Algoritmi e Strutture Dati 2023-24
(M. Benerecetti)

Contents

1 Lezione 02 - 15/09/2023

1.1 Algoritmo di Conteggio

Descrivere un algoritmo che accetta come input un intero $N \geq 1$ e produce in output il numero di coppie ordinate $i, j \in \mathbb{N} \quad (i,j) : 1 \leq i \leq j \leq \mathbb{N}$
Esempio:

- Input:N=4
- Output:10 {(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,3),(3,4),(4,4)}

```
1 Conta(N):
2   ris = 0; //Assegnamento costante (1 operazione elementare)
3   for i=1 to N do //Assegnamento/Incremento + confronto (2 op.
       elementari)
4     for j=1 to N do //idem for di sopra
5       if i<=j then //2 letture + confronto (3 op. elementari)
6         ris = ris+1 //lettura+scrittura+assegnamento (3 op. elementari)
7   return ris //1 op. elementare
```

Ogni riga ha un costo che corrisponde alle operazioni elementari effettuate, mediamente ogni op. elementari ha un costo di 1 unità di tempo, prendiamo come esempio il for al primo giro: Assegnamento + Confronto (2 op. elementari), invece i successivi giri: Incremento+confronto (2 op. elementari).

Ognuna di queste operazioni (righe) vengono eseguite più di una volta, quindi il costo sarà maggiore, andiamo ad esprimerlo:

- 2) Costo = 1 (fuori dal ciclo)
- 3) La testa viene eseguita $n+1$ volte poiché abbiamo anche l'ultima operazione per uscire dal ciclo, quindi Costo $= 2 * (n+1) = 2 * \sum_{i=1}^{N+1} 1$
- 4) Questo for verrà ripetuto N volte poiché il corpo del for viene eseguito N volte, quindi il suo costo sarà:

$$\underbrace{2}_{\text{costo dell'operazione}} * \underbrace{\sum_{i=1}^{N}}_{\text{for esterno}} \underbrace{\sum_{j=1}^{N+1} 1}_{\text{for interno}}$$

- 5) L'if stando in entrambi i for avrà un costo di: $3 * \sum_{i=1}^{N} \sum_{j=1}^{N+1} 1$
- 6) Questa operazione non ha un numero fisso di volte di esecuzione. Pertanto e necessario stabilirne un algoritmo per decretarne il numero. Pensandoci il numero di volte che questa operazione esegue dipende da N e dall' i fissate in precedenza. Calcolando, anche banalmente a mano, quante operazioni vengono eseguite ci troveremo con:

$$N - i + 1 \text{volte che l'operazione viene eseguita.}$$

- 7) Costo = 1 (fuori dal ciclo)

Dopo che viene effettuata l'analisi, possiamo andare a sommare tutti i risultati che abbiamo ottenuto in termini di unita (correggere accento) di tempo.
La funzione $T(n)$ e(correggere) la funzione che ci tiene traccia della complessita dell'algoritmo.

Andiamo semplicemente a sommare i nostri risultati di ogni riga:

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

Questo risultato e ottenuto semplificando le nostre sommatorie:

- 3) $2 * \sum_{i=1}^{N+1} 1 = 2 * (N+1)$
- 4) $2 * \sum_{i=1}^{N} \sum_{i=1}^{N+1} 1 = 2 * \sum_{i=1}^{N} N + 1 = 2 * (N^2 + N)$
- 5) $3 * \sum_{i=1}^{N} \sum_{i=1}^{N} 1 = 3 * \sum_{i=1}^{N} N = 3N^2$
- 6) $\sum_{i=1}^{N}(N-i+1) = N-(k-1)$ cioe ad ogni ciclo il numero delle volte che viene eseguita questa operazione diminuisce costantemente di 1 (cioe dipendente dal salire di i).

$$T(n) = \frac{13}{2}N^2 = \frac{9}{2}N + 4$$

Come vediamo questa funzione e quadratica, quindi cresce esponenzialmente nel tempo, molto pesante e lenta come funzione.

1.1.1 Algoritmo di conteggio v2

Dopo aver ottenuto i risultati dell'analisi sopra, possiamo dire che e sicuramente possibile semplificare il nostro codice in modo tale da far eseguire meno operazioni al nostro processore e quindi utilizzare meno tempo.

```
1 Conta(N):
2   ris = 0;
3   for i=1 to N do
4     ris = ris + (N-i+1)
5   return ris
```

Cosi facendo abbiamo semplicemente detto al nostro codice che deve sommare soltanto gli elementi che nel momento in cui i e fissato, sono $\leq$ di se stesso.
Cosi facendo si dovrebbero eliminare molte operazion inutili, analizziamo.

- 2)sempre 1 operazione
- 3) $2 * \sum_{i=1}^{N+1} 1$
- 4) $\sum_{i=1}^{N} 7 = 7 \cdot N$

Come possiamo osservare abbiamo eliminato il secondo for, dunque abbiamo eliminato la quadraticita, ora l'operazionea riga 4 viene eseguita solamente N volte, e il numero di operazioni semplici che esegue e fissato.

$$T(n) = 1 + 2(N+1) + 7N + 1 = 9N + 4$$

1.1.2 Algoritmo di conteggio v3

Da come possiamo notare e possibile di nuovo semplificare l'ultima sommatoria della riga 4 dello scorso algoritmo.
Da

$$\sum_{i=1}^{N} N - i + 1 \rightarrow \sum_{i=1}^{N} i$$

Il risultato della sommatoria e lo stesso, se andiamo a semplificarlo.

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

```
1 Conta(N):
2   ris = 0
3   ris = ris+(N-i+1)
4   return ris
```

In questo modo abbiamo eliminato qualsiasi ciclo e quindi il risultato sara un numero fisso di operazioni.

- 1) 1 operazione
- 2) 5 operazioni elementari

$$T(n) = 5 + 1 = 6$$

In questo caso la funzione tempo per eseguire queste operazioni e costante, non dipendente da nessun N, dunque e la migliore soluzione possibile per questo algoritmo.

1.2 Algoritmo della massima sottosequenza contigua

Preso un array di n elementi, vorremmo provare a trovare la sottosequenza la cui somma di tutti i valori e massima.

Come fare? La soluzione piu naive possibile e effettuare 3 cicli for innestati:

```
1 int Max_seq_sum_1(int N, array a[])
2   maxsum = 0
3     for i=1 to N
4       for j=i to N
5         sum = 0
6         for k=i to j
7           sum = sum + a[k]
8         maxsum = max(maxsum,sum)
9   return maxsum
```

Come possiamo notare l'avere 3 for innestati ci fa avere una complessita computazionale di N^3, comunemente rappresentato dalla formula della notazione asintotica $O(N^3)$.

1.2.1 Algoritmo v2

Come e facile notare, nel terzo for innestato c'e una ripetizione abbastanza inutile di operazioni che effettuiamo per andarci a sommare i valori delle sottosequenze. In particolare andiamo a ripetere scorrere piu volte lo stesso numero di celle, solamente per trovare la sottosequenza poco piu grande (a volte anche di una cella).

Gli stessi valori delle sottosequenze possono essere semplicemente trovati scorrendo avanti l'indice e sommando alla sequenza precedente il valore successivo. Analiticamente ci troveremmo che:

$$\sum_{k=i}^{j+1} a_k = a_{j+1} + \sum_{k=i}^{j} a_k$$

Il valore sum rimarra inalterato, ma verra solamente aggiornato del valore successivo. Il codice si presentera in questo modo:

```
1 int Max_seq_sum_1(int N, array a[])
2   maxsum = 0
3     for i=1 to N
4       sum = 0
5       for j=i to N
6         sum = sum + a[j]
7         maxsum = max(maxsum,sum)
8   return maxsum
```

2 Lezione 03 - 21-09-2023

2.1 Algoritmo v3

L'algoritmo può essere anche migliorato, riusciendo ad arrivare ad una complessità **lineare**, nel seguente modo:

```
int Max_seq_sum_3(int N, array a[])
  maxsum = 0
  sum = 0
  for j=1 to N
    if (sum + a[j] > 0) then
      sum = sum + a[j]
    else
      sum = 0
    maxsum = max(maxsum,sum)
  return maxsum
```

Il ragionamento è il seguente: Se prendiamo un insieme di numeri da sommare, (da i ad a), possiamo controllare se esso è positivo o negativo. Nel caso in cui $\sum_{e=i}^{a} A[e]$ risultasse positiva, andiamo a espandere il nostro range fintantochè il risultato della sommatoria riamanga positivo. Nel caso in cui invece il risultato fosse negativo, non ci conviene tenere traccia dei numeri più piccoli di quel range, dato che se quella sommatoria è minore del numero successivo alla sommatoria, non ha senso tenerne conto. E quindi invece ha senso tenere traccia del numero successivo. Da quel numero poi sommare i numeri successivi continuando il processo sopracitato.

2.2 Strutture Dati - Insieme Dinamico

Vediamo come rappresentare un insieme di dati dinamico S (con insieme dinamico si intende una collezione di elementi variabile nel tempo, quindi è possibile aggiungere o rimuovere elementi);

$$S = \{a_1, a_2, ..., a_n\} \quad n \geq 0$$

Andiamo a definire alcune operazioni:

- Insert$(S, a) \rightarrow S'$ $(S' = S \cup \{x\})$
- Deletes$(S, a) \rightarrow S'$ $(S' = S \backslash \{x\})$
- Search$(S, a) \rightarrow \{True, False\}$
- Massimo$(S) \rightarrow a$
- Minimo$(S) \rightarrow a$
- Successore$(S, a) \rightarrow a'$
- Predecessore$(S, a) \rightarrow a'$

3 Lezione 04 - 23/09/2023

3.1 Array non ordinato

Abbiamo un insieme S che vogliamo rappresentare in A.

$$|S| = A.free$$

La funzione $A.free$ ci restituirà la posizione della prima cella libera.

3.1.1 Ricerca

Dato un array e un elemento da cercare, restituisce la posizione in cui è presente il valore

```
Search(A,e) //A: array in input, e: elemento da cercare
  pos = A.free-1
  while A[pos] != e and pos >= 0 do
    pos = pos-1
  return pos
```

Ci posizionamo all'ultima cella piena e mano a mano tornando all'indietro andiamo a cercare il valore che abbiamo in input, se non è presente nell'array andiamo a restistuire il valore -1. La complessità computazione sarà **lineare** nello specifico:

$$T_s(n) = c \cdot n$$

Dove c è il costo fisso delle operazione e n è quante volte si ripete il ciclo.

3.1.2 Inserimento

Dato un array e un elemento da inserire, andiamo a verificare tramite la funzione **ricerca** se l'elemento non sia già presente poiché non vogliamo elementi duplicati.

```
Inserimento(A,e) //A: array in input, e: elemento da cercare
  pos = search(A,e) //-1: non trovato, >=0: indice del valore
  if pos = -1 then
    if(A.free < length(A)) then
      A=resize(A)
    A[A.free]=e
    A.free=A.free+1
```

La complessità computazione sarà **lineare** nello specifico:

$$T_i(n) = 2c \cdot n + c'$$

3.1.3 Cancellazione

Dato un array e un elemento da cancellare, andiamo a verificare tramite la funzione **ricerca** se l'elemento sia presente in modo da portelo eliminare.

```
Delete(A,e) //A: array in input, e: elemento da cercare
  pos = search(A,e) //-1: non trovato, >=0: indice del valore
  if pos >= 0 then //Elemento trovato
    A[pos]=A[A.free-1]
    A.free=A.free-1
```

3.2 Array Ordinato

Uno dei grosso problemi dell'array visto in precendenza era la ricerca, poiché al più avevo costo **lineare** cioè la lunghezza di tutto l'array, con l'array ordinario andiamo a sopperire a questo problema ma ovviamente aggiungendone degli altri.

3.2.1 Ricerca Binaria

La ricerca binaria è un algoritmo applicabile solo ad array ordinato che permette una ricerca molto rapida.
L'algoritmo è simile al metodo usato per poter trovare una parola sul dizionario: sapendo che il vocabolario è ordinato alfabeticamente, l'idea è quella di iniziare la ricerca non dal primo elemento, ma da quello centrale, cioè a metà del dizionario. Si confronta questo elemento con quello cercato:

- se corrisponde, la ricerca termina indicando che l'elemento è stato trovato;
- se è superiore, la ricerca viene ripetuta sugli elementi precedenti (ovvero sulla prima metà del dizionario), scartando quelli successivi;
- se invece è inferiore, la ricerca viene ripetuta sugli elementi successivi (ovvero sulla seconda metà del dizionario), scartando quelli precedenti.

Andiamo a definirlo per ricorsione:

```
BinSearch(A, e, i, j) //i: punto inizio, j: punto fine
  if i <= j then
    q=(i+j)/2 //punto centrale
    if A[q] > e then
      i=BinSearch(A, e, i, q-1) //ricerca a "sinistra"
    else if A[q] < e then
      i=BinSearch(A, e, q+1, j) //ricerca a "destra"
    else return q //trovato
  else return -1 //non trovato
```

Per studiare la complessità di questo algoritmo (ricorsivo), andiamo a rappresentare le varie chiamate tramite un "albero degenere" in cui ogni nodo ha un solo figlio e così via, il costo di ogni nodo sarà c, ad ogni chiamatà andiamo a dimezzare la lunghezza dell'array $n...n/2...n/4....$ il massimo delle chiamate possono essere $h+1$ dove h è

l'altezza dell'albero andiamo a generallizare con:

$$\frac{n}{2^h} \qquad h : \text{è il numero dei richiami}$$

Dobbiamo trovare il valore di h per il quale l'intervallo si riduce a 1 elemento, in quanto l'elemento desiderato sarà stato trovato. Quindi, dobbiamo risolvere l'equazione:

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

Per risolvere questa equazione per h, possiamo moltiplicare entrambi i lati per 2^h:

$$n = 2^h$$

Ora, per isolare h, possiamo applicare il logaritmo in base 2 ad entrambi i lati:

$$h = log_2(n)$$

Quindi, il numero di chiamate ricorsive necessarie per trovare l'elemento desiderato è logaritmico rispetto alla dimensione dell'array n. Pertanto, la complessità computazionale dell'algoritmo di ricerca binaria è $O(logn)$ nel caso peggiore.
N.B Una complessita computazionale logaritimica è più efficente di una lineare.

3.2.2 Inserimento/Cancellazione nella ricerca binaria

Abbiamo visto come in un array ordinato la ricerca (binario) è molto efficemente ma abbiamo come contro che l'inserimento/cancellazione hanno costo peggiore poiché dobbiamo mantenere l'ordinamento, quindi se vogliamo inserire un elemento dobbiamo andare a spostare i valpri per creare un posto disponibile.

4 Lezione 05 - 26-09-2023

4.1 Ricerca Binaria Iterativa

Proviamo a riscrivere la ricerca binaria su un array ordinato in maniera iterativa:

```
BinSearchIterative(A,e) //A: array in input, e: elemento da cercare
  ret=-1
  i=0
  j=length(A)-1
  while i <=j AND ret=-1 do
    q=(i+j)/2 //punto medio
    if A[q] < k then
      i=q+1 //ricerca a "destra"
    else if A[q] > k then
      j=q-1 //ricerca a "sinistra"
    else
      ret=q //trovato
  return ret
```

Possiamo notare che a meno di piccoli cambiamenti il funzionamento di questo algoritmo rispetto alla sua versione ricorsiva è pressoche identico.

4.2 Somma

Andiamo a creare un algoritmo per sommare tutti i valori presenti in un array tramite ricorsione.
Come tutti gli algortmi ricorsivi dobbiamo andare a trovare una base di induzione (caso base) e poi un passo di induzione, rappresentiamolo tramite sommatorie:

$$\sum_{i=i}^{n} i = \begin{cases} 0 & \text{se } n = 0 \quad \text{base induzione} \\ n + \sum_{i=1}^{n-1} i & \text{se } n \geq 0 \quad \text{passo induttivo} \end{cases}$$

Ora che abbiamo capito il ragionamento andiamo a scrivere l'algoritmo

```
Sum(A,i,n) // i=inizio, n=fine
  if n=0 then
    ret = 0
  else
    x=sum(A,1,n-1)
    ret=n+x
  return ret
```

4.3 Lista Linkata

La lista rappresenta una struttura dati dinamica dove le operazioni di inserimento e cancellazione sono meno dispendiose, a differenza di quanto accade negli array (che sono implementati come struttura statica, il che rende problematiche le suddette operazioni).
La lista condivide con l'array la proprietà di linearità (o sequenzialità) ma è una struttura più flessibile poiché non richiede la contiguità in memoria come l'array. La lista permette di avere gli elementi in una qualsiasi area di memoria rendendo le operazioni di inserimento e cancellazione eseguibili in tempo costante; infatti, la sua struttura è composta da nodi, i quali contengono sia un certo dato (key), sia un'informazione su dove si trovi il nodo successivo (next).

4.3.1 Ricerca (interativo)

Andiamo a definire un algoritmo di ricerca su lista linkata tramite iterazione

```
Search(L,k) // L=lista, k=elemento da cercare
  ris=-1
  temp=L // "puntatore" al primo elemento della lista
  while temp != NIL and ris=-1 do
    if temp->key = k then
      ris=temp
    else
      temp = temp->next
  return ris
```

L'algoritmo è molto semplice tramite la variabiale "temp" andiamo a scorrerci tutta la lista tramite i puntatori **next**, se troviamo il valore lo restituiamo il nodo in cui è presente altrimenti restituiamo -1.

4.3.2 Ricerca (ricorsiva)

Andiamo a rifare lo stesso algoritmo ma tramite la ricorsione

```
Search(L,k) // L=lista, k=elemento da cercare
  ris=NIL
  if L != NIL then
    if L->Key = k then
      ris=L
    else
      ris=search(L->Next, k)
  return ris
```

In questo caso ad ogni prossima chiamata di "search" andiamo a passare una "sottolista" cioè la stessa ma partendo da un nodo più avanti fino ad arrivare alla sua fine.
Andiamo a calcoalre la complessità computazione, dato che ad ogni chiamata andiamo a passare la lista ma ridotta di un nodo avremo $n, n-1, n-2, n-i$ dove i sarà il numero di nodi, quindi abbiamo la sequenza dei primi i numeri naturali (formula di

Gauss) quindi l'algoritmo ha complessità **lineare**.

4.3.3 Ragionamento Ricerca Binaria su Lista Ordinata

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

5 Lezione 06 - 29/09/2023

Continuiamo i nostri algoritmi sulle liste linkate

5.1 Inserimento

Andiamo a scrivere un algoritmo per l'inserimento in testa, cioè andiamo ad "attaccare" il nostro nuovo nodo all'inizio della lista, è il tipo di inserimento più semplice perché non prevede particolari modifiche alla struttura.

Funzione Ausiliaria Useremo una funzione ausiliare per aiutarci in tutte le situazione di creazioni di un nuovo nodo:

```
newNode(K,L) //K: Elemento, L: Lista
  temp = allocanodo() //tipo una malloc
  temp->key = k //assegnamento valore
  temp->next = L //attacchiamo il nodo al primo elemento della lista
     in input
  return temp
```

Ora che abbiamo definito la nostra funzione ausiliaria andiamo a scrivere l'algoritmo di inserimento

```
Insert(L,K)
  ret = search(L,K) //cerchiamo il valore
  if ret=NIL then //se non e' presente lo inseriamo
    L=newNode(K,L)
  return L
```

5.2 Inserimento Ordinato (iterativo)

Per strutturare al meglio le liste possiamo prevedere un inserimento ordinato, fare questo nelle liste è molto più efficiente rispetto a un array poiché non dobbiamo spostare tutti i valori per fare spazio al valore da inserire ma possiamo banalmente staccare i puntatori e "riattaccarli" nel modo corretto:

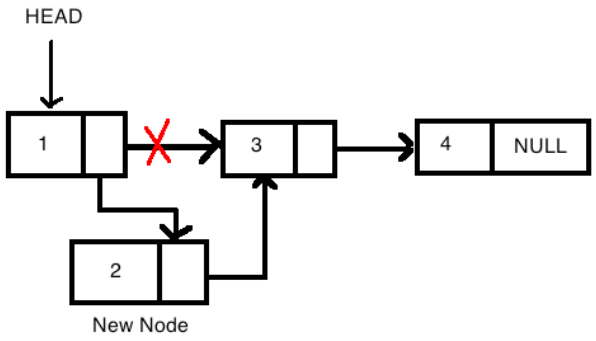


Fig. Insert after a given Node

Andiamo a definire l'algoritmo in maniera iterativa:

```
InsertO(L,K)
  temp=L //salviamo il puntatore al primo elemento
  P=NIL //creiamo una variabile di appoggio
  while [temp != NIL and temp->key < k] do
    P=temp //mettiamo P sul nodo "precedente" a dove inseriamo
    temp=temp->next //mettiamo temp sul nodo "successivo"

  if [temp = NIL or temp->key > k] then
    new=newNode(k,temp) //creiamo il nodo attaccandolo al "successivo"
    if P != NIL then //se "esiste" il precedente allora
      P->next = new //attacchiamo il "precedente" al nuovo
    else //se non esiste
      L=new //spostiamo la testa al nuovo valore

  return L
```

5.3 Inserimento Ordinato (ricorsivo)

Come sempre per scrivere un buon algoritmo ricorsivo bisogna ragionare per casi, andiamo ad esaminare i vari possibili:

- Inserimento in testa (valore minimo)
- Inserimento "centrale" (valore compreso tra due numeri)
- Inserimento in coda (valore massimo)

Sulla base di ciò andiamo a scrivere il nostro algoritmo:

```
InsertOR(L,K)
  if L=NIL then //Inserimento in coda
    L=newNode(K,L)
  else if [L->key > K] then //Inserimento in testa
    temp=L
    L=newNode(K, temp)
  else //Inserimento "centrale"
    L->next = InsertOR(L->next, K) //da esaminare
  return L
```

5.4 Cancellazione (iterativa)

Andiamo a scrivere un algoritmo di cancellazione simile a quello visto per l'inserimento, cioè andiamo a cercare il valore e ci posizioniamo sia "prima" che "dopo" il valore da cancellare.

```
Delete(L,K)
  temp=L //salviamo il puntatore al primo elemento
  P=NIL //creiamo una variabile di appoggio
  while [temp != NIL and temp->key != k] do
    P=temp //mettiamo P sul nodo "precedente" a quello da cancellare
    temp=temp->next //mettiamo temp sul nodo da cancellare

  if temp != NIL then
    if P != NIL then
      P->next=temp->next
    else
      temp=L
      L=L->next
    dealloca(temp)
  return L
```

5.5 Cancellazione (ricorsiva)

Ragioniamo per casi anche se in questo caso sono solo i due banali, cioè il valore è presente oppure no, nello specifico noi andiamo a considerare sempre la testa della lista che piano a piano a decrementarsi fino ad arrivare ad essere vuota

```
Delete(L,K)
  if L!=NIL then //La lista ha almeno un valore
    if L->key=k then //elemento in testa
      temp=L
      L=L->next
      dealloca(temp)
    else //elemento non in testa, "spostiamo" la testa in avanti
      L->next = delete(L->next, k)
  return L
```

6 Lezione 07 - 29/09/2023

6.1 Albero Binario

L'albero binario è una particolare struttura dati composto da nodi, ogni nodo ha tre campi:

- Valore
- Figlio Sinistro (puntatore)
- Figlio Destro (puntatore)

Ogni nodo può avere al più due figli, il primo nodo è chiamato **radice** e i nodi finali (senza figli) sono chiamati **foglie**.
Immagine
Andiamo a dare una definizione più corretta:

$$T \begin{cases} 1) T = NIL \\ 2) T = \text{ nodo + sottoalbero} \end{cases}$$

6.2 Altezza di un Albero

Definiamo l'altezza di un albero come la **quantità** di nodi che scorreremo per raggiungere il nodo desiderato. Nota bene: Ad ogni livello (di altezza) il numero di nodi dell'albero aumenta esponenzialmente con un ritmo di

$$2^i$$

. Questo vale solo per un albero bilanciato. Osserveremo la sua complessità tra poco.

6.3 Complessità Computazione

Un albero binario può essere degenere, cioè un albero i cui nodi hanno solo un figlio. Questo tipo di albero, come si vede in foto, creano in un certo senso una lista dinamica. Infatti questo tipo di albero condividerà la stessa complessità computazionale di una lista ordinata.
Analizziamo l'equazione che definisce quanti nodi ha ogni livello di un albero.
$n = \sum_{i=0}^{n} 2^i$ Questa sommatoria è risolvibile con una serie.

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

Diventerà nel nostro caso:

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

Il risultato che abbiamo ottenuto è il valore della tabellina del 2 2^{n+1}, a meno di un numero per il -1.
Per andare dunque a calcolare la complessità computazione per la creazione di ogni nodo per ogni grado d'altezza dell'albero è dunque utile andare a prendere il risultato di questa funzione qui sopra scritta e risolverlo.

$$n = 2^{n+1} - 1 \rightarrow \log_2 n = \log_2 2^{n+1} - 1 \rightarrow \rightarrow$$

Sapendo che $n = 2^{n+1} - 1$ allora potremmo sostituirlo all'interno della seconda parte dell'equazione, facendo in modo che l'equazione diventi

$$\log_2 n = \log_2 n$$

Andiamo a ragionare su quando possa valere $2^{n+1} - 1$. Andiamo a considerare in quale spazio esso può essere compreso (a seconda dell'altezza h): $2^h \leq 2^{n+1} - 1 \leq 2^{n+1}$
In questo caso potremmo dire che sicuramente esso sara piu grande di 2^h quindi andiamo a suddividere la disequazione per trovarci la h.

$$\log_2 2^h \leq \log_2 2^{h+1} - 1 \rightarrow h = \log_2 2^{h+1} - 1$$

Ma per quanto abbiamo visto prima:

$$(n = 2^{h+1} - 1) \text{e in questo modo avremo che} : \rightarrow n = \log_2 n$$

E questa sara la nostra complessita dell'algoritmo.

6.4 Visite

Esistono vari tipi di visita ognuna con i suoi pregi e difetti

6.4.1 PreOrder

L'algoritmo di visita pre-order è un particolare algoritmo usato per l'esplorazione in profondità dei nodi di un albero. L'esplorazione dell'albero parte dalla radice per poi scendere alle foglie, prima il sotto albero sinistro poi quello destro, che sono gli ultimi nodi ad essere visitati.

```
1 PreOrder(T):
2   T != NIL then
3     Visita(T->key) //funzione qualsiasi
4     PreOrder(T->sx)
5     Preorder(t->dx)
```

6.4.2 InOrder

L'algoritmo di visita in-order è un particolare algoritmo usato per l'esplorazione in profondità dei nodi di un albero binario. In questo tipo di visita, per ogni nodo, si esplora prima il sottoalbero sinistro poi si visita il nodo corrente ed infine si passa al sottoalbero destro. Più precisamente, l'algoritmo esplora i rami di ogni sottoalbero fino ad arrivare alla foglia più a sinistra dell'intera struttura, solo a questo punto si accede al nodo. Terminata la visita del nodo corrente si procede poi con l'esplorazione del sottoalbero a destra, visitando sempre i nodi a cavallo dell'esplorazione del sottoalbero sinistro e quello destro.

```
InOrder(T):
  T != NIL then
    InOrder(T->sx)
    Visita(T->key) //funzione qualsiasi
    InOrder(t->dx)
```

6.4.3 PostOrder

L'algoritmo esplora i rami dell'albero fino ad arrivare alle foglie prima di accedere ai singoli nodi, ad esempio si supponga di essere alla ricerca di alcuni oggetti molto pesanti e che per trovarli si debba esplorare un sentiero che comporta diverse diramazioni (albero).
Essendo gli oggetti pesanti non conviene raccoglierli subito e portarli con sé lungo il cammino ma conviene prima esplorare tutto il territorio quindi prelevarli quando si torna indietro.
L'algoritmo post-order visita un albero nello stesso modo, arrivando prima più in fondo possibile ad ogni diramazione ed accedendo agli elementi solamente al ritorno, il che corrisponde ad un'implementazione ricorsiva al fronte di risalita della ricorsione.

```
PostOrder(T):
  T != NIL then
    PostOrder(T->sx)
    PostOrder(t->dx)
    Visita(T->key) //funzione qualsiasi
```

6.5 Ricerca Albero Binario

Andiamo a definire la ricerca su l'albero binario, la miglior visita per eseguire una ricerca è la **preorder** poiché è la visita che subito controlla i nodi senza aspettare la visita dei sottoalberi.

```
Search(T,k):
  ret=T
  if T != NIL then
    if T->key != k then //se il valore non c'e' in testa
      ret=Search(T->sx, k) //allora cerchiamo nel sottoalbero sinistro
      if ret=NIL then //se non c'e' a sinistra
        ret=Search(T->dx,k) //cerchiamo a destra
  return ret
```

7 Lezione 08 - 03/10/2023

7.1 Nomi Utilizzati delle visite

Vi sono due tipo di visita che analizziamo per gli alberi:

- **DFS** o visite in **profondità** sono i tipi di visite che, in generale, tendono a visitare prima i figlio dei padri, si sviluppano "in verticale". Sono intrinsecamente ricorsivi.
- **BFS** o visite in **ampiezza** sono i tipi di visite che tendono ad andare per livelli, visitando i nodi che sono tra loro "fratelli" e "cugini".

7.2 Visita In Ampiezza Albero

Come già accennato, la visita in ampiezza di un albero prevede la visita dei nodi in modo "orizzontale". Questa visita di prim'occhio potrebbe sembrare piu complessa, poiche rispetto alle altre che abbiamo visto fino ad ora non è possibile effettuarla con un algoritmo ricorsivo/iterativo e basta. Infatti, per questa visita sarà necessaria una struttura dati astratta già studiata in precedenza: **la coda**. Iniziando la visita dalla radice, inseriremo i figli del nodo corrente all'interno della coda (se presenti), per poi proseguire la visita con un nodo estratto dalla coda, che diventa così il nodo corrente, e così via, fin quando la coda non sarà vuota.

```
1 BFS(T):
2 Q=NIL
3 Q=Accoda(Q,T)
4 while(Q!=NIL) DO
5   X=Testa(Q)
6   "Visita del nodo"
7   Q=Accoda(Q,X->sx)
8   Q=Accoda(Q,X->dx)
9   Q=Decode(Q) //Toglie della coda il primo
```

Ma quanto può essere complesso fare una cosa del genere? Le operazioni all'interno e prima del while sono a tempo costante, poichè sono operazioni di assegnazione. Il numero di volte in cui viene eseguito il while, però, dipende dalla quantità di nodi all'interno dell'albero, dunque è **lineare sul numero di nodi**. Per quanto riguarda la memoria occupata dalla coda, invece:

- Nel caso migliore l'albero è degenere (ogni nodo ha un solo figlio ad eccezione della foglia), nella coda sarà presente al più un nodo per volta, quindi è un costo di memoria **costante**.
- Nel caso migliore l'albero è bilanciato (ogni nodo ha 2 figli ad eccezione delle foglie), il momento in cui vi sarà più memoria utilizzata sarà quando tutte le foglie si troveranno in coda: circa n/2 nodi. Cioè nel caso più sfortunato si occuperà memoria **lineare sul numero di nodi**.

7.3 Albero Binario di Ricerca

E' un tipo di albero ordinato. Preso qualsiasi nodo, si ha che il suo dato è:

- Maggiore o uguale a tutti i dati contenuti nei nodi del suo sottoalbero sinistro
- Minore o uguale a tutti i dati contenuti nei nodi del suo sottoalbero destro

Il suo acronimo è **BST** (Binary Search Tree). L'algoritmo di ricerca per questo albero è uno dei piu efficienti.

7.3.1 Search Ricorsiva - BST

```
SearchBSTr(T,k)
ret=T
if T!= NIL then
  if T->key < k then
    ret = SearchBSTr(T->dx, k)
  else if T->key > k then
    ret = SearchBSTr(T-sx, k)
return ris
```

Questo algoritmo sicuramente ci permetterà di cercare in modo piu efficiente un dato, poichè, un po' come la ricerca binaria (ma non propriamente), ci permette di andare a "dimezzare" l'insieme di ricerca ad ogni confronto.

Esiste anche la versione iterativa di questo algoritmo:

```
SearchBSTi(T,k)
Tmp = T
while Tmp != NIl && Tmp-> key != k do
  if Tmp->key < k then
    Tmp=Tmp->dx
  else
    Tmp=Tmp->sx
return Tmp
```

Proprio come l'algoritmo ricorsivo andremo a dividere l'albero in due, ma ci avvarremo di un puntatore temporaneo per risalire nell'albero (questo puntatore viene memorizzato implicitamente nella versione ricosrsiva all'interno dello stack di attivazione).

7.3.2 Min - BST

La ricerca del minimo in un albero binario di ricerca è molto semplice. Seguendo la regola secondo cui "l'elemento piu piccolo si trova a sinistra", andremo a scendere a sinistra finchè sarà possibile, l'elemento che non ha figlio sinistro conterrà il dato più piccolo dell'albero.

```
MinR(T)
ret = T
if ret->sx != NIL then
  ret= MinR(ret->sx)
return ret
```

7.3.3 Algoritmo del Successore Ricorsivo - BST

Il successore di un numero è il primo numero piu grande dello stesso. Non è detto che un successore sia presente all'interno dell'albero (caso in cui tutti i valori nell'albero sono minori o uguali al valore di cui si cerca il successore) L'algoritmo diventa un po' piu complesso: vi sono le 3 casistiche:

- Caso **T->key = k**. L'elemento successore si troverà sicuramente nel sottoalbero destro, poichè cerchiamo un valore più grande. In particolare, è il piu piccolo elemento del sottoalbero destro, quindi **Min(T->dx)**
- Caso **T->key < k**. L'elemento successore si troverà sicuramente nel sottoalbero destro, poichè cerchiamo un valore più grande. Non sappiamo nient'altro però, quindi eseguiamo l'algoritmo sulla radice del sottoalbero destro: **Succ(T->dx)**.
- Caso **T->key > k**. L'elemento successore potrebbere trovarsi nel sottoalbero sinistro, poichè il valore corrente è più grande, quindi eseguiamo l'algoritmo sulla radice del sottoalberso sinistro **Succ(T->sx, k)**, se la ricerca non dovesse andare a buon fine, allora il vaore corrente T è il successore.

```
1  SuccR(T,k)
2  ret = NIL
3  if ret != NIL then
4    if ret->key = k then
5      ret = Min(ret->dx)
6    else if ret->key < k then
7      ret = SuccR(ret->dx,k)
8    else
9      ret = SuccR(T->sx,k)
10     if ret= NIL then
11       ret = T
12 return ret
```

7.3.4 Algoritmo del Successore Iterativo - BST

Per questo tipo di algoritmo, dobbiamo ragionare in modo diverso. E' necessario fare controlli che nella versione ricorsiva vengono omessi poichè queste informazioni vengono ricavate dal punto di ripresa dell'esecuzione di ogni chiamata. Bisogna fare particolare attenzione al caso in cui il valore di cui vogliamo il successore non ha figli destri, poichè è necessario risalire fino a trovare un padre "da sinistra" (la terza casistica). Nell'algoritmo il valore tmp viene usato per visitare l'albero. Quando si scende a sinistra, viene salvato prima il nodo corrente nella variabile ret, poichè se non si trova il successore in quel sottoalbero sinistro, allora è il nodo salvato in ret il successore.

```
1  SuccI(T,k)
2  Tmp = T
3  ret = NIL
4  while Tmp != NIL andd TMP->key != k then
5    if Tmp->key < k then
6      Tmp = Tmp->dx
```

```
    else
      ret = Tmp
      Tmp = Tmp->sx
  if Tmp != NIL && Tmp->dx != NIL then
    ret = Min(Tmp->dx)
  return ret
```

8 Lezione 9 - 05/10/2023

8.1 Successore Iterativo - BST

Per questo tipo di algoritmo, dobbiamo ragionare in modo diverso. In questo caso non iterativo non possiamo permetterci di omettere determinati controlli a posteriori. In particolare il controllo nel caso in cui il valore di cui vogliamo il successore non ha figli destri ed e una foglia. In questo caso particolare non abbiamo la possibilita di risalire a ritroso ricorsivamente ma dobbiamo tenere traccia ogni volta che il nodo scende a sinistra, segnandoci il puntatore di quest'ultimo.

```
SuccI(T,k)
  Tmp = T
  ret = NIL
  while Tmp != NIL andd TMP->key != k then
    if Tmp->key < k then
      Tmp = Tmp->dx
    else
      ret = Tmp
      Tmp = Tmp->sx
    if Tmp != NIL && Tmp->dx != NIL then
      ret = Min(Tmp->dx)
return ret
```

8.2 Predecessore Ricorsivo - BST

L'algoritmo del predecessore e simile al successore strutturalmente parlando ma invertendo segni e qualche operazione

```
PredR(T,k)
  ret = NIL
    if ret != NIL then
      if ret->key = k then
        ret = Max(ret->sx)
      else if ret->key < k then
        ret = PredR(ret->dx,k)
      else
        ret = PredR(T->sx,k)
        if ret = NIL then
          ret = T
  return ret
```

8.3 Insert Ricorsiva - BST

L'algoritmo dell'inserimento in un albero in un albero binario di ricerca puo vantare del fatto che e piu facile trovare il nodo nel quale si puo aggiungere il valore che abbiamo in input alla funzione. Ci bastera semplicemente scorrere a destra o a sinistra

il nostro puntatore per poi arrivare nel primo punto NIL favorevole e "returnare" a cascata i puntatori dei padri.

```
InsertR(T,k)
ret = T
  if T = NIL then
    ret = new_node(k)
  else if T->key < k then
    T->dx = InsertR(T->dx,k)
  else if T->key > k then
    T->sx = InsertR(T->sx,k)
return ret
```

8.4 New Node - Generico

La funzione new node va a creare un nuovo nodo dinamico all'interno dell'albero.

```
new_node(k)
  ret = alloca_nodo() //andiamo a restituire il puntatore a nuovo
     nodo allocato in memoria a ret
  ret->key = k
  ret->sx = NIL
  ret->dx = NIL
  return ret
```

8.5 Insert Iterativa - BST

Versione iterativa della insert prevede dei controlli in piu per quanto riguarda la ricerca e inserimento. In questo caso specifico abbiamo bisogno di un puntatore in piu che ci segue nello scorrimento, chiamato **P** e sta a indicare il Padre del nodo a cui stiamo scorrendo.

```
InsertI(T,k)
ret = T
P = NIL
Tmp = T
while Tmp != NIL && Tmp->key != k do
  P = Tmp
  if Tmp->key < k then
    Tmp = Tmp->dx
  else
    Tmp = Tmp->sx

if Tmp = NIL then
  x = new_node(k)
  if P = NIL then
    ret = x
```

```
16    else if P->key < k then
17      P->dx = x
18    else
19      P->sx = x
20  return ret
```

8.6 DeleteR - BST

La delete prevede la delete del nodo e la restituzione dell'albero con quel nodo mancante. A primo acchitto non sembra un'operazione cosi difficile ma dobbiamo come sempre andare a ragionare per casi.

- Caso albero vuoto. In questo caso dobbiamo semplicemente restituire T, il puntatore (vuoto) alla radice dell'alberoe
- Caso albero non vuoto. In questo caso la radice del sottoalbero ha un sottoalbero destro e sinistro.
  - Se T->key $<$ k then Delete(T->dx,k)
  - Se T->key $>$ k then Delete(T->sx,k)
  - Se T->key $=$ k, dobbiamo distinguere dei casi
    - Nel caso in cui il nodo non ha figli allora, si puo procedere all'eliminazione del nodo.
    - Nel caso in cui il nodo o ha figlio destro o figlio sinistro e basta, questo diventerà la nuova radice.
    - Nel caso in cui il nodo ha entrambi i figli, deleghiamo la distruzione del nodo ad un'altra funzione chiamata **StaccaMin(T->dx,T)**.

Dunque per il richiamo della funzione Delete ricorsiva abbiamo bisogno di due funzioni ausiliarie. La prima è utilizzata nel caso in cui vogliamo eliminare un nodo che ha entrambi i figli.

StaccaMin(T->sx, T)

```
1   StaccaMin(T,P)
2     ret = T
3     If T != NIL then
4       ret = StaccaMin(T->sx,T)
5       if ret = NIL then
6         if P != NIL then
7           if P->sx = T then
8             P->sx = T->dx
9           else
10            P->dx = T->dx
11    return ret
```

La seconda e chiamata per l'eliminazione del nodo andando a operare sull'intero sottoalbero dove quel nodo e radice.

DeleteRoot(T)

```
1   DeleteRoot(T)
2     ret = T
```

```
  if T!= NIL then
    Tmp = T
    if T->sx = Nil then
      ret = T->dx
    else if T->dx = NIL then
      ret = T->sx
    else
      Tmp = StaccaMin(T->dx,T)
      T->key = Tmp->key

    dealloca(tmp)
  return ret
```

In questo modo la chiamata a questa funziona delega il compito di capire in quale caso dei tre possibili si e dentro e di conseguenza deallocare in modo sicuro.
Infine andiamo a scrivere la funzione generale per la distruzione di un nodo nel modo seguente:

```
DeleteR(T,k)
  ret = T
  if T!= NIL then
    if T->key < k then
      T->dx = Delete(T->dx,k)
    else if T->key > k then
      T->sx = Delete(T->sx,k)
    else
      ret = DeleteRoot(T)
  return ret
```

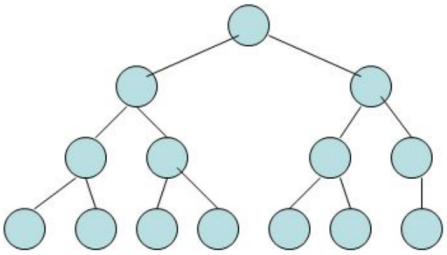
9 Lezione 10 - 06/10/2023

9.1 Alberi Perfettamente Bilanciati

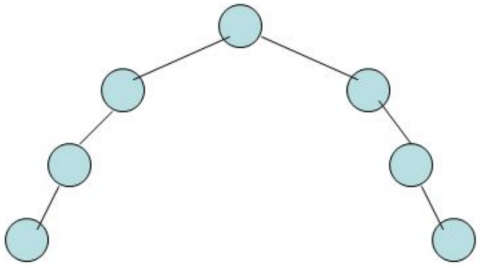
Gli alberi perfettamente bilanciati sono particolare tipi di albero binario in cui vale la seguente condizione:

$$||T- > sx| - |T- > dx|| \leq 1$$

La cardinalità (numeri di elementi) del sottoalbero sinistro deve differire di **al più 1** elemento del sottoalbero destro.
Non tutti gli alberi completi sono perfettamente bilanciati ma tutti gli alberi perfettamente bilanciati sono pieni.



(a) È un APB



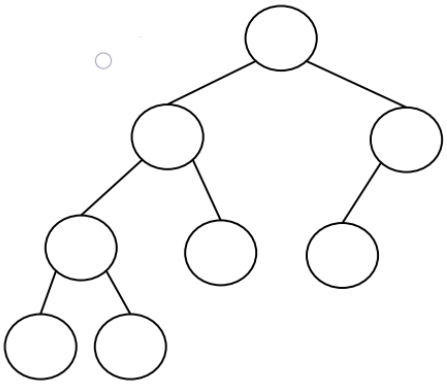
(b) NON è un APB

9.2 Alberi AVL

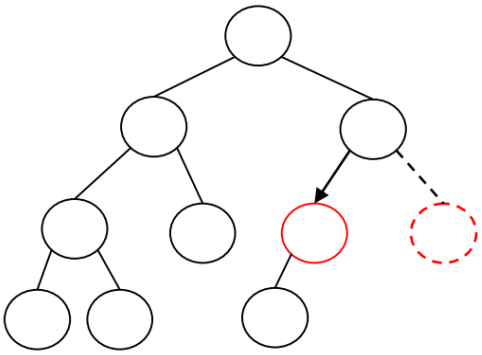
Gli alberi AVL sono particolare tipi di albero binario di ricerca in cui vale la seguente condizione:

$$|h(T- > sx) - h(|T- > dx)| \leq 1$$

Quindi l'altezza del sottoalbero sinistro di T e quella del sottoalbero destro di T differiscono al più di uno, ovviamente si applica ad ogni sottonodo.
A differenza degli alberi perfettamente bilanciati, non si pone un limite sulla cardinalità dell'insieme ma bensi sull'**altezza** dei sottoalberi.



(a) È un AVL

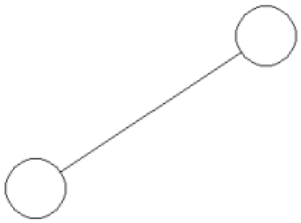


(b) NON è un AVL

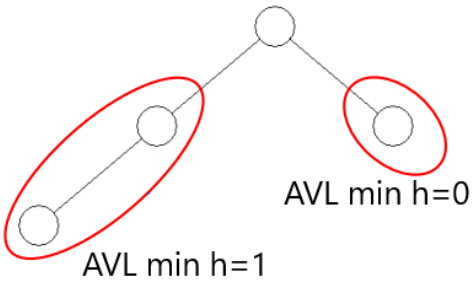
Un albero pieno è sia ABL che AVL

9.3 Alberi AVL Minimi

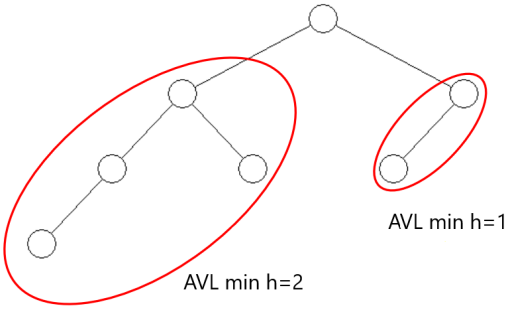
Fissato h, l'albero AVL minimo di altezza h è l'albero AVL di altezza h col minor numero di nodi possibile.
Per ogni altezza andiamo a mostrare un possibile albero:



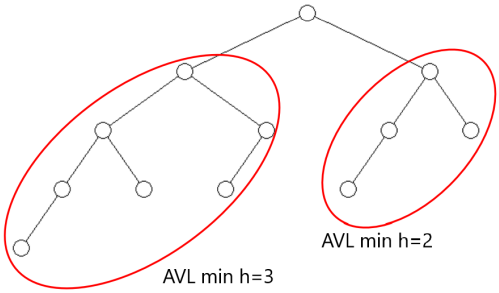
(a) AVL minimo di altezza 1



(b) AVL minimo di altezza 2



(c) AVL minimo di altezza 3



(d) AVL minimo di altezza 4

Possiamo notare un certo pattern che si ripeto, nello specifico dato un albero di altezza h il sottoalbero sinistro sarà $h-1$ e il ciaosottoalbero destro $h-2$.
Andiamo a generalizzare questa osservazione:

$$N(h) = \begin{cases} h+1 & \text{se } h = 0,1 \\ 1 + N(h-1) + N(h-2) & \text{e poniamo per assurse } h \geq 2 \end{cases}$$

DIM $h \geq 2$**:**
Prediamo un generico AVL T minimo, e poniamo per assurdo che il suo sottoalbero sinistro **è un AVL non minimo**, dunque esisterà un albero T' con sottoalbero sinistro che sarà **AVL minimo**. Quindi è un assurdo il fatto che esisterà un sottoalbero di T' di altezza $h-1$ con un numero minore di nodi rispetto al sottoalbero di T. In generale dunque se andiamo a dire che T è un Albero AVL minimo, non è possibile che esiste un T' con un numero di nodi **minore** di un albero AVL minore.
Data la formula precendete facciamo una considerazione:

Altezza	0	1	2	3	4	5	6	7	8
Numeri Nodi	1	2	4	7	12	20	33	54	88
Fibonacci	0	1	1	2	3	5	8	13	21

Possiamo notare come ci siamo un certo collegamento tra i numeri di nodi e la

sequenza di fibonacci, nello specifico notiamo come:

$$N(h) = F(h+3) - 1$$

Facciamo un ragionamento su come ricaverci l'altezza dato la formula chiusa di Fibonacci:

$$F(x) = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^x - \cancel{(\frac{1-\sqrt{5}}{2})^x}]$$

Andiamo a rimuovere la seconda parte poiché tende a zero.

$$F(x) = c \cdot k^x$$
$$N(h) = F(h+3) - 1$$
$$N(h) = c \cdot k^{h+3} - 1$$
$$\frac{N(h)-1}{c} = k^{h+3}$$
$$h = \log_k(\frac{N(h)-1}{c}) - 3$$

Abbiamo dimostrato che l'altezza è logaritmica sul numero di nodi.
Adesso andiamo a dimostrare che la formula dei nodi vale per ogni h
DIM:

- **Caso base:** $N(0) = F(0+3) - 1 = 1$
- **Caso induttivo:**

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

  Per ipotesi:

  $$N(h-1) = F(h+2) - 1$$
  $$N(h-2) = F(h+1) - 1$$

  Quindi:

  $$1 + (F(h+2) - 1) + (F(h+1) - 1) = F(h+3) - 1$$

10 Lezione 11 - 10/10/2023

10.1 Altezza

Per aiutarci nel bilanciamento andiamo a scrivere una funzione ausilaria che ci sarà molto utile:

```
Altezza(T)
    ret=-1
    if T != NIL then
        sx=Altezza(T->sx)
        dx=Altezza(T->dx)
        ret=1+max(sx,dx)
return ret
```

Questa funzione è **lineare** ma nel nostro caso voglio che sia logaritmo, quindi per ovviare a questo problema andiamo a inserire un valore h all'interno di ogni nodo.

```
Altezza(T)
    if T = NIL then
        ret=-1
    else
        ret=T->h
return ret
```

10.2 Inserimento AVL

Andiamo a vedere come scrivere un algoritmo per l'inserimento, essendo l'AVL un albero binario di ricerca sfruttiamo un ragionamento simile, ma con l'aggiunta del bilanciamento per rispettare la condizone degli AVL.

```
InsertAVL(T, k)
if T != NIL then
    if T-key < k then
        T->dx=InsertAVL(T->dx, k)
        T=Bilanciadx(T)
    else if T->key > k then
        T->sx=InsertAVL(T->sx, k)
        T=Bilanciasx(T)
else
    T=newnodo(k)
    T->h=0
return T
```

10.3 Bilaciamento

Dato che l'inserimento/cancellazione può sbilanciare un albero abbiamo bisogno di bilanciare l'albero in modo da far rispettare sempre la condizione, abbiamo diversi casi di bilanciamento andiamo ad esaminarli:

10.3.1 Bilanciamento Sinistro

```
Bilanciasx(T)
if T != NIL then
    if Altezza(T->sx) - Altezza(T->dx) = 2 then
        if Altezza(T->sx->sx) > Altezza(T->dx->dx) then
            T=Rotazionesx(T)
        else
            T->sx=Rotazionedx(T->sx)
            T=Rotazionesx(T)
    else
        T->h=1+max(Altezza(T->sx), Altezza(T->dx))
return T
```

11 Lezione 12 12/10/2023

11.1 Cancellazione albero AVL

Per la cancellazione di un elemento nell'albero AVL dobbiamo sempre fare delle considerazioni simili a quelle dell'aggiunta di un elemento nell'AVL.

Nel caso specifico pero c'e bisogno di considerare che la DeleteAVL del nodo. Nella diminuzione di un elemento K del sottoalbero, **il padre** dell'elemnto K puo essere attaccato facilmente a uno dei due sottoalberi di sinistra o destra. Nel farlo andiamo a diminuire di pesantezza uno dei due sottoalberi rendendo il successivo piu pesante. In quel caso la strategia che potremmo attuare e quella del ribilanciamento, ma condizionato dal fatto che ci troviamo in un AVL.

I tre algoritmi sono concettualmente gli stessi che usiamo per il bilanciamento di un BST.

11.1.1 DeleteAVL

Questa funzione e pressoche simile a quella vista negli Alberi binari di ricerca con l'aggiunta del bilanciamento.

```
DeleteAVL(T,k)
  if T != NIL then
    if T->key > k then
      T->sx = DeleteAVL(T->sx,k)
      T = BilanciaDx(T)
    else if T->key < k then
      T->dx = DeleteAVL(T->dx,k)
      T = BilanciaSx(T)
    else
      T = DeleteRootAVL(T)
  return T
```

11.1.2 DeleterootAVL

La delete Root, chiamata quando viene trovato il nodo da eliminare, non solo copiera il valore del piu piccolo elemento del sottoalbero destro di T, ma andra a bilanciare nuovamente a sinistra (perche piu pesante) il sottoalbero di T.

```
  DeleteRootAVL(T)
    if T != NIL then
      tmp = T
      if T->sx = NIL then
        T = T->dx
      else if T->dx = NIL then
        T = T->sx
      else
        tmp = stacca\_minAVL(T->dx,T)
```

```
        T->key = tmp->key
        T = BilanciaSx(T)
        dealloca(tmp)
    return T
```

11.1.3 Stacca_min

Per lo staccamin abbiamo bisogno di una variabile in piu in questo caso perche non ci e possibile andare a salvare l'elemento da cancellare dopo il bilanciamento, che si perderebbe nell'albero. In tal caso l'uso di una variabile e il controllo successivo se si trova nel nodo figlio di destra o sinistra ci verra in aiuto.

```
Stacca\_minAVL(T,P)
  if T != NIL then
    If T->sx != NIL then
      ret = Stacca\_minAVL(T->sx, T)
      newt = BilanciaDx(T)
    else
      ret = T
      newt = T->dx
    if T = P->sx then
      P->sx = newt
    else
      P->dx = newt
  return ret
```

Le operazioni di cancellazione andranno a effettuare una operazione di diminuizione dell'altezza di un albero e conseguentemente un'altra a cascata per il bilanciamento, che solitamente cambia le dimensioni dell'albero finale. Questo solitamente non ci dara problemi, ma solo in caso di alberi AVL ci dara problemi, poiche ci fara perdere le proprieta di quest'ultimo. In un esempio di AVL minimo la cancellazione di un nodo nel sottoalbero meno pesante andrebbe a comportare un ribilanciamento che va a peggiorare la differenza delle altezze tra alberi, portandola a 2, e qundi perdendo la proprieta di albero.

12 Lezione 13 - 13/10/2023

12.1 Alberi Red - Black

Gli alberi Red - Black, sono alberi binari di ricerca che associano dei colori ai loro nodi. La colorazione ovviamente andrà a braccetto con alcune proprieta (vincoli) che di seguito andremo a definire.

1) Ogni nodo deve essere rosso o nero.
2) **I nodi foglie possono essere solo neri (NIL)**, quindi i nodi rossi potranno essere soltanto all'interno.
3) Ogni nodo rosso ha **solo** figli neri.
4) Per ogni nodo X preso all'interno dell'albero, ogni percorso da X al nodo foglia contiene **lo stesso numero** di nodi neri.

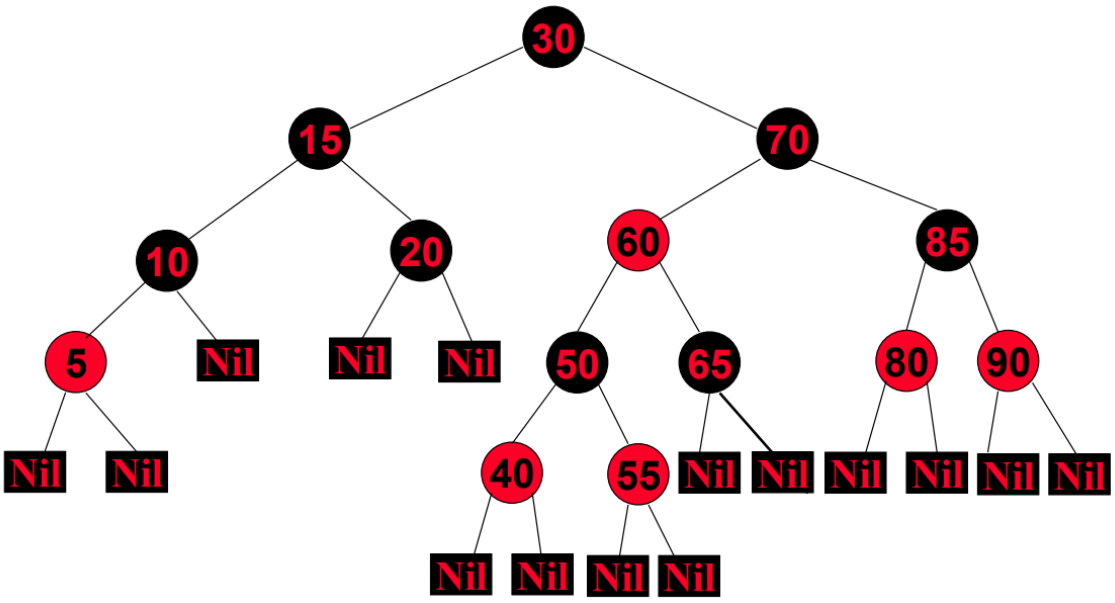


Figure 4: Questo è un albero RB perché soddisfa tutti e 4 i vincoli

Non tutti gli alberi possono essere colorabili

Per vedere se un albero è colorabile ci sono delle considerazioni da fare:

- Colorare subito le foglie e la radice di nero.
- Osservare se esiste un sottoalbero è visibilmente piu pesante di un altro. In tal caso l'albero è quasi sicuramente non colorabile. Teoricamente se la differenza di altezza di alberi è maggiore di due allora probabilmente non è colorabile.

12.1.1 Altezza Nera di un albero R-B

L'altezza nera, di un albero R-B, è il numero di nodi neri che, preso un nodo X, si contano da X fino alle foglie escludendo X.
L'altezza nera è sicuramente minore dell'altezza dell'albero e al massimo uguale.
Dimostriamo dunque che l'altezza è sicuramente:

$$h \leq 2^{bh(x)} - 1$$

Preso un nodo all'interno di un albero il numero di nodi interni non può essere più piccolo di un albero completamente nero.

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

Dimostriamo per induzione

- Base Induttiva: Albero di altezza zero, quindi il numero di nodi interni di un albero di $h = 0$ è **zero**, andiamo a svolgere l'equazione con la base induttiva:

$$0 \geq 2^{bh(x)} - 1 \Rightarrow 2^0 - 1 = 0 \Rightarrow \text{ VERO}$$

- Caso Induttivo $h > 0$: L'albero contiene almeno un nodo interno. Andiamo a scomporre il nostro albero come sottoalbero sx del figlio sinistro e sottoalbero dx del figlio destro.

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

  Noi sappiamo che

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

  In questo caso, ragionando analiticamente possiamo dire che l'altezza nera di X, il nostro nodo padre del sottoalbero dipende dal fatto che y, il suo sottoalbero sinistro, sia nero o rosso.
  - Nel caso in cui il nodo sia rosso, allora l'altezza nera di x e y sono uguali.
  - Nel caso in cui il nodo sia nero, allora l'altezza nera di x e uguale a quella di y + 1.

  Esplicitiamo $bh(y)$ dalle due equazioni perche ci interessa esplicitare tutto per $bh(x)$
  In questo caso vedremo che $bh(y) > bh(y) - 1$ poiche o e uguale, o e sicuramente maggiore di $bh(x) - 1$.
  Questo vale anche per z, dunque avremo:

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

  Questo vuol dire che scendendo di altezza, andro a diminuire al massimo di uno l'altezza del sottoalbero. Grazie a queste equazioni possiamo ritornare a ritroso alla tesi.
  Usiamo questo ragionamento matematico: Se io so che $n \geq m$, allora avro anche che $2^n \geq 2^m$ poiche l'esponenziale e crescente e non andiamo a modificare il risultato comunque finale.
  Applichiamo dunque la stessa proprieta alle stesse equazioni scritte sopra. In tal caso avremo

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

  sottraiamo una stessa quantita a entrambi i membri

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

Dunque entrambi vedremo che la somma tra $2^{bh(y)} e 2^{bh(z)}$ sono maggiori o uguali di $2^{bh(x)-1}$.
Il numero di nodi interni di X e dato da $1 + ni(y) + ni(z)$. Sostituendo abbiamo che $1 + 2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 \rightarrow 2 * 2^{bh(x)-1}$. Il "-1" puo essere semplificato portando dentro il 2 moltiplicato avanti all'espressione. In tal modo avremmo che indipendentemente dall'altezza che io ho in entrata, il numero di nodi interni di quel sottoalbero e almeno uguale a $2^{bh(x)} - 1$, quindi la tesi inziale e dimostrata.

12.1.2 Considerazioni sull'altezza nera di un albero

Se sappiamo che il numero di nodi n e maggiore o uguale a $2^b h - 1$, possiamo intuitivamente e approssimativamente andare a trovare l'altezza nera dell'albero.
Nel caso in cui avesse tutti i nodi neri allora l'altezza nera e $\leq h$, mentre se ha alternati rossi e neri, abbiamo il limite minimo dell'altezza nera, cioe $\frac{h}{2}$.
Dunque l'altezza nera e compresa tra : $\frac{h}{2} \leq bh \leq h$.
Usando la matematica e le nozioni della dimostrazione precedente...

$$bh \geq \frac{h}{2} \rightarrow 2^{bh} - 1 \geq 2^{\frac{h}{2}-1}$$

Se sappiamo che n e $\geq 2^{bh} - 1$ allora,

$$n \geq 2^{\frac{h}{2}-1}$$

$$n + 1 \geq 2^{\frac{h}{2}}$$

$$\log_2 h + 1 \geq \frac{h}{2}$$

$$h \geq 2 \log_2 n + 1$$

12.1.3 Inserimento Albero R-B

Gli algoritmi di inserimento e bilanciamento usati fino ad ora non andranno più bene per questo tipo di struttura. Nonostante abbiamo più libertà da un certo punto di vista, dobbiamo considerare che la proprieta 4 degli alberi Red-Black ci impedisce di fare degli inserimenti nella struttura dati in modo efficiente.

In particolare nell'inserimento di un valore in un nodo NIL, l'algoritmo deve occuparsi di creare il nodo, colorarlo e di creare e colorare a sua volta i figli NIL (di nero ovviamente).

Successivamente la colorazione del nodo k non è immediata e semplice, va considerato il colore del padre e non va rotto il vincolo dello stesso numero di nodi neri per ogni percorso dei sottoalberi.

Abbiamo due possibilità di colore all'inserimento. Solitamente per non creare problemi con il padre del sottoalbero a cui dobbiamo inserire, si inserisce nero. Anche in quel caso non e detto che l'inserimento del nero non abbia creato problemi per la proprieta 4 dei R-B.
Nei Red - Black la violazione di una di questi due algoritmi puo essere scoperta solo ricorsivamente.

```
InsertRB(T,k)
  if T != NIL then
    if T->key < k then
      T->dx = InsertRB(T->dx, k)
      T = BilanciaRBdx(T)
    else if T->key > k then
      T->sx = InsertRB(T->sx, k)
      T = BilanciaRBsx(T)
    else
      T = new_nodeRB(k,r) //Creazione nodo rosso (r) e figli a NIL
  return T
```

Questa funzione verra supportata dalle funzioni di bilanciamento che sono specificatamente scritte apposta per la R-B. In tal caso abbiamo 3 casistiche generali di problemi. Nel caso in cui il nodo che andiamoa inserire sia rosso...

- Caso 1) Il padre rosso e il fratello rosso. Sia a destra che a sinistra del padre rosso.
- Caso 2) Inserimento a destra del sottoalbero il cui padre e rosso e il fratello nero.
- Caso 3) Inserimento a sinistra del sottoalbero il cui padre e rosso e il fratello e nero.

La risoluzione del Caso 1, si scambia il nodo padre rosso con il nonno nero, in tal caso abbiamo che i figli diventeranno perforza di cose neri. In questo modo non andiamo a violare la proprieta 4, poiche i percorsi sia a destra che a sinistra avranno lo stesso numero di nodi neri, ma andiamo soltanto ad aumentare l'altezza nera.
Il Caso 2 si risolve ruotando in modo tale da arrivare al caso 3.
Per il caso 3 ci conviene ruotare l'albero a destra e sostituire il nodo radice (precendetemente nero) con un nodo rosso (visto che era figlio di nero). In tal modo abbiamo la radice nera, i figli rossi e i sottoalberi non cambiano.
Aggiustare assolutamente

13 Lezione 14 17/10/2023

13.1 Bilanciamento All'inserimento in Albero R-B

```
BilanciaInsSx(T)
  if !NIL(T->sx) and (!NIL(T->sx->sx) or !NIL(T->sx->dx)) then
    v = ViolazioneSxInsRB(T->sx,T->dx)
    Case V of :
      1: T = Caso1(T)
      2: T = Caso2(T)
      3: T= Caso3(T)
  return T
```

La violazione verifica se esiste una violazione di regola numero 4 (degli R-B), mandando il sottoalbero sinistro e destro rispettivamente all'interno di una funzione controllo che restituira il tipo di caso di violazione che stiamo riscontrando in quel sottoalbero.

13.2 Controllo Violazione a Sinstra Inserimento R-B

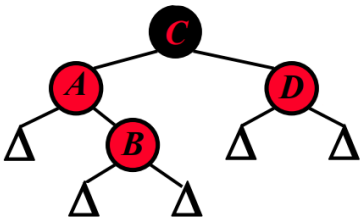
```
ViolazioneSxInsRB(s,d)
  v=0
  if s->col = R then
    if d->col = R then
      if s->sx->col = R or s->dx->col = R then
        v=1
    else if s->dx->col = R then
      v=2
    else if s->sx->col = R then
      v=3
  return v
```

13.3 Risoluzione Violazione Inserimento a sinistra

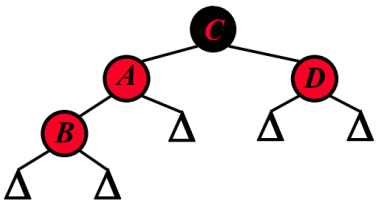
Quando andiamo ad inserire un nodo in un albero RB è facile violare le propietà, quindi andiamo a indentificare i 3 casi in cui avvengono e come andare a risolverli. Generalmente il problema principale è non rompere la proprieta 4. Per far si che non si rompa la proprieta diamo al nodo appena creato il colore rosso, che potrebbe compromettere i sottoalberi destro e sinistro.

13.4 Caso 1

Possiamo indentificare il **Caso 1**, osservando solamente se il figlio destro è un nodo rosso. La violazione puo avvenire sia sul sottoalbero sinistro del nodo sinistro della radice, sia nel destro.



(a) Albero RB Caso 1 (n1)

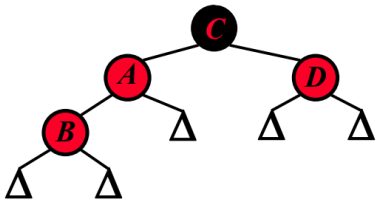


(b) Albero RB Caso 1 (n2)

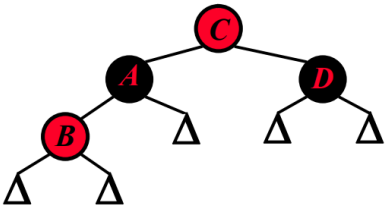
Per corregere questa violazione andiamo a colorare la "radice" di rossa(anche se per convenziene dovrebbe essere nera), e poi di conseguenza andando a colorare i figli sinistri e destri di nero.

```
Caso1(T)
  T->dx->col=black;
  T->sx->col=black;
  T->col=red;
  return T;
```

In questo tipo di soluzione eliminiamo la violazione sui sottofigli, facciamo in modo che l'altezza nera sia rispettata in tutti i nodi, ma andiamo a "spostare" il problema verso l'alto, che poi verra corretto a cascata, fino eventualmente al nodo radice.



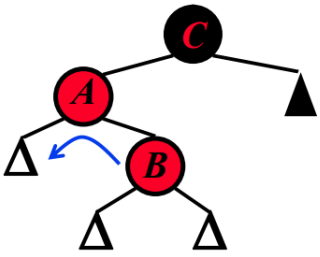
(a) Violazione Caso 1



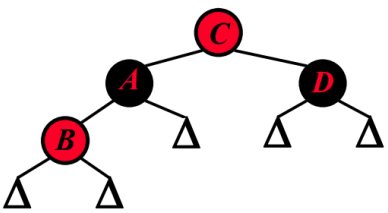
(b) Correzione Violazione

13.5 Caso 2

Il caso 2 (vedi foto) comporta il fatto che il **figlio destro** del sottoalbero su cui stiamo operando e **nero**. La strategia di risoluzione per il caso 2 prevede di far "ruotare" il problema a sinistra (quindi far salire di altezza la violazione) e far si che diventi caso 3, in questo modo andiam semplicemente a richiamare la strategia di risoluzione del caso 3.



(a) Violazione Caso 1



(b) Correzione Violazione

```
Caso2(T)
  T->sx = rotate(T->sx)
  t = Caso3(T)
  return T
```

13.6 Caso 3

Il caso 3, come nel caso 2, si verifica nel momento in cui, il figlio destro del sottoalbero su cui stiamo operando e nero. La differenza col caso 2 e che il figli del sottoalbero sinistro non sono entrambi rossi, ma il rosso si trova soltanto nel sottoalbero destro (come si nota in foto). La strategia di risoluzione e dunque andare a ruotare sulla sinistra l'albero e scambiare i colori della radice e del figlo destro.

```
Caso3(T)
  T = rotazioneSx(T)
  T->col = N
  T->dx->col = R
  return T
```

Ovviamente queste stesse violazioni sono possibili sul sottoalbero destro. Analogamente le soluzioni sono le stesse con opportune inversioni di nodi su cui devono essere fatte le operazioni.

13.7 Delete Albero R-B

Per la distruzione del nodo di un albero R-B ci vogliono molte piu accortezze della creazione poiche:

- La distruzione di un nodo nero comporterebbe una failure nella proprieta 4 dell'altezza nera.
- Un nodo da disruggere potrebbe avere figli , quindi bisogna ragionare su come disporli e colorarli.

Gli unici nodi da distruggere sono i nodi interni.
L'elemento da sostituire cambiera colore in base a quale colore sia stato cancellato.

- La distruzione di un nodo **rosso** comportera che il nodo acquistera il colore di nero.
- La distruzione di un nodo **nero** comportera che il nodo acquistera il colore di **doppio nero**, per non perdere la proprieta 4.

```
DeleteRB(T,k)
if !NIL(T) then
  if T->key > k then
    T->sx = DeleteRB(T->sx,k)
    T = BilanciaDelsxRB(T)
  else if T->key < k then
    T->dx = DeleteRB(T->dx,k)
    T = BilanciaDeldxRB(T)
  else
```

```
    T = DeleteRootRB(T)
return T
```

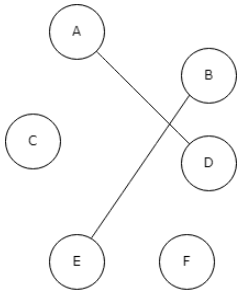
Parallelamente alla delete degli AVL la funzione di DeleteRB hanno la stessa struttura di funzioni.

```
DeleteRootRB(T)
  if !NIL(T) then
    TMP = T
    if NIL(T->sx) then
      T = T->dx
      if TMP->col = N then
        PropagateBlack(T)
    else if NIL(T->dx) then
      T = T->sx
      if TMP->col = N then
        PropagateBlack(T)
    else
      TMP = StaccaMinRB(T->dx, T)
      T = BilanciaDeldxRB(T)
    dealloca(TMP)
  return T
```

14 Lezione 17 24/10/2023

14.1 Grafi

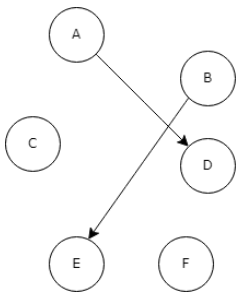
I grafi sono una coppia di insiemi definita come:(V,E) (chiamato oggetto), dove V e un insieme finito di elementi che compongono il grafo chiamati **vertici**. L'insieme E rappresenta l'insieme degli **archi**.
L'insieme E e costituito da una coppia composta cosi definita: $E \subseteq VxV$, quindi una coppia formata da due vertici dell'insieme **V**.
L'insieme E e in relazione simmetrica.
I grafi non posseggono foglie, ma esistono grafi con archi uscenti soltanto, verso nodi che non hanno altri collegamenti.
Un grafo si dice **completo** quando il suo numero di archi e uguale a n.



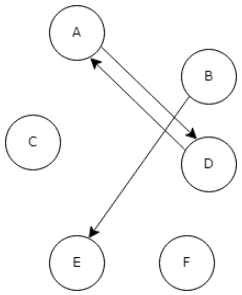
$$V = \{A, B, C, D, E, F\}$$
$$E = \{(A, D), (B, E)\}$$

14.2 Grafo Orientato

Un grafo orientato e un grafo che ha gli archi direzionati, cioe che vanno da una direzione all'altra, solitamente nella rappresentazione grafica a coppie, la direzione va dal primo elemento verso il secondo.
(I secondi elementi nella coppia sono i cosiddetti "Carrier della relazione").



Per "tornare" al caso in cui il grafo non e orientato, bisogna avere aggiungere degli archi orientati nel verso opposto a quelli che gia esistono:



14.3 Alberi e Grafi

Gli alberi sono particolari tipi di grafi dove la relazione che vige tra i nodi e quella di parentela (padre figli). In esso i tragitti non sono infiniti come possono essere in un grafo. Un grafo non puo essere ricostruito come un albero nel caso in cui un nodo si perde, poiche non sappiamo quanti e quali archi abbia perso a differenza di un nodo di un albero che puo avere un solo padre e uno o 2 figli.

14.4 Grafi Pesati

I grafi pesati sono un'applicazione dei grafi ordinati e non, dove ad ogni arco viene associato un numero (peso).

14.5 Grado di un Vertice

Il grado di un vertice puo dividersi in

- Grado entrante, cioe il numero di archi che entrano nell'arco
- Grado uscente, cioe il numero di archi che escono dall'arco

P.S. un cappio e un arco che esce e entra nello stesso nodo.
Ogni vertice puo avere come massimo numero di archi (entranti e uscenti), il numero di vertici del grafo.

$$n = |V|$$

14.6 Percorsi

Sia $v, w \in V$ e definito $\pi =$ il percorso da $v$in$w$ nel grafo G. Per percorso si intende la sequenza di vertici che si impegnano per arrivare al vertice stabilito.

$$\pi = v_0, v_1, v_2, \ldots, v_k$$

tale che

- $v_0 = u$
- $v_k = v$
- $\forall 0 \leq i \leq k-1 = (v_i, v_{i+1})$

La coppia di valori tali che il secondo e il successivo del primo.

14.7 Raggiungibilita di un nodo

Un nodo v e detto raggiungibile da u un un grafico $G \iff \exists \pi | \pi$e un percorso da $u$in v.

Quindi solo se esiste un percorso (insieme di nodi raggiungibili attraverso degli archi) da u in v.
La raggiungibilita puo essere scritta come funzione Reach $\in VxV$. La funzione Reach possiede delle proprieta:

- Riflessiva v e raggiungibile da v poiche (v, v) e una coppia $\in E$, ed e chiamato percorso senza archi.
- Transitiva Se $(u, z) \in Reach$ e $(z, v) \in Reach$ allora avremo che sicuramente $(u, v) \in Reach$. Se esiste un percorso che da u va in z e da z in v, allora esistera un percorso che raggiunge (u, z), concatenando i percorsi dei due.

14.8 Percorsi Ciclici

Un grafo orientato in cui non esistono cicli semplici e detto **Grafo Ciclico**, dove per ciclo semplice si intende una "ripetizione di un vertice in una sequenza".

14.9 Sottografo

Dato $G = (V, E)$, allora $G' = (V', E')$ e detto sottografo di G. La condizione principale per far si che G' sia sottografo e che $V' \in V$. In questo caso il numero degli archi e : $E' \in E \cap (V'xV')$. Nel caso in cui $E' = E$, allora ci troviamo in un **sottografo indotto**.

15 Lezione 18 26/10/2023

15.1 Complessità sui grafi

I grafi hanno una ciclicità variabile, ovviamente dal numero di archi che ognuno di essi avrà. In un grafo completo ad esempio c'è una probabilità certa di trovare delle ripetizioni (ciclicità) in un percorso.
Data una famiglia di grafi $\{G_i\}_{i<N}$ (grafo finito) e dove i è il numero di vertici del grafo. Ciclicamente avremo che i grafi completi con "i" numeri di vertici avranno: Il grafo con 2 vertici:
Il grafo con 3 vertici:
Il grafo con 4 vertici:
Generalmente il numero dei percorsi che si hanno per un grafo è uguale alla somma del numero di percorsi che si hanno nei sottografi piu piccoli insieme al suo, che avrà i percorsi.

$$P_{(i)} = \sum_{k=1}^{i-1} P(k)$$

ovviamente questo vale se $i > 2$.
Se andiamo a esprimere questa somma avremo che:

$$P(i) = \begin{cases} 1 \text{ se } i \leq 2 \\ 2P(i-1) \text{ se } i > 2 \end{cases}$$

Questa equazione è chiamata **Equazione di ricorrenza**, equazione che si risolve con algoritmo ricorsivo per verificare il numero di percorsi in un grafo completo.
Questa stessa equazione è utilizzabile per un albero binario di ricerca completo poichè, ogni nodo ha due figli e quindi due sottoalberi dove al massimo possono essere di 2^{i-1} nodi. Anche in questo caso dunque avremo che:

$$P(i) = \begin{cases} 1 \text{ se } i \leq 2 \\ 2nf(i-1) \text{ se } i > 2 \end{cases}$$

Quindi il numero di percorsi in un grafo è uguale a 2^{i-1}.
Questo risultato ci rende impossibile l'esplorazione totale del grafo (come anche in un caso analogo di un albero completo), poiché è esponenziale sul numero di nodi.

15.2 Come Rappresentiamo un Grafo?

Come possiamo andare a "disegnare un grafo" e quali informazioni minime abbiamo bisogno per farlo? Un modo abbastanza semplice e intuitivo è quello di andarlo a descrivere come lo abbiamo definito in precedenza quindi come $E \in VxV$. In questo modo abbiamo coppie di nodi a cui abbiamo associato i propri archi.
Un altro modo di rappresentare un grafo è attraverso l'uso della **funzione caratteristica**, cioè quella funzione "booleana" che dato in input un dato ci restituirà 0 o 1 a seconda della presenza o meno di quel dato nella struttura dati su cui vogliamo cercare.

15.2.1 Matrice di bit

Possiamo dunque andare a creare una specie di scacchiera (matrice) dove dal risultato della casella potremmo capire se quel nodo è collegato o meno con un altro.
La matrice è chiamata **Matrice di Bit**, che ci permette di avere un **vantaggio** nella ricerca di un eventuale arco (a tempo costante).Lo **svantaggio** invece è quello che dovremmo creare una matrice con nxn con l'eventuale utilizzo di memoria indesiderata in più. Inoltre non sappiamo dapprima quandi archi potremmo trovare nella matrice, non abbiamo un'informazione a tempo costante ma a tempo **lineare** in questo caso. E nella costruzione non c'è da dimenticarsi che è quadratica sul numero intero di bit.
Alternativamente un altro modo per visualizzare il nostro grafo è quello di sfruttare la proprietà di adiacenza dei grafi, e quindi creare una struttura dati che ci tiene traccia di tutte le adiacenze che ogni vertice ha.
In questo caso ci viene in aiuto un array unidimensionale dove ogni cella punta a ogni vertice del grafo e da lì poi tutte le adiacenze saranno messe all'interno di ogni struttura.

15.2.2 Liste di adiacenza

Questa struttura dati è chiamata Lista di adiacenza. Lo **svantaggio** rispetto alla matrice è che per trovare un arco bisogna scorrere linearmente sul numero dei vertici della lista. Il **vantaggio** è che lo spazio impiegato per la lista intera è logaritmico sul numero di vertici $\log_2 n$.
Solitamente la soluzione scelta è la lista di adiacenza per convenzione e comodità. Le liste hanno una migliore esplorazione dei grafi, se non sono densi. In generale però è una ottima scelta di compattezza.

16 Lezione 19 - 27/10/2023

16.1 Visite Grafi

Come per gli alberi anche sui grafi esistono vari tipi di visite, avendo maggiore libertà di "movimento" dato che ogni vertice può essere raggiunto da più percorsi dobbiamo a definire alcuni concetti importati.

Vertice Sorgente Dato che in un grafo non esiste un punto di partenza come la radice negli alberi assegniamo questo ruolo ad un considetto **vertice sorgente** s che avrà il ruolo di farci esplorare tutti gli altri nodi del grafo (frontiera).

Colorazione Come abbiamo detto ogni nodo può essere raggiunto da diversi percorsi per evitare di visitare più volte lo stesso vertice assoceremo ad ognuno di essi un colore che ci permetterà di capire lo "stato" di esplorazione:

- Bianco: Vertice non ancora scoperto (caso base)
- Grigio: Vertice scoperto ma non visitato
- Nero: Vertice scoperto e visitato (caso finale)

16.1.1 BFS

Rispetto agli alberi la visità in ampiezza non può sfruttare il concetto di "livello" poiché in un grafo non abbiamo questo tipo di concetto ma sfrutteremo la **distanza**, quindi verranno esplorati prima i vertici più vicini al **vertice sorgente** e poi quelli più distanti.

```
1  Init(G) //G:grafo
2      for each v in V do //per ogni vertice del grafo
3          color[v]=b //coloriamo di bianco il vertice
```

La funziona **init** ha lo scopo di colorare tutti i nodi di bianco, il costo di questa funzione è $|V|$ cioè il numeri dei vertici del grafo.

```
1  BFS(G, s) //G:grafo, s: vertice sorgente
2      Init(G)
3      Q={s} //creiamo una coda con dentro s
4      color[s]=g //coloriamo di grigio
5      while Q != NIL do
6          x=Testa(Q) //prendiamo l'elemento in testa
7          for each v in ADJ(x) do //scorriamo gli adiacenti
8              if color[v] = b then //se e' bianco
9                  Q=Accoda(Q,v) //mettiamo il nodo bianco nella frontiera
10                 color[v]=g //mettiamo a grigio
11         //operazione su x
12         Q=Decoda(Q) //x ha finito gli adiacenti e quindi lo togliamo
13         color[x]=n //lo mettiamo a nero (stato finale)
```

La funzione **BFS** ha un costo $|V| + (|E| + |V|) = |V| + |E| = |G|$

16.1.2 Variante BFS Tempo

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

17 Lezione 21 - 02/11/2023

17.1 Visita in Profondità - DFS

La Depth-First Search (DFS) è un algoritmo di esplorazione dei grafi il cui funzionamento consiste nel visitare prima i nodo più lontani dal nodo sorgente e poi tornando progressivamente verso il punto di partenza.

Rispetto alla BFS vista in precedenza, la DFS viene impiegata per scopi diversi. Mentre la BFS è utilizzata per scoprire percorsi minimi, la DFS può essere usato per verifica la ciclicità di un Grafo. Come per la BFS useremo di "strutture ausiliarie":

- Col[ ]: $V \to \{b, g, n\}$ (array dei colori)
- Pred[ ]: $V \to V$ (array dei predec., cioè quale vertice è stato scoperto da chi)
- d[ ]: $V \Rightarrow \mathbb{N}[\{1, 2, ..., 2n\}]$ (il tempo in cui il vertice è stato scoperto)
- f[ ]: $V \Rightarrow \mathbb{N}[\{1, 2, ..., 2n\}]$ (il tempo in cui il vertice è stato visitato)

Dato che per la DFS vogliamo visitare tutti i vertici di un grafo, dato che non sempre un vertice sorgente ci permette di farlo, andiamo a definire due funzioni:

```
Init(G) //G:grafo
    for each v in V do //ogni vertice del Grafo
        col[v]=b
        prev[v]=NIL
    tempo=1 //variabile globale
```

La funzione DFS ci permette di far ripartire la visita da un nuovo vertice sorgente (bianco) in modo da poter esplorare ogni parte del grafo.

```
DFS(G) //G:grafo
    Init(G)
    for each v in V do //ogni vertice del Grafo
        if col[v]=b then
            DFS_Visit(G,v)
```

Ogni volta che troviamo un nuovo nodo (bianco) andiamo a richiamare ricorsivamente la visita in modo da scendere in Profondità nel grafo e poi successivamente "risalire".

```
DFS_Visit(G, s) //G:grafo, s: vertice sorgente
    col[s]=g //coloriamo di grigio
    d[s]=tempo++ //scoperto a tempo corrente (+1)
    for each v in ADJ[s] do
        if col[v]=b then
            pred[v]=s
            DFS_Visit(G,v) //visita ricorsiva
    f[s]=tempo++ //visitato a tempo corrente (+1)
    col[s]=n
```

Costo Come visto per la BFS possiamo anche qua ogni nodo viene visatato una solo volta quindi il costo sarà sempre |V|+|E|=|G|

18 Lezione 22 03/11/2023

18.1 Dimostrazione Teorema delle parentesi

La dimostrazione della condizione di sufficienza ($\Rightarrow$) del teorema delle parentesi.
Per induzione sulla lunghezza di π (che rappresenta la lunghezza del percorso tra u e v):

- **Base d'induzione**: $|\pi| = 1$. IN questo caso esiste un arco che va direttamente da v a u. Essendo che u e figlio di v e v = prev[u], allora sicuramente avremo che $d[v] < d[u]$. Ora ci rimane da capire quando vengono chiuse entrambe. Per coerenza con il nostro algoritmo di DFSvisit, sappiamo che sicuramente verranno chiusi prima i discendenti e poi gli antenati a ritroso, in questo caso ,dunque, verra chiuso prima u che v. In questo modo abbiamo dimostrato la nostra tesi, cioe che $d[v] < d[u] < f[u] < f[v]$.
- **Passo d'induzione** : Caso $|\pi| > 1$: In questo caso abbiamo che $\exists z \in V : v \Rightarrow z \Rightarrow u$. In questo caso allora abbiamo un sotto-percorso π' costituito da un solo elemento, come nella base induttiva. In tal senso abbiamo dunque che $d[v] < d[z] < d[z] < d[v]$. Possiamo estendere questo ragionamento agli n nodi che dividono v da u e creare quindi una catena di $dedif$, ma quello che vedremo semplificando tutti i nodi che non interessano alla dimostrazione che $d[v] < d[u] < f[u] < f[v]$.

18.2 Teorema del percorso bianco - Definizione

Dato un grafo $G = (V, E)$ ed eseguita una DFS su G per ogni coppia di vertici $v, u \in V$ e con $v \neq u$:
u e detto discendente di v nella FDF $\iff$ al tempo $d[v]$ esiste un percorso (da u a v) fatto da solo vertici bianchi.

18.2.1 Dimostrazione Teorema - Condizione Necessaria $\Rightarrow$

La foresta ha gia almeno un percorso con vertici bianchi, quindi quel percorso puo stare nella foresta. Se prendiamo un arbitrario vertice $z \in V$ del percorso, vedremo che z e discendente di v solo se $d[v] < d[z] < f[z] < f[v]$. E questo e verificato dal teorema precedente. Se la condizione e rispettata, tutti i vertici fino a z erano bianchi (a tempo v), e quindi rispetta una parte della tesi. Lo stesso ragionamento si puo applicare a tempo z per tutti i vertici fino a u.

18.2.2 Dimostrazione Teorema - Condizione Sufficiente $\Leftarrow$

Abbiamo un percorso fatto da tutti i vertici bianchi, dimostreremo per assurdo che non esista discendenza di u su v.
Ipotizziamo dunque che esiste un vertice che non e discendente da v, nel percorso di tutti bianchi fino a u e lo chiameremo t.
L'ultimo vertice discendete da v lo chiameremo z e si trovera prima di t, quindi $d[v] < d[z] < f[z] < f[v]$. Ci chiediamo ora quando verra scoperto t, non essendo discendente di v o di z.

- Non potrebbe essere scoperto prima di z poiche a tempo di $d[v]$, sono tutti bianchi per la tesi del teorema.
- Non potrebbe essere scoperto al ritorno (durante la chiusura di v), perche dovrebbe essere gia stato scoperto in chiusura, senno andrebbe in contrasto con il teorema delle parentesi.

Il punto piu probabile di scoperta e tra $d[z]ef[z]$, e cio andrebbe in tesi con cio che abbiamo detto poiche diventerebbe $d[v] < d[z] < d[t] < d[u] < f[u] < f[t] < f[z] < f[v]$. Andando a semplificare i nodi che non ci interessano, abbiamo che il nodo t rispetta il teorema delle parentesi e quindi il concetto di discendenza.

18.3 I problemi dell'algoritmo DFS

L'algoritmo DFS non prende in considerazione un ordine di scoperta dei nodi, ma fa "come piu gli piace", o "scopre il primo che trova". In questo modo posiamo incorrere in problemi sugli archi. In tal senso andiamo a distinguere vari tipi di archi che possiamo trovare all'interno dell'albero. Tra i vari tipi abbiamo:

- **Archi dell'albero**, cioe archi che scoprono un nodo **bianco** a partire da un nodo **grigio**. E il normale metodo di scoperta dei nodi durante una DFS.
- **Archi all'indietro**, cioe archi che connettono un discendente appena scoperto (quindi **grigio**) a un vertice gia scoperto **grigio**.
- **Archi in avanti**, sono archi che connettono un antenato in fase di scoperta Deep (**grigio**), e un discendente gia scoperto e usato **nero**.
- **Archi di attraversamento**, sono archi che connettono 2 nodi di FDF differenti, quindi non si trovano mai nello stesso albero, ma hanno gli stessi colori degli archi in avanti : da **grigio** a **nero**.

Gli archi all'indietro sono archi che rispettano questa formula: $d[v] < d[u] < f[u] < f[v]$. E la scoperta dell'arco all'indietro avviene al centro delle disequazioni.
I tipi di archi 3 e 4 hanno lo stesso vertice su cui puntano di colore nero, per distinguerli abbiamo bisogno di discriminare i loro tempi:

- Per un arco in avanti (da v a u), la scoperta dello stesso avviene dopo che u e terminato (quindi dopo f[u]) e sempre prima della terminazione di v (prima di f[v]).
- Per un arco di attraversamento i tempi di discovery possono essere disgiunti : d[v] < f[v] < d[u] < f[u].

18.4 Come aggiungere queste informazini all'algoritmo DFS

Per far si che questi nostri ragionamenti sui tipi di archi dobbiamo aggiungere delle righe di codice alla nostra DFS visit.

```
1 DFSvisit(G,s)
2 Col[s] = g
3 d[s] = tempo++ // il tempo viene prima incrementato e poi aggiunto
     a d[s]
4 for each v in Adj[s] do
5 if col[v] = b then
6 pred[v] = s
7 DFS_visit(G,v)
```

```
  else
  if Col[v] = g then
  print '(s,v) e un arco all'indietro'
  else if d[s] < d[v] then
  print '(s,v) e arco in avanti'
  else}
print '(s,v) e un arco di attraversamento'
f[s] = tempo++
col[s] = n
```

18.5 Proprieta archi di ritorno

(Questa sezione ha fonte solo da valentina che era presente a lezione li)

- Se all'interno di una DFS incontriamo un arco di ritorno, allora c'e necessariamente un ciclo. Dimostriamola prendendo π che rappresenta il percorso tra (v, u). Se in questo percorso esiste un arco (u, v), allora c'e un ciclo.
- La non esistenza di un arco all'indietro implica che non esiste ciclo all'interno del percorso. Per dimostrare questa proprieta consideriamo $G = (V, E)$, cioe un grafo che contiene un ciclo. Supponiamo che v_1 sia il primo vertice del percorso e che per il teorema dei percorsi bianchi allora tutti i suoi vertici discendenti ancora dovranno essere scoperti. A seconda di quale dei vertici discendenti abbia un arco di ritorno allora avremo che esistera ciclo. Quindi e dimostrato che se esiste almeno 1 arco all'indietro esiste un ciclo e viceversa se non esiste.

18.6 Algoritmo del grafico Aciclico

```
Acicliclo(G)
For Each v ∈ V do
if Col[v]=b then
ret  = Acicliclo_Visit(G,v)
if ret = false then
return false
return true
```

```
Aciclico_Visit(G,s)
Col[s] = g
For Each v in Adj[s] do
if Col[v] = b then
ret = Aciclico_Visit(G,v)
if ret = false then
return False
else if Col[v] = g then
return false
return true
```

19 Lezione 14/11/2023

19.1 Ordinamento Topologico

Come da titolo il grafo avra bisogno di un tipo di ordinamento per far si che il grafo venga scoperto in maniera tale da rispettare determinate caratteristiche. Dunque sia dato un grafo $G = (V, E)$ esso e in ordinamento topologico $iff \exists$ permutazione di V, cioe una sequenza senza ripetizioni di nodi, tale che $\forall (v, u) \in V : (v, u) \in E \Rightarrow v$ sta prima di u. E non vi siano ripetizioni
Esempio: In questo esempio possiamo notare che abbiamo 2 modi di descrivere un ordinamento topologico:

- ABC
- ACB

Come riconoscere l'ordinamento topologico?
Bisogna andare a osservare soltanto gli archi uscenti. Nel caso in cui un nodo non ha archi entranti, allora da li partira l'ordinamento. Verranno a catena eliminati (dallo scorrimento), il nodo e tutti i suoi archi uscenti associati. E cosi via per tutti i nodi aggiornati.
In questo esempio possiamo notare che ogni nodo ha almeno un nodo entrante, tolgiendone qualsiasi dei 3, non possiamo fare a meno di riscrivere uno dei 3 nodi cancellati, in questo caso incapperemo in un ciclo.
Proprieta : Quando ci troveremo in un ciclo non e possibile avere ordinamento topologico.

19.1.1 Dimostriamo che in un grafo aciclico esiste almeno un ordinamento topologico

- Se G e un grafo aciclico e G' un sottografo di G, allora G' e aciclico $G' = (V', E')$ con $V' \subseteq V$ e $E' \subseteq E \cap (V'xV')$. Questa proprieta si dimostra per assurdo. Allora supponiamo che esiste un percorso ciclico in G' tale che $\pi = v_1, v_2, \ldots, v_k, \ldots, v_1$. Se e un sottografo di G, allora si trattera di sottovertici di G, ma se si tratta di sottovertici di G, allora anche gli archi associati faranno anche parte in G. Dunque se abbiamo che il grafo di partenza e aciclico, non e possibile che il suo sottografo (con gli stessi archi) abbia un ciclo. Questa proprieta pero non vale quando $G' \supseteq G$.
- Se G aciclico allora esiste un vertice $v \in V$ tale che v ha grado entrante 0, ma non viceversa. La dimostrazione di questa sconda proprieta si puo fare negando sia ipotesi che tesi, cosi avremo una dimostrazione equivalente a quando entrambe erano vere. Ogni vertice ha almeno un arco entrante, quindi preso $v_1 \in V$, vertice arbitrario. sappiamo che a quel vertice e attaccato almeno 1 arco e cosi via viceversa per ogni vertice, ma ricostruendo la catena di archi vedremo che i vertici si ripeteranno e quindi ci sara un ciclo. Questo va in contraddizione con la nostra prima proprieta.

Il metodo di creazione di un ordinamento topologico e costruire continuamente un sottografo G', tale che $G' = (V \backslash \{v_1\}, E) \backslash \{(v_1, v) \in E | u \in V\}$
Dunque ad ogni ciclo andiamo e eliminare il vertice con grado 0 con tutti i suoi archi.

19.2 Topologic Order

Per sapere dunque che la struttura possa ammettere un ordine topologico abbiamo bisogno di una struttura per tenerci traccia di tutti i gradi entranti dei nodi. Attraverso la matrice di adiacenza possiamo sapere se un nodo e stato esplorato o meno.
La struttura di cui ci avvarremo e un array dove in ogni cella corrisponde un vertice e al suo interno il numero di archi entranti che ha.

```
1  OrdinamentoTopologico(G)
2    GE = GradoEntrante(G)
3    ForEach v ∈ V do
4      if GE[v]=0 then
5        Q=Accoda(Q,v)
6    While Q!=NIL Do
7      v = Testa(Q)
8      print(v)
9      ForEach u ∈ Adj[v] do
10       GE[u] = GE[u] -1
11       if GE[u] = 0 then
12         Q = Accoda(Q,u)
13   Q=Decoda(Q)
```

19.2.1 Grado Entrante

```
1  GradoEntrante(G)
2    ForEach v ∈ V do
3      GE[v] = 0
4    ForEach v ∈ V do
5      ForEach u ∈ Adj[v] do
6        GE[u]=GE[u]+1
```

Abbiamo usato uno stack ma e possibile anche usare una coda, basta che poi tutte le operazioni siano a tempo costante e siano coerenti con le caratteristiche della Topologic order.

19.3 Ordinamento Topologico in una DFS

```
1  OTDFS(G)
2    Init(G)
3    ForEach v ∈ V do
4      if Col[v]=b then
5        s = OTDFSvisit(G,v,s)
6    printStack(S)
```

```
1  OTDFSvisit(G,v,s)
2    Col[v]=g
```

```
    d[v]=tempo++
    ForEach v ∈ Adj[v] do
      if Col[v] = b then
        s = OTDFSvisit(G,v,s)
    f[v]=tempo++
    Col[v]=n
    s=push(s,v)
```

Lo stack andra a tenere conto di ogni vertice che e stato appena visitato e messo a nero. Il push nello stack avviene precisamente nel momento in cui il tempo di fine e stato dichirato per quel vertice. Per ogni arco dunque $((u, v) \in E)$ avremmo che il tempo di inserimento di u (da cui parte l'arco) e piu avanti del vertice v a cui punta l'arco poiche viene inserito prima quello piu in fondo e poi quello piu in alto nel percorso.
Questa considerazione ci permette essere in regola con il teorema delle parentesi visto che $f[u] > f[v]$. Ma questa condizione non vale per tutti gli archi di un grafo che viene visitato in DFS...
Andremo dunque a distinguere 3 tipologie di archi che possono emergere presi $\forall(v, u) \in E$:

- u e **bianco**, questo, come appena detto precedentemente rispetta il teorema delle parentesi e quindi la condizione per cui e possibile ordinarlo in modo topologico.
- u e **nero**, in questo caso il nodo piu profonodo o e scoperto da un altro nodo di un altro albero FDF oppure gia scoperto da un adiacente del nodo da cui stiamo cercando. In questo caso ancora abbiamo la condizione di Topologic Order perche viene comunque scoperto prima del nodo che stiamo visitando ora.
- u e **grigio**, in questo caso vuol dire che il nodo e stato scoperto perforza gia da un adiacente di v. A questo punto vuol dire che un nodo nella Deep avra come adiacente il nodo u, e quindi vuol dire che un arco punta da quel nodo x a u, questo comporta un ciclo.

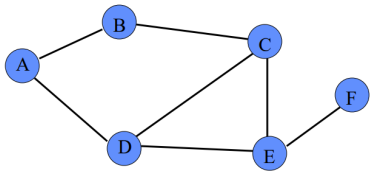
20 Lezione 24 - 16/11/2023

20.1 Grafo (Fortemente) Connesso

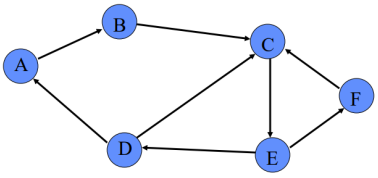
Dato un Grafo $G = (V, E)$ esso è detto fortemente connesso $\iff$

$$\forall (u, v) \in V, (u, v) \text{ sono reciprocamente raggiungibili}$$

(Useremo il termine **fortemente** solo per quanto riguardo i grafi **orientati**)
Nel caso di grafi **non orientati** la **reciproca raggiungibilita**(RR) coincide con la "normale" **raggiungibilita**(R), (RR=R), dato che se $u \sim v \Rightarrow v \sim u$.



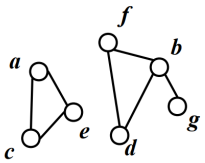
(a) Grafo Connesso



(b) Grafo Fortemente Connesso

20.1.1 Sottografi (Fortemente) Connessi

In un Grafo non (fortemente) connesso, può essere presente un **sottografo** (fortemente) connesso, nella figura seguente possiamo notare un grafo non connesso ma che presenta due sottografi connessi.



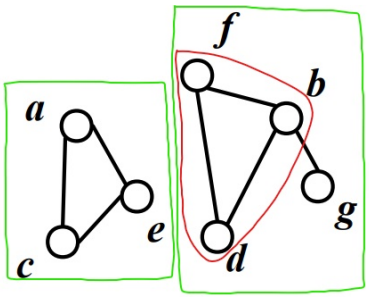
20.2 Componente (Fortemente) Connessa

Dato un grafo $G = (V, E)$ e un suo sottografo G', G' è C(F)C di G $\iff$

- È un sottografo (fortemente) connesso
- È Massimale

Cosa significa Massimale? G' è un sottografo **massimale** se non esiste un altro sottografo fortemente connesso di G che contiene G' come sottografo.

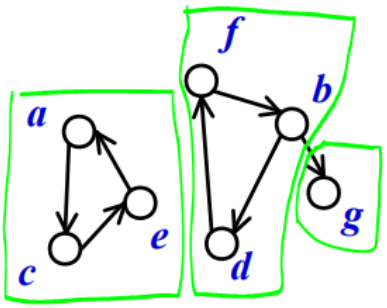
In poche parole vuol dire che non esiste un altro sottografo C(F)C con gli stessi vertici e stessi archi; nel caso in cui si aggiungesse un vertice si verrebbe a perdere la proprietà di massimale.



Consideriamo il seguente grafo, come già osservato non è connesso ma sono presenti dei sottografi connessi, andiamo a verificare quale di questi siano C(F)C.

- (a,c,e): è **componente connessa** poiché è un sottografo connesso, inoltre è **massimale** poiché non possiamo aggiungere altri vertici/archi.
- (b,d,f,g): è componente connessa per gli stessi motivi di sopra
- (b,d,f): **non è componente connessa**, è un sottografo connesso ma non è massimale, poiché il sottografo (b,d,f,g) contiene a sua volta questo sottografo.

Caso Orientato Nel caso dello stesso Grafo ma orientato avremo i seguenti CFC:



Rispetto al caso precedente il grafo (b,d,f,g) non è più cfc poiché se aggiungessimo il nodo g non sarebbe più sottografo connesso ma solo massimale, inoltre il sottografo in cui è presente solo g è cfc poiché non è compreso in nessun altro grafo.

20.3 Raggiungibilità nei C(F)C

Dato u e v in un grafo G, u e raggiungibile da v in $G \iff$

Esiste un percorso (π) da un qualsiasi vertice
della CFC(u) ad un qualsiasi vertice della CFC(v).

In altre parole, stiamo collegando solo i vertici che appartengono a entrambe le Componenti Fortemente Connesse (CFC), senza alcun altro collegamento tra di esse. Quindi, se prendiamo due vertici arbitrari (z dalla CFC di u e k dalla CFC di v), il percorso che stiamo considerando li connetterà le CFC.

Il vantaggio è di poter rappresentare il grafo (magari con molti vertici), in maniera piu compatta e leggera.

Dunque il grafo "compatto" sarà espresso così:

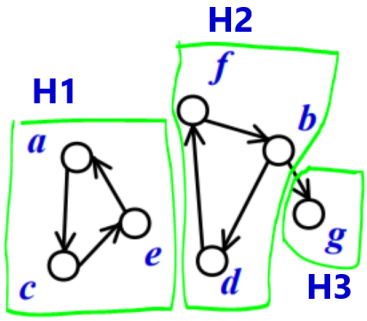
$$G_{CFC} = (V_{CFC}, E_{CFC})$$

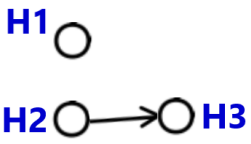
.

Le sue componenti saranno:

- V_{VFC}: contiene un nodo per ogni CFC di G
- E_{CFC}: $(u, v)|u, v \in V_{CFC}, \exists$ un arco in E da un vertice in u a un vertice in v.

Prendiamo il grafo precedente e proviamo a costruire il grafo CFC:





Possiamo notare come per ogni CFC è stato preso un vertice che lo rappresenta, e l'unico arco serve a "simulare" la connessione tra il nodo b e il nodo g che appartengono a due CFC diverse.

Proprietà Il grafo delle CFC di G è un grafo **aciclico**.

20.3.1 Parrallelismo con Algebra

Se fossimo in Algebra, i sottografi CFC sarebbero delle sottoclassi dell'insieme quoziente che è il grafo G stesso. In poche parole i V_{CFC} vanno a rappresentare una classe di equivalenza nella quale tutti i vertici sono in relazione con la classe attraverso la reciproca raggiungibilita (RR). La RR dunque sarà una relazione di equivalenza.

20.4 Costruire un Grafo CFC

Dato un grafo $G = (V, E)$ vogliamo calcolare l'insieme delle componenti fortemente connesse, ognuno degli elementi appartente all'insieme è **un sottografo indotto**, poiché se prendiamo come esempio $CFC(U)$ sarà composto da:

- V_u=$\{v \in V|$ v è reciprocamente raggiungibile con $u\}$
- E_u=$E \cap (V_u x V_u)$ [è proprio la definizione di grafo indotto]

Come possiamo notare la difficoltà sta nel calcolare V_u poiché E_u è facilmente ricavabile dalla definizione.
Inoltre c'è la complessità di calcolo è diversa a secondo del grafo, per un grafo orientato, due vertici sono mutualmente raggiungibili solamente se esiste un ciclo. Ogni coppia di vertici nella mutua raggiungibilita è compresa nell'insieme della raggiungibilita tra vertici.

$$RR \subseteq R \text{ ma non } RR \not\supseteq R$$

.

20.5 Costruzione Grafo CFC - Caso non Orientato

Per un grafo non orientato e piu semplice trovare questo grafo poiche avremo la proprieta sopra menzionata che $RR = R$, quindi qualsiasi arco sara automaticamente reciproco.
Ragionamento 1 *Possiamo eseguire dunque la DFS sul grafo non orientato per trovare la raggiungibilita, che ricordiamo equivalere alla raggiungibilita reciproca.*
La DFS creera un insieme di alberi che andranno a stabilire il numero di CFC. Ogni albero ha dunque l'insieme dei vertici fortemente connessi (V_{CFC}). E ognuno di essi costituira un array(CC) che associa un numero naturale ad ogni sottoalbero creato dalla DFS.

$$CC : V_{CFC} \to N$$

```
1 DFS_Visit(S,i)
2   CC[s]=i
3   DFS_Visit(...,i)
```

Con questa modifica del codice possiamo cosi costruire V_u, dove $V_u = v \in V | CC[v] = CC[u]$
Questo algoritmo impieghera tempo lineare su G e tempo di costruzione dell'array sempre lineare, sommato alla visita.
L'array k non sara vuoto, ma avra come elemento minimo di creazione 1 e elemento massimo $|V|$.
Suggerimento 1 *Nella DFS normale si partira con i = 1 e incrementera ogni volta che incontra una sorgente bianca.*

20.6 Ragionamento per un grafo Orientato

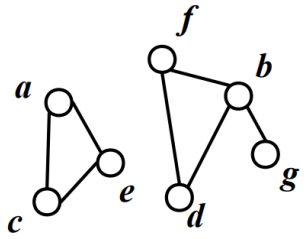
Questo stesso ragionamento non puo valere per un grafo orientato poiche da subito non e vero che ogni albero Della DFS e CFC di G. Dobbiamo ragionare quindi su diverse proprieta che abbiamo a disposizione:
Idea 1
Al termine di una DFS su G siamo sicuri che

Se u e v ∈ alla stessa componente, allora u e v ∈ allo stesso albero Questo vuol dire che una stessa componente non puo stare in piu alberi diversi. Il problema che si propone dunque e dividere gli alberi nelle loro CC diverse, nonstante la DFS crei un albero con piu CC insieme. Dimostrazione nella prossima lezione

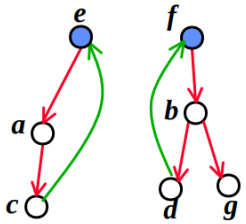
21 Lezione 25 17/11/2023

21.1 Dimostrazione Decomposizione di un albero FDF in CC

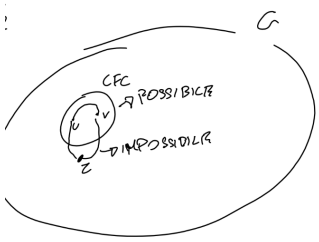
Prendiamo per esempio un albero costruito dalla DFS su questo grafo G:



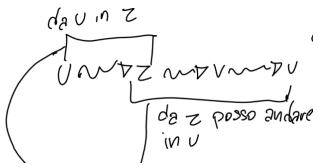
Possiamo notare che nell'albero sono presenti 2 Componenti Connesse insieme.



Ragionamento per Componente Connessa Possiamo dire che se u e $v \in V$ appartengono alla stessa componente (fortemente) connessa ($C(F)C(u) = C(F)C(v)$), allora ogni percorso da u a v appartiene **sempre** alla Componente Fortemente Connessa.



Dimostrazione Potremmo dimostrare questa affermazione per assurdo, prendendo un vertice z, che non fa parte di $CFC(u)$.
Siccome esiste un percorso da z in u, e che sia fortemente connesso per la proprieta di sopra. Vedremo che esistera dunque un percorso da $z \to u$, ma che essendo fortemente connesso per ipotesi allora, $\exists$ un percorso $u \to z$, andando a creare un ciclo. E la condizione necessaria per la quale un grafo sia fortemente connesso e la presenza di un ciclo.



Ragionamento 2 Ora che sappiamo che u e v appartengono alla stessa $C(F)C$, allora possiamo dire che apparterranno allo stesso albero costruito dalla DFS.

Alla partenza della DFS, l'algoritmo avra mandato nella DFSVisit un nodo radice che prendera il nome di z. Da qui possiamo dare per sicure 2 proprieta:

- z sara il primo vertice visitato dalla DFS, ed e anche ovvio.
- all'istante $d[z]$, per il teorema del percorso bianco ogni percorso da $z \to v$ e da $z \to u$ sono completamente bianchi.

Dunque arriveremo a dire che :

- I percorsi da z dell'albero sono tutti una $C(F)C$.
- All'istante $d[z]$ i percorsi sono tutti bianchi e per il teorema del percorso bianco, v e u sono discendenti della foresta di z.

21.1.1 Come sfruttiamo queste proprieta?

Nella foresta avremo CFC diverse in una stessa FDF dopo il passaggio della DFS. In ogni FDF c'e una radice che racchiude tutti i nodi di una componente connessa. Dunque ci basta individuare quali nodi fanno parte della stessa C(F)C, e da quali radici partono quei percorsi per raggiungere i nodi. Dunque potremmo applicare un ragionamento inverso, nella quale dai nodi vorremmo trovare i percorsi che portano alla radice.
Nel nostro esempio, nella prima parte del grafo, i 3 nodi a, c, e raggiungono la stessa radice, mentre per l'altra parte del grafo:
I nodi b, d riescono a raggiungere senza problemi la nostra radice f, mentre g no.

21.1.2 Il Grafo Trasposto

Per ovviare a problemi del genere possiamo creare un **Grafo Trasposto** G^T costruito in questo modo:

Se in $G \exists \pi : u \to v$, allora $G^T \exists \pi' : v \to u$ e viceversa. Quindi un grafo le cui direzioni degli archi sono completamente invertite.

Soluzione La proprieta che prende questo grafo invertito e che ci servira per la definizione delle C(F)C e che se ogni percorso dalla radice r arriva ai suoi vertici v nell'albero FDF e ogni vertice v nel grafo G^T arriva a r, allora potremmo ipotizzare che ogni vertice del grafo faccia parte della stessa $C(F)C(r)$.

Problema Il nostro ragionamento funziona a patto che la DFS sappia di doversi fermare all'interno della stessa C(F)C.
Infatti se dovessimo ricostruire l'albero creato dalla DFS nel grafo trasposto, osserveremo che invece di dividere ancor di piu le C(F)C, ha unito tutto il grafo.

Il problema di questo ragionamento e stato solamente la presenza di un **arco di attraversamento**, che ha unito, nel grafo trasposto, i 2 sottografi, e che, al passaggio della DFS, ha permesso di aggiungere tutti i rimanenti nodi nello stesso albero FDF.

21.1.3 La soluzione agli archi di attraversamento

Se ragioniamo su come funzionano gli archi di attraversamento, vedremo che ognuno di esse connette alberi creati dalla DFS sempre in una stessa direzione. La direzione che prende ha lo stesso verso del **tempo** che impiega la DFS a compiere le sue funzioni. Quindi in poche parole gli archi di attraversamento hanno tutti uno stesso verso.

Ragionamento Nel caso in cui dunque ci troviamo nell'ultimo albero da visitare, non esisteranno archi di attraversamento, quindi la visita DFS sulla k-esima radice dell'albero raggiungera vertici contenuti solamente nell'albero creato da k. In effetti allora, questi stessi vertici andranno a creare la CFC(r_k).

Idea Dunque potremmo utilizzare questa nozione per trovare il modo di "ragionare a ritroso" e selezionare soltanto quei vertici da mandare poi in DFSVisit (Per il grafo inverso) in un certo ordine.

21.2 La soluzione per i grafi orientati

Dunque gli step per riuscire a trovare queste C(F)C sono i seguenti:

- Fare una DFS su G normalmente ma salvarsi le radici in uno stack.

```
1  DFSVariante(G)
2    Init(G)
3    For Each v ∈ V do
4      if col[v]=b then
5        S=DFSVisitVariante(G,v,S)//dove S e uno stack vuoto che
             viene riempito a ogni radice di sottoalbero
```

```
1  DFSVisitVariante(G,v,S)
2    col[v]=G
3    For Each u ∈ Adj[v] do
4      if col[u] ≠ b then
5        S = DFSVisitVariante(G,u,S)
6    S=Push(S,v)
7    Col[v]=n
8    return S
```

- Fare una seconda DFS (modificata) su G^T e selezionando le radici in ordine inverso, cioe dal tempo piu lontano di visita al piu vicino.

```
DFS'(G^T,S)
  Init(G^T)
  While S ≠ ∅ do
    v = Top(S)
    if Col[v] = b then
      DFS'Visit(G',v)
    S = Pop(S)
```

```
DFS'Visit(G',v)
  col[v]=g
  For Each u ∈ Adj[v] do
    if col[v]=b then
      pred[u]=v
      DFS'Visit(G',u)
```

21.2.1 Algoritmo CFC

L'algoritmo C(F)C che regola appunto queste funzioni che abbiamo creato sopra e il seguente:

```
CFC(G)
  s=DFS(G)
  G^T=Trasposta(G)
  DFS'(G^T,S)
```

Questo algoritmo racchiude le funzioni per trovare le componenti fortemente connesse di un grafo.
Al termine di questo algoritmo ogni albero della seconda DFS contiene tutti e soli i nodi della C(F)C(v) dove v e la radice dell'albero formato, senza incorrere in problemi come arco trasposto.

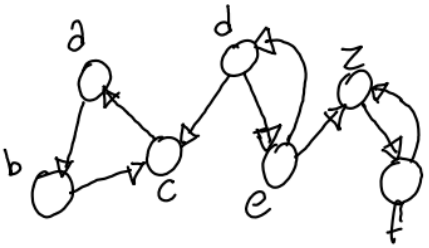
21.2.2 Il costo dell'algoritmo

Il costo della creazione del grafo trasposto e da calcolare:

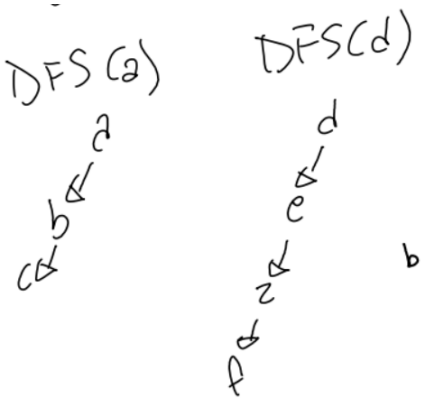
```
Trasposto(G)
  V^T = NIL
  For Each v ∈ V do
    V^T=V^T ∪ {v}
  For Each v ∈ V do
    For Each u ∈ Adj[v]
      Adj'[u] = add(Adj',v)//liste dinamiche
  return (V^T, Adj^T[u])
```

Questo algoritmo di so occupa di scorrere tutte le liste di adiacenza di ogni vertice e invertire l'arco.

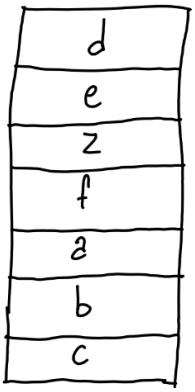
21.3 Esempio



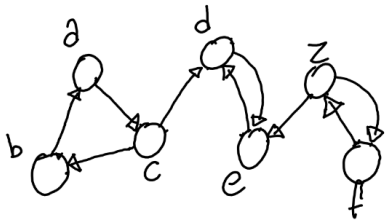
Dato questo grafo, vogliamo provare a trovare le sue componenti fortemente connesse. Dunque, come abbiamo detto andiamo ad eseguire la DFS(a) e la DFS(d) (Partiamo con il primo vertice del grafo *a* e continuiamo con il primo non visitato in ordine alfabetico *d*), visto che con una DFS non si riesce a visitare tutto l'albero.



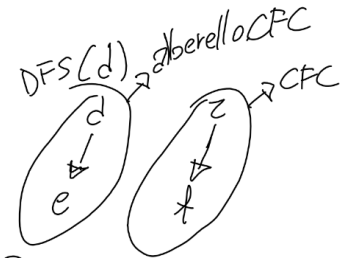
Durante questo processo di DFS, andiamo a prenderci ogni vertice della DFSVisit', che corrisponde a ogni vertice dell'albero di adiacenti creato, in ordine di visita. In particolare andremo ad inserire (Push) dal primo vertice che viene terminato (f[c]) all'ultimo (f[d]) dato che f[d]>f[c].



Dopodichè andiamo a costruire il grafo trasposto grazie alla nostra funzione di trasposizione.



A questo punto prendiamo dallo stack da noi creato il primo elemento e da li creiamo la DFS, continuamo cosi finchè lo stack non diventa vuoto. A quel punto come vedremo avremo creato i nostri alberi C(F)C, come da foto qui sotto.



22 Lezione 26 - 21/11/2023

22.1 Grafi Pesati

I **grafi pesati** come dice la parola si differenziano dai grafi normali per la presenza di pesi sugli archi, andremo ad indicarlo nel seguente modo:

$$G = (V, E, w)$$

in cui w è una "funzione" che associa ad ogni arco un numero reale (peso)

$$w : E \to \mathbb{R}$$

Preso un generico percorso $\pi = v_1v_2v_3...v_k$ il peso di questo percorso sarà dato da:

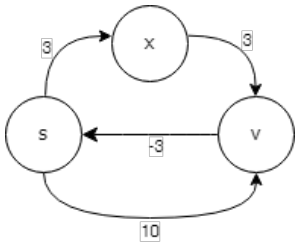
$$w(\pi) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

In questa tipologia di struttura è molto utile andare a calcolare il **percorso minimo**, cioè l'insieme di archi che collegano un vertice u ad un vertice v che sommati hanno il peso minore, possiamo indicarlo così:

$$\delta(u, v)$$

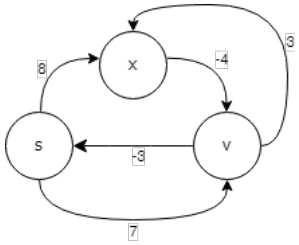
il peso del percorso da u a v con peso minimo (possiamo chiamarlo anche lunghezza)

Esempio 1 Prendiamo come riferimento questo grafo pesato:



prendiamo un percorso $\pi = sxv$ il suo peso sarà $w(\pi) = 3+3 = 6$ questo è il **percorso minimo** da s a v, un altro esempio di percorso $\pi' = sxvsxv$ avrà peso $w(\pi') = 9$

Esempio 2 Prendiamo invece questo grafo con pesi negativi



In questo caso **non esiste un percorso minimo da** s **a** v poiché:

$$\pi = sxv \; w(\pi) = 4$$

$$\pi = sxvxv \; w(\pi) = 3$$

$$\pi = sxvxvxv \; w(\pi) = 2$$

possiamo notare come il peso va sempre a diminuire quindi non c'è un percorso minimo, dovuto dalla preso di un ciclo a peso negativo che ad ogni "iterazione" fa diminuire di unità il peso.

Il problema del percorso minimo non è presente nei grafi privi di cicli a peso negativo

22.2 Algoritmi visita Grafi Pesati

Dato un grafo pesato tramite un Algoritmo di visita BFS possiamo andare a calcolare la stima del peso di un percorso π, ne possiamo distinguere due:

Algoritmo di Dijkstra è usato solo per grafi con pesi non negativi.

Algoritmo Bellman-Ford per grafi pesati arbitrari. Entrambi sono basati sul concetto di "rilassamento" degli archi.

22.3 Rilassamento

Il rilassamento è una tecnica che consente di stimare il percorso con peso minore.
Consideriamo la funzione:

$$d : V \to \mathbb{R}$$

che ad ogni vertice viene associato un reale che corrisponde a una stima del peso.

```
relax(u,v,w):
    if d[v] > d[u] + w(u,v) then //d: distanza, w: peso
        d[v]=d[u]+w(u,v)
        pred[v]=u
```

Se la nuova stima è migliore aggiorno, ovviamente esistono casi in cui la stima è migliore anche se bisogna percorrere più archi.

22.4 Propietà Percorsi Minini su Grafi Pesati

22.4.1 Lemma 1

Dato $G = (V, E, w)$ e $\pi = v_1 v_2 ... v_{k-1} v_k$ percorso minimo da v_1 a v_k allora:

$$\forall 1 \leq i \leq j \leq k, \pi_{ij} = v_i v_{i+1} ... v_j$$

π_{ij} è un percorso minimale da v_i a v_j

Ogni percorso minimo tra due qualsiasi vertici è formato da percorsi minimi ad altri vertici. Non è possibile che un percorso minimo tra due vertici contenga all'interno una sequenza che non è percorso minimo.

Dimostazione Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

22.4.2 Corollario 1

Dato $G = (V, E, w)$ e $\pi = v_1 v_2 ... v_{k-1} v_k$ percorso minimo da v_1 a v_k allora:

$$\delta(v_1, v_k) = \delta(v_1, v_{k-1}) + w(v_{k-1}, v_k)$$

La distanza tra due vertici è il peso del percorso più corto tra i due.

Dimostazione

$$\pi = \underbrace{v_1 v_2 ... v_{k-1}}_{\pi_{1k-1}} v_k$$

$$w(\pi) = \delta(v, v_k)$$

Per il Lemma 1: $w(\pi_{1k-1}) = \delta(v_1, v_{k-1})$

22.4.3 Lemma 2 (Disuglianza Triangolare)

Dato $G = (V, E, w)$ e $s \in V$ per ogni arco $(u, v) \in E$ abbiamo che:

$$\delta(s, v) \leq \delta(s, u) + w(u, v)$$

Il percorso minimo che va da s a v non può essere più lungo del percorso minimo da s ad u con aggiunto l'arco u, v.

Dimostazione Banale.

22.5 Proprietà di Relax

22.5.1 Lemma 3

Dato $G = (V, E, w)$ e $(u, v) \in E$, immediatamente dopo l'esecuzione di relax(u,v,w) avremo che:

$$d[v] \leq d[u] + w(u, v)$$

Dimostazione Banale.

22.5.2 Lemma 4

Dato $G = (V, E, w)$ e posti $d[v] = \infty$ e $\forall v \in V \backslash \{s\}$ e $d[s] = 0$ lungo una qualsiasi sequenza di operazioni di rilassamento avremo che:

$$d[v] \geq \delta(s, v) \forall v \in V$$

In pratica non possiamo mai sottostimare.

Dimostazione Induzione sul numero delle operazioni di rilassamento

Caso base i=0 banale

Passo induttivo i > 0 Ipotesi: prima i-esima operazione $\forall v \in V d[v] \geq \delta(s, v)$.
Eseguiamo relax(x,y,w), poiché modifica solo $d[y]$ sicuramente:

$$\forall v \in V \backslash \{y\}\ d[v] \geq \delta(s, v)$$

Abbiamo due possibilità casi prima di relax:

- 1) $d[y] \leq d[x] + w(x, y)$ allora $d[y] \geq \delta(s, y)$ (per ipotesi)
- 2) $d[y] > d[x] + w(x, y)$ allora $d[y] = d[x] + w(x, y)$

DA RIVEDERE

$$d[y] \geq \delta(s, x)$$

$$\delta(s, x) \leq \delta(s, x) + w(x, y)$$

22.5.3 Corollario 2

Se v non è raggiungibile da s (con inizializzazione delle stime ad ∞ e 0), in ogni momento lungo una sequenza arbitraria di rilassamenti vale:

$$d[v] = \delta(s, v)$$

Dimostazione Se v non è raggiungibile da s allora $\delta(s, v) = \infty$
Per il lemma 4 $d[v] \geq \delta(s, v) \forall v \in V$ quindi avremo:

$$d[v] = \delta(s, v) = \infty$$

22.5.4 Lemma 5

Dato $G = (V, E, w)$ e $\pi = v_1 v_2 ... v_{k-1} v_k$ percorso minimo da v_1 a v_k, inizializzando $d[v] = \infty$ e $d[v_1] = 0$, presa un arbitraria sequenza di rilassamenti che contiene **relax(v_{k-1}, v_k, k)**, se prima di relax $d[v_{k-1}] = \delta(v_1, v_{k-1})$ allora dopo relax $d[v_k] = \delta(v_1, v_k)$

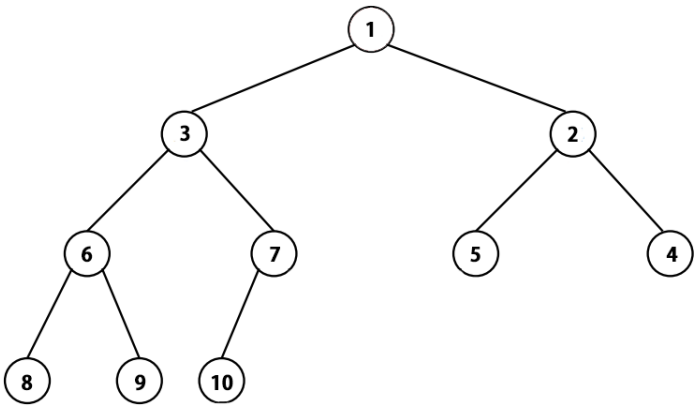
Dimostazione Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortis facilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdiet mi nec ante. Donec ullamcorper, felis non sodales commodo, lectus velit ultrices augue, a dignissim nibh lectus placerat pede. Vivamus nunc nunc, molestie ut, ultricies vel, semper in, velit. Ut porttitor. Praesent in sapien. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Duis fringilla tristique neque. Sed interdum libero

ut metus. Pellentesque placerat. Nam rutrum augue a leo. Morbi sed elit sit amet ante lobortis sollicitudin. Praesent blandit blandit mauris. Praesent lectus tellus, aliquet aliquam, luctus a, egestas a, turpis. Mauris lacinia lorem sit amet ipsum. Nunc quis urna dictum turpis accumsan semper.

23 Lezione 28 - 24/11/2023

23.1 Max/Min Heap

Max/Min Heap è un albero binario **completo** (ogni foglia di trova a profondità h o $h-1$, ogni nodo interno tranne al massimo uno ha grado 2) in cui ogni nodo ha valore maggiore/minore uguale al valore dei sui figli.

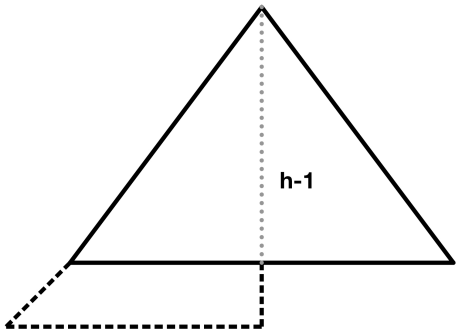


MinHeap

Proprietà Nel Max/Min Heap il max/min è in radice (tempo costante)

23.1.1 Altezza di uno Heap (Albero Completo)

Prendiamo un albero completo di altezza h:



Il "sottoalbero" $h-1$ è un albero pieno quindi contiene $2^h - 1$ nodi.

L'ultimo livello cioè h sappiamo che contiena almeno un nodo quindi i nodi di un albero pieno $+1$ cioé $(2^h - 1 + 1)$, quindi avremo:

$$\underbrace{2^h}_{\text{min. nodi albero compl.}} \leq n \leq \underbrace{2^{h+1} - 1}_{\text{max. nodi albero compl.}}$$

Possiamo riscriverlo nel seguente modo:

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

Andiamo ad applicare il logaritmo:

$$\log_2(2^h) \leq \log_2 n < \log_2(2^{h+1})$$

Andando a svolgere i logaritmi abbiamo:

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

Da qui segue (arrotondando per difetto) che:

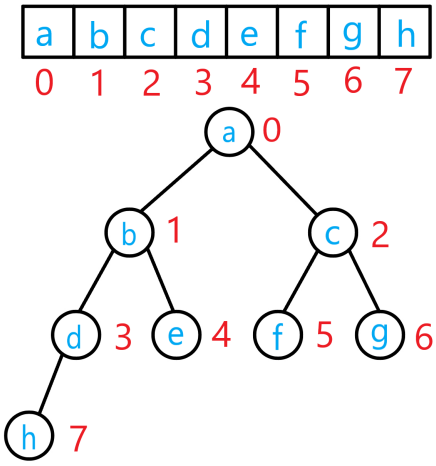
$$h = [log_2 n]$$

23.2 Rappresentare un albero tramite Array

Un altro modo per poter rappresentare un albero è tramite l'ausilio di un array, in cui ogni cella corrisponderà ad un nodo con un indice i e tramite quest'ultimo possiamo risalire ai suoi figli o genitore:

- FiglioSX(i)=$2i$
- FiglioDX(i)=$2i + 1$
- Parent(i)=$\frac{i}{2}$

GLI INDICi PARTONO DA 1 NON DA 0 NON HO VOGLIO DI RIFARE L'IMMAGINE



Per la nostra rappresentazione di Max/Min Heap andremo ad usare proprio un array.

23.3 Rimozione Heap

Rimuovere un nodo da uno Heap non è neccasseriamente un problema, esempio la rimozione di una foglia è abbastanza sicura (anche se possono violare delle proprietà), diverso se vogliamo rimuovere il max/min poiché andiamo a togliere la radice, quindi necessitiamo di una funzione per ripristinare lo heap.

23.3.1 Heapify

La funziona Heapify ripristina la proprietà di Heap al sottoalbero radicato nella posizione i, assumendo che i suoi sottoalberi destro e sinistro siano già degli Heap.

```
Heapify(Q,i): //Q: albero come array, i: posizione sottoalbero
    j=sx(i) //indice figlio sinistro
    k=dx(i) //indice figlio destro
    //Se il figlio sinistro e' piu' piccolo del padre e' min
    if j <= Q.size AND Q[j].val < Q[i].val then
        min=j
    else //altrimenti il min e' il padre
        min=i
    //se il figlio destro e' piu' piccolo del min' precedente
    if k <= Q.size AND Q[k].val < Q[min].val then
        min=k //allora lui diventa il min
    //se il min non e' il padre
    if min != i then
        SCAMBIA Q[i] CON Q[min] //scambiamo min(sx o dx) col padre
        Heapify(Q, min) //chiamata ricorsiva sull'indice min (sx o dx)
```

Ricorsivamente questo algoritmo andrà a ripristinare l'intero heap, il costo di sarà $log_2 n$.

23.3.2 ExtractMin

Andiamo a vedere proprio il caso di cui parlavamo cioè di estrattare il valori in radice (in questo caso il min) e andare ad "aggiustare" l'heap:

```
Extract_Min(Q): //Q: albero come array
    if Q.size >0 then
        ret=Q[1] //la radice (min)
        Q[1]=Q[Q.size] //mettiamo come radice l'ultima foglia
        Q.size=Q.size-1
        Heapify(Q,1) //Heapify su radice (1) per aggiustire l'heap
    else
        ret=NIL
    return ret
```

Tutte queste operazione sono costanti ma dobbiamo aggiungere il costo di Heapify.

23.4 Inserimento Heap

Per andare ad inserire i valori nello heap andiamo a considerare ogni valore come una coppia formato da (valore, priorità),

```
DecreaseKey(Q,i,k): //Q: array, i: indice attuale, k: valore
    if Q[i].val > k then
        O=Q[i].obj //oggetto coppia
        while i > 1 AND Q[Parent(i)].val > k DO
            Q[i]=Q[Parent(i)]
            i=Parent(i)
        Q[i]=(O,k)
```

24 Lezione 30 - 30/11/2023

24.1 Insertion Sort

```
InsertionSort(A, N): //A: array, N: dimensione max array
    //primo valore gia' "ordinato", partiamo dal secondo
    for j=2 to N do
        i=j-1 //valore precedente
        x=A[j] //valore "corrente"
        //se il valore precedente e' maggiore del corrente
        while (i>0 AND A[i]>x) do
            A[i+1]=A[i] //nella cella corrente metto il valore prec.
            i=i-1 //mettiamo i al prec. prec.
        A[i+1]=x
```

24.1.1 Caso Peggiore

Il caso peggiore è quando la sequenza è totalmente in disordine cioé quando i valori sono in ordine decrescente, in questo caso $t_j = j$ e quindi avremo:

$$4n - 3 + 2\sum_{j=2}^{n}(j+1) + \sum_{j=2}^{n} j$$

Andiamo a spezzare la prima sommatoria:

$$4n - 3 + 2\sum_{j=2}^{n} j - 2\sum_{j=2}^{n} 1 + \sum_{j=2}^{n} j$$

Accorpiamo la prima e terza sommatoria

$$4n - 3 + 3\sum_{j=2}^{n} j - 2\sum_{j=2}^{n} 1$$

La prima sommatoria è la somma dei prima n numeri naturali (formula di gauss) meno uno perché partiamo da 2, la seconda è banalemente n tranne uno sempre perché cominciamo da due:

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

Nel caso peggiore è **quadratica**

24.1.2 Caso Medio

Nel caso medio andiamo a fare appunto la media dei possibili valori di $t_j = 1, 2, ..., j$:

$$\frac{\sum_{i=1}^{j} i}{j} = \frac{1}{\cancel{j}} * \frac{\cancel{j}(j+1)}{2} = \frac{j+1}{2}$$

Ora che sappiamo il valore di j andiamo a sostituirlo qua:

$$4n - 3 + 2\sum_{j=1}^{n}(j-1) + \sum_{j=2}^{n} j$$

Quindi avremo:

$$4n - 3 + 2\sum_{j=1}^{n}(\underbrace{\frac{j+1}{2}}_{\frac{j-1}{2}} - 1) + \sum_{j=2}^{n} \frac{j+1}{2}$$

Andiamo a spezzare le sommatorie:

$$4n - 3 + 2\sum_{j=2}^{n} \frac{j}{2} - 2\sum_{j=2}^{n} \frac{1}{2} + \sum_{j=2}^{n} \frac{j}{2} + \sum_{j=2}^{n} \frac{1}{2}$$

max Sommiamo le sommatorie simili:

$$4n - 3 + 3\sum_{j=2}^{n} \frac{j}{2} - \sum_{j=2}^{n} \frac{1}{2}$$

Andiamo a sviluppare le due sommatorie, per $\frac{j}{2}$ possiamo portale il mezzo fuori dalla sommatoria dato che non dipende da j:

$$4n - 3 + \frac{3}{2}\sum_{j=2}^{n} j - \sum_{j=2}^{n} \frac{1}{2}$$

La prima sommatoria è sempre la formula di gauss meno uno, invece la seconda è sempre n meno uno ma diviso due:

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

Il caso medio è **quadratico**

25 Lezione 31 - 01/12/2023

25.1 Selection Sort

Questo è il sort più comune e naive nella scelta. Il sort che viene usato intuitivamente per mettere in ordine le carte da gioco ad esempio.

25.2 Insertion Sort

25.3

25.4 Heap Sort

25.5 Dimostrazione Heap Sort

26 Lezione 32 - 05/12/2023

26.1 MergeSort

COPIA E INCOLLATO DA ALEX
Merge Sort, data un array di lunghezza n idea: Se la lunghezza è sufficientemente grande (arbitrariamente >1), la decompongo in suddivisioni di grandezza 1 che è piu facile ordinare. La soluzione è imporatnte per lunghezza >1. Prendo quindi l'elemento di mezzo dell'array e decompongo la seqeunz in 2 sottosequenze.
Separatamente ordino (con 2 chiamate ricorsive) le due sottosequenze. Mi aspetto che le due sottoseuqenze siano internamente ordinate. Le chiamate fanno ricorsivamente la stessa cosa dell'inizio. cioe vanno a suddividere le sequenze in sottosequenze ordinariamente piu piccole e a metà
Avere 2 sottosequenze ordinate non vuol dire che tutta la sequenza (unita) è ordinata.
A noi interessa dunque costruire la soluzione del problema principale, cioe ordinare completamente le due sottosequenze.
Problema suddiviso in 2 sottoprocedure: - Decomposizione - Ricomposizione delle due sottosequenze

```
MergeSort(A,p,r) //A: Array, p: inizio, r: fine
    if p < r then:
        q=(p+r)/2 //prendiamo punto medio
        MergeSort(A,p,q) //chiamata ricorsiva sulla prima meta'
        MergeSort(A,q+1,r) //chiamata ricorsiva sulla seconda meta'
        Merge(A,p,q,r) //unione delle due meta' ordinate
```

Questo algoritmo funziona solo se vale la seguente condizione:

$$p \leq q < r$$

Questo perché se non valesse ci sarebbe un loop infinito che non porterebbe a termine la compilazione.

Esiste una versione più semplice che usa una struttura d'appoggio (Array), inserendo ogni volta l'elemento più piccolo tra i due insiemi, scartandolo poi quello che è stato inserito, seleziona i minimi delle due sottosequenze ed una volta che ha riempito questa sequenza la prende e la copia in A.

Andiamo a definire la funzione *Merge* che andrà ad unire i due array assumendo che l'input fornito sià già ordinato

```
Merge(A,p,q,r)
    i=p
    j=q+1
    k=p // indice che scorre l'array di appoggio
    while i <= q && j <= r
        if A[i] <= A[j] then
            B[k] = A[i]
            i=i+1
        else
```

```
            B[k] = A[j]
            j=j+1
        k = k+1

    if i ≤ q then
        x=i
    else
        x=j

    while k ≤ r do
        B[k] = A[x]
        x=x+1
        k=k+1

    for k=p to r do
        A[k] = B[k]
```

Quanto costa? Dobbiamo misurare la dimensione dell'input, che è la dimensione delle sottosequenze che dobbiamo sommare (il numero di elementi tra p ed r).

Chiameremo $n = r - p + 1$ il numero di elementi presenti nell'array.

Il for di costruzione farà $n + 1$ operazioni, quindi complessivamente il numero di operazioni è n per il tempo di ogni singola esecuzione di ogni while, while e for (che sono costanti).

$$T_M(n) = n \cdot \theta(1) + \theta(1) = \theta(n)$$

26.2 Equazione di Riccorenza di MergeSort

Andiamo a definire l'equazione di riccorenza del MergeSort nel seguente modo:

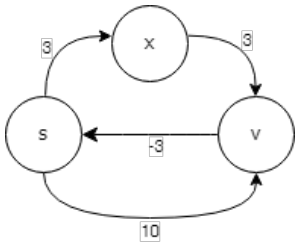
$$T_{MS}(n) = \begin{cases} \theta(1) & n \leq 1 \\ \underbrace{2 \cdot T(\frac{n}{2})}_{\text{due tempo chiamate ricorsive}} + \underbrace{\theta(n)}_{\text{tempo locale}} & n > 1 \end{cases}$$

Dobbiamo trovare una funzione che rende vera questa funzione $T_{MS}(n)$.

Per risolvere dobbiamo costruire l'albero di riccorenza che sarà isomorfo all'albero delle chiamate ricorsive.

Ogni nodo avrà 2 figli con dimensione dell'input uguale, quindi potremmo raccogliere nell'equazione il tempo delle chiamate. Il tempo locale $\theta(n)$ è ovviamente il tempo di $T_{MS}(n)$.

Andiamo a definire l'input della chiamata figlia in funzione dell'input del padre.



(a) Albero di Riccorenza

Ogni nodo ha due valori:

- Dimensione input
- Costo locale

Ad ogni chiamata sia l'input che il costo vanno a dimezzarsi fino ad arrivare alla foglia che sarà il caso base.

Livello	Contributo
0	n
1	$2 * \frac{n}{2} = n$
2	$4 * \frac{n}{4} = n$
$\vdots$	$\vdots$
i	$2^i * \frac{n}{2^i} = n$

Se andiamo a sommare tutti i contributi di ogni livello dell'albero appena creato avremo che danno tutti lo stesso valore n. Grazie a ciò possiamo creare una sommatoria:

$$T_{MS}(n) = \sum_{i=0}^{?} n$$

Come noteremo la sommatoria dovremmo farla sull'altezza dell'albero, cioè fino a quando l'albero delle chiamate ricorsive arriva al caso base. Sappiamo però che quando incontriamo una foglia, allora allo stesso livello tutte le altre chiamate saranno foglie.

Dunque andremo a definire i la nostra altezza incognita.

Raccogliamo le informazioni che abbiamo fin ora:
Man mano che ci addentriamo nell'albero, la grandezza dell'input diminuisce equamente, e all'ultimo livello avrà formula: $\frac{n}{2^i}$. Però sappiamo che all'ultimo livello, essendo foglia, avremo che la dimensione dell'input e quindi della complessità sarà **1**.

Potremmo quindi impostare un'equazione cosi espressa:

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

Abbiamo quindi trovato la dimensione di i, possiamo quindi sostituirla al punto interrogativo.

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

Possiamo dunque approssimare la complessità computazionale di questa equazione di ricorrenza a un caso $n \log_2 n$.
In questo caso è stato semplice arrivare alla soluzione poichè il valore dell'input fornito ad ogni chiamata ricorsiva è sempre lo stesso e non cambia nel tempo. Potremmo dunque provare a fare un altro esempio andando a cambiare la quantità di chiamate ricorsive ad ogni nodo.

26.2.1 Esempio 1 con Equazione modificata

$$T_{MS}(n) = \begin{cases} \theta(1) & n \leq 1 \\ 2 \cdot T(\frac{n}{2}) + n^2 & n > 1 \end{cases}$$

Andiamo sempre a ragionare come nell'equazione di sopra.
Come vedremo per ogni livello ora il contributo locale non è piu n, ma n^2.
In tal senso andiamo a osservare quanto costa ogni livello.

$$T(n) = \sum_{i=0}^{\log_2 n} \frac{n^2}{2^i} = n^2 \sum_{i=0}^{\log_2 n} (\frac{1}{2})^i = n^2 \cdot \theta(1) = \theta(n^2)$$

Il motivo per cui abbiam approssimato a $\theta(1)$ è grazie alla dimostrazione che abbiamo effettuato nelle scorse lezioni.

26.3 Quicksort

Il quicksort è un'altro algoritmo di sorting, considerato il miglior algoritmo di ordinamento data la complessità: $\theta(n \lg n)$. Come analizzeremo più avanti questo algoritmo non sarà sempre conveniente, anzi avrà un caso particolare che lo renderà molto sfavorevole $\theta(n^2)$.

La complessità dell'algoritmo non dipende dalla dimensione di input, ma come viene presentato dall'input stesso.

Questo algoritmo sfrutta sempre lo stesso concetto dell'heapsort dividere l'array in 2 parti. La differenza è che gli elementi vengono ordinati dal minore al maggiore, già alla suddivisione dell'array, così da avere due semiarray ordinati già.

Concettualmente dobbiamo creare una funzione che ci distribuisce gli elmenti minori di un pre-determinato **pivot**, un valore dell'array preso per confrontare tutti gli altri elementi. Quel valore sarà un valore di divisione dei due sottovettori.

Questi confronti devono produrre, dopo già il primo passaggio:

- 2 Sequenze non vuote.
- Tutti i valori di sinistra sono minori dei valori di destra.

```
1  QS(A,p,r)
2      If p<r then
3          q=Partiziona(A,p,r)
4          QS(A,p,q)
5          QS(A,q+1,r)
```

26.3.1 Requisiti e Ragionamento dietro Partiziona

- $p \leq q < r$ che sta a indicare che le partizioni non siano vuote e che gli indici siano distinti di almeno un elemento all'interno dell'array.
- $\forall p \leq i \leq q, \forall q+1 \leq j \leq r, A[i] \leq A[j]$. Questo sta a indicare che per ogni elemento all'interno della sottosequenza di sinistra (i) deve essere minore di ogni elemento della sottosequenza di destra(j).

Una volta stabilite e rispettate queste regole possiamo procedere nel ragionamento alla base del quicksort.

26.3.2 Ragionamento alla base di partiziona

Utilizzerò 2 indici che farò partire rispettivamente all'inizio e alla fine dell'array. Sceglierò un elemento pivot dell'array, che solitamente è il primo elemento della lista. Dunque inizierò iterativamente a far scorrere i due array, confrontando l'elemento in scorrimento con l'elemento pivot. Di seguito verrà fornito il codice di spiegazione dell'algoritmo per capire cosa fa meglio.

```
1   Partiziona(A,p,r)
2       i=p-1
3       j=r+1
4       x=A[p]
5       Repeat //che sta ad indicare una specie di do while
6           Repeat
7               j=j-1
8           Until A[j]<=y
9           Repeat
10              i=i+1
11          Until A[i]>=x
12          If i<j then
13              Swap(A[i],A[j])
14      Until i>=j
15      Return j
```

Dunque andremo a confrontare e scambiare gli elementi dell'array che non rispettano la condizione da loro data.
Come detto nel capitolo precedente questo algoritmo deve rispettare la condizione di $p \leq j < r$. Proviamo che questo algoritmo non funzioni quindi andiamo a dimostrare per assurdo che questo algoritmo termini violando le due disequazioni:

- $p \leq r$
- $j < r$

Assumiamo in partenza che $p < r$, l'algoritmo Partiziona ha appena finito di lavorare e ritorniamo j.

Caso 1 Per assurdo poniamo $j \geq r$, quindi la nostra relazione sarebbe che $p \leq j \leq r \Rightarrow j = r$. Poichè noi inizializziamo j allo stesso valore di r (che inizialmente è uguale a $r+1$), e l'algoritmo fa almeno una volta un decremento di r, j sarà uguale a r. E da lì (se il nostro algoritmo farà il return di quella posizione), non dovrebbe muoversi

più per rispettare la nostra ipotesi.

Dunque i, dall'altra parte confronterà con il primo elemento dell'array, che corrisponde al pivot (x). A questo punto dunque il secondo indice si ferma. Un nuovo ciclo di del repeat esterno partirà e j decrementerà. Ma decrementando andremo a rompere la nostra condizione iniziale, facendo venire la $j < r$.

Caso 2 Poniamo per assurdo che $j < p$. Questo Vuol dire che potrebbe esistere un valore minore del pivot che si trova in posizione p, ma p o è **x** o è stato scambiato quindi sarà minore.

26.3.3 Analisi Asintotica del tempo di esecuzione

Alla fine della chiamata a Partiziona, possiamo ritrovarci in 2 casi:

In entrambi i casi abbiamo al più $n + 1$ incrementi di i e j dal loro inizio (Caso dove j supera i). Mentre in entrambi i casi abbiamo al massimo $\frac{n}{2}$ scambi.
Da ciò possiamo dedurre che abbiamo un $\theta(n)$ di complessità.

$$T_p(n) = \theta(n)$$

26.3.4 Analisi della complessità di QuickSort

Quicksort effettua una chiamata a partizione dunque già avremo la complessità data dalla funzione $\theta(n)$, poi effettua due chiamate ricorsive di seguito. Da ciò possiamo andare a creare un'equazione di ricorrenza date le chiamate ricorsive a catena.

$$T_{QS}(n) = \begin{cases} 1 & \text{per} \quad n \leq 1 \\ T_{QS}(?) + T_{QS}(?) + \theta(n) & \text{se} \quad n > 1 \end{cases}$$

Come potremmo notare non sappiamo a priori (come nel mergesort) le dimensioni dei sottoarray che vengono mandati in chiamata.

Possiamo dunuqe riassumere in q una dimensione parametrizzata della chiamata che andremo a sostituire ai punti interrogativi.

$$T_{QS}(n) = \begin{cases} 1 & \text{per} \quad n \leq 1 \\ T_{QS}(q) + T_{QS}(n-q) + \theta(n) & \text{se} \quad n > 1 \end{cases}$$

Questa q sarà fondamentale nel calcolo della complessità dell'algoritmo. Ovviamente non sappiamo risolvere un'equazione di ricorrenza parametrica, ma a seconda della dimensione di q potremmo farci delle idee sulla complessità dell'algoritmo.

Infatti a seconda della grandezza di q, potremmo trovarci in un **caso peggiore**, **caso migliore** o **caso medio**.

Caso Peggiore Il caso paggiore viene identificato quando abbiamo un grandissimo sbilanciamento nell'albero di ricorrenza. Lo sbilanciamento più grande che un albero può avere è quando $q = 1$, cioè il sottoalbero di sinistra sarà sempre il caso base (1).

Questo sottoalbero sarà degenere verso destra e diminuirà la sua dimensione molto lentamente (di una grandezza a livello).

Come potremmo notare la complessità di questo tipo di albero sarà dato dalla sommatoria di :

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

La complessità di questo caso è quadratica, una complessità molto diversa da quello che ci saremmo aspettati da un quicksort.

Caso Partizioni Uguali Il caso in cui Partiziona riesce sempre a dividere l'array in due parti è letteralmente di complessità uguale al mergeSort.

$$T_{QS}(n) = \begin{cases} 1 & \text{per} \quad n \leq 1 \\ T_{QS}(\frac{n}{2}) + T_{QS}(\frac{n}{2}) + \theta(n) & \text{se} \quad n > 1 \end{cases}$$

Dunque la complessità computazione è la stessa del merge sort:

$$T_{QS}(n) = \theta(n \log_2 n)$$

26.3.5 Una piccola osservazione (Inutile?)

Osserviamo questo statement:

$$\log_2 n = \theta(\log_3 n)$$

questo perchè: $c_1 \log_3 n \leq \log_2 n \leq c_2 \log_3 n$
Per proprietà dei logaritmi possiamo scriver che:

- Proprietà 1:$\log_a b = \frac{1}{\log_b a}$
- Proprietà 2:$\log_2 n = \frac{\log_3 n}{\log_3 2} = \log_3 n \cdot \log_2 3$

Tutto sto ragionamento per dirci alla fine che asintoticamente un logaritmo di qualsiasi base avrà sempre la stessa complessità computazionale di $\theta(\log_3 n)$.

26.3.6 proprietà degli alberi di ricorrenza del Quicksort

Vogliamo dimostrre questa formula:

$$n = 2f - 1$$

dove f è il numero di foglie.

Proviamo a dimostrarlo - Per induzione La dimostrazione avviene sull'altezza dell'albero (h).

Caso Base - Base d'induzione $h = 0$ Allora avremmo un solo nodo nell'albero. $n = 1$, $f = 1 \Rightarrow 1 = 2 * 1 - 1$ Verificato.

Caso induttivo - Passo Induttivo $h > 0$ In questo caso allora il nodo radice avrà due sottoalberi (x, y) che avranno rispettivamente (f_x, f_y) di foglie.

Vogliamo allora dimostrare che valga questa proprietà per tutta l'altezza h, ma per farlo dobbiamo dimostrarlo che vale per $h - 1$. Sottoalberi non vuoti della nostra ipotesi ci possono essere utili.
Per ipotesi induttiva allora abbiamo che la nostra formula vale fino ad $h - 1$:

$$n_x = 2f_x - 1$$

$$n_y = 2f_y - 1$$

Quindi per farlo valere anche per h dobbiamo:

$$n = 1 + n_x + n_y = 1 + 2f_x - 1 + 2f_y - 1 = 2(f_x + f_y) - 1 = 2f - 1$$

Verificato.

Questa proprietà varrà sia per gli alberi completi che per quelli sbilanciati.
Nei pieni ad esempio avremo un numero di nodi che conosciamo 2^{h+1}:
$n = 2^{h+1} - 1 = 2 \cdot 2^h - 1$ $f = 2^h$ negli alberi pieni, dunque è verificato.