

--Definire il costo del criterio logaritmico per il modello RAM e si valuti il costo di ADD *<address> :	2
--Si dia la definizione di componente biconnessa in un grafo non orientato connesso :	2
--Si descriva la procedura ADDSON e se ne illustrino gli utilizzi :	2
--Si dia la definizione di albero binario di ricerca e se ne illustri l'utilizzo :	3
--Si dia un algoritmo che, dato un insieme di elementi con associate probabilità di accesso, costruisca un albero di ricerca ottimo :	3
-Si dia un programma RAM che, dato in input X, calcola $3X$. Si analizzi la complessità sia rispetto al modello di costo uniforme che a quello logaritmico :	4
--Si dia la definizione di funzioni in relazione polinomiale tra loro e si provi o si confuti che le funzioni $f(n) = 2n$ e $g(n) = 22n$ sono in relazione polinomiale tra loro :	3/4
--Si consideri un grafo diretto e pesato $G = (V, E)$, in cui ogni arco ha peso non-negativo, si dia un algoritmo che calcola i cammini ottimi da un vertice sorgente a tutti gli altri vertici (Single Source Shortest Path Algorithm). si analizzi l'algoritmo (Il punteggio terrà conto dell'efficienza dell'algoritmo da voi proposto) :	5
--Si dia la definizione di Macchina di Turing non-deterministica e si illustri come tale macchina può essere simulata da una macchina di Turing deterministica :	5
--Si dia un algoritmo che dato in Input n, Calcoli 3^n :	6
--Si descriva la procedura Implant :	6
--Provare che 3-Sat è Np-Completo :	7
--Si dia la definizione di relazione polinomiale fra due funzioni :	7/8
--Si illustri con un esempio la differenza fra criterio di costo uniforme e criterio di costo logaritmico per modelli di Calcolo RAM :	8
--Definire un algoritmo RAM che calcoli 3^n e analizzare la complessità secondo il criterio secondo il criterio di costo uniforme e il criterio di costo logaritmico :	9
--Definire un algoritmo RAM che calcoli x^4 e analizzare la complessità secondo il criterio di costo uniforme e il criterio di costo logaritmico :	10
--Si indichi in quali circostanze il modello RAM è in relazione polinomiale con la Macchina di Touring :	10
--Si considerino le funzioni x^k e k^x e con k costante, stabilire se le funzioni sono in relazione polinomiale fra loro :	10
--Definire il costo logaritmico per RAM e valutare il costo dell'istruzione LOAD *<address> :	11
--Si dia un vettore di interi positivi $V = \{v_1, v_2, \dots, v_n\}$ sia dia un algoritmo che per individuare la differenza massima fra due elementi dell'insieme V abbia complessità di tempo $O(n)$ e faccia al più $(3/2)n - 2$ confronti :	12
--Fornire un algoritmo per la moltiplicazione di due interi binari ad n cifre :	13
--Si dia un algoritmo di ordinamento basato sulla tecnica delle partizioni bilanciate e si analizzi la complessità nel caso pessimo. Il nostro algoritmo è attuale rispetto al modello di calcolo RAM? :	14/15
--Nel modello di calcolo dato da alberi di decisione, si provi che ordinare n numeri richiede $\Omega(n \log n)$ confronti :	16
--Date n matrici M_1, \dots, M_n tali che M_i e M_{i+1} sono compatibili per il prodotto	
è dia un algoritmo polinomiale che calcola il modo ottimo di fare il prodotto $M_1 \times M_2 \times \dots \times M_n$:	16/17
--Dato un nastro di dimensione L e siano p_1, \dots, p_n programmi di dimensione l_1, \dots, l_n tali che $\sum l_i \leq L$. Si dia un algoritmo che trovi le sequenze p_1, \dots, p_n di programmi da memorizzare sul nastro modo che il tempo medio di lettura di tutti i programmi sia minimo p :	18
--Trovare la soluzione attuale (versione greedy) del problema di uno zaino di capacità $M=80$, dati 8 oggetti di valore $P=\{18, 5, 6, 2, 9, 14, 36, 24\}$ e dimensione $W=\{9, 15, 12, 18, 36, 14, 4, 8\}$:	20
--Si descriva l'algoritmo di Union e Find che utilizza strutture dati ad albero e lo si analizzi. La complessità di tempo deve essere $O(1)$ e quella di Find deve essere $O(\log(n))$ dove n è il numero massimo di elementi dell'universo U :	22
--Albero Binario di Ricerca; si definisca un albero binario di ricerca e si diano le procedure di Member, Insert e Delete. Si analizzino tali procedure. Si dia poi una procedura per la costruzione di un albero binario di ricerca e si analizzi anche quella :	24/25
--Si definisca un albero binario di ricerca ottimo :	28
--Si definisca un albero di ricerca ottimo e si dia un algoritmo che lo costruisca in $O(n^3)$:	29
--Si dia la definizione di alberi 2-3 con esempio, e limitazioni superiori e inferiori all'altezza con n foglie :	31
--Si presenti la procedura ADDSONS e se ne discuta il suo utilizzo nell'ambito dei dizionari :	33
--dare la definizione di visita BFS e visita DFS di un grafo. :	34
--Si definisca il minimo albero di ricoprimento di un grafo e si enuncino gli algoritmi di Kruskal e Prim, facendo una loro valutazione :	37
--Si definisca il concetto di cammino minimo in un grafo e si fornisce un algoritmo di chiusura transitiva utilizzando un semianello generico e lo si analizzi :	39
--Si consideri un grafo diretto e pesato $G = (V, E)$ in cui ogni arco ha peso non negativo.	
Si dia un algoritmo che calcola i cammini attivi da un vertice sorgente a qualsiasi altro vertice del grafo (Single souce shortest Path). :	42
--Si dia la definizione di componente biconnessa in un grafo non orientato e connesso :	44
--Definire l'algoritmo di Bellman Ford :	45
--NP-COMPLETEZZA: CLASSI DI PROBLEMI:	47
--Dimostrare che il problema 3-SAT è NP-COMPLETO :	48
--Dimostrare che 3-sat sia NP-completo :	49
--Problema del Clique per Grafi non Diretti :	50
--Dimostrare che il Vertex-Cover è un problema NP-Completo :	52
--Subgraph somorph problem :	52
--Tabella Complessità	53

DOMANDA (07/2010): Definire il costo del criterio logaritmico per il modello RAM e si valuti il costo di ADD *<address>. 5 pt.

RISPOSTA: Il criterio del costo logaritmico è usato per calcolare il costo spaziale e temporale di un algoritmo.

Definiamo una funzione logaritmica $l(i)$, definita come:

$$l(i) = \begin{cases} \log(i)+1 & i \neq 0 \\ 1 & i=0 \end{cases}$$

e una funzione $c(i)$ che mappa la memoria (c accede al valore contenuto nell'indirizzo di memoria i) della macchina RAM.

Il costo dell'istruzione ADD *<address> è dato da:

- il costo dell'istruzione ADD: $l(c(0))$;
- il costo dell'operatore * che prende il contenuto di address: $l(c(c(address)))$;
- il costo dell'indirizzo stesso: $l(address)$.

DOMANDA (07/2010): Si dia la definizione di componente biconnessa in un grafo non orientato e connesso. 5 pt.

RISPOSTA: Un grafo indiretto e connesso (o una componente) G , è biconnesso se, data una qualsiasi tripla di vertici distinti v, w, a , esiste un cammino tra v e w che contiene a (chiamato "punto di articolazione", ovvero un vertice, che, se eliminato, separa G in una o più parti).

DOMANDA (01/02/2012): Si descriva la procedura ADDSON e se ne illustrino gli utilizzi. 15 pt.

RISPOSTA: La procedura ADDSON è usata per l'inserimento di elementi in un albero 2-3. Richiede in ingresso un albero non vuoto T , r radice e a elemento da inserire. Restituisce l'albero 2-3 con l'aggiunta del nuovo elemento. L'algoritmo ha un costo di $O(\log n)$ tempo.

- Se T contiene un solo elemento I di valore b , creiamo una nuova radice r' . Creiamo la foglia v con valore a . I e v li facciamo diventare figli di r' rendendo I figlio sinistro se $b < a$, altrimenti sarà figlio destro.
- Se T ha più di un vertice, sia f la radice in cui inserire il nodo a , rendendolo il figlio o di sinistra o di centro o di destra. Se f ha già tre figli, si rende a figlio di f e si chiama ADDSON(f) per inserire f in T .

DOMANDA (01/02/2010): Si dia la definizione di albero binario di ricerca e se ne illustri l'utilizzo. 5 pt.

RISPOSTA: Un ABR è un tipo di albero in cui ogni nodo può avere al più due figli,e con la particolarità che partendo dalla radice r:

$$\begin{aligned}\forall a, & \text{ se } a < r, a \in T_s \\ \forall b, & \text{ se } b > r, b \in T_d\end{aligned}$$

Nella categoria degli insiemi, se rappresentiamo un insieme S con un ABR, possiamo implementare in modo efficiente la funzione MEMBER(a,S) che restituisce "vero" se l'elemento a fa parte dell'insieme S.

Anche le funzioni DELETE e MIN (in un ABR il valore minimo si trova nella foglia più a sinistra di tutto l'albero) possono essere implementate in tempo efficiente con un ABR.

DOMANDA (01/02/2010): Si dia un algoritmo che, dato un insieme di elementi con associate probabilità di accesso, costruisca un albero di ricerca ottimo. 12 pt.

RISPOSTA: Sia dato S l'insieme di elementi $\{a_1, \dots, a_n\}$, e due insiemi q_1, \dots, q_n e p_1, \dots, p_n di probabilità.

I valori q_1, \dots, q_n sono le probabilità che venga eseguita una istruzione del tipo MEMBER(a, S) con q_i (con i compreso tra 1 ed n) tale che $a_i < a < a_{i+1}$; i valori p_1, \dots, p_n denotano invece la probabilità che MEMBER(a_i , S) venga eseguita.

- Per $0 \leq i < j \leq n$ calcoliamo r_{ij} e c_{ij} i in modo da incrementare il valore di $j-i$, usando la programmazione dinamica:

Begin

for $i \leftarrow 1$ **until** n **do**

begin

$w_{ii} \leftarrow q_i;$

$c_{ii} \leftarrow 0$

end;

for $l \leftarrow 1$ **until** n **do**

for $i \leftarrow 0$ **until** $n-l$ **do**

begin

$j \leftarrow i+1;$

$w_{ij} \leftarrow w_{i,j-1} + p_j + q_j;$

 sia m un valore di k , $i < k \leq j$, per il quale $c_{i,k-1} + c_{kj}$ è minimo;

$c_{ij} \leftarrow w_{ij} + c_{i,m-1} + c_{mj};$

$r_{ij} \leftarrow a_m$

end

end

end.

- Dopo aver calcolato le r_{ij} , chiamiamo BUILDTREE(0,n) per costruire ricorsivamente un albero ottimale per T_{0n} .

Procedura BUILDTREE(i,j):

begin

 crea il vertice v_{ij} , radice di T_{ij} ;

 etichetta v_{ij} da r_{ij} ;

 sia m il sottoscritto di r_{ij} ;

```

if i < m-1 then
    rendi BUILDTREE(i,m-1) il sottoalbero sinistro di vij;
if m < j then
    rendi BUILDTREE(m,j) il sottoalbero destro di vij;
end

```

DOMANDA (01/02/2012): Si dia un programma RAM che, dato in input X, calcola 3^X . Si analizzi la complessità sia rispetto al modello di costo uniforme che a quello logaritmico.

RISPOSTA:

```

READ 1
LOAD =1
STORE 2
LOAD 1
JZERO END
WHILE:
    JZERO END
    SUB =1
    STORE 1
    LOAD 2
    MULT =3
    STORE 2
    LOAD 1
    JUMP WHILE
END:
    WRITE 2
    HALT

```

Per il criterio del costo uniforme (che considera che ogni istruzione vale 1 sia temporalmente che spazialmente), il costo è dato dal ciclo che viene eseguito X volte.

Per il criterio del costo logaritmico invece, sempre generalizzando per il ciclo, possiamo giungere a questa sommatoria:

$$\sum_{i=1}^n \log(3^i) + \log_3 31$$

che indica il costo a ogni iterazione della moltiplicazione nel ciclo. Risultato $O(n^2)$.

DOMANDA (02/2011): Si dia la definizione di funzioni in relazione polinomiale tra loro e si provi o si confuti che le funzioni $f(n) = 2^n$ e $g(n) = 2^{2n}$ sono in relazione polinomiale tra loro.

RISPOSTA: Due funzioni $f(n)$ e $g(n)$ sono in relazione polinomiale tra loro \Leftrightarrow esistono due polinomi P e Q tali che

$$f(n) \leq P(g(n)) \wedge g(n) \leq Q(f(n))$$

Nel nostro caso prendiamo $P(x)=x$ e $Q(x)=x^2$. Quindi sostituendo:

$$2^n \leq P(2^{2n}) = 2^{2n} \text{ che è vero}$$

$$2^{2n} \leq Q(2^n) = (2^n)^2 = 2^{2n} \text{ che è anche vero}$$

In conclusione, le due funzioni sono in relazione polinomiale tra loro.

DOMANDA (07/2010): Si consideri un grafo diretto e pesato $G = (V, E)$, in cui ogni arco ha peso non-negativo. Si dia un algoritmo che calcola i cammini ottimi da un vertice sorgente a tutti gli altri vertici (Single Source Shortest Path Algorithm). Si analizzi l'algoritmo (Il punteggio terrà conto dell'efficienza dell'algoritmo da voi proposto). 15 pt.

RISPOSTA: L'algoritmo per la ricerca del più piccolo cammino partendo da una singola sorgente, ci viene da Dijkstra, e funziona così:

- diamo in input un grafo diretto $G = (V, E)$, una sorgente $v_0 \in V$ e una funzione che va da E a \mathbb{R}^+ . Prendiamo $l(v_i, v_j)$ come $+\infty$ se (v_i, v_j) non è un arco, $v_i \neq v_j$, e $l(v, v) = 0$.
- come output avremo, per ogni vertice v , il minimo di tutti i percorsi P da v_0 a v della somma delle etichette di tutti gli archi di P .

Costruiamo un insieme $S \subseteq V$ tale che il percorso più piccolo dal vertice iniziale a ogni vertice v in S , si trovi interamente in S . L'array $D[v]$ contiene il costo del corrente percorso più piccolo da v_0 a v passando solo attraverso vertici di S .

Qui lo pseudo-codice:

```

begin
1  $S \leftarrow \{v_0\}$ ;
2  $D[v_0] \leftarrow 0$ ;
3 foreach  $v$  in  $V - \{v_0\}$  do  $D[v] \leftarrow l(v_0, v)$ ;
4 while  $S \neq V$  do
    begin
5      scegli un vertice  $w$  in  $V - S$  tale che  $D[w]$  è un minimo;
6      aggiungi  $w$  in  $S$ ;
7      foreach  $v$  in  $V - S$  do
8           $D[v] \leftarrow \text{MIN}(D[v], D[w] + l(w, v))$ ;
    end
end

```

La parte predominante dell'algoritmo è quella delle righe 7-8, che viene eseguita in $O(n)$ passi così come la riga 5. L'algoritmo ha un costo totale di $O(n^2)$.

DOMANDA (02/2011): Si dia la definizione di Macchina di Turing non-deterministica e si illustri come tale macchina può essere simulata da una macchina di Turing deterministica. 15 pt.

RISPOSTA: Una macchina di Turing non deterministica (NDTM abbr.) è definita come una tupla del tipo $NDTM = (Q, T, I, \square, b, q_0, q_f)$ dove tutti i componenti hanno lo stesso significato di quelli di una DTM (Macchina di Turing Deterministica).

L'unica differenza risiede nella funzione δ che è definita come $Q \times T^k \rightarrow Q \times (\{T \times \{L, R, S\}\})^k$.

Una NDTM partendo da uno stato iniziale, con la funzione di movimento si sposta parallelamente in diversi stati, dando origine di fatto a diverse copie della stessa macchina con nastro e stati diversi.

Possiamo simulare una NDTM di complessità $D(n)$ con una DTM di complessità $2^{D(n)}$ ovvero in tempo esponenziale. Immaginando lo sviluppo di una NDTM con una struttura ad albero (albero di computazione), basterà lanciare una BFS (visita in ampiezza) su tutto l'albero (che prenderà $O(C^{D(n)})$ tempo) fino a raggiungere lo stato di accettazione.

Domanda 1 Appello 02/2008:**Valutata 10/10**

Si dia un algoritmo che dato in Input n, Calcoli 3n

RISPOSTA:

```

READ1           Leggo n
LOAD1
JGTZ POS        Se n>0 Salta a Pos
WRITE =0
JUMP END

```

```

POS: LOAD=3
      STORE 2    Conservo il valore 3 nel registro 2
      LOAD 2
      MULT 1     Moltiplico 3 per il valore di n nel primo registro
      STORE 2
      WRITE 2    Scrivo il risultato
END : HALT

```

Secondo il criterio di costo logaritmico la complessità dell'algoritmo è $O(\log_n)$ cioè il logaritmo della dimensione dell'imput.

Domanda 2 02/2008- Si descriva la procedura Implant:**Valutata 15/15****Risposta:**

```

IMPLANT( T1,T2 )
begin
  if H( T1 ) = H( T2 ) then
    crea la radice R e rendi T1 e T2 i figli sinistro e destro di R else Begin
    "assegniamo che H(T1)>H(T2) altrimenti basta scambiare T1 con T2 e destra
    con sinistra"
    sia V il nodo più a destra dell'albero T1 con
    d(v)=H(T1)-H(T2)
    sia f il padre di v
    rendi T2 il figlio più a destra di f
    if f ora ha 4 figli then
      addson(f)
    end
  end
  con H(T) Altezza albero D(V)=Profondità nodo

```

La procedura presi due alberi T1 e T2 in imput controlla che le altezze siano uguali,in quel caso crea una radice r e rende T1 e T2 figli di r, in questo modo l'altezza dell'albero è aumentata di 1, altrimenti, dopo aver controllato quale albero è più alto, certa in quell'albero un nodo più a destra che abbia profondità uguale alla differenza delle due altezze, cerca il padre f di v e rende l'albero più basso figlio di quel vertice se il vertice ha adesso 4 figli, richiama la procedura adson sul nodo f così da ribilanciare l'albero. La procedura impant viene utilizzata negli alberi 2-3 che rappresentano insiemi (ad esempio i mergeable heaps) per effettuare l'operazione di unione fra i due alberi, e quindi fra i due insiemi, secondo il criterio che l'albero di altezza maggiore, cioè l'insieme con cardinalità

minore viene unito all'insieme che ha cardinalità maggiore, considerando che gli elementi nell'albero non sono ordinati, nella procedura non si effettuano controlli per mantenere gli elementi in un certo ordine, però bisogna conservare la struttura dell'albero 2-3, cioè ogni nodo che non è una foglia ha 2 o 3 figli, e lo si fa con la procedura addson che sistema l'albero mantenendo la struttura.

Domanda 4- Provare che 3-Sat è Np-Completo

Valutata 15/15:

Risposta:

(I valori sovraccennati sono le Y)

Dobbiamo dimostrare che 3-Sat è polinomialmente trasformabile nel problema SAT

Data una formula $f = x_1 + x_2 + \dots + x_k$ con $k \geq 4$ introducendo $k-3$ nuove variabili y , riscriviamo f come:

$f' = (x_1 + x_2 + y_1)(x_2 + \bar{y}_1 + y_2) \dots (x_{(k-2)} + y_{(k-4)} + y_{(k-3)}) (x_{(k-1)} + y_k + y_{(k-3)})$ diremo che f è soddisfattibile "se e solo se" f' è soddisfattibile

Dimostriamo la prima implicazione

Se la f è vera vuol dire che almeno uno degli x_i ha valore 1 e assegnando i valori x_j con $1 \leq j \leq k$ lasciamo il proprio valore di verità,

$$Y_j = 1 \text{ con } 1 \leq j \leq i-2$$

$$Y_j = 0 \text{ con } i-2 < j \leq k-3$$

In questo modo siamo sicuri che anche f' è soddisfattibile, cioè assume valore vero.

Dimostriamo la seconda implicazione

Se f' è soddisfattibile vuol dire che ogni fattore di f' assume valore 1, se consideriamo:

$y_1 = 0$ nel fattore $(x_1 + x_2 + y_1)$ avremo che o $x_1 = 1$ o $x_2 = 1$, visto che il fattore ha valore 1 e quindi anche f risulta vera.

$y_{(k-3)} = 1$ Nel fattore $(x_{(k-1)} + y_k + y_{(k-3)})$ il letterale compare complementato ma considerando che il fattore ha valore 1 vuol dire che o $x_{(k-1)} = 1$ o $x_k = 1$ e quindi anche f è verificata

Si può fare lo stesso ragionamento con $y_1 = 1$ e $Y_{(k-3)} = 0$

Domanda:

Si dia la definizione di relazione polinomiale fra due funzioni:

Risposta

Date due funzioni $f(n)$ e $g(n)$, diremo che $f(n)$ è in relazione polinomiale con $g(n)$ per tutti i valori di n , "se e solo se" esisteranno due polinomi $P_{1(x)}$ e $P_{2(x)}$ tali che

$$f(n) \leq P_1(g(n)) \text{ e } g(n) \leq P_2(f(n))$$

Equivalentemente, diremo che le funzioni $f(n)$ e $g(n)$ sono in relazione polinomiale

se la funzione $f(n)$ risulta limitata superiormente rispetto a un elemento polinomiale della funzione $g(n)$ e viceversa, la funzione $g(n)$ risulta limitata superiormente rispetto ad un elemento polinomiale della funzione $f(n)$.

Con $P_1(g(n))$ e $P_2(f(n))$ si intendono le sostituzioni di $f(n)$ e $g(n)$ all'incognita di $P_1(x)$ e $P_2(x)$.

Ad esempio date le funzioni $f(n) = \log(n)$ allora esisterà il polinomio $P_{(x)} = C1x^2 + C2x + C4$ ovvero $P(f(n)) = C1\log^2 n + C2\log(n) + C4$

Domanda:

Si illustri con un esempio la differenza fra criterio di costo uniforme e criterio di costo logaritmico per modelli di Calcolo RAM

Risposta:

Il criterio di costo uniforme ed il criterio di costo logaritmico sono due diverse metodologie per la valutazione della complessità di un algoritmo per il modello RAM.

Il criterio di costo uniforme associa a ciascuna istruzione una complessità pari ad una unità di tempo, trovando lo spazio di memoria necessario a contenere le informazioni, il tipo di operazione e la grandezza degli operandi delle operazioni.

Al contrario il criterio di costo logaritmico effettua uno studio mirato a fornire un costo per istruzione più vicino alla realtà.

Nel criterio di costo logaritmico, infatti, per ciascuna istruzione si considera il numero di bit necessari a rappresentare un operando, il tipo di operazione ed uno spazio dipendente dai registri occupati in memoria.

In particolare, secondo il criterio di costo logaritmico si definisce la seguente funzione (logaritmica) su interi.

$$\begin{aligned} L(i) &= 1 \text{ se } i = 0 \\ L(i) &= \log(|i|+1) \text{ se } i \neq 0 \end{aligned}$$

Tale funzione descrive la problematica secondo la quale per rappresentare un'informazione di n bit in memoria siano necessari $\log_2(n) + 1$ bits.

Inoltre, secondo il criterio di costo logaritmico, il costo di un'istruzione dipende dal tipo di istruzione e soprattutto dal tipo di indirizzamento utilizzato per l'esecuzione dell'istruzione. I tipi di indirizzamenti possibili sono : diretto(`<adress>`), indiretto (`*<adress>`) e immediato (`=<valore>`):

Tipo di indirizzamento	Costo
<code>=i</code>	$L(i)$
<code>i</code>	$L(i) + L(c(i))$
<code>*i</code>	$L(i) + L(c(i)) + L(c(c(i)))$

Domanda:

Definire un algoritmo RAM che calcoli 3^n e analizzare la complessità secondo il criterio di costo uniforme e il criterio di costo logaritmico.

Risposta:

Pseudo-codice

```
Begin
    read(n); ris=1;
    for i=1 to n do
        ris=ris*3;
    write (ris);
end
```

Codice RAM

READ1	$L(1)+L(n)=\log_2 n$
LOAD=1	$L(1)=\log_2 n$
STORE 2	$L(2)+L(c(0))=L(2)+L(1)=1$
LOAD1	$L(1)+L(n)=\log_2(n)$
5: JZERO 13	$L(1)=1$
LOAD2	$L(2)+L(c(2))=L(2)+L(3^n)$
MULT=3	$L(3)+Lc(0)=O(n \log_3)=\log_2 3^n=O(n \log_2 3)$
STORE 2	
STORE 2	
LOAD 1	
SUB =1	
STORE 1	
JUMP 5	
13: WRITE 2	
HALT	

$$O(n \log_2 3)$$

Costo uniforme:

$$4+6n+2=6n+6$$

Domanda:

Definire un algoritmo RAM che calcoli x^4 e analizzare la complessità secondo il criterio di costo uniforme e il criterio di costo logaritmico.

Risposta:

Pseudo-codice

```
begin
    redo(x)
    rego (y); ris =1;
    for i=1 to y do
        ris =x*x
    write (ris);
end
```

RAM

```
READ1 (read(x))
READ2 (read(y))
LOAD=1
STORE=3
LOAD 2
6 JZERO 14
LOAD 3
MULT 1
STORE 3
LOAD 2
SUB 1
STORE 2
JUMP 6
14 WRITE 3
HALT
```

Domanda:

Si indichi in quali circostanze il modello RAM è in relazione polinomiale con la Macchina di Touring

Risposta:

Sia L un linguaggio accettato da un programma RAM con complessità $L(n)$ calcolata mediante il modello a costo logaritmico. Se il programma RAM non prevede operazioni di moltiplicazione o divisione allora esiste una macchina di Touring multinastro che riconosce lo stesso linguaggio con complessità di tempo $L^2(n)$

Domanda:

Si considerino le funzioni $f(x)=x^k$ e $g(x)=k^x$ con k costante, stabilire se le funzioni sono in relazione polinomiale fra loro.

Risposta:

Le due funzioni prese in esame non sono in relazione polinomiale fra loro. Al variare di x infatti la funzione $f(x)=x^k$ cresce polinomialmente mentre la funzione $g(x)=k^x$ cresce esponenzialmente.

Non è pertanto possibile identificare due polinomi $P_1(x)$ e $P_2(x)$ tali che

$$f(x) \leq P_1(g_{(n)}) \quad \text{e} \quad g(x) \leq P_2(f(n))$$

Domanda:

Definire il costo logaritmico per RAM e valutare il costo dell'istruzione LOAD *<address>

Risposta:

Il criterio di costo logaritmico è una metologia per il calcolo della complessità di un algoritmo RAM.

Secondo tale criterio, per valutare la complessità di un istruzione bisogna considerare il tipo di istruzione, la grandezza degli operandi e il contenuto dei registri occupati in memoria.

Il criterio di costo logaritmico è basato infatti sull'assunzione che la complessità computazionale di un istruzione dipenda fortemente dalla lunghezza degli operandi delle istruzioni. Tale concetto è espresso mediante la seguente funzione logaritmica:

$$\begin{aligned} L(i) &= 1 \text{ se } i = 0 \\ L(i) &= \log(|i| + 1) \text{ se } i \neq 0 \end{aligned}$$

Tale funzione assume che per rappresentare/valutare l'intero i siano necessari esattamente $\log|i| + 1$ bits.

Il costo logaritmico, oltre che della lunghezza degli operandi, dipende fortemente anche dal tipo di indirizzamento. I possibili tipi di indirizzamento sono:

- Indirizzamento Diretto Istruzione <address>
- indirizzamento Indiretto Istruzione *<address>
- indirizzamento Immediato istruzione =<value>

In base al tipo di indirizzamento cambia la modalità del calcolo dei costi come segue:

Tipo di indirizzamento	Costo
=i	$L(i)$
i	$L(i) + L(c(i))$
*i	$L(i) + L(c(i)) + L(c(c(i)))$

Infine il calcolo del costo logaritmico dipende dal tipo di istruzione considerata:

$$\begin{aligned} \text{Load *}<\text{address}> &\rightarrow L(i) + L(c(i)) + L(c(c(i))) \\ \text{Load } *i & \end{aligned}$$

dove $L(i)$ identifica il costo per valutare l'indirizzo “e”, $L(c(i))$ identifica il costo per valutare il contenuto del registro i , che a sua volta è un indirizzo, e $L(c(c(i)))$ è il costo per valutare il contenuto del registro puntato dall'indirizzo contenuto nel registro i .

Domanda:

Si dia un vettore di interi positivi $V = \{v_1, v_2, \dots, v_n\}$ sia dia un algoritmo che per individuare la differenza massima fra due elementi dell'insieme V abbia complessità di tempo $O(n)$ e faccia al più $\frac{3}{2}n-2$ confronti

Risposta:

Considerato l'insieme V di n elementi, per determinare l'algoritmo richiesto è sufficiente analizzare il problema e ricordare che per effettuare le differenza massima fra 2 elementi dell'insieme basta individuare l'elemento massimo M e l'elemento minimo m dell'insieme stesso e calcolare M-m.

Esiste un algoritmo che chiameremo "Minmax" che applica il paradigma di calcolo del "Divide et impera" e viene eseguito con complessità $O(n)$ per $\frac{3}{2}n-2$ confronti. La

tecnica del "Divide et impera" consiste nel suddividere il problema originale in diversi sottoproblemi di taglia più piccola che verranno a loro volta suddivisi fino a giungere a tanti sottoproblemi risolvibili per mezzo di operazioni elementari.

Una volta risolti i sottoproblemi di taglia più piccola, il paradigma del Divide et impera prevede che queste soluzioni locali vengano fuse e utilizzate per risolvere i sottoproblemi di taglia più grande, fino a giungere alla risoluzione del problema di partenza.

Nel problema in esame, per individuare l'elemento massimo e l'elemento minimo tale insieme viene suddiviso in due parti della stessa dimensione, ognuna di queste parti verrà a sua volta suddivisa in 2 porzioni uguali fino a giungere ad insieme costituiti da 2 soli elementi, risolvibili per mezzo di un semplice confronto elementare. Una volta risolti i sottoproblemi più elementari, tali soluzioni verranno fuse per risolvere i sottoproblemi di taglia maggiore fino a risolvere il problema originario.

L'algoritmo Minmax precedentemente descritto viene qui presentato in pseudocodice:

```
(M,m)= MinMax(v)
{
    if |v| =2           // sia l'insieme V formato da 2 elementi V={a,b}
        return (Max(a,b), Min (a,b));
    else
        // divido l'insieme V in due sottoinsieme V1 e V2 della stessa dimensione
        (M1,m2)=MinMax(V1);
        (M2,m2)=MinMax(V2);
        return (Max(M1,M2),Min (m1,m2));
}
diff_massima= M-m;
```

Il tempo per eseguire tale procedure è proporzionale al numero dei confronti effettuati. Indicando con T(n) il numero di confronti effettuati su n elementi:

Relazione di ricorrenza

$$\begin{aligned} T(n) &= 1 \quad se \quad n=2 \\ T(n) &= 2T_{\frac{n}{2}} + 2 \quad se \quad n>2 \end{aligned}$$

Dimostriamo per induzione che tale relazione di ricorrenza ha risultato $\frac{3}{2}n-2$;

– per $n=2$ $T(2)=\frac{3}{2}2-2=1$ che è proprio la base della ricorrenza.

– Per un certo n suppongo vera: $T(m)=\frac{3}{2}m-2$ e dimostriamo $n>m$

$$T(n)=2T\left(\frac{n}{2}\right)+2=2\left(\frac{3}{2}\frac{n}{2}-2\right)+2=2\left(\frac{3}{4}n-2\right)=\frac{3}{2}n-4+2=\frac{3}{2}n-2$$

La complessità di tale algoritmo sarà $O(n)$ e vi saranno $\frac{3}{2}n-2$ confronti dove n è il numero di elementi dell'insieme

Domanda:

Fornire un algoritmo per la moltiplicazione di due interi binari ad n cifre

Risposta:

Dati due interi binari ad n cifre A e B

$$\begin{aligned}A &= a_{(n-1)} a_{(n-2)} \dots a_1 a_0 \\B &= b_{(n-1)} b_{(n-2)} \dots b_1 b_0\end{aligned}$$

L'algoritmo banale per calcolare il prodotto dei due numeri ha complessità $O(n^2)$ applicando il paradigma di calcolo del Divide et impera, tale algoritmo può essere reso più efficiente e ottenere una complessità $O(n^{1.59})$.

Il paradigma di calcolo del Divide et impera risolve il problema suddividendo in sottoproblemi di taglia più piccola che verranno a loro volta suddivisi ricorsivamente fino a giungere a sottoproblemi di taglia piccola risolvibili per mezzo di operazioni elementari (Fase di Divide). Una volta risolti i sottoproblemi "elementari" questo paradigma ne prevede un'opportuna fusione per risolvere sottoproblemi di taglia maggiore, operando ricorsivamente fino a giungere ad una soluzione per il problema originario (Fase di Impera)

Per la moltiplicazione di due numeri interi binari A e B, si partirà col suddividere tali numeri di n cifre in due parti da $\frac{n}{2}$ cifre, come segue:

$$\begin{aligned}A &= A_0 + A_1 2^{\left(\frac{n}{2}\right)} \\B &= B_0 + B_1 2^{\left(\frac{n}{2}\right)}\end{aligned}\quad \text{Operazioni di Shift}$$

Seguendo tale ragionamento è possibile scrivere il prodotto fra due numeri come segue:

$$A * B = (A_0 + A_1 2^{\left(\frac{n}{2}\right)}) * (B_0 + B_1 2^{\left(\frac{n}{2}\right)}) = (A_0 * B_0) + (A_0 * B_1) 2^{\left(\frac{n}{2}\right)} + (A_1 * B_0) 2^{\left(\frac{n}{2}\right)} + (A_1 * B_1) 2^{(n)}$$

Abbiamo così ottenuto che il prodotto fra 2 numeri ad n cifre è paragonabile a 4 prodotti di numeri a $\frac{n}{2}$ cifre più operazioni di somma e shift

$$T(n) = b \text{ se } n=1$$

$$T(n) = 4T\left(\frac{n}{2}\right) + bn \text{ se } n>1$$

$$T(n) = O(n^{(\log_c a)}) = O(n^{(\log_2 4)}) = O(n^2)$$

Applicando il paradigma del “Divide et impera” come descritto ci rendiamo però conto di ottenere un algoritmo con la stessa complessità dell’algoritmo banale pari a $O(n^2)$.

Effettuando però gli opportuni accorgimenti ci rendiamo conto di poter scrivere diversamente il precedente prodotto.

$$A * B = (A_0 * B_0) + (A_0 * B_1)2^{\left(\frac{n}{2}\right)} + (A_1 * B_0)2^{\left(\frac{n}{2}\right)} + (A_1 * B_1)2^n =$$

$$(A_0 * B_0) + 2^{\left(\frac{n}{2}\right)}((A_0 * B_1)(A_1 * B_0)) + ((A_1 * B_1)2^n)$$

Considero il prodotto:

$$(A_0 + A_1) * (B_0 + B_1) = (A_0 * B_0) + (A_0 * B_1) + (A_1 * B_0) + (A_1 * B_1)$$

Possiamo allora scrivere:

$$(A_0 * B_1) + (A_1 * B_1) = (A_0 + A_1) * (B_0 + B_1) - (A_0 * B_0) - (A_1 * B_1)$$

Questo suggerisce che il prodotto fra i numeri A e B possa essere riscritto, diminuendo il

numero di prodotto fra numeri di $\frac{n}{2}$ cifre, come segue:

$$A * B = (A_0 * B_0) + 2^{\left(\frac{n}{2}\right)} * ((A_0 + A_1) * (B_0 + B_1) - (A_0 * B_0) - (A_1 * B_1)) + (A_1 * B_1)2^{\left(\frac{n}{2}\right)}$$

Questo ci porta a calcolare il prodotto fra 2 numeri ad n cifre tramite 3 prodotti fra numeri di $\frac{n}{2}$ cifre $((A_0 * B_0), (A_0 + A_1) * (B_0 + B_1), (A_1 * B_1))$ più operazioni di somme e differenze e operazioni di shift.

$$T(n) = b \text{ se } n=1$$

$$T(n) = 3T\left(\frac{n}{2}\right) + 2 \text{ se } n>1$$

$$T(n) = O(n^{(\log_c a)}) = O(n^{(\log_2 c)}) = O(n^{1,59})$$

Di conseguenza, trovato il paradigma di calcolo di divide et impera siamo riusciti ad ottenere un algoritmo per il prodotto di 2 numeri interi binari di n cifre con complessità di $O(n^{1,59})$ migliore dell’algoritmo banale risolvibile in $O(n^2)$

Domanda: Si dia un algoritmo di ordinamento basato sulla tecnica delle partizioni bilanciate e si analizzi la complessità nel caso pessimo.

Il nostro algoritmo è attuale rispetto al modello di calcolo RAM?

Risposta: un esempio di algoritmo di sorting(ordinamento) basato sulla tecnica delle partizioni bilanciate è il Mergesort

la tecnica del bilanciamento è un caso particolare del paradigma di calcolo “divide ad coquer”. Tale paradigma permette di individuare la soluzione di un problema subdividendo lo stesso in più sottoproblemi di taglia più piccole e risolvendo questi ricorsivamente per mezzo di operazioni elementari(fase et divide)

Le soluzioni dei sottoproblemi di taglia unica vengono poi appositamente fuse per risolvere i sottoproblemi di taglia maggiore e, operando ricorsivamente secondo questa

metodologia, giunge facilmente alla risoluzione del problema.

La tecnica del bilanciamento viene utilizzata quando il problema, e poi per i sottoproblema, viene suddiviso in i sottoproblemi di uguale dimensione, dividendo la taglia del problema di partenza in due parti uguali.

Di seguito sono proposti gli algoritmi per l'ordinamento del MERGESORT

procedure mergesort(A,i,f)

```
{  
    if i<f  
    {  
        m=(i+f)/2;  
        mergesort(A,i,m);  
        mergesort(A,m+1,f);  
        marge(A,i,m,f);  
    }  
}
```

procedure merge(A,i_1,f_1,f_2)
{sia per un vettore di taglia f_2-i_1+1

```
i_1=0  
i_2=f_1+1  
while(i_1<=f_1 and i_2<=f_2)
```

{if(A[i_1]<=A[i_2])

```
{  
    x[i]=A[i_1]  
    i1++  
    i2++  
}
```

else{

```
    x[i]=A[i_2]  
    i1++  
    i2++}  
}
```

```
}
```

la complessità dell'algoritmo è definita della seguente relazione di ricorrenza:

$T(n)=b$ se $n=1$

$2T(n/2)+2n$ se $n>1$

la soluzione di tale relazione di ricorrenza è: $T(n)=O(n\log n)$ nel caso migliore, medio e peggiore.

E' importante ricordare che tale complessità è un LOWERBOUND(limite inferiore)

per gli algoritmi di ordinamento basati su confronti questo vuol dire che non è possibile, per questa tipologia di algoritmi, ottenere una complessità più bassa.

L'algoritmo del Mergesort descritto non è attico rispetto al modello di calcolo RAM, poiché esistono algoritmi RAM che riescono a riordinare n elementi in ordine di $O(n\log n/\log \log n)$ e $O(n\log n/\log \log n) < O(n\log n)$

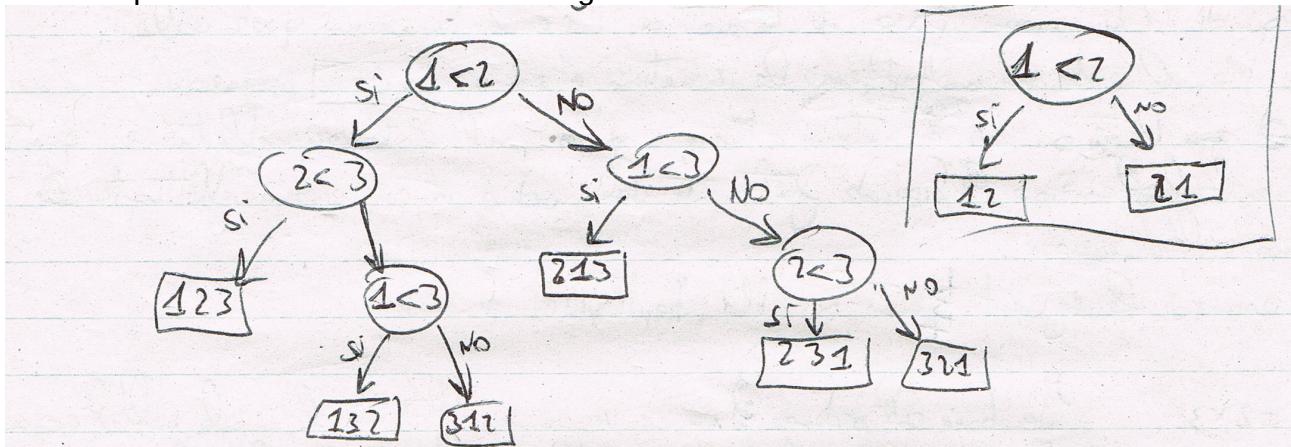
Nel modello di calcolo dato da alberi di decisione, si provi che ordinare n numeri richiede $\Omega(n \log n)$ confronti

un albero di decisione è un particolare modello di calcolo adoperato per la risoluzione del problema dell'ordinamento di n numeri.

Le strutture che meglio si presta all'implementazione di un albero di decisione è quella di un albero binario.

La caratteristica principale di un albero di decisione consiste nel fatto che l'algoritmo risolutivo è costituito da una sequenza di decisione ove la prima decisione è effettuata sulla radice e, in base all'esito della prima e di ogni decisione, si percorrerà l'albero fino a giungere ad una foglia. Ciascuna decisione può essere vista come la risposta alla domanda $i < j$? Dove i e j sono due generici elementi dell'insieme di ordine.

Un esempio di albero di decisione è il seguente



come è possibile notare, ciascuna foglia rappresenta una permutazione degli n elementi presi in considerazione e tutte le permutazioni vengono prese in considerazione. L'output è il contenuto di una foglia. Questo ci fa presumere che le foglie siamo in numero di $n!$

Nel caso migliore, ovvero quando l'albero considerato è bilanciato, allora l'altezza h dell'albero è un logaritmo del numero di foglie, ovvero $h = \log n!$ Che, data l'approssimazione di strologo $n! \approx n^n$, diventa $h = n \log n$.

Considerato che la complessità dell'algoritmo di ordinamento è data dal numero di decisione prese per giungere dalla radice fino ad una foglia, proprio all'altezza h dell'albero, avremo che la complessità dell'algoritmo di ordinamento basato su alberi di decisione $h = \Omega(n \log n)$

Date n matrici $M_1 \dots M_n$ tali che M_i e M_{i+1} sono compatibili per il prodotto è dia un algoritmo polinomiale che calcola il modo ottimo di fare il prodotto $M_1 \times M_2 \times \dots \times M_n$

Matrici compatibili: date due matrici M_1 e M_2 , è possibile effettuare il prodotto soltanto se queste sono "compatibili", ovvero se il numero di colonne della matrice M_1 sarà pari al numero delle righe della matrice M_2

date quindi le dimensioni $p \times q$ della matrice M_1 e le dimensioni $q \times r$ della matrice M_2 il costo per moltiplicare le 2 matrici è pari a $p \times q \times r$ operazioni.

Date quindi n matrici M_1, \dots, M_n , il costo per effettuare questa moltiplicazione dipende fortemente dall'ordine in cui vengono effettuate le i -esime moltiplicazioni.

Data ad esempio 4 matrici M_1, M_2, M_3, M_4 di dimensione:

$$M_1 = 2 \times 3$$

$$M_2 = 3 \times 4$$

$$M_3 = 4 \times 5$$

In base all'ordine in cui vengono effettuate le singole moltiplicazioni

fra coppie di matrice cambierà il costo finale in termini di operazione

M4=5x2

$$(M1 \times M2) \times (M3 \times M4) \Rightarrow (2 \times 3 \times 4) \times (4 \times 5 \times 2) = 24 \times 40 = 96$$

$$M1 \times (M2 \times (M3 \times M4)) \Rightarrow (2 \times (3 \times (4 \times 5 \times 2)) = 2 \times (3 \times (40)) = 24$$

questo piccolo esempio dimostra come una diversa paranteizzazione delle n matrici da moltiplicare possa incidere parecchio sul costo dell'operazione.

L'obiettivo è quello di determinare il modo ottimale per effettuare il prodotto delle n matrici ovvero trovare la migliore parentarizzazione che quindi ci permette di minimizzare il numero di operazioni eseguite.

Per risolvere tale problema viene adoperato il paradigma di calcolo noto come paradigma di "programmazione dinamica".

La programmazione dinamica consiste sostanzialmente nel suddividere iò problema principale in più sottoproblemi di taglia più piccole, iniziando a risolvere tale sottoproblema di taglia minima, risalendo per risolvere il problema di taglia più grande secondo ma metodologia di risoluzione bottom up.

Caratteristica principale di tale paradigma è costituita dal concetto di "tabulazione", che consiste nel conservare, in una struttura dati ausiliare come ad esempio una matrice la soluzione dei sottoproblemi così da farvi riferimento in tempo costante nel momento in cui durante il procedimento di risoluzione dell'algoritmo ci ripresenta un sottoproblema già risolto. Per meglio descrivere l'algoritmo di risoluzione utilizzata si ritiene necessario esprire i punti fondamentali del paradigma di programmazione dinamica.

Passi generici programmazione dinamica

1. si identificano i sottoproblemi del problema originale, utilizzando una tabella per mantenere le soluzioni di ciascun sottoproblema incontrato
2. si definiscono i valori iniziali di determinati sottoproblemi, risolti per mezzo di operazioni elementari, conservando tali soluzioni appositamente nella tabella
3. si effettua il primo generico del procedimento risolutivo, risolvendo determinati sottoproblemi in funzione dei risultati dei sottoproblemi precedentemente risolti
4. si restituisce la soluzione del problema originario, opportunamente memorizzate come elemento della tabella.

Per il problema del prodotto di m matrici preso in esame, possiamo determinare il generico sottoproblema M_{ij} con $i \leq j$ di determinare il minor costo per moltiplicazione le matrici $M_i \dots M_j$. Il costo di tale moltiplicazione verrà indicato con $M[i,j]$ inserito in una matrice nella posizione i

i sottoproblemi elementari si hanno considerando il prodotto m_{ij} , che consiste nella sola matrice M_i . In questo caso non sarà prevista alcuna moltiplicazione e il costo sarà $m[i,i] = 0$

per definire il passo generico, bisogna prendere in considerazione il seguente ragionamento:

considerati $i < j$ il costo per calcolare il prodotto M_i, M_{i+1}, \dots, M_j può essere visto come il costo dato un generico K con $i \geq K \leq j$, per calcolare il prodotto $(M_i \dots \times M_k) \times (M_{k+1} \dots \times M_j)$.

Ma questo si trasforma analiticamente in:

$$m[i,j] = \min(i \leq k \leq j) \{ m[i,k] + m[m+1,j] + p_{i-1} * p_k * p_j \}$$

proprio perché k com è costo a priori, l'obiettivo è quello di individuarlo per ciascun sottoproblema, così da risolvere il costo del prodotto $(M_i \dots \times M_k) \times (M_{k+1} \dots \times M_j)$

la funzione costo sarà pertanto definita in generale come segue:

$$m[i,j] = \{ 0 \text{ se } i=j \}$$

$$\min(i \leq k \leq j) \{ m[i,k] + m[m+1,j] + p_{i-1} * p_k * p_j \} \text{ se } i < j$$

come è possibile notare grazie al passo ricorsivo è possibile determinare la soluzione di un sottoproblema in funzione soluzioni di certi sottoproblemi opportunamente memorizzate nella tabella.

L'algoritmo preso in considerazione utilizzerà due matrici, la matrice $m[i,j]$, per conservare il costo minimo di moltiplicazione delle matrici $M_1 \times \dots \times M_j$, e $s[i,j]$ per conservare il valore K che ha permesso, per ciascun sottoproblema, di raggiungere la soluzione ottima

Matrice(P)

```
{
    m=lengthp-1
    for i=1 to n do
        m[i,i]=0
    for l=2 to n do
        for i=1 to n-l+1 do {
            j=i+l-1
            m[i,j]=inf
            for k=1 to j-1 do
            {
                q=m[i,j] +m[k+1,j] + pi-1*pk*pj
                if (q<m[i,j]) then {
                    m[i,j]=p
                    s[i,j]=k}
            }
        }
}
```

Domanda:

dato un nastro di dimensione L e siano p_1, \dots, p_n programmi di dimensione l_1, \dots, l_n tali che $\sum l_i \leq L$. Si dia un algoritmo che trovi le sequenze p_1, \dots, p_n di programmi da memorizzare sul nastro modo che il tempo medio di lettura di tutti i programmi sia minimo p

Risposta:

la soluzione ottima per memorizzare gli n programmi consiste nel memorizzare in ordine con decremento di dimensione.

Il problema consiste nel determinare il modo ottimo per memorizzare p_1, \dots, p_n programmi di dimensione rispettivamente l_1, \dots, l_n in un nastro di dimensione L .

dato un generico programma p_{ij} appartenente ad una sequenza di programmi p_1, \dots, p_n sul nastro, il costo per leggere tale programma sarà dipendente dalla posizione in cui esso è memorizzato come segue:

$$cost(P_{ij}) = \sum_{k=1}^j l_{ik} \quad f_j = \sum_{k=1}^j l_{ik}$$

questo dice che per accedere al generico programma i andranno da prima letti $i-1$ programmi che loro precedono.

Si definisce costo medio di accesso ad un programma la sommatoria:

$$\frac{1}{n} \sum_{j=1}^n f_j = \frac{1}{n} \sum_{j=1}^n \left(\sum_{k=1}^j l_{ik} \right)$$

per risolvere tale problema viene adoperato il paradigma di calcolo Greedy. Tale paradigma viene adoperato per risolvere problemi di ottimizzazione in cui risultati necessario minimizzare o massimizzare una certa "funzione obiettivo" effettuando una sequenza di salto.

Secondo questo paradigma, la scelta da effettuare risulterà localmente ottima partendo dal

concetto che scegliendo ad ogni passo gli ottimi locali si otterrà la selezione ottima assoluta.

In questo contesto si distinguerà la soluzione ammissibile, ovvero una possibile soluzione di un generico passo che rispetta i vincoli imposti del problema, e la SOLUZIONE OTTIMA, ovvero quella soluzione salta per mezzo del metodo Greedy che permetta di minimizzare la funzione obiettivo:

Visto che nel nostro caso la funzione obiettivo da minimizzare sarà:

$$\frac{1}{h} \sum_{j=1}^n t_j = \frac{1}{n} \sum_{j=1}^n \left(\sum_{k=1}^j l_{ik} \right) \quad \text{Funzione obiettivo}$$

L'obiettivo risulta essere quello di determinare una permutazione $i=(i_1, i_2, \dots, i_n)$ tale che, memorizzando i programmi sul nastro secondo tale permutazione, risulti minimo il tempo medio di recupero (MEAN RETRIVAL TIME). Consideriamo il costo globale $D(i)$

$$D(i) = \sum_{j=1}^n t_j$$

$$t_1 = l_{i1}$$

$$t_2 = l_{i1} + l_{i2}$$

$$t_3 = l_{i1} + l_{i2} + l_{i3}$$

$$t_k = l_{i1} + \dots + l_{ik}$$

$$t_n = l_{i1} + \dots + l_{in}$$

La soluzione ottima del problema consiste nell'ordinare gli n programmi in ordine non decrescente di dimensione tali che i programmi p_1, p_2, \dots, p_n vengono memorizzate sul nastro secondo la permutazione i tale che :

$$l_{i1} \leq l_{i2} \leq \dots \leq l_{in}$$

L'algoritmo ad ogni passo sceglie secondo metodo greedy la soluzione ottima locale dall'insieme di programmi non ancora considerati ovvero scegli quello con dimensione minima.

E' possibile dimostrare che tale permutazione Π è la soluzione ottima

$$\text{dato } D(\Pi) = \sum_{j=1}^n t_j \quad t_j = \sum_{k=1}^j l_{ik} \quad t_1 = l_{i1} \quad t_2 = l_{i1} + l_{i2}$$

$$D(\Pi) = \sum_{j=1}^n t_j = \sum_{j=1}^n \left(\sum_{k=1}^j l_{ik} \right)$$

$$t_k = l_{i1} + l_{i2} + \dots + l_{ik}$$

$$t_n = l_{i1} + l_{i2} + \dots + l_{ik} + \dots + l_{in}$$

$$D(\Pi) = \sum_{j=1}^n t_j = n * l_{i1} + (n-1) * l_{i2} + \dots + 2 * l_{in-1} + l_{in} = \sum_{k=1}^n (n-k+1) l_{ik}$$

consideriamo 2 programmi a e b dalle permutazione tali che $a < b \quad l(a) > l(b)$, dimostrano che la soluzione ottimale è quella della permutazione

$$i_1 i_2 \dots i_a \dots i_b > i_1 i_2 \dots i_b \dots i_a \dots i_n$$

$$D(\Pi') > D(\Pi)$$

$$D(\Pi) = \sum_{k=1}^n (n-k+1) l_{ik}$$

$$D(\Pi') = \left(\sum_{(k=1 \ k \neq a \ k \neq b)} (n-k+1) l_{ik} \right) + (n-a+1) l_{ia}$$

Consideriamo la differenza $D(\Pi) - D(\Pi')$ otteniamo un valore < 0 , questo vuol dire che

$$D(\Pi) - D(\Pi') = (n-a+1) l_{ia} + (n-b+1) l_{ib} - (n-a+1) l_{ib} - (n-b+1) l_{ia} = (n-a+1+b-1) l_{ia}$$

$$= (b-a) l_{ia} + (a-b) l_{ib} = (b-a) l_{ia} - (b-a) l_{ib} = (b-a)(l_{ia} - l_{ib}) < 0$$

Domanda: Trovare la soluzione attuale (versione greedy) del problema di uno zaino di capacità $M=80$, dati 8 oggetti di valore $P=\{18,5,6,2,9,14,36,24\}$ e dimensione $W=\{9,15,12,18,36,14,4,8\}$. Motivare.

Questo esercizio fa riferimento al problema dello zaino risolto mediante il paradigma di calcolo noto come "Metodo di Greedy".

Secondo il problema dello zaino, si hanno a disposizione n oggetti e secondo certi vincoli, vanno inseriti in uno zaino di capienza M.

Preso in considerazione il generico i-esimo oggetto, a quanto vengono associati z valori:
 - P_i , pari al profitto ottenuto inserendo l'oggetto i nello zaino;
 - W_i , pari al peso dell'i-simo oggetto.

Naturalmente, gli n oggetti potrebbero essere tutti inseriti nello zaino se la somma dei loro pesi risultasse inferiore alla capacità dello zaino, ma secondo problema preso in esame è possibile inserire tutti gli n oggetti.

Ogni qualvolta un oggetto i viene inserito nello zaino il profitto aumenta di una quantità pari a $P_i X_i$, dove X_i indica la frazione dell'oggetto uscente e $0 \leq X_i \leq 1$.

E' importante ricordare che è possibile frazionare soltanto l'ultimo degli oggetti inseriti nello zaino, al fine di riempire e occupare tutta la capienza dello stesso.

In generale, inseriti n oggetti nello zaino, una soluzione ottima è tale da:

- massimizzare la funzione obiettivo $\rightarrow \sum_{1 \leq i \leq n} P_i X_i$ Funzione obiettivo
- Soggetta al seguente vincolo $\rightarrow \sum_{1 \leq i \leq n} W_i X_i \leq M$ con $0 \leq X_i \leq 1$, $W_i > 0$ $P_i > 0$

Per risolvere il problema si possono seguire delle alternative, tutte facenti riferimento al Metodo di Greedy.

Il metodo di Greedy è un paradigma di calcolo di risoluzione di problemi di ottimizzazione, dove la soluzione che risulta ottimale dovrà massimizzare (o minimizzare) una certa funzione, detta FUNZIONE OBIETTIVO, sarà sottoposta a certi vincoli (nel nostro caso la funzione da massimizzare è la sommatoria dei profitti e il vincolo che non si superi la capacità M dello zaino).

Caratteristica fondamentale del Metodo di Greedy è quella di fornire un algoritmo che lavora su livelli, considerando gli elementi di input (nel nostro caso gli n oggetti) uno alla volta.

Secondo tale paradigma, infatti, per individuare la soluzione ottima si prevede di definire, passo dopo passo, una sequenza di decisioni, dove una decisione consiste nello scegliere quale elemento dell' input, fra gli elementi non ancora presi in considerazione, aggiungere alla soluzione.

L'idea è quella di identificare la soluzione ottima dalle soluzioni ottima "locali" , una per ciascuna decisione.

Per il Problema dello zaino, esistono 3 possibili applicazioni del metodo di Greedy.

- 1) effettuare la scelta sugli oggetti da inserire minimizzando in funzione del profitto. In questo modo andranno inseriti per primi gli oggetti con maggiore profitto. Questo tipo di applicazione non porta sempre alla soluzione ottimale in quanto a causa dei pesi degli oggetti, si corre il rischio di raggiungere fin dall'inizio la capienza dello zaino
- 2) Un'altra possibile alternativa prevedere di scegliere ad ogni passo, l'oggetto con il minor peso fra gli oggetti non inseriti. Come è possibile dimostrare, anche in questo caso non siamo sicuri di raggiungere la soluzione attuale, proprio perché il fatto di rallentare l'aumento del peso non si riflette sull'aumento del profitto.
- 3) L'ultima alternativa prevede di considerare, per ciascuno oggetto il rapporto $\frac{P_i}{W_i}$ e inserire uno per volta gli oggetti in ordine non crescente del rapporto considerato. Esiste un teorema che afferma che applicando tale metodo si ottiene sempre una soluzione ottimale.

Provvediamo allora ad utilizzare il metodo appena descritto per gli n oggetti del problema in esame.

$$M=80$$

$$P=\{18,5,6,2,9,14,36,24\}$$

$$W=\{9,15,12,18,36,14,4,8\}$$

$$\frac{P}{W} = \left\{ \frac{18}{2}, \frac{5}{15}, \frac{6}{12}, \frac{2}{18}, \frac{9}{36}, \frac{14}{14}, \frac{36}{4}, \frac{24}{8} \right\} = \{2, 0.33, 0.5, 0.11, 0.25, 1, 9, 3\} =$$

ordiniamo questo insieme ottenuto in ordine non crescente:

$$=\{9, 3, 2, 1, 0.5, 0.33, 0.25, 0.11\}$$

Determiniamo in base al peso W_i , quanti elementi sia possibile inserire nello zaino secondo l'ordine individuato:

$$\sum_{i \leq i \leq n} W_i X_i = W_7 * 1 + W_8 * 1 + W_1 * 1 + W_6 * 1 + W_3 * 1 + W_2 * 1 + W_5 * 0.5 = \\ 4+8+9+14+12+15+18=80$$

Calcoliamo il valore della soluzione ottima trovata:

$$\sum_{1 \leq i \leq n} P_i X_i = P_7 * 1 + P_8 * 1 + P_1 * 1 + P_3 * 1 + P_2 * 1 * P_5 * 0.5 = 36+24+18+14+6+5+4.5=107.5$$

Per il teorema precedentemente descritto, siamo sicuri che questa sia la soluzione ottima

Domanda: Si descriva l'argoritmo di Union e Find che utilizza strutture dati ad albero e lo si analizzi. La complessità di tempo deve essere $O_{(1)}$ e quella di Finda deve essere $O_{(\log n)}$ dove n è il numero massimo di elementi dell'universo U.

Dato un insieme universo U limitato, ad esempio costituito dai primi n numeri naturali $U=\{1,2,\dots,n\}$

Suddividiamo inizialmente l'insieme U in n insieme disgiunti, ognuno contenente un unico elemento.

(*) $S_1=(1)$, $S_2=(2), \dots, S_n=(n)$

Data una situazione del genere è possibile definire 2 tipi di operazioni, la Union e la Find
(*) Per ciascun di questi insieme viene scelto un elemento dell'insieme stesso detto "rappresentante dell'insieme". Possiamo rappresentare ogni insieme disgiunto come un albero avente come radice il rappresentante



Su tale organizzazione logica dei dati possibile definire le seguenti due operazioni:

- Find(x), restituisce il nome dell'insieme S_i tale che $x \in S_i$, restituisce quindi il rappresentante di S_i
- Union (S_i, S_j) , dati due insiemi disgiunti S_i e S_j , restituisce l'unione $S_i \cup S_j$. Si tratta di un'operazione distruttiva in quanto si perde traccia degli insiemi S_i e S_j singoli ma si ottiene solo l'unione.

Possiamo implementare questi insiemi disgiunti mediante alberi il cui grado non è definito a priori e la cui radice funge da rappresentante per l'intero insieme.

OPERAZIONE DI UNION (tempo costante $O_{(1)}$):

Supponiamo di avere 2 insiemi disgiunti, S_i e S_j , rappresentati da alberi T_1 e T_2 di altezza rispettivamente h_1 e h_2 , è possibile definire l'operazione di Union in tempo costante effettuando essenzialmente un collegamento fra la radice di uno di due alberi e la radice dell'altro albero.

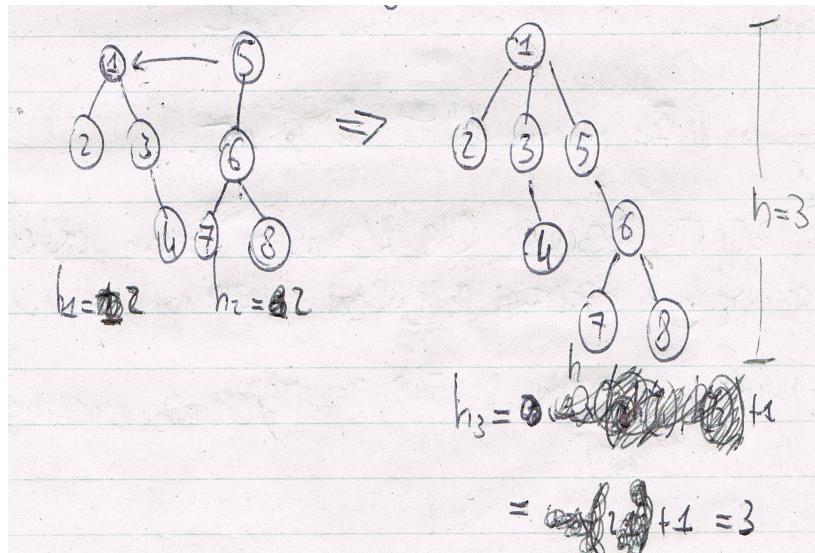
In questo modo una radice diverrà figlia dell'altra.

L'albero Union $S_i \cup S_j$ risultante dell'operazione dipende dall'altezza degli insiemi di partenza h_1 ed h_2 . Infatti:

- Se $h_1 = h_2$, allora effettuando una Union fra h_1 e h_2 si otterrà sempre un albero S_3 di altezza pari a $h+1$ dove $h = h_1 + h_2$:

$$h(T_3) = \max(h(T_1), h(T_2))$$

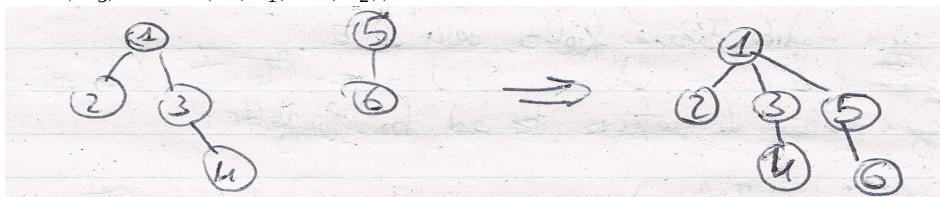
Ad esempio dati i 2 alberi seguenti:



- Se $h_1 \neq h_2$, in questo caso si hanno 2 alternative, collegare l'albero con altezza minore in comune tale che la sua radice divenga figlia della radice dell'albero con altezza maggiore, o viceversa, collegare la radice dell'albero con altezza maggiore come figlia della radice dell'albero con altezza minore.

Si dimostra che nel primo caso si ottiene sempre un albero bilanciato T_3 con altezza $h(T_3)$ pari all'altezza massima fra i sue alberi T_1 e T_2

$$h(T_3) = \max(h(T_1), h(T_2))$$



$$h(T_1) = 2 \quad h(T_2) = 1 \Rightarrow h(T_3) = \max(h(T_1), h(T_2)) = \max(2, 1) = 2$$

Nel secondo caso si ottiene $h_3 = \max(h(T_1), h(T_2)) + 1$ e albero non bilanciato.

In termini di complessità, il costo per effettuata in ogni caso l'operazione di Union è $O_{(1)}$, ma questa operazione viene spesso adoperata in seguito all'esecuzione di 2 operazioni di Find per trovare gli insiemi S_i e S_j .

L'operazione di Union può essere effettuata solo se $i \neq j$.

E' importante ricordare che al massimo possono essere eseguite $(n-1)$ operazioni di union mentre non vi sono vincoli sul numero di operazioni di Find eseguite, quindi l'operazione di Find ha la priorità.

Operazione Find(x) (Complessità di tempo $O_{(\log n)}$)

Abbiamo visto che tramite l'operazione di Union è possibile definire in tempo costante alberi bilanciati.

L'operazione di Find consiste nel ricercare x in un centro albero bilanciato e, se trovato il nodo x, restituisce il rappresentante dell'albero, ovvero la radice, Questo vuol dire che, a partire dal nodo x prescelto dell'albero, l'operazione di Find

consiste nel risalire l'albero fino alla radice.

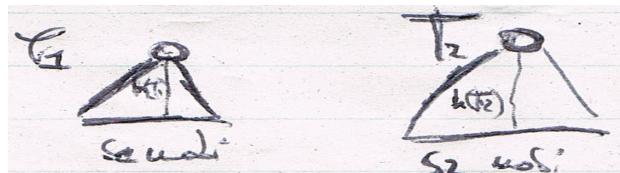
La complessità di tale operazione dipende dall'altezza dell'albero bilanciato, che dimostriamo essere minore o uguale al logaritmo del numero di nodi.

Dato un albero bilanciato $T_{(k-ario)}$, l'altezza di questo albero è pari a $h(T) \leq \log_2(nT)$

Dimostriamolo per induzione sul numero dei nodi

- per $h=0$ si ha un solo elemento rappresentato della radice, quindi $0 \leq \log_2(1)=0$

- Supponiamo tale teorema sia vero per ogni albero avente $S' < S$ nodi e dimostriamolo per un albero con S nodi. Consideriamo 2 alberi T_1 e T_2 aventi S_1 e S_2 nodi e dimostriamo che la loro Unione da un albero = $S_1 + S_2$ nodi per cui voglio avere il teorema:



Il teorema va dimostrato sia che $h(T_1) = h(T_2)$, sia che $h(T_1) < h(T_2)$ (o viceversa)

- $h(T_1) < h(T_2)$ Allora:

$$h(T_3) = h(T_2) \leq \log_2 S_2 \leq \log_2(S_1 + S_2)$$

ecco dimostrato che $h(T_1) = h(T_2)$

- $h(T_1) = h(T_2)$

In questo caso, ricordiamo in generale che:

$$h(T) \leq \log_2 n \Rightarrow \log_2 2^{h(T)} \leq \log_2 n \log_2 n \Rightarrow 2^{(h(T))} \leq n$$

Nel nostro caso avremo che:

$$2^{(h(T_1))} \leq S_1 \Rightarrow 2^{(h(T_2))} \leq S_1$$

$$2^{(h(T_2))} \leq S_2 \Rightarrow 2^{(h(T_1))} \leq S_2 \quad \text{da qui segue } 2^{(h(T_1))} \leq S_1 + S_2$$

Tornando al logaritmico:

$$\log_2[h(T_1)+1] \leq \log_2(S_1 + S_2)$$

$$[h(T_1)+1]\log_2 2 \leq \log_2(S_1 + S_2)$$

$$h(T_1)+1 \leq \log_2(S_1 + S_2)$$

$$h(T_3) \leq \log_2(S_1 + S_2)$$

Domanda: Albero Binario di Ricerca; si definisca un albero binario di ricerca e si diano le procedure di Member, Insert e Delete. Si analizzino tali procedure. Si dia poi una procedura per la costruzione di un albero binario di ricerca e si analizzi anche quella.

Un albero binario di ricerca (ABR) è una particolare struttura dati utilizzata come implementazione fisica di struttura dati astratta, come ad esempio il "Dizionario".

Una struttura dati astratta può essere intesa come un organizzazione logica su cui sono definite ben precise operazioni, nel caso di dizionari esempi di operazioni sono MERBER(X), INSERT(X), DELETE(X), ed è compito dell'ABR fornire un'implementazione

di tali operazioni.

FORMALMENTE: una struttura dati T è un albero binario di ricerca se e solo se la visita in ordine simmetrico restituisce una lista ordinata di elementi

Strutturalmente , sia S un insieme, è possibile definire un ABR nel seguente modo:

- ogni vertice v dell'albero ABR è etichettato con etichette $l(v)$ tale che $l(v) \in S$
- per ogni vertice u del sottoalbero sinistro di v si ha che $l(u) < l(v)$
- per ogni vertice w del sottoalbero destro di v si ha che $l(w) > l(v)$
- con una visita in ordine simmetrico dell'ABR otteniamo tutte le chiavi dell'albero ordinate in ordine simmetrico

Analizziamo le operazioni MEMBER(X), INSERT(X) e DELETE(X) per un albero binario di ricerca.

* Member: dato un albero binario di ricerca e un elemento a, l'operazione MEMBER deve stabilire se l'elemento appartiene o meno all'ABR sfruttando l'ordine degli elementi nell'ABR stesso. In particolare, l'idea algoritmica sarà la seguente:

- se $a=r$ allora si restituisce la risposta $a \in S$
- se $a < r$ allora si riscrive "a" nel sottoalbero sinistro con radice r
- se $a > r$ allora si riscrive "a" nel sottoalbero destro con radice r

La procedura che implementa l'operazione di MEMBER è la procedura SEARCH:

```
PROCEDURE SEARCH (a,r)
{
    if(a=l(r)) then return ( a ∈ S );
    else
        if(a < l(r)) then
            if esiste un sottoalbero sinistro di r con radice v
                search(a,v);
            else return (a ∉ S) ;
        else
            if esiste un sottoalbero destro di r con radice w
                search(a,w);
            else return (a ∉ S)
}
```

la procedura partirà confrontando l'elemento a da ricercare con l'etichetta della radice, se questi valori saranno uguali allora l'elemento apparterrà all'insieme S rappresentato dall'albero, altrimenti in base al confronto si ricercherà il valore "a" in uno dei sottoalberi, se esistono. Tale operazione verrà effettuata applicando la procedura search ricorsivamente sulle radici dei sottoalberi fino a giungere alle foglie.

La complessità di tempo sarà quindi = (h) con h altezza dell'albero (si ricordi che l'ABR non è un sottoalbero sempre bilanciato.)

--Insert

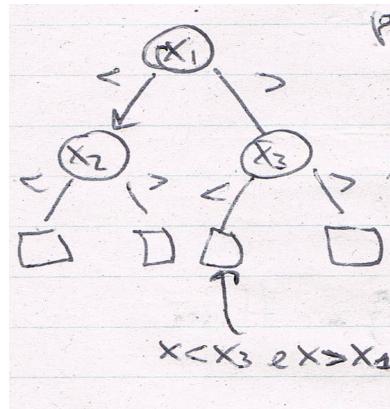
Le operazioni più complesse di un Abr sono le operazioni in cui si va a indicare l'insieme dielementi (Insert e Delete) poiché vi è il vincolo dell'ordinamento degli elementi da rispettare.

Per inserire un nuovo elemento si creano 2 foglie fintizie come figli per ciascuna foglia dell'Abr. Ciascuna foglia fittizia identifica un range di valori e viene preso in considerazione quando si vuole inserire nell'abr un nuovo elemento compreso fra quei valori.

Una volta individuata la foglia fittizia in cui inserire il nuovo elemento, tale foglia viene sostituita dal nuovo elemento al quale verranno aggiunte come figlie due ulteriori foglie fintizie.

In questo modo siamo sicuri che la struttura dell'abr venga rispettata. Sostanzialmente l'operazione di Insert consiste in un'operazione di Member più una costante C per l'operazione di inserimento.

Quindi tale operazione ha complessità $O(H) + C$ con H altezza dell'abr



Procedura Insert_tree(a,r)

Complessità $O(h) + C$

```
{  
    If(r!=NULL)  
        If(a=L(r)) then "Elemento già presente in s"  
        else  
            if (a<L(r)) then  
                // sia il sottoalbero sx di T con radice V  
                inserttree(a,v);  
  
            else  
  
                // sia il sottoalbero destro di r con radice w  
                Insert_tree(a,w)  
            else  
                // inseriamo l'elemento x al posto della foglia fittizia individuata e aggiungiamo 2 figli per x alle due nuove foglie fintizie  
    }  
}
```

-Delete (x):

In questo caso l'elemento x ricercato nell'abr andrà successivamente eliminato.

Si tratta di un operazione molto pericolosa, che tende a liberare la struttura dell'abr.

Per questo motivo, la procedura di eliminazione di un elemento dovrà prevedere tutti i possibili casi;

- L'elemento da eliminare è una foglia. Questa è la situazione più semplice, in questo caso si procederà semplicemente ad eliminare la foglia
- se il nodo da eliminare ha un figlio che è foglia, si sostituisce il figlio con il padre e si elimina la foglia.
- Se il nodo da eliminare ha un solo figlio che è un nodo intero, si tratta di un figlio sinistro e si sostituisce l'elemento figlio più grande (più a destra) del sottoalbero di radice, il figlio sinistro con il nodo da eliminare e si elimina le foglie. Se di tratta di figlio destro, il nodo da eliminare si sostituisce con l'elemento figlio minimo (più a destra) del sottoalbero con padre il figlio destro e si elimina le foglie
- Se il nodo da eliminare ha 2 figli, si sostituisce il nodo da eliminare con l'elemento foglio vanno del sottoalbero destro, l'elemento foglie unico del sottoalbero destro del nodo da eliminare e si eliminano le foglie

Procedura Delete(a,r)

```
{  
    if(i!=null)  
        if a!=L(r)  
            if a < L(r) e sia v il figlio sinistro di r  
                delete (a,v)  
            else  
                //sia w il figlio destro di r  
                delete (r,w);  
            else // elemento trovato, si considerano i casi possibili  
                v e w potenziali figli sinistri e destri  
                if(v==NULL)and(W==NULL)  
                    // e è una foglia, si elimina  
                else  
                    if(v!=NULL)  
                        // si considera il max del sottoalbero sinistro e si sostituisce al figlio ed elimina le  
foglie  
                    else  
                        // uguale per sottoalbero destro  
                else  
                    // l'elemento a non appartiene ad s  
  
}
```

Domanda: Si definisca un albero binario di ricerca ottimo

Risposta:

Per definire un albero binario di ricerca ottimo, partiamo col definire cosa sia un albero binario di ricerca (ABR).

Un albero binario di ricerca è un tipo particolare di albero radicato utilizzato come implementazione fisica per diverse strutture astratte, come ad esempio i "Dizionari". Formalmente, dato un insieme S , è possibile definire un albero binario di ricerca secondo la seguente struttura.

- Ogni vertice V dell'ABR è etichettato con una etichetta $L(v)$ tale che $L(v)$ appartiene ad S
- Per ogni vertice u nel sottoalbero sinistro di v si ha che $L(u) < L(v)$
- Per ogni vertice w nel sottoalbero destro di v si ha che $L(w) > L(v)$

Formalmente, una struttura dati T è un albero binario di ricerca "se e solo se" la visita in ordine simmetrico dei suoi vertici, restituisce una lista ordinata di elementi.

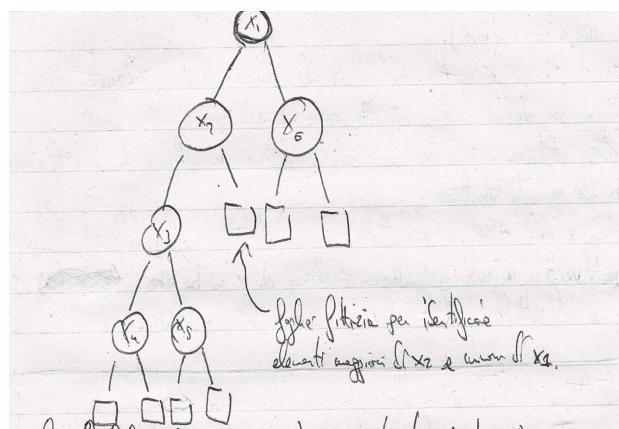
Dato un insieme $S = \{a_1, \dots, a_n\}$ con a_1, \dots, a_n in ordine crescente un sottoinsieme dell'insieme universo U e supponiamo di avere una sequenza di operazioni MEMBER sull'albero binario di ricerca associato ad S . Supponiamo di avere anche la probabilità che ciascuna istruzione di MEMBER (a, S) appaia nella sequenza $\forall a \in U$.

Le probabilità in nostro possesso si dividono in 4 categorie:

- p_i , Probabilità che l'istruzione Member(A_i, S) con A_i "appartenente" ad S , sia presente nella sequenza di istruzioni S
- q_0 , probabilità che l'istruzione Member (A, S) , con $a < a_1$, sia presente nella sequenza di istruzioni S
- q_i , probabilità che l'istruzione Member (A, S) con $a_{i-1} < a < a_i$, sia presente nella sequenza di istruzioni S
- q_n , probabilità che l'istruzione Member(A, S) con $a > a_n$, sia presente nella sequenza di istruzioni S

L'obiettivo è quello di trovare un albero binario di ricerca ottimo, ovvero un ABR per cui risulta minimo il numero di confronti necessario ad eseguire le istruzioni member della sequenza.

Per calcolare il costo di un Abr per la sequenza di istruzioni MEMBER, consideriamo un ABR con $(n+1)$ foglie fintizie, 2 figli per ciascuna foglia, per identificare i range di valori fra 2 elementi dell'ABR come segue:



La cosa scritta è "Foglie fintizie per identificare elementi maggiori di x_2 e minori di x_6 "

Numeriamo le foglie fintizie in ordine crescente da 1 ad $(m+1)$

In questo modo rimaniamo a rappresentare l'esistenza degli elementi dell'insieme U-S- Per definire formalmente il costo di un abr bisogna considerare due casi possibili:

- Dato un elemento $a \in S$ (con label $L(v) \in S$) e considerate l'istruzione $\text{MEMBER}(a,s)$ per eseguirla il numero di vertici visitati saranno pari alla profondità del vertice V, aumentata di 1, Quindi $\text{DEPTH}(V) + 1$.
- Se $a \notin S$ e $a_i < a < a_{i+1}$ allora, data l'istruzione $\text{MEMBER}(a,S)$ il numero di vertici visitati sarà pari alla profondità della foglia fittizia i (ricordando che le foglie fittizie sono aumentate in ordine crescente).

Allora il costo dell'abr sarà definito come:

$$C = \sum_{i=1}^n p_i * [\text{DEPTH}(a_i) + 1] + \sum_{i=0}^n q_i * \text{DEPTH}(i)$$

Per minimizzare il costo dell'albero si dovranno posizionare i nodi a con accesso più frequente (maggiore probabilità per MEMBER (a,S) di appartenere alla sequenza S) ad una profondità minore,mantenendo la struttura ABR

Domanda:

Si definisca un albero di ricerca ottimo e si dia un algoritmo che lo costruisca in $O(n^3)$.

Risposta:

Un albero binario di ricerca ottimo è un tipo particolare di albero binario di ricerca.

In particolare definiamo albero binario di ricerca (ABR) in cui è definita una relazione d'ordine fra le etichette dei vertici.

“ Una struttura dati T è un albero binario di ricerca “se e solo se” visitando con una visita simmetrica i suoi vertici si ottiene una lista ordinata di elementi.

Data una sequenza $S=\{a_1,\dots,a_n\}$ un sottoinsieme dell'insieme Universo U,i cui elementi a_1,\dots,a_n sono ordinati in ordine crescente.

Supponiamo di avere una sequenza S di istruzioni $\text{MEMBER}(a,s)$ dove a “appartiene” a U e di avere la probabilità che ognuna delle gerarchie Member appartiene alla sequenza U.

Un albero binario di ricerca ottimo è un ABR per cui risulta minimo il numero di confronti effettuati per eseguire le istruzioni della sequenza.

Abbiamo definito il costo di un ABR come:

$$C = \sum_{i=1}^n p_i * [\text{DEPTH}(a_i) + 1] + \sum_{i=0}^n q_i * \text{DEPTH}(i)$$

- p_i , Probabilità che l'istruzione $\text{Member}(A_i,S)$ con A_i “appartenente” ad S, sia presente nella sequenza di istruzioni Si

- q_0 ,probabilità che l'istruzione $\text{Member}(A,S)$, con $a < a_1$, sia presente nella sequenza di istruzioni S

- q_i , probabilità che l'istruzione $\text{Member}(A,S)$ con $a_{i-1} < a < a_i$, sia presente nella sequenza di istruzioni s

- q_n , probabilità che l'istruzione $\text{Member}(A,S)$ con $a > a_n$,sia presente nella sequenza

di istruzioni s

Si è dimostrato che l'abr ottimo si ottiene posizionando i vertici dell'abr in maniera tale che i vertici relativi ad istruzioni MEMBER più probabile abbiano una profondità inferiore.

Costruzione di un ABR ottimo.

Per determinare l'albero di ricerca ottimo si utilizza un paradigma di calcolo noto come "Programmazione Dinamica"

Tale metodologia consiste nel suddividere il problema principale in tanti sottoproblemi di taglie più piccole, individuando una soluzione ottimale per questi, al fine di determinare, risalendo dal sottoproblema più piccolo fino al principale, la soluzione ottima cercata

Una caratteristica fondamentale della programmazione dinamica è il concetto di tabulazione, che prevede la memorizzazione dei risultati di ottimizzazione dei sottoproblemi al fine di un più veloce riferimento futuro (può capitare che durante il procedimento un altro sottoproblema si ripeta, in questo modo si evita di risolverlo nuovamente)

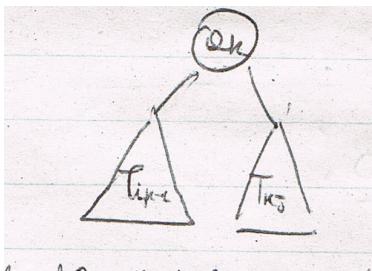
seguendo i passi fondamentali della programmazione dinamica, definiamone l'applicazione nel problema preso in esame:

-identifichiamo il tipico sottoproblema a caso, la tabella di programmazione dinamica dovrà contenere. L'obiettivo è trovare l'albero di costo minimo T_n degli elementi $S = \{a_0, \dots, a_n\}$. ciascun sottoproblema sarà identificato da un albero T_{ij} , con $0 \leq i < j \leq n$ tale che T_{ij} sarà l'albero di costo minimo per gli elementi $\{a_{(i+1)}, a_{(i+2)}, \dots, a_j\}$ ad ogni albero si associerà il costo C_{ij} e il peso W_{ij} , definiti come:

$$W_{ij} = q_i + (p_{(i+1)} + q_{(i+1)}) + \dots + (p_j + q_j)$$

$$C_{ij} = w_{(i,k-1)} * p_k + w_{kj} + C_{(ik-1)} * C_{kj}$$

come è possibile notare che le funzione costo C_{ij} è calcolato in base ai costi ed p_i pesi di 2 sottoalberi, T_{ik-1} e T_{kj} . Ogni albero φ_{ij} è infatti costituito da una radice $r_{ij} = a_k$, un sottoalbero sinistro minimo T_{ik-1} ad un sottoalbero dentro T_{kj}



Se $i=k-1$ con v sarà sottoalbero sinistro
Se $K=J$ con v sarà sottoalbero destro

-I valori iniziali dei sottoproblemi di taglia più piccoli, ovvero gli alberi T_{ii} sono $C_{ii} = 0$ $w_{ii} = q_{ii}$

la profondità di tutti i vertici T_{ik-1} e T_k viene incrementato di 1, considerazione l'albero φ_{ij} a causa della sua radice.

Dato il sottoproblema T_{ij} (con $i < j$), allora per ottenere il suo costo minimo bisogna memorizzare la somma $C_{(k-1)} + C_{kj}$ con $i < k \leq j$

questo vuol dire che l'algoritmo calcolerà il costo per qualsiasi valore di k fra i e j , ovvero scegliendo come vertici i valori fra $a_{(i+1)}$ e a_j , individuando la radice $r_{ij} = a_k$ che permette di ottenere il costo minimo.

Questo procedimento si effettua tramite procedura ROOT_OPT_SUBTREES:
Procedure ROOT_OPT_SUBTREES:

```

begin
  for i=0 to n do
    begin
      Cii=0
      wii=qij
    ed
  for l=1 to n-1 do
    for i=0 to n-l do
      begin
        j=i+l
        wi,j=wi,j-1 + pj+qj // trovato n il valore di k con i ≤ k ≤ j per cui è minima la somma
        Ci,k-1 + Ck,j
        Ci,j=wij + Ci,m-1 + Cm,j
        rij=am
      end
    end
end

```

una volta calcolato per ogni sottoalbero T_{ij} la radice $r_{ij}=a_m$ che minimizza il costo di T_{ij} , costruiamo ricorsivamente l'ABR ottimo:

Procedure BUILDTREE(i,j)

```

begin
  crea il vertice Vij(radice) etichetta vij con rij sia m l indice di Vij (rij=am)
  if i<m-1
    BUILDTREE(i,m-1)
  if m<j
    BUILDTREE(m,j)
end

```

Domanda:

Si dia la definizione di alberi 2-3 con esempio, e limitazioni superiori e inferiori all'altezza con n foglie.

Risposta:

Un'abero 2-3 è una particolare struttura ad albero caratterizzato dalla seguenti proprietà:

- tutte le foglie dell'albero si trovano allo stesso livello
- ogni nodo dell'albero può avere 2 o 3 figli

la visita in pre-ordine(radice,sx,dx) fornisce la lista ordinata dagli elementi.

Gli alberi 2-3 vengono utilizzati come implementazione fisica di strutture dati astratti.

In particolare una struttura dati astratta definisce l'organizzazione logica di un insieme di dati e il tipo di operazioni per questi dati.

Una struttura dati concreta, cosa gli alberi binari di ricerca o gli alberi 2-3, deve provvedere ad implementare fisicamente la struttura dati astratta (ad esempio il dizionario) e le sue operazioni. Un esempio iniziale di implementazione fisica sono proprio gli alberi binari di ricerca, che con complessità di tempo $O(h)$, con h altezza dell'albero, riescono ad eseguire le principali operazioni previste.

Nell'utilizzare gli ABR come implementazione fisica nasce però il problema di mantenere, in seguito ad operazioni di INSERT e DELETE, la struttura di albero.

Un'implementazione alternativa è di fornire dagli alberi 2-3.

In questo caso si dimostra che, poiché tutte le foglie si trovano allo stesso livello, l'altezza è in funzione del numero di foglie. Dimostreremo infatti che l'altezza di un albero 2-3 è $O(\log n)$, risultato che ABR si ottiene soltanto se gli ABR considerati sono completi e bilanciati.

Dimostrando che gli alberi 2-3 hanno altezza $O(\log n)$, si dimostrano che gli algoritmi su alberi 2-3

sono efficienti.

LEMMA:

Sia T un albero 2-3 di altezza h , allora il numero di foglie n è compreso fra 2^h e 3^h .

Negli alberi 2-3 le informazioni saranno memorizzate nelle foglie, mentre i nodi interi verranno utilizzati per **maggiorare** le informazioni e giungere agevolmente ad una foglia. In ogni modo verranno memorizzate 2 informazioni

- m , il valore massimo del sottoalbero sinistro

- M il massimo del sottoalbero centrale o, se esiste solo il dx del sottoalbero dx

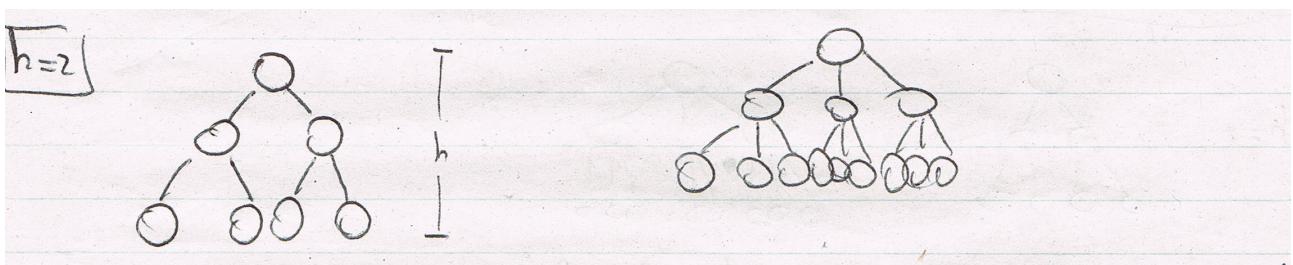
LEMMA-ALTEZZA ALBERI 2-3:

sia T un albero di altezza h , allora il numero di foglie di T sarà compreso fra 2^h e 3^h .

Dimostrazione:

sappiamo che ciascun nodo di un albero 2-3 può possedere 2 o 3 figli.

Consideriamo i seguenti due casi limite di alberi, di alberi binari e ternario completi e bilanciati:



Il numero minimo di foglie che un albero 2-3 di altezza $h=2$ è $2^h = 2^2 = 4$ foglie

Il numero minimo di foglie che un albero 2-3 ternario deve avere è $3^h = 3^2 = 9$ foglie
 $2^h \leq n \leq 3^h$

In termini di altezza (consideriamo i logaritmi), risulterà che l' altezza sarà $O(\log n)$

$$\log 2^h \leq \log n \leq \log 3^h$$

$$h \log 2 \leq \log n \leq h \log 3$$

$$\log 2 / \log_2 n \leq 1/h \leq \log_2 3 / \log_2 n$$

$$\log_2 n \leq h \leq \log_2 3 * \log_2 n$$

In generale

$$c_1 \log_2 n \leq h \leq c_2 * \log_2$$

$h = O(\log n)$ questo rende le operazioni più efficienti

Domanda : Si presenti la procedura ADDSONS e se ne discuta il suo utilizzo nell'ambito dei dizionari

La procedura ADDSONS è una procedura relativa al problema dell'inserimento di un nuovo elemento di un albero 2x3, implementazione di un Dizionario.

Un dizionario è una struttura dati astratta per insiemi dinamici, utilizzate per insiemi di elementi su cui è definita una relazione d'ordine e su cui è definita una relazione d'ordine e su cui sono definite operazioni di MEMBER, INSERT e DELETE. Una struttura dati astratta fornisce un'organizzazione logica di dati, pertanto risulta necessaria una struttura dati concreta che ne fornisce un'implementazione.

Un dizionario può essere implementato per mezzo di alberi binari di ricerca e alberi 2-3. Prendendo in considerazione l'implementazione di un dizionario tramite alberi 2-3, bisogna ricordare che ciascun nodo dell'albero potrà avere 2-3 figli. Questo equivale a dire che, nel caso in cui venga inserito un nuovo nodo come figlio di un padre di 3 figli, allora si otterrà un nodo padre di 4 figli. In questo caso si dovrà provvedere a "bilanciare" l'albero 2-3 applicando la procedura ADDSONS.

Nell'inserire un nuovo nodo nell'albero 2-3 possono presentarsi i seguenti casi:

- Il padre in cui inserire il nuovo vertice ha 2 figli, e si aggiunge il nuovo nodo come figlio di questo. Aggiungendo m ed M dei costi, lavorando sull'altezza impiego $O(\lg(n))$.
- Il padre f ha già tre figli. Allora se questo padre è un nodo interno ed ha un padre, allora si inserisce il nuovo figlio. Adesso f ha quattro figli. Si crea un nuovo vertice g, fratello di f (figlio dello stesso padre) e si aggiungono come figli di g i figli più a destra di f. Se il padre di f e g aveva già 3 figli prima della creazione di g allora si reitera il procedimento sul padre del padre di f e g.
- Se f è la radice e ha già 3 figli, gli si aggiunge il nuovo figlio. Sicuramente si creerà un nuovo vertice fratello delle radici in cui si inseriranno i figli più a destra di f, e una nuova radice che contiene la radice di f e g.

In ogni caso si avranno $\lg(n)$ trasformazioni locali. (si lavora bottom-up)

Il procedimento descritto viene eseguito dalla procedura ADDSONS su un vertice V che ha già 4 figli (è già stato inserito il nuovo nodo)?

PROCEDURA ADDSON (V) /v è il vertice con 4 figli

```
begin
    creo un nuovo vertice V'
    sposto i 2 figli più a destra di V come figlio di V'
if(v con "h" padre) then V è radice
    si crea una nuova radice r'
    v e v' diventeranno figli di r'
else //ovvero se V ha padre f
    v' diventa figlio del padre f di V precedentemente alla destra di V
if(f ha 4 figli) then
    ADDSONS(f);
else
    l'albero è bilanciato.
```

End.

La procedura ADDSON impiega complessità di tempo costante ad ogni chiamata viene richiamata al più $O(\log(n))$ volte, ossia al più altezza dell'albero 2-3.

Ripasso dimostrazione $h=O(\ln(n))$:

Sia P un albero 2-3 di altezza h , allora il numero di foglie dell'albero sarà compresa fra 2^h e 3^h .

Sia T un albero 2-3 di altezza h , allora il suo numero di foglie è compreso fra 2^h e 3^h

Prendiamo i casi limite:

$h=2$

$$2^h = 2^2 = 4 \text{ foglie} \quad 3^h = 3^2 = 9 \text{ foglie}$$

$$2^h \leq n \leq 3^h$$

$$\begin{aligned} \log_2 2^h &\leq \log_2 n \leq \log_2 3^h \\ h \log_2 2 &\leq \log_2 n \leq h \log_2 3 \\ \log_2 2 \leq \frac{(\log_2 n)}{h} &\leq \log_2 3 \Rightarrow \frac{(\log_2 2)}{(\log_2 n)} \leq \frac{1}{n} \leq \frac{(\log_2 3)}{(\log_2 n)} \\ \frac{(\log_2 n)}{(\log_2 2)} &\leq h \leq \frac{(\log_2 n)}{(\log_2 3)} \\ c1 \log_2 n &\leq h \leq c2 \log_2 3 \end{aligned}$$

Domanda : dare la definizione di visita BFS e visita DFS di un grafo.

Per visita BFS e visita DFS si intendono due algoritmi per la visita dei grafi.

Un grafo G è definito da 2 insiemi V e E , come segue:

$G=(V,E)$

V = insieme dei vertici

E = insieme degli archi $E= V \times V$

Un grafo è un modello utilizzato per rappresentare relazioni fra entità e può essere per questo considerato come un'astrazione della realtà.

In particolare, dati 2 vertici $a,b \in V$, le loro connessione è rappresentata da un arco costituito dalle coppie $(a,b) \in E$.

Un grafo può essere rappresentato mediante matrici di adiacenza e liste di adiacenza.

Nella rappresentazione mediante matrici dei gradi, si implementa una matrice $V \times V$, ove la generica componente (i,j) sarà uguale a 1^n se esisterà un arco fra i vertici V_i e V_j , 0 altrimenti. Tale struttura viene adoperata quando il grafo da implementare è denso, ovvero il numero di archi è circa il quadrato del numero di vertici $|E|$ circa $|V|^2$. La complessità di spazio è pari a $|V|^2$, una possibile accedere a un arco in tempo costante. Una rappresentazione alternativa è possibile per mezzo delle liste di adiacenza. Una lista di adiacenza è costituita da un vettore dei vertici, ove ciascuna i -esima componente è un puntatore a una lista che rappresenta gli archi adiacenti (per grafi non diretti) gli archi uscenti (per grafi diretti) al vertice i .

Questo tipo di struttura è consigliata quando il grafo è sparso, ovvero il numero di archi è basso. La complessità di spazio è $O(|V|+|E|)$, ma per verificare se due nodi risultano adiacenti bisogna considerare la componente del vettore relativa al nodo di partenza e verificare se è presente nella sua lista degli archi l'arco etichettato con il costo di arrivo.

Visita BFS :

Dato un grafo $G=(V,E)$ e salto un vertice $s \in V$ come sorgente, la visita BFS in ampiezza visita sistematicamente tutti i vertici $v \in V$ raggiungibili dalla sorgente s .

Tale visita viene denominata visita in ampiezza poiché ad ogni generico passo visita tutti i vertici a distanza k dalla sorgente e solo dopo aver visitato questi provvede a visitare tutti i vertici a distanza $k+1$.

Una caratteristica fondamentale della visita in ampiezza è quella di determinare i cammini minimi di qualsiasi vertice $v \in V$ della sorgente.

Durante la visita, l'algoritmo prende inoltre una colorazione dei vertici per tenere traccia del loro stato. Un vertice sarà colorato di bianco se non sarà ancora stato visitato, sarà colorato di grigio se è in "via" di esplorazione e sarà colorato di nero quando è stato visitato e sono stati visitati tutti i nodi ad esso adiacenti. La visita BFS produce l' ALBERO BFS avente la sorgente s come radice a cui si collegano, in maniera opportuna, tutti i vertici visitati durante la visita con sorgente s . Di seguito viene presentato l'algoritmo.

```
Procedure BFS_visit(G,s){
for ogni  $v \in V(G) - \{s\}$  do
    begin
        color[v] = WHITE;
        u[v] = nil;
        dist[v] = 00;
    end
color[s] = GRAY;
pi[s] = NIL
dist[s] = 0;
Q<-  $\{s\}$ 
while( $Q \neq \emptyset$ )
    begin
        u<- head(Q)
```

```
For ogni  $v \in \text{Adj}(u)$  do
begin
    color[v] = GRAY;
    dist[v] = dist[u]+1;
    u[v] = u;
    ENDQUEUE(Q);
end;
DEQUEUE(Q);
color[u] = BLACK;
end
}
```

VISITA DFS : (visita in profondità):

Dato un grafo $G=(V,E)$ la visita DFS provvede, ad ogni istante, a visitare in "profondità" rispetto ad un certo vertice v fino a che esiste un arco che lo permette. Il grafo viene esplorato proseguendo sempre più in profondità fra i nodi. Ad un certo istante, infatti, se il vertice v possiede ancora vertici uscenti si prosegue scegliendone uno e continuando fino a che ci è possibile. Una volta giunti ad un vertice che non possiede archi uscenti, si ripercorre il cammino effettuato nell'ordine inverso fino al primo vertice con ulteriori archi uscenti. Se qualche nodo non viene visitato si ri-applica la visita DFS scegliendo tale vertice come sorgente. A differenza della visita BFS in cui veniva generato un albero BFS, qui viene generata una foresta di alberi, ove ogni radice è la sorgente di una visita DFS. Come nella visita BFS, anche in questo caso si prevede una colorazione dei vertici. Un vertice sia bianco se non sarà ancora stato visitato, diverrà grigio in fase di esplorazione e infine nero una volta che tutti i suoi archi uscenti saranno stati considerati. La tecnica di colorazione evita che su un vertice venga applicata più volte la visita DFS, questo si intende negli alberi della foresta DFS che quindi saranno tutti disgiunti. (gli alberi non avranno vertici in comune).

Oltre al vettore dei colori ed al vettore dei predecessori vengono implementati altri due vettori rappresentanti le informazioni temporali relative all'istante in cui un vertice v viene colorato di grigio (inizializza la sua fase di esplorazione) e di nero all'istante in cui si è terminato di visitare i nodi

accessibili da v. Per ogni vertice si ha $d[u] < f[u]$.

Il tempo di esecuzione totale dell'algoritmo sarà costituito da 2 cicli che prendono $O(N)$, più le chiamate alla DFS-visit che coinvolgeranno una volta tutti gli archi del grafo $O(|E|)$. La complessità totale dell'algoritmo sarà $O(|V|+|E|)$.

Algoritmo DFS

Procedure DFS(G)

```

begin
    for ogni vertice  $v \in V[G]$  do
        begin
            color[u]=white;
            "simbolo"[u]=nil;
        end
        true $\in 0$ 
    for ogni vertice  $v \in V[G]$  do
        if(color[V]=white)
            DFS-VISIT(v)
end

```

Procedure DFI-VISIT(u)

```

begin
    color[u]<- grady;
    d[u]<- time <-time+1
    for ogni vertice  $v \in ADJ[u]$  do
        if(color[v]=white)
        {
            pgreco[v]=u;
            DFS-VISIT(v);
        }
    color[u]<-black;
    f[u]<time<-time+1;
end

```

La complessità di tempo di questo algoritmo dipende da 2 cicli for della procedura DFS entrambi $O(|V|)$, e dal fatto che tutti gli archi vengono visitati una solta volta.(quindi $O(|E|)$). L'algoritmo avrà pertanto complessità $O(M+|E|)$;

Questo algoritmo può essere utilizzato per determinare se un grafo non orientato è connesso. Basta contare il numero di volte che nel secondo ciclo for della procedura DFS viene richiamata la procedura DFS-VIST. Se cont==1 allora il grafo è connesso

Domanda: Si definisca il minimo albero di ricoprimento di un grafo e si enuncino gli algoritmi di Kruskal e Prim, facendo una loro valutazione.

Albero di ricoprimento:

Dato un grafo $G=(V,E)$ connesso e non orientato, pesato positivamente, con V insieme dei vertici ed E insieme degli archi, si definisce albero di ricoprimento per il grafo G un albero avente ttti i vertici $v \in V$ del grafo G e un sottoinsieme di archi $T \subseteq E$, tali che tutti i vertici in V risultino connessi.

E' possibile determinare il costo associato ad un albero di ricoprimento considerando la somma dei costi di tutti gli archi in esso presenti.

Dato pertanto un albero di ricoprimento $S=(V,T)$ con $T \subseteq E$, e sia $c(l)$ il costo di un arco,

si definisce costo dell'albero di ricoprimento:

$$C(S) = \sum_{l \in T} C(l)$$

Minimo albero di ricoprimento:

Dato una foresta $F = \{S_1, S_2, \dots, S_n\}$ di alberi di riempimento sul grafo $G = (V, E)$, si definisce Minimo Albero di Ricoprimento (MAR) l'albero il cui costo $C(S_i)$ è minimo

$$C(S) = \min_{1 \leq i \leq n} (C(S_i))$$

Esistono principalmente due algoritmi per il calcolo del Minimo Albero di Ricoprimento, l'algoritmo di Kruskal e l'algoritmo di Prim.

Sostanzialmente la differenza principale fra i 2 algoritmi consiste nel metodo scelto per determinare il MAR. Nell'algoritmo di Kruskal il MAR si determina a partire da una foresta e l'i-simo passo consiste nel determinare un generico arco fra 2 alberi della stessa foresta e nel fondere gli stessi.

Al contrario l'algoritmo di Prim costruisce il MAR di un singolo albero a cui viene sempre aggiunto un arco di peso minimo tale che connette un vertice del MAR in costruzione con un vertice non ancora aggiunto all'albero.

ALGORITMO DI KRUSKAL

L'obiettivo dell'algoritmo è quello di trovare il minimo albero di ricoprimento di un grafo $G = (V, E)$, definito come $S = (V, T)$ di costo minimo, dove:

$$C(S) = \min_{1 \leq i \leq n} (C(S_i))$$

L'algoritmo considera una foresta di alberi disgiunti e costruisce il MAR ricercando l'arco di costo minimo che possa collegare due vertici di due alberi della foresta e, una volta trovato, fondere opportunamente i due alberi.

Tale algoritmo applica un metodo Greedy poiché ordina in ordine non decrescente di peso gli archi del grafo considerando l'arco di peso minore che collega due alberi distinti della foresta. La struttura dati adoperata è quella della UNION-FIND, sfruttando le caratteristiche secondo cui, applicando al vertice di un albero u l'operazione FIND(u) si ottenga il rappresentante (la radice) dell'albero in cui compare u .

Tale operazione è importante in quanto, dato un arco che collega $(u, v) \in E$, possiamo fondere gli alberi u e v soltanto se sono non sono lo stesso albero, ovvero se i due vertici hanno diverso rappresentante: $\text{FIND}(u) \neq \text{FIND}(v)$.

Se i due vertici considerati appartengono a differenti alberi della foresta, possono essere allora fusi tramite l'operazione di UNION

MST-KRISKAL(G, w)

$A \leftarrow$ insieme vuoto

for ogni vertice $v \in V[G]$ do

 MAKE-SET(v),

//ordina gli archi di E per peso non decrescente

for ogni arco $(u, v) \in E$, in ordine di peso con elementi, do

 if($\text{FIND}(u) \neq \text{FIND}(v)$) then

$A \leftarrow A \cup \{(u, v)\}$

 UNION(u, v);

return(A);

ALGORITMO DI PRIM:

L'algoritmo di Prim è un algoritmo operato per il calcolo del minimo albero di ricoprimento in un grafo.

Dato un grafo connesso e con diretto pesato positivamente $G = (V, E)$, si definisce albero di ricoprimento $S = (V, T)$ del grafo G , un albero avente tutti i vertici del grafo e come archi un

sottoinsieme $T \subseteq E$ dell'insieme degli archi.

Dato un albero di ricoprimento $S=(V,T)$ è possibile definire il costo come la somma dei pesi degli archi contenenti in T , tale che se I è un arco di T e $c(I)$ è il suo peso , avremo che il costo dell'albero di ricoprimento è $C(S) = \sum_{(I \in T)} C(I)$

Dati quindi una foresta di alberi di ricoprimento $F\{ S_1, S_2, \dots, S_n \}$, è possibile determinare il minimo albero di ricoprimento come quell'albero di ricoprimento il cui costo ,ovvero la somma dei pesi degli archi, risulta minimo:

$$C(S) = \sum_{(1 \leq i \leq n)} (C(S_i)) \quad \text{con} \quad S_i \in F$$

I due algoritmi che permettono di calcolare il nimimo albero di ricoprimento solo l'algoritmo di Kruskal e l'algoritmo di Prim, la cui complessità è asintoticamente paragonabile e uguale a $O(E \log E)$ per l'algoritmo di Kruskal e $O(E \log V)$ per Prim.

Spiegazione dell'algoritmo di Prim:

l'obiettivo dell'algoritmo di Prim è quello di costruire direttamente l'albero A.

La differenza più evidente fra questo algoritmo e quello di Kruskal sta proprio nella costruzione del minimo albero di ricoprimento, costruito tramite un singolo albero per Prim e creato da una foresta di alberi per Kruskal.

L'albero A sarà inizialmente costituito da un singolo vertice r e verrà costruito fintanto che non saranno presenti insieme tutti i vertici del grafo $G=(V,E)$ di riferimento.

Ad ogni passo, secondo una strategia Greedy, l'albero viene esteso scegliendo un arco di peso minimo che collega un vertice di A con un vertice di $V-A$ (non ancora presente in A, che così verrà aggiunto).

La complessità dell'algoritmo è strettamente legata al modo in cui si sceglie un nuovo arco che collega un vertice di A con un vertice di $V-A$.

Per ottimizzare tale scelta , tutti i vertici che NON sono ancora in A vengono inseriti in una cosa a priorità A basata su un campo Key.

Per ogni vertice $v \in V-A$ si fa riferimento al valore $Key[v]$, che rappresenta il minimo tra i pesi degli archi che collegano v ad un quaunque vertice di A. Se tale arco non esiste , per convenzione, allora $Key[v]=\infty$.

L'albero A,output dell'algoritmo, sarà rappresentato dal vettore dei predecessori che per il vertice v appena aggiunto si denoterà con $pgreco[v]$ e indicherà il predecessore di v nell'albero A, ovvero il vertice di A a cui v viene collegata.

Durante l'algoritmo ,l'albero A è rappresentato da $A=\{(v, pgreco[v]): v \in V-r-a\}$

Al termine dell'algoritmo l'albero A conterrà tutti i vertici del grafo, pertando la coda Q, che conterrà i vertici del grafo on ancora in A, sarà vuota e scriveremo

$A=\{(v, pgreco[v]): v \in V-r-Q\} \Rightarrow A=\{(v, pgreco[v]): v \in V-\{r\}\}$

Pseudocodice Algoritmo Primo:

```

MST-PRIM(G,w,r)
{
    Q<-V[G] //tutti i vertici vengono inseriti nella coda perché A non ha vertici
    for ogni  $v \in Q$   $d_0$ 
        key[v]=  $\infty$  ;
        key[r]=0
         $\pi$  [r]=NIL;
    while(Q ≠ Ø) do
    {
        u<- extract_min(a); //il vertice u viene invertito all'albero A
        for ogni  $v \in Ad_j(u)$  do
            if(  $v \in Q$  ) and ( $w(u,v) < key[v]$ ) then
                 $\pi$  [v]=u
                key[v]<=w(u,v)
    }
}

```

La complessità dell'algoritmo di Prim dipende dall'implementazione della struttura dati ausiliaria per contenere i nodi ancora nell'albero A(i nodi $v \notin V-A$). Utilizzando una coda con priorità $O(V \log V + E \log V) \rightarrow O(V \log V)$ tempo per le operazioni di estrazione della coda, $\epsilon \log V$ tempo per scorrere le liste di adiacenza, in totale $O(E \log V)$.

DOMANDA: Si definisca il concetto di cammino minimo in un grafo e si fornisce un algoritmo di chiusura transitiva utilizzando un semianello generico e lo si analizzi.

-Dato un grafo $G=(V, \epsilon)$ orientato e pesato, con una funzione peso $w: E \rightarrow \text{NumeriReali}$ che associa ad ogni arco un peso a calore nei reali, è possibile definire il concetto di cammino minimo in un grafo e di peso di un cammino.

-Il peso di un cammino $P=(V_0, V_1, \dots, V_k)$ è la somma dei pesi degli archi che lo costituiscono:

$$W(P) = \sum_{i=1}^k W(V_{i-1}, V_i)$$

-Il peso di un cammino minimo da u a v è invece definito tramite la seguente funzione:

$$\begin{aligned} d(u, v) &= \min \{ w_{(p)} \mid U \xrightarrow{\quad} V \} && \text{se esiste un cammino tra } u \text{ e } v \\ d(u, v) &= \infty && \text{negli altri casi} \end{aligned}$$

Dalla definizione di peso di un cammino minimo è possibile definire un cammino massimo.

-Un cammino minimo dal vertice u al vertice v è definito come un qualunque cammino p con peso $w(p)=d(u,v)$.

Il problema della chiusura transitiva consiste nel determinare se esiste un cammino fra una data coppia di vertici. Per questo algoritmo è utile il concetto di semianello chiuso. Un semianello chiuso è una struttura $(S, \oplus, \otimes, 0, 1)$ dove S è un insieme di elementi che soddisfa le seguenti 5 proprietà:
1) $(S, \oplus, 0)$ è un monoide (chiuso rispetto a \oplus , \oplus è associativa, 0 è l'elemento centro e annullatore rispetto al prodotto);

2) \oplus è commutativa e gode della proprietà di idempotenza

3) vale la distribuzione del \otimes rispetto alla \oplus

4) Data una sequenza a_0, a_1, \dots, a_n , se è possibile stabilire una corrispondenza con l'insieme dei numeri naturali, allora a_0, a_1, \dots, a_n , esiste ad è unica.

5) L'operazione \otimes è distributiva rispetto all'operazione \oplus , sia al finito che all'infinito, ovvero:

$$\sum_i a_i \otimes \sum_j b_j = \sum_{i,j} a_i \otimes b_j = \sum_i (\sum_j (a_i \otimes b_j))$$

ALGORITMO GENERICO PER IL CALCOLO DEL CAMMINO MINIMO FRA VERTICI

Dato un grafo $G=(V, E)$ diretto e pesato, con $V=\{V_1, V_2, \dots, V_n\}$ e una funzione di labeling tale che $l_i: V \times V \rightarrow S$, con il seguente semianello chiuso $(S, \oplus, \otimes, 0, 1)$

L'algoritmo fornisce in output, per ogni coppia di vertici (V_i, V_j) con $\forall 1 \leq i, j \leq n$, l'elemento $C(V_i, V_j)$ che rappresenta tutta la collezione di tutti i label che vanno da V_i a V_j .

$\forall 1 \leq i \leq n, 1 \leq j \leq n, 0 \leq k \leq n$, si calcola C_{ij}^k come la somma dei label di tutti i cammini da V_i a V_j tale che tutti i vertici intermedi del cammino sono contenuti nell'insieme $\{V_1, \dots, V_k\}$.

Algoritmo: Procedure CamminoMinimo() [complessità di tempo $O(n^3)$]

begin

for $i=1$ to n do } "graffa che comprende anche il rigo d sotto"
 $C_{ii}^0 \leftarrow 1 + l(V_i, V_i)$ } $O(n)$

for $1 \leq i, j \leq n$ con $i \neq j$ do } "graffa che comprende anche il rigo d sotto"
 $C_{ij}^0 \leftarrow L(V_i, V_j)$ } $O(n^2)$

for $x=0$ to n do } "graffa che comprende anche i 2righi d sotto"
for $1 \leq i, j \leq n$ do } $O(n^3)$

$1 \leq i, j \leq n \quad C_{ij}^k \leftarrow -C_{ij}^{k-1} + C_{ik}^{k-1} (C_{kk}^{k-1})^* C_{kj}^{k-1};$
 $C(V_i, V_j) \quad C_{ij}^h \quad } O(n)$

ALGORITMO PER LA CHIUSURA TRANSITIVA:

partendo dall'algoritmo generico per il calcolo del cammino minimo fra 2 vertici del grafo è possibile, apportando le opportune modifiche, risolvere il problema di determinare se esiste un cammino fra una coppia di vertici (V_i, V_j) . La funzione di labeling $l_i: V \times V \rightarrow S$ viene modificata, considerando il semianello $(S=\{0,1\}, V, \wedge, 0, 1)$. In particolare tale funzione diviene:

$$\begin{aligned} l(V_i, V_j) &= 1 \text{ se } (V_i, V_j) \text{ è un arco di } G \\ l(V_i, V_j) &= 0 \quad \text{altrimenti} \end{aligned}$$

In questo modo avremo che $C(V_i, V_j) = 1 \Leftrightarrow$ esiste un cammino del vertice V_i al vertice V_j

Algoritmo per la chiusura transitiva:

Procedure chiusura_transitiva

```

{for i=1 to n do }”graffa che comprende anche il rigo d sotto “
   $C_{ii}^0 \leftarrow 1 + l(V_i, V_j)$  } 0(n)

for 1≤i, j≤n con i ≠ j do }”graffa che comprende anche il rigo d sotto “
   $C_{ij}^0 \leftarrow L(V_i, V_j)$  } 0( n2 )

for k=0 to n do }”graffa che comprende anche i2 righi d sotto “
  for 1≤i, j≤n do }0( n3 )
     $C_{ij}^k \leftarrow -C_{ij}^k \forall (C_{ik}^{k-1} \wedge C_{kj}^{k-1})$ 

for 1≤i, j≤n do }”graffa che comprende anche i2 righi d sotto “
   $C(V_i, V_j) = C^n ij$  }0( n2 )

La complessità di tempo dell'algoritmo sarà 0( n3 ).
```

ALL PAIRS SHORTEST PATH:

L'algoritmo fa riferimento al problema dei cammini minimi in un grafo ,in particolare al problema di stabilire il cammino minimo fra una qualsiasi coppia di vertici in un grafo.Per la realizzazione di questo algoritmo si utilizza il semianello chiuso (\mathbb{R}^+ ,min ,+, $+\infty$,0).

Tale algoritmo fa riferimento all'algoritmo generale del calcolo del cammino minimo fra 2 vertici,modificando la funzione di labeling $l_i : V \times V \rightarrow S$ in

$$\begin{aligned}
 l(U, V) &= c \quad \text{con } c \text{ costo dell'arco fra } u \text{ e } v \text{ tale che } c = w(u, v) \\
 l(U, V) &= +\infty \quad \text{altrimenti}
 \end{aligned}$$

Viene inoltre modificato il semianello chiuso,rispetto a quello generico dell'algoritmo,con quello sopra descritto.

Algoritmo ALLPAIRS SHORTEST PATH:

Procedure

begin

```

  for i=1 to n do
     $C_{ij}^0 \leftarrow 1 + l(V_i, V_j)$ 

  for 1≤i, j≤n con i ≠ j do
     $C_{ij}^0 \leftarrow L(V_i, V_j)$ 

  for k=0 to n do
    for 1≤i, j≤n do
       $C_{ij}^k \leftarrow \text{MIN}(C_{ij}^{k-1}, C_{ik}^{k-1} + C_{kj}^{k-1})$ 
    for 1≤i, j≤n do
       $C(V_i, V_j) \leftarrow C_{ij}^n$ 
end
```

Algoritmo per il problema Single Soure Shortest Path:

Algoritmo di Dijkstra

Domanda:

Si consideri un grafo diretto e pesato $G=(V,E)$ in cui ogni arco ha peso non negativo.

Si dia un algoritmo che calcola i cammini attivi da un vertice sorgente a qualsiasi altro vertice del grafo (Single souce shortest Path). Si analizzi l'algoritmo (Il punteggio terrà conto dell'efficienza dell'algoritmo proposto)

L'algoritmo per il problema del Single Source shortest Path fa riferimento alla teoria dei cammini minimi in un grafo, per poter descrivere il problema in esame bisogna quindi introdurre il concetto di peso di un cammino e peso di un arco. Dato un grafo $G(V,E)$ diretto e pesato positivamente, considerando il cammino $p=(v_0, \dots, v_n)$ dal vertice V_0 al vertice V_n , si definisce peso del cammino P la somma dei pesi degli archi incontrati lungo il cammino:

$$w(p) = \sum_{1 \leq i \leq n} w(v_{i-1}, v_i)$$

E' possibile trovare la nozione di peso di un cammino, definire il peso di un cammino minimo da un vertice u ad un vertice v come segue.

$$d(u, v) = \min(w(p) : u \rightarrow v) \text{ se esiste un cammino da } U \text{ a } V \\ \text{oppure } \infty \text{ altrimenti}$$

Dato un vertice $s \in V$ detto sorgente, il problema del singolo source shortest path consiste nel determinare il cammino minimo dal vertice sorgente a qualsiasi altro vertice del grafo, se tale cammino esiste

L'algoritmo utilizzato per risolvere tale problema viene chiamato "Algoritmo di Dijkstra". L'algoritmo in questione lavora costruendo un insieme S di vertici del grafo per i quali si conosce la distanza (in termini di pesi degli archi) dalla sorgente.

Si costruisce l'insieme S in maniera tale che il cammino fra la sorgente e qualsiasi vertice V dell'insieme S è interamente contenuta in S , questo proprio perché durante la costruzione inseriamo nell'insieme S , un nuovo vertice che è raggiungibile da V grazie ad altri altri vertici in S . Ad ogni passo si aggiunge ad S il vertice $V \notin S$ che ha somma dei pesi minima dalla sorgente.

L'algoritmo che viene ora descritto viene utilizzato esclusivamente per grafi diretti e pesati positivamente, mentre per i grafi pesati negativamente si adopera l'algoritmo Bellman-Ford.

Formulazione Algoritmo di Dijkstra

Sia $G=(V,E)$ un grafo diretto e pesato positivamente con una funzione di labeling $L(V_i, V_j)$ tale che $L(V_i, V_i) = 0$ e $L(V_1, V_0) = +\infty$ se $(V_i, V_j) \notin E$ sia V_0 la sorgente V .

L'algoritmo restituisce in output $\forall n \in V$ il cammino minimo da V_0 a V_1 calcolato sulla somma dei pesi degli archi lungo il cammino.

L'idea è quella di costruire un insieme $d \subseteq V$ che conterrà tutti i vertici per cui è nota la distanza minima della sorgente e talché che scelto un qualsiasi vertice $u \in d$ l'insieme dei vertici del cammino da V_0 a u sia contenuto tutto in d .

(Questo concetto è fondamentale per la verifica della correttezza dell'algoritmo).

Durante l'esecuzione dell'algoritmo si tiene traccia di un array $D[v]$ che contiene il costo del cammino più breve da V_0 a V individuato fino a quel momento tramite gli altri vertici in S .

Procedura dell'Algoritmo di Dijkstra

se $(V_0, V_I) \notin \square$ allora $l(V_0, V_i) = \infty$

Begin

```

1   S<-{V0};
2   d[V0]<-0;
3   for ogni vertice v    $\in V - \{V_o\}$  do
        d[v]= L (V0,V1)
4   While  ( $S \neq V$ ) do
      begin
5       scegli un vertice w $\in V - S$  e con D[w] minimo
6       aggiungi w ad s
7       for ogni v $\in V - S$  do
8         d[v]<- min (d[v],d[w]+ l(w,v))    ("tecnica del rilassamento")
9   end
end

```

Questo algoritmo avrà complessità $O(n^2)$ condizionata parzialmente dal ciclo While, eseguito approssimativamente $|V|$ volte quindi asintotico su $O(n)$ e che contiene un ciclo for eseguito anch'esso $O(n)$ volte.

La complessità di tempo dell'algoritmo sarà asintotica a $O(n^2)$

Verifica della correttezza dell'algoritmo di Dijkstra

E' possibile dimostrare per induzione sulla dimensione dell'insieme S che, per ogni vertice $V \in S$, $D[V]$ è pari alla lunghezza (in termini di peso) del più piccolo cammino da V_0 a V .

Induzione sulla Dimensione di Si

Passo base $|S|=1$

E' vero, in questo l'unico elemento dell'insieme S è la sorgente ed ha distanza 0 da se stesso.

$D[V_0]=0$

Passo Induttivo.

Supponiamo che al generico passo si scelga w come vertice appartenente a $V-S$ ma con $D[W]$ minimo. Se $D[w]$ non è la lunghezza del più breve cammino da V_0 a w esisterà allora un cammino minimo P che conterrà alcuni vertici, altre w, che non stessa in S (altrimenti il cammino sarebbe stato preso in considerazione). Ipotizziamo che v sia proprio il primo di tali vertici $\in V-S$.

Sicuramente però la lunghezza di V sarà minore di quella di $D[w]$ (visto che il cammino p da v a w ha lunghezza minore di $d[w]$ ma sua parte avrà a maggior ragione lunghezza minore), pertanto $d[v] < d[w]$.

Ma v è il primo dei vettori del cammino che non appartiene a S e non può essere che $D[V] < D[w]$ perché $L[w]$ è stato scelto inizialmente come minimo, altrimenti si sceglierrebbe $d[v]$. Si conclude affermando che il cammino P con ente proprio per tale contraddizione e $D[w]$ è la lunghezza del cammino minimo da V_0 a w

Domanda:

Si dia la definizione di componente biconnessa in un grafo non orientato e connesso

Risposta:

Il concetto di componente biconnessa di un grafo fa parte dei problemi di connettività dei grafi. In generale un grafo si dice connesso se e solo se preso comunque una coppia di vertici u, v appartenenti a v esiste un cammino che li collega, ovvero ogni nodo del grafo è raggiungibile da un altro.

Legato al concetto di componente biconnessa è il concetto di punto di articolazione.

Dato un grafo G connesso e non diretto, definiamo un vertice "a" punto di articolazione "se e solo se" esistono almeno 2 vertici w_1 e w_2 tali che ogni cammino che ha w_1 e w_2 passi per "a".

Un punto di articolazione è un particolare vertice del grafo connesso e non orientato, la cui rimozione dal grafo divide il grafo in 2 o più parti.

Un grafo viene detto biconnesso se e solo se non esistono vertici del grafo che sono punti di articolazione.

Questo equivale a dire che un grafo è biconnesso se comunque scelta una tripla di vertici v, w, a esiste un cammino da v a w che non passa per a .

Per definire una componente connessa di un grafo bisogna introdurre una particolare relazione di ricorrenza R sui grafi con le seguenti proprietà:

- $c_1 R c_2 \Leftrightarrow C_1 = C_2$ oppure esiste un ciclo
- riflessiva $e \in E$ per ogni $e \in S$
- simmetrica $e_1 \in E$ se e solo se $e_2 \in E$
- transitiva $e_1 \in E$ e $e_2 \in E$ se $e_1 \in E$

Applicando all'insieme degli archi E tale relazione di equivalenza è possibile definire delle cosiddette "classi di equivalenza" che suddividono l'insieme E in più parti (E_1, E_2, \dots, E_k)

Un grafo G può essere diviso in componenti biconnesse, ossia sottografi che non hanno punti di articolazioni. $1 \leq i \leq k$ sia V_i l'insieme dei vertici degli archi in E_i . Ogni $G_i = (V_i, E_i)$ è una componente connessa.

Domanda : Definire l'algoritmo di BellMan Ford

L'algoritmo Djmstrei può non essere definito in grafi che presentano archi con pesi minimi, in quanto se esiste un ciclo con costo totale negativo non è definita la distanza minima per tutti i vertici raggiungibili tramite questo ciclo. In queste circostanze si fa uso dell'Algoritmo BellMan Ford che seppur con complessità di tempo maggiore rispetto a Djmstrei (che risolveva in $O(|E|^*|V|)$ o $O((V+E)\log(V))$ con coda a priorità) riesce a trattare il problema con complessità $O(|E|^*|V|)$. (Restituisce true se il grafo contiene cicli di peso negativo, altrimenti restituisce false)

L'algoritmo di BellMan Ford, considerando un vertice v e un cammino da v a y , si può applicare il seguente passo:

$$\text{if } (D_{[y]} > D_{[v]} + w(pi_{vy})) \text{ then } D_{[y]} \leq D_{[v]} + w(pi_{vy})$$

Concetti principali – Problemi di connettività

Punto di articolazione :

Dato un grafo $G=(V,E)$ connesso e non orientato, definiamo un suo vertice a punto di articolazione se e solo se esistono almeno due vertici w_1 e w_2 in cui ogni cammino da w_1 a w_2 passa per a.

Grafo connesso e punti di articolazione:

Un grafo $G=(V,E)$ si dice connesso se e solo se presi comunque due suoi vertici u e v è possibile definire un cammino che lo collega (ogni vertice del grafo è raggiungibile da qualcun altro vertice del grafo).

Se a è un punto di articolazione allora esistono 2 vertici w_1 e w_2 che sono collegati da cammini che passano tutti per a. Questo vuol dire che rimuovendo il vertice a si disconnette il grafo, dividendolo in 2 o più parti.

Grafo Biconnesso:

Un grafo $G=(V,E)$ si dice Biconnesso se e solo se nessuno dei suoi vertici è punto di articolazione. Un grafo che si dice biconnesso se, scelta comunque una tripla di vertici v, w, a è possibile definire un cammino da v a w che non passi per a.

Se un grafo è biconnesso, rimuovendo uno qualsiasi dei suoi archi si ottiene sempre un grafo connesso.

Componente biconnessa di un grafo

Per definire le componenti biconnesse di un grafo si introduce la relazione di equivalenza fra ordini R:

La relazione di equivalenza sull'insieme degli archi E implica una partizione dell'insieme E in classi di equivalenza che definiscono le cosiddette “componenti connesse”.

Un grafo G può essere suddiviso, in base a una classe di Equivalenza E_i , in componenti biconnesse. Una componente biconnessa è un sottografo di G che non presenta punti di articolazione e tale che:

$1 \leq i \leq k$ con V_i insieme dei vertici degli archi E_i

Ciascuna $G_i=(V_i,E_i)$ è una componente biconnessa.

Relazioni fra componenti biconnesse (classi di equivalenza) e punti di articolazione:

Lemma: Per $1 \leq i \leq k$ sia $G_i=(V_i,E_i)$ una componente biconnessa del grafo connesso e non diretto allora:

- 1) G_i è biconnessa per ogni i tale che $1 \leq i \leq k$
- 2) Per ogni $i \neq j$, $V_i \cap V_j$ contiene al più un vertice come intersezione

- 3) “a” è un punto di articolazione di G se e solo se $a \in V_i \cap V_j$ per qualche $i \neq j$

Dimostrazione relazione componenti biconnesse e punti di articolazione

1 : Si deve dimostrare che G_i è ancora biconnessa. Affermare che il sottografo $G_i(V_i, E_i)$ è ancora biconnesso equivale a dire che nessun vertice in G_i è punto di articolazione. Dimostriamo per assurdo, supponendo che esista in G_i un vertice punto di articolazione.

Per definizione allora esisteranno due vertici v e w i cui cammini da v a w passano tutti necessariamente da a . Questo vuol dire che l'arco (v,w) non appartiene a E' . Per costruzione, tutti gli archi appartenenti alla classe di equivalenza definita da E' sono in relazione di equivalenza fra loro, ed esiste un ciclo che li contiene.

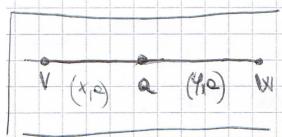
Ma questo vuol dire che esisterà un ciclo che contiene anche gli archi (v,v') (w,w') perché appartengono a E' . Per definizione tutti gli archi e tutti i vertici appartenenti a questo ciclo sono in E' e V' . Considerando questo ciclo, possiamo affermare che esistono dei cammini da v a w e soltanto uno di loro comprenderà il vertice “ a ”, pertanto “ a ” è punto di articolazione.

2: Vogliamo dimostrare che l'intersezione degli insiemi dei vertici di due differenti componenti biconnesse può avere al più un punto di intersezione. Si dimostra per assurdo partendo dall'ipotesi che esistono due vertici u e v punti di intersezione di 2 insiemi tali che esista l'intersezione fra questi. Per definizione esisterà un ciclo C_1 in G_i che comprende i vertici v e w e un ciclo C_2 in G_j che comprende v e w .

E_1 e E_2 sono classi di equivalenza e sono 2 insiemi disgiunti, questo vuol dire che l'insieme degli archi in C_1 e C_2 devono essere disgiunti. Ma avendo 2 cicli C_1 e C_2 che contengono v e w vuol dire che esisterà almeno un arco in comune fra C_1 e C_2 , ovvero un arco in comune fra E_1 e E_2 , che quindi ci sono più classi di equivalenza, come supposto.

Dobbiamo dimostrare che “ a ” è un punto di articolazione $\Leftrightarrow a \in V_i \cap V_j$, con $i \neq j$.

Prima dimostrazione \Rightarrow : supponiamo che a sia punto di articolazione per G allora esisteranno 2 vertici v e w tali che v, w e a sono distinti e ogni cammino tra v e w contiene a .



dato che G è connesso, vi sarà almeno un cammino.

Supponiamo esistono 2 archi (x,a) e (y,a) che collega i vertici v e w e passino per a .

non può esistere un ciclo fra gli archi (x,a) e (y,a) perché altrimenti ci sarebbero 2 cammini da v a w e uno di questi non passerebbe per a , ma questo non può accadere perché a è punto di articolazione e tutti i cammini da a a w devono passare per a . questo vuol dire che, se non esiste un ciclo che contiene (x,a) e (y,a) , questi due archi si trovano in componenti biconnesse differenti e l'unico punto di intersezione è proprio “ a ” punto di articolazione.

Seconda dimostrazione \Leftarrow : se a è punto di intersezione fra gli insiemi di vertici di due differenti componenti biconnesse $a \in V_i \cap V_j$ allora a è punto di articolazione per il grafo G .

se $a \in V_i \cap V_j$ allora vi saranno 2 archi (x,a) e (y,a) che appartengono rispettivamente ad E_i E_j visto che questi due archi appartengono a E_i E_j con $i \neq j$, allora appartiene a 2 cicli disgiunti, ma consegue che ogni cammino da x a y debba passare necessariamente per questi due archi e

quindi per a, che sarà quindi punto di articolazione.

NP-COMPLETEZZA: CLASSI DI PROBLEMI:

E' possibile definire in genere 3 classi di problemi:

- P: problema che ammettono algoritmi di risoluzione efficienti cioè in tempo polinomiale
- NP: problemi che non possono essere risolti con algoritmi efficienti ma in tempo esponenziali
- NP-C: Non sono stati definiti algoritmi efficienti in tempo polinomiale ma non è stato definito il lower bound

$P = \{L \mid \text{esiste una macchina di turing deterministica che riconosce } L \text{ in tempo polinomiale}\}$

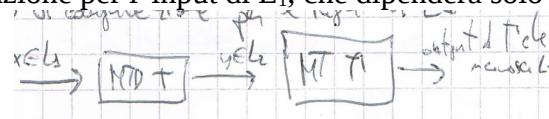
$NP = \{L \mid \text{esiste una macchina di turing non deterministica che riconosce } L \text{ in tempo polinomiale}\}$

Trasformazione polinomiale:

Dati due linguaggi L_1 ed L_2 si dice che L_1 è polinomiale trasformazione in L_2 ovvero $L_1 \leq_p L_2 \iff \exists \text{MTD } T \text{ tale che } \forall x \in L_1 \exists y \in L_2 \text{ tale che } y \in T(x)$



connettendo l' uscita di questa MTD T all' ingresso di una MT T' che riconosce il linguaggio L_2 , si otterrà un tempo di computazione per l' input di L_1 , che dipenderà solo dal tipo di MT di T' :



– Se T' è una MTD che

in tempo polinomiale, allora potremo affermare che $L_2 \in P$.

- se quindi $L_1 \leq_p L_2$. allora potremo riconoscere $x \in L_1$ tramite la MT y e affermare che anche $L_1 \in P$ (perchè in qualche modo esiste una MTD, finora di T e T' che riconosce L_1 in tempo polinomiale)
- se T' è una MTND che riconosce L_2 in tempo polinomiale allora potremo affermare che $L_2 \in NP - \{P\}$ per lo stesso motivo di $L_1 \in NP$ perchè sarà riconosciuto da una MTD^y e una $MTD^{y'}$ entrambi polinomiali e quindi sarà riconosciuta da una MTND in tempo polinomiale.

Classe dei linguaggi NP_completi:

definiamo la classe NP-completi(NP-C) come quella classe di linguaggi che soddisfa le seguenti proprietà:

- $L \in NP$

-ogni linguaggio $L' \in NP$ è trasformandole polinomialmente in L .

Teorema di Cook:

Il teorema di cook dimostra che il problema della soddisfattibilità di espressioni booleane (SAT) è un problema NP-COMPLETO.

Il SAT determina se un' espressione booleana è soddisfacibile, ovvero se esiste una qualche combinazione di valori da assegnare ai letterali per rendere "vera" la funzione booleana:

$$f(x_1, \dots, x_n) = \begin{cases} 1 & \text{se } f \text{ è vera} \\ 0 & \text{se } f \text{ è falsa} \end{cases}$$

esistono 2^n possibili combinazioni dei valori assegnati alla n-pla x_1, \dots, x_n , ognuno con un proprio valore di verità della funzione f .

una funzione f è soddisfattibile \iff esiste un n-tuple di assegnazione di verità ai letterali x_1, \dots, x_n che la rende vera.

Una funzione f non è soddisfattibile \Leftrightarrow tutte le 2^n possibili ennuple rendono falsa la funzione.

Un algoritmo che trova tutte le combinazioni di ennuple e la testa per vedere se la funzione è vera ha complessità $O(T(n)2^n)$. Il teorema di cook (e il successivo corollario a questo teorema riferito) dimostrano che il problema SAT è un problema NP-C sia nella forma FND che FNC

Forme di espressione booleane

-**FND-Forma Normale Disgiunta:** un'espressione booleana espressa in FND viene vista come una disgiunzione di conseguenze tali che:

disgiunzione = \vee

congiunzione = \wedge

$$f(x_1, x_2, \dots, x_n) = (x_1 \wedge x_2 \wedge x_3) \vee (x_4 \wedge x_5 \wedge x_6) \vee (\dots) \vee (x_{(n-2)} \wedge x_{(n-1)} \wedge x_n)$$

In questa forma, per essere vera la funzione f basta che sia vera almeno una clausola.

Il teorema di cook fa riferimento a questa forma (SET \in NP-C).

-**FNC-Forma Normale Conguittiva:** Un'espressione booleana espressa in FNC viene vista come una **congiuntore** di disgiunzioni:

$$\text{3-SAT} \rightarrow f(x_1, x_2, \dots, x_n) = (x_1 \wedge x_2 \wedge x_3) \vee (\dots) \vee (x_{(n-2)} \wedge x_{(n-1)} \wedge x_n)$$

tale che la funzione risulta vera se in ogni clausola almeno un letterale è uguale ad 1.

Dimostrare che il problema 3-SAT è NP-COMPLETO:

Il problema 3-SAT è NP-COMPLETO.

Per dimostrarlo basterà dimostrare che 3-SAT \in NP e che un linguaggio che sappiamo essere NP-COMPLETO è trasformabile polinomiale nel 3-SAT.

- Per dimostrare che 3-SAT \in NP basta considerare una macchina di turing non deterministica che, tramite il metodo del guess and verify indovini una soluzione e possa verificare che sia quella giusta in tempo polinomiale.

Supponiamo infatti che un linguaggio L \in NP se esiste una macchina di turing non deterministica che riconosce L con complessità di tempo polinomiale. Definiamo che $SAT \alpha 3\text{-SAT}$.

In generale, un linguaggio L_1 si dice trasformabile polinomialmente nel linguaggio $L_2(L_1 \alpha L_2)$ se esiste una macchina di turing deterministica T che “traduce” gli input di L_1 negli input di L_2 , ovvero una MTP T che in tempo polinomiale dato $x \in L_1$ restituisce $y \in L_2$ $x \in L_1 \Leftrightarrow y \in L_2$

Dato il problema SAT (soddisfattibilità di espressioni booleane), sappiamo per il teorema di cook essere un problema NP-COMPLETO, anche nella forma di conseguenze di disgiunzione (FNC). Per dimostrare che 3-SAT è NP-COMPLETO basta dimostrare che SAT è trasformabile polinomialmente nel problema 3-SAT, ovvero che esiste una macchina di turing deterministica che preso un x in input restituisce y in output con $x \in SAT \Leftrightarrow y \in 3\text{-SAT}$.

DOMANDA : Dimostrare che 3-sat sia \in in Np-completo

RISPOSTA:

Data una espressione booleana nella forma normale congiuntiva FNC, il problema di determinare se è soddisfattibile è Np-completo. Dimostriamo che FNC-SAT è trasformabile polinomialmente nel 3SAT, così da dimostrare che anche 3-SAT \in Np-Completo.

Consideriamo una qualsiasi espressione booleana nella forma FNC

$$(X_1 \vee X_2 \vee \dots \vee X_n) \wedge (X_{n+1} \vee \dots) \wedge \dots$$

Per dimostrare che FNC-SAT \sqsubseteq 3-SAT, dimostriamo che ogni clausula $(X_1 \vee \dots \vee X_n)$ può essere trasformata in una 3-SAT, ovvero in un'espressione booleana in forma usuale continua in ciascuna clausula che comprende esattamente 3 letterali.

Prendiamo una clausola di un'espressione booleana FNC-SAT e trasformiamola in un 3-Sat come segue:

$$(X_1 + X_2 + \dots + X_n) \quad \text{con } n \geq 4 \quad \text{la somma viene riscritta in}$$

$$(X_1 + X_2 + Y_1)(X_3 \bar{Y}_1 + Y_2)(X_4 + \bar{Y}_2 + Y_3) \dots (X_{k-2} + \bar{Y}_{k-4} + Y_k)(X_{k-1} + K_4 + \bar{Y}_{k-3})$$

Introducendo così le nuove variabili y_1, \dots, y_{k-3} .

La nuova espressione riscritta sarà uguale ad 1 \Leftrightarrow sarà uguale ad 1 uno qualsiasi dei letterali x_1, \dots, x_n della clausola iniziale, ovvero sarà uguale ad 1 la clausola iniziale.

La clausola 3Sat sarà soddisfattibile \Leftrightarrow la clausola aggiunta FNC sarà soddisfattibile. La nuova formula trovata è soddisfattibile \Leftrightarrow la clausola del SATFNC è soddisfattibile, ovvero quando un qualsiasi letterale x_1, \dots, x_k è uguale a 1.

Dimostrazione della prima implicazione \Leftarrow

Se la clausola del FNC-SAT è soddisfattibile allora quando risulta vera ci sarà un certo $X_i = 1$.

Dato un certo $x_i = 1$, imponiamo il valore dei vari y_j tali che:

$$\begin{aligned} Y_j &= 1 \quad \text{se} \quad j \leq i-2 \\ Y_j &= 0 \quad \text{se} \quad j > i-2 \end{aligned}$$

E' possibile verificare che anche le formule 3-Sat trovata è in questo modo soddisfatta.

Dimostrazione della seconda implicazione \Rightarrow

Supponiamo che la formula del 3-SAT trovata sia vera per qualche valore degli Y_j

- Se $Y_1 = 0$ allora X_1 o X_2 saranno uguali ad 1
- Se $Y_{k-3} = 1$ allora $x_{n-1} \circ x_n$ sarà uguale ad 1
- Se $Y_1 = 1$ e $Y_{k-3} = 0$ allora per qualche i , con $1 \leq i \leq k-4$ si avrà $Y_i = 1$ e $Y_{i+1} = 0$
allora si avrà che $X_{i+2} = 1$

Data una qualsiasi espressione in FNC possiamo trovare, applicando una trasformazione ad ogni somma, un'espressione in 3-FNC E che è soddisfattibile \Leftrightarrow l'espressione SAT-FNC originale è soddisfattibile

L'espressione 3-Fnc E può essere trovata in forma polinomiale rispetto alla larghezza di E (vuol dire che esiste una MTD che effettua la trasformazione in tempo polinomiale).

DOMANDA :

Problema del Clique per Grafi non Diretti

RISPOSTA:

Dato un grafo $G(V, E)$ non diretto.

Un K-Clique in un grafo è un sottografo completo (ogni distinta coppia di vertici è connessa da un arco) di un grafo G con K vertici. Il problema del k-clique consiste nel determinare se in un grafo esiste un clique di taglia k

Il problema del Clique per grafi è NP-COMPLETO

Per dimostrare che è NP-COMPLETO basta dimostrare che il clique \in NP e che un problema che sappiamo essere NP-COMPLETO è trasformabile polinomialmente nel clique problem.

Il problema del clique appartiene ad NP in quanto è possibile implementare una macchina di Turing non deterministica che “indovini” quali k vertici possono rappresentare un sottografo completo e verificare se C’è un arco fra ogni coppia di vertici.

Una tale macchina di touring non deterministica ha complessità $O(n^3)$, ove n è la lunghezza del clique (attualmente non esiste una MTD per risolvere il clique problem in tempo polinomiale).

Dimostrare che un linguaggio L1 è trasformabile polinomialmente in linguaggio L2 equivale a dimostrare che esiste una macchina di Touring deterministica che in tempo polinomiale, prende in input x e restituisce in output y, con $x \in L_1 \iff y \in L_2$.

Quanto L_1 è trasformabile polinomialmente in L_2 si scriverà $L_1 \leq L_2$

Dimostriamo che un problema NP-COMPLETO il problema della soddisfattibilità di espressioni booleane in forma normale congiuntiva (FNC-SAT) è trasformabile polinomialmente nel clique problem.

La dimostrazione consiste nel costruire un grafo t da un’espressione di lunghezza q booleana in FNC “F” tale che G ha un clique di dimensione q se e solo se F è soddisfattibile.

Sia $F = F_1 F_2 \dots F_q$ un’espressione booleana in FNC, ove ogni F_i è ordinato fattore è rappresenta una clausola (una disunione di letterali) della forma $(X_{i1} + \dots + X_{in})$ dove X_{ij} è un letterale.

Si costruisce un grafo $G(V, E)$ non diretto i quali vertici sono coppie di interi $[i, j]$ ove i rappresenta il fattore F_i e k rappresenta il letterale X_{ij} del fattore i . Per costruzione esisterà un arco $([i, j][k, l])$ fra i vertici $[i, j]$ e i vertici $[k, l]$ se e solo se i “diverso” da k e X_{ij} “diverso” da X_{kl} .

Intuitivamente $[i, j]$ e $[k, l]$ sono due vertici adiacenti in G e corrispondono a diversi fattori ed è possibile assegnare i valori alle variabili letterali in X_{ij} e X_{kl} in modo tale che insieme i letterali possano assumere valore 1 (ovvero in maniera equivalente F_i e F_k assumono valore 1). I letterali X_{ij} e X_{kl} possono essere lo stesso letterali (che evidentemente si trovano in clausole differenti) o X_{ij} e X_{kl} sono complementate e non complementate versioni di differenti letterali.

Il numero dei vertici nel grafo G costruito è chiaramente inferiore alla lunghezza di F e il numero di archi è al più il quadrato del numero di vertici del grafo.

Così si può essere codificato come stringa la cui lunghezza è limitata da un polinomio nella lunghezza di F .

Ancora più importante, tale codifica può assumere calcolata in un tempo limitato da un polinomio di larghezza F .

Il grafo G ha un q -clique (un sottografo completo di q vertici) se e solo se F di lunghezza q è soddisfattibile.

Ne consegue che noi possiamo costruire un algoritmo in tempo polinomiale per il FNC-SAT convertendo un’espressione booleana in FNC di lunghezza q in un grafo G tale che G contenga un q -clique se e solo se F di lunghezza q è soddisfattibile.

Dimostrazione

F è soddisfattibile se e solo se il grafo G costruito ha u “chiave” di lunghezza q con q vertici.

Dimostrazione prima implicazione =>

Assumiamo che F è un'espressione booleana in FNC di lunghezza Q soddisfattibile allora esisterà un assegnazione di 0-1 letterali tali che $F=1$ e di conseguenza ogni fattore $F_i=1$. Ogni fattore F_i contiene almeno un Fattore $X_{in}=1$

Affermiamo che l'insieme di tali letterali per ogni fattore, forma un clique di q vertici nel grafo: $[i, mi]$, $1 \leq i \leq q$ è un insieme di vertici del grafo G codificato a forma di un q-clique

Altrimenti esisterebbero i e j tali che non ci sia un arco tra i vertici.

Ma questo è impossibile dal momento che $X_{imi}=X_{jn}=1$ dal modo in cui sono stati selezionati per ipotesi

Dimostrazione della seconda implicazione \Leftarrow

Assumiamo che G ha un clique di dimensione q.

Ogni vertice nel grafo (i,j) deve avere una diversa prima componente, in quanto 2 vertici aventi stessa prima componente non sono collegati e in q-clique tutti i vertici sono collegati tra loro.

Ci sono esattamente q vertici nel grafo, in quanto ogni vertice ha una corrispondenza 1 ad 1 con un fattore dell'espressione booleana F.

Siano $[i, mi]$ i vertici del clique con $1 \leq i \leq q$ definiamo gli insiemi S1 e S2.

$S_1 = y$ tale che $X_{imi} = y$ con $1 \leq i \leq q$ e y è una variabile

$S_2 = \bar{y}$ tale che $x_{imi} = \bar{y}$ con $1 \leq i \leq q$ e y è una variabile

Sappiamo che $S_1 \cap S_2 = \emptyset$ altrimenti ci sarebbero archi $[S, ns] e [T, nt]$

dove $X_{ns} = \overline{X}_{nt}$ impossibile per definizione.

Per costruzione, variabili di $S_1=1$ e le variabili di $S_2=0$, fanno diventare i valori di $F_i=1$ per ogni $1 \leq i \leq q$ Questo vuol dire che $F=1$ ed è soddisfattibile.

DOMANDA :

Dimostrare che il Vertex-Cover è un problema NP-Completo

RISPOSTA:

Dato un Grafo $G(V, E)$ non diritto, è possibile definire il vertex cover come un certo sottoinsieme $S \subset V$ di vertici tali che ogni arco di G è incidente su qualche vertice di S . Il problema del vertex-cover è un problema Np_completo.

Per dimostrarlo basta considerare il problema del clique, che sappiamo essere Np-COMPLETO e dimostrare che questo è trasformabile polinomialmente nel problema del VERTEX-COVER.

Dimostrazione:

Dato un grafo $G(V, E)$ non diretto e si consideri il suo complemento $\overline{G} = (V, \overline{E})$ ricordiamo che $\overline{E} = \{(v, w) \text{ tale che } v \in V, v \neq w \text{ e } (v, w) \text{ non appartiene ad } E\}$. Si dimostra che un sottoinsieme proprio $S \subset V$ è un clique di $G \Leftrightarrow$ l'insieme $V-S$ è un

Vertex-cover in G' .

Dimostrazione della prima Implicazione ==>

Sia $S \subset V$ un clique per il grafo G .

Sappiamo che in un clique tutti i vertici, intestati sono collegati da un arco (è un sottografo completo) Questo vuol dire che nel grafo Complementato G' non esiste alcun arco che collega 2 vertici in S .

Questo vuol dire che tutti i vertici in S avranno archi adiacenti $V-S$, in G' , il che implica che $V-S$ è un Vertex-cover per G'

Dimostrazione della seconda Implicazione <=>

Sia $V-S$ un Vertex-Cover in G' . questo vuol dire che tutti gli archi in G' sono adiacenti in $V-S$, e che da qualsiasi vertici in S non sono Connessi in G' . Ma questo vuol dire che 2 Vertici Qualsiasi in S sono connessi in G , e che S è un clique di G

In Generale, per trovare se esiste un clique di taglia k in G , bisogna costruire G' e determinare se esso ha un vertex-cover di $|V|-c$.

Sicuramente si può trovare una rappresentazione di G' e $|V|-k$ in tempo polinomiale nella lunghezza della rappresentazione di G e K .

DOMANDA :

Subgraph isomorphism problem

RISPOSTA:

In generale vi è un isomorfismo fra 2 grafi G_1 e G_2 quando esiste una funzione biettiva dai vertici di G_1 ai vertici di H che **"pesava"** la struttura relazionale del grafo, nel senso che c'è un arco dal vertice u al vertice v (con u,v appartenenti a G_1)

se e solo se c'è un **"creo"** dal vertice $f(n)$ al vertice $f(u)$ in G_2 (G_1 e G_2 rappresentano lo stesso grafo ma con nomi dei vertici distinti).

L'isomorfismo di sottografi è un problema Np-Completo

Dati 2 grafi G e H il problema consiste nel dimostrare se un sottografo di G è isomorfo al grafo H . La risposta in output è SI o NO (problema decisionale)

Dimostrazione:

Per dimostrare che il problema degli isomorfismi è NP_completo basterà dimostrare che il Clique-problem , problema np-completo, è trasformabile polinomialmente nel problema in esame.

Il clique problem dato un grafo G e un valore K , consiste nel determinare se il grafo G contiene un sottografo completo di k vertici.

Per tradurre questo concetto nell'isomorfismo di sottografi basta scegliere il grafo H in un grafo completo K_k . La risposta al problema dell'isomorfismo fra sottografi per G e H è equivalente alle risposte per il Clique-Problem di un grafo G e K Questa "traduzione" può essere eseguita in tempo polinomiale mostrando che il problema dell'isomorfismo fra sottografi è Np-Completo

Tabella di Complessità delle operazioni

Instruction	Cost
1. LOAD a	$t(a)$
2. STORE i	$l(c(0)) + l(i)$
STORE $*i$	$l(c(0)) + l(i) + l(c(i))$
3. ADD a	$l(c(0)) + t(a)$
4. SUB a	$l(c(0)) + t(a)$
5. MULT a	$l(c(0)) + t(a)$
6. DIV a	$l(c(0)) + t(a)$
7. READ i	$l(\text{input}) + l(i)$
READ $*i$	$l(\text{input}) + l(i) + l(c(i))$
8. WRITE a	$t(a)$
9. JUMP b	1
10. JGTZ b	$l(c(0))$
11. JZERO b	$l(c(0))$
12. HALT	1

COMPUTATIONAL COMPLEXITY OF RAM PROGRAMS

Fig. 1.11. Logarithmic cost of RAM instructions, where $t(a)$ is the cost of operand a , and b denotes a label.

takes into account the fact that $\lfloor \log n \rfloor + 1$ bits are required to represent an integer n in a register. Registers, we recall, can hold arbitrary values. The logarithmic cost criterion is based on the crude assumption that