

Algoritmi e Strutture Dati

a.a. 2022/23

Compito del 20/6/2023

Cognome: _____

Nome: _____

Matricola: _____

E-mail: _____

Parte II

(2.5 ore; ogni esercizio vale 6 punti)

Avvertenza: Si giustifichino tecnicamente tutte le risposte. In caso di discussioni poco formali o approssimative gli esercizi non verranno valutati pienamente.

1. Sia T un albero binario di ricerca (BST) avente n nodi.
 - a. Si scriva una funzione efficiente in C o C++
PTree Modify_key(PTree t, PNode x, int key) che modifica $x \rightarrow \text{key}$ con key e ritorna t se t è ancora un BST, `nullptr` altrimenti.
 - b. Valutare e giustificare la complessità della funzione.
 - c. Specificare il linguaggio di programmazione scelto e la definizione di PTree e PNode.
2. Sia A un vettore in cui ogni elemento contiene due campi: $A[i].value$ contiene un numero intero ed $A[i].color$ contiene un colore (BIANCO o NERO). Gli elementi di A sono ordinati in ordine crescente rispetto al campo *value*.
 - a. Si consideri il problema di ordinare gli elementi di A rispetto al campo *color* secondo l'ordinamento $\text{BIANCO} < \text{NERO}$, facendo in modo che gli elementi dello stesso colore siano ordinati rispetto al campo *value*. Si scriva lo pseudocodice di un algoritmo efficiente per risolvere il problema proposto. Si valuti e giustifichi la complessità.
 - b. Si consideri il problema di ordinare gli elementi di A rispetto al campo *value*, facendo in modo che gli elementi che hanno stesso *value* siano ordinati rispetto al campo *color* (sempre con la convenzione $\text{BIANCO} < \text{NERO}$). Si scriva lo pseudocodice di un algoritmo efficiente per risolvere il problema proposto. Si valuti e giustifichi la complessità.
3. Si stabilisca quale problema decisionale risolve il seguente algoritmo, che riceve in ingresso un grafo orientato $G = (V, E)$ e un intero positivo k (si assuma per semplicità che il grafo non contenga cappi):

MyAlgorithm(G, k)

1. $m = +\infty$
2. for each $u \in V[G]$
3. $m = \min \{m, \text{MyFunction}(G, u)\}$
4. if $m = k$ then
5. return TRUE
6. else
7. return FALSE

/* continua alla pagina successiva */

Algoritmi e Strutture Dati

a.a. 2022/23

Compito del 20/6/2023

Cognome: _____

Nome: _____

Matricola: _____

E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

Avvertenza: Si giustifichino tecnicamente tutte le risposte. In caso di discussioni poco formali o approssimative gli esercizi non verranno valutati pienamente.

1. Si risponda vero o falso alle seguenti domande, motivando brevemente le risposte:
 - a. la scelta dell'elemento pivot nella *Partition* non influenza la complessità asintotica di *QuickSort*.
 - b. *Partition* può restituire valori diversi se le vengono forniti elementi pivot diversi.
 - c. Dato un *maxHeap* incrementare il valore di una chiave di un elemento in posizione data ha costo $O(n)$.
 - d. *Counting Sort* è un algoritmo che ordina un qualunque array di n interi in tempo $O(n)$.
 2. Si scriva la formula di aggiornamento delle matrici dell'algoritmo di Floyd-Warshall e si mostri che, in assenza di cicli negativi, la diagonale principale delle matrici prodotte durante l'esecuzione dell'algoritmo è sempre nulla.
 3. Siano \mathcal{P} e \mathcal{Q} due problemi in NP con complessità $T_{\mathcal{P}}(n)$ e $T_{\mathcal{Q}}(n)$, rispettivamente, e si supponga che $\mathcal{P} \leq \mathcal{Q}$. Si stabilisca se le seguenti implicazioni sono vere o false:
 - a) $T_{\mathcal{Q}}(n) = O(n^3) \Rightarrow T_{\mathcal{P}}(n) = O(n^3)$
 - b) $T_{\mathcal{Q}}(n) = O(3^n) \Rightarrow T_{\mathcal{P}}(n) = O(3^n)$
 - c) $\mathcal{Q} \in \text{P} \Rightarrow \mathcal{P} \in \text{P}$
 - d) $\mathcal{P} \in \text{NPC} \Rightarrow \mathcal{Q} \in \text{NPC}$

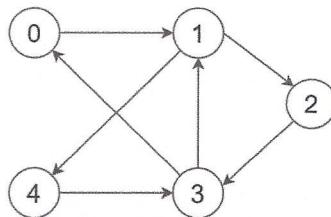
```

MyFunction(G, x)
1. for each  $u \in V[G] \setminus \{x\}$ 
2.    $d[u] = +\infty$ 
3.    $d[x] = 0$ 
4.    $Q = V[G]$                                 /* Q è una coda con priorità con campo chiave d */
5.   for  $i = 1$  to  $|V[G]|$ 
6.      $u = \text{ExtractMin}(Q)$ 
7.     for each  $v \in V[G]$ 
8.       if  $(u,v) \in E[G]$  and  $d[u] + 1 < d[v]$  then
9.          $d[v] = d[u] + 1$ 
10.     $a = +\infty$ 
11.    for each  $u \in V[G] \setminus \{x\}$ 
12.      if  $(u,x) \in E[G]$  then
13.         $a = \min \{a, d[u] + 1\}$ 
14.  return

```

Si dimostri la correttezza dell'algoritmo e si determini la sua complessità computazionale.

Cosa restituisce l'algoritmo in presenza del grafo sottostante con $k=2$, $k=3$ e $k=4$? Perché?



Si dica inoltre cosa restituisce MyFunction in corrispondenza dei 5 vertici del grafo.

4. Il sindaco di Venezia ha deciso di avviare un'operazione di ristrutturazione dei ponti della città al fine di tutelare la sicurezza dei suoi cittadini. Una limitata disponibilità finanziaria, tuttavia, impedisce di ristrutturare tutti i ponti e per questo impone le seguenti due condizioni:
 1. Deve essere possibile per i cittadini spostarsi da un punto all'altro della città attraversando *soltanto* ponti ristrutturati;
 2. Il costo complessivo della ristrutturazione deve essere il minore possibile.

Si formuli il problema in termini di ottimizzazione su grafi (indicando con precisione cosa rappresentano i nodi, gli archi e i pesi sugli archi) e si descriva un algoritmo per la sua risoluzione.

Si dimostri la correttezza dell'algoritmo utilizzato e si determini la sua complessità.

Infine, si simuli accuratamente la sua esecuzione su una versione semplificata del problema riguardante soltanto i sei sestieri di Venezia. A tal fine si utilizzi la tabella dei costi (in migliaia di euro) riportata di seguito (il simbolo “ ∞ ” significa che, tra due sestieri, non esiste un ponte):

	4 Cannaregio	2 Castello	3 Dorsoduro	4 Santa Croce	5 San Marco	6 San Polo
1 Cannaregio	--	56	∞	150	25	∞
2 Castello	56	--	∞	∞	30	∞
3 Dorsoduro	∞	∞	--	45	170	60
4 Santa Croce	150	∞	45	--	∞	40
5 San Marco	25	30	170	∞	--	240
6 San Polo	∞	∞	60	40	240	--

PART I

1) a) FALSO

La scelta del pivot è cruciale per la complessità di QuickSort

L'affermazione sarebbe vera solo se il pivot fosse scelto in modo randomizzato (complessità attesa $O(n \log n)$ indipendentemente dall'input).

b) VERO

L'algoritmo Partition riorganizza l'array in base al pivot:

- Se il pivot è 5 in $[3, 5, 1, 4]$, la partizione potrebbe essere $[3, 1, 4, 5]$
- Se il pivot è 3, il risultato sarebbe $[1, 3, 5, 4]$

L'output dipende dalla posizione finale del pivot scelto

c) FALSO

L'operazione di increase-key in un maxHeap ha costo $O(\log n)$ perché stiamo effettuando un cammino da un nodo alla radice che nel caso più lungo (il nodo è una foglia) percorre tutta la sua altezza

d) FALSO

Counting sort ha complessità $O(n+k)$ dove k è l'intervallo dei valori se non poniamo un limite superiore allora l'algoritmo non risulta più efficiente e la complessità potrebbe diventare anche $O(n^2)$.

$$2) d_{ij}^{(k)} = \min \left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right)$$

IPOTESI: Il grafo non contiene cicli negativi

TESI: Per ogni matrice $D^{(k)}$ prodotto dell'algoritmo, gli elementi sulla diagonale soddisfano

$$D_{ii}^{(k)} = 0 \text{ per ogni nodo } i.$$

DIMOSTRAZIONE

CASO BASE ($k=0$)

$D_{ii}^{(0)} = 0$ per definizione (distanza di un nodo da se stesso è zero)

PASSO INDUTTIVO ($k > 0$)

Supponiamo che $D_{ii}^{(k-1)} = 0$ per ogni i

Calcoliamo D_{ii}^K con la formula di aggiornamento: $D_{ii}^{(K)} = \min(D_{ii}^{(k-1)}, D_{ik}^{(k-1)} + D_{ki}^{(k-1)})$

Il primo termine è $D_{ii}^{(k-1)} = 0$ (per ipotesi induttiva)

Il secondo termine è $D_{ik}^{(k-1)} + D_{ki}^{(k-1)}$, che rappresenta la lunghezza di un ciclo passante per k :

- Se $D_{ik}^{(k-1)} + D_{ki}^{(k-1)} < 0$, il grafo avrebbe un ciclo negativo (contro l'ipotesi)
- Quindi $D_{ik}^{(k-1)} + D_{ki}^{(k-1)} \geq 0$

Il minimo tra 0 e un valore ≥ 0 è 0.

3) a) FALSO

La riduzione $P \leq_p Q$ implica che esiste una trasformazione polinomiale da P a Q , ma non garantisce che le complessità siano identiche

b) VERO

La funzione di riduzione è in tempo polinomiale. Se Q è insolubile in tempo esponenziale $O(3^m)$, allora anche P , grazie alla riduzione sarà insolubile in tempo esponenziale.

$T_P(n) = \text{tempo riduzione} + \text{tempo per risolvere } Q$

$$\begin{array}{ccc} \downarrow & & \downarrow \\ \text{POLINOMIALE} & & O(3^m) \end{array}$$

Componendo i due dà comunque una funzione $(3^P)^n$ per un qualche polinomio $p(n)$. Tuttavia è dominato da $O(3^m)$, quindi l'implicazione è vera.

c) VERO

Se $Q \in P$ e $P \leq Q$ allora P è insolubile in tempo polinomiale perché: possiamo trasformare ogni istanza di P in un'istanza di Q in tempo polinomiale, possiamo poi risolvere Q in tempo polinomiale.

Combihando due tempi polinomiali otteniamo un tempo polinomiale.

d) VERO

Il fatto che $P \in \text{NPC}$ e $P \leq_p Q$ NON IMPLICA che anche Q sia NPC. Perche' per essere NPC un problema deve:

1) Essere in NP (che lo è)

2) Essere NP-HARD, cioè ogni problema in NP si deve ridurre a Q

$$\begin{array}{c} P \in \text{NPC} \\ Q \in \text{NP PER IPOTESE:} \end{array} \xrightarrow{\left\{ \begin{array}{l} P \in \text{NP} \\ \forall E \in \text{NP}: E \leq_p P \end{array} \right.} P \leq_p Q \quad E \leq_p P \leq_p Q = \forall E \in \text{NP}: E \leq_p Q \quad \left\{ \begin{array}{l} Q \in \text{NP} \\ \forall E \in \text{NP}: E \leq_p Q \end{array} \right. \Rightarrow Q \in \text{NPC}$$

TRANSITIVITÀ

PART II

3) L'algoritmo verifica se il ciclo orientato più corto in G ha lunghezza esattamente k .

- My Function(G, x): Calcola la distanza minima da x a ogni altro nodo u , poi cerca il ciclo minimo che ritorna a x tramite archi (u, x)
- My Algorithm(G, k): Trova il ciclo minimo tra tutti i nodi e controlla se è uguale a k .

CORRETTEZZA

My Function:

- Initializza le distanze $d[u]$ da x (righe 1-3)
- Usa una coda a priorità per rilassare gli archi (righe 5-9), aggiornando $d[u]$ come $d[u] + 1$ (peso unitario)
- Trova la distanza minima per tornare a x tramite archi (u, x) (righe 10-13)

OUT PUT: Restituisce la lunghezza del ciclo minimo che parte e termina in x

My Algorithm:

- Calcola il ciclo minimo globale m (righe 2-3) e verifica se $m = k$ (righe 4-7)

COMPLESSITÀ COMPUTAZIONALE

My Function: $\mathcal{O}(|V|^2)$ My Algorithm: Chiama My Function $|V|$ volte = $\mathcal{O}(|V|^3)$

NODO X	CICLO MINIMALE	OUTPUT My Function
0	0-1-2-3-0	4
1	1-2-3-1	3
2	2-3-1-2	3
3	3-1-2-3	3
4	4-3-1-4	3

CONCUSSIONE

MyAlgorithm restituisce:

- FALSE per $k=2$ e $k=4$ (nessun ciclo di lunghezza 2; il ciclo di lunghezza 4 non è il minimo)
- TRUE per $k=3$ (cicli minimi di lunghezza 3 presenti)

MyFunction restituisce:

- 4 per il nodo 0
- 3 per i nodi 1, 2, 3, 4

4) Il problema può essere modellato come un GRAFO PESATO NON ORIENTATO (poiché i punti sono bidirezionali), dove:

- NODI: Rappresentano i sestieri di Venezia
- ARCHI: Rappresentano i ponti tra due sestieri
- PESO DEGLI ARCHI: Rappresenta il costo di ristrutturazione del ponte

OBIETTIVO

Trovare un sottoinsieme di archi tale che:

1) Il grafo rimanga连通的 (condizione 1: garantisce che sia possibile spostarsi tra qualsiasi coppia di sestieri usando solo ponti ristrutturati)

2) La somma dei pesi degli archi selezionati sia minima (condizione 2: minimizzazione dei costi)

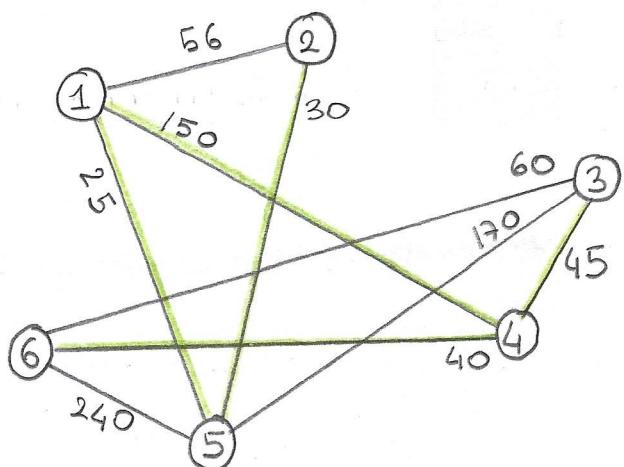
Questo corrisponde al problema del MST

L'algoritmo di Kruskal è ideale per trovare un MST. Ecco i passaggi:

- 1) Ordina tutti gli archi in ordine crescente di peso
- 2) Aggiungi gli archi uno per uno all'albero, evitando di formare cicli
- 3) Fermati quando tutti i nodi sono connessi

COMPLESSITÀ COMPUTAZIONALE

$$T(n) = O(|E| \log |E|)$$



$\{1,5\}, \{2,5\}, \{6,4\}, \{3,4\}, \{1,2\}, \{6,3\}, \{1,4\}, \{5,3\}, \{6,5\}$ ARCHI
 25 30 40 45 56 60 150 170 240
 ↑ ↑ ↑ ↑ Skip Skip ↑

$$25 + 30 + 40 + 45 + 150 = 290 \text{ migliaia di euro}$$

$\{1\} \{2\} \{3\} \{4\} \{5\} \{6\}$

$\{1,5\} \dots$

$\{1,2,5\} \dots$

$\{1,2,5\} \{4,6\} \{3\}$

$\{1,2,5\} \{3,4,6\}$

$\{1,2,3,4,5,6\}$

```

1) typedef struct Node {
    int key;
    Node* left;
    Node* right;
    Node* p;
    Node(int k, Node* padre, Node* sx=nullptr, Node* dx=nullptr) : key{k}, p{padre}, left{sx}, right{dx} {}
} * PNode;

typedef struct Tree {
    PNode root;
    Tree(PNode r=nullptr) : root{r}
}

// Funzione di supporto per verificare se un sottoalbero è un BST
bool isBSTUtil(PNode node, int min, int max) {
    if (node==nullptr)
        return true;
    if (node->key < min || node->key > max)
        return false;
    return isBSTUtil(node->left, min, node->key-1) && isBSTUtil(node->right, node->key+1, max);
}

PTree ModifyKey(PTree t, PNode x, int new_key) {
    if (t==nullptr || x==nullptr)
        return nullptr;
    int old_key = x->key;
    x->key = new_key; // Modifica temporanea
    bool valid = isBSTUtil(t->root, INT_MIN, INT_MAX);
    if (!valid) {
        x->key = old_key; // Ripristina la chiave originale
        return nullptr;
    }
    return t;
}

```

COMPLESSITÀ

MODIFICA DELLA CHIAVE : $O(1)$ VERIFICA DEL BST : Visita tutti i nodi dell'albero una volta $O(n)$ COSTO TOTALE $O(n)$

2)

a) Crea due liste vuote : bianchi e neri

Per ogni elemento x in A :Se $x.\text{color} == \text{BIANCO}$ Aggiungi x alla lista bianchi

Altrimenti :

Aggiungi x alla lista neri

Concatena bianchi e neri

Restituisce il risultato

COMPLESSITÀ

Scansione lineare per separare gli elementi $O(n)$ Concatenazione $O(1)$

Ordinamento per value è già garantito dall'input

b) Poiché l'array è già ordinato per value, possiamo sfruttare questa proprietà:

 $i=0$
while $i < \text{lunghezza}(A)$: $j=i$ while $j < \text{lunghezza}(A)$ AND $A[i].\text{value} == A[j].\text{value}$: $j++$ Ordina $A[i:j+1]$ per color (BIANCO < NERO) $i=j$

COMPLESSITÀ

Scansione dei gruppi $O(n)$

Ordinamento per colore solo quando necessario (in gruppi con stesso value)