Esame di

Algoritmi e strutture dati (parte di Fondamenti di informatica II 12 CFU)
Algoritmi e strutture dati (V.O., 5 CFU)
Algoritmi e strutture dati (Nettuno, 6 CFU)

Appello (straordinario) del 14-05-2020 – a.a. 2019-20 – Tempo: 4 ore – somma punti: 32

Alla fine di ogni domanda trovate qualche indizio sulla soluzione. Si noti che non si tratta di risposte esaurienti, ma soltanto bozze di risposta.

Istruzioni

Occorre rispondere a partire dal testo delle domande.

Avviso importante. La risposta ai quesiti di programmazione va data nel linguaggio scelto (C o Java), usando le interfacce messe a disposizione dal docente. Il programma va scritto usando l'editor interno di exam.net.

Mentre non ci aspettiamo che produciate codice compilabile, una parte del punteggio sarà comunque assegnata in base alla leggibilità e chiarezza del codice che scriverete, *a cominciare dall'indentazione*, oltre che rispetto ai consueti requisiti (aderenza alle specifiche ed efficienza della soluzione proposta). Inoltre, errori grossolani di sintassi o semantica subiranno penalizzazioni.

Quesito 1: Analisi algoritmo

Il seguente codice Java implementa il celebre algoritmo di Horner per il calcolo del valore di un polinomio in un punto.

```
    public static double horner(double p[], double x) {
        return horner(p, x, 0);
    }

    private static double horner(double p[], double x, int i) {
        if(i == p.length) return 0;
        return p[i] + x * horner(p, x, i+1);
    }
```

Si risponda ai seguenti quesiti:

1. Determinare il costo temporale asintotico dell'algoritmo `horner(double, double)`, in funzione della dimensione dell'input.

   Punteggio: [4/30]

2. Determinare il costo spaziale asintotico dell'algoritmo `horner(double, double)`, esplicitando anche lo spazio necessario alla gestione delle ricorsioni, in funzione della dimensione dell'input.

   Punteggio: [3/30]

Risposta. Per quanto riguarda la prima domanda, si noti che, in ciascuna invocazione abbiamo: i) un costo costante (al più c per una costante c opportuna), ii) un costo per un'invocazione ricorsiva su una porzione dell'array che la cui dimensione è di un'unita più piccola (questo perché nell'invocazione ricorsiva l'indice passa da i a i+1). Detto $T(n)$ il costo di caso peggiore rispetto alla dimensione iniziale n = p.length dell'array, avremo:

$$T(n) \leq c + T(n-1),$$

da cui segue immediatamente che $T(n) = O(n)$. Si noti che tale costo è lineare rispetto alla dimensione dell'input, perché quest'ultima sarà almeno pari a $n \cdot b$, dove b è il numero costante di bit con cui rappresentiamo un `double`.

Per quanto riguarda la seconda domanda, ogni invocazione richiede una quantità di memoria aggiuntiva costante, in quanto di `p` viene copiato il riferimento (che occupa uno spazio costante) e non l'array stesso. Il costo spaziale è quindi anch'esso lineare.

Quesito 2: Progetto algoritmi C/Java [soglia minima: 5/30]

In questo problema si fa riferimento a grafi semplici. I grafi sono rappresentati attraverso liste di adiacenza. A ciascun nodo u è dunque associata una lista collegata contenente $grado(u)$ elementi, ciascuno dei quali è il riferimento a uno dei vicini di u. I nodi sono rappresentati dalle classi/strutture `GraphNode/graph_node`, mentre il grafo è rappresentato dalle classi/strutture `Graph/graph`. La gestione delle liste di nodi deve essere effettuata mediante il tipo `linked_list` (C) o la classe `java.util.LinkedList` (Java); per la classe `LinkedList` i principali metodi per la gestione dovrebbero essere noti allo studente, ma alcuni di essi sono riportati in fondo a questo documento per comodità. Analogamente, in fondo a questo documento sono descritti i principali metodi per accedere al tipo `linked_list` in C. Le interfacce delle classi/moduli che implementano il grafo e i suoi nodi sono descritti nell'appendice di questo documento. Sono riportati i soli campi/metodi/funzioni utili allo svolgimento degli esercizi proposti.

Ciò premesso, rispondere ai seguenti punti:

1. Implementare il metodo `public static <V> void connectedComponents(Graph<V> g)` (funzione `void connectedComponents(graph *g)` in C) della classe `GraphServices` (modulo `graph_services` in C), che stampa a schermo le componenti connesse del grafo. Più precisamente, per ogni componente connessa, il metodo/funzione deve stampare la lista dei vertici che ne fanno parte (i loro valori), andando a capo per ogni nuova componente connessa. Ad esempio, per il grafo in Fig. 1 di seguito un possibile output del metodo `connecteComponents` (l'ordine con cui appaiono le componenti connesse, e quello con cui appaiono i relativi nodi, è irrilevante; in altre parole, è accettabile qualunque permutazione delle righe e, per ogni riga, qualunque permutazione dei suoi nodi):

   ```
   2 1 4 3 6 8 5 7
   14
   9 10
   12 11 13
   ```

   Il costo dell'algoritmo deve essere asintoticamente ottimale.

   Punteggio: [5/30]

Suggerimento. L'algoritmo che usate per implementare `connectedComponents` potrebbe semplificarvi le cose nella risposta al secondo quesito, ma non perdete troppo tempo a individuare il nesso. I due esercizi possono comunque essere risolti indipendentemente l'uno dall'altro.

2. Implementare il metodo/funzione `public static <V> void distances(Graph<V> g, GraphNode<V> s)` (funzione `void distances(graph *g, graph_node *s)`) della classe `GraphServices` (modulo `graph_services` in C) che, dato un nodo sorgente, stampa per ogni altro nodo appartenente alla componente connessa della sorgente, il *numero minimo di archi* che lo separano da essa. Ad esempio, per il grafo in Fig. 2, se il nodo sorgente fosse quello con etichetta 2, un output possibile di `distances` potrebbe essere:

```
2:0 1:1 4:1 3:2 6:3 8:4 5:4 7:5
```

`a:b` significa che il nodo `a` si trova a distanza `b` archi dalla sorgente. Si noti che in questo esempio l'ordine esatto in cui i nodi vengono stampati può dipendere dall'implementazione, mentre le distanze associate ai nodi devono essere quelle indicate.

Il costo dell'algoritmo deve essere asintoticamente ottimale

Punteggio: [5/30]

Suggerimenti. i) Si ricordi che trovare i percorsi minimi a partire da una sorgente è relativamente semplice in un grafo non pesato, lo è meno in un grafo pesato; ii) si consiglia di usare il campo `timestamp` della classe `GraphNode` (o del modulo `graph_node` in C, si veda più avanti) per memorizzare la distanza dalla sorgente.

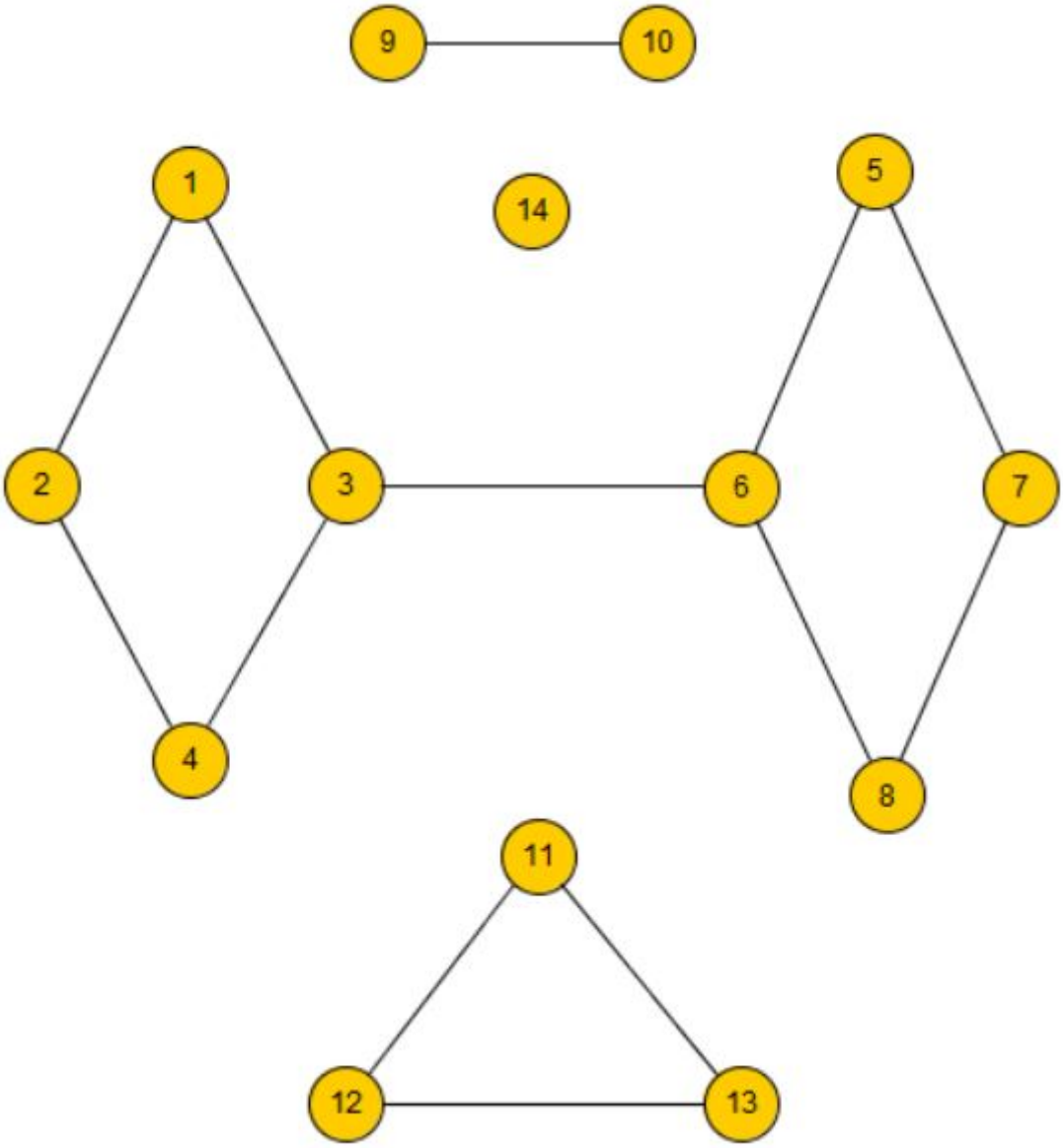

Fig. 1. Grafo con più componenti connesse.

Risposta. Per risolvere il primo quesito si può usare un qualsiasi algoritmo di visita, quindi anche la BFS. Quest'ultima permette anche di risolvere il problema dei cammini minimi in grafi non pesati (o con pesi uguali sugli archi). Quindi, usando una BFS per rispondere alla prima parte della domanda si sarebbe avuta (quasi) gratis la soluzione alla seconda domanda.

Quesito 3: Algoritmi

1. Si supponga di avere un array di interi di dimensione n, inizialmente vuoto. Si supponga che vengano inseriti in successione n interi, in modo tale da mantenere l'array sempre ordinato rispetto agli elementi che contiene. Si calcoli il il *costo complessivo degli n inserimenti nel caso peggiore*.

   Occorre offrire un' argomentazione quantitativa e convincente, non basta scrivere il risultato che si ritiene corretto.

   Punteggio: [3/30]

   Risposta. La chiave è nel fatto che occorre mantenere l'array ordinato ad ogni passo. Supponendo di aver già inserito i elementi nell'array e che questi siano ordinati, l'inserimento $i+1$-esimo richiederà nel caso peggiore lo spostamento degli i elementi inseriti in precedenza e avrà quindi costo (nel caso peggiore) $\Theta(i)$. Da qui si conclude

facilmente che il costo di caso peggiore è $O(n^2)$ (in effetti, anche $\Theta(n^2)$).

2. Si consideri un heap *minimale* (min-heap) a chiavi intere. Si supponga che vengano inserite, in successione, n chiavi in ordine *decrescente*. Calcolare il costo asintotico complessivo per la successione degli n inserimenti. *Occorre giustificare adeguatamente la risposta.*

   Punteggio: [3/30]

   Risposta. Siamo in presenza del caso peggiore per un heap minimale, perché a ogni inserimento occorrerà ripristinare la condizione di heap. Per l'i-esimo inserimento, il costo sarà $O(\log i)$. Sommando per $i = 1, \ldots, n$ avremo un costo complessivo al più pari a $c \sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n = cn \log n$.

3. Con riferimento alla Fig. 1, descrivere la sequenza con in cui vengono visitati i nodi dell'albero binario, per ciascuna delle visite *simmetrica*, in *pre-ordine* e in *post-ordine*.

   Punteggio: [3/30]

   Risposta. Simmetrica: 1 3 5 6 7 8 12 15; pre-ordine: 6 3 1 5 12 7 8 15; post-ordine: 1 5 3 8 7 15 12 6

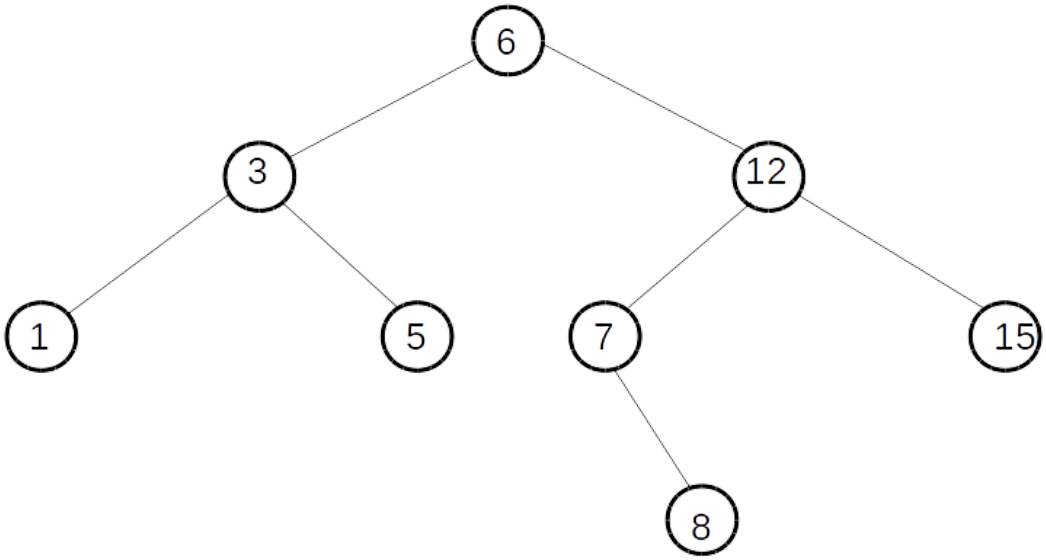

Figura 2. Stabilire l'ordine di visita simmetrica, in pre-ordine e in post-ordine.

Quesito 4:

Una società di consulenza deve realizzare un progetto di sviluppo estremamente complesso, che richiede lo svolgimento di numerose attività, tra molte delle quali sussistono vincoli di precedenza del tipo: "l'attività B non può iniziare prima del completamento dell'attività A", oppure "le attività C e D non possono iniziare prima del completamento dell'attività A" o ancora "l'attività Z non può avere inizio prima del completamento delle attività X, Y e W" giusto per fare qualche esempio. L'azienda deve implementare un algoritmo `vincoli` che permetta di stabilire un possibile ordine nel quale svolgere le attività, in modo da soddisfare *tutti i vincoli di precedenza*. Ciò premesso si risponda alle domande seguenti:

- Definire con precisione il grafo usato per rappresentare il problema, specificando cosa rappresentano i nodi e qual è l'insieme degli archi. Descrivere le proprietà principali del grafo, in particolare rispondendo alle domande seguenti: i) il grafo è diretto o indiretto? ii) è pesato? iii) il grafo può avere cicli (diretti se il grafo è diretto)? Se non può avere cicli, per quale motivo?

  Punteggio: [3/30]

- Si descriva (è sufficiente lo pseudo-codice o comunque una descrizione dettagliata) l'algoritmo `vincoli`. La descrizione può essere anche ad alto livello concettuale ed usare primitive non elementari, ad esempio corrispondenti ad algoritmi noti studiati nel corso. Descrizioni basate su (pseudo) codice C/Java sono comunque considerate accettabili.

  Punteggio: [3/30]

Risposta. Abbiamo un grafo diretto in cui ciascun vertice rappresenta un'attività e l'arco *diretto* (u, v) esiste se e solo se l'attività u deve essere completata prima che v abbia inizio. Il grafo non è ovviamente pesato e la presenza dei cicli implicherebbe l'impossibilità di soddisfare qualcuno dei vincoli, quindi il grafo deve essere un DAG.

Per quanto riguarda la seconda domanda, modellato in tal modo il problema si può usare un algoritmo efficiente per effettuare l'ordinamento topologico per risolverlo. Ad esempio, usando la DFS modificata vista a lezione possiamo risolvere il problema con un costo lineare corrispondente a una visita.

Appendice: interfacce dei moduli/classi

Di seguito sono descritti i campi/moduli/funzioni delle classi Java o moduli C che si suppone siano utilizzabili.

Interfacce Java

In Java si suppone sia suppone siano già implementate e disponibili le classi `GraphNode` e `Graph`, che rispettivamente implementano il generico nodo e un grafo non diretto. Ovviamente, in Java si assumono disponibili tutte le classi delle librerie standard, come ad esempio `java.util.LinkedList` ecc.

Classe GraphNode

```
public class GraphNode<V> implements Cloneable{
    public static enum Status {UNEXPLORED, EXPLORED, EXPLORING}

    protected V value; // Valore associato al nodo
    protected LinkedList<GraphNode<V>> outEdges; // Lista dei nodi adiacenti

    // keep track status
    protected Status state; // Stato del nodo
    protected int timestamp; // Campo intero utilizzabile per vari scopi

    @Override
    public String toString() {
        return "GraphNode [value=" + value + ", state=" + state + "]";
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return (GraphNode<V>) this;
    }
}
```

Metodi messi a disposizione dalla classe Graph. Di seguito si descrivono le interfacce dei metodi utili alla risoluzione degli esercizi e messi a disposizione dalla classe `Graph`.

```
// Restituiscee una lista di riferimenti ai nodi del grafo
public List<GraphNode<V>> getNodes();

// Restituisce una lista con i riferimenti dei vicini del nodo n
public List<GraphNode<V>> getNeighbors(GraphNode<V> n);
```

Metodi potenzialmente utili della classe LinkedList.

```
// Appende e alla fine della lista. Restituisce true
boolean add(E e);

// Rimuove e restituisce l'elemento in testa alla lista.
E remove(); // E indica il tipo generico dell'elemento
```

Scheletro della classe GraphServices

Di seguito, lo scheletro della classe `GraphServices` con le segnature dei metodi che essa contiene.

```
public class GraphServices<V>{

  public static <V> void connectedComponents(Graph<V> g) {
        /* DA IMPLEMENTARE */
  }

  public static <V> void distances(Graph<V> g, GraphNode<V> s) {
        /* DA IMPLEMENTARE */
  }

}
```

Interfacce C

graph.h (solo tipi principali)

```
#include "linked_list.h"
#include <stdio.h>

typedef enum {UNEXPLORED, EXPLORED, EXPLORING} STATUS;

/**
* Grafo semplice non diretto rappresentato mediante lista delle adiacenze.
*/

typedef struct graph {
    linked_list* nodes;     // lista di graph_node
    int n_nodes;
    int n_edges;
} graph;
```

```c
typedef struct graph_node {
    int key; // progressivo creazione, a partire da zero
    int timestamp;
    STATUS state;
    int value; // naturale letto da file
    linked_list* out_edges;  // lista di adiacenza
} graph_node;
```

linked_list.h (solo parte)

```c
typedef struct linked_list_node {
    void *value;
    struct linked_list_node *next;
    struct linked_list_node *pred;
} linked_list_node;

typedef struct linked_list {
    linked_list_node *head;
    linked_list_node *tail;
    int size;
} linked_list;

typedef struct linked_list_iterator linked_list_iterator;

/**********************
      linked_list
**********************/
/**
Crea una nuova lista.
*/
linked_list * linked_list_new();

/**
Aggiunge in testa alla lista ll, un nodo che punta a value.
*/
void linked_list_insert_head(linked_list* ll, void* value);

/**
Aggiunge in coda alla lista ll, un nodo che punta a value.
*/
void linked_list_insert_tail(linked_list* ll, void* value);

/**
Come linked_list_insert_tail(linked_list* ll, void* value).
*/
void linked_list_add(linked_list * ll, void * value);


/**
Aggiunge alla lista ll un nodo che punta a value, subito dopo predec
*/
void linked_list_insert_after(linked_list * ll, linked_list_node *predec, void
* value);


/**
```

```
Rimuove dalla lista ll il nodo in testa e ritorna il valore puntato da tale
nodo.
*/
void *linked_list_remove_head(linked_list* ll);

/**
Rimuove dalla lista ll il nodo in coda e ritorna il valore puntato da tale
nodo.
*/
void* linked_list_remove_tail(linked_list * ll);

/**
Ritorna un puntatore al valore puntato dal nodo in input.
*/
void *linked_list_node_getvalue(linked_list_node* node);

/**
Ritorna la dimensione della lista ll.
*/
int linked_list_size(linked_list *ll);


/**
Ritorna 1 se la linked list contiene value, 0 altrimenti.
*/
int linked_list_contains(linked_list *ll, void *value);


/**
Stampa a video una rappresentazione della lista ll.
*/
void linked_list_print(linked_list *ll);
```

Scheletro del modulo C graph_services

Di seguito lo scheletro del metodo `graph_services.c` e le segnature delle funzioni da implementare.

```
#include "graph.h"

void connectedComponents(graph *g) {
        /* DA IMPLEMENTARE */
}

void distances(graph *g, graph_node *s) {
        /* DA IMPLEMENTARE */
}
```