Appello del 18/09/2019 - Prova di teoria (12 punti)

1. (2 punti)

Sia data la sequenza di interi, supposta memorizzata in un vettore:

3 31 72 41 71 0 73 1 10 32 19 13 55 91 14 7

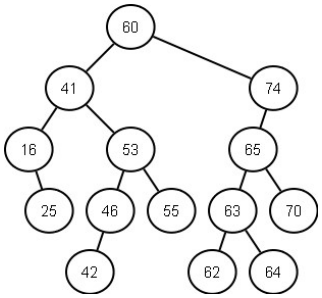
Si eseguano i primi 2 passi dell'algoritmo di quicksort per ottenere un ordinamento **ascendente.** NB: I passi sono da intendersi, impropriamente, come in ampiezza sull'albero della ricorsione, non in profondità. Si chiede, pertanto, che siano ritornate le due partizioni del vettore originale e le due partizioni delle partizioni trovate al punto precedente.

2. (2 punti)

Sia data la sequenza di chiavi intere 13 181 267 302 98 110 45 207. Si riporti il contenuto di una tabella di hash di dimensione 17, inizialmente supposta vuota, in cui avvenga l'inserimento della sequenza indicata. Si usi l'open addressing con quadratic probing. Si definiscano gli opportuni coefficienti.

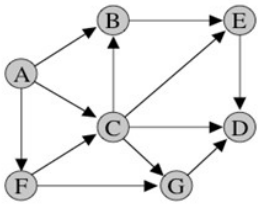
3. (2 punti)

Si inseriscano in **radice** nel BST di figura in sequenza le chiavi 6, 19 e 22 e poi si cancelli la chiave 19. Si disegni l'albero ai passi significativi.



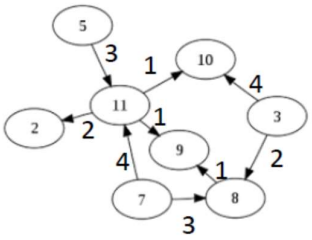
4. (1.5 punti)

Si effettui una visita in ampiezza del seguente grafo orientato, considerando **A** come vertice di partenza. Qualora necessario, si trattino i vertici secondo l'ordine alfabetico.



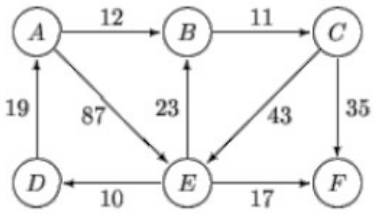
5. (2.5 punti)

Sia dato il seguente DAG pesato: considerando **5** come vertice di partenza, si determinino i cammini **massimi** tra **5** e tutti gli altri vertici. Qualora necessario, si trattino i vertici secondo l'ordine numerico si assuma che la lista delle adiacenze sia anch'essa ordinata numericamente.



6. (**2 punti**)

Sia dato il seguente grafo orientato pesato:



Si determinino i valori di tutti i cammini minimi che collegano il vertice **A** con ogni altro vertice mediante l'algoritmo di Dijkstra. Si assuma, qualora necessario, un ordine alfabetico per i vertici e gli archi.

03MNO Algoritmi e Programmazione

Appello del 18/09/2019 - Prova di programmazione (12 punti)

1. (**2 punti**)
Sia dato un vettore `V` di `N` interi non negativi. Si supponga di avere a disposizione le seguenti funzioni:

- `int findBound(int *V, int N, int *lp, int *rp);` che ritorna mediante puntatore gli indici sinistro (`lp`) destro (`rp`) del sottovettore di `V` di lunghezza massima contenente soli valori positivi. Il valore intero ritornato è 1 se l'intervallo esiste, 0 in caso contrario (il vettore contiene tutti 0)
- `void dec(int *V, int l, int r);` che decrementa di 1 tutte le celle del vettore `V` comprese tra gli indici di sinistra `l` e di destra `r`, inclusi

Si scriva una funzione C che, con il minimo numero di chiamate a `findBound` e `dec`, trasformi `V` nel vettore di tutti 0. ATTENZIONE: Non occorre realizzare `findBound` e `dec`. Non è un problema di ottimizzazione, la soluzione è univoca.

Esempio: se `V= {0,1,2,0,0,3,4,5}` sono necessarie 7 chiamate (a entrambe le funzioni), se `V={3,1,2,0,0,3,0,5}` ne servono 12, se `V={3,1,2,0,0,3,4,5}` ne servono 9 che danno come risultato a ciascuna chiamata {**2,0,1**,0,0,3,4,5}, {2,0,1,0,0,**2,3,4**}, {2,0,1,0,0,**1,2,3**}, {2,0,1,0,0,**0,1,2**}, {2,0,1,0,0,0,**0,1**}, {**1**,0,1,0,0,0,0,1}, {**0**,0,1,0,0,0,0,1}, {0,0,**0**,0,0,0,0,1}, {0,0,0,0,0,0,0,**0**}.

2. (**4 punti**)
Sia dato un albero binario T, cui si accede tramite il puntatore `root` alla radice. I nodi dell'albero hanno come chiavi stringhe di lunghezza massima `maxC`. Si scriva una funzione C con prototipo

```
linkL tree2List(linkT root, int visit);
```

che visiti l'albero secondo la strategia specificata nel parametro `visit` (1: inorder, 2: preorder, 3: postorder), memorizzando le chiavi in una lista concatenata, di cui ritorna il puntatore alla testa.
Si definiscano il nodo dell'albero e relativo puntatore (tipo `linkT`), il nodo della lista e il relativo puntatore (tipo `linkL`). **Si scriva esplicitamente la funzione di inserzione in lista senza fare uso di funzioni di libreria.**

3. (**6 punti**)
Un grafo non orientato, connesso e pesato è rappresentato come insieme di archi. Gli `nV` vertici del grafo sono identificati come interi tra 0 e `nV-1`. Gli archi sono rappresentati mediante il tipo `Edge`:

```
typedef struct { int v; int w; int wt;} Edge;
```

in cui `v` e `w` sono indici di vertici e `wt` è un peso. Gli `nE` archi del grafo sono memorizzati in un vettore dinamico `Edge *edges` (già allocato e contenente i dati). Un albero ricoprente minimo (minimum-spanning tree) è un albero che tocca tutti i vertici e per cui è minima la somma dei pesi degli archi che vi appartengono. Usando i **modelli del Calcolo Combinatorio** si scriva una funzione C che calcoli il peso di un albero ricoprente minimo (**4 punti**). Si scriva una funzione C che, dato un insieme di archi, verifichi se esso rappresenta un albero ricoprente, non necessariamente minimo (**2 punti**). Si osservi che la seconda funzione è usata nella prima in fase di verifica di accettabilità della soluzione. **È vietato l'uso degli algoritmi classici di Kruskal e Prim.**

PER ENTRAMBE LE PROVE DI PROGRAMMAZIONE (18 o 12 punti):

- ***indicare nell'elaborato e nella relazione nome, cognome e numero di matricola.***
- ***se non indicato diversamente, è consentito utilizzare chiamate a funzioni standard, quali ordinamento per vettori, funzioni su FIFO, LIFO, liste, BST, tabelle di hash, grafi e altre strutture dati, considerate come librerie esterne.***
- ***gli header file devono essere allegati all'elaborato (il loro contenuto riportato nell'elaborato stesso). Le funzioni richiamate, inoltre, dovranno essere incluse nella versione del programma allegata alla relazione. I modelli delle funzioni ricorsive non sono considerati funzioni standard.***
- ***consegna delle relazioni (per entrambe le tipologie di prova di programmazione): entro sabato 21/09/2019, alle ore 23:59, mediante caricamento su Portale. Le istruzioni per il caricamento sono pubblicate sul Portale nella sezione Materiale). QUALORA IL CODICE CARICATO CON LA RELAZIONE NON COMPILI CORRETTAMENTE, VERRÀ APPLICATA UNA PENALIZZAZIONE. Si ricorda che la valutazione del compito viene fatta, senza la presenza del candidato, sulla base dell'elaborato svolto in aula. Non verranno corretti i compiti di cui non sarà stata inviata la relazione nei tempi stabiliti.***

03MNO Algoritmi e Programmazione
Appello del 18/09/2019 - Prova di programmazione (18 punti)

Sia data una mappa rettangolare di dimensione `NR` x `NC` caratterizzata dalla presenza di celle libere e celle *occupate (per rappresentare, in casi reali, ostacoli o muri)*. Due celle libere della mappa sono considerate adiacenti se condividono un lato; non sono adiacenti celle poste "in diagonale" l'una rispetto all'altra*: la nozione di adiacenza permette quindi di considerare la mappa come un grafo (implicito) nel quale le celle libere rappresentano i vertici, mentre le adiacenze (al massimo 4 per ogni vertice) rappresentano gli archi.*
Sulla mappa è possibile posizionare ***risorse*** nelle celle libere, in modo tale da *"coprire"* un insieme di caselle sufficientemente vicine. Ogni risorsa *"copre"* la cella da essa occupata e tutte le celle libere raggiungibili da questa in al più `k` passi (quindi tutte le celle a distanza $d \leq k$ nel grafo implicito).
Data una distribuzione delle risorse, ***non è possibile che una cella sia "coperta" da più di una risorsa*** (mentre è possibile che non sia coperta da alcuna risorsa). Lo scopo del programma è duplice: da un lato verificare una copertura, dall'altro individuare una copertura ***ottima*** della mappa.

Si scriva un programma in C che:

- legga un primo file di testo `mappa.txt` organizzato come segue:
  - la prima riga contiene una coppia di interi `NR NC`, che rappresentano le dimensioni della mappa
  - segue l'elenco delle caselle occupate, una per riga (al massimo `NR` x `NC` righe), rappresentate dalle coordinate `X Y`.
- legga un secondo file `proposta.txt` e valuti se esso rappresenta una copertura ammissibile rispetto alle regole di cui sopra, **senza** alcun criterio di ottimalità. Il file sia organizzato come segue:
  - la prima riga contiene il valore intero `k` (distanza massima di copertura) e il numero (intero) `Z` di risorse presenti
  - seguono `Z` coppie `X Y`, una per riga, a rappresentare le coordinate della posizione di ciascuna risorsa
  - seguono `NR` righe di `NC` interi separati da spazi a rappresentare la mappa con la copertura associata alla soluzione proposta. Gli interi sono nell'intervallo `[0..Z]`: 0 indica una cella *non coperta* (indifferentemente libera o occupata/ostacolo) mentre un numero `i` diverso da `0` indica la copertura della cella da parte dell'`i`-esima risorsa (risorse numerate a partire da 1). *Per le dimensioni della mappa e la collocazione delle caselle occupate occorre far riferimento al file `mappa.txt`.*
- nel caso la valutazione di ammissibilità del punto precedente fallisca ed esista una soluzione ammissibile con le stesse risorse, la generi e la memorizzi su di un file `corretto.txt` con lo stesso formato
- individui, se possibile, una soluzione ottima usando ***esattamente `Z` risorse***: `Z` sia ricevuto in input oppure come argomento al main. La soluzione è ottima se massimizza il numero di caselle coperte
- individui una soluzione ottima ***avente il minor numero possibile di risorse*** (in sostanza, a parità di copertura massima di caselle, si sceglie la soluzione che usa meno risorse)

NOTE: *Non sono ammissibili configurazioni in cui una risorsa copra una casella occupata (ostacolo/muro presente nella mappa) o già coperta da un'altra risorsa. La copertura include obbligatoriamente tutte le celle che rispettino il criterio di raggiungibilità: non è possibile escludere arbitrariamente (ad esempio per evitare sovrapposizioni) una o più celle dalla zona di copertura di una risorsa.*
A seguire, un esempio per i file `mappa.txt` e `proposta.txt`. Per rendere più immediato l'esempio, alla rappresentazione del secondo file si è aggiunta una visualizzazione della mappa con ulteriori dettagli grafici: le celle con sfondo grigio sono quelle occupate (quindi inutilizzabili a causa di ostacoli/muri). I numeri in grassetto e sottolineati (**i**) rappresentano la cella in cui ogni `i`-esima risorsa è posizionata.

mappa.txt	proposta.txt	Mappa corrispondente a proposta.txt
6 5 1 1 1 4 3 2 4 3	2 2 2 1 4 4 0 1 0 0 0 1 1 1 0 0 1 1 1 1 2 1 1 1 2 2 0 1 0 2 2 0 0 0 2 2	0 1 0 0 0 1 1 1 0 0 1 **1** 1 1 2 1 1 1 2 2 0 1 0 2 **2** 0 0 0 2 2

Si noti, che la configurazione d'esempio riportata in `proposta.txt` **NON** è valida. Le risorse occupano correttamente celle libere, e non vi sono sovrapposizioni nella copertura delle aree. Per via dell'ostacolo, la risorsa 1 non può coprire le celle in `(1,1)` e in `(3,2)` e la risorsa 2 non può coprire la cella in `(4,3)`. La risorsa `1` inoltre non può coprire la cella `(0,1)` in quanto troppo lontana (distanza 4). Per queste ragioni la configurazione è errata.