Laboratorio di Algoritmi e Strutture Dati
Anno Accademico 2005/06
Prof. Carlo Blundo

Raccolta di esercizi
Versione del 10 ottobre 2005

Alcuni di questi esercizi sono tratti dai libri di testo del corso ("*Data structures and algorithms*" di Goodrich e Tamassia e "*Introduction to Algorithms*" di Comer, Rivest….) Altri esercizi sono stati presi in vari siti di corsi di Algoritmi e Strutture Dati o Laboratorio di ASD di cui non ricordo i riferimenti esatti. Infine, alcuni esercizi sono stati proposti da me durante le varie edizioni del corso di LASD o durante le sessioni di esame. Se qualcuno, non vuole che il proprio esercizio venga inserito in questa raccolta oppure vuole che la fonte dell'esercizio venga citata opportunamente è pregato di contattarmi.

Carlo Blundo

Stack

1. Implementare l'interfaccia **Stack** (scrivere la classe **ArrayStack**) usando un array di tipo **Object** di lunghezza fissata (e.g., private static final int CAPACITY = 1024;). Se lo stack è pieno lanciare l'eccezione **StackFullException**.

2. Testare l'implementazione della classe **ArrayStack** scrivendo un programma **TestStack** che utilizzi tutti i metodi dell'interfaccia **Stack**.

3. Modificare la classe **ArrayStack** in modo che se l'array è pieno, allora venga allocato dello spazio sufficiente a contenere nuovi elementi e non venga lanciata l'eccezione **StackFullException.** Lo stack deve essere *estendibile.*

4. Usando la struttura dati **ArrayStack**, invertire una stringa ricevuta in input. Ad esempio, dando in input la stringa "casa" in output deve essere fornita la stringa "asac".

5. Scrivere una funzione Java che ricevuto in input uno stack restituisce un nuovo stack che corrisponde ad una copia palindroma dello stack passato per argomento. Lo stack originale non deve essere modificato. Ad esempio se lo stack passato in input è <1, 2, 4, 5>, allora lo stack ritornato come output della funzione sarà <1, 2, 4, 5, 5, 4, 2, 1>.

6. Si scriva una funzione Java **Stack inverti(Stack s)** che restituisca un nuovo stack ottenuto invertendo il contenuto di **s**. Si osservi che la funzione deve lasciare inalterato il contenuto di **s** alla fine dell'esecuzione (lo può modificare durante).

7. Scrivere la funzione Java **Stack StackSum(Stack s)** che restituisce un nuovo stack ottenuto a partire da s aggiungendo dopo ogni elemento la somma degli elementi che lo precedono in **s**. Ad esempio se lo stack **s** passato in input è <1, 2, 4, 5> (il valore 1 è al top dello stack), allora lo stack restituito in output sarà: <1, 1, 2, 3, 4, 7, 5, 12>. La funzione deve lasciare inalterato il contenuto di **s** alla fine dell'esecuzione (lo può modificare durante). Quale è la complessità della funzione proposta? Il problema può essere risolto in tempo lineare nel numero degli elementi nello stack **s**?

8. In un'espressione aritmetica sono ammesse indifferentemente le parentesi tonde "()" o le parentesi quadre "[]", purché siano bilanciate correttamente. Scrivere un funzione che, ricevendo in input una stringa, restituisce **true** se e solo se le parentesi sono bilanciate correttamente. Ad esempio le parentesi nelle seguenti espressioni sono bilanciate
   a. Rd(3e)d(d(i)d)
   b. C(f(f(f(g(t))d)iu)d)dfe

   Ma le parentesi nelle seguenti espressioni non sono bilanciate
   c. Rwe(fd)(fd(ffs)c
   d. Fdr(((d)(d)d(f(id(i))f)(d))

9. Aggiungere alla classe **ArrayStack** il metodo **multiPop**(int **num**) che restituisce un array contenente i primi **num** elementi presenti nello stack. Il metodo deve cancellare dallo stack gli elementi restituiti. Inoltre, deve lanciare l'eccezione **NotEnoughElements** se nello stack vi sono meno di **num** elementi.

10. Si aggiunga alla classe **ArrayStack** (implementa l'interfaccia **Stack** usando un array) il metodo **Stack clone()** che restituisce un nuovo stack avente lo stesso contenuto dello stack

su cui è invocato. Il metodo deve lasciare inalterato il contenuto dello stack su cui è invocato. Quale è la complessità del metodo proposto (giustificare la risposta).

11. L'interfaccia **DoppioStack,** che gestisce due stack contemporaneamente, consente le seguenti operazioni:
    - **push1():** inserimento nel primo stack.
    - **push2():** inserimento nel primo stack.
    - **pop1():** rimozione e restituzione dell'elemento nel primo stack.
    - **pop2():** rimozione e restituzione dell'elemento nel secondo stack.
    - **top1():** restituzione (senza rimozione) dell'elemento nel primo stack.
    - **top2():** restituzione (senza rimozione) dell'elemento nel secondo stack.

    Si implementi la classe **ArrayDoppioStack** usando un array di tipo **Object** di lunghezza fissata (e.g., private static final int CAPACITY = 1024;).

12. Testare l'implementazione della classe **ArrayDoppioStack** scrivendo un programma **TestArrayDoppioStack** che utilizzi tutti i metodi dell'interfaccia **DoppioStack**.

13. Aggiungere alla classe **ArrayStack** il metodo **String toString()** che restituisce una stringa rappresentante il contenuto dello stack su cui è invocato. Gli elementi dello stack dovranno essere separati da virgola e racchiusi da parentesi quadre. L'elemento al top dello stack deve essere preceduto dalla parola top. Ad esempio se lo stack è <1, 6, 4, 7>, allora il metodo deve restituire la stringa "[top 1, 6, 4, 7]". Il metodo deve lasciare inalterato il contenuto dello stack su cui è invocato. Quale è la complessità del metodo proposto (giustificare la risposta).

14. Si consideri la seguente interfaccia:

```
public interface ExtendedStack {
   // Restituisce il numero degli elementi nello stack
   public int size();

   // Restituisce true se lo stack è vuoto, false altrimenti
   public boolean isEmpty();

   // Restituisce l'elemento al top dello stack.
   // Se lo stack è vuoto  lancia l'eccezione StackEmptyException
   public Object top()  throws StackEmptyException;

   // Inserisce nello gli elementi nel vettore elements
   // elements[0] è il primo elemento inserito
   // elements[elements.length] è il primo elemento inserito
   public void mulitPush(Object[] elements);

   // Rimuove i primi n elementi al top dello stack
   // restituendoli in un array. Se lo stack è vuoto lancia
   // l'eccezione StackEmptyException. Se ci sono meno di n elementi
   // lancia l'eccezione NotEnoughElements
   public Object multiPop(int n) throws
                  StackEmptyException, NotEnoughElements;
}
```

    Implementare l'interfaccia **ExtendedStack**. La classe **ArrayExtendedStack** che implementa tale interfaccia deve far uso di una sola variabile istanza di tipo **ArrayStack**.

15. Testare l'implementazione della classe **ArrayExtendedStack** scrivendo un programma **TestArrayExtendedStack** che utilizzi tutti i metodi dell'interfaccia **ExetendedStack**.

16. Aggiungere alla classe **ArrayStack** il metodo **void removeAll(Object elem)** che cancella dallo stack tutte le occorrenze dell'elemento **elem**. Se lo stack è vuoto il metodo deve lanciare l'eccezione **StackEmptyException** (modificare opportunamente la firma del metodo). Quale è la complessità del metodo proposto? Giustificare la risposta.

17. Scrivere la funzione **void removeAllFromStack(Object elem, Stack s)** che cancella dallo stack **s** tutte le occorrenze dell'elemento **elem**. Se lo stack **s** è vuoto il metodo deve lanciare l'eccezione **StackEmptyException** (modificare opportunamente la firma del metodo). Quale è la complessità del metodo proposto? Giustificare la risposta.

18. Scrivere una funzione Java **Stack concat(Stack s1, Stack s2)**. La funzione restituisce la concatenazione dei due stack **s1** ed **s2** ricevuti in input (gli elementi di **s1** vengono inseriti prima degli elementi di **s2**). Se **s1** è <1, 6, 4, 7> mentre **s2** è <12, 16, 44, 72>, allora lo stack restituito sarà: <12, 16, 44, 72, 1, 6, 4, 7>La funzione deve lasciare inalterato il contenuto degli stack **s1** ed **s2** alla fine dell'esecuzione (lo può modificare durante). Quale è la complessità di tempo di tale programma?

Coda

1. Implementare l'interfaccia **Queue** (scrivere la classe **ArrayQueue**) usando un array di tipo Object di lunghezza fissata (e.g., public static final int CAPACITY = 1024;). Se la coda è piena lanciare l'eccezione **QueueFullException**.

2. Testare dell'implementazione della classe **ArrayQueue** scrivendo un programma **TestQueue** che utilizzi tutti i metodi dell'interfaccia **Queue**.

3. Modificare la classe **ArrayQueue** in modo che se la coda è piena, allora venga allocato dello spazio sufficiente a contenere nuovi elementi e non venga lanciata l'eccezione **QueueFullException**. (La coda deve essere estendibile).

4. Sia **ArrayQueue** la classe che implementa l'interfaccia **Queue** mediante un array. Aggiungere il metodo **void reverse()** che inverte il contenuto della coda.

5. Scrivere una funzione Java che ricevuto in input una coda q restituisce in output true se q è palindroma. Ad esempio, le code <1,2,3,2,1> <1,2,2,1>sono palindrome; mentre la coda <1,2,3,3,1> non lo è. La funzione deve lasciare inalterato il contenuto di **q** alla fine dell'esecuzione (lo può modificare durante). Quale è la complessità della funzione proposta?

6. Descrivere come implementare il TDA Stack usando due code. Quale è la complessità di tempo dei metodi pop() e push()?

7. La classe **StackQueue** implementa l'interfaccia **Stack** utilizzando come variabili istanza solo due code **q1** e **q2**. Supponendo che i valori contenuti nello stack sono conservati nella coda **q1**, una bozza della classe **StackQueue** è la seguente:

```
public class StackQueue implements Stack {

    Queue q1, q2;
    public boolean isEmpty() { return q1.isEmpty(); }
    public boolean size() { return q1.size(); }
        ...
```

}

completare l'implementazione della classe **StackQueue**.

8. Si aggiunga alla classe **ArrayQueue** (implementa l'interfaccia **Queue** usando un array) il metodo **Queue clone()** che restituisce un nuova coda avente lo stesso contenuto della coda su cui è invocato. Il metodo deve lasciare inalterato il contenuto della coda su cui è invocato. Quale è la complessità del metodo proposto (giustificare la risposta).

9. Aggiungere alla classe **ArrayQueue** il metodo **String toString()** che restituisce una stringa rappresentante il contenuto della coda su cui è invocato. Gli elementi della coda dovranno essere separati da virgola e racchiusi da parentesi quadre. L'elemento al rear della coda deve essere seguito dalla parola rear. L'elemento al front della coda deve essere preceduto dalla parola front.Ad esempio se la coda è <1, 6, 4, 7>, allora il metodo deve restituire la stringa "[front 1, 6, 4, 7 rear]". Il metodo deve lasciare inalterato il contenuto della coda su cui è invocato. Quale è la complessità del metodo proposto (giustificare la risposta).

10. Scrivere una funzione **EstraiDaCoda(Queue q, int n)** che ricevuto in input una coda **q** di interi (**Integer**) ed un numero intero (**Integer**) **n** restituisca una coda **r** contenente tutti gli elementi di **q** che sono multipli di n e maggiori di 27. Gli elementi di **r** devono comparire nello stesso ordine in cui comparivano in **q**. La funzione deve lasciare inalterato il contenuto di **q** alla fine dell'esecuzione (lo può modificare durante). Quale è la complessità della funzione proposta?

19. Aggiungere alla classe **ArrayQueue** il metodo **void removeAll(Object elem)** che cancella dalla coda tutte le occorrenze dell'elemento **elem**. Se la coda è vuoto il metodo deve lanciare l'eccezione **QueueEmptyException** (modificare opportunamente la firma del metodo). Quale è la complessità del metodo proposto? Giustificare la risposta.

20. Scrivere la funzione **void removeAllFromStack(Object elem, Queue q)** che cancella dalla coda **q** tutte le occorrenze dell'elemento **elem**. Se la coda **q** è vuota il metodo deve lanciare l'eccezione **QueueEmptyException** (modificare opportunamente la firma del metodo). Quale è la complessità del metodo proposto? Giustificare la risposta.

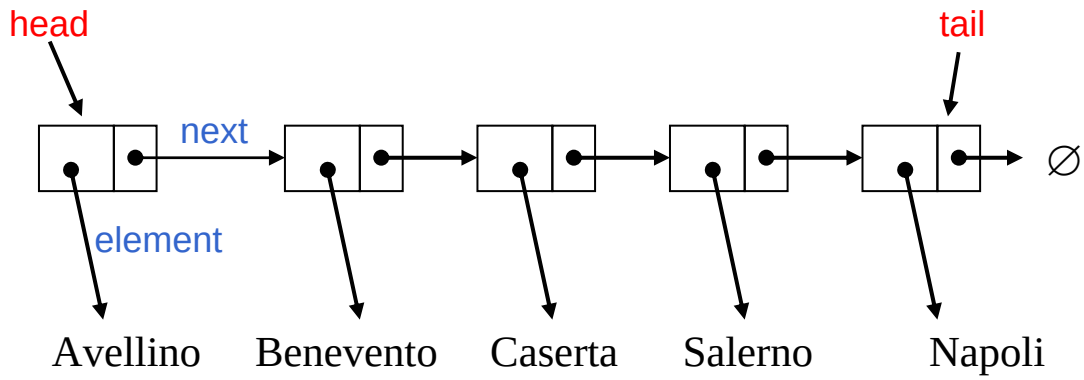
21. Scrivere una funzione Java **Queue concat(Queue q1, Queue q2)**. La funzione restituisce la concatenazione delle due code **q1** e **q2** ricevute in input (gli elementi di **q1** vengono prima degli elementi di **q2**). La funzione deve lasciare inalterato il contenuto delle code **q1** e **q2** alla fine dell'esecuzione (lo può modificare durante). Quale è la complessità di tempo di tale programma?

22. Scrivere una funzione Java **Queue concatNoDup(Queue q1, Queue q2)**. La funzione restituisce la concatenazione senza duplicati delle due code **q1** e **q2** ricevute in input (gli elementi di **q1** vengono prima degli elementi di **q2**). La concatenazione senza duplicati indica il fatto che se un elemento non può comparire due volte nella coda restituita in output. Per semplicità si può supporre che le code contengono numeri interi. La funzione deve lasciare inalterato il contenuto delle code **q1** e **q2** alla fine dell'esecuzione (lo può modificare durante). Quale è la complessità di tempo di tale programma?

23. Scrivere una funzione Java **Queue dammiPari(Queue q)** che ricevuta in input una coda **q** di elementi qualsiasi restituisce in output una coda contenente solo gli elementi in posizione pari. La funzione deve lasciare inalterato il contenuto della coda **q** alla fine dell'esecuzione (lo può modificare durante).

Lista Lincata

Una lista lincata non è il TDA Lista, ma una versione semplificata della struttura dati Lista. Una lista lincata nella sua forma più semplice è una collezione di **nodi** che formano un ordine lineare. Ogni **nodo** è un oggetto composto che memorizza un riferimento ad un elemento ed un riferimento ad un altro **nodo**. Il primo nodo della lista è detto *head*, l'ultimo *tail*. Ad esempio,



Una possibile implementazione del nodo è la seguente:

```
public class Node {
        private Object element;
        private Node next;

        public Node() {  this(null, null); }

         public Node(Object e, Node n) { element = e;  next = n; }

        public Object getElement() { return element;   }

        public Node getNext() {  return next; }

        public void setElement(Object newElem) { element = newElem;  }

        public void setNext(Node newNext) { next = newNext; }
}
```

1. Implementare il TDA Stack attraverso una lista lincata (scrivere la classe **LinkedStack**).

```
public class LinkedStack implements Stack {
        private Node top;    //riferimento al nodo head
        private int size;       // numero di elementi nello stack

        public LinkedStack()  // inizializza uno stack vuoto
        . . .
}
```

2. Aggiungere alla classe **LinkedStack** (implementa l'interfaccia **Stack** usando una lista lincata) un costruttore che ricevuto in input un array A di oggetti, inizializzi lo stack con gli elementi dell'array (il primo elemento da inserire nello stack è A[0], il secondo A[1], ...).

3. Implementare il TDA Queue attraverso una lista lincata (scrivere la classe **LinkedQueue**).

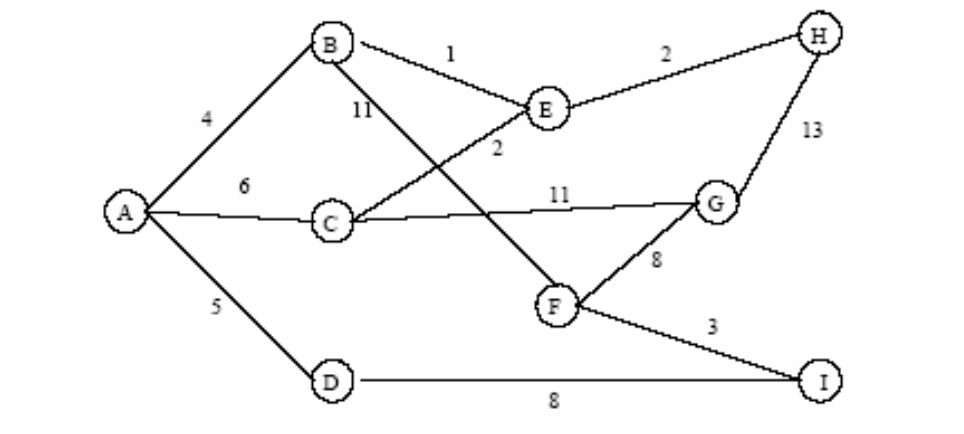
```
public class LinkedQueue implements Queue {
        private Node front;  // riferimento al nodo head
        private Node rear;       // riferimento al nodo tail
        private int size;        // numero di elementi nella coda
        public LinkedQueue()  // inizializza una coda vuota
        . . .
}
```

4. Sia **LinkedQueue** la classe che implementa l'interfaccia **Queue** mediante una lista lincata. Aggiungere il metodo **void reverse()** che inverte il contenuto della coda.

5. Modificare il metodo **int size()** della classe **LinkedQueue** in modo che conti in maniera ricorsiva i nodi presenti nella coda.

Grafi

1. Si consideri il grafo pesato in figura



   Rappresentare il grafo sia mediante liste delle adiacenze (nelle liste i nodi vengono inseriti in ordine lessicografico) sia mediante matrice delle adiacenze.

2. Sia $G = (V,E)$ un grafo orientato con $n = |V|$ vertici ed $m = |E|$ archi. Si supponga che puntatori e interi richiedano 32 bit. Indicare per quali valori di n ed m la lista delle adiacenze richiede meno memoria della matrice delle adiacenze.

3. Il trasposto di un grafo orientato $G = (V,E)$ è il grafo $G^T = (V,E^T)$ in cui $E^T = \{(u,v) : (v,u) \in E\}$. Descrivere due algoritmi efficienti per calcolare G^T con la rappresentazione con liste delle adiacenze e con la rappresentazione con matrice delle adiacenze.

4. Data una rappresentazione con liste di adiacenza di un grafo orientato, quanto tempo occorre per calcolare: il grado uscente di ogni vertice (numero di archi uscenti dal vertice); il grado entrante di ogni vertice (numero di archi entranti nel vertice). Cambia il tempo necessario per risolvere il problema precedente se il grafo non è orientato? Se si, come?

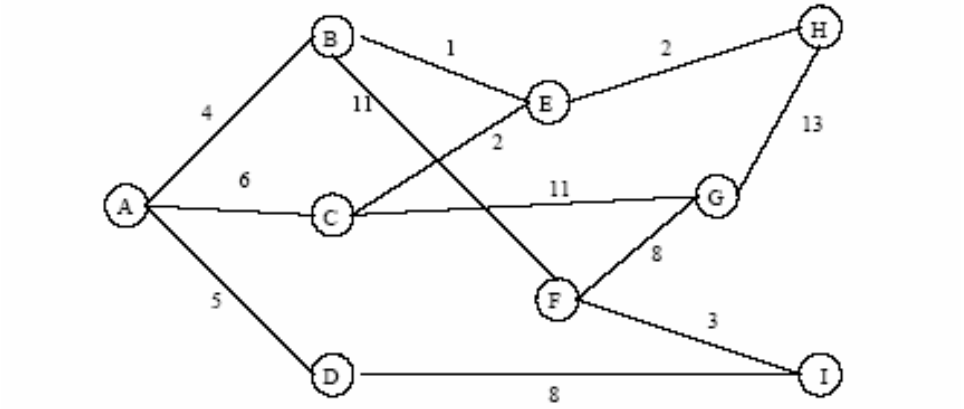
5. La matrice delle incidenze di un grafo orientato G=(V,E) è una matrice $B=(b_{i,j})$ di dimensione |V|X|E| tale che: $b_{i,j} = -1$ se l'arco j esce dal vertice i; $b_{i,j} = 1$ se l'arco j entra nel vertice i; 0 negli altri casi. Descrivere, che cosa rappresentano gli elementi del prodotto matriciale BB^T, dove B^T è la matrice trasposta di B.

6. Il quadrato di un grafo orientato G = (V,E) è il grafo $G^2 = (V,E^2)$ tale che $(u,w) \in E^2$ se e soltanto se, per qualche $v \in V$, si abbia $(u,v) \in E$ e $(v,w) \in E$. Ovvero G^2 contiene un arco fra u e w, se G contiene un cammino con due soli archi fra u e w. Descrivere degli algoritmi efficienti per calcolare G^2 da G, rappresentando G sia con le liste di adiacenze sia con la matrice di adiacenza. Analizzare i tempi di esecuzione degli algoritmi proposti.

7. Data una rappresentazione con liste di adiacenza di un multigrafo G = (V,E), descrivere un algoritmo con tempo O(V + E) per calcolare la rappresentazione con liste di adiacenza del grafo non orientato G' = (V,E'), dove E' è formato dagli archi di E con tutti gli archi multipli fra due vertici distinti sostituiti da un singolo arco e con tutti i cappi rimossi. Per la soluzione, è possibile utilizzare una struttura di appoggio.

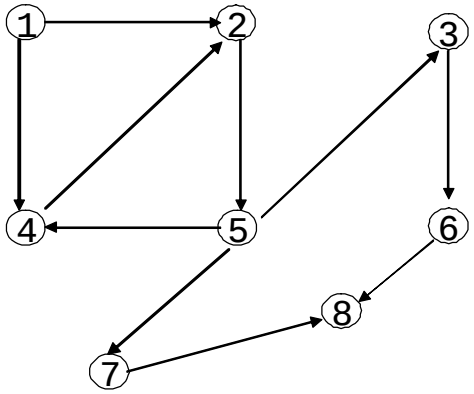
8. Scrivere lo pseudocodice dell'algoritmo di Dijkstra. Quale è la sua complessità? Giustificare la risposta.

9. Si consideri il grafo pesato in figura



Rappresentare il grafo mediante liste delle adiacenze (nelle liste i nodi vengono inseriti in ordine lessicografico) ed eseguire l'algoritmo di Dijkstra su di esso per il calcolo dei cammini minimi, a partire dal nodo A. Si riporti per ogni ciclo principale dell'algoritmo: il contenuto dell'insieme Q dei nodi non ancora visitati con le relative priorità; il contenuto dell'insieme R dei nodi già visitati; il vettore delle distanze.

10. Si consideri il grafo in figura



Rappresentare il grafo mediante lista delle adiacenze (in ciascuna lista i nodi vengono inseriti in ordine crescente) e mediante matrice delle incidenze.

11. Data la rappresentazione mediante lista delle adiacenze del grafo nell'esercizio al punto precedente, disegnare l'albero BFS avente come radice il nodo 1.

12. Data la rappresentazione mediante lista delle adiacenze del grafo nell'esercizio al punto precedente, disegnare l'albero DFS avente come radice il nodo 1.

13. Scrivere lo pseudocodice dell'algoritmo per la visita in profondità di un grafo (DFS). Commentare l'uso delle strutture dati utilizzate. Indicare, giustificando la risposta, la complessità dell'algoritmo.

14. Illustrare l'algoritmo di Prim per risolvere il problema del Minimo Albero Ricoprente. Servirsi dello pseudocodice e di commenti relativi alle strutture dati utilizzate. Quale è la complessità dell'algoritmo? Giustificare la risposta.

15. Provare il seguente teorema. Se si fa riferimento ad altri lemmi e/o corollari è necessario enunciarli.
    Teorema: Se f è un flusso in una rete di flusso G con sorgente s e destinazione t, allora le seguenti condizioni sono equivalenti:
    1. f è un massimo flusso in G
    2. G_f non contiene augmenting path
    3. |f|= c(S,T) per un taglio (S,T)

16. Descrivere un algoritmo per individuare il minimo taglio (S,T) di una rete di flusso. Fornire lo pseudo-codice dell'algoritmo proposto indicando la strategia adottata per computare i cammini aumentanti. Analizzare la complessità dell'algoritmo proposto.

17. Descrivere le quattro versioni del problema dei cammini minimi in un grafo orientato (SSSP, SDSP, SPSP, APSP).

18. Mostrare, come un algoritmo utilizzato per risolvere il problema SSSP possa essere utilizzato per risolvere gli altri

19. Scrivere lo pseudocodice dell'algoritmo per la visita in ampiezza di un grafo (BFS). Commentare l'uso delle strutture dati utilizzate. Indicare inoltre, giustificando la risposta, la complessità dell'algoritmo BFS.

20. Provare il seguente teorema. Se si fa riferimento ad altri lemmi e/o corollari è necessario enunciarli.
    > Sia G=(V,E) un grafo connesso non direzionato con una funzione di peso *w* a valori reali definita su E. Sia A un sottoinsieme di E che è incluso in un minimo albero ricoprente per G, sia (S, V-S) un qualsiasi taglio di G che rispetta A, e sia (u,v) un arco leggero (*light edge*) che attraversa (S, V-S). Allora, l'arco (u,v) è un arco sicuro (*safe edge*) per A.

21. Codificare l'algoritmo di Prim in Java (scrivere una funzione Java che implementi l'algoritmo). Il codice deve far riferimento alle interfacce illustrate durante il corso.

22. Codificare l'algoritmo di Dijkstra in Java (scrivere una funzione Java che implementi l'algoritmo). Il codice deve far riferimento alle interfacce illustrate durante il corso.

23. Codificare l'algoritmo BFS in Java (scrivere una funzione Java che implementi l'algoritmo). Il codice deve far riferimento alle interfacce illustrate durante il corso.

24. Codificare l'algoritmo DFS in Java (scrivere una funzione Java che implementi l'algoritmo). Il codice deve far riferimento alle interfacce illustrate durante il corso.