediatamente utilizzabili. Tuttavia, esistono diversi metodi che risultano anche più semplici di quelli precedentemente studiati.

2.3.3 Digressione: complessità di un algoritmo ricorsivo, il fattoriale

$$n! = \prod_{i=1}^{n} i = n(n-1)(n-2) \times \cdots \times 1 = n \times \prod_{i=1}^{n-1} = n(n-1)!$$

Ricordiamo che:

$$\prod_{i=1}^{0} i = 1$$

Quindi:

$$0! = 1$$

In definitiva: $n! = 1$ se $n = 0$, oppure $n(n-1)!$ se $n > 0$. Definiamo ora l'algoritmo:

Listing 3: Implementazione dell'algoritmo del fattoriale

```
1 Fact(n)
2     if n > 0 then
3         ret = n * Fact(n-1)
4     else
5         ret = 1
6     return ret
```

Il tempo della funzione fattoriale $T_F(n)$ è:

$$T_F(n) = \begin{cases} 2 & se\ n = 0 \\ 3 + T_F(n-1) & se\ n > 0 \end{cases}$$

Questa equazione viene detta **equazione di ricorrenza**.

2.3.4 Calcolo della complessità del "merge sort"

Dato:

$$r - p + 1 = n$$

Allora:

$$T_{MS}(n) = \begin{cases} 1 & se\ n \leq 1 \\ 1 + T_{merge}(n) + 2T_{MS}(\frac{n}{2}) & se\ n \geq 2 \end{cases}$$

Sviluppiamo la seconda equazione. $T_{merge}(n)$ è sicuramente almeno $\Theta(1)$ (lineare), quindi "assorbe" il coefficiente 1 nell'equazione. Si suppone inoltre che $T_{merge}(n) = \Theta(n)$. In definitiva, si ottiene:

$$T_{MS}(n) = \begin{cases} 1 & se\ n \leq 1 \\ 2T_{MS}(\frac{n}{2}) + \Theta(n) & se\ n \geq 2 \end{cases} \quad (1)$$

2.3.5 Merge di due sequenze ordinate

Consultare le slide per un'implementazione. Come precedentemente supposto, la complessità è $\Theta(n)$.
Nota: il professore non ha mai mostrato o spiegato l'effettiva implementazione dell'algoritmo di merge. Che sia traducibile in un "non è necessario studiarlo" è a discrezione del lettore.

2.3.6 Digressione: risoluzione dell'equazione di ricorrenza del fattoriale

$$T_F(n) = \begin{cases} 1 & se\ n = 0 \\ T_F(n-1) + 1 & se\ n \geq 1 \end{cases}$$

$$\text{chiamata } (\Theta(1)) \rightarrow \textit{chiamata } (\Theta(1)) \rightarrow \textit{chiamata}(\Theta(1)) \dots$$

Questa catena dipende da n.

Table 1: Analisi dei livelli di ricorsione

Livello ricorsione	**Input**	**N. op elementari**
$\emptyset$	n	$\Theta(1)$
1	$n-1$	$\Theta(1)$
2	$n-2$	$\Theta(1)$
...	...	$\Theta(1)$

$$T_F(n) = \sum_{i=0}^{n} \Theta(i) = \Theta(n)$$

2.3.7 Risoluzione dell'equazione di ricorrenza del "merge sort"

October 12, 2018

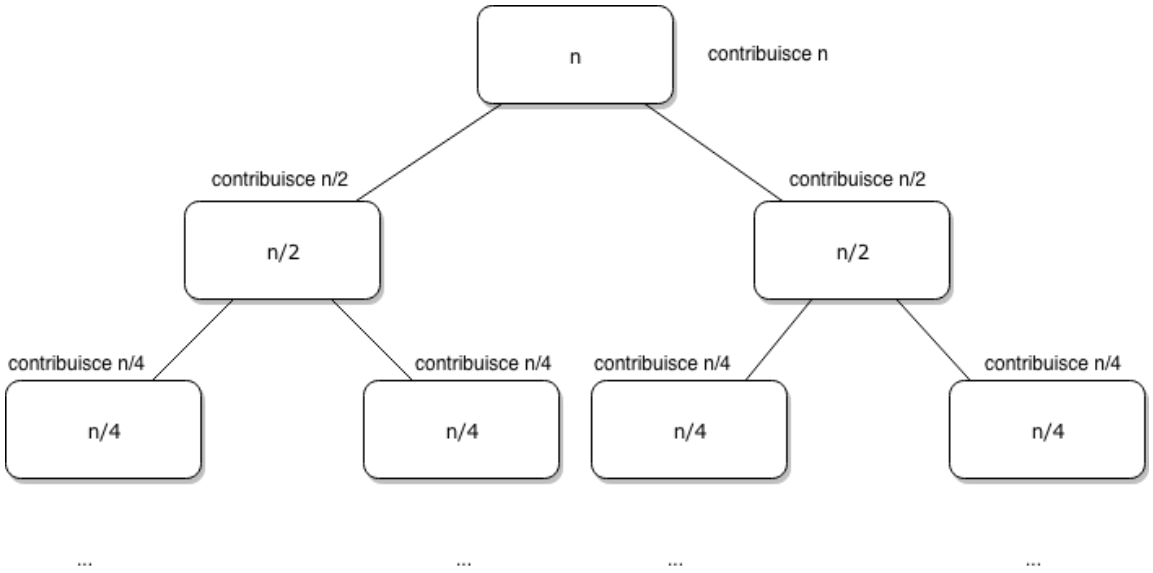


Figure 1: Insieme delle ripetizioni dell'algoritmo "merge sort" (call-stack)

Si tratta di un albero pieno, ovvero un albero in cui tutti i percorsi dalla radice ad una foglia hanno la stessa lunghezza. L'altezza di questo albero corrisponde al livello i cui nodi ricevono in ingresso un'istanza di dimensione 1.

Table 2: Analisi dei livelli di ricorsione

Livello ricorsione	**Dimensione input**	**N. op elementari**
0	n	$\Theta(1)$
1	$\frac{n}{2}$	$\Theta(1)$
2	$\frac{n}{4}$	$\Theta(1)$
...	$\frac{n}{2^i}$	$\Theta(1)$

La soluzione dell'equazione dipende dall'altezza dell'albero:

$$\frac{n}{2^i} = 1 \Rightarrow n = 2^i \Rightarrow i = log_2 n$$

Il tempo si può quindi esprimere come:

$$T_{MS}(n) = \sum_{i=0}^{log_2 n} n = n \times \sum_{i=0}^{log_2 n} 1 = n \times (log_2 n + 1) = \Theta(n log_2 n)$$

Il tempo di esecuzione dell'algoritmo è indipendente dalla composizione delle sequenze, e dipende solo dalla dimensione dell'istanza. Caso medio, caso base e caso peggiore coincidono sempre.

2.4 Digressione: altri esempi di equazioni di ricorrenza

2.4.1 Analisi di un'equazione di ricorrenza con contributo quadratico

$$T(n) = \begin{cases} 1 & se\ n \leq 1 \\ 2T(\frac{n}{2}) + n^2 & se\ n > 1 \end{cases} \quad (2)$$

Il contributo di n è quadratico, e potrebbe ad esempio rappresentare la versione del *merge sort* dove il tempo del merge è quadratico.

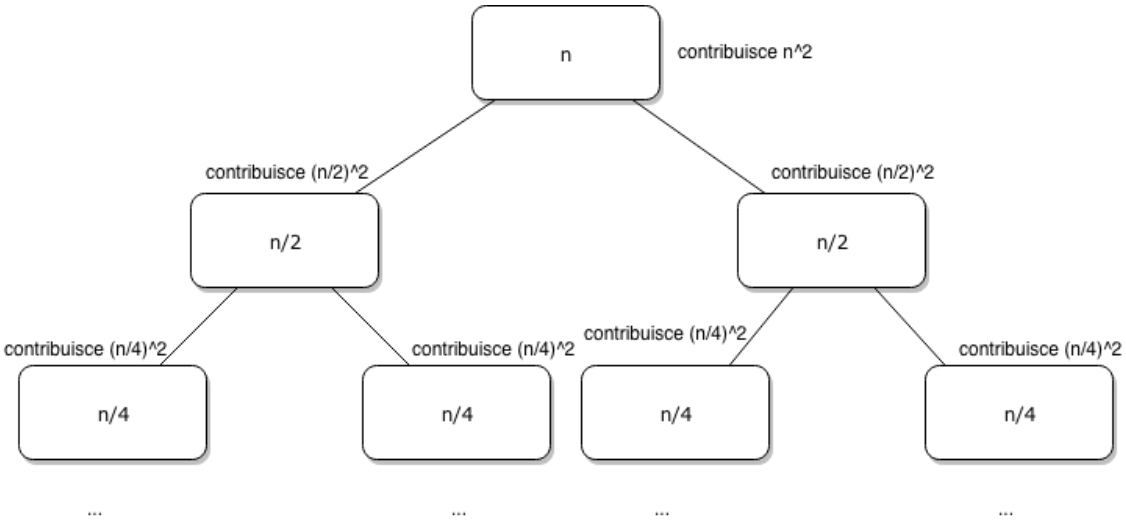


Figure 2: Insieme delle ripetizioni di un algoritmo con contributo quadratico

Qual è il contributo dei nodi al livello i_{esimo}? Al livello 0, è n^2, al livello 1 è $2(n/2)^2$, al livello 3 è $4(n/4)^2$. Un qualsiasi nodo al livello i_{esimo} riceverà un input $\frac{n}{2^i}$. Al livello i_{esimo} ci sono 2^i nodi, quindi il contributo dell'i_{esimo} livello è:

$$2^i \times \left(\frac{n}{2^i}\right)^2$$

Pertanto:

$$T(n) = \sum_{i=0}^{log_2 n} \left(2^i \times \left(\frac{n}{2^i}\right)^2\right)$$

Si procede alla semplificazione della sommatoria. Si osserva che:

$$2^i \times \frac{n^2}{(2^i)^2} = \frac{n^2}{2^i}$$

Quindi:

$$T(n) = \sum_{i=0}^{log_2 n} \frac{n^2}{2^i} = n^2 \times \sum_{i=0}^{log_2 n} \left(\frac{1}{2}\right)^i \tag{3}$$

Questa sommatoria è la somma parziale della serie geometrica x^i.

$$\sum_{i=0}^{k} x^i = \frac{x^{n+1} - 1}{x - 1} = \frac{1 - x^{n+1}}{1 - x}$$

Si dimostra facilmente svolgendo le varie somme:

$$(1+x^1+x^2+x^3+\cdots+x^k)(1-x) = (1+x+x^2+x^3+x^k)-(x+x^2+x^3+x^k+x^{k+1}) = 1-x^{k+1}$$

Quindi nel caso richiesto avremo:

$$T(n) = n^2 \times \frac{1 - (\frac{1}{2}^{1+log_2 n})}{\frac{1}{2}} = 2n^2 \times \left(1 - \frac{1}{2 \times 2^{log_2 n}}\right) = 2n^2 \times \left(1 - \frac{1}{2n}\right) = 2n^2 - n = \Theta(n^2)$$

Postilla $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ se $0 < x < 1$, ovvero la sommatoria quando la ragione è compresa tra 0 e 1 è costante.

Come si può dimostrare che questa funzione è una soluzione dell'equazione di ricorrenza 2? Supponiamo che l'equazione 3 sia vera, si avrà:

$$T\left(\frac{n}{2}\right) = \sum_{i=0}^{log_2^{\frac{n}{2}}} \left(2^i \left(\frac{\frac{n}{2}}{2^i}\right)^2\right)$$

Ovvero:

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

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

Ricordando che:

$$n^2 = 2^0 \times \left(\frac{n}{2^0}\right)^2$$

Sostituendo, si ottiene che:

$$T(n) = \sum_{i=0}^{log_2 n} 2^i \left(\frac{n}{2^i}\right)^2$$

Tale risultato soddisfa l'equazione 2. ■

2.4.2 Equazione di ricorrenza con albero non pieno

$$T(n) = \begin{cases} 1 & se\ n \leq 1 \\ T(\frac{n}{3}) + T(\frac{n}{2}) + n & se\ n > 1 \end{cases} \quad (4)$$

In questa equazione una delle chiamate ricorsive riceve un terzo della dimensione dell'input di partenza, e l'altra un mezzo della dimensione dell'input di partenza.

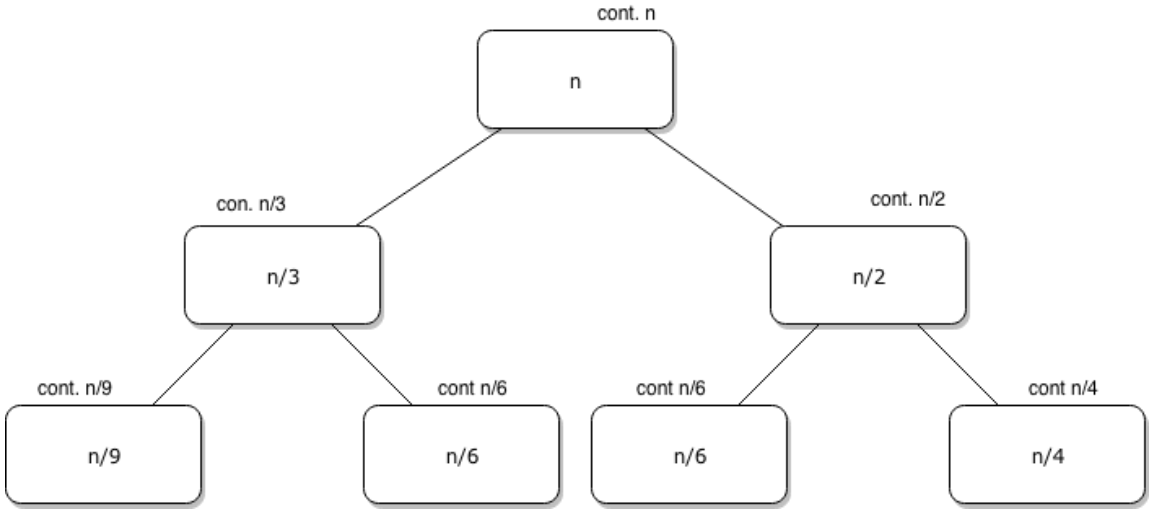


Figure 3: Insieme delle ripetizioni

Osservando l'albero di ricorrenza, è evidente che nell'albero ci saranno nodi molto più corti degli altri, ad esempio il ramo di sinistra arriverà molto più velocemente al caso base. La forma di questo albero è quindi sostanzialmente diversa rispetto a quella di quelli studiati precedentemente. Questo albero quindi **non è pieno**. Come

si rappresentano i contributi di un albero di questa tipologia?
Possiamo calcolare facilmente i contributi dei livelli solo della parte dell'albero piena. La somma dei contributi sarà, per la prima porzione dell'albero, $n + \frac{5}{6}n + \frac{25}{36}n$. Si osserva che:

$$\frac{25}{36}n = \left(\frac{5}{6}\right)^2 n$$

Si può verificare che il contributo dei livelli pieni sarà:

$$\left(\frac{5}{6}\right)^i n$$

Ad un dato livello dell'albero, l'uguaglianza da soddisfare sarà:

$$\frac{n}{3^i} = 1 \Leftrightarrow n = 3^i \Rightarrow i = log_3 n$$

$$\sum_{i=0}^{log_3 n} \left(\frac{5}{6}\right)^i n = n \sum_{i=0}^{log_3 n} \left(\frac{5}{6}\right)^i = \Theta(n)$$

Per le considerazioni fatte prima, la sommatoria si può considerare come una costante e pertanto il contributo sarà $\Theta(n)$. Tuttavia, soltanto una parte dei nodi è stata sommata, quindi non è certo che il risultato sia corretto. Si può affermare che $T(n)$ è sicuramente limitato inferiormente da una funzione lineare ($\Theta(n)$).

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

Per approssimare per difetto, supponiamo che esistano anche i nodi che non esistano (cioè supponendo che l'albero sia pieno, aggiungendo nodi che non ci sono). Facendo ciò, il termine generale del contributo che vale per la parte piena dell'albero varrà anche per l'altra parte dell'albero, e sarà più grande di quello effettivo. L'altezza è determinata dalla catena di nodi più lunghi, e sarà $log_2 n$. Se si calcolasse la sommatoria dei termini determinati su tutto l'albero che approssima l'albero effettivo, si ottengono gli stessi livelli, ma la sommatoria invece di andare a $log_3 n$ andrà a $log_2 n$. A prescindere dal termine della sommatoria, essa sarà sempre una funzione lineare. Quindi sarà limitata superiormente:

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

Ma se $T(n) = O(n)$ e $T(n) = \Omega(n)$, allora

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

Qual è un'equazione di ricorrenza per la quale non vale questa proprietà?

$$T(n) = \begin{cases} 1 & se\ n \leq 1 \\ T(\frac{2n}{3}) + T(\frac{n}{2}) + n & se\ n > 1 \end{cases}$$

Nota bene Sembra che ci sia un errore nell'equazione di sopra - i due termini dovrebbero essere invertiti. Consultare la registrazione del 16 ottobre per ulteriori informazioni.

$$\frac{\left(\frac{7}{6}\right)^{log_3 n+1} - 1}{\frac{1}{6}} \leq T(n) \leq \frac{\left(\frac{7}{6}\right)^{log_2 n+1} - 1}{\frac{1}{6}}$$

In che relazione si trovano $\left(\frac{7}{6}\right)^{log_3 n}$ e $\left(\frac{7}{6}\right)^{log_2 n}$?

$$\left(\frac{7}{6}\right)^{log_3 n} = n^{log_3 \frac{7}{6}}$$

$$\left(\frac{7}{6}\right)^{log_2 n} = n^{log_2 \frac{7}{6}}$$

Queste due funzioni sono asintoticamente equivalenti? n^x e n^y sono asintoticamente costanti se e solo se $x = y$. Tuttavia:

$$log_2 \frac{7}{6} \neq log_3 \frac{7}{6}$$

Si può solo affermare:

$$T(n) = \Omega(n^{log_3 \frac{7}{6}})$$

$$T(n) = O(n^{log_2 \frac{7}{6}})$$

October 16, 2018

2.5 Selection sort

Dato un vettore di n elementi, il più grande deve stare nella posizione n (la più alta). Una volta trovato il più grande (in posizione i ad esempio), si inverte l'elemento di posizione n con quello di posizione i. Si procede iterando considerando $n = n - 1$ e così via.

Nota Sono arrivato in ritardo, la spiegazione è incompleta. Consultare la registrazione video del giorno.

Listing 4: Implementazione dell'algoritmo selection sort

```
1  Findmax(A, J)
2      max = 1
3      for i = 2 to j do
4          if A[i] > A[max] then
5              max = i
6      return max
7  SelectionSort(A, n)
8      j = n
9      i = Findmax(A, j)
10     while j > 1 do
11         swap(A, i, j)
12         j = j - 1
13         if j > 1 then
14             i = Findmax(A, j)
```

2.5.1 Complessità del selection sort

La seguente è la complessità della linea 14:

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

In definitiva, la complessità del selection sort è la seguente:

$$T_{SS} = \Theta(n^2)$$

Impiega esattamente lo stesso tempo a prescindere della sequenza in input.

2.5.2 Ottimizzazione del selection sort

Per ottenere il massimo in linea 9, l'algoritmo effettua necessariamente dei confronti. Una singola ricerca del massimo non fornisce dati sulle relazioni tra tutte le copie di elementi, ma una parte sì (eccetto il caso in cui la sequenza in input sia ordinata). Il modo per rendere l'algoritmo più "furbo" e quindi efficiente è mantenere le informazioni determinate da Findmax. Naturalmente la prima chiamata a Findmax non può essere ottimizzata ulteriormente in quanto non ci sono informazioni pregresse sulla lista.

Determinare cosa mantenere L'idea è rappresentare un ordinamento parziale, ovvero un insieme di elementi (cioè di elementi contenuti nella sequenza di ingresso) in una struttura che permetta di dedurre informazioni su alcune di queste - lasciando aperta la possibilità che su alcuni elementi non si abbiano informazioni sulla loro relazione d'ordine. Mentre una sequenza è adeguata per rappresentare un ordinamento totale, non si presta altrettanto per una situazione come questa. In una struttura ad albero, invece, sì - si può infatti dire che due elementi collegati tra loro sono in relazione d'ordine, mentre due elementi non collegati non sono in relazione tra loro.

3 Introduzione alle strutture dati specializzate

3.1 Albero binario

Un albero binario può o essere:

$$AB = \emptyset$$

Oppure un insieme che può essere partizionato in tre sottoinsiemi disgiunti:
Un albero binario pieno è albero binario in in cui ogni nodo interno ha due figli e tutte le foglie si trovano alla stessa profondità.

Proprietà degli alberi pieni

1. Data un'altezza h, esiste uno e un solo modo per rappresentare un albero binario pieno di quell'altezza.
    - Aggiungendo o togliendo nodi, le proprietà che definiscono un albero binario pieno non sono più rispettate.
2. Ogni sottoalbero di un albero pieno è pieno.

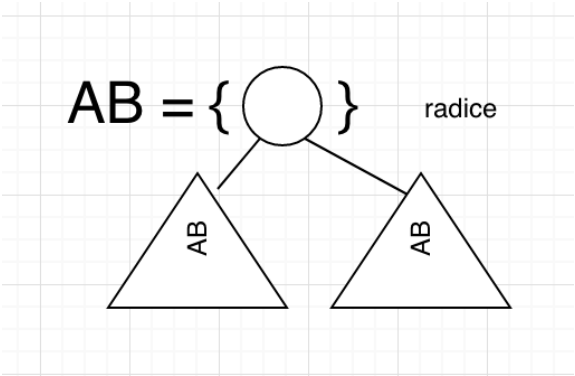


Figure 4: Definizione di albero binario

3. Il numero di nodi di un albero pieno di altezza h si calcola con:

$$nn(h) = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \text{ (somma parziale della serie geometrica)}$$

Una limitazione degli alberi binari è che possono essere rappresentati soltanto un numero di nodi pari ad una potenza di 2 meno 1. *Non tutti gli insiemi si possono quindi rappresentare con un albero binario pieno.*
Dato il numero di nodi si può inoltre ricavare l'altezza dell'albero.

$$n = 2^{h+1} - 1 \Rightarrow log_2 n = log_2\left(2^{h+1} - 1\right) < log_2\left(2^{h+1}\right) = h + 1$$

$$2^h \leq 2^{h+1} - 1 \Rightarrow h \leq log_2 n < h + 1$$

$$h = \lfloor log_2 n \rfloor$$

Dati n elementi, non esiste alcun albero che ha altezza inferiore ad $h = \lfloor log_2 n \rfloor$. Inoltre, gli alberi binari pieni sono quelli che a parità di numero di nodi, hanno altezza minima.

3.2 Albero binario completo

Un **albero binario completo** è un albero binario in cui:

1. ogni nodo interno ha due figli tranne al più uno (cioè tutti i nodi interni hanno grado due tranne al più uno)

2. tutte le foglie sono a profondità h o $h - 1$ (alla stessa profondità)

Non è più automaticamente vera la proprietà secondo la quale data l'altezza si può determinare il numero di nodi. Il numero di nodi di un albero completo di altezza h si può includere in un intervallo. L'albero più piccolo di altezza h è l'albero di altezza $h-1$ con un nodo in più, mentre l'albero più grande di altezza h è l'albero binario pieno di altezza h. Quindi:

$$1 + 2^h - 1 \leq n \leq 2^{h+1} - 1 \Rightarrow 2^h \leq n \leq 2^{h+1} - 1$$

In definitiva:

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

Applicando i logaritmi ad entrambi i membri, risulta:

$$h \leq log_2 n < h + 1$$

Un qualsiasi numero compreso tra h ed $h+1$ escluso, se è un intero è uguale ad h, altrimenti è strettamente compreso tra i due. La base è intera, quindi:

$$h = \lfloor log_2 n \rfloor$$

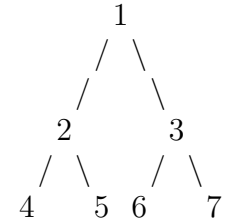
h risulta uguale – a partire dal numero di nodi – sia per gli alberi binari completi che pieni.
Si può osservare che:

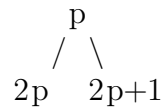
$$\forall n, \exists h : 2^h \leq n \leq 2^{h+1} - 1$$

Ogni n può quindi essere rappresentato con un albero binario completo, quindi si possono rappresentare insiemi di cardinalità arbitraria.

3.2.1 Rappresentazione di un array tramite un albero binario completo

Data una sequenza $1, 2, 3, 4, 5, \ldots, n$. La radice dell'albero sarà 1, e gli elementi possono essere semplicemente disposti naturalmente in successione nell'albero.





Dato il padre p, il figlio sinistro è $2p$, il figlio destro è $2p+1$. Quindi, sapendo i valori dei figli, è possibile determinare quello del padre.

October 18, 2018

3.3 Heap

Si passerà ora alla nozione della struttura dati necessaria per poter codificare la relazione di ordine parziale di cui si era precedentemente parlato.
Un heap è:

1. un albero binario completo
2. per ogni nodo i dell'albero, $val(i) \geq val(j)$ con j figlio di i, ovvero se prendo un qualsiasi nodo di questo albero completo, il valore di quel nodo è maggiore uguale dei valori associati ai suoi figli.

Si sta codificando un sottoinsieme di relazioni di ordine tra gli elementi. Infatti, ogni arco che connette due nodi indica che il primo elemento dell'arco è maggiore o uguale del secondo elemento dell'arco. Si tratta di un'informazione parziale perché non è nota la relazione tra tutti gli elementi.

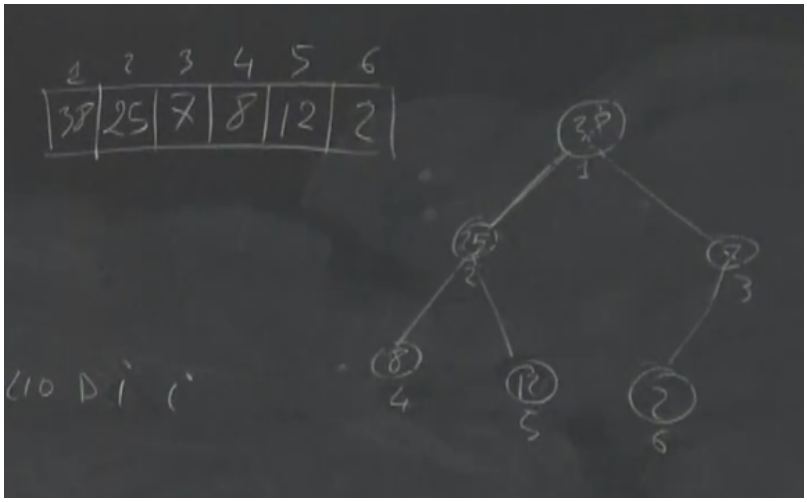


Figure 5: Rappresentazione di una sequenza con un heap

"Appiattendo" l'heap, si può dire:

$$A[i] \geq A[2i] \text{ e } A[i] \geq A[2i+1]$$

Una proprietà implicita è che in un heap l'elemento massimo è nella radice.

4 Algoritmi di ordinamento: parte due

4.1 Heap sort

Supponiamo di avere un albero completo che ha radice i i cui sottoalberi siano heap ma che non sia necessariamente un heap: ciò può avvenire solo se unicamente la radice viola la proprietà di heap. L'obbiettivo è definire una procedura che trasformi questo sottoalbero radicato in i in un heap. L'operazione consiste nel posizionare in i il massimo tra i e il massimo dei due heap, che è collocato nella loro radice. E' quindi possibile a tempo costante trovare il massimo tra i tre, ma non è sufficiente perché spostando la radice di un heap si violerà a sua volta la priorità di heap. Ma j è considerabile ancora come un albero completo con radice j i cui sottoalberisiano heap, e si può quindi ripetere la procedura. Si tratta di una procedura lineare sull'altezza dell'albero e logaritmica sul numero dei nodi.

Listing 5: Algoritmo "heapify"

```
1  Heapify(A, i)
2      max = i
3      l = 2 * i
4      r = 2 * i + 1 {figli di i}
5      if l <= HEAPSIZE and A[l] > A[max] then
6          max = l
7      if r <= HEAPSIZE and A[r] > A[max] then
8          max = r
9      if max != i then
10         Swap(A, i, max)
11         {ritrasformare A in heap}
12         Heapify(A, max)
```

La correttezza di questo algoritmo si basa sul fatto che i due sottoalberi di i siano heap, pertanto non è adatto per convertire un'intera sequenza in heap.
Banalmente, si ricava che:

$$T_{Heapify}(h) = \Theta(h) \text{ (sull}'\text{altezza dell}'\text{albero)}$$

$$T_{Heapify}(n) = \sum_{i=0}^{log_2 n} \Theta(1) = \Theta(log_2 n) \text{ (sul numero di nodi)}$$

4.1.1 Costruire il primo heap

Adesso ci si può concentrare sul problema di costruire il primo heap data una sequenza arbitraria. La sequenza si può in ogni caso rappresentare come un albero completo per la definizione stessa di albero completo, ma è errato chiamare Heapify sulla radice. Si può osservare però che facendo più chiamate a Heapify su tutti i nodi interni partendo da quelli con profondità maggiore si riesce a trasformare l'intera sequenza in heap! Per prima cosa bisogna trovare dove si trovano le foglie all'interno della sequenza: l'indice della massima foglia diviso due corrisponde al padre, che è il nodo interno di posizione massima. Non possono esserci altri nodi interni di posizione maggiore di $\lfloor \frac{n}{2} \rfloor$, perché se ci fosse ci dovrebbe essere almeno una foglia che è in posizione $2 * \text{foglia}$. Pertanto, da 1 fino a $\lfloor \frac{n}{2} \rfloor$ si avranno solo nodi interni, mentre i successivi ($> \lceil \frac{n}{2} \rceil$) saranno tutte foglie. Quindi, applicando Heapify dalla posizione detta fino all'inizio scorrerà i nodi esattamente nell'ordine descritto in precedenza.

Listing 6: Algoritmo "CostruisciHeap"

```
1 CostruisciHeap(A, m)
2     HEAPSIZE = m
3     for i = floor(n/2) down to 1 do
4         Heapify(A, i)
```

4.1.2 Implementazione algoritmo di ordinamento

Listing 7: Algoritmo "HeapSort"

```
1 HeapSort(A, m)
2     CostruisciHeap(A, m)
3     for i := m down to 2 do
4         Swap(A, 1, HEAPSIZE)
5         HEAPSIZE = HEAPSIZE - 1
6         Heapify(A, 1)
```

4.1.3 Analisi complessità

Nota: questa analisi potrebbe essere leggermente incompleta.

$$T_{CostruisciHeap} = O(nlog_2 n)$$

$$T_{HeapSort} = O(nlog_2 n)$$

$$h_i \leq h_1 = \lfloor log_2 n \rfloor$$
$$T_{Heapify} = \Theta(h1) = O(log_2 n)$$
$$\Theta(1) = \frac{n}{4}; \Theta(2) = \frac{n}{8}$$

Preso un livello i_{esimo} di nodi interni, ci sono $\frac{n}{2^{i+1}}$ alberi di altezza i, e per ciascuno di essi il costo sarà $\Theta(i)$. Il contributo totale è:

$$\sum_{i=1}^{h} \left(\frac{n}{2^{i+1}} \times \Theta(i) \right)$$

Quindi:

$$T_{CostruisciHeap}(n\ nodi) = \sum_{i=1}^{h} \left(\frac{n}{2^{i+1}} \times \Theta(i) \right)$$

h è l'altezza dell'albero, quindi:

$$T_{CostruisciHeap}(n\ nodi) = \sum_{i=1}^{\lfloor log_2 n \rfloor} \left(\frac{n}{2^{i+1}} \times \Theta(i) \right) = \Theta \left(\sum_{i=1}^{\lfloor log_2 n \rfloor} \left(\frac{n}{2^{i+1}} \times i \right) \right)$$

In definitiva:

$$T_{CostruisciHeap}(n\ nodi) = \Theta \left(\frac{n}{2} \sum_{i=1}^{\lfloor log_2 n \rfloor} \left(i \times \left(\frac{1}{2} \right)^i \right) \right)$$

Si può osservare che $i \times x^i = x \times i \times x^{i-1}$, e che:

$$i \times x^{i-1} = \frac{d}{dx}(x^i)$$

$$\sum_{i=0}^{k} i \times x^i = \sum_{i=0}^{k} \left(x \times \frac{d}{dx}(x^i) \right)$$

Si può osservare che sia x che n non dipendono da i, quindi possono essere raccolti:

$$x \times \sum_{i=0}^{k} \frac{d}{dx} x^i$$

La derivata è un operatore lineare, il che significa che distribuisce o commuta con le somme. Quindi:

$$x \cdot \frac{d}{dx}\left(\sum_{i=0}^{k} x^i\right)$$

Si può osservare che:

$$\sum_{i=1}^{\lfloor log_2 n \rfloor} i \cdot x^i \leq \sum_{i=0}^{\infty} i \cdot x^i$$

Si può usare ∞ al posto di k nella precedente, rendendola:

$$x \cdot \frac{d}{dx}\left(\sum_{i=0}^{\infty} x^i\right)$$

Ricordando che:

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \text{ quando } 0 < x < 1$$

Sostituendo, si ottiene quando $0 < x < 1$:

$$x \cdot \frac{d}{dx}\left(\sum_{i=0}^{\infty} x^i\right) = x \cdot \frac{d}{dx}\left(\frac{1}{1-x}\right)$$

Si può facilmente derivare $\frac{1}{1-x}$:

$$\frac{d}{dx}\left(\frac{1}{1-x}\right) = \frac{d}{dx}(1-x)^{-1} = (-1) \cdot (1-x)^{-2} \cdot (-1) = \frac{1}{(1-x)^2}$$

Sostituendo nella precedente:

$$x \cdot \frac{d}{dx}\left(\frac{1}{1-x}\right) = \frac{x}{(1-x)^2}$$

Quindi:

$$T_{CostruisciHeap}(n \ nodi) = \Theta\left(\frac{n}{2} \cdot \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2}\right) = \Theta(n)$$

Da ciò si deduce che utilizzando *CostruisciHeap* al posto di *FindMax* non si perdono prestazioni, ma anzi, si guadagnano informazioni in più. Se facendo un'analisi più precisa di *CostruisciHeap* si è determinato che è $O(n)$ invece di $O(nlog_2 n)$ come precedentemente approssimato, si può essere sicuri che $T_{HeapSort}$ non possa essere migliore di $O(nlog_2 n)$?

4.1.4 Analisi in dettaglio della complessità di heap sort

HeapSort esegue $n-1$ chiamate a *Heapify*, ma la differenza rispetto a *CostruisciHeap* consiste che le chiamate ad Heapify in *CostruisciHeap* sono effettuate nella forma Heapify(A, i), mentre in *HeapSort* sono nella forma Heapify(A, 1).
Ciò significa che per ogni $\frac{n}{2}$ iterazioni, heap sort creerà alberi alti almeno $h-1$, quindi il costo dovrà essere almeno $\frac{n}{2}(h-1)$. Ci sono anche altre chiamate fatte ad Heapify fatte da heap sort, quindi il tempo sarà almeno:

$$\frac{n}{2}(h-1)+K$$

Ricordando che $h = \lfloor log_2 n \rfloor$, il tempo sarà

$$\Theta\left(\frac{n}{2} log_2 n\right) + K$$

Quindi:

$$T_{HeapSort} = \Omega(n log_2 n) \text{ } e \text{ } T_{HeapSort} = O(n log_2 n)$$

Nel caso peggiore, la complessità di HeapSort sarà quindi $\Theta(n log_2 n)$.

4.2 Ricapitolazione

Fino ad adesso sono stati trattati 4 algoritmi: *insertion sort*, *merge sort*, *selection sort* ed *heap sort* (che è la versione più efficace di *selection sort*). Tre di questi algoritmi (*insertion sort*, *selection sort* ed *heap sort*) risolvono il problema in maniera incrementale, mentre merge sort è un'implementazione dell'approccio *divide et impera*. Tuttavia, la soluzione del *merge sort* era soddisfacente dal punto di vista asintotico, ma meno rispetto alla complessità di spazio (per quanto ottimizzabile). Adesso si studierà un'altra implementazione dell'approccio *divide et impera*, che ha delle proprietà non asintoticamente buone quanto il merge sort, ma ha il vantaggio di essere asintoticamente ottimale nel caso medio. Nella pratica, questo algoritmo è quasi sempre superiore agli algoritmi visto fino ad adesso – tuttavia è possibile che per alcuni casi di input si comporti peggio, in quanto il tempo impiegato nel caso peggiore non è ottimale.

4.3 Quick sort

Il problema del *merge sort* è che una volta divisa la sequenza in due parti e ordinate entrambe, non è possibile fare alcuna supposizione tra gli elementi che si trovano tra una porzione della sequenza e l'altra. Se invece si potesse garantire:

$$\forall 1 \leq i \leq q, \forall q+1 \leq j \leq n, A[i] \leq A[j] \tag{5}$$

(cioè tutti gli elementi che sono a sinistra sono minori o uguali di tutti gli elementi che stanno a destra)
Si evita completamente il problema della fusione, e quindi il fulcro del problema si sposta dalla fusione alla suddivisione della sequenza stessa. L'operazione di suddivisione non potrà più essere a tempo costante come il *merge sort* – sarà quindi necessario trovare un equilibrio, cioè fare in modo che q sia il più vicino possibile a $\frac{n}{2}$.

4.3.1 Prima stesura dell'algoritmo

Listing 8: Prima stesura del "quick sort"

```
1 QuickSort(A, p, r)
2     if p < r then {non siamo in un caso base}
3         q = Partiziona(A, p, r) {indice di suddivisione
              calcolato da una funzione che cerchera' di
              garantire la proprieta' precedente}
4         QuickSort(A, p, q)
5         QuickSort(A, q + 1, r)
```

Quale delle seguenti proprietà è accettabile, ovvero permette a *quick sort* di terminare?

1. $p \leq q \leq r$
   Se $q = r$, la seconda chiamata non ha problemi (diventa infatti QuickSort(A, r+1, r)) perché ordina una sequenza vuota. Quindi si esclude questa.

2. $p \leq q < r$
   Se $q = p$, la prima chiamata ordinerebbe la sequenza da p a p, che è strettamente più piccola delle precedente che è grande almeno 2. La seconda va da $p+1$ ad r ed è strettamente più piccola della precedente – pertanto, $q = p$ non crea problemi.

3. $p < q \leq r$
   Questa può essere esclusa per le stesse motivazioni della **(1)**.

4. $p < q < r$
   Poiché la **(2)** è accettabile, questa (che è più restrittiva) può essere esclusa.

In definitiva, le proprietà che il quick sort deve garantire sono:

$$\forall p \leq i \leq q, \forall q + 1 \leq j \leq r, A[i] \leq A[j] \tag{6}$$

$$p \leq q < r \tag{7}$$

4.3.2 Sviluppo dell'algoritmo di partizionamento

Assumendo che $p < r$ e considerando come requisiti la 6 e la 7, consideriamo $x = A[p]$. Se l'algoritmo garantisce:

$$\forall p \leq i \leq q, A[i] \leq x \text{ e } \forall q + 1 \leq j \leq r, x \leq A[j]$$

Allora anche la 6 sarà soddisfatta per la transitività dell'operatore 'minore uguale'.

Figure 6: Reazione di uno studente ad "algoritmi e strutture dati"

Listing 9: Algoritmo di partizionamento

```
1  Partiziona(A, p, r)
2      x = A[p]
3      i = p - 1 {indice che scorre la sequenza da sinistra}
4      j = r + 1 {indice che scorre la sequenza da destra}
5      repeat
6          repeat
7              j = j - 1
8          until (A[j] <= x)
9          repeat
10             i = i + 1
11         until (A[i] >= x)
12         {ora ci si trova nella situazione del diagramma "
               post-two-repeats"}
13         if i < j then
14             swap(A, i, j)
15     until (i >= j)
16     return j
```

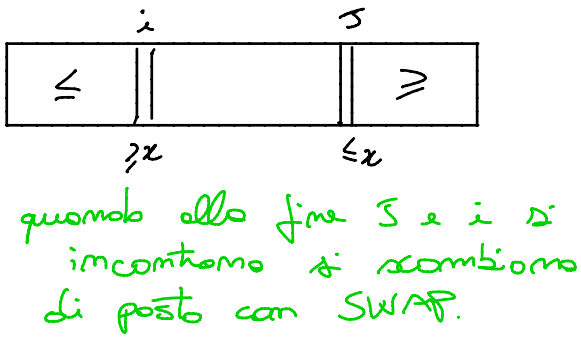


Figure 7: Situazione della lista dopo i due 'repeat'

Il requisito 7 è garantito – garantire il 6 è più delicato, in quanto il problema è legato ai due **repeat** in linea 6 e 9. **Non** possono essere vere le seguenti:

$$j \geq r$$

$$j < p$$

Tuttavia, la prima può essere semplificata in $j = r$ in quanto non può essere $j > r$. Nella prima iterazione si può garantire che in p ci sia x, ma nelle successive non è più

possibile farlo. Tuttavia, se x viene spostato, al suo posto viene sicuramente messo qualcosa che è $\leq x$, quindi nella posizione p ci sarà in ogni momento o x o qualcosa di strettamente minore di esso. La seconda proprietà è quindi verificata.
Bisogna ora garantire che sia i che j non possano arrivare entrambi contemporaneamente a r. Affinché j si fermi in r, poiché parte da $r+1$ per fermarsi in r deve essere eseguito un solo decremento. Ma se ciò è vero, vuol dire che quindi ci si trova nella prima ripetizione del **repeat** esterno. Ma ciò vuol dire che fino a quel momento non sono ancora stati fatti scambi, e i si trova a $p-1$. Se i si ferma a p e r a j, ci si trova sicuramente nella prima iterazione. Se $p < r$ (come da assunzione), e le precedenti sono vere, allora sicuramente $i < r$. Avviene lo scambio, e siccome sicuramente $i < r$ verrà effettuata sicuramente almeno un'altra iterazione e quindi j verrà decrementato, e non potrà mai diventare di nuovo uguale ad r.
Sia la 6 che la 7 sono quindi verificate.

4.3.3 Complessità del quick sort

October 23, 2018

$$n = r - p + 1$$

Ecco una prima stesura dell'equazione di ricorrenza del quick sort:

$$T_{QS}(n) = \begin{cases} 1 & se\ n \leq 1 \\ \Theta(1) + T_P(n) + T_{QS}(?) + T_{QS}(?) & se\ n \geq 2 \end{cases}$$

Dove il primo $\Theta(1)$ è il costo di confronti e assegnazione. E' necessario capire il tempo dell'operazione di partizionamento e sapere cosa inserire al posto dei punto di domanda. Per fare quest'ultima operazione, è necessario determinare la relazione tra la dimensione dell'input e il numero di chiamate eseguite. Le due sequenze, a sinistra e destra, hanno dimensione rispettivamente di:

$$q - p + 1 \text{ (sx) e } r - (q+1) + 1 \text{ (dx)}$$

Inoltre, è noto che:

$$p \leq q < r$$

Sapendo ciò, è chiaro che $q-p+1$ deve necessariamente essere ≥ 1. Il valore massimo che può assumere q è $r-1$, e sostituendo nell'equazione di sinistra si ottiene $r-p$ che è strettamente minore di $r-p+1$. Quindi:

$$1 \leq q - p + 1 \leq n - 1 \text{ (sx)}$$

$$1 \leq r - (q+1) + 1 \leq n - 1 \text{ (dx)}$$

La dimensione della sottosequenza di destra è ovviamente $dx = n - sx$, e per un abuso di notazione verrà indicata con q la dimensione della porzione di sinistra: $dx = n - q$ (*attenzione*, non coincide col q presente nell'algoritmo, ma è strettamente collegato con essa). Possiamo adesso aggiornare l'equazione di ricorrenza:

$$T_{QS}(n) = \begin{cases} 1 & se\ n \leq 1 \\ \Theta(1) + T_P(n) + T_{QS}(q) + T_{QS}(n-q) & se\ n \geq 2 \end{cases}$$

Tuttavia, dato n, q non è univocamente determinato - fissato un n, esistono diversi valori possibili di q. Pertanto, la forma dell'equazione di ricorrenza sembra dipendere proprio a causa di q da qualcosa in più oltre che alla dimensione della sequenza n, come nell'*insertion sort*. Essendo una chiamata ricorsiva, per ogni chiamata ci sarà un valore diverso di q (perché ogni chiamata determina un diverso partizionamento rispetto alla chiamata padre). Dentro l'equazione di ricorrenza ci sono quindi tanti parametri da prendere in considerazione. Si è quindi di fronte ad una famiglia di equazioni di ricorrenza, una diversa per ogni possibile valore di q. Ciò significa che il numero di possibili funzioni di tempo associabili al *quick sort* è molto elevato, e potenzialmente potrebbe essere uno diverso per ogni istanza. Nel caso peggiore, avremo infatti $n!$ possibili funzioni diverse.
Queste stesse considerazioni sono già state fatte durante il trattamento dell'*insertion sort*, e si applicheranno anche qui. Tuttavia, il fatto che il parametro di q possa cambiare a parità di n, induce a funzioni che sono asintoticamente diverse tra di loro? Nel caso dell'*insertion sort* ciò avveniva: esistevano sequenze nel quale il tempo era $\Theta(n^2)$ e altre in cui era $\Theta(n)$. Va fatto il medesimo studio in questo caso.

4.3.4 Complessità dell'algoritmo di partizionamento

Osservando l'algoritmo, si può dire che il **repeat** esterno può essere eseguito al più $n/2$ volte. Il minimo decremento possibile di j è 1, e il minimo incremento possibile di i è 1–ciò vuol dire che se ad ogni iterazione si ottengono i rispettivi minimi incrementi, uno si sposterà tanto quanto l'altro. Questo vuol dire che si incroceranno nel mezzo. Questa è solo una soluzione parziale, in quanto ancora non è noto il numero esatto di volte in cui verrà eseguito il ciclo esterno.
Rammentando la lezione precedente, i e j si incontrano necessariamente quando una delle seguenti condizioni è vera:

$$i = j \tag{8}$$
$$i = j + 1 \tag{9}$$

Se i due indici si incontrano senza effettuare scambi, si verificherà sempre il caso 8. Fatte le dovute considerazioni, è facile osservare che proprio perché al termine i e j

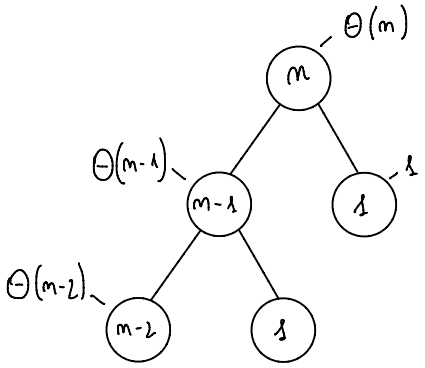


Figure 8: A. di ricorrenza del "quick sort"

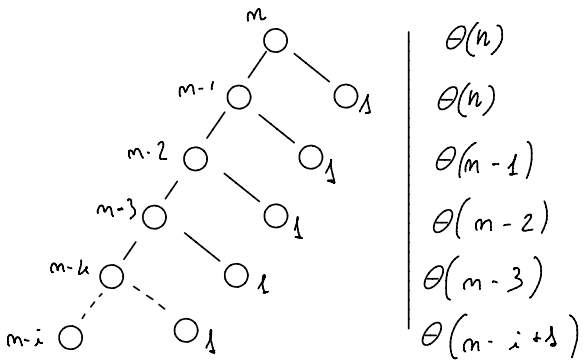


Figure 9: Albero degenere

dovranno soddisfare o la 8 o la 9 è noto il numero degli incrementi di i e di j. $n+2$ (ricavato nel caso in cui $j = n$) è il limite superiore di quante volte può essere eseguito il **repeat** esterno—è, cioè, il tempo asintotico richiesto dall'algoritmo. Quindi:

$$T_P = c(n+2) = \Theta(n)$$

4.3.5 Determinazione del caso migliore e del caso peggiore

L'equazione di ricorrenza del *quick sort* aggiornata è quindi:

$$T_{QS}(n) = \begin{cases} 1 & se\ n \leq 1 \\ \Theta(n) + T_{QS}(q) + T_{QS}(n-q) & se\ n \geq 2 \end{cases}$$

Ricordando che $1 \leq q \leq n-1$, si può sperimentare cosa succederebbe nei casi in cui q sia sempre 1 o $n-1$. Nel primo caso, l'equazione di ricorrenza diventerebbe:

$$T_{QS}(n) = \begin{cases} 1 & se\ n \leq 1 \\ T_{QS}(1) + T_{QS}(n-1) + \Theta(n) & se\ n \geq 2 \end{cases}$$

Si tratta di un caso specifico: ovvero quello in cui tutte le chiamate a Partiziona corrispondono ad una divisione in una sottosequenza lunga 1 e l'altra col numero restante di elementi. Inoltre, sostituendo l'altro estremo, si ricava la stessa equazione. Qual è la soluzione? L'albero di ricorrenza è quello in figura 8. La forma è molto simile ad un albero degenere di altezza n, come quello in figura 9.
Il nodo al livello i_{esimo} ha input $n-i$, e il contributo è $\Theta(n-i+1)$ (eccetto per il livello 0 in cui il contributo è $\Theta(n)$). L'altezza sarà $h = n-1$.

$$T_{QS}(n) = \Theta(n) + \sum_{i=1}^{n-1} \Theta(n-i+1)$$

La sommatoria avrà la classica risoluzione di $n(n+1)/2$ e quindi

$$T_{QS}(n) = \Theta(n^2)$$

Pertanto, se esistesse una sequenza input tale che ogni sua sottosequenza divida sempre la sequenza che riceve un elemento da una parte e tutti gli altri dall'altra, avrà tempo quadratico. Tale caso è verificato nel cui la sequenza in input sia ordinata in modo strettamente crescente o strettamente decrescente (**nota: verificare. Wikipedia parla di sequenze con valori tutti uguali**).
Esiste un valore di q nell'equazione di ricorrenza principale che permette di ricondurre ad un'equazione di ricorrenza già studiata? Se $q = \frac{n}{2}$ (cioè se esistesse una sequenza in input che viene sempre divisa a metà, compresa tutte le sequenze da lei generate), l'equazione di ricorrenza sarebbe:

$$T_{QS}(n) = \begin{cases} 1 & se\ n \leq 1 \\ \Theta(n) + 2T_{QS}(\frac{n}{2}) & se\ n \geq 2 \end{cases}$$

Che è esattamente l'equazione di ricorrenza del *merge sort*, che ha tempo

$$T_{QS}(n) = \Theta(nlog_2 n)$$

Esiste una sequenza che - invece - divide sempre in parti uguali l'input? Per ottenere ciò, è sufficiente che la sequenza chieda a Partiziona di effettuare il massimo numero di scambi (cioè proprio $n/2$, dove i due indici si incrociano nel mezzo). Un modo semplice per ottenere ciò è avere tutti elementi uguali al pivot. Si può quindi essere sicuri dell'esistenza di un *caso migliore* e di un *caso peggiore*. Come verificare se le soluzioni trovate corrispondono proprio ai due casi?
Un'osservazione è che la differenza tra gli alberi indotti dalle due equazioni di ricorrenza è la forma: hanno entrambi contributi lineari, ma uno ha altezza ridotta ($log_2 n$) e l'altro invece molto maggiore (n). Inoltre, gli alberi generabili dalla forma generale dell'equazione di ricorrenza del quick sort 4.3.5 sono alberi binari in cui ogni nodo interno ha necessariamente grado due. Il numero delle foglie è pertanto fisso: gli alberi generabili dall'equazione generica sono alberi binari, i cui nodi interni hanno sempre 2 figli e il numero di foglie è n. Pertanto, si tratta di alberi noti e ciò ci permette di concludere che hanno tutti lo stesso numero di nodi.

$$n_{foglie} = n \Rightarrow n_{nodi} = 2n - 1$$

4.3.6 Digressione: principio di induzione matematica

Si vuole dimostrare il seguente:

$$\forall n \geq 1, P(n)$$

Per farlo, si può dimostrare:

$$\begin{cases} P(1) \\ \forall k > 1, P(k-1) \Rightarrow P(k) \end{cases}$$

Esempio di dimostrazione

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

Si verifica banalmente il caso $P(1)$:

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

Si può procedere con la seconda parte della dimostrazione. Dato un $k > 1$ arbitrario, si può supporre che il seguente sia vero ($P(k-1)$)–detta **ipotesi induttiva**:

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

Si può dimostrare anche $P(k)$?

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

■

4.3.7 Digressione: dimostrazione del principio di induzione matematica (cont.)

October 25, 2018
Ero assente, integrare con videolezioni o altri appunti. Argomenti svolti: cont. tempo di esecuzione del quick sort (in particolare ricerca del caso medio e possibili tentativi per poter rendere il comportamento di quick sort più o meno distante dai casi estremi).

4.3.8 Tempo medio del quick sort

October 30, 2018
Scelto un "perno" (cioè pivot), e definito il rango è il numero di elementi della sequenza che sono minori o uguali del pivot, in funzione del rango è possibile determinare q_r (cioè la dimensione della sequenza di sinistra se il rango del pivot è uguale ad r).
Se il rango del perno è 1, la dimensione della porzione di sinistra sarà 1 e $n-1$ quella di destra, se il rango è 2 la porzione di sinistra sarà 1 e quella di destra $n-1$, da questo punto in poi i due valori si inseguono a distanza di 1 (quindi rango 2 dimensione 1, rango 3 dimensione 2, fino a rango n dimensione $n-1$). Facendo in modo che Partiziona scelga un elemento casuale come pivot, questo dà la stessa probabilità a tutti i valori del rango (quindi stessa probabilità di avere un pivot che dà rango 1 rispetto ad uno che dà rango 7). Ogni possibile valore di rango nella versione di 'Partiziona' randomizzata ha probabilità $1/n$ di scegliere un pivot che abbia un certo rango. Il caso induttivo della funzione quick sort è il seguente:

$$T(n) = T(q) + T(n-q) + \Theta(n)$$

Lo scopo è ricavare $T_{medio}(n)$. Si prende adesso in esame una chiamata ricorsiva qualunque; si supponga che la scelta casuale del pivot sia tale che il pivot abbia rango x. Questo vuol dire che si avranno due sottosequenze, una di dimensione q_x e l'altra di dimensione $n-q_x$. Il problema è adesso ridotto all'ordinamento delle due sottosequenze; se si vuole calcolare il tempo medio del quick sort per ordinare queste sequenze si può sommare il tempo per la prima chiamata ricorsiva e quanto tempo si impiega mediamente per ordinare una sequenza q_x e quanto tempo si impiega per ordinare una sequenza $n-q_x$. Il tempo medio di una sequenza si ottiene sommando tutte le possibili scelte del rango e dividerle per quante scelte sono possibili. Quindi:

$$T_{medio}(n) = \frac{1}{n}\sum_{r=1}^{n}\left(T_{medio}(q_r) + T_{medio}(n-q_r) + \Theta(n)\right)$$

Nota: la precedente riguarda il caso induttivo (se $n > 1$), perché se $n = 1$ il tempo è costante.

4.3.9 Risoluzione dell'equazione di ricorrenza del tempo medio

Quello che si vuole dimostrare è che la soluzione di questa equazione è di tipo $nlogn$, cioè che

$$T_{medio}(n) = O(nlog_2 n) \Leftrightarrow \exists c, n_0 > 0 / \forall n \geq n0, T_{medio}(n) \leq c \cdot nlog_2 n$$

Questo perché si è già a conoscenza che il caso migliore è proprio $nlog_2 n$, quindi non può essere migliore di questo. n rappresenta la lunghezza di una sequenza che è un valore intero, e ricordando quanto detto nelle precedenti lezioni, si può procedere ad una *dimostrazione per induzione*. La scrittura è infatti del tipo $\forall n \geq n_0, P(n)$.
Il primo passaggio è rimuovere q_r esplicitando la relazione che c'è tra r e q_r. Scorporiamo il caso in cui $r = 1$ dando per scontato quindi che r sia maggiore di 1. Si ottiene:

$$\sum_{r=2}^{n} \left(T_{medio}(q_r) + T_{medio}(n - q_r) + \Theta(n)\right) = \sum_{q=1}^{n-1} \left(T_{medio}(q) + T_{medio}(n - q) + \Theta(n)\right)$$

Nel caso in cui $r = 1$

$$\frac{1}{n}\left[T_{medio}(1) + T_{medio}(n-1) + \Theta(n) + \sum_{q=1}^{n-1} \left(T_{medio}(q) + T_{medio}(n - q) + \Theta(n)\right)\right]$$

Sappiamo il seguente:

$$T_{medio}(1) = \Theta(1) \tag{10}$$
$$T_{medio}(n-1) = O(n^2) \tag{11}$$

La somma dei tempi delle funzioni sarà una somma al peggio quadratica ($O(n^2)$). Quindi

$$T_{medio}(n) = O(n^2) + \frac{1}{n}\sum_{q=1}^{n-1} \left(T_{medio}(q) + T_{medio}(n - q) + \Theta(n)\right)$$

Per la proprietà distributiva della somma:

$$T_{medio}(n) = O(n^2) + \frac{1}{n}\sum_{q=1}^{n-1} \left(T_{medio}(q) + T_{medio}(n - q)\right) + \frac{1}{n}\sum_{q=1}^{n-1} \Theta(n)$$

Si può osservare che

$$\frac{1}{n}\sum_{q=1}^{n-1}\Theta(n) = \Theta(n)$$

$$O(n) + \Theta(n) = \Theta(n)$$

Quindi:

$$T_{medio}(n) = \Theta(n) + \frac{1}{n}\sum_{q=1}^{n-1}(T_{medio}(q) + T_{medio}(n-q))$$

Espandendo le somme risulta:

$$T_{medio}(q) = T_{medio}(1) + T_{medio}(2) + \cdots + T_{medio}(n-1)$$

$$T_{medio}(n-q) = T_{medio}(n-1) + T_{medio}(n-2) + \cdots + T_{medio}(2) + T_{medio}(1)$$

In definitiva

$$T_{medio}(n) = \Theta(n) + \frac{2}{n}\sum_{q=1}^{n-1}(T_{medio}(q))$$

A questo punto si è pronti per iniziare la dimostrazione per induzione.

Dimostrazione del caso base Il caso base si può scegliere arbitrariamente, perché è sufficiente che il caso base esista (cioè che ci sia una costante strettamente positiva per cui la proprietà è vera). Poiché c'è un logaritmo, la costante dovrà essere ≥ 2, quindi si sceglie $n_0 = 2$. Si vuole dimostrare cioè

$$\exists c \geq 0 / \forall n \geq 2, T_{medio}(n) \leq c \cdot nlog_2 n$$

Caso base con $n = 2$:

$$T_{medio}(2) \leq c \cdot 2 \cdot log_2 2 = 2c$$

Ma cos'è $T_{medio}(2)$?

$$T_{medio}(2) = \Theta(2) + \sum_{q=1}^{1} T_{medio}(q) = \Theta(2) + \Theta(1) = k_1 + k_2$$

La dimostrazione si riduce a

$$k1 + k2 \leq 2c$$

Affinché ciò sia vero, basta scegliere una costante $c \geq \frac{k1+k2}{2}$, che è il *primo vincolo del caso base.*

Dimostrazione del caso induttivo Dato $k > 2$, bisogna ricavare $T_{medio}(k)$.

$$T_{medio}(k) = \Theta(k) + \frac{2}{k}\sum_{q=1}^{k-1} T_{medio}(q)$$

Per ipotesi, si ha che $\forall 1 \leq q \leq k-1, T_{medio}(q) \leq cqlog_2 q$ (cioè che la proprietà sia vera per tutti gli interi strettamente minori di questo). Ciò implica che se vale questa disuguaglianza per tutti i valori strettamente minori di k, si può dire che

$$T_{medio}(k) = \Theta(k) + \frac{2}{k}\sum_{q=1}^{k-1} T_{medio}(q) \leq \Theta(n) + \frac{2c}{n}\sum_{q=1}^{k-1}(qlog_2 q)$$

Si deve cercare di dimostrare che

$$\Theta(n) + \frac{2}{n}\sum_{q=1}^{k-1}(qlog_2 q) \leq \mathbf{cklog_2 k}$$

Si può dimostrare il seguente:

$$\sum_{q=1}^{k-1} qlog_2 q \leq \frac{k^2}{2}log_2 k - \frac{k^2}{8}$$

Dando per scontata la precedente, si può dire che:

$$\Theta(k) + \frac{2}{n}\sum_{q=1}^{k-1}(qlog_2 q) \leq \Theta(k) + \frac{2c}{4}\cdot\left(\frac{k^2}{2}log_2 k - \frac{k^2}{8}\right)$$

Per la proprietà transitiva, si ha quindi:

$$T_{medio}(k) \leq \Theta(k) + \frac{2c}{4}\left(\frac{k^2}{2}log_2 k - \frac{k^2}{8}\right) = \mathbf{cklog_2 k} - \frac{ck}{4} + \Theta(k)$$

Quindi:

$$cklog_2 k - \frac{ck}{4} + c_1 \cdot k$$

Si risolve la seguente:

$$-c\frac{k}{4} + c_1 k \leq 0 \Leftrightarrow c_1 k \leq \frac{ck}{4} \Leftrightarrow c \geq 4c_1$$

Tale costante rende quindi vero:

$$cklog_2 k - \frac{ck}{4} + c_1 k \leq cklog_2 n$$

I vincoli definitivi sono quindi:

1. $c \geq \frac{c1+c2}{2}$
2. $c \geq 4c_1$

Le costanti necessarie esistono e quindi la dimostrazione è verificata. ■

Ulteriore dimostrazione Per permettere di concludere la dimostrazione, prima si è dato per scontato che la seguente fosse vera:

$$\sum_{q=1}^{k-1} qlog_2 q \leq \frac{k^2}{2} log_2 k - \frac{k^2}{8}$$

Si può osservare che:

$$\forall 1 \leq q \leq k-1, log_2 q \leq log_2 k$$

Quindi la precedente sommatoria sarà minore o uguale della seguente:

$$\sum_{q=1}^{k-1} qlog_2 k = log_2 k \cdot \sum_{q=1}^{k-1} q = log_2 k \cdot \left(\frac{k(k-1)}{2}\right) = \frac{k^2}{2} log_2 k - \frac{klog_2 k}{2}$$

L'approssimazione tuttavia non è precisa - l'approssimazione è eccessiva. Cosa si può fare per migliorarla? Si divide la sommatoria iniziale in due parti:

$$\sum_{q=1}^{k-1} qlog_2 q = \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} q_l og 2q + \sum_{q=\lceil \frac{k}{2} \rceil}^{k-1} qlog_2 q$$

Si può osservare che:

$$\sum_{q=\lceil \frac{k}{2} \rceil}^{k-1} qlog_2 q \leq log_2 k \cdot \sum_{q=\lceil \frac{k}{2} \rceil}^{k-1} q$$

E che:

$$\forall 1 \leq q \leq \left\lceil \frac{k}{2} \right\rceil - 1, log_2 q \leq log_2 \frac{k}{2}$$

Quindi:

$$\sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} q_l og 2q \leq log_2 \frac{k}{2} \cdot \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} q$$

Si ottiene:

$$log_2 k \cdot \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} q + log_2 k \cdot \sum_{q=\lceil \frac{k}{2} \rceil}^{k-1} q - \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} q$$

Sapendo che

$$- \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} q \leq - \sum_{q=1}^{\frac{k}{2}-1} q$$

In definitiva, l'equazione originale è minore o uguale a

$$log_2 k \cdot \sum_{q=1}^{k-1} q - \sum_{q=1}^{\lceil \frac{k}{2} \rceil - 1} q = log_2 k \left(\frac{k^2 - k}{2} \right) - \sum_{q=1}^{\frac{k}{2}-1} q = \frac{k^2}{2} log_2 k - \frac{k log_2 k}{2} - \frac{\frac{k}{2}(\frac{k}{2} - 1)}{2}$$

$$= \frac{k^2}{2} log_2 k - \frac{k log_2 k}{2} - \frac{k^2}{8} + \frac{k}{4}$$

Quindi:

$$- \frac{k log_2 k}{2} + \frac{k}{4} \leq 0$$

E' quindi dimostrato che:

$$\sum_{q=1}^{k-1} q log_2 q \leq \frac{k^2}{2} log_2 k - \frac{k^2}{8}$$

Nota: vorrei sapere se il me futuro capirà qualcosa. ■

November 2, 2018

5 Limite d'efficienza di un problema

Tra tutti gli algoritmi esistenti o immaginabili, qual è quello che minimizza il caso peggiore o minimizza il caso medio? Si tratta di capire qual è la complessità intrinseca di un problema, non di uno specifico algoritmo. Fare ciò permette di capire quando è stata raggiunta la migliore soluzione possibile per un dato problema. Si vuole determinare una stima della complessità del problema sia nel caso peggiore che nel caso medio, e permetterà di capire se tra gli algoritmi studiati fino ad ora se almeno uno è asintoticamente ottimale. Fino ad ora si è sempre parlato di $nlog_2n$ come limite minimo del tempo di esecuzione di qualsiasi algoritmo di ordinamento. Come si dimostra? In questo caso non si parte da un algoritmo ma da un problema, quindi le tecniche viste precedentemente non sono valide—è quindi necessario introdurre un livello di astrazione nell'analisi. Come si può organizzare un insieme di confronti gli esiti dei quali permetterebbero di determinare la corretta sequenza o permutazione che rappresenta l'ordinamento della sequenza in input? Ogni algoritmo di ordinamento può essere visto come una *politica di confronti*, ogni algoritmo ciò decide quando e come fare i confronti. Gli scambi di fatto servono a tenere traccia dell'esito di questi confronti–non sono necessari realmente per risolvere il problema.

Esempio Si supponga di avere una sequenza di tre elementi k_1, k_2, k_3. Per capire qual è la permutazione corretta corrispondente alla sequenza ordinata (che è 1 in $3! = 6$) è chiaramente necessario conoscere i valori delle costanti. Si supponga ora che, in seguito ad un confronto, si scopre che $k1 < k2$ e che $k2 < k3$. Una volta saputo l'esito di questi confronti, si conosce la permutazione corretta—ovvero quella in ingresso.
Se invece $k_1 > k_2$ e $k_2 < k_3$, k_2 deve venire prima di k_1 e k_2 deve venire prima di k_3, ma non si conosce effettivamente la posizione di k_3. Pertanto, sarà necessario un ulteriore confronto tra k_1 e k_3 per conoscere la posizione corretta—se infatti risultasse $k_1 < k_3$, la permutazione corretta sarebbe k_2, k_1, k_3. Il confronto $k_2 < k_3$ non è stato utile in questo caso, ma non si può sapere con certezza quando un confronto sarà utile oppure no. È questo il motivo per cui **il problema dell'ordinamento non si può risolvere in tempo lineare**.
Se si potessero rappresentare questi scambi in strutture facili da analizzare, allora si può riuscire a capire il minimo numero massimo di confronti per risolvere il dato problema. Ottenendo ciò, si ha infatti una stima del numero di confronti nel caso peggiore

che il miglior algoritmo di ordinamento dovrà necessariamente fare per risolvere il problema. Questo ragionamento si applica anche al caso medio. Si cercherà quindi di studiare lo spazio di questi confronti. La struttura di confronti si può rappresentare naturalmente con un albero, dove ogni confronto corrisponde ad un nodo di un albero binario. Riprendendo l'esempio di prima: Data una sequenza in arrivo k_1, k_3, k_4, k_2,

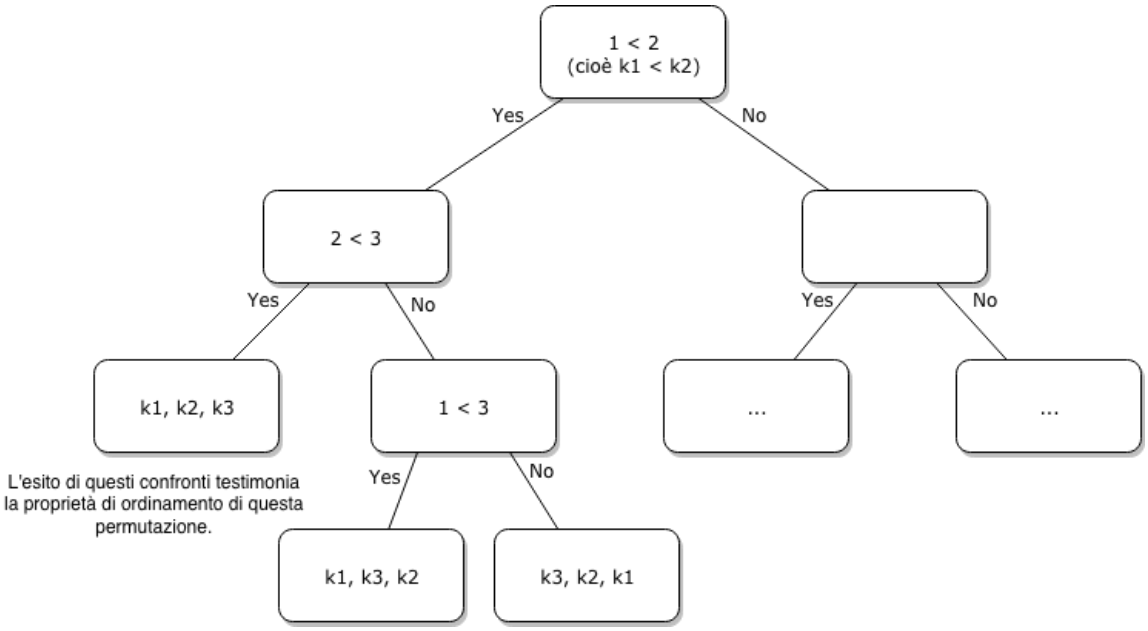


Figure 10: Rappresentazione dei confronti necessari per ordinare una sequenza di 3 elementi. *NB: c'è un errore nell'ultima foglia a destra, dovrebbe essere* k_3, k_1, k_2. *Vedere albero successivo.*

nel percorso dalla radice a questa foglia sarà necessario almeno un confronto tra 1 e 3, tra 3 e 4 e tra 4 e 2. Se non ci fosse il confronto tra k_4 e k_3, come si farebbe a distinguere tra k_1, k_3, k_4, k_2 e k_1, k_4, k_3, k_2. Ogni confronto 'partiziona' l'insieme delle possibili sequenze in insiemi compatibili con quel confronto–discrimina, cioè, le possibili permutazioni. Questo tipo di alberi viene indicato come **"albero di decisione"**, le cui foglie sono alberi binari i cui i nodi interni hanno tutti grado due e devono avere un numero di foglie di almeno $n!$. Riassumendo, le caratteristiche di un albero di decisione di un albero di decisione di ordine n sono:

1. È un albero binario.
2. Numero di foglie minimo di $n!$.

3. I nodi interni hanno grado 2.

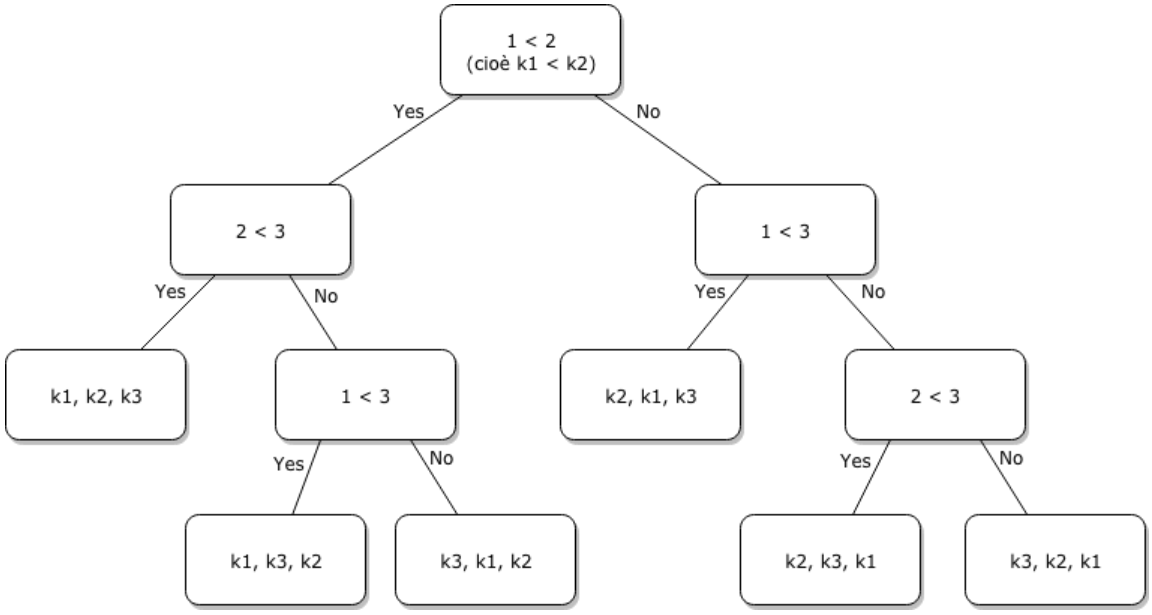
4. Data una sequenza $k_{i_1}, k_{i_2}, k_{i_3}, \ldots, k_{i_n}$, si ha che $\forall z \in [1 \ldots n-1], \exists$ un nodo interno che confronta k_{i_z} e $k_{i_{z+1}}$. Da ciò deriva che devono esserci almeno n nodi in quanto sono necessari almeno n confronti, e che l'altezza non può essere minore di n.

L'altezza di un albero di decisione (cioè la lunghezza del percorso più lungo) permette di ricavare il *massimo numero di confronti* che l'albero richiede per arrivare ad una specifica soluzione—è, cioè, il *massimo lavoro possibile*. Se esistesse una relazione tra ogni algoritmo di ordinamento fissato n:

$$\text{algoritmo di ordinamento} \overset{n}{\leftrightarrow} \text{albero di decisione di ordine } n$$

L'altezza dell'albero permette di determinare qual è il caso peggiore di un algoritmo. Nel caso dell'*insertion sort*, ci si aspetta infatti che l'altezza dell'albero sia quadratica, perché è tale il tempo di esecuzione dell'algoritmo nel caso peggiore. L'astrazione mancante è proprio questa, e permette di associare ad ogni algoritmo una classe di alberi di decisione che contiene uno specifico albero di decisione per ogni dimensione, ed è generalizzabile per ogni algoritmo di ordinamento, uno per ogni ordine.
Dalla proprietà **(4)** degli alberi di decisione, si determina che un algoritmo di ordinamento *non può avere tempo migliore di lineare*, in quanto sono necessari almeno $n-1$ confronti.

5.1 Esempio: albero di decisione dell'insertion sort con $k = 3$



Listing 10: Implementazione dell'"insertion sort"

```
1  InsertionSort(A, n)
2      for j = 2 to n do
3          x = A[j]
4          i = j - 1
5          while (i > 0 && A[i] > x)
6              A[i + 1] = A[i]
7              i = i - 1
8          A[i + 1] = x
```

Data una sequenza di esempio $6, 5, 3$, i parte da 6 e j da 5. Nella prima iterazione, vengono confrontati k_1 e k_2, e la sequenza diventa $5, 6, 3$. In quella successiva, vengono confrontati k_1 e k_3 e k_2 e k_3, e la sequenza diventa $3, 5, 6$.

5.2 Analisi del limite inferiore asintotico per il caso peggiore

Dato un albero di decisione generico, si può procedere a ricavare il limite inferiore asintotico per il caso peggiore. Per ogni albero di decisione, l'altezza è il caso peggiore dell'algoritmo. Dato un ordine n, si vuole capire qual è la minima altezza che può

avere un albero di decisione di ordine n—cioè, il miglior caso peggiore possibile. Si tratta di un albero binario di cui si conosce il numero di foglie ($N_{foglie} = n!$), e poiché si tratta di un albero binario, si ha:

$$\forall \text{ albero binario di altezza } h, N_{foglie} \leq 2^h \Leftrightarrow n! \leq 2^h$$

Non esiste quindi un albero binario che abbia più foglie di 2^h. Applicando i logaritmi, si ricava:

$$h \geq log_2(n!)$$

L'altezza di un albero di decisione deve quindi essere **almeno** $log_2(n!)$, anche nel caso migliore. Quanto vale asintoticamente questo valore?

5.2.1 Analisi asintotica di $log_2(n!)$

Con l'approssimazione di Stirling si ha che:

$$n! \geq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Con la proprietà di monotonicità del logaritmo, si ha che:

$$\begin{aligned} log_2(n!) \geq log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) &= log_2(\sqrt{2\pi n}) + log_2\left(\frac{n}{e}\right)^n \\ &= \frac{1}{2}log_2 n + \frac{1}{2}log_2(2\pi) + n(log_2 n - log_2 e) \\ &= \Theta(nlog_2 n) \end{aligned}$$

Quindi:

$$\begin{aligned} log_2(n!) \geq \Theta(nlog_2 n) &\Rightarrow \\ h \geq log_2(n!) \geq \Theta(nlog_2 n) &\Rightarrow \\ h \geq \Theta(nlog_2 n) &= \Omega(nlog_2 n) \end{aligned}$$

5.3 Analisi del limite inferiore asintotico per il caso medio

Per calcolare il caso medio, è necessario valutare la lunghezza del percorso esterno dell'albero, in quanto si vuole ottenere la media delle esecuzioni che arrivano fino alla fine (le cosiddette esecuzioni 'monche' non sono d'interesse in questo specifico caso).

$$T_{medio}(n) = \frac{\text{percorso esterno}(n)}{n!}$$

L'unico modo per minimizzare questa quantità è minimizzare il numeratore (fissato un $n!$), quindi si vuole cercare l'albero di decisione che minimizza il percorso esterno. L'albero di decisione che minimizza il percorso esterno è essenzialmente un *albero completo*. Non potrà mai essere un *albero pieno* perché non esistono *alberi di decisione pieni* (perché un albero pieno ha per forza un numero di foglie che un albero di decisione non può avere). Perché un albero completo risponde alla richiesta di un albero di decisione che minimizza il percorso esterno? Date due foglie qualsiasi e messe in una profondità diversa da quella in cui si trovano (l'unico modo per farlo è andando più in basso rispetto a dove si trovano, creando tre livelli di profondità), il percorso esterno del nuovo albero è peggiore del precedente, e l'unico modo per renderlo migliore è eliminare delle foglie, ma non è possibile farlo mantenendolo un albero di decisione. Si indica con N_h il numero di foglie a profondità h, e N_{h-1} il numero di foglie a profondità $h-1$. Si ha che la somma dei due livelli restituisce il numero di foglie:

$$N_h + N_{h-1} = N!$$

Inoltre:

$$N_h + 2N_{h-1} = 2^h$$

Risolvendo il sistema di equazioni, si ottiene:

$$N_{h-1} = 2^h - N! \tag{12}$$

$$N_h = N! - 2^h + N! = 2N! - 2^h \tag{13}$$

La lunghezza del percorso esterno sarà quindi:

$$P.E. = h \cdot N_h + (h-1)N_{h-1}$$

Ritornando alla formula ricavando in precedenza, ricordando che in un albero completo di cui si conosce il numero di foglie k, l'altezza è $h = \lceil log_2 k \rceil$:

$$\begin{aligned} T_{medio}(N) &= \frac{\text{percorso esterno}(N)}{n!} \\ &= \frac{h \cdot 2N! - h \cdot 2^h + (h-1)(2^h - N!)}{N!} \\ &= \frac{h \cdot 2N! - h \cdot 2^h + h \cdot 2^h - hN! - 2^h + N!}{N!} \\ &= \frac{hN! - 2^h + N!}{N!} = h + 1 - \frac{2^h}{N!} \\ &= \lceil log_2 N! \rceil + 1 - \frac{2^{\lceil log_2 N! \rceil}}{N!} \end{aligned}$$

Eliminando il tetto (che asintoticamente non porta differenze), si ha:

$$T_{medio}(N) \approxeq log_2(N!) = \boxed{\Omega(Nlog_2N)}$$

Da questo risultato si conclude che tra gli algoritmi che abbiamo visto gli unici ottimali nel caso peggiore sono *merge sort e heap sort*, mentre per il caso medio sono ottimali *merge sort*, *heap sort* e *quick sort*.

November 6, 2018

6 Strutture dati

NB: ho perso i primi 15-20 minuti della lezione di introduzione sulle strutture dati e l'introduzione non è pertanto qui riportata.

6.1 Struttura astratta: insieme (dinamico)

Le operazioni definibili su in insieme dinamico sono (dato un elemento k ed un insieme I):

- **Operazione di ricerca**: $\boxed{k \in I}$
- **Operazione di inserimento/unione**: $I_1, I_2 \to \boxed{I_1 \cup I_2}$
- **Operazione di sottrazione/cancellazione**: $I_1, I_2 \to \boxed{I_1 \setminus I_2}$
- **Ricerca/Estrazione elemento minimo/massimo**

Si tratta di una prima ipotesi di un tipo di dato astratto per la rappresentazione di insiemi. Si può immaginare di utilizzare un array come rappresentazione di una collezione di dati, tuttavia l'esecuzione delle operazioni precedenti potrebbe risultare costosa con un'implementazione basata su un array. Se l'insieme di elementi che si vuole mantenere nella collezione è ordinabile (quindi è definita una relazione d'ordine), allora si potrebbe usare un array ordinato per rendere più efficiente l'operazione di ricerca generica e quella del minimo/massimo—alcune operazioni diventerebbero quindi abbastanza efficienti, mentre altre abbastanza inefficienti (come quelle di inserimento o sottrazione). Compromessi simili si otterrebbero usando una lista, sia nella variante ordinata che non ordinata.
Si possono prendere in esame altre strutture dati che non siano lineari, come strutture ramificate–un esempio evidente è *un albero*, in cui ci sono due dimensioni: una orizzontale e una verticale, ed è possibile muoversi lungo entrambe. Esistono, tuttavia, altre strutture che premesse alcune proprietà permettono di ottimizzare sia le operazioni di ricerca che quelle di modifica—proprietà che con le strutture lineari non esiste, in quanto–come visto–una volta ottimizzata una classe di operazioni l'altra subiva un rallentamento.

6.2 Lista

Una lista è un insieme di elementi L che può essere o l'insieme vuoto $\emptyset$ o un insieme che è partizionabile in due sottoinsiemi disgiunti, uno dei quali contiene un solo elemento, detto *testa della lista*, seguito da L', che è un'altra lista (e mantiene le stesse proprietà).

$$L = \emptyset \text{ oppure} \tag{14}$$
$$L = \{x\} \cup L'$$
$$= \{x\} \rightarrow L' \tag{15}$$

Si tratta della definizione di *lista non ordinata*.

6.2.1 Lista ordinata

Una lista ordinata mantiene le stesse proprietà di una lista non ordinata, con un vincolo di ordinamento per garantire che ogni decomposizione sia ordinata:

$$L = \emptyset \text{ oppure}$$
$$L = \{x\} \rightarrow L' \wedge x \leq y \ \forall y \in L'$$

6.2.2 Definizione generale

Una lista è una sequenza di oggetti, ciascuno dei quali contiene un dato (es. x, y, z) con arbitraria atomicità. Affinché gli elementi possano formare una lista, sono necessari dei collegamenti, realizzabili ad esempio tramite puntatori:

$$\boxed{x|\cdot} \rightarrow \boxed{y|\cdot} \rightarrow \boxed{z|\cdot}$$

La struttura rappresentate precedentemente è quindi del tipo $\boxed{\text{dato}|\text{next}}$, e si accede ai campi con L−>next e L−>key.

6.2.3 Implementazione operazioni

Tenendo a mente le proprietà 15 e 14, si può procedere ad implementare le operazioni sulla lista.

Listing 11: Implementazione dell'algoritmo di ricerca in una lista

```
1 Ricerca(L, k)
2   if L = NULL then
```

```
3       return NULL
4     else if L->key = k then
5       return L
6     else
7       return Ricerca(L->next, k)
```

Listing 12: Implementazione dell'algoritmo di cancellazione in una lista (ricorsivo)

```
1  Cancella(L, k)
2    if L = NULL then
3      ret = NULL
4    else if L->key = K then
5      ret = L->next
6      free(L)
7    else
8      L->next = Cancella(L->next, k)
9      ret = L
10   return ret
```


Analisi della complessità dell'algoritmo di cancellazione Lo stack di chiamate ricorsive è uno stack di tipo LIFO (*last in, first out*). Nel caso peggiore sarà necessario avere in memoria contemporaneamente tanti record di attivazione quant'è il numero massimo di chiamate annidate che possono essere contemporaneamente in esecuzione. La più lunga catena di chiamate ricorsive che si possono eseguire si ottiene quando la chiave da eliminare non è presente—si dovranno cioè avere tanti record di attivazione quanti elementi nella lista. Esiste un algoritmo iterativo che risolve lo stesso problema senza utilizzare la stessa quantità di spazio.

Digressione: ricorsione vs. iterazione Un algoritmo ricorsivo è meno efficiente di uno implementato iterativamente? In generale esistono forme ricorsive che non richiedono l'utilizzo dello stack. Ad esempio, si provi ad immaginare un algoritmo ricorsivo tale per cui ogni chiamata ricorsiva è immediatamente seguita da un'uscita dalla procedura—cioè, l'ultima istruzione che è eseguita prima dell'uscita è una chiamata ricorsiva. Nel caso dell'algoritmo di cancellazione, questa condizione **non è verificata** in quanto dopo la chiamata ricorsiva viene effettuata un'assegnazione che accede al valore di L. È proprio la lettura di L (e non la scrittura, che non genera problemi particolari—infatti, L=NULL non richiede accesso al precedente valore di

L) ciò che richiede l'utilizzo dei record di attivazione e quindi genera l'inefficienza della ricorsività. In generale, non è vero che ogni algoritmo iterativo non ha bisogno di memoria aggiuntiva. Non è questo infatti il caso del quick sort. Utilizzare un algoritmo ricorsivo, poiché richiede il costo aggiuntivo dello stack, è quindi sconveniente in questo caso? La risposta è no, in quanto è possibile implementare una soluzione ricorsiva che non richieda memoria aggiuntiva. Se è possibile invece costruire un algoritmo ricorsivo che rispetta il vincolo specificato precedentemente, allora è possibile implementare tale algoritmo ricorsivamente senza necessità di stack, perché è sempre possibile sovrascrivere le variabili locali in quanto non serviranno mai.

Ricorsione in coda Si definisce ricorsione in coda un tipo di ricorsione dove ogni chiamata ricorsiva è l'ultima istruzione della funzione. Se è possibile risolvere un problema iterativamente senza occupare spazio aggiuntivo, è ragionevole supporre che si possa sviluppare un algoritmo ricorsivo in coda.

Ottimizzazione dell'algoritmo di cancellazione L'assegnazione L->next = Cancella(L->next, k) è necessaria e non eliminabile, ma non deve sempre avvenire. Si può procedere a riscrivere l'algoritmo in forma ottimizzata, sia nella sua implementazione interna che esterna:

Listing 13: Implementazione della parte interna dell'algoritmo di cancellazione (ricorsivo e ottimizzato)

```
1 CancellaRc(L, P, k) {P rappresenta il predecessore di L
    nella lista}
2   if L != NULL then
3     if L->key = k then
4       if P != NULL then
5         P->next = L->next
6         free(L)
7     else
8       CancellaRc (L->next, L, k)
```

Listing 14: Implementazione della parte esterna dell'algoritmo di cancellazione (ricorsivo e ottimizzato)

```
1 Cancella(L, k)
2   if L != NULL then
3     if L->key = k then
4       tmp = L
```

```
5         L = L->next
6         free(tmp)
7       else
8         CancellaRc (L, NULL, k)
9     return L
```

La tecnica utilizzata per ottenere la *tail recursion* è esattamente la stessa che si utilizzerebbe implementando l'algoritmo iterativamente.
November 9, 2018

6.2.4 Duplicazione di una lista

Adesso si può procedere ad implementare un algoritmo per duplicare una lista.

Listing 15: Implementazione di un algoritmo per la duplicazione di una lista

```
1  Dup(L)
2    Ldup = NULL
3    if L != NULL then
4      Ldup = alloc_list_element() {generica funzione per
           creare un elemento vuoto}
5      Ldup->key = L->key
6      Ldup->next = Dup(L->next)
7    return Ldup
```

Questa funzione non è *ricorsiva in coda*, perché fa riferimento a valori che è necessario salvare e ripristinare man mano che lo stack di chiamate procede. Tutti i vari Ldup richiedono di essere salvati. Per rendere la funzione ricorsiva in coda, bisogna evitare l'assegnazione al ritorno della chiamata ricorsiva. I collegamenti devono quindi essere fatti prima del ritorno della chiamata ricorsiva.

Listing 16: Implementazione di un algoritmo per la duplicazione di una lista (ottimizzato)

```
1  Dup(L)
2    Ldup = NULL
3    if L != NULL then
4      {si procede a creare la prima copia -- quella
           problematica da fare ricorsiva}
5      Ldup = alloc_list_element()
6      Ldup->key = L->key
```

```
7       {per copiare tutto il resto, si fa una chiamata all'
           algoritmo ricorsivo che copia tutta la coda di L
           avendo come ultimo predecessore Ldup}
8       DupRc(L->next, Ldup) {Ldup e' l'ultimo elemento copiato}
9     return Ldup
10
11  DupRc(L, P) {P = predecessore della copia di L}
12    if L != NULL then
13      Ldup = alloc_list_element()
14      Ldup->key = L->key
15      P->next = Ldup
16      DupRc(L->next, Ldup)
17    return
```

Questa variante dell'algoritmo è ricorsiva in coda ed è ottimizzata. Ancora una volta, il modo in cui il problema è stato risolto è esattamente lo stesso che si sarebbe usato implementando l'algoritmo iterativamente.

6.3 Alberi

6.3.1 Il problema della ricerca

Le strutture lineari sono troppo rigide per garantire una ricerca efficiente senza aggiungere vincoli restrittivi. Le strutture ad albero, invece, non presentano la stessa limitazione, ma è necessario trovare un sistema per fornire una relazione d'ordine, rendendo una struttura ad albero effettivamente come un array.

6.3.2 Albero k-ario

Un albero k-ario si può definire come:

- $T = \emptyset$ *oppure*
- Partizionabile in $k + 1$ sottoinsiemi tale che
  - Uno contiene un solo elemento (radice) *oppure*
  - k sottoinsiemi alberi k-ari (sottoalberi)

Ogni albero può sempre essere trasformato opportunamente in un albero binario equivalente.

6.3.3 Ricerca all'interno di un albero

Non avendo alcuna informazione aggiuntiva su un albero, l'elemento ricercato potrebbe trovarsi in qualunque posizione dell'albero. È quindi ragionevole supporre che per trovare l'elemento sia necessario esplorare l'albero. Ci si concentrerà su una versione più semplice del problema della ricerca, il problema dell'esplorazione.

6.3.4 Algoritmi di visita

Supponendo di avere un albero, l'oggetto di riferimento sarà composto dai campi sx e dx per riferirsi ai figli nelle rispettive posizioni, oltre ad un campo con il dato.

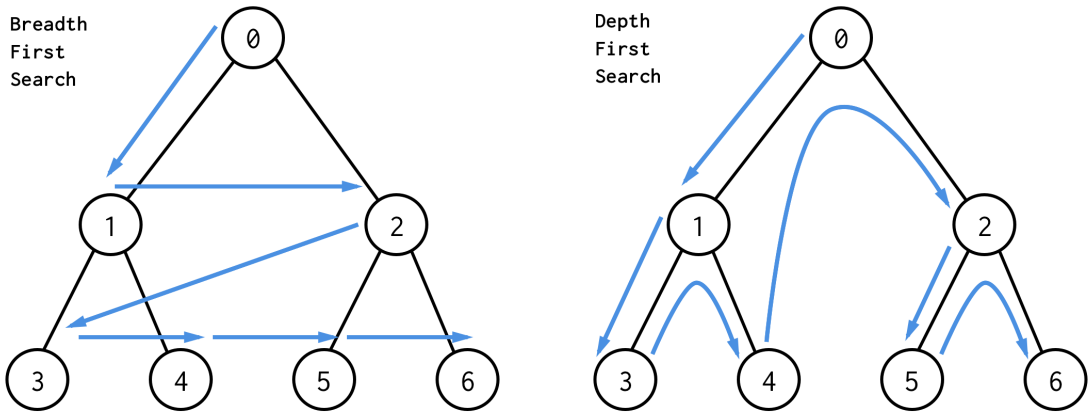


Figure 11: Ricerca in profondità vs. ricerca per ampiezza

Visita in profondità Un primo tipo di algoritmo di visita è quello in profondità, dove si esplora l'albero arrivando ai nodi più in profondità e continuando.

Tipi di visita in profondità Esistono tre tipi di visita in profondità:

- Visita preorder
- Visita inorder
- Visita postorder

L'implementazione ricorsiva di questi algoritmi è immediata.

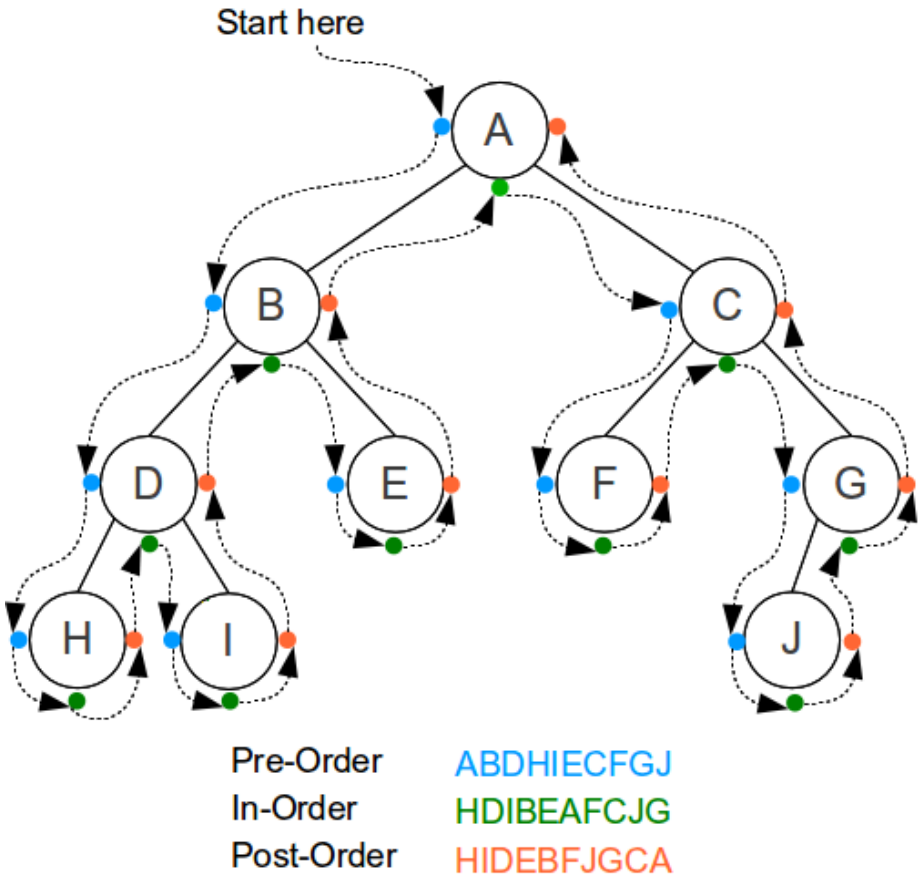


Figure 12: Tipi di visita ("traversal") di un albero

Listing 17: Implementazione della visita di un albero

```
1  Visita(T)
2    if T != NULL then
3      {esplora nodo T (qui e' visita pre-order)}
4      Visita(T->sx)
5      {esplora nodo T (qui e' visita in-order)}
6      Visita(T->dx)
7      {esplora nodo T (qui e' visita post-order)}
```

Ogni tipo di visita ha applicazioni diverse. Si può osservare che a prescindere dal tipo di visita impiegato, l'algoritmo non può essere ricorsivo in coda, in quanto c'è una chiamata ricorsiva dopo la cui terminazione non si esce immediatamente.

Analisi dei record di attivazione nello stack Supponendo un albero binario coi nodi $[a => [b => [d, c], c => [f, g]]]$. Il numero massimo di record di attivazione presenti contemporaneamente nello stack è 4, che equivale all'altezza dell'albero più 1. L'albero ha 7 nodi, il numero di chiamate ricorsive è 15 e il numero di elementi nello stack è 4. In definitiva, la lunghezza massima assunta dallo stack durante l'esecuzione di questo algoritmo sarà pari all'altezza dell'albero, cioè alla lunghezza del percorso più lungo. Quindi:

$$DimStack(h) = \Theta(h)$$

La dimensione dello stack può variare–relativamente alla forma dell'albero da visitare– da lineare nel caso migliore (ad esempio un albero con tutti i nodi sinistri vuoti) a logaritmico nel caso peggiore (un albero binario con nessun nodo vuoto).

Visita in ampiezza Un altro algoritmo di visita è quello per ampiezza, dove l'algoritmo si muove nell'albero per livelli. Si vorranno visitare prima tutti i nodi di livello 0, poi tutti quelli di livello 1, e così via. Si consulti la figura 11. Il livello di un nodo all'interno dell'albero misura una "distanza" del nodo dalla radice, l'algoritmo prevede quindi di visitare l'albero per distanze crescenti. Per realizzare l'algoritmo si parte dalla radice dell'albero, e si mantiene una coda in cui verranno mantenuti i nodi che devono ancora essere visitati. Inizialmente l'unico nodo in coda sarà la radice, poi si aggiungeranno i figli della radice e si procederà così. Si può osservare che inizialmente in coda c'è il primo livello, poi esattamente il secondo, e rimarrà così per tutti i successivi livelli–permettendo, in effetti, la visita in ampiezza. Qual è il costo in spazio di questo algoritmo?

Analisi dello spazio utilizzato dalla visita in ampiezza La dimensione massima della coda è proprio il numero di foglie. Il costo massimo è strettamente legato al numero massimo di foglie che può avere un albero–in un albero pieno, il numero di foglie è la metà del numero dei nodi, questo implica che nel caso peggiore (quindi in un albero pieno) il costo in spazio dell'algoritmo di visita in ampiezza è lineare sul numero di nodi. Un albero degenere è il caso migliore, e in quel caso l'algoritmo utilizza spazio costante, perché la coda conterrà al più un elemento.
November 13, 2018

Implementazione dell'algoritmo della visita in ampiezza Per sviluppare l'algoritmo della visita in ampiezza, si suppone l'esistenza di una struttura dati coda che fornisca il seguente prototipo:

Listing 18: Prototipo struttura dati "coda"

```
1  {coda first in first out Q}
2  Qnew = Enqueue(Q, k)
3  Qnew = Dequeue(Q)
4  k = First(Q)
```

Si può adesso procedere con l'implementazione dell'algoritmo di visita in ampiezza.

Listing 19: Implementazione della visita in ampiezza di un albero

```
1  Ampiezza(T)
2    Q = EmptyQueue {valore speciale - indica coda vuota}
3    if T != NULL then
4      Q = Enqueue(Q, T)
5      while (Q != EmptyQueue) do
6        E = First(Q)
7        {visita E}
8        for each x in Figli(E) do
9          Q = Enqueue(Q, x)
10       Q = Dequeue(Q)
```

Stima della dimensione della coda Stimare la dimensione della coda ha senso se la coda viene rappresentata con una struttura statica (esempio un array). Con una struttura dinamica—come ad esempio una lista—saperlo è meno critico, tuttavia, dal punto di vista di analisi dei costi computazionali è sensato stimare il costo in termini di spazio in funzione della dimensione dell'albero da visitare. Indicando la dimensione dell'albero con:

$$n = |T|$$

Il caso peggiore in termini di occupazione di spazio si ha quando l'albero è pieno, e quindi il livello che contiene il massimo numero di nodi è il livello delle foglie. In un albero k-ario pieno, come si può stimare la dimensione massima della coda? Le foglie di un albero k-ario pieno di altezza h sono:

$$n_{foglie} = k^h$$

E il numero di nodi è:

$$n_{nodi} = \sum_{i=0}^{h} k^i = \frac{k^{h+1} - 1}{k - 1}$$

La dimensione della coda deve poter essere esprimibile in:

$$|Q| = \lceil c \cdot n \rceil = k^h$$

$$c \cdot n \approx k^h \Rightarrow c \approx \frac{k^h}{n} = \frac{k^h}{\left(\frac{k^{h+1}-1}{k-1}\right)} = \frac{k^h(k-1)}{k^{h+1}-1} = \frac{k^h(k-1)}{k^h\left(k-\frac{1}{k^n}\right)} \Rightarrow c \approx \frac{k-1}{k-\frac{1}{k^h}}$$

Poiché $\frac{1}{k^h}$ è molto piccolo, si può assumere $\frac{1}{k^h} \approxeq 0$ e quindi:

$$c \approx \frac{k-1}{k}$$

Si osserva che in realtà diventa un'uguaglianza, in quanto:

$$\left\lceil \frac{k-1}{k} \cdot \frac{k^{h+1}-1}{k-1} \right\rceil = \left\lceil \frac{k^{h+1}-1}{k} \right\rceil = \left\lceil \frac{k(k^h - \frac{1}{k})}{k} \right\rceil = \left\lceil k^h - \frac{1}{k} \right\rceil = k^h$$

Data la dimensinoe dell'albero la dimensione della coda è sempre:

$$|Q| \leq \frac{k-1}{k} \cdot n$$

Conversione dell'algoritmo di visita in profondità in iterativo Per effettuare la conversione dell'algoritmo in iterativo, l'algoritmo non dovrà soltanto simulare le istruzioni, ma anche la sospensione di una chiamata padre e l'attivazione di una chiamata figlia e la riattivazione di una chiamata sospesa. Uno schema generico per fare ciò è:

Listing 20: Schema generico per conversione algoritmo da ricorsivo a iterativo

```
1  {fase di inizializzazione}
2  while (!terminazione) do
3    {e' necessario effettuare manualmente le "copie" delle
        variabili e l'isolamento dei contesti, effettuate di
        norma naturalmente dalla ricorsione}
4    if (nuova chiamata) then
5      {simula inizio nuova chiamata}
6    else {stato di riattivazione di una vecchia chiamata}
7      {recupero contesto dallo stack}
8      if (Prima chiamata)
9        {effettuare seconda chiamata}
10     else
11       {termina simulazione}
```

Come ultimo prerequisito, si definisce un prototipo di 'stack':

Listing 21: Prototipo struttura dati 'stack'

```
1 {stack S}
2 k = Top(S) {elemento in cima}
3 Snew = Push(S, k) {aggiunta elemento in cima}
4 Snew = Pop(S) {rimozione elemento in cima}
```

Tenendo in mente lo schema precedente, si può procedere con l'implementazione dell'algoritmo in forma iterativa:

Listing 22: Implementazione dell'algoritmo di visita post-order in profondità (iterativo)

```
1  VisitaIter(T)
2    CurrentT = T {il nodo che si sta attualmente visitando}
3    S = EmptyStack {valore speciale per indicare uno stack
         vuoto}
4    LastT = NULL {ultimo valore di T passato, simula l'
         indirizzo di ritorno}
5    {Se CurrentT != NULL, si vuole iniziare una nuova chiamata
         . Se invece S != EmptyStack, si vuole riprendere una
         chiamata precedente}
6    while (CurrentT != NULL || S != EmptyStack) do
7      if (CurrentT != NULL) then
8        {salvo il valore di CurrentT}
9        S = Push(S, CurrentT)
10       {sovrascrivo il valore precedente}
11       CurrentT = CurrentT->sx
12     else
13       {ripristino il valore di CurrentT dallo stack}
14       CurrentT = Top(S)
15       {se il destro e' NULL e si viene dalla sinistra, si
            termina}
16       if (LastT != CurrentT->dx && CurrentT->dx != NULL)
            then
17         {l'unica operazione da fare e' istanziare il
              parametro formale con il valore del parametro
              attuale, e non c'e' nient'altro tra le due
              chiamate ricorsive}
18         CurrentT = CurrentT->dx
```

```
19        else
20          {caso in cui o e' appena terminata una chiamata a
              destra (prima cond.) o il destro e' NULL. In
              entrambi i casi, si simula il resto delle
              istruzioni fino all'exit}
21          {visita CurrentT}
22          LastT = CurrentT
23          S = Pop(S)
24          CurrentT = NULL
25          {si forza una risalita. alla prossima esecuzione del
               while o ci sara' qualcosa in sospeso oppure si
              termina}
```

Se ci fosse una terza chiamata ricorsiva, nell'ultimo **else** servirà un altro **if** che discrimini tra la seconda e la terza chiamata. Lo schema è quindi estendibile.
Nota: durante la lezione il professore ha spiegato le variazioni che subirebbe l'algoritmo in caso di pre-order e in-order, che non ho riportato qui.
November 15, 2018

6.3.5 Algoritmo di ricerca

Nota: sono stato assente durante i primi 20 minuti di lezione. Integrare.

Listing 23: Algoritmo di ricerca in un albero (ricorsivo)

```
1  Find(T, k)
2    elem = NULL
3    if T != NULL then
4      if T->key = k then
5        elem = T
6      else
7        elem = Find(T->sx, k)
8        if elem = NULL then
9          elem = Find(T->dx, k)
10   return elem
```

Per convertire l'algoritmo in forma iterativa, non è necessario uno stack—quest'ultimo viene infatti utilizzato per salvare i parametri di una chiamata prima di effettuare una chiamata ricorsiva, per poter mantenere il contesto. Il valore di ritorno non fa parte del contesto di valutazione della chiamata, ma è un'informazione che viene dal figlio. Si utilizzerà quindi una specifica variabile sovrascritta dalle varie iterazioni e

sarà tale che ogni volta che verrà terminata una chiamata a quella variabile verrà assegnato il valore che si vuole restituire, e al ritorno si andrà a prendere il valore contenuto in quella variabile per metterlo in elem.

Listing 24: Algoritmo di ricerca in un albero (ricorsivo)

```
1  FindIterative(T, k)
2    CurrentT = T
3    LastT = NULL
4    S = EmptyStack
5    Ret = NULL
6    elem = NULL
7    while (CurrentT != NULL || S != EmptyStack) do
8      if (CurrentT != NULL) then
9        elem = NULL
10       if CurrentT->key = k then
11         elem = CurrentT
12         LastT = CurrentT
13         CurrentT = NULL {si forza una terminazione}
14         Ret = elem {simulazione del return}
15       else
16         S = Push(S, CurrentT)
17         CurrentT = CurrentT->sx
18     else {CurrentT e' null quando e' appena terminata una
          chiamata ricorsiva}
19       CurrentT = Top(S)
20       if (LastT != CurrentT->dx && CurrentT->dx != NULL)
           then
21         {questo indica che si e' tornato da una chiamata
             diversa dalla dx}
22         {adesso si sa che il padre si e' fermato alla prima
             chiamata ricorsiva}
23         elem = Ret {simulazione dell'assegnamento di elem}
24         if elem = NULL then
25           CurrentT = CurrentT->dx
26           {non c'e' bisogno di fare push nello stack perche'
               il valore non e' stato tolto}
27         else
28           S = Pop(S)
```

```
29              Ret = elem {ridondante in questo caso ma non
                    sempre}
30              LastT = CurrentT
31              CurrentT = NULL
32          else
33            elem = Ret
34            S = Pop(S)
35            LastT = CurrentT {e' un figlio che sta terminando,
                  in LastT va il suo parametro}
36            CurrentT = NULL
37            Ret = elem
38    return elem
```

6.3.6 Digressione: conversione algoritmo ricorsivo a iterativo di altro tipo

Quick sort Ricordando l'algoritmo ricorsivo di quick sort 8, nello stack andrà sicuramente messo r e q ma p no perché viene utilizzato prima di ogni chiamata ricorsiva e non dopo. Per riconoscere il punto di sospensione, si possono confrontare q e r che come è stato precedentemente dimostrato non saranno mai uguali ($p \leq q < r$).

Listing 25: Implementazione dell'algoritmo di quick sort (iterativo)

```
1  QuickSortIterative(A, p, r)
2    CurrentP = r
3    CurrentR = r
4    StackR = EmptyStack
5    StackQ = EmptyStack
6    while (CurrentP <= CurrentR || StackR != EmptyStack) do
7      if (CurrentP <= CurrentR) then
8        if (CurrentP < CurrentR) then
9          q = Partiziona(A, CurrentP, CurrentR)
10         StackR = Push(StackR, CurrentR)
11         StackQ = Push(StackQ, q)
12         CurrentR = q
13       else
14         LastR = CurrentR
15         CurrentR = CurrentP - 1
16     else
```

```
17          CurrentR = Top(StackR)
18          CurrentQ = Top(StackQ)
19          if (LastR != CurrentR) then
20            CurrentP = q + 1
21          else
22            StackR = Pop(StackR)
23            StackQ = Pop(StackQ)
24            LastR = CurrentR
25            CurrentR = CurrentP - 1
```

Merge sort *Nota: Implementazione mancante, integrare successivamente.*

6.3.7 Rappresentare un'espressione matematica con un albero

Si supponga di voler scrivere un programma che in ingresso riceve un'espressione aritmetica con variabili e un assegnamento per quelle variabili. Ad esempio:

$$E = x_1 \cdot (x_2 - x_3) + x_2 \cdot (x_3 - x_1) \tag{16}$$

E un array (in questo caso di tre elementi) che conterrà i valori delle variabili.

$$A = [5, 7, 2]$$

L'espressione aritmetica è composta da termini atomici e operatori aritmetici, ed è rappresentabile naturalmente tramite un albero binario, rappresentato nella figura 13. Il modo più naturale per eseguire l'espressione è mediante una visita in postorder, in quanto gli altri tipi di visita in questo caso non servirebbero a nulla.
November 16, 2018

6.3.8 Albero binario di ricerca

Un albero binario di ricerca è un albero binario in cui vale la seguente proprietà:

$$\begin{aligned} \forall x \in T : &\forall y \in x \to sx, y \to key \leq x \to key \\ &\forall z \in x \to dx, x \to key \leq z \to key \end{aligned}$$

Questa definizione è anche rappresentabile come

$$max(T \to sx) \leq x$$

$$min(T \to dx) \geq x$$

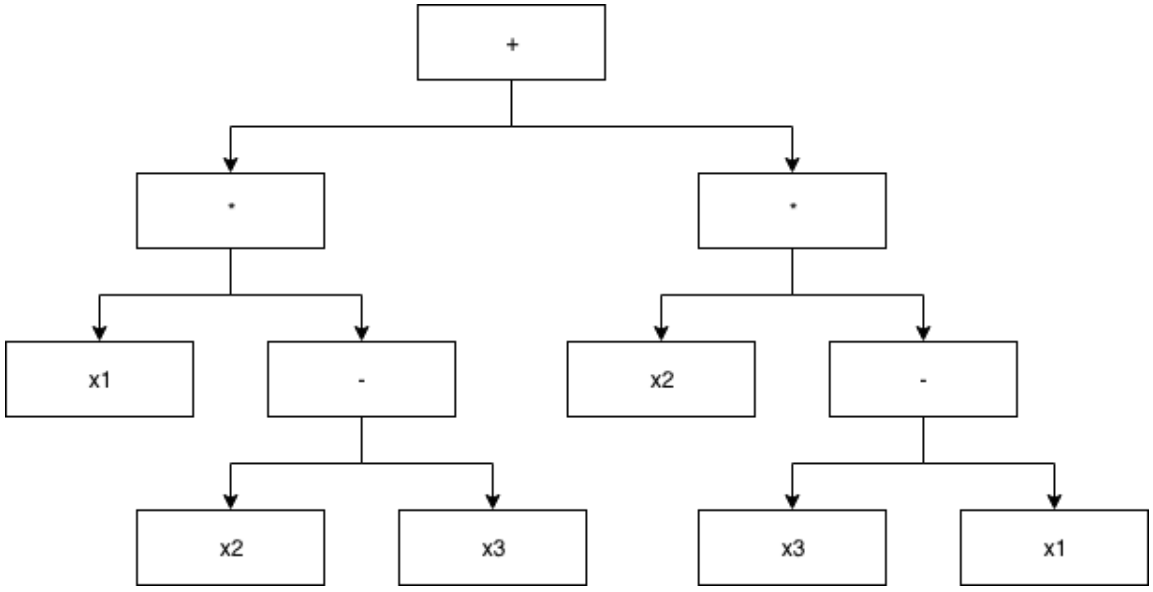


Figure 13: Rappresentazione dell'espressione 16 con un albero binario

Controllare se un albero è di ricerca Per fare ciò, si sviluppa il seguente algoritmo:

Listing 26: Algoritmo per controllare se un albero binario è di ricerca

```
1  CheckABR(T)
2    if (T != NULL) then
3      y = max(T->sx)
4      z = min(T->dx)
5      {le funzioni max e min sono banalmente realizzabili con
           una visita, e la somma del tempo di esecuzione delle
           due funzioni sara' proporzionale al numero di nodi}
6      if (y != NULL && y->key > x->key)
7        return 0
8      if (z != NULL && z->key < x->key)
9        return 0
10     return CheckABR(T->sx) && CheckABR(T->dx)
11   else
12     return 1
```

L'algoritmo nell'operazione di visita ha un costo quadratico e non è pertanto molto efficiente. È possibile ridurre il costo di min e max in questo specifico problema

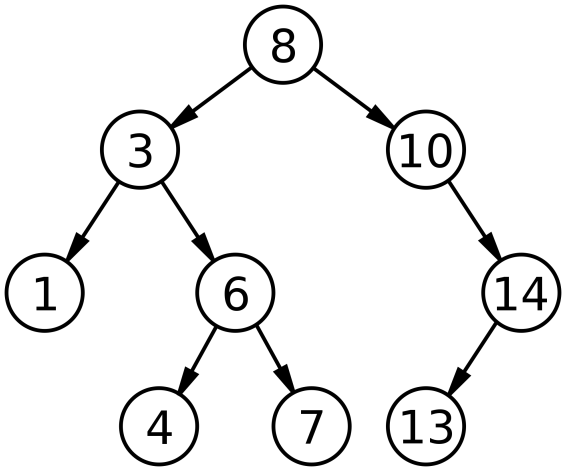


Figure 14: Albero binario di ricerca

riducendolo a $\Theta(n \cdot h)$. L'altezza dell'albero è esprimibile in termini di n (seppur con una certa incertezza), questo vuol dire che qualunque implementazione che faccia uso di *min* e *max* avrà un costo superiore al lineare.

Ottimizzazione dell'algoritmo di controllo di un ABR L'idea alla base dell'algoritmo è che ogni sottoalbero binario deve ricevere il range di valori che devono rispettare i figli.

Listing 27: Algoritmo per controllare se un albero binario è di ricerca (ottimizzato)

```
1  CheckABR(T, min, max)
2    if (T != NULL) then
3      if (min <= T->key <= max) then
4        sx = CheckABR(T->sx, min, T->key - 1)
5        dx = CheckABR(T->dx, T->key + 1, max)
6        return sx && dx
7      else
8        return 0
9    else
10     return 1
```

Questo algoritmo rispetto al precedente oltre ad essere sintatticamente più semplice, ha un costo complessivo corrispondente alla somma dei costi locali, che è determinato sostanzialmente solo da operazioni a tempo costante—di conseguenza, l'algoritmo ha

tempo di esecuzione lineare. La chiamata principale dall'esterno potrebbe avere la seguente forma:

Listing 28: Esempio di chiamata principale all'algoritmo dall'esterno

```
1 Check(T)
2   return CheckABR(T, MININT, MAXINT)
```

Visita di un ABR Effettuando una visita inorder di un albero binario si ottiene la sequenza nella sua forma ordinata. Nel caso della figura 14, effettuando una visita inorder l'elenco dei nodi acceduti sarebbe $1, 3, 4, 6, 7, 8, 10, 13, 14$.

Calcolo del minimo e del massimo Il minimo e il massimo di un ABR possono essere calcolate con costo $O(h)$, in quanto è sufficiente accedere al nodo più a sinistra e quello più a destra per ottenere rispettivamente i due valori.

Listing 29: Algoritmo per ricavare il minimo di un ABR (ricorsivo)

```
1 min(T)
2   if (T != NULL) then
3     if (T->sx != NULL) then
4       return min(T->sx)
5     else
6       return T
7   else
8     return T
```

L'algoritmo è ricorsivo in coda, quindi il costo di questo algoritmo in termini di tempo e spazio può essere reso identico a quello della versione iterativa.

Algoritmo di ricerca in un ABR L'algoritmo di ricerca in un albero ordinato diventa molto più semplice rispetto ad un albero normale, in quanto non è più necessario visitare tutti gli elementi di un albero.

Listing 30: Implementazione dell'algoritmo di ricerca in un ABR

```
1 Find(T, k)
2   if (T != NULL) then
3     if T->key < k then
4       return Find(T->dx, k)
5     else if T-key > k then
6       return Find(T->sx, k)
```

```
7       else
8         return T
9     else
10      return T
```

Anche questo algoritmo è ricorsivo in coda. Non è possibile dire che questo algoritmo di ricerca abbia complessità logaritmica sul numero dei nodi come invece era possibile con gli array, in quanto è necessario prima che l'albero abbia altezza logaritmica rispetto al numero dei nodi. Il costo effettivo della ricerca in un albero dipende quindi dalla sua struttura, ma specificatamente *dalla sua altezza*, a differenza della ricerca in un albero generico.

Complessità delle operazioni su ABR Tutte le operazioni sugli ABR possono essere effettuate nel caso peggiore in $O(h)$, e li rende molto più appetibili rispetto agli array ordinati, in quanto ad esempio in questi ultimi le operazioni di eliminazione sono molto costose.

Ricerca del successore/predecessore di una chiave Cercare il successore o il predecessore di T con una chiave k è un'operazione che cerca di trovare il più piccolo elemento di T che sia maggiore di k. *Nota: questa sezione è incompleta. Consultare la registrazione del giorno per renderla completa.*
November 20, 2018

Complessità di un ABR e tempo di esecuzione per l'ordinamento di una sequenza Dato un ABR è possibile estrarre un ordinamento tramite una visita inorder in tempo $O(n)$, ma è stato precedentemente dimostrato che non è possibile produrre una sequenza ordinata in tempo minore di $O(nlogn)$. Come è possibile? Se l'overhead non si trova nell'operazione di lettura, si troverà quindi nella *fase di costruzione dell'ABR*, altrimenti sarebbe possibile definire un algoritmo di ordinamento che risolverebbe il problema in tempo inferiore. Si dimostrerà che non è possibile costruire un ABR con tempo minore di $nlogn$.

Implementazione dell'algoritmo per ricavare il successore di una chiave Si può ora procedere a fornire un algoritmo per ricavare il successore di una chiave k all'interno di un ABR T. L'algoritmo può essere schematizzato in base a tre casi principali in base a come la chiave contenuta nel nodo corrente si confronti con k:

- Se k è uguale alla chiave contenuta in T:

  - Se il sottoalbero destro non è vuoto, allora il risultato è il valore minimo del sottoalbero destro.
  - In caso contrario, viene ritornato NULL.

- Se k è maggiore della chiave contenuta in T, allora vuol dire che k può essere soltanto nel sottoalbero destro–e anche il successore di k. Il valore da restituire pertanto coincide col valore che restituisce la ricerca del successore nell'albero destro.

- Se k è minore della chiave contenuta in T, in questo caso i valori più grandi k possono trovarsi ovunque: sia nella radice che a destra e a sinistra. Di questi valori, tuttavia, si è interessati solo al più piccolo, pertanto è ragionevole pensare di andare a cercare il nodo prima a sinistra. Nel caso ciò fallisse, allora il successore deve essere il nodo stesso.

Listing 31: Implementazione di un algoritmo per ricavare il successore di un valore k in un ABR (ricorsivo)

```
1  Succ(T, k)
2    succ = NULL
3    if T != NULL then
4      if T->key = k then
5        if T->dx != NULL then
6          succ = min(T->dx)
7      else if T->key < k then
8        succ = Succ(T->dx, k)
9      else
10       succ = Succ(T->sx, k)
11       if succ = NULL then
12         succ = T
13   return succ
```

Questo algoritmo non ha una forma ricorsiva in coda, in quanto dopo la seconda chiamata ricorsiva viene effettuata un'altra istruzione che effettua una lettura di T. Tuttavia, utilizzando artifici simili a quelli visti precedentemente è possibile definire un algoritmo ricorsivo in coda che risolve il problema. La ricerca del successore non richiede l'esplorazione di tutto l'albero (come ad esempio una visita) ma solo la visita di uno specifico percorso. Lo stack non è pertanto necessario per la ricerca in sé, ma solo per il *modo* in cui è definito l'algoritmo stesso. Come si può procedere quindi a rendere l'algoritmo ricorsivo in coda? L'idea è tenere traccia dell'ultimo nodo in cui

si è scesi a sinistra, in modo tale che quando si arrivi ad accorgersi che la soluzione si trova sopra essa sia già a disposizione e sia quindi possibile restituirla senza risalire. Si può osservare che la soluzione è molto simile a quella già precedentemente trovata nel caso delle liste, in cui si conservava il valore precedente.

Listing 32: Implementazione di un algoritmo per ricavare il successore di un valore k in un ABR (ricorsivo in coda)

```
1  SuccInCoda(T, k, c) {c sta per candidato, l'equivalente del
       padre nella versione delle liste}
2    if T != NULL then
3      if T->key = k then
4        if T->dx != NULL then
5          {in questo caso c e' irrilevante, quindi si puo'
               semplicemente ritornare}
6          return min(T->dx)
7        {altrimenti si e' arrivati a k senza albero destro,
             quindi il successore e' sopra e il nodo sopra e'
             proprio c (si arriva al return in basso)}
8      else if T->key < k then
9        {se k e' piu' grande, a sinistra non c'e' niente, T
             non e' e quindi l'unica possibilita' e' che sia a
             destra. siccome si sta scendendo a destra, il
             candidato rimane lo stesso}
10       return SuccInCoda(T->dx, k, c)
11     else
12       {se la chiave corrente in T e' piu' grande di k,
             allora il risultato e' la chiamata ricorsiva in cui
             il candidato e' proprio T, cioe' l'ultimo nodo dal
             quale si e' scesi a sinistra}
13       return SuccInCoda(T->sx, k, T)
14   {se si arriva al punto di aver capito che k non e'
       presente, la soluzione e' proprio il candidato}
15   return c
16
17 Succ(T, k)
18   return SuccInCoda(T, k, NULL)
```

È quindi possibile fornire un'implementazione iterativa equivalente (che quindi non presenta alcuna differenza in termini di complessità rispetto all'altro) dello stesso

algoritmo in modo semplice:

Listing 33: Implementazione di un algoritmo per ricavare il successore di un valore k in un ABR (iterativo)

```
1  SuccIt(T, k)
2    cur = T
3    c = NULL
4    while cur != NULL && cur->key != K do
5      if cur->key < k then
6        cur = cur->dx
7      else
8        c = cur
9        cur = cur->sx
10   {alla fine del while o cur e' nil oppure e' diverso da nil
       . nel primo caso, si restituisce c, altrimenti k e'
       stato trovato e bisogna vedere se ha destro o no. se ha
       destro, si restituisce min(dx), altrimenti si
       restituisce c}
11   if cur != NULL && cur->dx != NULL then
12     succ = min(cur->dx)
13   else
14     succ = c
15   return succ
```

Esercizio per il lettore: implementare anche l'algoritmo per ricavare il predecessore.

Operazioni di espansione e contrazione Dato un albero di ricerca in ingresso T, si vuole inserire un nuovo elemento nell'insieme in modo tale da ottenere come risultato un ABR espanso.

Inserimento In un ABR, l'inserimento è molto semplice. Al fine di preservare l'ordinamento l'idea alla base è la stessa dell'inserimento in una sequenza ordinata: bisogna trovare la giusta posizione nel quale effettuare l'inserimento, e la posizione giusta è proprio dove ci si aspetta di trovare il nuovo nodo (figura 15). In un ABR è sempre possibile inserire un nuovo elemento senza effettuare alcuno spostamento, in quanto è sempre presente lo spazio per farlo. In definitiva, l'idea è:

1. Si effettua la ricerca di k. Se k viene trovato, allora il nodo è già presente e quindi non c'è nulla da fare.

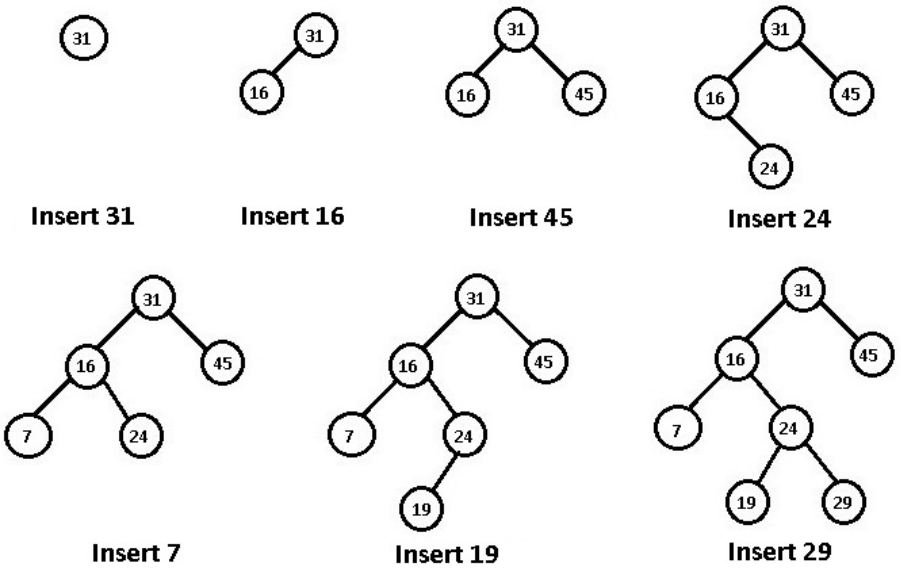


Figure 15: Esempio di operazioni di inserimento in un ABR

2. In caso contrario, la ricerca si fermerà a NULL.

3. In questa posizione si crea un nodo che viene collegato all'ultimo nodo visitato.

Il problema dell'inserimento è un'iterazione del problema della ricerca, da ciò segue che la complessità asintotica sarà al più lineare sull'altezza, ovvero $O(h)$. Dal punto di vista algebrico, l'operazione di inserimento ritorna un nuovo albero T' tale che:

$$T' = T \cup \{k\}$$

Si può quindi procedere ad un'implementazione.

Listing 34: Implementazione dell'inserimento in un ABR (ricorsivo)

```
1 Insert(T, k)
2   if T != NULL then
3     if T->key = k then
4       return T
5     else if T->key < k then
6       T->dx = Insert(T->dx, k)
7       return T
8     else
```

```
 9          T->sx = Insert(T->sx, k)
10          return T
11      else
12        Tnew = {alloca elemento}
13        Tnew->key = k
14        return Tnew
```

L'algoritmo non è ricorsivo in coda a causa della lettura di T in seguito alle chiamate ricorsive. *Esercizio per il lettore: rendere l'algoritmo ricorsivo in coda utilizzando la stessa tecnica precedentemente usata per le liste.*

Cancellazione La cancellazione concettualmente non è semplice quanto l'inserimento. Analizzando i casi in figura 16:

1. Un nodo senza figli può essere semplicemente eliminato dall'albero.
2. Un nodo con due figli *non viene effettivamente eliminato* ma viene sostituito, scegliendo il suo successore inorder o il suo predecessore inorder (chiamato p) da posizionare al suo posto. A quel punto viene ricorsivamente chiamata la procedura di cancellazione su p finché non si raggiunge uno dei due casi base.
3. Un nodo con un figlio *non viene effettivamente eliminato* ma viene sostituito dal suo figlio.

L'algoritmo viene diviso in due fasi:

1. La prima frase dove viene cercato l'elemento.
2. Se e solo se la ricerca ha successo, si procede con la seconda fase dove si considera il sottoalbero che ha come radice k–il problema sarà quindi eliminare da un albero la sua radice, e si presentano i casi visti precedentemente.

Si può quindi procedere ad implementare l'algoritmo di cancellazione della radice di un albero.

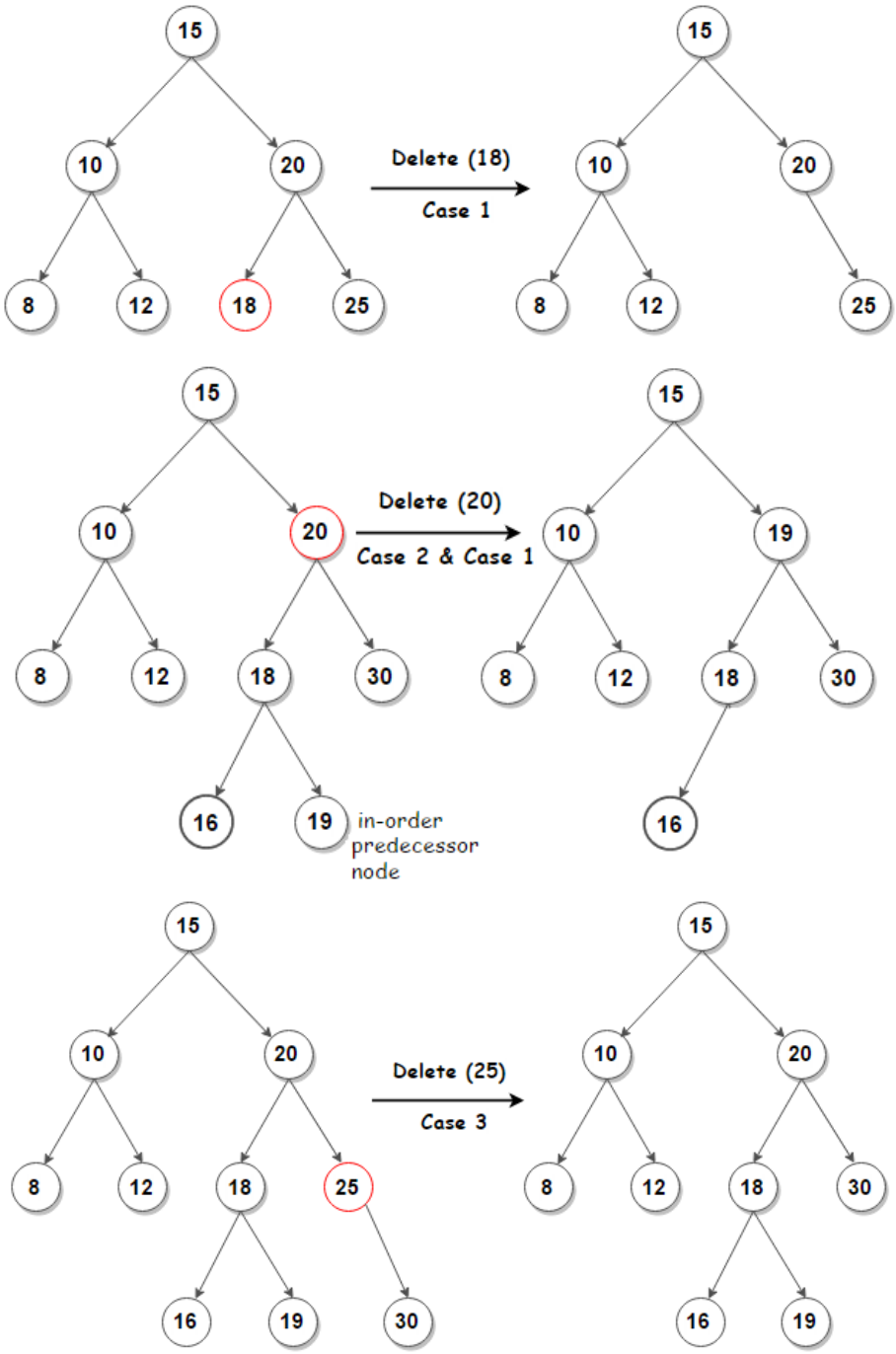


Figure 16: Cancellazione in un ABR

Listing 35: Implementazione della cancellazione della radice in un ABR (ricorsivo)

```
1  CancellaRadice(T)
2    if T != NULL then
3      if (T->sx = NULL || T->dx = NULL) then
4        {caso 1 o 3}
5        if T->sx = NULL then
6          root = T->dx
7        else
8          root = T->sx
9        {deallocazione di T}
10       return root
11     else
12       {caso 2 - due nodi}
13       {si cancella il minimo del sottoalbero e lo si ottiene
             (nb: la funzione di cancellazione del minimo
           restituisce non l'albero, ma il minimo cancellato)}
14       tmp = StaccaMin(T->dx, T)
15       T->key = tmp->key
16       {deallocazione di tmp}
17       return T
```

L'algoritmo fa uso di una funzione chiamata StaccaMin che rimuove il minimo da un sottoalbero e ritorna il nodo appena rimosso. L'algoritmo per staccare il minimo è una variante di quello che cerca il minimo ed è il seguente:

Listing 36: Implementazione dell'algoritmo per staccare il minimo in un ABR)

```
1  StaccaMin(T, P)
2    if T != NULL then
3      if T->sx != NULL then
4        return StaccaMin(T->sx, T)
5      else
6        {se il sottoalbero e' la foglia sinistra del genitore,
             la foglia sinistra del genitore diventa la foglia
           destra di questo sottoalbero}
7        if T = P->sx then
8          P->sx = T->dx
9        else
10         P->dx = T->dx
11       return T
```

```
12    return NULL
```

Si può quindi procedere all'algoritmo di cancellazione generica.

Listing 37: Implementazione dell'algoritmo di cancellazione in un ABR)

```
1  Cancella(T, k)
2    if T != NULL then
3      {due casi: caso in cui la chiave sia maggiore di k e si
           va a sx, o viceversa}
4      if T->key > k then
5        T->sx = Cancella(T->sx, k)
6      else if T->key < k then
7        T->dx = Cancella(T->dx, k)
8      else
9        T = CancellaRadice(T)
10   return T
```

Questo algoritmo ha la stessa complessità della ricerca e dell'inserimento, e nel caso peggiore può seguire un percorso lungo quanto l'altezza dell'albero stesso. Il tempo dell'algoritmo di cancellazione è quindi proporzionale all'altezza dell'albero. Si può osservare che l'algoritmo non è ricorsivo in coda ma utilizzando le strategie viste precedentemente (aggiungendo un parametro alla funzione) si può rendere tale.

6.3.9 Albero binario di ricerca bilanciato

Un albero binario di ricerca si dice bilanciato quando l'altezza rimane piccola a fronte di arbitrarie aggiunte e cancellazioni di elementi, garantendo che le operazioni di inserimento e cancellazione vengano eseguite al più in $O(logn)$, dove n è il numero di nodi dell'albero. Ogni ABR bilanciato ha quindi altezza logaritmica sul numero di nodi.

Definizione formale Una classe A di alberi si dice bilanciata se per ogni albero appartenente a questa classe l'altezza di T è $O(log_2 n)$ con $n = |T|$.

$$\forall T \in A, h(T) = O(log_2 n) \text{ con } n = |T|$$

Raffronto con la ricerca binaria L'idea alla base della ricerca binaria è quella di dividere un array in due parti grandi circa $n/2$ ciascuna, utilizzando l'elemento al centro come discriminante per proseguire la ricerca. L'idea potrebbe essere quella di sfruttare lo stesso principio per gli ABR bilanciati, ovvero: se si avesse un albero la

cui radice è tale per cui a sinistra di quella radice ci sono tanti elementi quanti ce ne stanno a destra (quindi un albero in cui l'insieme è suddiviso in tre parti: la radice e due sottoalberi che hanno la stessa cardinalità), ogni volta che si scende lungo un arco la cardinalità dell'albero raggiunto è la metà di quella da cui si è partiti. L'altezza di un albero di questo genere sarebbe proprio $log_2 n$, e si definisce **albero perfettamente bilanciato**.

Albero perfettamente bilanciato Un albero perfettamente bilanciato è l'albero col bilanciamento migliore possibile. Non tutti gli alberi completi sono perfettamente bilanciati ma tutti gli alberi perfettamente bilanciati sono pieni.

$$T \in APB \Leftrightarrow \forall x \in T, |T(x) \to sx| \text{ differisce al piu}' \text{ di 1 di } |T(x) \to dx|$$

Se si è a conoscenza di tutti i dati durante la fase di costruzione dell'albero, è facile creare un albero perfettamente bilanciato: si dispongono i dati ordinati in un array e si prende il valore medio come radice dell'albero; a questo punto tutti i nodi a sinistra sono minori della radice e andranno nel sottoalbero sinistro, tutti quelli maggiori nel sottoalbero destro. Tuttavia, la classe di alberi perfettamente bilanciati è fin

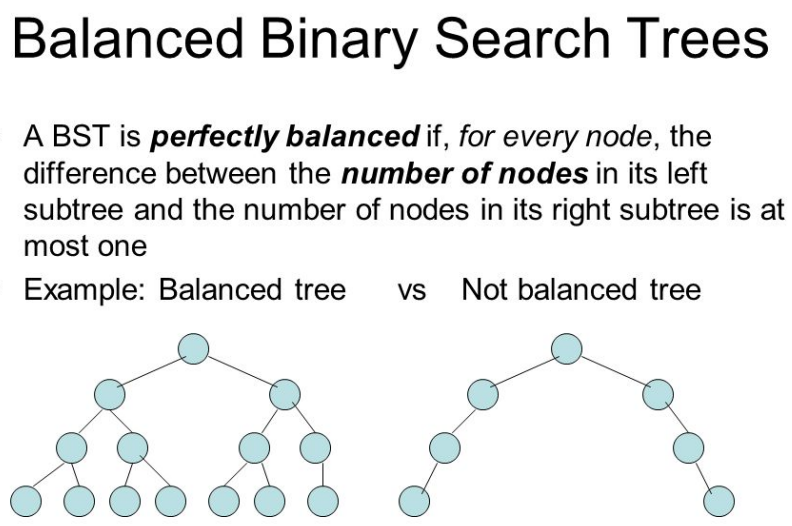
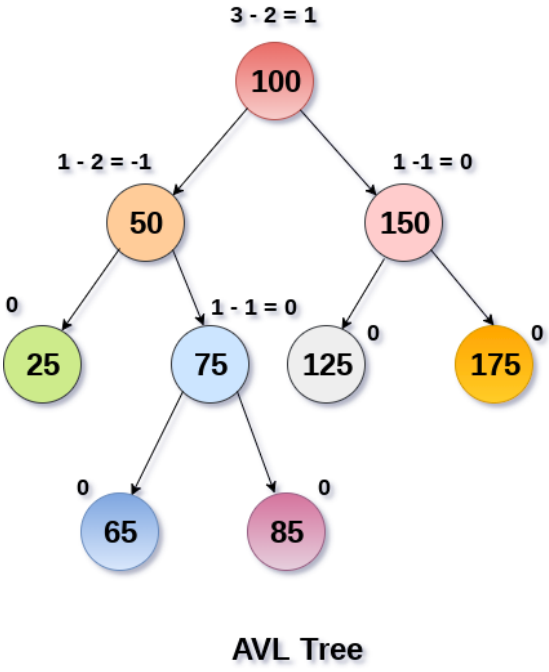
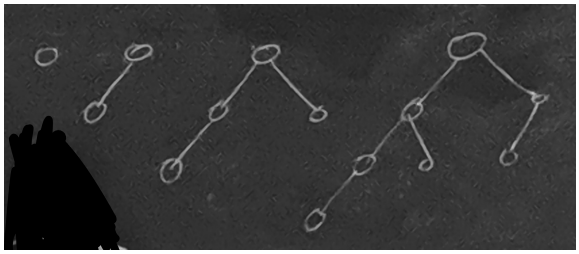


Figure 17: Definizione ed esempio di un ABR bilanciato



(a) Visualizzazione di un AVL



(b) AVL minimi per $h = 0, 1, 2, 3$

troppo stringente, e aumenta il costo delle operazioni di inserimento e cancellazioni a $O(nlog_2n)$—è, pertanto, praticamente poco utile.

Albero AVL (Adelson-Velsky e Landis) La definizione di questa classe di ABR è molto semplice e simile a quella degli ABR perfettamente bilanciati, tuttavia invece di porre un vincolo tra le cardinalità degli insiemi dei sottoalberi viene fatto per le altezze dei due sottoalberi di un nodo. Formalmente:

$$T \in AVL \Leftrightarrow \forall x \in T, |h(T(x) \to sx) - h(T(x) \to dx)| \leq 1$$

Ogni albero perfettamente bilanciato è anche un albero AVL. Non è facile quanto un albero perfettamente bilanciato dimostrare che ogni albero AVL abbia altezza logaritmica sul numero dei nodi.

Relazione tra numero di nodi e altezza di un albero AVL Sapendo l'altezza di un albero AVL non è possibile dire con certezza il numero di nodi, e sapendo il numero di nodi non è possibile dire con certezza l'altezza dell'albero, come per gli alberi binari. Per risolvere il problema, si può trovare una sottoclasse degli AVL

(scelta opportunamente) che possa stabilire un limite superiore all'altezza di un AVL in base al numero dei nodi. Si definisce "AVL minimo" una particolare classe di AVL che presenta la proprietà di avere le peggiori altezze possibili. Stabilendo un limite superiore su questa classe, è possibile estendere il risultato per tutta la classe, trattandosi di un campione significativo per tutta la classe degli AVL.

Albero AVL minimo di altezza h Un albero AVL minimo è l'albero AVL di altezza h con il minimo numero di nodi. Un esempio di AVL minimi si ha nella figura 18b per $h = 0, 1, 2, 3$. Eliminando qualsiasi foglia, si violerebbe o la proprietà di appartenenza agli AVL oppure l'albero diventerebbe di altezza inferiore.
Osservando la figura, si può osservare che il sottoalbero sinistro dell'AVL minimo di $h = 3$ è un AVL minimo di $h = 2$ e che il sottoalbero destro è un AVL minimo di $h = 1$. Per dimostrare questa proprietà, data l'ipotesi che un albero sia un AVL di altezza h e supponendo che quello a sinistra di altezza $h - 1$ non sia minimo, allora si potrebbe costruire un AVL di altezza h che ha meno nodi. In questo caso, tuttavia, si andrebbe contro l'ipotesi (perché l'albero di partenza non sarebbe più un AVL), dimostrando per assurdo che la proprietà osservata è verificata per un AVL di qualsiasi altezza.
A questo punto, è facile osservare che il numero di nodi di un AVL minimo è funzione di h e non dipende dalla forma. Si tratta della stessa proprietà presente negli alberi pieni, assente in quasi tutte le classi di alberi. Per qualsiasi albero binario, la funzione numero di nodi di un albero T è:

$$\forall T \in AB, N(T) = 1 + N(T \to sx) + N(T \to dx)$$

In un AVL minimo, la funzione $N_M(h)$ è:

$$N_M(h) = 1 + N_M(h-1) + N_M(h-2)$$

Si tratta di un'equazione di ricorrenza che necessità di due casi base:

$$N_M(0) = 1; N_M(1) = 2$$

L'ultima considerazione da fare è dimostrare che l'analisi sia significativa per tutti gli AVL. Per farlo, si può dimostrare:

$$\forall h \geq 0 \text{ sia } T_h \in AVL_M(h), \forall T \in AVL, |T| = |T_h| \Rightarrow h(T) \leq h(T_h) = h$$

Si può procedere con una dimostrazione per assurdo. Affinché tale proprietà sia falsa, vuol dire che deve esistere almeno un'altezza tale che preso un AVL minimo

di quell'altezza, allora è possibile trovare almeno un AVL T con altezza maggiore ma stesso numero di nodi. Si definisce T' l'albero T dove tutti i nodi sotto h sono stati scartati. Poiché sicuramente almeno un nodo è stato eliminato, allora T' ha cardinalità minore di n (cioè $|T'| < n$) ed ha la stessa altezza di T_h (cioè $h(T') = h$). Hanno quindi la stessa altezza ma T_h ha più nodi di T'—se fosse vero, si avrebbe una contraddizione. Non si è certi tuttavia che T' sia un AVL, in quanto se non lo fosse allora si andrebbe contro l'ipotesi. Supponendo che T' non sia un AVL, dev'essere falsa la proprietà di AVL, deve cioè esistere un nodo per cui la relazione sia falsa—un nodo a caso deve avere due sottoalberi tale che la differenza di altezze tra questi due sottoalberi sia maggiore di 1. Il taglio non può avere aumentato l'altezza di un qualsiasi sottoalbero o ridotto quella di un altro, se ci fosse una violazione della proprietà sarebbe preesistente: se è un AVL, quindi, il taglio di nodi sotto all'altezza scelta non può far perdere la proprietà di AVL. Si è quindi dimostrato che l'albero T non esiste, e che il campione che è stato scelto è valido per tutti gli AVL.
November 23, 2018

Sono stato assente questo giorno. Si è parlato di rotazioni di alberi. Recuperare!
November 27, 2018
Una rotazione da sinistra verso destra permette di far salire il sottoalbero sinistro di un livello facendo scendere l'altro di un livello.

Cancellazione in un AVL E' possibile usare gli stessi algoritmi BilanciaSinistro e BilanciaDestro che sono già stati utilizzati precedentemente nell'inserimento. Tutta la fase di discesa o di risalita deve prevedere la presenza di uno sbilanciamento nel nodo corrente. La cancellazione potrebbe far diminuire l'altezza di un sottoalbero, e poiché anche la rotazione potrebbe farlo, di sicuro l'altezza dell'albero non sarà uguale a quella di prima, e non è possibile dare per scontato che il numero di rotazioni sarà uno o due–tuttavia, è sempre limitato.

Listing 38: Implementazione dell'algoritmo di cancellazione in un AVL

```
1 CancAVL(T, k)
2   if T != NULL then
3     if T->key > k then
4       T->sx = CancAVL(T->sx, k)
5       {come nell'inserimento, adesso si e' nella situazione
          in cui il sottoalbero sinistro e' un AVL, il
          sottoalbero destro e' un AVL perche' non e' stato
          toccato ma non e' piu' detto che l'albero radicato
          in T sia un AVL}
```

```
6         T = BilanciaDx(T) {versione simmetrica di BilanciaSx}
7       else if T->key < k then
8         T->dx = CancAVL(T->dx, k)
9         T = BilanciaSx(T)
10      else
11        T = CancRadiceAVL(T)
12    return T
```

Listing 39: Implementazione dell'algoritmo di cancellazione della radice in un AVL

```
1  CancRadiceAVL(T)
2    if T != NULL then
3      if T->sx = NULL || T->dx = NULL then
4        {si e' nel caso in cui o un albero ha solo il sx/dx o
             nessuno dei due. l'algoritmo deve allora restituire
             il sottoalbero, ed e' sicuramente un AVL se e' un
             sottoalbero, non c'e' nulla da bilanciare}
5        Temp = T
6        if T->sx = NULL then
7          T = T->dx
8        else
9          T = T->sx
10     else
11       Temp = StaccaMinAVL(T->dx, T)
12       T->key = Temp->key
13       {l'altezza del sottoalbero destro potrebbe essere
             diminuita di 1, e potrebbe determinare una
             differenza di 2 tra sx e dx. quindi bisogna
             assicurarsi che l'albero che si trova in T sia un
             AVL}
14       T = BilanciaSx(T)
15     {dealloca Temp}
16   return T
```

Listing 40: Implementazione dell'algoritmo per staccare il minimo da un AVL

```
1  StaccaMinAVL(T, P)
2    if T != null then
3      if T->sx != NULL then
```

```
4         {T non contiene il minimo, quindi il minimo si trova
              nel sottoalbero sx di T}
5         Ret = StaccaMinAVL(T->sx, T)
6         NewRoot = BilanciaDx(T)
7         {adesso si hanno due valori: la nuova radice dell'
              albero radicato in T dopo aver staccato il suo
              minimo e il puntatore al minimo}
8       else
9         Ret = T
10        NewRoot = T->dx
11      if P->sx = T then
12        {in questo caso il nodo T e' il figlio sinistro di P,
              quindi bisogna aggiornare le referenze}
13        P->sx = NewRoot
14      else
15        P->dx = NewRoot
16    return Ret
```

Osservazione Si può osservare che nessuno degli algoritmi visti è ricorsivo in coda, e non è facile effettuare una conversione come lo era in un ABR in quanto a valle di un'operazione potrebbe sempre esserci bisogno di un bilanciamento. Si consulti il libro di testo per vedere rappresentazioni iterative di questi algoritmi, che non utilizzano esplicitamente uno stack ma fanno uso di un puntatore al padre. Lo spazio necessario tuttavia diventa logaritmico nel caso peggiore rispetto invece allo spazio lineare negli ABR, un sostanziale vantaggio. Nel caso peggiore in un AVL il numero di rotazioni è lineare all'altezza ed è essenzialmente $\frac{h}{2}$.

Albero rosso-nero Un albero rosso-nero è un ABR in cui i dati sono mantenuti solo nei nodi interni–le foglie di questi alberi non contengono dati. I vincoli strutturali non sono immediatamente collegati al bilanciamento come potevamo apparire quelli degli AVL.

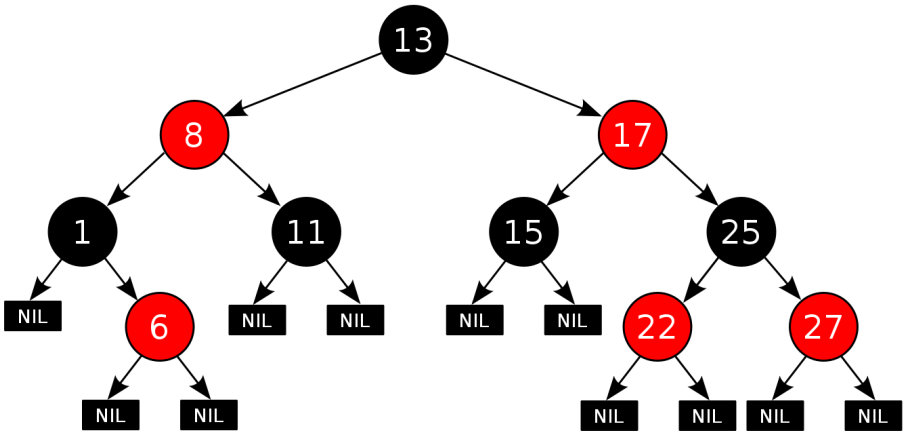


Figure 19: Albero rosso-nero

I vincoli di un ARB sono:

1. Ogni nodo è colorato di rosso o di nero;
2. Ogni nodo rosso può solo avere figli neri;
3. Per ogni nodo x: tutti i percorsi da x a una foglia contengono lo stesso numero di nodi neri;
4. Le foglie sono *sempre nere*.

La classe di questi alberi è più tollerante di quella degli AVL. È infatti possibile ottenere un albero in cui la differenza tra le altezze dei due sottoalberi non è 1.
November 29, 2018

Nozioni Si vuole dimostrare:

$$h = O(log_2 n)$$

Con $h(x)$ si indica l'altezza dell'albero radicato in x. $bh(x)$ (black height) è il numero di nodi neri lungo un qualunque percorso da x ad una foglia. Indicando con $NI(x)$ il numero di nodi interni dell'albero radicato in x (rappresenta cioè la cardinalità dell'insieme dei dati contenuto nell'albero radicato in x perché i dati sono contenuti solo nei nodi interni), si vorrà dimostrare:

$$NI(x) \geq 2^{bh(x)} - 1$$

$2^{bh(x)} - 1$ rappresenta il numero di nodi dell'albero pieno alto $bh(x)$. Si può procedere con una dimostrazione per induzione. Si è a conoscenza che $T(x)$ o è l'insieme vuoto oppure ha un nodo interno e due sottoalberi; il numero di nodi interni di $T(x)$ è esprimibile con:

$$NI(x) = 1 + NI(x \to sx) + NI(x \to dx)$$

Qualunque misura associata ad x deve essere strettamente minore di $x \to sx$ e $x \to dx$. Un buon candidato da utilizzare per l'ipotesi induttiva è quindi *l'altezza*, che preserva questa proprietà. L'ipotesi induttiva sarà quindi:

$$\forall h \geq 0, NI(x) \geq 2^{bh(x)} - 1 \text{ con } h = h(x)$$

Non è vero che l'altezza nera di un albero è strettamente maggiore di quella dei suoi sottoalberi, quindi non è possibile fare induzione con l'altezza nera.
L'unico albero rosso-nero x di altezza $h = 0$ è quello che contiene solo NULL, e si ha $NI(x) = 0, bh(x) = 0 \Rightarrow 2^{bh(x)} - 1 = 2^0 - 1 = 0$. Il caso base sia quindi soddisfatto.
Supponiamo ora di avere un albero rosso-nero di altezza $h(x) > 0$. In questo caso supponendo che la radice sia x e che i due sottoalberi immediatamente discendenti siano y e z, si può dire che:

$$h(z) < h(x) \text{ e } h(y) < h(x)$$

Si sa che:

$$NI(x) = 1 + NI(y) + NI(z)$$

Per induzione si può dire che (cioè discendono dall'ipotesi induttiva):

$$NI(y) \geq 2^{bh(y)} - 1 \tag{17}$$
$$NI(z) \geq 2^{bh(z)} - 1 \tag{18}$$

Che relazione c'è tra l'altezza nera di x e quella di y e z? Si può dire che:

$$bh(y) \geq bh(x) - 1$$
$$bh(z) \geq bh(x) - 1$$

Poiché la funzione esponenziale è una funzione monotona, si ottiene:

$$2^{bh(y)} \geq 2^{bh(x)-1}$$
$$2^{bh(z)} \geq 2^{bh(x)-1}$$

Sottraendo la stessa quantità a destra e sinistra, si preserva la relazione:

$$2^{bh(y)} - 1 \geq 2^{bh(x)-1} - 1$$
$$2^{bh(z)} - 1 \geq 2^{bh(x)-1} - 1$$

Applicando la transitività nella equazione 18, si ha:

$$NI(y) \geq 2^{bh(x)-1} - 1$$
$$NI(z) \geq 2^{bh(x)-1} - 1$$

Si ottiene:

$$NI(y) + NI(z) \geq 2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1$$
$$NI(x) = NI(y) + NI(z) + 1 \geq 2^{bh(x)-1} + 2^{bh(x)-1}$$
$$NI(x) = NI(y) + NI(z) + 1 \geq \boxed{2 \cdot 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1}$$

L'ipotesi induttiva è stata quindi dimostrata.
L'unico modo per aumentare il percorso di un albero rosso-nero è quello di inserire nodi rossi, e quindi si ha:

$$\frac{h(x)}{2} \leq bh(x) \leq h(x) \Rightarrow 2^{\frac{h(x)}{2}} \leq 2^{bh(x)} \Rightarrow NI(x) \geq 2^{\frac{h(x)}{2}} - 1 \Leftrightarrow n+1 \geq 2^{\frac{h}{2}} \Rightarrow log_2(n+1) \geq \frac{h}{2}$$

Sappiamo quindi che è un albero bilanciato e le altezze sono sempre logaritmiche.
Da questo punto in poi consultare le slide, "Violazione delle proprietà per inserimento"
November 30, 2018

NOTA: mancano i primi 30 minuti di lezione.
NOTA: consultare le slide, "Cancellazione in RB".

6.4 Grafi

6.4.1 Definizione

Un grafo è una struttura dati che si può definire come segue. Dato un insieme V:

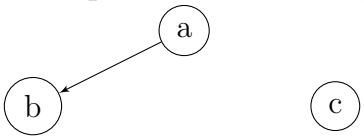
$$E \subseteq V \times V$$

Un grafo è:

$$G = \langle V, E \rangle$$

Dove V rappresenta i *vertici* del grafo e E rappresenta gli *archi*.

Esempio Dato un $V = \{a, b, c\}$ e $(a, b) \in E$:



Alberi e grafi La differenza tra alberi e grafi è che gli alberi sono grafi, ma sono grafi particolari—in cui la relazione è di tipo strettamente gerarchica.

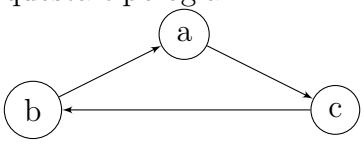
6.4.2 Uso

Moltissimi problemi possono essere riformulati in termini di grafi. Un grafo potrebbe rappresentare un qualsiasi tipo di rete (stradale, di calcolatori, ...). In linea di principio moltissimi sistemi dinamici possono essere rappresentati coi grafi. I grafi non sono strutture dati usate tipicamente per fare ricerche, ma vengono usate per calcolare o risolvere problemi di raggiungibilità. I grafi non vengono quindi proposti come *alternativa* alle strutture dati viste precedentemente.

6.4.3 Nozioni

Grafi orientati e non orientati La differenza risiede esclusivamente nel fatto che le coppie nell'insieme E vengano considerate nel primo caso come un insieme non ordinato e nel secondo come un insieme ordinato. I grafi non orientati sono più specifici di quelli orientati—questo perché un grafo non orientato rappresenta un insieme la cui relazione E è una relazione necessariamente simmetrica. La scelta di quale tipo di grafo da utilizzare per risolvere un problema dipende da un lato dalle caratteristiche del problema e dall'altro qualora è possibile fare delle scelte in base al problema reale da risolvere.

Rappresentazione di un grafo mediante matrici Supponendo un grafico di questa tipologia:



La matrice corrispondente a questo grafo è:

Table 3: Matrice corrispondente al grafo precedente

	A	B	C
a	1	0	1
b	1	0	0
c	0	1	0

Rappresentazione della "taglia" di un grafo La cardinalità "ottimale" di un grafo rappresentato come prima è:

$$|G| = |V| + |E|$$

Si ha:

$$0 \leq |E| \leq |V|^2$$

Questa rappresentazione non è particolarmente vantaggiosa dal punto di vista dello spazio. Nel caso in cui i grafi siano particolarmente densi (cioè che presentano un alto numero di archi) spesso la cardinalità è $|V|^2$. Aggiungere, eliminare o verificare l'esistenza un arco sono operazioni estremamente efficienti. L'aggiunta di vertici è invece particolarmente onerosa, in quanto richiederebbe la creazione di una nuova matrice.
December 4, 2018

6.4.4 Rappresentazione tramite matrice di adiacenza

Ricapitolando, un grafo si può rappresentare mediante una matrice quadrata le cui dimensioni sono $|V|x|V|$. Le coppie si indicano con (i, j) dove $i, j \in V$, e l'insieme degli archi è $E \subseteq V \times V$. Si ha:

$$M_G[i, j] = 1 \Leftrightarrow (i, j) \in E$$

Le coppie nella matrice che hanno il valore assegnato 1 sono esattamente tutte e solo le coppie che sono nell'insieme E (cioè gli archi). Si potrà quindi rappresentare ogni grafo che abbia cardinalità finita ($|V| < \infty$). Questa rappresentazione può risultare particolarmente costosa, specialmente per grafi in cui la cardinalità dell'insieme degli archi è molto inferiore al quadrato della cardinalità dei vertici (grafi "sparsi"). Si definisce *rappresentazione insensibile alla cardinalità di E*. Lo spazio occupato da questa rappresentazione è (con un abuso di notazione):

$|M_G| = |V|^2 >> |V| + |E|$ (puo$'$ essere molto maggiore di quel che ci si aspetta)

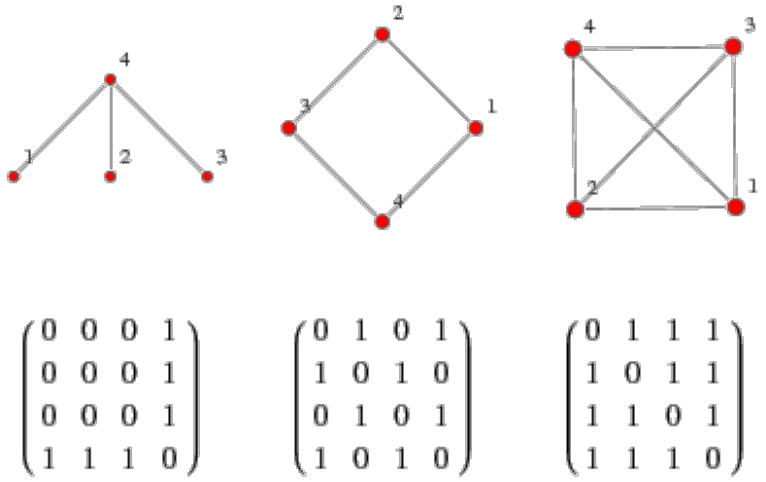


Figure 20: Rappresentazione mediante matrice di adiacenza

Si indicherà la taglia del grafo come $|G|$. La visita di un grafo è un'operazione che attraversa tutti i vertici e tutti gli archi del grafo—dato un vertice v di un grafo, l'operazione considera tutti gli archi uscenti dal vertice per raggiungere e visitare quelli adiacenti. Con questa rappresentazione attraversare gli archi uscenti costa tanto quanto il numero di vertici presenti nel grafo—è indipendente quindi dall'effettivo numero di archi uscenti da v.
Viene detta matrice di adiacenza perché un arco da i a j porta j ad essere definito *adiacente* ad i.

6.4.5 Rappresentazione tramite liste di adiacenza

Ricerca di una rappresentazione più efficiente La rappresentazione precedente non è efficiente né dal punto di vista di spazio né rispetto alle operazioni comuni effettuate su un grafo. A valle della critica corrispondente al fatto che lo spazio utilizzato da una matrice di adiacenza è indipendente dal numero archi, si può considerare una rappresentazione il cui costo è invece lineare sia al numero dei vertici che al numero di archi.
In questa nuova rappresentazione si intende associare ad ogni vertice l'insieme dei suoi vertici adiacenti. Si utilizza una rappresentazione con un array che ha dimensione pari alla cardinalità di V (ogni elemento rappresenta un vertice). Questa rappresentazione viene detta *rappresentazione a liste di adiacenza*. Si tratta di una sequenza di liste ciascuna delle quali codifica l'adiacenza del corrispondente vertice.

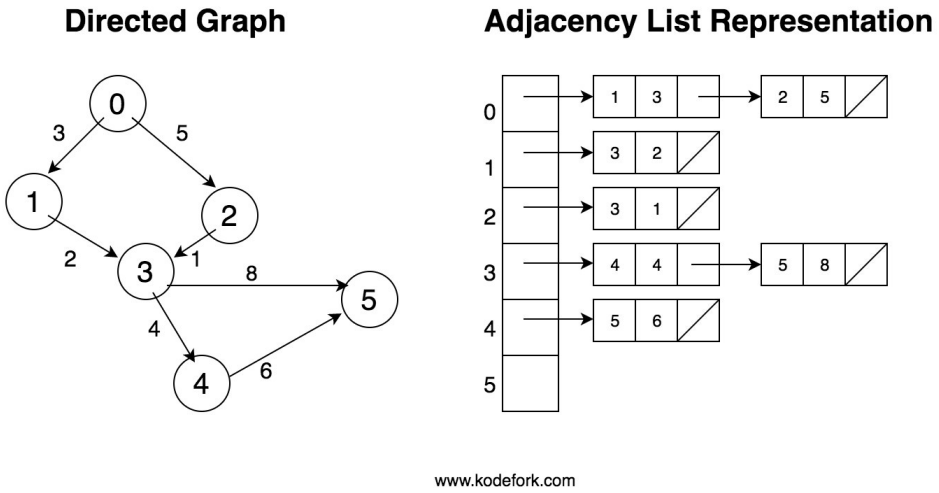


Figure 21: Rappresentazione mediante liste di adiacenza

Spazio utilizzato Il costo di questa rappresentazione è:

$$a|V| + b|E| = \Theta(|V| + |E|)$$

Ed è quindi lineare al numero di vertici e al numero di archi. Tuttavia, questa struttura include delle strutture dati che non sono necessariamente ad accesso diretto—alcune operazioni saranno quindi meno efficienti. Ad esempio, verificare se una coppia di vertici è un arco non ha più lo stesso costo di prima: dati (i, j), scoprire se $(i, j) \in E$ implicherebbe accedere ad i e poi scorrere la lista associata, implicando un costo lineare di $O(|V|)$.

Costo operazioni di base Aggiungere un arco potrebbe avere lo stesso costo di una matrice di adiacenza se si aggiungesse in testa ad una lista. L'eliminazione è il costo di un'eliminazione in una lista. Dal punto di vista delle operazioni di espansione e contrazione dell'insieme non è direttamente confrontabile con una matrice di adiacenza (cioè è meglio in alcune operazioni e peggio in altre, esiste quindi una forma di compromesso)—tuttavia, l'operazione di esplorazione degli archi uscenti non ha più un costo indipendente dal numero di archi uscenti come nel caso di una matrice di adiacenza, ma costa esattamente quanto il numero di archi uscenti da esplorare. Per questo caso d'uso è quindi una rappresentazione ottimale.
Questa rappresentazione sarà quella maggiormente usata nel corso delle lezioni. Si

tratta di rappresentazioni general-purpose: possono essere utilizzate per rappresentare qualunque tipo di grafo. Nel caso di applicazioni più specifiche, è certamente possibile considerare rappresentazioni più vincolate ma che hanno benefici specifici nel contesto dell'applicazione.

6.4.6 Nozioni fondamentali

Se una coppia di vertici (u, v) appartiene ad E (e quindi c'è un arco che va da u a v), chiameremo v l'adiacente di u.
Un percorso di G è una sequenza di vertici $v_1, v_2, \ldots, v_k$ che ha la seguente proprietà:

$$\forall 1 \leq i \leq k-1, (v_i, v_{i+1}) \in E$$

Ogni coppia di vertici adiacenti in questa sequenza, v_i e v_{i+1} sono archi del grafico. In altre parole, la sequenza è una sequenza di vertici tra loro connessi da archi. Con la seguente condizione:

$$\forall 1 \leq i \leq k, v_i \in V$$

Ovvero questa sequenza contiene solo vertici di V. Dato il grafo orientato in figura

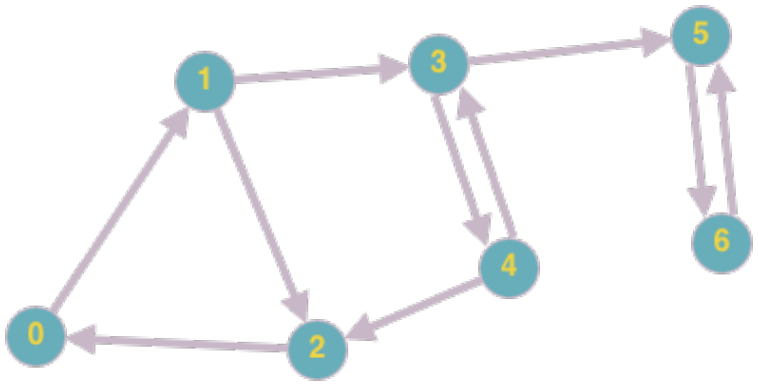


Figure 22: Esempio di grafo di cui considerare il percorso

22, un percorso di esempio è $0, 1, 3, 5, 6$.
Dato un percorso v_1, v_2, v_3, v_4 copre 4 vertici ma attraversa 3 archi. Il numero di vertici nel percorso è sempre 1 in più rispetto al numero di archi percorsi.
Quanti percorsi si possono individuare in figura 22? Infiniti. Anche solo considerando sequenze finite, questo grafo ammette un numero infinito di percorsi finiti. Anche il grafo in figura 23 ammette infiniti percorsi. Se la cardinalità di V è finita, si può definire una nozione più stringente di percorso: i percorsi semplici.

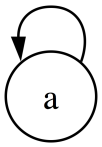


Figure 23: Grafo con un ciclo semplice

Percorso semplice Un percorso si definisce semplice se *non contiene più copie dello stesso vertice.* L'insieme dei percorsi semplici di un grafo è finito a prescindere dagli archi. Un percorso semplice non può essere più lungo del numero di vertici. Se la lunghezza di un percorso semplice è limitata dal numero di vertici, il numero di percorsi semplici in un grafo sarà rappresentato da tutte le sequenze con lunghezza minore o uguale della cardinalità di V. **Ogni percorso che contiene più di $|V|$ vertici implica l'esistenza di un *ciclo* all'interno del grafo**, cioè una sequenza di archi che portano da un vertice allo stesso vertice (come in figura 23).

Ciclo semplice Un ciclo semplice è una sequenza di vertici $v_1, \ldots, v_{k-1}, v_k$ tale che:

1. $v_1 = v_k$
2. $v_1 \ldots v_{k-1}$ e' un percorso semplice

Cioè un ciclo semplice è un percorso semplice alla fine del quale si aggiunge il primo vertice della sequenza. Non è vero che ogni percorso semplice può essere esteso a un ciclo semplice—è possibile se e solo se esiste un arco che da v_{k-1} porta a v_1.
In ogni percorso che non è semplice è contenuto un ciclo semplice. L'assenza di cicli semplici implica l'assenza di percorsi infiniti. Quando il grafo non contiene cicli viene detto **grafo aciclico**. Bisognerà porre particolare attenzione ai cicli in un algoritmo di visita.
Ogni albero è un grafo aciclico, ma ogni grafo non è detto sia un albero.

Sottografo Dato un grafo $G = \langle V, E \rangle$, si definisce $G' = \langle V', E' \rangle$ un sottografo di G se valgono le seguente condizioni:

1. $V' \subseteq V$
2. $E' \subseteq E \cap (V' \times V')$

Si definisce *sottografo massimale* un sottografo che massimizza l'insieme degli archi inseribili da G:

1. $V' \subseteq V$
2. $E' = E \cap (V' \times V')$

December 6th, 2018

6.4.7 Raggiungibilità in un grafo

La raggiungibilità in $G = \langle V, E \rangle$ è una relazione Reach $\subseteq V \times V$. Si ha:

$$\forall u, v \in V, (v, u) \in \text{Reach} \Leftrightarrow \exists \pi(\pi \text{ e' percorso in G } \wedge first(\pi) = v \wedge last(\pi) = u)$$

Se esiste un arco che va da v a u allora sicuramente (v, u) appartiene a *Reach*.

$$(v, u) \in E \Leftrightarrow (v, u) \in Reach$$

Esiste un percorso che va da v a v anche se non esiste un arco che lo collega:

$$(v, v) \notin E, (v, v) \in Reach$$

Ogni coppia che è in E implica l'appartenenza a *Reach*:

$$(v, u) \in E \Rightarrow (v, u) \in Reach$$

Vale la proprietà di transitività:

$$(v, u) \in Reach \wedge (u, z) \in Reach \Rightarrow (v, z) \in Reach$$

In definitiva, *Reach* è la più piccola relazione tale che:

1. $E \subseteq Reach$
2. $\forall v \in V, (v, v) \in Reach$ (riflessività)
3. $(v, u) \in Reach \wedge (u, z) \in Reach \Rightarrow (v, z) \in Reach$ (transitività)

Indipendentemente da come è definito E, *Reach* è sicuramente riflessiva e transitiva. Se E fosse simmetrico (cioè se esiste un arco da a a b allora ne esiste anche uno da b ad a), anche *Reach* erediterebbe la stessa proprietà di simmetria. In questo caso, *Reach* diventerebbe una relazione d'equivalenza. Invece, è possibile che E non sia simmetrica ma lo sia *Reach*, come nel caso del grafo ciclo in figura 24.

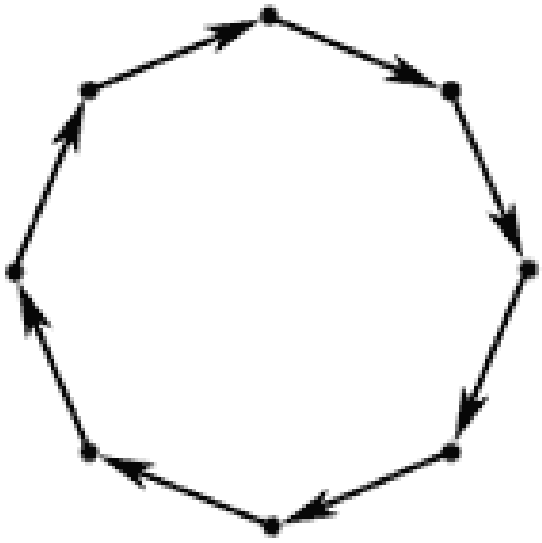


Figure 24: Esempio di un grafo ciclo in cui *Reach* è simmetrica ma E no

6.4.8 Grafo connesso o fortemente connesso

Si parla di grafo connesso quando il grafo è non orientato e di grafo fortemente connesso quando il grafo è orientato. Un grafo non orientato (in cui come detto precedentemente la relazione E è simmetrica) si dice connesso se:

$$\forall (u, v) \in V, (u, v) \in Reach$$

In questo caso, *Reach* è una relazione d'equivalenza. Se vale che ogni coppia di vertici soddisfa l'appartenenza a *Reach*, allora l'intero grafo è un'unica classe di equivalenza.
Un grafo orientato si dice fortemente connesso se vale la stessa proprietà esposta precedentemente.
In un grafo non orientato, le classi di equivalenza indotte dalla relazione di equivalenza *Reach* si chiamano **componenti connesse**. In figura 25 i due sottografi superiori e inferiori sono componenti connesse massimali, mentre il sottografo formato dai nodi 1 e 2 *non* è una componente connessa.
In un grafo orientato si parlerà invece di **componenti fortemente connesse**. Poiché la relazione di raggiungibilità non è detto che sia simmetrica e quindi d'equivalenza, si definisce una relazione di mutua raggiungibilità: due vertici appartengono a tale relazione se sia la coppia (u, v) sia la coppia (v, u) sono in *Reach*. Si definisce come

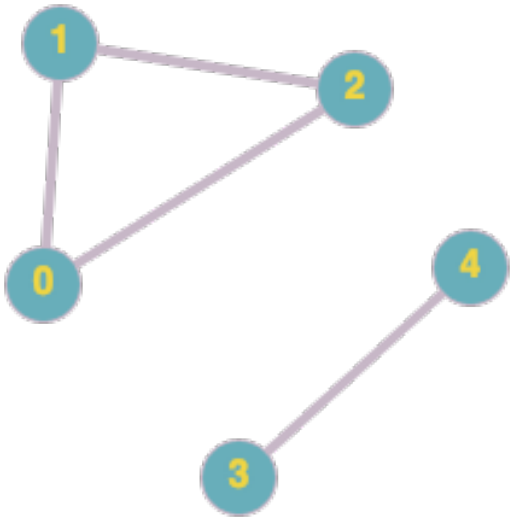


Figure 25: Esempio di **componenti connesse**—in questo grafo sono presenti due classi di equivalenza, una corrispondente ai nodi $0, 1, 2$ e l'altra corrispondente ai nodi $3, 4$.

Figure 26: In questo grafo orientato la relazione di mutua raggiungibilità ha 3 classi di equivalenza: $(0, 0), (1, 1), (2, 2)$

segue la relazione di mutua raggiungibilità:

$$(u, v) \in MReach \Leftrightarrow (u, v) \in Reach \wedge (v, u) \in Reach$$

In un grafo non orientato non c'è differenza tra $Reach$ e $MReach$ (questo perché $MReach$ è la più grande relazione simmetrica contenuta in $Reach$, ma in un grafo non orientato $Reach$ rispetta già questa proprietà). È possibile vedere ogni componente fortemente connessa come un unico nodo, considerando quel nodo come *rappresentante* dell'annessa classe d'equivalenza, in quanto basta accedere a quella componente per accedere a tutti i nodi di quella classe d'equivalenza.

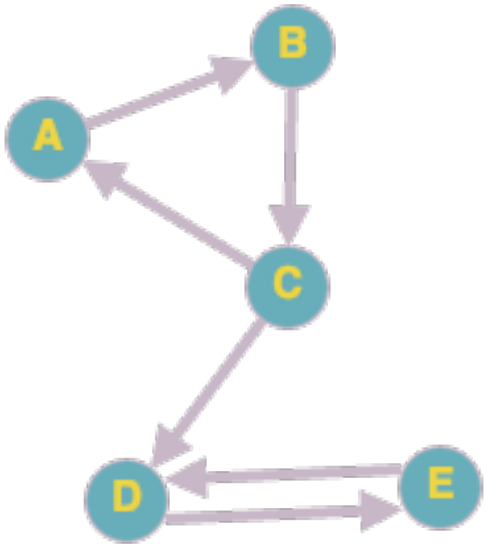


Figure 27: In questo grafo orientato $MReach$ distingue due classi di equivalenza (e due componenti fortemente connesse), quella composta dal sottografo coi nodi A, B, C e quella composta dal sottografo coi nodi D, E.

6.4.9 Distanza tra 2 vertici

La distanza è definita come il numero minimo di archi da attraversare per arrivare da un vertice all'altro (si definisce cioè in termini di distanza più breve). La distanza si indica con $\delta(v, u) = n$. Il problema della distanza è intrinsecamente legato al problema della raggiungibilità.
Se due nodi w e x non sono raggiungibili tra di loro, la loro distanza è $\delta(w, x) = \infty$. Si usa la nozione di ∞ per indicare due nodi non raggiungibili tra di loro.

6.4.10 Considerazioni per grafi con infiniti percorsi e raggiungibilità

Il problema della raggiungibilità esteso ad un numero infinito di percorsi (come è ammissibile in un grafo) potrebbe rendere il problema non calcolabile o difficoltoso dal punto di vista computazionale. È possibile considerare un insieme finito di percorsi da verificare per poter garantire una risoluzione al problema della raggiungibilità. Un percorso non semplice v, z, z, v ha sicuramente un ciclo. E' possibile creare un percorso che da z va direttamente a v, ovvero un percorso in cui il ciclo viene percorso 0 volte, eliminando effettivamente le due occorrenze di z. *Per risolvere il problema*

della raggiungibilità è sufficiente limitarsi ai percorsi semplici. Tuttavia, risolvere il problema della raggiungibilità con un algoritmo di tipo *brute force* non è efficiente dal punto di vista computazionale, ma è possibile definire varianti più efficienti.

6.4.11 Visita di un grafo

Visita in ampiezza È possibile rapportare la visita in ampiezza effettuabile in un albero ad un grafo, considerandola come una visita a distanze crescenti da un nodo. In particolare, dato un vertice, la visita comincerà da quel vertice e continuerà per distanze crescenti dal nodo fornito in partenza.
December 7th, 2018

Implementazione Questo algoritmo calcola l'insieme di tutti i vertici raggiungibili, ovvero:

$$R(s) = \{u | (s, u) \in Reach\}$$

Si distingueranno i vertici con dei colori in base allo stato in cui si trovano durante la ricerca:

- I vertici **scoperti** saranno colorati di grigio;
- I vertici **non scoperti** saranno colorati di bianco;
- I vertici **visitati** di nero.

NOTA: sono stato assente per circa i primi 20 minuti di lezione, integrare.

Listing 41: Implementazione della breadth-first-search in un grafo

```
1  BFS(G: grafo, S: vertice) {con S appartenente a r, G grafo}
2    for v in V do
3      C[v] = b {colore uguale a bianco}
4    Q = {S} {coda contenente solo il vertice passato in input}
5    C[S] = g {vertice scoperto}
6    while Q != EmptySet do
7      v = Testa(Q)
8      for each u in ADJ(v) do {vertici adiacenti}
9        if C[u] = b then {se e' bianco}
10         Q = Accoda(Q, v)
11         C[u] = g {allora diventa grigio e va in coda}
12     Q = Dequeue(Q)
13     C[v] = n {visitato}
```

Ora si introduce un vettore chiamato "vettore delle stime" che conterrà le distanze:

$$d : V \to N, \forall v \in V, d[v] = \delta(s, v)$$

Consultare la registrazione e le slide in http://wpage.unina.it/benerece/LASD−2016/6−1%20Grafi%20+%20Esercizio%205.pdf per l'integrazione delle distanze.
December 11, 2018

NOTA: prima ora di lezione mancante. Integrare. Recap BFS?

Visita in profondità Come si può immaginare, la visita in profondità dal punto di vista concettuale procederà in maniera analoga a quella che si effettua sugli alberi: si cercherà di visitare i vertici seguendo un percorso alla volta, fin quando è possibile. La definizione "possibile" era molto semplice negli alberi, in quanto era possibile seguire un percorso finché tale percorso non terminasse. Nei grafi la questione si complica nel caso in cui ci siano cicli, e andranno affrontati gli stessi problemi affrontati con l'algoritmo di BFS.

Listing 42: Implementazione ricorsiva della visita in profondità di un grafo

```
1  DFS_Visit(G: grafo, s: vertice)
2    {il primo step e' colorare s di grigio perche' e' il nuovo
         vertice che si sta visitando}
3    C[s] = grigio
4    d[s] = Tempo; Tempo = Tempo + 1
5    {adesso si scorrono gli adiacenti}
6    for each in v in Adj(s) do {esplorazione archi uscenti dal
         vertice}
7      {se il vertice v non e' bianco, allora o e' gia' stato
          visitato oppure se non e' bianco e' gia' stato
          incontrato}
8      if C[v] = bianco then
9        DFS_Visit(G, v) {chiamata ricorsiva sul vertice bianco
             }
10       {alla terminazione si assegna al vettore dei
             predecessori il vertice che l'ha scoperto}
11       P[v] = s
12   {quando sono stati visitati tutti gli adiacenti, si chiude
         la visita del vertice assegnando lo stato di terminato}
13   C[s] = nero {colore di s diventa nero cioe' visitato}
14   f[s] = Tempo; Tempo = Tempo + 1
```

L'algoritmo è quasi identico alla visita in profondità di un albero dove il caso base (quello delle foglie) è il caso in cui il corpo del **for** each non è mai eseguito non avendo nodi adiacenti. Il libro fornisce una versione della DFS che prende in input il grafo e lo visita completamente:

Listing 43: Implementazione ricorsiva della visita in profondità di un intero grafo

```
1  DFS(G)
2    for each v in V do {parte di inizializzazione}
3      C[v] = bianco
4      P[v] = NULL
5    for each v in V do
6      {qui vengono selezionate sorgenti di visita in
          profondita'}
7      if C[v] = bianco then
8        DFS_Visit(G, v)
9        {terminata questa visita, tutti i vertici saranno o
            bianchi o neri perche' per uscire da ogni chiamata
            ricorsiva il vertice corrispondente diventa nero}
```

Si può osservare che la visita, a seconda del nodo passato in input può avvenire in modo diverso e una chiamata a DFS_Visit non è garantito che visiti l'intero grafo. Poiché le distanze sono una proprietà del grafo e non della rappresentazione, le distanze saranno quelle. Ciò che cambia sono i percorsi minimi individuati: uno stesso vertice v raggiungibile da due percorsi con la stessa lunghezza potrebbe avere diversi percorsi minimi individuati. Ciò ha impatto sul contenuto dell'array dei predecessori. La coppia $(P[v], v)$ è necessariamente un arco del grafo.

Limitazioni della visita in profondità Questo algoritmo ha una natura molto diversa rispetto a quello di visita in ampiezza, e può essere usato per risolvere problemi diversi; se il problema è calcolare i percorsi minimi, questo algoritmo non garantisce che nell'array dei predecessori ci sia un percorso minimo. Quello che questo algoritmo non è in grado di fare è seguire percorsi definiti dallo sviluppatore–una DFS applicata ad un grafo può seguire percorsi casuali in quanto lo stesso grafo può essere rappresentato in molti modi diversi e ci sono altrettanti percorsi diversi. Siccome non può essere fatta nessuna assunzione sul modo in cui un grafo è rappresentato, dal punto di vista dello sviluppatore la DFS segue percorsi *a caso*.

Applicazioni della visita in profondità Uno dei problemi che può essere risolto efficientemente con una DFS è il verificare se un grafo è o meno aciclico (problema

che una BFS non è in grado di risolvere). Il motivo per cui la DFS può è che la visita in profondità segue i percorsi, quindi prende un percorso e comincia a seguirlo finché ha senso seguirlo. La visita in ampiezza estende tutti i percorsi portati avanti di uno, ma non ha modo di discriminare la presenza o meno di cicli.
December 13, 2018

Considerazioni sui tempi della visita in profondità Al termine della visita in profondità, una delle seguenti sarà vera $\forall u, v \in V$ (intervalli di inizio e fine visita):

1. $d[v] < d[u] < f[u] < f[v]$ o $d[u] < d[v] < f[v] < f[u]$
   Quando si verifica una di queste due, nella foresta ci sarà una relazione di discendenza opportuna tra due vertici.

2. $d[v] < f[v] < d[u] < f[u]$ o $d[u] < f[u] < d[v] < f[v]$

È impossibile che si verifichi per un qualsiasi ordine dei vertici d che i due intervalli si accavallino. Sarà implicito che $d[s] < f[v]$ sia vero. Se si rappresenta $d[v]$ come $(_v$ e $f[v]$ come $)_v$, la DFS ci dice che ad ogni parentesi aperta corrisponde una parentesi chiusa. Ciò significa che si può avere (**teorema della struttura a parentesi**):

$$(_v \cdots (_u \cdots)_u \cdots)_v$$

O anche:

$$(_v)_v(_u)_u$$

Ogni chiamata a DFS_Visit su un certo vertice s può solo modificare il tempo di inizio e fine visita di quel vertice e non di altri vertici. L'unica operazione che viene fatta su un vertice diverso è quella che imposta il predecessore di un vertice adiacente ad s. Affinché i tempi di visita possano essere modificati in cima allo stack dei record di attivazione deve essere presente un record associato ad una chiamata che ha ricevuto quel vertice in ingresso.
Si può procedere a dimostrare che non è possibile ottenere tramite questo algoritmo una sequenza di tempi del tipo $d[v] \to d[u] \to f[v] \to f[u]$. Nel momento temporale corrispondente a $d[u]$ lo stack conterrà $u \to \cdots \to v \to \cdots$. Da questa configurazione per arrivare ad una configurazione in cui v sia il top dello stack (per arrivare a $f[v]$), lo stack potrà essere configurato come uno dei seguenti casi:

1. $v \to \cdots$ (riportando sopra il v della prima chiamata eliminando il resto)

2. Aggiungendo in cima un altro v: $v \to \cdots \to u \to \cdots \to v \to \cdots$

Nel caso (1), significa che allora le chiamate ad u devono essere eliminate e quindi terminate, ma ciò provoca un *aggiornamento del tempo*, facendo quindi in modo che nella linea temporale prima di $f[v]$ ci sia $f[u]$, avendo una sequenza temporale $d[v] \to d[u] \to f[u] \to f[v]$ diversa da quella preposta. Per arrivare al caso (2), invece, affinché il record di attivazione di v possa essere inserito quel vertice deve essere necessariamente bianco; ma quando è stato inserito v la prima volta nello stack è diventato grigio e sarà non-bianco fino alla fine dell'algoritmo. È quindi impossibile avere una situazione come quella del caso (2), dimostrando che l'algoritmo di DFS non può realizzare una sequenza di tempi come quella considerata.

Corollario Si vuole dimostrare che $d[v] < d[u] < f[u] < f[v]$ se e solo se u è discendente di v nella foresta.

- $\Rightarrow$. Se lo stack dei record di attivazione contiene (ordine bottom-to-top) $s \to a \to b \to c \to d \to v$, allora esiste certamente un arco tra s ed a. In più, il predecessore di a sarà proprio s, ciò vuol dire che nell'albero che contiene s, a sarà sicuramente figlio di s. Lo stesso ragionamento vale tra a e b, avendo $P[b] = a$ e l'arco tra a e b viene inserito nella foresta. Nello stack dei record di attivazione è quindi contenuto uno dei percorsi del grafo che connette il vertice alla fine (s) con il vertice in cima allo stack (v) seguito dalla DFS. Ogni elemento nello stack corrisponde ad una chiamata ricorsiva non terminata, e tutti i vertici nello stack saranno quindi grigi.
  Se u viene scoperto all'interno dell'intervallo di v ($d[v] < d[u] < f[u] < f[v]$), lo stack sarà nell'istante $d[v]$ uguale a $\cdots \to v$. Nell'istante $d[u]$ sarà uguale a $\cdots \to v \to \cdots \to u$. Esisterà quindi un percorso nella foresta da v ad u testimoniando che u è discendente di v.

- $\Leftarrow$. Indicando con π il percorso che va da v (in alto nella foresta) ad u, nel caso $|\pi| = 0$, la tesi è banalmente falsa. Nel caso $|\pi| = 1$, quando l'algoritmo scopre v se u fosse non-bianco, allora non potrebbe essere discendente di v. Affinché u sia discendente di v, deve essere infatti vero $P[u] = v$—in questo caso, la tesi $d[v] < d[u] < f[u] < f[v]$ è verificata.
  Se $|\pi| > 1$, vuol dire che esiste un vertice z immediatamente precedente ad u (l'ultimo nodo nella foresta prima di u, ovvero $z \to u \in E$). Sarà vero che $d[v] < d[z] < f[z] < f[v]$ perché il percorso tra v a z è più piccolo–la proprietà sarà vera per induzione. Per quello visto prima, si avrà $d[v] < d[z] < f[z] < f[v]$. Dove si posiziona $d[u]$? Non può trovarsi dopo $d[v]$ in quanto u deve essere necessariamente scoperto dopo z. Se $P[u] = z$ (per ipotesi), vuol dire

che $d[u]$ deve trovarsi dopo $d[z]$, e per il teorema della struttura a parentesi allora si avrà $d[v] < d[u] < f[u] < f[v]$ (per transitività).

Teorema del percorso bianco Si vuole dimostrare che $\forall u, v \in V$ vale la seguente equivalenza: u è discendente di v nella foresta $\Leftrightarrow$ al tempo $d[v]$ esiste un percorso π in G che contiene solo vertici bianchi. L'istante $d[v]$ è l'istante in cui viene scoperto l'antenato. La raggiungibilità di un vertice ad un altro (cioè se da v esiste un percorso di u) non implica che u sia discendente di v. Si può procedere alla dimostrazione.

- $\Rightarrow$. Dire che u è discendente di v nella foresta vuol dire che deve esistere un albero che contiene v e contiene un percorso che raggiunga u. Con un vertice z immediatamente successivo a v (che si trova in cima alla foresta), si avrà sicuramente $d[v] < d[z]$. Ma questa proprietà è vera anche per il vertice immediatamente successivo a z, ovvero si ha $d[v] < d[z] < d[k] < \cdots < d[u] \cdots f[v]$. Ma quando $d[v]$ viene scoperto, affinché gli altri vertici vengano esplorati vuol dire che devono essere tutti *bianchi*, dimostrando l'implicazione verso destra della proprietà.

- $\Leftarrow$. Bisogna dimostrare che data l'esistenza di un percorso completamente bianco da v ad u allora la proprietà di discendenza è verificata. Supponendo che ci siano più percorsi bianchi che vanno da v ad u, si può intuire che la DFS sceglierà uno di quelli. Poiché non si può sapere quale verrà scelto, si dimostrerà invece che a prescindere dal percorso scelto tutti i vertici connessi da almeno un percorso bianco diventeranno discendenti di v. Per assurdo si supponga che almeno uno di questi vertici non sia un discendente, e che quindi u non sia discendente di v. Si sceglie il primo vertice t lungo la sequenza che non sia discendente, cioè t è il primo vertice di π che non diventa discendente. Si supponga che z sia il vertice immediatamente precedente a t e che quindi sia discendente di v. Si ha che $z \to t \in \pi, z$ e' discendente di v. Per il teorema della struttura a parentesi, sarà vero $d[v] < d[z] < f[z] < f[v]$. Quando viene scoperto t? All'istante $d[v]$, t è sicuramente bianco per ipotesi, e può quindi essere scoperto solo dopo v. Distinguiamo le possibili posizioni in cui t può essere scoperto:

  - Se venisse scoperto dopo $d[v]$ o $d[z]$ o $f[z]$, allora per il teorema della struttura a parentesi t è sicuramente discendente di v.
  - L'unico caso in cui non si avrebbe una contraddizione della tesi per assurdo è se t fosse scoperto dopo $f[v]$, ovvero: $d[v] < d[z] < f[z] < f[v] < f[t]$. In questo caso, significa che nell'istante $f[z]$ t sarà bianco, e che z termina

con uno degli adiacenti bianco. Ma prima di terminare deve analizzare tutti gli adiacenti di z e cambiare il colore. Non è infatti neanche possibile che venga scoperto prima di $f[v]$!

Poiché possono esserci più percorsi bianchi, la DFS seguirà solo uno di essi ed è possibile che t venga scoperto anche prima di $d[z]$ o prima di $f[z]$. In ogni caso, $d[t]$ sarà sicuramente contenuto nell'intervallo di v, provando che la tesi è assurda.

Tipologie di archi in una foresta Dato il grafo G, eseguendo la DFS uno dei possibili prodotti estraibili dall'esecuzione della DFS è la foresta contenuta nell'array dei predecessori. Tale foresta contiene alcuni degli archi di G: se $G = \langle V, E \rangle$, $F = \langle V, E' \rangle$ con $E' \subseteq E$. Gli archi che finiscono in F (**archi dell'albero**) sono gli archi che la DFS ha utilizzato per scoprire i vertici bianchi: (u, v) appartiene a questa classe se il colore di v è bianco quando viene esplorato l'arco (u, v) dalla DFS. Al contrario, nella foresta non si potranno mai trovare archi di questo tipo:

- Gli **archi in avanti** sono gli archi che se inseriti nella foresta connetterebbero un antenato con un discendente (che non è un figlio). Se il colore di v è nero e se v è discendente di $u \Rightarrow d[u] < d[v] < f[v] < f[u] \Rightarrow d[u] < d[v]$, l'arco (u, v) appartiene a questa classe.
- Gli **archi di ritorno** sono gli archi che se inseriti nella foresta connetterebbero un discendente con un antenato (che non è il padre). Se il vertice di arrivo v è grigio, l'arco (u, v) appartiene a questa classe.
- Gli **archi di attraversamento** sono archi che connettono due nodi dello stesso albero che non sono in relazione di discendenza l'uno con l'altro. Se il colore di v è nero e $d[u] > d[v]$, l'arco (u, v) appartiene a questa classe.

Si è adesso vicini a definire un algoritmo che permette di dire se un grafo contiene o meno cicli. La presenza di archi di ritorno sarà chiave nel permettere di dire ciò.

NOTA: Lezione di venerdì 14 dicembre mancante.

December 18, 2018

NOTA: primi 15-20 minuti di lezione mancanti.

6.4.12 Ordinamento topologico in un grafo

```
1  GradoEntrante(G, ge)
2    for each v in V do
3      ge[v] = 0
4    for each v in V do
5      for each u in Adj(v) do
6        ge[u] = ge[v] + 1
7  {restituisce una coda con tutti i vertici che hanno grado
       uguale a 0 in questo array}
8  {tempo lineare sull'insieme dei vertici -- Theta(|V|)}
9  InitQueue(G: grafo, ge: array)
10   Q = EmptySet
11   for each v in V do
12     if ge[v] = 0 then
13       Q = Enqueue(Q, v)
14   return Q
```

Adesso si può costruire l'algoritmo che calcola l'ordinamento topologico. La coda sicuramente non sarà vuota e si avrà almeno un elemento.

Listing 44: Algoritmo di ordinamento topologico

```
1  OrdTopologico(G)
2    GradoEntrante(G, ge) {in questo modo l'array ge viene
       popolato - costo: Theta(|V|+|E|)}
3    Q = InitQueue(G, ge) {generazione della coda con vertici
       con grado inizialmente uguale a 0 - costo Theta(|V|)}
4    while Q != EmptySet do {Theta(|V|+|E|)}
5      v = Testa(Q)
6      {v e' un vertice con grado entrante 0, per dimostrazione
         si stampera'}
7      Stampa(v)
8      for each u in Adj(v) do
9        {bisogna decrementare il grado entrante per ogni
           vertice adiacente}
10       ge[u] = ge[u] - 1
11       if ge[u] = 0 then
12         Q = Enqueue(Q, u)
13     Q = Dequeue(Q)
```

Il tempo complessivo dell'algoritmo è lineare sulla dimensione del grafo: $\Theta(|V|+|E|)$. L'ordinamento topologico inizia da un nodo che non ha nessun vincolo e man mano

che procede rilassa i vincoli sui nodi successivi in modo tale che possano essere selezionati. Un vertice non è vincolato da nessuno se non ha archi entranti, e un vertice non vincola nessuno se non ha archi uscenti. In tal caso, tale vertice può rappresentare la fine del grafo. Si effettuerà una DFS con la differenza che ogni volta che la DFS termina l'esplorazione di un vertice (colorandolo di nero) viene inserito come "ultimo elemento corrente"–si costruirà quindi l'ordinamento dalla fine verso l'inizio.

Listing 45: Versione dell'algoritmo di ordinamento topologico mediante DFS.

```
1  OrdTopologico2(G)
2    Init(G) {inizializzazione dei colori dei vertici - costo:
         Theta(|V|)}
3    O = EmptySet {struttura che contiene l'ordinamento,
         riempita come stack}
4    for each v in V do
5      if C[v] = b then
6        O = DFS_Visit2(G, v, O)
7    return O
8
9  DFS_Visit2(G, v, O)
10   C[v] = g
11   for each u in Adj(v) do
12     if C[u] = b then
13       O = DFS_Visit2(G, u, O)
14   C[v] = n
15   {in questo momento si sa che tutti i vertici raggiungibili
         da v sono neri}
16   O = Push(O, v)
17   return O
```

Il costo complessivo dell'algoritmo è $\Theta(|V| + |E|)$. Questo algoritmo è asintoticamente equivalente a quello precedente sia in termini di spazio che in termini di costo computazionale.

Dimostrazione di correttezza dell'algoritmo L'output dell'algoritmo è una sequenza di vertici $O = v_1, v_2, \ldots, v_n$ dove $n = |V|$. Con $v <_O u$ si indica che v compare in O prima di u. Bisognerà dimostrare la proprietà definitoria dell'ordinamento topologico:

$$\forall u, v \in V, (u, v) \in E \Rightarrow u <_O v$$

Considerando la versione della DFS che tiene conto dei tempi di visita–quindi nel momento in cui un vertice viene annerito e inserito in cima allo stack O viene anche assegnato il tempo di fine visita. Si può osservare che per come è costruito O vale la seguente proprietà:

$$\forall u, v \in V, u <_O v \Leftrightarrow f[v] = f[u]$$

Ciò è vero perché se u compare prima di v, vuol dire che è stato inserito nello stack dopo v. Ma se gli inserimenti dei vertici nello stack coincidono con l'assegnamento dei loro tempi di fine visita (che possono solo crescere), chi viene inserito per primo avrà tempo di fine visita più piccolo di quelli aggiunti successivamente. Poiché le due proprietà sono equivalenti, è possibile sostituirle nella tesi da dimostrare ottenendo:

$$\forall u, v \in V, (u, v) \in E \Rightarrow f[v] = f[u]$$

Un arco verrà attraversato nel momento in cui v è grigio, pertanto tra $d[u]$ e $f[u]$ avverrà l'attraversamento dell'arco. Quando l'arco u viene attraversato e si giunge a v, la DFS può trovare un vertice con uno dei possibili tre colori.

- Se v fosse bianco vuol dire che ci sarà una chiamata ricorsiva su v e $d[v]$ si troverà sicuramente tra $d[u]$ e $f[u]$. Per il teorema della struttura a parentesi, si avrà quindi $d[u] < d[v] < f[v] < f[u]$. Questo ci dice che $f[v] < f[u]$, e quindi la condizione desiderata è soddisfatta.

- Se v fosse nero, $f[v]$ sarà sempre minore di $f[u]$ a prescindere di dove inizi esattamente.

- Se v fosse grigio, significa che il vertice è stato scoperto prima di v e terminerà successivamente. Si avrà quindi una sequenza temporale $d[v] < d[u] < f[u] < f[v]$. $f[u] < f[v]$ viola la condizione. Questo, tuttavia, può succedere soltanto in grafi arbitrari–perché tale condizione può accadere se e solo se è presente un ciclo nel grafo. v non può quindi essere grigio in un grafo aciclico.

La tesi è stata quindi dimostrata. ■

6.4.13 Calcolo componenti (fortemente) connesse

Ricordando le definizioni dette precedentemente: G' è una componente (fortemente) connessa di un grafo non orientato (orientato) G se G' è un sottografo (fortemente) connesso massimale di G. Un grafo connesso è un grafo che ha un percorso per raggiungere ogni vertice. Le componenti connesse di un grafo rappresentano una classe di equivalenza. Si vogliono calcolare le componenti connesse e fortemente connesse.

- In un grafo non orientato (considerando quindi componenti connesse), effettuando una DFS classica si può procedere ad analizzare la foresta costruita (FDF). Se due nodi di un grafo non orientato finiscono in due alberi diversi, allora necessariamente fanno parte di due componenti connesse differenti. Se infatti esistesse un arco esterno che connette due nodi tra due alberi allora la DFS avrebbe percorso l'arco e visitato i nodi annessi, annerendolo, cosa che è falsa per ipotesi. Bisogna dimostrare che ogni albero rappresenta una e una sola componente connessa. Prendendo un qualsiasi albero costruito da una DFS su un grafo non orientato, si sa che tutti i vertici presenti nell'albero sono raggiungibili dalla radice. Poiché dalla radice è possibile visitare tutti i vertici nell'albero, è vero anche il contrario: i vertici sono reciprocamente raggiungibili, ed appartengono quindi alla stessa componente connessa. Ogni albero è un insieme di nodi connessi nel grafo ed è massimale. Usando la BFS invece della DFS, si potrebbe risolvere ugualmente il problema in quanto l'albero costruito è l'albero dei percorsi minimi dalla sorgente che possono sempre essere attraversati in entrambi le direzioni e per questo specifico problema hanno le stesse proprietà di quelli costruiti dalla DFS.
  December 20, 2018

- In un grafo orientato, si può considerare il grafo delle componenti fortemente connesse. Non è possibile che nel grafo delle componenti connesse esistano dei cicli, in quanto se contenesse dei cicli vorrebbe dire che le componenti all'interno non sono massimali e che non fanno parte della stessa classe di equivalenza. Quindi, il *grafo delle componenti connesse* è sempre aciclico. Si vuole dimostrare la seguente proprietà: sia C una componente fortemente connessa di un grafo $G = \langle V, E \rangle$ e $u, v \in C$. Allora vale che ogni percorso $\pi = u \to v$ (che parte da uno dei due vertici e arriva all'altro vertice) è tale che ogni vertice di π è contenuto nella componente ($\pi \subseteq C$). Per la dimostrazione consultare la registrazione video che contiene i disegni necessari per comprenderla. In conclusione, se C è una componente fortemente connessa di G e $u, v \in C$ allora al termine di $DFS(G)$, u e v sono contenuti nello stesso albero della foresta. L'idea è avere un grafo G con una componente fortemente connessa C con un numero arbitrario di vertici. Supponendo che il primo vertice scoperto dalla DFS di C sia v_1, in quell'istante tutti gli altri sono bianchi. Si conosce inoltre che v_1 può visitare tutti i vertici presenti, e tutti questi percorsi non possono uscire da C. Ma la componente in questo momento è tutta bianca, e il percorso da v_1 a u il teorema del percorso bianco dice che u sarà discendente di v_1. Adesso si sa che la DFS ha costruito un certo insieme di alberi e che le

componenti si trovano tutte in un albero. Come è possibile separare le componenti che potrebbero stare nello stesso albero? Un grafo con un solo nodo senza archi è una *componente fortemente connessa banale* ed è l'unica componente fortemente connessa senza cicli. Si supponga che la DFS lanciata sul grafo G costruisca un solo albero T_s–esso sarà radicato in un certo vertice s, la sorgente della visita in profondità. Se $v \in T_s \Rightarrow (s, v) \in Reach$, ma se $v \in T_s$ non si conosce se $(v, s) \in Reach$. Se si potesse verificare la seconda proprietà, allora i due nodi sarebbero mutualmente raggiungibili nel grafico e quindi (v, s) sarebbe una componente connessa. Verificare ciò è un'operazione costosa–bisognerebbe sbiancare nuovamente tutto l'albero ed effettuare la DFS. Con un numero limitato di visite si riescono ad identificare i vertici che nel grafo originale hanno un percorso che va a v?

Grafo trasposto È possibile farlo con una sola DFS osservando che dato un grafo G si può definire la nozione di "grafo trasposto" G^T:

$$(u, v) \in E \Leftrightarrow (v, u) \in E^T$$

Se G^T è il grafo trasposto di G ed esiste un percorso in G^T da u a v allora esiste un percorso in G da v a u. Effettuando la DFS su G^T, se nell'albero generato ho un percorso da s ad u allora si è certi che da u si può giungere ad s nel grafo originale. La visita raggiunge tutti i vertici del grafo che possono raggiungere s, e raggiungerà solo i vertici di T_s che sono mutualmente raggiungibili. Se invece ci sono due alberi radicati in s_1 ed s_2, gli unici archi che potrebbero causare problemi al sistema precedente sono gli archi di attraversamento da un albero a un altro, in quanto una volta fatto il grafo trasposto tale nodo cambierebbe verso e la DFS fatta sul nodo del primo albero non riuscirebbe mai a raggiungere il secondo. Come si può forzare la seconda DFS a non muoversi mai dall'albero della prima DFS in cui parte? Ciò è fattibile osservando una proprietà degli archi di attraversamento.

Proprietà archi di attraversamento Considerando la prima DFS eseguita sul grafo, essa costruirà un certo insieme di alberi. Si può trovare un albero in cui la seconda DFS sul grafo non esce. *Integrare con video se necessario.* Si può ora procedere a stendere l'algoritmo.

Listing 46: Costruzione del grafo trasposto

```
1 GrafoTrasposto(G)
```

```
2     V_T = V
3     for each v in V_T do
4       for each u in Adj[v] do
5         Adj_T[u] = Add(Adj_T[u], v)
6     Return (<V_T, Adj_T>)
```

Listing 47: Algoritmo per ricavare le componenti fortemente connesse

```
1  DFS2(G, L)
2     Init(G)
3     for each v in L do
4       if C[v] = b then
5         DFS_Visit(G, v)
6  CFC(G)
7     L = DFS1(G)
8     G_T = GrafoTrasposto(G)
9     DFS2(G_T, L)
```

Fine delle lezioni. Congratulazioni!