Progetti di Algoritmi e Strutture Dati

Università di Bologna, corso di laurea in Informatica per il Management

Anno Accademico 2022/2023, sessione estiva

Ultimo aggiornamento 02/05/2023

Istruzioni

Il progetto consiste in quattro esercizi di programmazione da realizzare in Java. Lo svolgimento del progetto è obbligatorio per poter sostenere l'orale nella sessione cui il progetto si riferisce.

I progetti dovranno essere consegnati entro le ore 23:59 del 03/06/2023. La prova orale potrà essere sostenuta in uno dei tre appelli della sessione estiva (è obbligatoria l'iscrizione tramite AlmaEsami).

L'esito dell'esame dipende sia dalla correttezza ed efficienza dei programmi consegnati, sia dal risultato della discussione orale, durante la quale verrà verificata la conoscenza della teoria spiegata in tutto il corso (quindi non solo quella necessaria allo svolgimento dei progetti). L'orale è importante: **una discussione insufficiente comporterà il non superamento della prova**.

Modalità di svolgimento dei progetti

I progetti devono essere **esclusivamente frutto del lavoro individuale di chi li consegna; è vietato discutere i progetti e le soluzioni con altri** (sia che si tratti di studenti del corso o agenti terzi). La similitudine tra progetti verrà verificata con strumenti automatici e, se confermata, comporterà l'immediato annullamento della prova per TUTTI gli studenti coinvolti senza ulteriori valutazioni dei progetti.

È consentito, salvo dove espressamente vietato nel testo dell'esercizio, l'uso di algoritmi e strutture dati definite nella libreria standard Java, nonché di codice messo a disposizione dai docenti sulla pagina del corso o sulla piattaforma "Virtuale"; è responsabilità di ciascuno verificare che il codice sia corretto (anche e soprattutto quello fornito dai docenti!). **Non è consentito fare uso di altro codice, anche se liberamente disponibile in rete**.

I programmi devono essere realizzati come applicazioni a riga di comando. Ciascun esercizio deve essere implementato in un singolo file sorgente chiamato `EsercizioN.java`, (`Esercizio1.java`, `Esercizio2.java` eccetera). Il file deve avere una classe pubblica chiamata `EsercizioN`, contenente il metodo statico main(); altre classi, se necessarie, possono essere definite all'interno dello stesso file (ovviamente non potranno essere `pubblic`). I programmi **non devono specificare il package** (quindi **non devono contenere** l'intestazione "`package Esercizio1;`" o simili).

I programmi devono iniziare con un blocco di commento contenente nome, cognome, numero di matricola e indirizzo mail (@studio.unibo.it) dell'autore. Nel commento iniziale è possibile indicare per iscritto eventuali informazioni utili alla valutazione del programma (ad esempio, considerazioni sull'uso di strutture dati particolari, costi asintotici eccetera).

I programmi verranno compilati dalla riga di comando utilizzando Java 11 (OpenJDK 11) con il comando:

```
javac EsercizioN.java
```

ed eseguiti sempre dalla riga di comando con

`java -cp . EsercizioN` *`eventuali_parametri_di_input`*

I programmi non devono richiedere nessun input ulteriore da parte dell'utente. Il risultato deve essere stampato a video rispettando **scrupolosamente** il formato indicato in questo documento, perché i programmi subiranno una prima fase di controlli semiautomatici. **Non verranno accettati programmi che producono un output non conforme alle specifiche**.

Si può assumere che i dati di input siano sempre corretti. Vengono forniti alcuni esempi di input per i vari esercizi, con i rispettivi output previsti. **Un programma che produce il risultato corretto con i dati di input forniti non è necessariamente corretto**. I programmi consegnati devono funzionare correttamente su *qualsiasi* input: a tale scopo verranno testati anche con input differenti da quelli forniti.

Alcuni problemi potrebbero ammettere più soluzioni corrette; in questi casi – salvo indicazione diversa data nella specifica – il programma può restituirne una qualsiasi, anche se diversa da quella mostrata nel testo o fornita con i dati di input/output di esempio. Nel caso di esercizi che richiedano la stampa di risultati di operazioni in virgola mobile, i risultati che si ottengono possono variare leggermente in base all'ordine con cui vengono effettuate le operazioni, oppure in base al fatto che si usi il tipo di dato float o double; tali piccole differenze verranno ignorate.

I file di input assumono che i numeri reali siano rappresentati usando il punto ('.') come separatore tra la parte intera e quella decimale. Questa impostazione potrebbe non essere il default nella vostra installazione di Java, ma è sufficiente inserire all'inizio del metodo `main()` la chiamata:

```
Locale.setDefault(Locale.US);
```

per impostare il separatore in modo corretto (importare `java.util.Locale` per rendere disponibile il metodo).

Ulteriori requisiti

La correttezza delle soluzioni proposte deve essere dimostrabile. In sede di discussione dei progetti potrà essere richiesta la dimostrazione che il programma sia corretto. Per "dimostrazione" si intende una dimostrazione formale, del tipo di quelle descritte nel libro o viste a lezione per garantire la correttezza degli algoritmi. Argomentazioni fumose che si limitano a descrivere il programma riga per riga e altro non sono considerate dimostrazioni.

Il codice deve essere leggibile. Programmi incomprensibili e mal strutturati (ad es., contenenti metodi troppo lunghi, oppure un eccessivo livello di annidamento di cicli/condizioni – "if" dentro "if" dentro "while" dentro "if"…) verranno fortemente penalizzati o, nei casi più gravi, rifiutati.

Usare nomi appropriati per variabili, classi e metodi. L'uso di nomi inappropriati rende il codice difficile da comprendere e da valutare. L'uso di nomi di identificatori deliberatamente fuorviante potrà essere pesantemente penalizzato in sede di valutazione degli elaborati.

Commentare il codice in modo adeguato. I commenti devono essere usati per descrivere in modo sintetico i punti critici del codice, non per parafrasarlo riga per riga.

Esempio di commenti inutili	*Esempio di commento appropriato*

v = v + 1; // incrementa v
if (v>10) { // se v e' maggiore di 10
v = 0; // setta v a zero
}
G.Kruskal(v); // esegui l'algoritmo di Kruskal
	// Individua la posizione i del primo valore
	// negativo nell'array a[]; al termine si ha
	// i == a.length se non esiste alcun
	// valore negativo.
	int i = 0;
	while (i < a.length && a[i] >= 0) {
	i++;
	}

Ogni metodo deve essere preceduto da un blocco di commento che spieghi in maniera sintetica lo scopo di quel metodo.

Lunghezza delle righe di codice. Le righe dei sorgenti devono avere lunghezza contenuta (indicativamente minore o uguale a 80 caratteri). Righe troppo lunghe rendono il sorgente difficile da leggere e da valutare.

Usare strutture dati adeguate. Salvo dove diversamente indicato, è consentito l'utilizzo di strutture dati e algoritmi già implementato nella JDK. Decidere quale struttura dati o algoritmo siano più adeguati per un determinato problema è tra gli obbiettivi di questo corso, e pertanto avrà un impatto significativo sulla valutazione.

Modalità di consegna

I sorgenti vanno consegnati tramite la piattaforma "Virtuale" caricando i singoli file .java (`Esercizio1.java`, `Esercizio2.java`, eccetera). Tutto il codice necessario a ciascun esercizio deve essere incluso nel relativo sorgente; non sono quindi ammessi sorgenti multipli relativi allo stesso esercizio.

Forum di discussione

È stato creato un forum di discussione sulla piattaforma "Virtuale". Le richieste di chiarimenti sulle specifiche degli esercizi (cioè sul contenuto di questo documento) **vanno poste esclusivamente sul forum** e non via mail ai docenti. Non verrà data risposta a richieste di fare debug del codice, o altre domande di programmazione: queste competenze devono essere già state acquisite, e verranno valutate come parte dell'esame.

Valutazione dei progetti

Gli studenti ammessi all'orale verranno convocati per discutere i progetti, secondo un calendario che verrà comunicato sulla pagina del corso. Di norma, **potranno accedere all'orale solo coloro che avranno svolto gli esercizi del progetto in modo corretto.**

La discussione includerà domande sugli esercizi consegnati e sulla teoria svolta a lezione. Chi non sarà in grado di fornire spiegazioni esaurienti sul funzionamento dei programmi consegnati durante la prova orale riceverà una valutazione insufficiente con conseguente necessità di rifare l'esame da zero in una sessione d'esame successiva su nuovi progetti. Analogamente, una conoscenza non sufficiente degli argomenti di teoria, anche relativi a temi non trattati nei progetti, comporterà il non superamento della prova.

La valutazione dei progetti sarà determinata dai parametri seguenti:

- Correttezza dei programmi implementati;
- Efficienza dei programmi implementati;
- Chiarezza del codice: codice poco comprensibile, ridondante o inefficiente comporterà penalizzazioni, indipendentemente dalla sua correttezza. **L'uso di nomi di identificatori fuorvianti o a casaccio verrà fortemente penalizzato**.
- Capacità dell'autore/autrice di spiegare e giustificare le scelte fatte, di argomentare sulla correttezza e sul costo computazionale del codice e in generale di rispondere in modo esauriente alle richieste di chiarimento e/o approfondimento da parte dei docenti.

Checklist

Viene riportata in seguito un elenco di punti da controllare prima della consegna:

- ☐ Ogni esercizio è stato implementato in un UNICO file sorgente `EsercizioN.java`?
- ☐ I programmi compilano dalla riga di comando come indicato in questo documento?
- ☐ I sorgenti includono all'inizio un blocco di commento che riporta cognome, nome, numero di matricola e indirizzo di posta (@studio.unibo.it) dell'autore?
- ☐ I programmi consegnati producono il risultato corretto usando gli esempi di input forniti?

Esercizio 1

La ditta ACME conduce un progetto di analisi di documenti testuali.

Il sistema software utilizzato contiene un componente the analizza la scansione di una pagina di un libro ed estrae una sequenza di coppie <parola, occorrenze> per ogni parola contenuta nella pagina.

Tale sequenza viene memorizzata in un file nel quale vengono aggiunte, una dopo l'altra, le coppie risultanti dall'analisi di tutte le pagine del libro.

Questo file viene processato, da un programma che lo studente deve realizzare, per alimentare una struttura dati **`S`** che contiene le informazioni relative ad ogni parola presente nel libro e il suo numero totale di occorrenze.

Tale struttura dovrà mettere a disposizione le seguenti operazioni:

- `void aggiungiOccorrenze(String parola, int n_occorrenze)` aggiunge `n_occorrenze` al computo totale delle occorrenze di `parola`; se `parola` non ha nessuna occorrenza fino a questo momento, il suo computo totale diviene `n_occorrenze`; il computo si riferisce sempre a `parola` come stringa contenente solo lettere minuscole, è possibile passare come parametro una stringa con caratteri diversi ma questa viene convertita nella versione con solo lettere minuscole;
- `int occorrenzeParola(String parola)` ritorna il computo totale delle occorrenze di `parola`; se `parola` non appare nel testo il valore ritornato deve essere; per quello che riguarda l'uso delle lettere maiuscole valgono le stesse considerazioni espresse per l'operazione precedente.

Realizzare **`S`** come struttura dati dinamica di dimensione iniziale `K` (con `K` definito attraverso una costante nel codice), in modo che il costo temporale medio di `aggiungiOccorrenze` e di `occorrenzeParola` sia costante (assumendo `K` >> numero di parole da gestire).

Scrivere un programma che legga da un file i dati per alimentare `S`; il file deve essere un file testuale composto da più righe ognuna della forma:

<parola>, <occorrenze>

Il programma deve quindi leggere ad un altro file una lista di parole (file di testo con una parola per linea) e visualizzare per ognuna di queste parole il numero totale di occorrenze memorizzate in S (nella forma di righe `<parola>, <occorrenze>`).

I nomi dei due file devono essere passati come parametri di linea di comando al programma. Non è possibile in questo esercizio utilizzare nessuna delle classi collezione del package `java.util`.

Esempio (per compattezza di rappresentazione il carattere / rappresenta una nuova linea):

Contenuto di `occorrenze.txt`:

```
1,lor/1,furori/1,cavallier/1,la/1,gli/1,cortesie/1,donne/1,Agramante/1,il/1,passaro/1
,amori/1,arme/1,tempo/1,tanto/2,e/1,imperator/1,al/1,audaci/1,in/1,Le/1,nocquer/1,mor
te/1,furo/1,giovenil/1,Africa/2,d/3,l/1,canto/1,vendicar/1,Troiano/3,i/1,mare/1,roman
o/1,vanto/1,sopra/2,re/1,si/1,Carlo/1,ire/3,che/1,Francia/1,le/1,io/1,seguendo/2,di/1
,Mori/1,imprese/1,diè/1,tanto/1,mai/1,matto/1,da/1,prima/1,fatto/1,però/1,furore/1,un
/1,colei/1,a/1,Orlando/1,sarà/1,stimato/1,concesso/1,prosa/1,quanto/1,medesmo/1,tal/1
,finir/1,detta/1,uom/1,promesso/4,in/1,Dirò/1,e/1,ingegno/5,che/1,amor/1,quasi/1,ha/1
,rima/1,tratto/1,per/1,saggio/2,mi/1,basti/2,d/1,l/1,se/1,cosa/1,lima/1,m/1,sì/1,ne/2
,ad/1,me/1,venne/1,ho/1,non/1,poco/1,né/2,or/1,era
```

Contenuto di `parole.txt`:

```
carlo/troiano/tanto/in/che
```

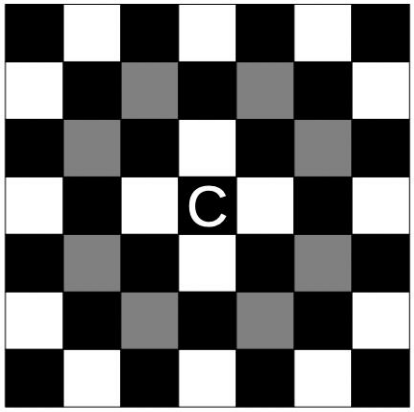
Output di `java -cp . Esercizio1 occorrenze.txt parole.txt`:

```
carlo, 1/troiano, 1/tanto, 3/in, 2/che, 2
```

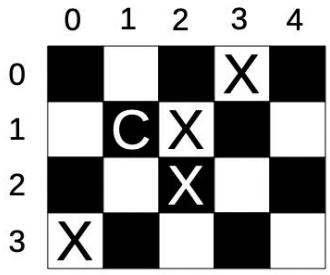
Esercizio 2.

Consideriamo una scacchiera rettangolare con *n* righe e *m* colonne, in cui le celle sono identificate come gli elementi di una matrice: la cella (0, 0) si trova in alto a sinistra, mentre la cella (*n* - 1, *m* - 1) si trova in basso a destra. Un cavallo degli scacchi, più un numero qualsiasi di altri pezzi (ad esempio, pedoni), sono disposti sulla scacchiera. *Scopo del programma è di determinare se il cavallo puo' raggiungere, in una o più mosse, tutte le caselle libere della scacchiera, senza mai dover "mangiare" un altro pezzo.*

Una mossa del cavallo può essere descritta come due passi in orizzontale seguiti da un passo in verticale, oppure un passo in orizzontale seguito da due passi in verticale, in modo che il tragitto percorso formi idealmente una "L". Ad esempio, nella figura seguente le otto caselle in grigio indicano le posizioni raggiungibili da un cavallo posizionato nella casella *C*. E' ovviamente necessario che la casella di destinazione ricada all'interno della scacchiera.



Si noti che il cavallo puo' “saltare” sopra altri pezzi. Nell'esempio seguente, se *C* indica il cavallo e *X* indica una casella occupata da un altro pezzo, lo spostamento del cavallo dalla casella (1, 1) alla casella (2, 3) è ammesso, mentre la mossa che porta da (1, 1) a (0, 3) comporta la cattura del pezzo che si trova in (0, 3) e quindi non va considerata ai fini del nostro problema.



Il programma accetta su riga di comando il nome di un file di input avente il contenuto simile all'esempio seguente:

```
8 numero di righe n
10 numero di colonne m
...XXXX..X
..XXX....X
...XC..XX.
.X....X..X
XX.X..X...
.XX.X..XX.
..XX...X.X
...XXX....
```

La prima riga contiene il numero di righe n (intero, $n \geq 1$), la seconda riga contiene il numero di colonne m (intero, $m \geq 1$); seguono n righe composte da m caratteri ciascuna. I caratteri ammessi sono la lettera `C` maiuscola, che indica la posizione iniziale del cavallo, il punto (`.`) che indica una casella vuota, e la `X` maiuscola che indica la casella occupata da un altro pezzo. Esiste sempre esattamente una singola casella etichettata con `C`.

Il programma deve stampare a video un output come quello che segue:

```
CCCXXXXCCX
CCXXXCCCCX
CCCXCCCXXC
CXCCCCXCCX
XXCXCCXCCC
CXXCXCCXXC
CCXXCCCXCX
.CCXXXCCC.
false
```

L'output deve contenere la scacchiera di input, in cui tutte le caselle raggiungibili dal cavallo sono etichettate con il carattere `C` maiuscolo (le caselle occupate restano etichettate con una `X` maiuscola). Segue una riga che contiene la stringa `true` se il cavallo è in grado di raggiungere tutte le caselle libere, `false` altrimenti. Nell'esempio sopra il cavallo non è in grado di raggiungere due caselle, quindi il programma ha stampato `false`.

Esercizio 3.

Disponiamo di $n \geq 1$ files le cui dimensioni (in MB) sono numeri interi positivi, minori o uguali a 650. Disponiamo anche di una scorta illimitata di CD-ROM, aventi capacità di 650MB ciascuno, su cui vogliamo copiare i files in modo da non eccedere la capienza dei supporti, e assicurandosi che nessun file venga spezzato su più supporti diversi. I files possono essere memorizzati senza rispettare alcun ordine prefissato, purché la somma delle dimensioni dei file presenti sullo stesso supporto non superi 650MB.

Realizzare un algoritmo per allocare i files sui CD-ROM senza spezzare nessun file, e minimizzando il numero di CD-ROM usati. Poiché non esistono algoritmi efficienti per risolvere questo problema, si chiede di implementare l'euristica seguente. Si riempie a turno ciascun CD-ROM, usando i files ancora da memorizzare, in modo da massimizzare lo spazio occupato senza eccedere la capienza dei supporti; per questo è richiesto l'uso di un approccio basato sulla programmazione dinamica. L'algoritmo termina quando tutti i files sono stati copiati sui supporti.

Il programma accetta sulla riga di comando il nome di un file di input, avente la struttura seguente:

`8` *numero di file (n)*
`progetti 143` *nome e dimensione del file 0*
`esempio 242`
`film1 133`
`film2 181`
`programma 74`
`programma2 189`
`documento 26`
`immagine 192` *nome e dimensione del file n-1*

Si può assumere che i nomi dei files non contengano spazi. Il programma, eseguito sull'input precedente, deve visualizzare il seguente output:

```
Disco: 1
progetti 143
film1 133
film2 181
immagine 192
Spazio libero: 1

Disco: 2
esempio 242
programma 74
programma2 189
documento 26
Spazio libero: 119
```

In questo caso vengono utilizzati due CD-ROM; per ciascuno viene visualizzato un header `Disco` *`X`* seguito dall'elenco dei nomi dei files (in un ordine qualsiasi) con le relative dimensioni. Infine, la riga `Spazio libero:` *`Y`* indica lo spazio (in MB) ancora disponibile sul disco. Una riga vuota separa ciascun blocco dal successivo.

La relazione deve contenere una descrizione precisa dell'algoritmo di programmazione dinamica usato per riempire i CD-ROM. In particolare, indicare:

- come sono definiti i sottoproblemi;
- come sono definite le soluzioni a tali sottoproblemi;
- come si calcolano le soluzioni nei casi “base”;
- come si calcolano le soluzioni nel caso generale.

Analizzare il costo asintotico dell'algoritmo implementato (ci si riferisce all'algoritmo completo, non solo a quello di programmazione dinamica usato per riempire ciascun CD-ROM).

Esercizio 4.

Consideriamo una rete di telecomunicazione che è composta da **15 nodi** e **22 link bidirezionali/edges** come illustrato nella seguente figura.

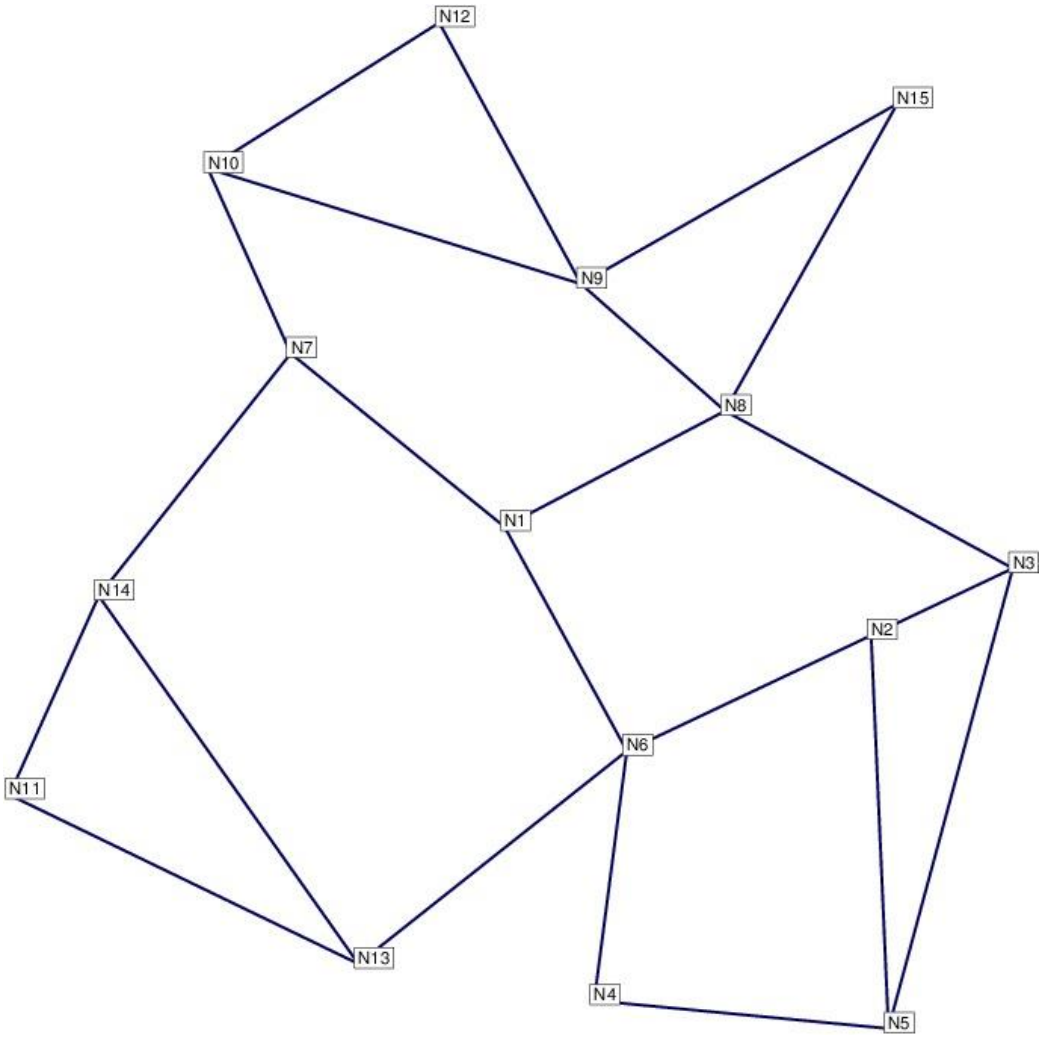


Figura 1: Atlanta network topology
(http://sndlib.zib.de/home.action?show=/problems.overview.action%3Fframeset)

Il programma accetta un unico parametro sulla riga di comando, che rappresenta un file di input

contenente la descrizione della rete di telecomunicazione illustrata nella Figura 1, come segue:

```
15    # numero di nodi n
22    # numero di edge

0 5         11000.00 # peso dell'edge
0 6         7000.00
0 7         4000.00
1 2         8000.00
1 4         1000.00
1 5         15000.00
2 4         5.00
2 7         8000.00
3 4         5.00
3 5         4000.00
5 12        2000.00
6 9         5000.00
6 13        5000.00
7 8         3000.00
7 14        3000.00
8 9         2000.00
8 11        3000.00
8 14        1000.00
9 11        5.00
10 12       1000.00
10 13       1000.00
12 13       1000.00
```

Per semplicità andiamo ad associare ai nodi un *id* intero che va da 0 a 14 al posto degli id N1, N2, …, N15.

Il programma da realizzare deve calcolare **tutti i cammini (completamente distinti) di costo minimo** fra tutte le coppie di nodi.

1. Si richiede di usare in questo esercizio una implementazione dell'algoritmo di Dijkstra che si basa sull'utilizzo delle LISTE durante il calcolo dei cammini minimi. Il programma deve stampare a video:
    - TUTTI i cammini completamente distinti di costo minimo fra il nodo sorgente ed il nodo destinazione, ed il costo totale del cammino minimo (si nota che possono esistere **uno o più** cammini di costo minimo fra una coppia di nodi).
    - il tempo totale in secondi per trovare la soluzione.

2. Calcolare il costo computazionale totale O(...) del programma in funzione del numero di nodi e di archi.