

# Algoritmi e Strutture Dati

a.a. 2023/24

## Compito del 12/06/2024

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. Completare la seguente tabella indicando la complessità delle operazioni che si riferiscono a un dizionario di  $n$  elementi. Per l'operazione **Cancellazione** si assuma di essere sull'elemento  $x$  a cui si applica l'operazione.

	Ricerca	Minimo	Cancellazione	Massimo	Costruzione
Tabelle Hash con liste di collisione (caso medio)*	$O(1+\alpha)$	$O(m+m)$	$O(1+\alpha)$	$O(m+m)$	$\Theta(m)$
Max-Heap	$O(n)$	$O(n)$	$O(\log n)$	$\Theta(1)$	$\Theta(m)$

\*La Tabella Hash ha dimensione  $m$  e il fattore di carico è  $\alpha$  e le liste sono doppiamente concatenate.

2. Sia  $G$  un grafo non orientato con  $n$  vertici ed  $m$  archi. Si stabilisca se le seguenti affermazioni sono corrette:

- a) Se  $m \geq n - 1$  allora  $G$  è连通的
- b) Se  $m = n - 1$  allora  $G$  è un albero
- c) Se  $m \leq n - 1$  allora  $G$  è aciclico

In caso affermativo si fornisca una dimostrazione, altrimenti un controesempio.

3. Un nuovo algoritmo per determinare un albero di copertura minima in un grafo pesato ha complessità

$$T(m) = 3T\left(\frac{m}{3}\right) + \frac{m}{3}$$

dove  $m$  rappresenta il numero di archi del grafo in ingresso. Si dica se l'algoritmo in questione è preferibile o meno all'algoritmo di Kruskal e perché.

# Algoritmi e Strutture Dati

a.a. 2023/24

## Compito del 12/06/2024

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. Siano  $T_1$  e  $T_2$  due **alberi binari di ricerca** tali che tutte le chiavi memorizzate in  $T_1$  sono minori delle chiavi memorizzate in  $T_2$ .

a. Scrivere una funzione in C o in C++ PTree MergeBST(PTree T1, PTree T2) che presi in input  $T_1$  e  $T_2$  restituisca un albero binario di ricerca  $T$  le cui chiavi sono tutte e sole le chiavi contenute in  $T_1$  e  $T_2$ .

b. Determinare la complessità della funzione proposta nel caso migliore e nel caso peggiore.

La rappresentazione dell'albero binario di ricerca è tramite puntatori e utilizza i tipi PTree e PNode.

2. Un giovane amante del trekking vuole organizzare la prossima camminata e ha raccolto nel vettore *alture* l'altitudine di  $n$  punti che vorrebbe raggiungere. Dal punto in posizione  $0 \leq i < n$ , il giovane può raggiungere qualsiasi altro punto in posizione  $i < j < n$ , data una condizione: l'altitudine del punto in posizione  $j$  deve essere minore o uguale a quella del punto in posizione  $i$ . Una sequenza di punti consecutivi forma un percorso, la cui lunghezza è il numero dei punti.

a. fornire una caratterizzazione ricorsiva della lunghezza  $l_i$  di un percorso di lunghezza massima che abbia come ultimo punto *alture[i]*;

b. tradurre tale definizione in un algoritmo di programmazione dinamica con il metodo bottom-up che determina la lunghezza del percorso più lungo percorribile dal giovane.

c. valutare e giustificare la complessità dell'algoritmo.

Il prototipo della funzione è il seguente:

```
int percorso_piu_lungo(vector<int>& alture)
```

3. Sia  $G = (V, E)$  un grafo orientato e pesato, con funzione peso  $w : E \rightarrow \mathbb{R}$ , che non contenga cicli negativi e cappi. Si assuma che i vertici siano numerati da 1 a  $n$ , ovvero  $V = \{1, \dots, n\}$ , e sia  $W$  la matrice di dimensione  $n \times n$  definita come segue:

$$W[i,j] = \begin{cases} 0 & \text{se } i = j \\ w(i,j) & \text{se } i \neq j \text{ e } (i,j) \in E \\ +\infty & \text{se } i \neq j \text{ e } (i,j) \notin E \end{cases}$$

Si stabilisca quali problemi risolvono i due algoritmi riportati di seguito (che prendono in ingresso la matrice  $W$  associata al grafo), se ne dimostri la correttezza e se ne calcoli la complessità.

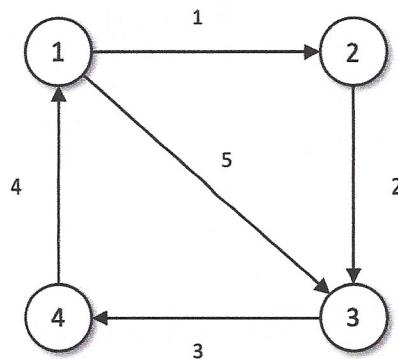
#### MyAlgorithm1(W)

1.  $n = \text{n.ro di righe di } W$
2.  $\text{alloca spazio per un vettore } D$   
di dimensione  $n$
3.  $D[1] = 0$
4.  $\text{for } i = 2 \text{ to } n$
5.    $D[i] = +\infty$
6.    $\text{for } k = 1 \text{ to } n$
7.      $\text{for } i = 1 \text{ to } n$
8.        $\text{for } j = 1 \text{ to } n$
9.          $D[j] = \min\{ D[j], D[i] + W[i,j] \}$
10.  $\text{return } D$

#### MyAlgorithm2(W)

1.  $n = \text{n.ro di righe di } W$
2.  $\text{alloca spazio per una matrice } D$   
di dimensione  $n \times n$
3.  $\text{for } i = 1 \text{ to } n$
4.    $\text{for } j = 1 \text{ to } n$
5.      $D[i,j] = W[i,j]$
6.    $\text{for } k = 1 \text{ to } n$
7.      $\text{for } i = 1 \text{ to } n$
8.        $\text{for } j = 1 \text{ to } n$
9.          $D[i,j] = \min\{ D[i,j], D[i,k] + D[k,j] \}$
10.  $\text{return } D$

Cosa restituiscono in uscita i due algoritmi in presenza del grafo seguente? (Scrivere esplicitamente il vettore  $D$  nel primo caso e la matrice  $D$  nel secondo. Non è necessario riportare la simulazione degli algoritmi.)



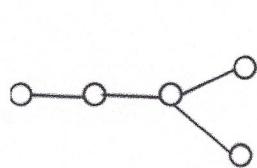
4. Il seguente algoritmo accetta in ingresso la matrice di adiacenza di un grafo non orientato e restituisce TRUE o FALSE. Qual è la funzione svolta dall'algoritmo e qual è la sua complessità?

```

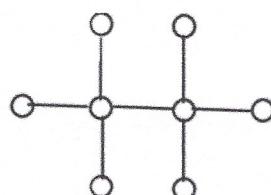
MyAlgorithm3(A)
1. n = n.ro di righe di A
2. crea due matrici n x n B e C
3. for i = 1 to n
4.   for j = 1 to n
5.     B[i,j] = 0
6.     for k = 1 to n
7.       B[i,j] = A[i,k]*A[k,j]
8.   for i = 1 to n
9.     for j = 1 to n
10.    C[i,j] = 0
11.    for k = 1 to n
12.      C[i,j] = B[i,k]*A[k,j]
13. sum = 0
14. for i = 1 to n
15.   sum = sum + C[i,i]
16. if sum = 0 then
17.   return TRUE
18. else
19.   return FALSE

```

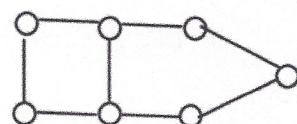
Cosa restituirebbe l'algoritmo nei casi seguenti? Perché?



$G_1$



$G_2$



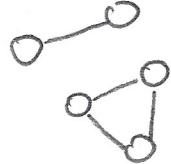
$G_3$

## PART I

2) Sia  $G$  un grafo non orientato con  $m$  vertici ed  $m$  archi

a) Se  $m \geq m-1$  allora  $G$  è连通的 (connesso)

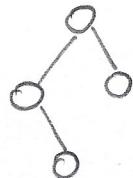
FALSO. Un grafo con  $m$  vertici e  $m=m-1$  archi potrebbe non essere connesso se le componenti sono disgiunte. Ad esempio,  $G$  con due componenti connesse.



b) Se  $m=m-1$  allora  $G$  è un albero.

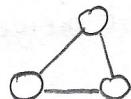
VERO. Un grafo con  $m$  vertici e  $m=m-1$  archi è un albero se e solo se è connesso.

La dimostrazione segue dal fatto che un albero è definito come un grafo aciclico connesso con esattamente  $m-1$  archi.



c) Se  $m \leq m-1$  allora  $G$  è aciclico

FALSO.



$$3) T(m) = 3T\left(\frac{m}{3}\right) + \frac{m}{3}$$

$$\alpha = 3 \quad b = 3 \quad f(n) = m/3 \quad g(n) = m^d = m^{\log_3 3} = m$$

$$f(n) \approx g(n)$$

$$f(n) = \Theta(m^d) \Rightarrow \frac{m}{3} = m$$

$$T(m) = (m^d \log m) = (m \log m)$$

L'algoritmo di Kruskal ha complessità  $O(m \log m)$ , che è più efficiente per grafi sparsi. Quindi il nuovo algoritmo è meno preferibile.

## PART II SOLUZIONE ERRATA VEDI ULTIMI FOGGI

1) struct PNode {

    int key;

    PNode\* left;

    PNode\* right;

    PNode(int val) : key(val), left(nullptr), right(nullptr) {}

};

typedef PNode\* PTree;

//Funzione per convertire un BST in un array ordinato

void BSTToArry(PTree root, int arr[], int & idx){

if (root){

    BSTToArry(root->left, arr, idx);

    arr[idx++] = root->key;

    BSTToArry(root->right, arr, idx);

}

//Funzione per effettuare il merge di due array ordinati

void mergeSortedArry(int arr1[], int size1, int arr2[], int size2, int merged[])

int i=0, j=0, k=0;

while (i < size1 && j < size2) {

if (arr1[i] < arr2[j])

    merged[k++] = arr1[i++];

else

    merged[k++] = arr2[j++];

}

while (i < size1) {

    merged[k++] = arr1[i++];

}

while (j < size2) {

    merged[k++] = arr2[j++];

}

}

//Funzione per costruire un BST bilanciato a partire da un array ordinato

```
PTree SortedArrayToBST(int arr[], int start, int end) {
```

```
    if (start > end)
```

```
        return nullptr;
```

```
    int mid = (start + end) / 2
```

```
    PTree root = new PNode(arr[mid]);
```

```
    root->left = SortedArrayToBST(arr, start, mid - 1);
```

```
    root->right = SortedArrayToBST(arr, mid + 1, end);
```

```
    return root;
```

```
}
```

//Funzione principale per il merge di due BST

```
PTree MergeBST(PTree T1, PTree T2) {
```

//1. Convertire T<sub>1</sub> e T<sub>2</sub> in array ordinati

```
int arr[100], arr2[100], merged[200]
```

```
int size1 = 0, size2 = 0;
```

```
BSTToArray(T1, arr1, size1);
```

```
BSTToArray(T2, arr2, size2);
```

//2. Effettuare il merge dei due array ordinati

```
MergeSortedArrays(arr1, size1, arr2, size2, merged);
```

//3. Costruire un BST bilanciato dal risultato del merge

```
return SortedArrayToBST(merged, 0, size1 + size2 - 1);
```

```
}
```

$T(n) = O(n+m)$ , dove n e m sono i nodi di T<sub>1</sub> e T<sub>2</sub> (merge lineare delle liste)

```

2) int percorsoPiùLungo (int altezza[], int n) {
    int dp[100];
    for (int i=0; i<n; i++) {
        dp[i] = 1;
    }
    for (int i=1; i<n; i++) {
        for (int j=0; j<i; j++) {
            if (altezza[j] <= altezza[i] && dp[i] < dp[j]+1) {
                dp[i] = dp[j]+1;
            }
        }
    }
    // Trovo il massimo valore in dp[]
    int max_length = dp[0];
    for (int i=1; i<n; i++) {
        if (dp[i] > max_length) {
            max_length = dp[i];
        }
    }
    return max_length;
}

```

$$T(n) = O(n^2) \text{ DUE cicli NIDIFICATI}$$

## 3) MyAlgorithm1

- Risolve il problema dei cammini minimi da sorgente singola in un grafo diretto pesato (senza cicli negativi)
- Complessità:  $O(n^3)$

## MyAlgorithm2

- Risolve il problema dei cammini minimi all-pairs (Floyd-Warshall)
- Complessità:  $O(n^3)$

0	1	3	6
1	2	3	4

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

4) L'algoritmo MyAlgorithm3 analizza la matrice di adiacenza A di un grafo non orientato per determinare se il grafo contiene TRIANGOLI (cicli di lunghezza 3). Restituisce TRUE se non ci sono triangoli, FALSE altrimenti.

Tutti e tre gli esempi ritornerebbero FALSE perché non sono presenti triangoli.

typedef struct Node {

④

ESAME 12/06/24

PART II ES. 1

int key;

Node\* left;

Node\* right;

Node\* p;

Node(int k, Node\* padre=nullptr, 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;

}

\* PTree;

void tree\_insert(PTree t, PNode z) {

/\* POST: inserisco il nodo z nell'albero t \*/

PNode y=nullptr;

PNode x=t->root;

while(x!=nullptr) {

y=x;

if(z->key < x->key)

x=x->left;

else

x=x->right;

}

z->p=y;

if(y==nullptr)

t->root=z;

else if(z->key < y->key)

y->left=z;

else

y->right=z;

}

SOLUZIONE SBAGLIATA

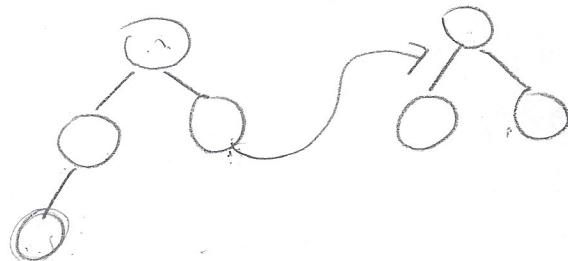
MA DISEGNI ESPPLICATIVI

O(h) dove h è l'altezza di t

```

void inOrder (PNode root, PTree newTree) {
    if (root != nullptr) {
        PNode newNode = new Node(root->key);
        inOrder (root->left, newTree);
        tree_insert (newTree, newNode);
        inOrder (root->right, newTree);
    }
}

```



```

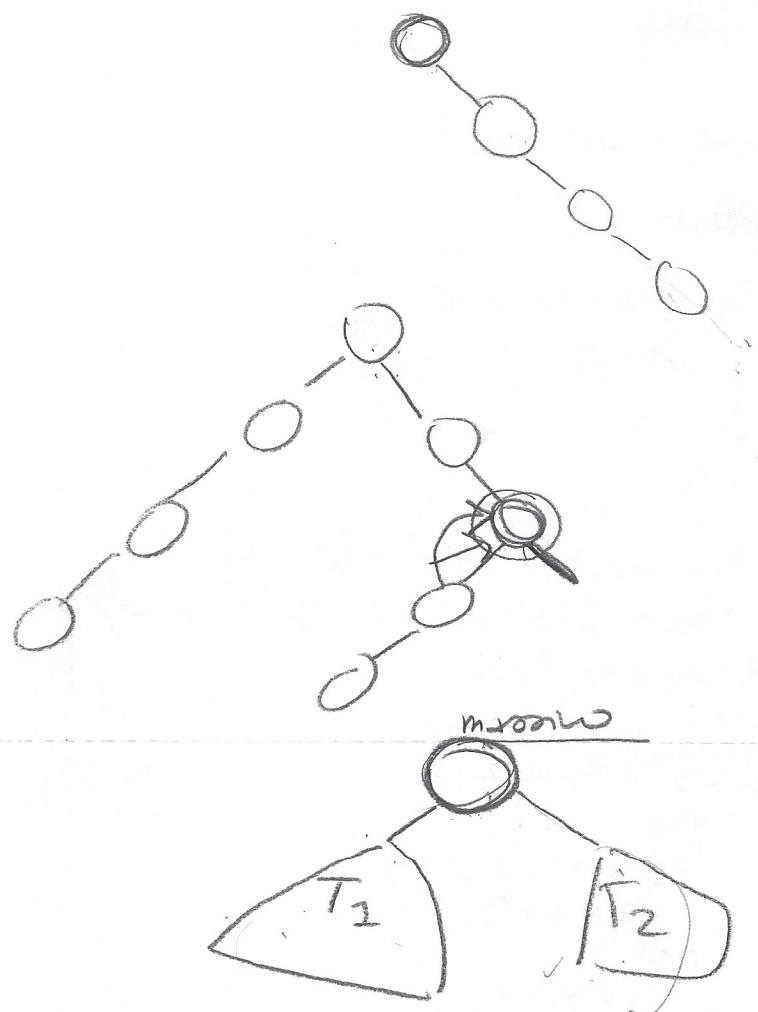
PTree MergeBST (PTree T1, PTree T2) {

```

```

    PTree newTree = new Tree();
    inOrder (T1->root, newTree);
    inOrder (T2->root, newTree);
    return newTree;
}

```



## PART II ES.1

## SOLUZIONE 1

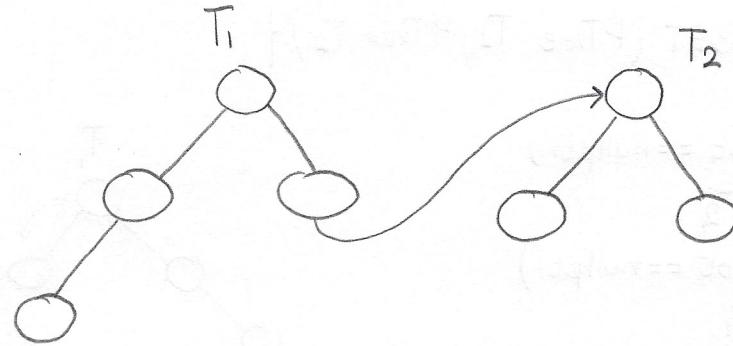
PTree merge BST 2 (PTree T1, PTree T2) {

```

PTree ris;
if (T1->root == nullptr)
    return T2
if (T2->root == nullptr)
    return T1

PNode iter;
iter = T1->root;
while (iter->right != nullptr)
    iter = iter->right;
iter->right = T2->root;
T2->root->p = iter;
delete T2;
return T1;
}

```



## SOLUZIONE 2

```

void transplant (PTree T, PNode u, PNode z) {
    if (u->p == nullptr)
        T->root = z;
    else {
        if (u->p->left == u)
            u->p->left = z;
        else
            u->p->right = z;
    }
    if (z != nullptr)
        z->p = u->p;
}

```

```

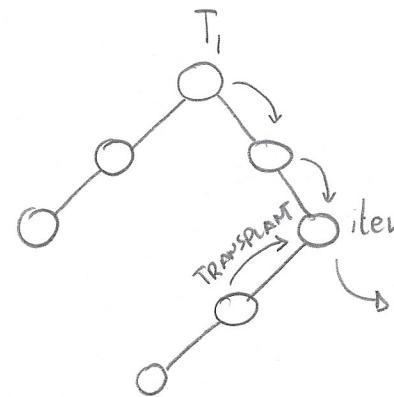
PTree mergeBST (PTree T1, PTree T2) {

```

```

    PTree nis;
    if (T1->root == nullptr)
        return T2;
    if (T2->root == nullptr)
        return T1;
    PNode iter;
    iter = T1->root;
    while (iter->right != nullptr)
        iter = iter->right;
    if (iter == T1->root)
        nis = T1;
    else {
        transplant (T1, iter, iter->left);
        nis = new Tree {iter};
        iter->p = nullptr;
        iter->left = T1->root;
        T1->root->p = iter;
        delete T1;
    }
    iter->right = T2->root;
    T2->root->p = iter;
    delete T2;
    return nis;
}

```



D ESTRAGO, DIVENTA LA RADICE  
DEL NUOVO ALBERO  
VADO A METTERE L'ALBERO T1  
ALLA SUA SINISTRA e L'ALBERO  
T2 ALLA SUA DESTRA.  
VIENE MANTENUTA LA PROPRIETÀ  
di BST

