

Esercizi algoritmi



QUESITO 1

1)

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

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

ALLORA ABBIAMO CHE IL NUMERO DI PASSI È PARI A Cn QUINDI IL COSTO ASINTOTICO È PARI A

$$\Theta(n)$$
 DOVE n È GIÀ IN DIMENSIONE DELL'INPUT.

3) L'ALGORITMO È IN-PLACE QUINDI LA MEMORIA USATA È COSTANTE RISPETTO ALL'INPUT. CIÒ CHE È IN FUNZIONE DELL'INPUT È LA MEMORIA USATA PER I RECORD DI ATTIVAZIONE E NE HANNO TANTI QUANTE SONO LE CHIAMATE RICORSIVO.

IL COSTO SPAZIALE ASINTOTICO È

$$\Theta(n)$$

QUESITO 2

1) L'INSERIMENTO DELL'INTERO NELL'ARRAY HA COSTO COSTANTE. IL COSTO È DATO DALL'ORDINAMENTO.

PER SVOLGERE L'ORDINAMENTO POSSO UTILIZZARE IL MERGE SORT COSÌ DA AVERE IL CASO PIÙ PEGGIOR SEMPRE GARANTITO A $\Theta(n \log n)$. TALE ORDINAMENTO VIENE CHIAMATO n VOLTE.

IL COSTO ASINTOTICO TOTALE È QUINDI

$$\Theta(n + n \log n) \cdot \Theta(n)$$

\nearrow n CHIAMATE DI MERGE SORT

\downarrow n INSERIMENTI



SBAGLIATO:

IL MERGE SORT COSTA $n \log n$ NON $\log n$, SI VIDE FACENDO L'EQ. DI RICORRENZA.

IL COSTO SAREBBO QUINDI $n \cdot n \log n = n^2 \log n$

CONVIENE ALLORA SCANDIRE TUTTO L'ARRAY AD OGNI INSERIMENTO PER TROVARE LA POSIZIONE CORRETTA IN CUI INSERIRE L'INTERO. QUESTO INFATTI COSTERÀ SOLO

$$1+2+3+\dots+n = \Theta(n^2)$$

2) AD OGNI INSERIMENTO DI 1 INTERO SARÀ NECESSARIO FAR UNA CHIAMATA DI UP-HEAP CHE MI COSTERÀ PARI ALL'ALTEZZA DELL'ALBERO. NEL CASO PIÙ PEGGIOR MI COSTA

$$\Theta(n \log n)$$

\nearrow

VIENE DA $\log(n) + \log(n-1) + \log(n-2) + \dots \approx \log(n!) = n \log n$

3)

SIMMETRICA 1 3 5 6 7 8 12 15

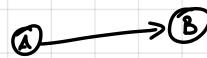
PRE ORDINE 6 3 15 12 7 8 15

POST ORDINE 15 3 8 7 15 12 6

QUESTO 4

1) PER RAPPRESENTARE IL PROBLEMA SI PUÒ USARE UN DAG (DIRECT ACYCLIC GRAPH).

Ogni nodo rappresenta una task da svolgere e gli archi che collegano i nodi rappresentano i vincoli di precedenza, ad esempio



Significa che prima di svolgere B devo aver svolto A.

Il grafo è quindi diretto, non pesato ed è aciclico.

Il grafo deve essere aciclico perché se avessesse cicli avrei delle contraddizioni, ad esempio



avrei che A precede B e B precede A

2) PER RISOLVERE IL PROBLEMA MI SERVE TROVARE UN ORDINAMENTO TOPOLOGICO.

Per fare ciò svolgo un **D-DFS** sul grafo, ogni volta che esco da un nodo significa che ho finito di visitare tutti gli archi uscenti da quel nodo e quindi tutte le task che ci sono dopo di lui allora inserisco in testa ad una lista tale nodo.

Faccio quindi lo stesso per ogni nodo, alla fine dell'elenco stampo la lista risultante la quale rappresenta un possibile ordinamento topologico.

CONVIENE FARE LO PSEUDO CODICE DELLA D-DFS PIÙ CHE QUESTA DESCRIZIONE:

VINCOLI (DAG) {

LINKED-LIST USTA_NODI;

UNNOD-LIST-NODE NODO_GRAFO: DAG → NODES → HEAD;

WHILE (NODO_GRAFO != NULL) {

IF (NODO_GRAFO È UNEXPLORERD)

D-DFS(USTA_NODI, NODO_GRAFO);

}

LINKED-LIST-NODE* NODO = USTA_NODI → HEAD;

WHILE (NODO != NULL) {

PRINTF(NODO);

NODO = NODO → NEXT;

}

}

D - DFS (USTA-NODI, NODO-GRAFO) {

NODO-GRAFO È EXPLORATO;

LINRAD-GST-NODE RDGRS = NODO-GRAFO \Rightarrow RDGRS \Rightarrow LINRAD;

WHILE (RDGRS != NULL) {

IF (NODO COURGATO DA RDGRS È UNEXPLORATO)

D-DFS (USTA-NODI, NODO COURGATO DA RDGRS);

}

AGGIUNGI A USTA-NODI NODO-GRAFO

}

► ESAME GIUGNO 2019

QUESTO 1

a) IL 1° CICLO IN RADDOPPIA COSTA $\Theta(z)$ IN DIMENSIONE DELL'INPUT INFATTI IL CICLO FA n ITERAZIONI COSTANTI SULL'INPUT IL QUALE È UN ARRAY, PERCIÒ LA DIMENSIONE DELL'INPUT È $z = cn \Rightarrow h = \Theta(z)$.

IL 2° CICLO FA $2n$ ITERAZIONI, AD OGNI ITERAZIONE CHIAMA INC. LA QUALE FA LA 1° VOLTA n PASSI, Poi $n-1$ PASSI, ... ALLORA IL COSTO È DATO DA $h + h-1 + h-2 + \dots = \Theta(n^2)$ E QUINDI L'INTERO CICLO CHE CHIAMA INC. COSTA

$$z = cn \Rightarrow h = \Theta(z) \Rightarrow \Theta(n^2) = \Theta(z^2) \text{ IN DIMENSIONE DELL'INPUT}$$

IN CONCLUSIONE SI HA

$$\Theta(z) + \Theta(z^2) = \Theta(z^2) \text{ CHE È IL COSTO DELL'ALGORITMO IN FUNZIONE DELL'INPUT.}$$

b) I PRIMI i . LENGTH ELEMENTI DI b SONO uguali AD a . I RESTANTI i . LENGTH ELEMENTI DI b SONO pari alla somma dei numeri nella prima metà di b da $i-1$. LENGTH AD i . LENGTH NON COMPRESA.

QUESTO 3

a) L'ALBERO DI VISITA SARÀ DI ALTEZZA $n-1$ INFATTI AVVIO LA DFS DA UN NODO QUALSIASI, DA TALE NODO PUÒ SOLO A VISITARE I RESTANTI $n-1$ NODI DEL GRAFO.
Avrà quindi la radice con $n-1$ FIGLI.

SBAGLIATO:

NON CONFONDERTI TRA VISITA IN PROFONDITÀ ED IN AMPIA

L'ALBERO DI VISITA SARÀ DI ALTEZZA $n-1$ INFATTI DA OGNI NODO VISITO 1 SOLO ALTRO NODO, QUINDI OGNI NODO AVRÀ 1 SOLO FIGLIO NELL'ALBERO DI VISITA.

b) NEL CASO DI UNA SEARCH ABBIANO CHE PER UNA TABELLA HASP IL COSTO È COSTANTE SE SI MANTIENE IL COEFFICIENTE DI CARICO $< \frac{1}{3}$. NEL CASO DI UN AVL INVECE IL COSTO È $\Theta(\log n)$.

PER LA RANGE QUERY, INVECE, NEL CASO DI UNA TABELLA HASP NON POSSO SFRUTTARE ALCUN ORDIMENTO, QUESTO VUOL DIRE CHE DEVO SCANDIRE TUTTA LA MAPPA PER DARE IN USCITA LA RANGE-QUERY. NEL CASO DEGLI AVL, INVECE, MI COSTA $\log(n)$ PER TROVARE k_i , E Poi HO IL COSTO DATO DALLA DIMENSIONE DELL'OUTPUT PARI A $O(\log n)$ COME DA SPERARCA.

QUINDI ABBIANO

• SEARCH \Rightarrow $\Theta(1) + \Theta(n) = \Theta(n)$ FACCIO m OPERAZIONI IN TOTALE

• AVL \Rightarrow $\Theta(\log n) + \Theta(\log n) = \Theta(\log n)$

CONVIENE QUINDI L'AVL.

c)

IMPORTANTE:

PER AVERE IL MINIMO IN TEMPO COSTANTE MI BASTA MANTENERE UN PUNTATORE AL NODO CON CHIAVE PIÙ PICCOLA DELL'AVL. QUANDO MI VIENE QUESTO IL MINIMO MI BASTA DARE IN OUTPUT IL NODO PUNTATO DA QUESTO PUNTATORE.

NEL CASO DI INSERIMENTO BISOGNA, QUINDI, FARE ATTENZIONE A MANTENERE CORRETAMENTE TALE PUNTATORE. PER FARLO CIÒ, AD OGNI INSERIMENTO SI VERIFICA SE IL NODO È PIÙ PICCOLO DEL MINIMO ATTUALE ED IN CASO AFFERMATIVO AGGIORNO IL PUNTATORE AL NUOVO NODO.

NEL CASO DI CANCELLAZIONE DEL MINIMO SI AGGIORNA IL PUNTATORE CON IL PADRE E SI RICREA IL NODO COME SI FA CON UN AVL, PER LA CANCELLAZIONE DI UNA CHIAVE CHE NON SIA IL MINIMO NON VOGLIO MODIFICHE.

SBAGLIATO, NON È DETTO CHE SIA IL MINIMO. DEVO FARE UNA RICERCA DEL MINIMO SULL'AVL CHE COSTA $\log n$, SONO DENTRO I COSTI DI HEAP.

QUESTO 4

```
INT MAX_DIFF(BST t){
```

```
    INT LEFT = -1;  
    INT RIGHT = -1;  
    INT RET = -1;  
    IF (t->LEFT != NULL)
```

```
        LEFT = MAX_DIFF(t->LEFT);
```

```
    IF (t->RIGHT != NULL)
```

```
        RIGHT = MAX_DIFF(t->RIGHT);
```

```
    IF (|t->KEY - t->RIGHT->KEY| > RIGHT)  
        RIGHT = |t->KEY - t->RIGHT->KEY|;
```

```
    ELSE IF (|t->KEY - t->LEFT->KEY| > LEFT)  
        LEFT = |t->KEY - t->LEFT->KEY|;
```

```
    IF (LEFT > RIGHT)
```

```
        RETURN LEFT;
```

```
    ELSE RETURN RIGHT;
```

```
}
```

SBAGLIATO:

NON FUNZIONA PER LA PARTE DESTRA, L'ALGORITMO CORRETO È:

```
int max_gap(BST t){  
    int prec[2];  
    prec[0] = -1;  
    prec[1] = -1;  
    max_gap_aux(t, prec)  
    return prec[1];  
}
```

```
void max_gap_aux(BST t, int* prec){  
    if(t == NULL)  
        return;  
    if(prec[0] == -1)  
        prec[0] = t->value;  
    max_gap_aux(t->left, prec);  
    if(|prec[0] - t->value| > prec[1])  
        prec[1] = |prec[0] - t->value|;  
    prec[0] = t->value;  
    max_gap_aux(t->right, prec);  
}
```

QUESTO 3

1)

All' $\left\lfloor \frac{n}{2} \right\rfloor + 1$ -esimo inserimento abbiamo, oltre al costo costante dell'inserimento, un costo dovuto alla mappazione dell'array per aumentarne la dimensione.

Il costo per aumentare la dimensione della hash-table è pari a $\Theta(n + \frac{n}{2} + 1) = \Theta(n)$. Infatti dovremo fare n inserimenti nella nuova tavola hash.

Il costo medio è dato da

$$\frac{\frac{n}{2} + n + \frac{n}{2} + 1}{n+1} = \Theta(1)$$

Infatti abbiamo $\frac{n}{2}$ operazioni con costo costante ed 1 operazione che ha costo $n + \frac{n}{2} + 1$, faccio cioè tanti inserimenti quante sono le entry della hash-table.

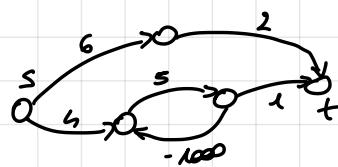
2)

Nel caso in cui inserisco nell'heap entry con priorità ordinate in ordine crescente allora avrò n inserimenti con costo costante. Infatti, per ogni inserimento, un heap non viene eseguito dato che le entry già presenti nell'heap sono più piccole di quella entry appena inserita come foglia.

In conclusione il costo totale delle n operazioni è

$$\Theta(n)$$

3)



QUESTO 4

1) IL GRAFO AMMETTE UN UNICO MST PERCHÉ NON HA ARCI CON PESI UGUALI, AVERE ARCI CON PESI UGUALI È INFATTI UNA CONDIZIONE NECESSARIA PER AVERE PIÙ DI UN MST.

QUESTO PERCHÉ, UTILIZZANDO AD ESEMPIO L'APPROCCIO DI PRIM E TARNIK, QUANDO DEVO INSERIRE UN NODO NELLA NUOVA POTREI AVERE CHE TALE NODO È COLLEGATO ALLA NUOVA ATTRAVERSO 2 ARCI, DI IDENTICO PESO, IN QUESTO CASO ESISTE PIÙ DI UN MST.

2)

L'ALGORITMO INIZIA CON UNA CODA DI PRIORITÀ CON TUTTI I NODI POSTI A DISTANZA ∞ MENTRE LA SORGENTE A DISTANZA 0. INIZIALMENTE $T = \emptyset$.

1° IT.

L'ALGORITMO ESTRARE b DALLA CODA ED IMPOSTA LE DISTANZE DI

$$d \rightarrow 20 \quad c \rightarrow 15 \quad d \rightarrow 3$$

$$T = \emptyset$$

2° IT.

L'ALGORITMO ESTRARE d DALLA CODA ED IMPOSTA

$$e \rightarrow 6 \quad f \rightarrow 10 \quad d \rightarrow 5 \quad c \rightarrow 7$$

$$T = \{b, d\}$$

3° IT.

L'ALGORITMO ESTRAE D DALLA CODA E NON LA MODIFICA

$e \rightarrow 6 \quad f \rightarrow 10 \quad c \rightarrow 7$

$T = \langle b, d \rangle, \langle cd, e \rangle$

4° IT.

L'ALGORITMO ESTRAE E DALLA CODA E IMPOSTA

$f \rightarrow 8 \quad c \rightarrow 7$

$T = \langle b, d \rangle, \langle cd, d \rangle, \langle cd, e \rangle$

5° IT.

L'ALGORITMO ESTRAE C DALLA CODA E NON LA MODIFICA

$f \rightarrow 8$

$T = \langle b, d \rangle, \langle cd, d \rangle, \langle cd, e \rangle, \langle cd, c \rangle$

6° IT.

ESTRAGGO f , CODA VUOTA

$T = \langle b, d \rangle, \langle cd, d \rangle, \langle cd, e \rangle, \langle cd, c \rangle, \langle cd, f \rangle$

► ESAME GENNAIO 2020

QUESTO 3

1) Gli alberi di Fibonacci sono tutti gli **ALBERI BINARI** bilanciati caratterizzati da fattore di bilanciamento 1 per tutti i nodi interni.
DEVO AVERE $|FDB|=1$, POSSO QUINDI AVERE $FDB=-1$

Gli alberi di Fibonacci sono importanti perché hanno l'altezza massima per un albero binario bilanciato con il minor numero di nodi per quell'altezza, questo permette di dimostrare che usando un AVL posso svolgere tutte le operazioni di BST con costo al massimo pari a $\log n$.

DATO UN ALBERO DI FIBONACCI DI $h=6$ ALLORA SE $F(6)$ È IL NUMERO DI NODI AD ALTEZZA 6 SI HA

$$\begin{aligned} F(6) &= F(5) + F(4) = 16 + 10 = 26 \\ F(5) &= F(4) + F(3) = 10 + 6 = 16 \\ F(4) &= F(3) + F(2) = 6 + 4 = 10 \\ F(3) &= F(2) + F(1) = 4 + 2 = 6 \\ F(2) &= 4 \\ F(1) &= 2 \end{aligned}$$

SBAGLIATO, IL NODO CORRETTO È IL SEGUENTE:

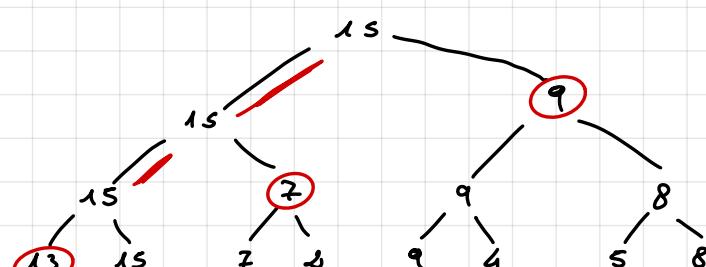
$$\begin{aligned} F(6) &= F(5) + F(4) + 1 = 20 + 12 + 1 = 33 \\ F(5) &= F(4) + F(3) + 1 = 12 + 7 + 1 = 20 \\ F(4) &= F(3) + F(2) + 1 = 7 + 4 + 1 = 12 \\ F(3) &= F(2) + F(1) + 1 = 4 + 2 + 1 = 7 \\ F(2) &= 4 \\ F(1) &= 2 \end{aligned}$$



Gli alberi di Fibonacci di una data altezza hanno tutti lo stesso numero di nodi perché va mantenuto il FDB 1 o -1 per tutti i nodi interni e, quindi, a parità di altezza ho sempre bisogno dello stesso numero di nodi.

2) IMPORTANTE

UN ESEMPIO DI TORNEO È



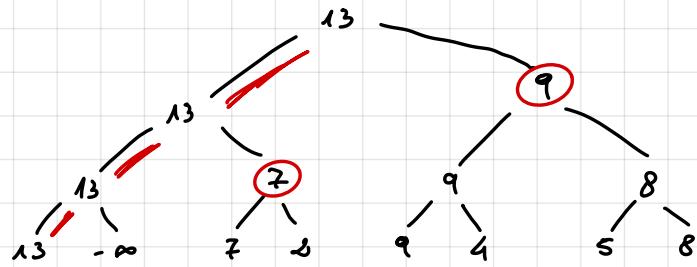
2° PIÙ GRANDE:

TENGO CONTO DI TUTTI QUELLI CHE HANNO PERSO CON IL 1° E SONO

13, 7, 9

VEDIAMO CHE I PERDENTI SONO 6, UNO PER OGNI LIVELLO. DEVO TROVARE QUINDI IL MASSIMO TRA I PERDENTI CHE MI COSTA 6.

PONIAMO QUINDI 13 PARI A -∞ IN MODO CHE 13 SALGA ALLA RADICE:



VEDO I PERDENTI CON 13 ED IL PIÙ GRANDE È 9, PONGO 13 A -∞ E FACCIO SULCA A. HO QUINDI TROVATO IL 3°.

ABBIANO CHE $b = \log n$ ESSENDO COMPLETO.

IL COSTO TOTALE PER ORDINARE È $n \log n$ DATO CHE FACCIO PER n VOLTE CIÒ CHE È STATO DETTO PRIMA.

3) Non posso determinarlo in tempo costante perché ho bisogno di ripetere una visita del grafo ciò richiede tempo $\Theta(n \cdot m)$.

SBAGLIATO:

È POSSIBILE DETERMINARLO IN TEMPO COSTANTE. UN GRAFO ACICICO E CONNESSO È CHIAMATO ALBERO ED HA $n-1$ ARCI.

$m < n-1$ NON PUÒ TESSELLA PERCHÉ È IL MINIMO NUMERO DI ARCI PER UN GRAFO CONNESSO.

$m = n-1$ È UN ALBERO QUINDI È ACICICO

$m > n-1$ SIGNIFICA CHE AGGIUNGO ARCI ALL'ALBERO, CAUSANDO UN CICLO

POSso QUINDI VERIFICARLO IN TEMPO COSTANTE

► Esercizi saltati su analisi di costo

• LUGLIO 2019

$$\begin{aligned} 1.1 \quad & \text{calcolo } g \text{ con } \frac{q}{2} \\ \left\{ \begin{aligned} C_f(q) &= c + C_g\left(\frac{q}{2}\right) \leq c + c'' + C_f\left(\frac{q}{2}\right) \\ C_f(1) &= c' \\ C_g(1) &= c'' \\ C_g(x) &= c'' + C_f(x+1) \rightarrow \text{pongo } x = \frac{q}{2} \Rightarrow C_g\left(\frac{q}{2}\right) = c'' + C_f\left(\frac{q}{2} + 1\right) = c'' + C_f\left(\frac{q}{2}\right) \end{aligned} \right. \end{aligned}$$

OTTENIAMO CHE

$$C_f(q) \leq c \log q$$

PERCHÉ OGNI VOLTA SI DIVIDE, CERCO $\frac{q}{2^k} = 1$

QUINDI $C_f \in O(\log q)$, MA ESPRESSO IN DIMENSIONE DELL'INPUT

$$z = |q| = \log_2 q \Rightarrow q = 2^z \quad \text{quindi } C_f \in O(z)$$

1.2

AD ESEMPIO PER $q=3$

1° IF FALLISCE
 2° IF FALLISCE
 CHIAMA $f(1)$
 CHIAMA $f(2)$
 1° IF FALLISCE
 2° IF FALLISCE
 CHIAMA $f(3) \rightarrow$ QUINDI LA COMPUTAZIONE NON TERMINA

► ESAME SETTEMBRE 2019

QUESTO 3

1. INSERISCO ENTRY CON LE SEGUENTI CHIAVI: 11, 22, 33, 44

ABBIANO $f(x)=0$ PER OGNI ENTRY. LA TAVOLA sarà QUINDI

0	11
1	22
2	33
3	44
4	-
	.
	:
	:
11	-

IL CLUSTERING PRIMARIO AVVIENE QUANDO HO IN INPUT CHIAVI CHE HANNO STESSO RISULTATO DALLA FUNZIONE DI HASH, QUESTO GENERA UNA COLLISIONE DI QUACK. SE RIUSCITA CON ESPLORAZIONE LINEARE, MI PORTA AD INSERIRE 24 ENTRY NELLA CELLA SUBITO SUCCESSIVA. CIÒ PORTA AD AUMENTARE LE PROBABILITÀ DI COLLUSIONE ED I COSTI DELLA SUA RISOLUZIONE.

IN QUESTO ESEMPIO SI VERIFICA PERCÙ INSERISCO TUTTE ENTRY CON PARI $f_i(x)$.

2. $O(n+k\log n)$

DATO L'ARRAY IN INPUT QUESTO PUÒ ESSERE TRASFORMATO IN HEAP ATTRAVERSO UNA COSTRUZIONE BOTTOM-UP CON COSTO $\Theta(n)$ CON L'ALGORITMO ARRAY TO HEAP, FATTO CIÒ FACCIO K RUOTONI DEL MINIMO CON COSTO $\log n$ QUINDI IL COSTO TOTALE È

$$\Theta(n+k\log n)$$

3. PRE-ORDINE

6 3 1 5 12 7 8 15

IN ORDINE

1 3 5 6 7 8 12 15

POST-ORDINE

1 5 3 8 7 15 12 6

QUESTO 4

1. IL GRAFO USATO È UN GRAFO ORIENTATO BIPARTITO.

AVREMO INOLTRE IL CLUSTER DI UTENTI ED IL CLUSTER DEI SERVIZI. DUE UTENTI NON POSSONO METTERSI LIKE TRA DI LORO E QUINDI NON CI SARANNO HAI ARCI FRA 2 UTENTI, LO STESSO VALE PER I SERVIZI.

È ORIENTATO PERCÙ È UN UTENTE CHE METTE LIKE AD UN SERVIZIO E NON ANCHE VICEVERSA. IL GRAFO NON AMMETTE CAPI PERCÙ UN UTENTE NON PUÒ METTERSI LIKE DA SOLO.

L'INPUT DELL'ALGORITMO SONO UN GRAFO E L'UTENTE DI CUI CONTARE GLI ANALOGI; L'OUTPUT È L'INTERO RAPPRESENTANTE IL NUMERO DI ANALOGI TROVATI.

2. CONSIDERANDO UN GRAFO RAPPRESENTATO DA LISTE DI ADIACENZA E CON LISTA PER UTENTI E LISTA PER SERVIZI.

conta Analoghi (Graph G, Utente μ) {

```
INT ESPLORATO = 0;
INT ANALOGHI = 0;
FOR NODO IN  $\mu$ .ADIACENTI DO {
    NODO. STATO = ESPLORATO;
    RSPCORTATI++;
}
```

```
FOR USER IN G. LISTA - UTENTI  $\neq \mu$  DO {
    INT TEMP = 0;
    FOR SERVICO IN USER. ADIACENTI DO {
        IF SERVICO. STATO == ESPLORATO
            TRNP++;
    }
    IF TEMP == RSPCORTATI
        ANALOGHI++;
}
```

}



► ESAME GENNAIO 2019

QUESTO 3

a) L'INSERIMENTO NELLA MOPPA AVVIENE ASSEGNUANDO UNA PSEUDO CHIAVE ALLA ENTRY IN INPUT IN FUNZIONE DELLA SUA CHIAVE USANDO UNA FUNZIONE DI HASH. TALE PSEUDO CHIAVE RAPPRESENTA LA POSIZIONE NELL'ARRAY IN CUI INSERIRE LA ENTRY.

SE IL FATTORE DI CARICO $\alpha = \frac{\text{NUM. DI ENTRY} + 1}{\text{DIM. HASH TABLE}} > \frac{1}{2}$ NEL MOMENTO IN CUI DEVO INSERIRE LA NUOVA ENTRY ALLORA SI RADOPPIA LA DIMENSIONE DELLA MOPPA PER POTER INSERIRE TUTTE LE ENTRY GIÀ PRESENTI PIÙ LA NUOVA.

NEL CASO IN CUI SI ABBIA UNA COLLISIONE ALLORA, PROSEGUENDO CON IL LINEAR PROBING, AVANZO DI 1 CELLA ALLA VOLTA FINO A CHE NON NE TROVO UNA LIBERA, QUI FACCIO L'INSERIMENTO.

OSSERVAZIONE: OPPURE LEGGO UNA ENTRY DUMMY

PER L'ELIMINAZIONE DELLA ENTRY VADO A VISITARE LA CELLA INDICATA DALLA FUNZIONE DI HASH IN BASE ALLA CHIAVE DELLA ENTRY DA ELIMINARE. A QUESTO PUNTO AVANZO DI 1 CELLA ALLA VOLTA FINCHÉ NON TROVO LA ENTRY DA ELIMINARE. SE LA TROVO RESTITUISCO LA ENTRY E NELLA TAVOLA LA SOSTITUISCO CON UNA ENTRY DUMMY, ALTRIMENTI SE NON LA TROVO ED INCOTRO UNA ENTRY DUMMY O CELLA VUOTA ALLORA TALE ENTRY NON ESISTE.

NO, MI FERMO SOLO QUANDO UNA CELLA VUOTA

b) UN INTEDIA UN κ ($1 \leq \kappa \leq n$)

AL INTEDIA PIÙ PICCOLI DELL'INSERIMENTO IN $O(n \log n)$ SE $n \in O(n)$ INSERIMENTO NON ORDINATO

FIRST_K (ARRAY a, INT κ) {

MIN-UKAP = ARR[0]. UKAP(a);

ARRAY FIRST-K[n];

FOR (i=0; i < n; i++) {

 FIRST-K[i] = RANK-K-MIN(MIN-UKAP);

}

RETURN FIRST-K;

c)

BOOL IS FOREST (Graph G) {

INT FOREST = 0;

FOR NODO IN G. LISTA - NODI DO {

 IF (NODO. STATO == UNEXPOSED) {

 INT CICLO = DFS(NODO);

 IF (CICLO == -1) RETURN FALSE;

 FOREST++;

 }

}

IF (FOREST > 1) RETURN TRUE;

}

```

INT DFS(NODO){
    NODO.STATUS = 'EXPLORER';
    FOR ADJ IN NODO.ADIACENTI DO {
        IF (ADJ.STATUS == 'UNEXPLORED')
            DFS(ADJ);
    }
    RETURN -1
}
RETURN 1;
}

```

QUESTO 4

PER RISOLVERE IL PROBLEMA POSSO APPLICARE UNA VISITA IN POST ORDINE

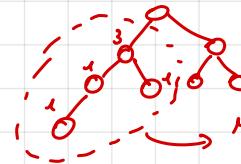
```

INT MAX_COMPLETE(BINARY_TREE t) {
    IF (t == NULL) RETURN 0;
    INT LEFT = MAX_COMPLETE(t->LEFT);
    INT RIGHT = MAX_COMPLETE(t->RIGHT);
    IF (LEFT == RIGHT) RETURN LEFT + RIGHT + 1;
    ELSE IF (LEFT > RIGHT) RETURN LEFT;
    ELSE RETURN RIGHT;
}

```

PERCÙ È UN CASO
ME LO CONSIDERA COME ALBERO COMPLETO, DEVO QUINDI AVERE UN NODO PER SEGNALETTICO.

}



ME LO CONSIDERA COME ALBERO COMPLETO, DEVO QUINDI AVERE UN NODO PER SEGNALETTICO.

► ESAHE FEBBRAIO 2020

QUESTO 3

1) TABUADA KASU $N=7 \Rightarrow$ RADDOPIO FINO A $\frac{7 \cdot 2^k}{2}$

IN INFILAMENTI CONSECUTIVI, QUANTO VOLTE DEHARMING?

DEVE SEMPRE VALERE $\frac{q}{N} < \frac{1}{2}$, OGNI VOLTA SI HA CHE N RADDOPIA QUANDO $\frac{q}{N} = \frac{1}{2}$ ALLORA NEL NOSTRO CASO, PARTENDO DA 7

$$\frac{q}{7} = \frac{1}{2} \rightarrow \frac{q}{7 \cdot 2} = \frac{1}{2} \rightarrow \frac{q}{7 \cdot 2 \cdot 2} = \frac{1}{2} \quad \text{SONO TUTTE LE VOLTE CHE RADDOPIO LA TAVOLA KASU E L'ULTIMA VOLTA È} \quad \frac{n}{7 \cdot 2^k} = \frac{1}{2} \Rightarrow n = \frac{7 \cdot 2^k}{2} \Rightarrow n = 7 \cdot 2^{k-1}$$

QUINDI $n = \log_2 \frac{n}{7} + 1$

2. NEL CASO IN CUI INSERIAMO IN ORDINE DECREScente ABBIANO I SEGUENTI COSTI

$$\log(1) + \log(2) + \dots + \log(n) = \Theta(\log(n!)) = \Theta(n \log n)$$

INFATI AD OGNI INSERIMENTO ANDIAMO AD INSERIRE UN NODO IL QUALE È IL PIÙ PICCOLO PRESENTE NELL'HEAP E QUINDI SARÀ IL NUOVO MINIMO.

NEL CASO SIA IN ORDINE CRESCENTE IL COSTO È $\Theta(n)$ INFATI OGNI INSERIMENTO È COSTANTE DATO CHE TUTTE LE ENTRY PRESENTI NELL'HEAP SONO PIÙ PICCOLE.

3) AMPIZZA: 6 3 12 1 5 7 15 8

PRE-ORDINE: 6 3 1 5 12 7 8 15

POST-ORDINE: 1 5 3 8 7 15 12 6

QUESTO 4

- I NODI RAPPRESENTANO LE LOCALITÀ DA RAGGIUNGERE CON LA FIBRA OTICA, GLI ARCI RAPPRESENTANO I TRATTI IN FIBRA OTICA E L'INSIEME DEGLI ARCI È CARATTERIZZATO DAGLI ARCI CHE HANNO PRETOSO MINIMO PER CONNETTERE TUTTI I NODI ED IL PRETO DI UN ARCO RAPPRESENTA LA LUNGHEZZA DI TALE TRATTO. IL GRAFO È INOLTRE SEMPRE E CONNESSO.

L'ALGORITMO BACKBONE HA COME INPUT UN GRAFO COMPLETO E CON LE CARATTERISTICHE DESCritte PRIMA, COME OUTPUT UN MINIMUM SPANNING TREE DI TALE GRAFO.

• PER L'ALGORITMO POSSIAMO USARE L'ALGORITMO DI PRIM CHE MI FORNISCE UN MST IL QUALE PRENDE IN INPUT IL GRADO DI CUI DETERMINARE IL MST E DA IN OUTPUT UN NUOVO GRAFO RAPPRESENTANTE IL MST

GRAPU BACKBONE (GRAPU G) {

 RETURN PRIM(G);

}

► ESERCIZI ANALISI DI COSTO SALTATI

• GENNAIO 2019

QUESTO 1

a) SIA n IL NUMERO DI ELEMENTI DELL'ARRAY

$$T(n) = T\left(\frac{n}{3}\right) + c = T\left(\frac{n}{3^k}\right) + kc \rightarrow \text{INFATTI NEL CASO PEGGIORE DIVIDO L'ARRAY SEMPRE IN 3 PARTI FINO A CHE END - START <= 2}$$

$$T\left(\frac{n}{3^k}\right) = T\left(\frac{n}{6}\right) + c$$

$$T(1) = c$$

$$\text{RAGGIUNGONO } \frac{n}{3^k} = 1 \text{ PER } k = \log_3 n \text{ ALLORA } T(n) = \Theta(\log n)$$

SIA $\gamma = |n|$ CIOÈ LA DIMENSIONE DELL'INPUT. SI HA $\gamma = |n| = 2 \cdot n \Rightarrow \Theta(\log n) = \Theta(\log \gamma)$ ESPRESO IN DIM. DELL'INPUT.

b)

RIGUARDANDO AI COSTI TEMPORALI, ASINTOTICAMENTE NON ABBIAMO DIFFERENZE INFATTI AVREMMO $\Theta(\log_2 n)$ CONTRO $\Theta(\log_3 n)$ I QUALI SI DIFFERENZIANO PER UNA COSTANTE. A LIVELLO SPAZIALE ABBIAMO CHE LO SPAZIO AGGIUNTIVO È DATO DALLE CHIAMATE RICORSIVE E QUINDI DAL RECORD DI ATTIVAZIONE. IN QUESTO CASO ABBIAMO $\log_3 n$ RECORD DI ATTIVAZIONI. MENTRE NEL CASO DI RICERCA BINARIA ABBIAMO $\log_2 n$ CHIAMATE RICORSIVE QUINDI ANCORA A LIVELLO SPAZIALE DIFFERISCE PER UNA COSTANTE.

• FEBBRAIO 2020

QUESTO 1

1) SIA n IL NUMERO DI ELEMENTI DELL'ARRAY.

ABBIAMO 2 CHIAMATE RICORSIVE DELLO STESSO ALGORITMO SU $\frac{n}{2}$, NEL CASO PEGGIORE TUTTO L'ARRAY È IN ORDINE CRESCENTE QUINDI

$$T(n) = 2T\left(\frac{n}{2}\right) + c = 2^k T\left(\frac{n}{2^k}\right) + kc$$

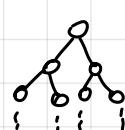
$$T\left(\frac{n}{2^k}\right) = 2T\left(\frac{n}{4}\right) + c$$

SI HA CHE $\frac{n}{2^k} = 1$ PER $k = \log_2 n$, ABBIAMO CHE PER TALE k L'ALGORITMO CONCLUDE E $T(1) = c$.

In conclusione $T(n) = \Theta(n)$, essendo $\gamma = |n| = d \cdot n$ ALLORA $\Theta(n) = \Theta(\gamma)$ IN DIMENSIONE DELL'INPUT.

2) IL COSTO SPAZIALE DELL'ALGORITMO È DATO DAL RECORD DI ATTIVAZIONE E DIPENDE, QUINDI, DIRETTAMENTE DAL NUMERO DI CHIAMATE RICORSIVE SVOLTE DATO CHE NON ABBIAMO COSTI AGGIUNTIVI.

PER OGNI CHIAMATA RICORSIVA ABBIAMO 2 ULTERIORI CHIAMATE, OTTENIAMO QUINDI UN ALBERO DI CHIAMATE COMPLETO



DI ALTEZZA MASSIMA $\log_2 n$. ABBIAMO 2^k CHIAMATE CON $k = \log_2 n$ QUINDI n CHIAMATE RICORSIVE. IL COSTO SPAZIALE È PERTOPO' $\Theta(n)$.

QUESTO 1

a) Abbiamo che la chiamata interna a funct1 dimette l'intero in input fino a quando $x \leq 1$. A quel punto la funzione interna ritorna x e viene fatta un'ulteriore chiamata a funct1 con $x \leq 1$ cui quindi termina subito. Abbiamo che facciamo chiamate interne fino a che $\frac{x}{2} = 1 \Rightarrow x = 2^k$ quindi faccio $\Theta(\log_2 x)$ chiamate interne e per ognuna ho una ulteriore chiamata a funct1.

Si ha, inoltre che $t = |x| = \log_2 x \Rightarrow x = 2^t \Rightarrow$ il costo dell'algoritmo in dimensione dell'input è $\Theta(\log_2 x) = \Theta(\log_2 t)$

b) Guardo funct2 fino a che $y \leq 1$, faccio quindi $y-1$ chiamate su funct2. Per ogni chiamata di funct2 vado a chiamare funct1 che mi costerà $\log_2 y$, andando a considerare tutte le chiamate di funct2 avrò

$$\log_2(1) + \log_2(y-1) + \log_2(y-2) + \dots = \log_2 y! \approx y \log_2 y$$

quindi il costo è dato dalle $y-1$ chiamate di funct2. Sia $t = |y| = \log_2 y \Rightarrow y = 2^t$ quindi il costo dell'algor. in dim. dell'input è

$$\Theta(2^t)$$

QUESTO 3

a) Si può eseguire la funzione più volte con dimensione dell'input x calcolando il tempo medio.

Eseguo con input $2x$ e calcolo il tempo medio. Eseguo con input $3x$ e calcolo il tempo medio.

Guardo quindi i tempi calcolati e guardo se c'è una crescita dovuta all'aumento della dimensione dell'input.

Ad esempio se con input di dimensione x il tempo è t_1 e con dimensione $2x$ il tempo è t_2 allora posso dire che il costo è lineare.

b) Per stabilire se un grafo diretto G è aciclico posso usare lo D-DFS: appena incontro un arco back il grafo ha un ciclo.

```
bool isAcyclic(Graph g){
    for(node in g){
        if(node.stato == UNEXPLORED){
            bool found_back = D-DFS(node);
            if (found_back == true)
                return false;
        }
    }
}

bool D-DFS(Node n){
    n.stato = EXPLORING;
    for(edge in n.out_edges){
        if(edge.destination.stato == UNEXPLORED){
            bool found_back = D-DFS(edge.destination);
            if(found_back == true)
                return true;
        }
        else if(edge.destination.stato == EXPLORING)
            return true;
    }
    n.stato = EXPLORED;
    return false;
}
```

c) Nella rappresentazione di alberi binari mediante array il valore dell'entry dell'albero viene inserita direttamente nella cella dell'array. La radice dell'albero si trova in posizione 0 ed i figli in posizione $2 \cdot 0 + 1$ (sinistro) e $2 \cdot 0 + 2$ (destro). Quindi per entry in posizione i si ha figlio sinistro in posizione $2 \cdot i + 1$ e $2 \cdot i + 2$.

Un prezzo dell'array è che, data una entry in posizione i , trovo il padre facendo $\frac{i}{2}$ oppure $\frac{i}{2} - 1$ se i è pari, accedo quindi al padre in tempo costante. Inoltre posso trasformare un albero binario in un heap con costo lineare.

Un difetto di questa rappresentazione è il costo spaziale che, nel caso di albero binario con albera massima, è esponenziale dovendo allocare celle anche per le entry non presenti. Un ulteriore difetto è che si ha un costo ammortizzato dovuto al dimensionamento dell'array e quindi al suo successivo raddoppiamento.

Questo tipo di rappresentazione è molto conveniente per alberi completi dato che il difetto sul giro spaziale decadrà. Per questo motivo è anche molto conveniente usare questa rappresentazione per la heap.

QUESTO 4

```
int k_childs(tree t, int k){  
    tree_node n = t -> radix  
    int ret = 0  
    linked_list bfs_visit <- lista collegata gestita come una coda  
    enqueue to bfs_visit <- n  
    while(bfs_visit non è vuota){  
        n <- estrai primo elemento di bfs_visit  
        tree_node child = n -> firstChild  
        if(child == NULL)  
            continue  
        int sons = 1  
        enqueue to bfs_visit <- child  
        while(child -> nextSibling != NULL){  
            child = child -> nextSibling  
            enqueue to bfs_visit <- child  
            sons++  
        }  
        if(sons == k)  
            ret++  
    }  
}
```

► ESAME APRILE 2019

QUESTO 1

a) Sia n il numero di elementi nell'array A .

Abbiamo che, nel caso peggiore, vengono svolte $\Theta(n)$ chiamate di S ed, ognuna di queste, fa una chiamata a P . Risulta che ogni chiamata di S ha un costo totale decrescente dovuto al crescere di i . In conclusione il costo asintotico è dato da

$$n + n-1 + \dots + 1 = \sum_{i=0}^n i = \Theta(n^2) \text{ che espresso in tm. dell'input è pari a } \Theta(z^2)$$

b) L'algoritmo descrive l'insertion sort. L'array viene divisa in 2 parti: una ordinata e una non ordinata. In questo caso partiamo dalla fine dell'array, a destra abbiamo l'array ordinato e a sinistra quello disordinato. Ad ogni chiamata l'array ord. viene incrementato di una cella e l'ordinamento viene ripristinato facendo scambi verso destra finché il nuovo numero non si trova in posizione corretta e cioè a destra di un numero maggiore.

QUESTO 3

d)

```
void array2heap(int[] a, int a_size){  
    for(int i = (a_size-1)/2; i >= 0; i--){  
        downheap(a, i);  
    }  
  
    void downheap(int[] a, int father){  
        int left_son = father*2+1;  
        int right_son = father*2+2;  
        if(a[left_son] > a[right_son] && a[father] < a[left_son]){  
            int temp = a[father];  
            a[father] = a[left_son];  
            a[left_son] = temp;  
            downheap(a, left_son);  
        }  
        else if(a[left_son] < a[right_son] && a[father] < a[right_son]){  
            int temp = a[father];  
            a[father] = a[right_son];  
            a[right_son] = temp;  
            downheap(a, right_son);  
        }  
    }  
}
```

b)

```
int completo_aux(BST t){  
    if(t == NULL)  
        return -1;  
    if(t->left_son != NULL && t->right_son == NULL || t->left_son == NULL && t->right_son != NULL)  
        return -2;  
    int left = completo_aux(t->left_son);  
    if(left == -2)  
        return -2;  
    int right = completo_aux(t->right_son);  
    if(right == -2)  
        return -2;  
    if(left != right)  
        return -2;  
    else return left+1;  
}  
  
bool completo(BST t){  
    if(completo_aux(t) > 0)  
        return true;  
    else return false;  
}
```

c) PREORDINE

D U E O C F P I K T B G N A L K

IN ORDINE

E U F C P I O D T G N B M A K L

POST-ORDINE

E F I P C O K N G B T K L A M D

d) GLI ALBERI DI FIBONACCI SONO BST BICONTATI (AVL) CHE HANNO FATTORE DI BILANCIAZMENTO 1, OPPURE -1, PER TUTTI I NODI INTERNI. QUESTO PORTA GLI ALBERI DI FIBONACCI AD AVERE L'ALTEZZA MASSIMA PER IL NUMERO DI NODI.
TUTTI GLI ALBERI SERVONO PER DEMONSTRARE CHE L'ALTEZZA DEGLI AVL È AL MASIMO $\Theta(\log n)$, QUESTO PORTA AD AVERE I COSTI DELL'OPERAZIONI DI BST PARI AL MASIMO A $\Theta(\log n)$

► ESAME GIUGNO 2018

QUESTO 3

a) AUCHE HA RAGIONE PERCHÉ CONSIDERANDO UN INSERTION SORT APPLICATO PIÙ VOLTE SU INPUT PICCOLI MI DÀRÀ UNA SOMMA DI COSTI PIÙ PICCOLI RISPETTO ALL'INSERT-SORT CHE MI DÀRÀ UNA SOMMA DI COSTI PIÙ GRANDI TESSENDO UN ALGORITMO PIÙ COMPLESSO E CON PIÙ PASSI.

c)

```
int numero_livelli(int i){  
    int res = 0  
    while(i != 0){  
        i = i/2  
        res++  
    }  
    return res+1  
}  
  
void copia_in_clone(heap H, heap clone, int i){  
    if(i>=H->size)  
        clone[i] = H[i]  
    copia_in_clone(H, clone, 2*i+1)  
    copia_in_clone(H, clone, 2*i+2)  
    return  
}  
  
heap clone(heap H, int i){  
    int n_livelli_clone = numero_livelli(i)  
    int n_livelli_real = numero_livelli(H->size - 1)  
    int numero_nodi = 2^(n_livelli_real - n_livelli)-1  
    int[] clone = malloc(sizeof(int)*numero_nodi)  
    copia_in_clone(H, clone, i)  
    return clone  
}
```

QUESTO 1

```

linked_list connectedComponent(graph* g, graph_node *s, int k){
    linked_list res = linked_list_new();
    int count = 0;
    connectedComponent_aux(s, k, &count, res);
    if(count == -1)
        return NULL;
    else return res;
}

void connectedComponent_aux(graph_node* node, int k, int* count, linked_list res){
    node->state = EXPLORING;
    *count++;
    if(*count > k){
        *count = -1;
        return;
    }
    linked_list_add(res, node->value);
    linked_list_node* edge = node->out_edges->head;
    while(edge != NULL){
        if(*count == -1)
            return;
        graph_node* r_edge = edge->value;
        if(r_edge->state == UNEXPLORED){
            connectedComponent_aux(r_edge, k, count, res);
        }
    }
    node->state = EXPLORED;
}

```

QUESTO 2

1) VISITA SIMMETRICA

7 3 8 1 9 4 10 0 11 5 2 6

VISITA POST-ORDINE

7 8 3 9 10 4 1 11 5 6 2 0

VISITA IN AMPIA

0 1 2 3 4 5 6 7 8 9 10 11

2) CODA DI PRORITÀ CON MIN POKER, LISTA ORDINATA

(1)(2)(3)(5)

Abbiamo che, per ogni inserimento dobbiamo scandire tutta la lista. Infatti, essendo in ordine crescente, ogni entry andrà inserito in ultima posizione e, non avendo accesso all'ultima posizione con un puntatore, mi costringe a scandire tutta la lista.

IL COSTO È QUINDI DATO DA

$$1+2+3+4+\dots+n-1 = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$$

3) CONSIDERO LA CODA DI PRORITÀ CON TUTTI I NODI CON PRORITÀ LA DISTANZA.

All'inizio dell'algoritmo ho

Q: e(0), f(∞), d(∞), b(∞), c(∞), a(∞)

T = ∅

1° IT.

ESTRAGO E E VISTO I NODI ADIACENTI, AGGIORNO QUINDI LE DISTANZE

Q: d(6), f(8), c(12), b(∞), a(∞)

T = ∅

2° IT.

ESTRAGGO d E VISITO TUTTI I NODI ADIACENTI, AGGIORNO LE DISTANZE

$Q = b(3), d(s), c(7), f(8)$

$T = \langle e, d \rangle$

3° IT.

ESTRAGGO b E VISITO TUTTI I NODI ADIACENTI, AGGIORNO LE DISTANZE

$Q = d(s), c(7), f(8)$

$T = \langle e, d \rangle, \langle d, b \rangle$

4° IT.

ESTRAGGO c E VISITO TUTTI I NODI ADIACENTI, AGGIORNO LE DISTANZE

$Q = c(7), f(8)$

$T = \langle e, d \rangle, \langle d, b \rangle, \langle d, a \rangle$

5° IT.

ESTRAGGO f E VISITO TUTTI I NODI ADIACENTI, AGGIORNO LE DISTANZE

$Q = \emptyset$

$T = \langle e, d \rangle, \langle d, b \rangle, \langle d, a \rangle, \langle c, d \rangle, \langle c, e \rangle$

6° IT.

ESTRAGGO f E VISITO TUTTI I NODI ADIACENTI

$Q = \emptyset$

$T = \langle e, d \rangle, \langle d, b \rangle, \langle d, a \rangle, \langle c, d \rangle, \langle c, e \rangle, \langle b, f \rangle$

IL MST È QUINDI DESCRITTO DA T

QUESTO 3

1)

Per risolvere il problema si può utilizzare l'algoritmo di Dijkstra che fornisce il SSSP del grafo. Tale SSSP imposta nei vertici diversi dalla sorgente la distanza minima tra tale nodo e la sorgente stessa. Supponiamo per semplicità che il grafo sia connesso.

```
distanza_media(graph G, graph_node* s){  
    SSSP(G, s);  
    int tot_dist = 0;  
    linked_list_node* nodes = G -> nodes -> head;  
    while(nodes != NULL){  
        graph_node* r_node = nodes -> value;  
        tot_dist += r_node -> dist_to_source;  
        nodes = nodes -> next;  
    }  
    return tot_dist/G -> num_nodi - 1;  
}
```

2)

```
max_medium_time(graph G, graph_node* s){  
    int max_med = 0;  
    graph_edge e;  
    for edge in G -> edges_list{  
        graph temp_G <- crea un clone grafo senza e  
        int temp = distanza_media(G, s)  
        if(temp > max_med){  
            max_med = temp  
            e = edge  
        }  
    }  
    return e  
}
```

QUESTO 1

```

linked_list* vicini_entro_k(graph* g, graph_node* s, int k){
    linked_list* bfs = linked_list_new();
    linked_list* res = linked_list_new();
    linked_list_insert_head(bfs, s);
    linked_list_insert_tail(bfs, NULL);
    for(i = 0; i <= k;){
        graph_node* n = linked_list_remove_head(bfs);
        if(n == NULL){
            i++;
            linked_list_insert_tail(bfs, NULL);
            continue;
        }
        linked_list_insert_tail(res, n);
        linked_list_node* edge = n -> out_edges -> head;
        while(edge != NULL){
            graph_node* r_edge = edge -> value;
            linked_list_insert_tail(bfs, r_edge);
            edge = edge -> next;
        }
    }
    return res;
}

```

QUESTO 3

1)

UN ESEMPIO DI AGGREGAZIONE È DATO DALL'INSERIMENTO DELLE ENTRY CON SEGUENTI PRIORITÀ:

11, 22, 33, 44 LA TABELLA VERRÀ SARÀ QUINDI

0	11
1	22
2	33
3	44

INFATTI ABBIAMO CHE SONO TUTTI MULTIPLI DI 11, LA FUNZIONE DI HASH DÀ QUINDI COME RISULTATO 0.

IL FENOMENO DI CLUSTER PRIMARIO CONSISTE NELL'AGGREGAZIONE IN SEQUENZA DI ENTRY CHE HANNO FATTO COLLUSIONE CON LA CELLA INIZIALE OPPURE UNA DELLE CELLE CHE FORMANO IL CLUSTER. TALE AGGREGAZIONE IN SEQUENZA AVVIENE NEL CASO IN CUI SI UTILIZZA L'ESPLORAZIONE LINEARE PER RISOLVERE LA COLLUSIONE. IN QUESTI CASI LA PROBABILITÀ DI COLLUSIONE AUMENTA ESSENDO DATA DALLA SOMMA DELLA PROBABILITÀ DI COLLUSIONE SU OGNI CELLA DEL CLUSTER.

NEL CASO DELL'ESEMPIO IL CLUSTERING PRIMARIO AVVIENE PERCHÉ LE PRIORITÀ SONO MULTIPLI DI 11, LA FUNZIONE DI HASH DA QUINDI COME RISULTATO 0.

2)

MAP. MIN LISTA NON ORD.

PER UN MIN-HEAP DI DIMENSIONE n CON LISTA NON ORDINATA IL COSTO DELLA RIMOZIONE DEL MINIMO È $\Theta(n)$. INFATTI DEVO SCANDIRE TUTTA LA LISTA PER TROVARE IL MINIMO.

PER FARLE RIMOZIONI SU UN MIN-HEAP DI n ENTRY HANNO I SEGUENTI COSTI:

$$n + n-1 + n-2 + n-3 + n-4 + \dots + 1 = \sum_{i=0}^n i = \frac{n(n+1)}{2} = \Theta(n^2)$$

3)

SIA Q LA CODA DI PRIORITÀ USATA DALL'ALGORITMO PER SCEGLIERE I NODI PIÙ VIGLI AL NUOVA.

1.

$Q = b(\infty), d(\infty), c(\infty), e(\infty), d(\infty), f(\infty)$

2.

ESTRAGGO b DA Q E VISITO GLI ARCII USCIENTI

$Q = c(2), d(3), d(\infty), e(\infty), f(\infty)$

3.
ESTRAGGO C DA Q E VISITO GLI ARCOLI USCENTI

Q = d(3), e(3), d(∞), f(∞)

4.
ESTRAGGO d DA Q E VISITO GLI ARCOLI USCENTI

Q = e(3), f(8), d(∞)

5.
ESTRAGGO e DA Q E VISITO GLI ARCOLI USCENTI

Q = f(5), d(∞)

6.
ESTRAGGO f DA Q E VISITO GLI ARCOLI USCENTI

Q = d(∞)

L'ALGORITMO TERMINA, A NON È RAGGIUNGIBILE DA b.

QUESTO 4

1)

```
to_connect(graph G){  
    vertice S <- scelgo a caso un grafo da G  
    priority queue Q <- inizializzo coda di priorità  
    linked_list T  
    linked_list RES  
    inserisco in Q la entry (0, (S, NULL))  
    for vertice V in G -> lista_nodi e n != S {  
        inserisco in Q la entry (+inf, (V, NULL))  
    }  
    while Q non è vuota{  
        (n, e) <- estrai il minimo da Q  
        inserisci e in T  
        for edge = (n, v) in n -> lista_out_edges{  
            if(edge -> weight < v -> dist){  
                v -> dist = edge -> weight  
                inserisci v -> dist nella chiave della entry con v in Q  
                inserisci edge al posto dell'arco precedente nella coppia con v in Q  
            }  
        }  
    }  
    return T  
}
```

2)

```
max_cost(graph G){  
    int cost_max = -1  
    linked_list res  
    for edge in G -> lista_out_edges{  
        G' <- creo copia di G senza arco edge  
        linked_list T = to_connect(G')  
        int c = 0  
        for e in T{  
            c += e -> weight  
        }  
        if c > cost_max{  
            cost_max = c  
            svuota res ed inserisci edge in res  
        }  
        else if(c == cost_max)  
            inserisci edge in res  
    }  
    return res  
}
```