Alberi Rosso-Neri

Si illustri la struttura degli alberi rosso-neri.

Gli alberi rosso-neri rappresentano una variante "bilanciata" degli alberi binari di ricerca.
I nodi di un albero rosso-nero contengono i seguenti attributi:

Left – puntatore a figlio sinistro
Right – puntatore a figlio destro
P – puntatore al padre
Key – valore del nodo
Color - colore del nodo (**rosso**/**nero**)
Più eventuali campi satelliti

Inoltre, un albero rosso-nero è un albero binario di ricerca che soddisfa le seguenti proprietà:

Ogni nodo è ***rosso*** o ***nero***.
La radice è **nera**.
Ogni foglia (NULL) è **nera**.

(proprietà di bilanciamento)
Un nodo **rosso** ha solo figli **neri**
Per ogni nodo tutti i suoi cammini semplici che vanno dal nodo ad una foglia discendente contengono lo stesso numero di nodi **neri**.

Si definisca l'altezza nera di un nodo in un albero rosso-nero. Quindi si enunci una minorazione del numero di nodi interni in un sotto albero radicato in un nodo x di un albero rosso-nero e la si utilizzi per dimostrare un limite superiore all'altezza di un albero rosso-nero con n nodi interni.

L' altezza nera di un nodo in un albero rosso-nero equivale al numero di nodi neri in un qualsiasi cammino semplice che va dal nodo ad una sua foglia discendente.
Un sotto-albero radicato in un nodo "x" contiene al più $2^{bh(x)}-1$ nodi interni dove bh(x) è l'altezza nera (black height) del nodo.

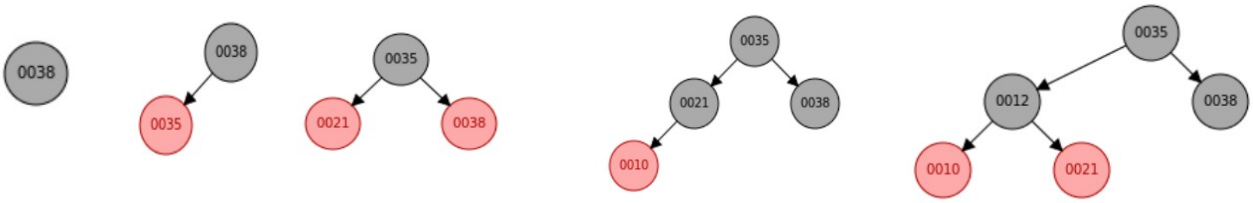
L'altezza (h) di un albero rosso-nero è al più 2*log(n+1).

Dimostrazione:
il numero di nodi **n** del sotto-albero radicato nella radice è >= $2^{bh(root)}-1$, almeno la metà dei nodi di un cammino semplice dalla radice ad una foglia saranno neri quindi

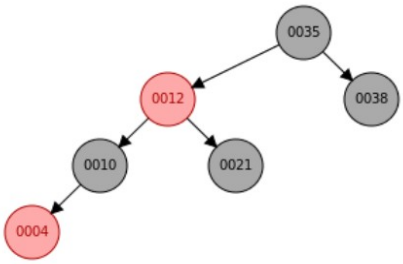
$$bh(root) \geqslant \frac{h}{2}$$

$$n \geqslant 2^{bh(root)}-1 \geqslant 2^{\frac{h}{2}}-1 \quad \text{quindi} \quad n \geqslant 2^{\frac{h}{2}}-1 \quad => \quad h \leqslant 2\log(n+1)$$

Si illustri l'inserimento delle chiavi 38, 35, 21, 10, 12, 4 in un albero rosso-nero inizialmente vuoto



Vedere i vari casi nelle slide di cantone



Huffman

Sia T un testo di 250 caratteri in un alfabeto con dieci caratteri a1 , . . . , a10 , le cui frequenze sono rispettivamente:

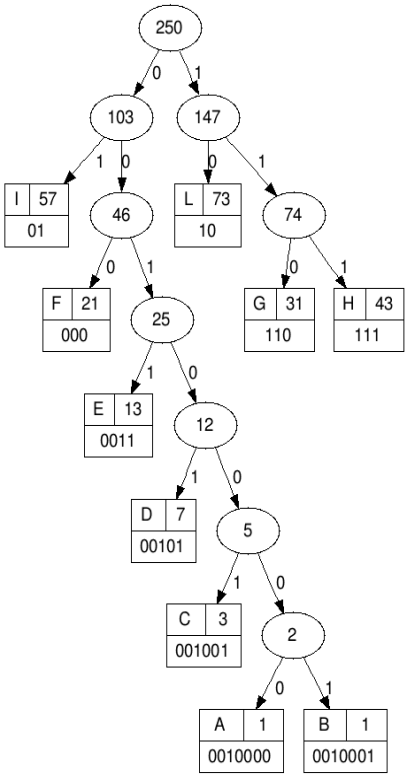
1, 1, 3, 7, 13, 21, 31, 43, 57, 73.

Dopo aver definito la nozione di codice prefisso, si determini il numero minimo di bit necessari per rappresentare il testo T utilizzando un codice prefisso ottimo, illustrando anche l'algoritmo utilizzato.

Una delle proprietà della codifica di Huffman è che nessuna codifica è prefisso di un'altra codifica, questa proprietà permette di leggere un qualsiasi codice codificato e di decodificarlo senza alcuna ambiguità.

lunghezza_stringa = 250

a = a1 = 1	a = 0010000
b = a2 = 1	b = 0010001
c = a3 = 3	c = 001001
d = a4 = 7	d = 00101
e = a5 = 13	e = 0011
f = a6 = 21	f = 000
g = a7 = 31	g = 110
h = a8 = 43	h = 111
i = a9 = 57	i = 01
l = a10 = 73	l = 10



Grafi

Descrizione algoritmo di visita in profondità:

Nella visita in profondità a partire da un vertice si esplorano i gli archi uscenti dell'ultimo vertice raggiunto. Se si scopre un nuovo vertice si esplorano gli archi uscenti di quest'ultimo e così via. Se esplorando un vertice non si scoprono nuovi vertici (quindi tutti gli archi uscenti sono stati già esplorati) si torna indietro dal vertice da cui è stato scoperto il vertice appena esplorato.

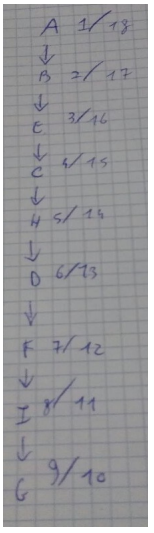
Si continua finché tutti i vertici raggiungibili dal vertice iniziale scelto sono stati scoperti. Se non tutti i vertici del grafo sono stati raggiunti, il procedimento viene ripetuto partendo da un qualunque vertice non ancora raggiunto.

Complessità algoritmo DFS: **O(|V|+|E|)**

Come nella BFS, i vertici sono colorati di
- **bianco** (vertici non ancora raggiunti)
- **grigio** (vertici scoperti)
- **nero** (vertici finiti, la cui lista delle adiacenze è stata completamente esplorata)

La DFS utilizza due marcatempi su ogni vertice **u**.
- **d[u]**: tempo di scoperta, quando il vertice è colorato di grigio
- **f[u]**: tempo di fine visita, quando il vertice è colorato di **nero**.

Sia dato il grafo non orientato G rappresentato dalle seguenti liste di adiacenza



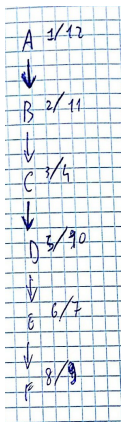
A → B, C, E **B → A, E, F, G** **C → A, E, H**
D → F, H, I **E → A, B, C** **F → B, D, I**
G → B, H, I **H → C, D, G** **I → D, F, G**

Si effettui la visita in profondità del grafo G a partire dal vertice A, indicando per ogni vertice i tempi di inizio e fine visita.

→ →

Sia dato il grafo orientato G con insieme di vertici V = {A, B, C, D, E, F, G} e i cui archi sono rappresentati dalle seguenti liste di adiacenza:

A → B, C, D **B → C, D** **C → A** **D → E, F** **F → D, E**



Si effettui la visita in profondità del grafo G a partire dal vertice A, indicando i tempi di inizio e fine visita per ciascun vertice, e la classificazione di tutti gli archi (es. archi d'albero, all'indietro, ecc.). Si rappresenti inoltre la foresta DFS ottenuta.

→ →

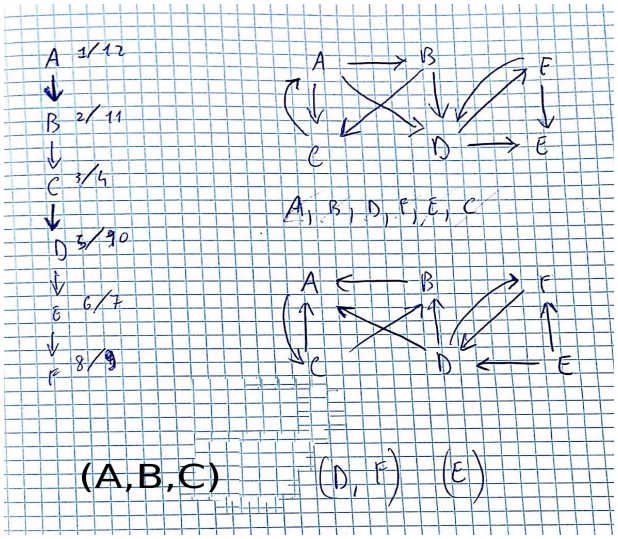
(Facoltativo) Si definisca la nozione di componente fortemente connessa (cfc) di un grafo orientato e si descriva un algoritmo per calcolare le cfc di un grafo orientato. Quindi si determinino le cfc del grafo G.

Una componente fortemente connessa di un grafo orientato è un G = (V,E) è l'insieme massimale dei vertici **U sottoinsieme V** tale che per ogni **u,v appartenenti** ad **U** esiste un cammino da **u** a **v** ed un cammino da **v** ad **u.**

Per calcolare il **cfc** del grafo G si eseguono i seguenti passaggi:

- *ricerca in profondità* del grafo **G** (si ordinano i vertici in ordine di tempo e di fine)
- calcolare grafo trasposto G^T del grafo **G**
- *ricerca in profondità* di G^T usando l'ordine dei vertici calcolati nel primo passaggio

Dopo aver eseguito questi passaggi gli alberi della ricerca in profondità di G^T rappresenteranno le componenti fortemente connesse.



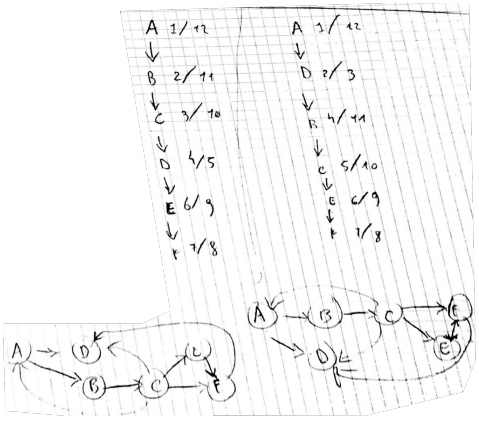
(A,C) (B) (D,F) e (E) sono le componenti strettamente connesse.
(video tutorial: https://www.youtube.com/watch?v=ju9Yk7OOEb8)

Sia dato il grafo orientato G rappresentato dalle seguenti liste di adiacenza:
A → B, D B → C C → A, D, E, F E → F F → D

Si effettui la visita in profondità del grafo G a partire dal vertice A, indicando per ogni vertice i tempi di inizio e di fine visita.

Si effettui inoltre la visita del grafo G a partire dal vertice A, nell'ipotesi che la lista di adiacenza di A sia ordinata così: A → D, B (mentre invece le rimanenti liste rimangano ordinate come sopra). Utilizzare i risultati delle visite per verificare se il grafo G è aciclico.

Il grafo G non è aciclico, perché è costituito da vertici con archi "all'indietro".



Programmazione dinamica

Sia ⊗ un'operazione associativa su matrici di numeri reali tale che, date due matrici *A* e *B* rispettivamente di dimensioni $p \times q$ e $q \times r$, produce una matrice $A \otimes B$ di dimensione $p \times r$, effettuando $p^2 q^2 + r^3$ operazioni elementari.

Sia A = (A1 , A2 , . . . , An) una sequenza di matrici di dimensioni $p_{i-1} \times p_i$ **, per** $i = 1, 2, \ldots, n$.
Utilizzando la metodologia della programmazione dinamica, si descriva un algoritmo per determinare la parentesizzazione della sequenza A che consenta di calcolare la matrice

$$A_1 \otimes A_2 \otimes \ldots \otimes A_n$$

con il minor numero possibile di operazioni elementari.

Pseudo-codice prodotto matriciale con $p^2 q^2 + r^3$ operazioni elementari.

```
PRODOTTO DI MATRICI "RIGHE-PER-COLONNE"
MATRIX-MULTIPLY (A,B)
  p:= rows[A]
  q:= columns[A]
  r:= columns[B]
  for i:= 1 to p^2 do
    for j:= 1 to r^3 do
      C[i,j]:=0
      for k:= 1 to q^2 do
        C[i,j]:= C[i,j] + A[i,k]· B[k,j]
  return C
                          COMPLESSITA' O(p·q·r)
```

Qual è la complessità dell'algoritmo trovato in funzione della lunghezza n della sequenza A ?
Complessità dela funzione Matrix-Multiply(A,B) => O(p * q * r) con operazioni elementari effettuate $O(p^2 q^2 r^3)$.

Per la funzione Matrix_Chain_Order(p), basta aggiungere **p = p * p** come mostra la seguente l'immagine, mentre per la funzione Matrix_Chain_Multiply(A,s,i,j) non bisogna modificare nulla.

```
MATRIX_CHAIN_ORDER(p)
for i:=1 to n do        ← p = p * p // p²
    m[i,i]:=0
for Δ:=1 to n-1 do
  for i:=1 to n-Δ do
      j:= Δ+i
      m[i,j]:=+∞
      for k:=i to j-1 do
          q:= m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
          if q < m[i,j] then
              m[i,j]:= q
              s[i,j]:= k
return m,s
```

Questo algoritmo è un algoritmo di programmazione dinamica, essa è una tecnica basata sulla divisione di un problema in sotto-problemi e sull'utilizzo di una sottostruttura ottimale. Per ***sottostruttura ottimale*** si intende una struttura dove la soluzione ottimale di un sotto problema può essere utilizzata per trovare la soluzione ottimale dell'intero problema.

Perchè è importante supporre che l'operazione ⊗ sia associativa?
Consideriamo i prodotti scalari che generano gli stessi risultati, se l'operazione non è associativa, non abbiamo nessuna certezza che i prodotti finali diano gli stessi risultati

Si consideri la seguente operazione ⊕ sui numeri naturali, definita da: a ⊕ b = Def 3a + 4b .
(a) Si verifichi con un esempio a scelta che l'operazione ⊕ non `e associativa.
(b) Utilizzando la metodologia della programmazione dinamica, si determini un algoritmo che, data una sequenza di numeri naturali $a_1, a_2, \ldots, a_n$, calcoli il valore massimo che l'espressione $a_1 \oplus a_2 \oplus \cdots \oplus a_n$ possa assumere al variare di tutte le possibili parentesizzazioni.

(a) l'operatore a ⊗ b: 3a + 4b

1 ⊗ 0 = 3 * 1 + 4 * 0 = 3
0 ⊗ 1 = 3 * 0 + 4 * 1 = 4

(1 ⊗ 0) ⊗ 1 = 3 ⊗ 1 = 3 * 3 + 4 * 1 = 13
1 ⊗ (0 ⊗ 1) = 1 ⊗ 4 = 3 * 1 + 4 * 4 = 19

(1 ⊗ 0) ⊗ 1 != 1 ⊗ (0 ⊗ 1), l'operatore ⊗ non è associativo

(b) $A = \{ a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus \ldots \oplus a_n \}$
Sia E una parentesizzazione ottima, tale che il risultato di tale parentesizzazione dia il massimo valore ottenibile dalle operazioni.
Supponiamo che n >= 2.
Con E_1 intendiamo la parentesizzazione che va da $(a_1 \oplus \ldots \oplus a_k)$
Con E_2 intendiamo la parentesizzazione che va da $(a_{k+1} \oplus \ldots \oplus a_n)$ $\quad 1 \leq k \leq n-1$
Quindi $E = E_1 \oplus E_2$
Poiché $E = \#(E) = 3E_1 + 4E_2$
Ne segue che le parentesizzazioni ottime di $E_1 = (a_1 \oplus \ldots \oplus a_k)$ ed $E_2 = (a_{k+1} \oplus \ldots \oplus a_n)$ verranno risolte in modo analogo, fino ad arrivare al sottoproblema più piccolo.
Pertanto la classe di sottoproblemi da risolvere è data da:
$\{(a_i \oplus \ldots \oplus a_j) : \quad 1 \leq i \leq j \leq n \quad \}$
con m[i.j] = valore massimo della parentesizzazione.

(definizione ricorsiva del costo di una parentesizzazione ottima adattata a questo problema***)***

$$m[i,j] = \begin{cases} A_i & \text{se } i = j \\ \max(3m[i,k] + 4m[k+1,j]) & \text{se } i < j \\ \quad \text{con} \quad i \leq k < j & \end{cases}$$

Matrix_Chain_Order(A)

```
for i = 1 to n
        m[i,i] = a_i
for x = i to n-1
        for i = 1 to n-x
                j = x+i
                m[i,j] = -infinito
                for k = i to j-1
                        q = 3m[i,k] + 4m[k+1, j]
                        if q > m[i,j]
                                m[i,j] = q
                                s[i,j] = k
return m, s;
```

Matrix_Chain_Multiply(A, s, i, j)

```
if i = j
        return A_i
else
        X = Operator(A, s, i, s[i,j])
        Y = Operator(A,s, s[i,j]+1, j)
        return 3X+4Y;
```

Greedy

Nel contesto della metodologia greedy, si enunci il problema di ottimizzazione relativo alla selezione di attività e se ne discuta una soluzione efficiente, valutandone la complessità computazionale e illustrandola sul seguente insieme S = {a1 , . . . , a10 } di attività, caratterizzate dai seguenti tempi iniziali e finali:

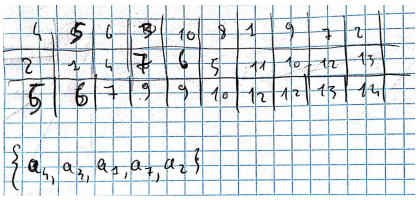
i	1	2	3	4	5	6	7	8	9	10
s_i	11	13	7	2	1	4	12	5	10	6
f_i	12	14	9	5	6	7	13	10	12	9

Il **problema della selezione di attività** consiste nel determinare un sottoinsieme di **cardinalità massima (A)** di attività mutuamente compatibili.

Una possibile soluzione potrebbe essere la “ricerca esaustiva” ma essa risulta essere una soluzione inefficiente (complessità: $\Omega(2^n)$). Una “soluzione ottima” ed alternativa alla “ricerca esaustiva” si può trovare tramite la metodologia greedy, un algoritmo greedy è un algoritmo che cerca di ottenere una soluzione ottima da un punto di vista globale attraverso la scelta della soluzione più *golosa* (greedy) ad ogni passo locale.

Per il **problema della selezione di attività** è possibile trovare un algoritmo di complessità $\mathbf{O(n^3)}$ che trova la cardinalità massima di attività mutuamente compatibili.

Gli insiemi delle attività mutuamente compatibili con cardinalità massima sono:



$\{a_4, a_3, a_1, a_7, a_2\}$

Greedy – Huffman

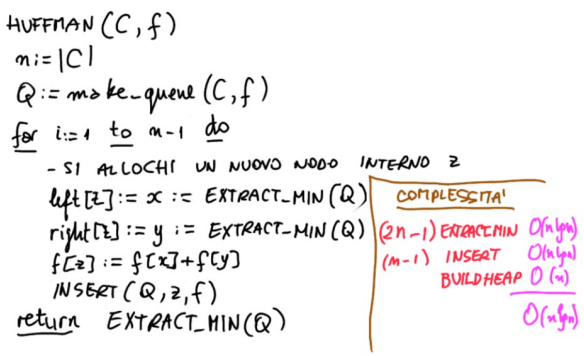
Si definisca la nozione di codice prefisso. Quindi si illustri l’algoritmo di Huffman (anche con pseudocodice) e la sua applicazione principale.

Codice prefisso è una proprietà di quelle codifiche che, una volta codificato un carattere, quest'ultimo non è prefisso di nessun altra codifica.

L'algoritmo di Huffman è un algoritmo di codifica che genera un codice con le seguenti proprietà ovvero:

- esso è un codice prefisso
- è un codice di lunghezza variabile
- i caratteri meno frequenti sono rappresentati con più bit, al contrario, i caratteri più frequenti sono rappresentati con meno bit

Psudocodice:



L' algoritmo di **Huffman**, inoltre, è un esempio di ***algoritmo greedy*** poiché fonde due nodi (caratteri) con frequenza minore senza considerare gli altri, una volta fatta questa scelta la mantiene fino alla fine.