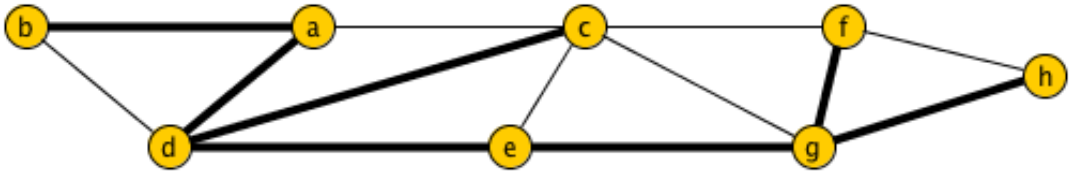
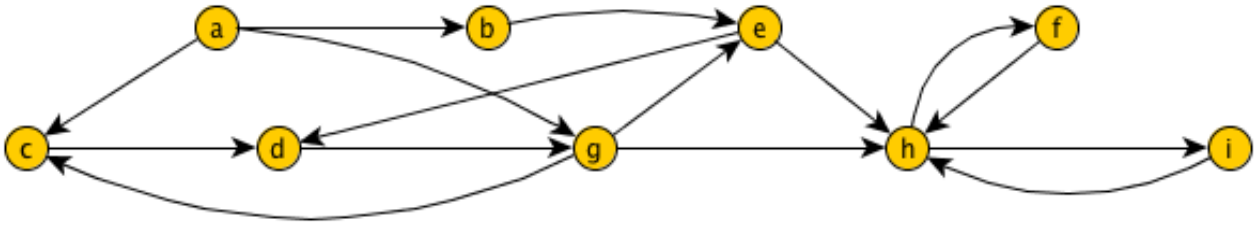
Possibili soluzioni della prova intermedia (14 aprile 2014)

Esercizio [DFS & BFS] (max 4) Si consideri il grafo *G* nella figura qui sotto e l'albero *T* formato dagli archi marcati. Partendo da quali nodi l'albero *T* può essere stato prodotto da una DFS e/o da una BFS? **Motivare bene la risposta**.



L'albero marcato non può essere stato prodotto da una BFS perchè tutti gli archi incidenti nel nodo di partenza devono appartenere all'albero è questo non si verifica per nessuno dei nodi. Vediamo la DFS nodo per nodo.
Partendo da ***a***: No, avrebbe visitato o *b* da *d* o *d* da *b*.
Partendo da ***b,c,d,e***: No, la visita arrivata a *g* avrebbe visitato o *h* da *f* o *f* da *h*.
Partendo da ***f***: No, la visita arrivata a *d* avrebbe visitato o *a* da *c* o *c* da *a*.
Partendo da ***g***: No, avrebbe visitato o *h* da *f* o *f* da *h*.
Partendo da ***h***: No, la visita arrivata a *d* avrebbe visitato o *a* da *c* o *c* da *a*.

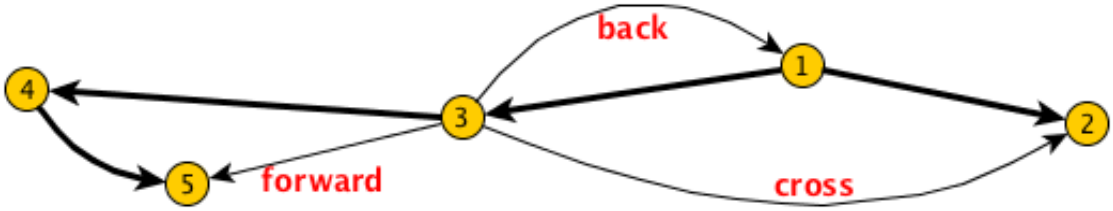
Esercizio [Tarjan] (max 4) Applicando l'algoritmo di Tarjan al grafo nella figura qui sotto, partendo dal nodo *a*, determinare le componenti fortemente connesse numerandole nell'ordine in cui sono trovate dall'algoritmo e per ognuna indicare la sua c-root.



Un possibile ordine con cui sono visitati i nodi dalla DFS dell'algoritmo di Tarjan è: *a,b,e,h,f,i,d,g,c*, Da questo si ottiene che le componenti fortemente connesse sono, nell'ordine: {***h***,*f,i*}, {***e***,*d,g,c*}, {***b***}, {***a***}; in grassetto sono indicate le c-root.

Esercizio [back-forward-cross] (max 4) Può esistere, in una visita DFS di un grafo diretto *G*, un nodo *v* che è incidente in almeno un arco all'indietro, in almeno un arco in avanti e in almeno un arco di attraversamento? In caso affermativo mostrare un esempio, altrimenti dimostrare l'impossibilità.

Sì può esistere, nel grafo qui sotto il nodo *v* è il nodo 3. I nodi sono numerati nell'ordine in cui sono visitati e gli archi dell'albero di visita sono marcati:



Esercizio [remove] (max 7) Dare lo pseudo-codice di un algoritmo che preso in input un grafo non diretto e connesso *G*, un suo nodo *u*, un vettore dei padri *P* relativo a una BFS da *u* in *G* e un arco {*v*, *w*} di *G*, ritorna *True* se e solo se la rimozione dell'arco {*v*, *w*} non cambia le distanze da *u*. L'algoritmo deve avere complessità O(*n*).

Se l'arco $\{v, w\}$ non appartiene all'albero della BFS, chiaramente la sua rimozione non cambia le distanze. Se invece fa parte dell'albero, allora o v è il figlio di w o w è il figlio di v nell'albero radicato in u dato dal vettore dei padri P. Nel primo caso, le distanze non cambiano se e solo se la distanza di v, dopo la rimozione di $\{v, w\}$, non aumenta e questo accade se e solo se esiste un adiacente $x \neq w$ di v tale che $distanza(x) < distanza(v)$. In modo simmetrico per l'altro caso. Per il calcolo efficiente delle distanze dal vettore dei padri si può usare l'algoritmo visto nella lezione 8:

```
DISTANZE(P: vettore dei padri)                    /* Ritorna l'array delle distanze */
    Dist: array delle distanze, inizializzato a -1
    FOR ogni nodo w DO
        IF Dist[w] = -1 THEN
            Dist[w] <- DIST(P, w, Dist)
    RETURN Dist

DIST(P: vettore dei padri, w: nodo, Dist: array)  /* Procedura di ausilio per DISTANZE */
    IF Dist[w] = -1 THEN
        IF P[w] = w THEN
            Dist[w] <- 0
        ELSE
            Dist[w] <- DIST(P, P[w], Dist) + 1
    RETURN Dist[w]
```

Ed ecco lo pseudo-codice dell'algoritmo:

```
DCHANGE(G: grafo, u: nodo, P: vettore padri, {v, w}: arco di G)
    IF P[v] ≠ w AND P[w] ≠ v THEN        /* {v, w} non fa parte dell'albero */
        RETURN True
    ELSE                                 /* {v, w} fa parte dell'albero */
        IF P[v] = w THEN z = v ELSE z = w
        Dist <- DISTANZE(P)              /* Calcola l'array delle distanze da P */
        FOR ogni adiacente x di z DO
            IF {x, z} ≠ {v, w} AND Dist[x] < Dist[z] THEN
                RETURN True
        RETURN False
```

La complessità di `DISTANZE` è $O(n)$ e la scansione degli adiacenti richiede al più $O(n)$, quindi la complessità dell'algoritmo è $O(n)$.

Esercizio [treni] (max 7) Consideriamo una rete ferroviaria relativa a *n* città che per comodità indichiamo con i numeri 1,2…,*n*. Tra ogni coppia (ordinata) *i*, *j* di città c'è al più un treno e questo impiega tempo t_{ij} per andare da *i* a *j*. Inoltre, per ogni città i c'è un tempo fisso di attesa A_i tra l'arrivo di un qualsiasi treno in *i* e la partenza di un qualsiasi treno da *i*. Dare lo pseudo-codice di un algoritmo che data la descrizione della rete ferroviaria (per ogni città *i* il suo tempo d'attesa A_i e i treni che partono da *i* con i relativi tempi t_{ij}), una città di partenza s e una città di destinazione d, calcola il minimo tempo per andare da s a d. L'algoritmo deve avere complessità $O((n + m)\log n)$ dove *m* è il numero di treni. Discutere la correttezza dell'algoritmo.

Possiamo modellare la rete ferroviaria con un grafo F diretto e pesato i cui nodi corrispondono alle città e vi è un arco da *i* a *j* se c'è un treno dalla città *i* alla città *j*, il peso dell'arco è $A_i + t_{ij}$. Nel grafo F i

cammini da s a d rappresentano tutti i possibili viaggi dalla città s alla città d e il peso di ognuno di essi è proprio uguale alla durata del viaggio, cioè la somma dei tempi dei treni e delle attese. Quindi, per calcolare il minimo tempo possiamo usare l'algoritmo di Dijkstra sul grafo F a partire dal nodo s. Basta modificare opportunamente lo pseudo-codice dell'implementazione dell'algoritmo di Dijkstra dato nella lezione 9:

```
TRENI(R: rete ferroviaria (n città e m treni), s: città di partenza, d: città di
destinazione)
    F <- il grafo costruito dalla rete ferroviaria R come descritto sopra
    Dist: array delle distanze, inizializzato a infinito
    Dist[s] <- 0
    H <- min-heap inizializzato con tutti i nodi di F e le priorità sono i valori di Dist
    v <- s
    WHILE H non è vuoto AND v ≠ d DO
        v <- H.get_min()       /* Preleva il nodo con distanza minima */
        FOR ogni adiacente w di v in F DO
            IF Dist[w] > Dist[v] + p(v, w) THEN
                Dist[w] <- Dist[v] + p(v, w)
                H.decrease(w)  /* Aggiorna l'heap a seguito del decremento */
    RETURN Dist[d]
```

La costruzione del grafo F (definito tramite liste di adiacenza) prende tempo $O(n + m)$ e l'algoritmo di Dijkstra, nell'implementazione data, ha complessità $O((n + m)\log n)$.

Esercizio [acqua] (max 8) Ci sono n case, indicate con gli interi $i = 1,2,\ldots,n$, ognuna delle quali necessita di una fornitura d'acqua. La costruzione di un pozzo nella casa i costa $p[i]$ e la costruzione di una tubazione fra le case i e j costa $c[i, j]$. La fornitura d'acqua per una casa o è un pozzo costruito nella casa o è un cammino di tubazioni dalla casa a qualche pozzo. Dare lo pseudo-codice di un algoritmo che determina le case in cui costruire i pozzi e le tubazioni da costruire per dare una fornitura d'acqua a tutte le case minimizzando il costo totale. L'algoritmo deve avere complessità $O(n^2)$. Discutere la correttezza dell'algoritmo.
Suggerimento: rappresentare il problema tramite un opportuno grafo pesato con n+1 nodi, il nodo in più rappresenta i pozzi…

Seguendo il suggerimento rappresentiamo il problema tramite un grafo A non diretto e pesato con $n+1$ nodi di cui n corrispondono alle case e un nodo z rappresentante i pozzi. Tra due nodi casa i e j c'è un arco di peso $c[i, j]$ e tra ogni nodo casa i e il nodo z c'è un arco di peso $p[i]$. Ogni soluzione ammissibile, che cioè garantisce la fornitura d'acqua ad ogni casa, è un sottoinsieme degli archi del grafo A tale che ogni nodo casa è connesso direttamente (con un arco, cioè la casa ha un pozzo) o indirettamente (con un cammino) al nodo z. Questo è garantito da un qualsiasi albero di copertura del grafo A. Quindi la soluzione al problema è un minimo albero di copertura di A. Siccome il grafo A e completo conviene usare l'implementazione dell'algoritmo di Prim che usa un array invece che un heap:

```
ACQUA(n: numero case, c: matrice costi tubazioni, p: array costi pozzi)
    T: vettore dei padri di dimensione n
    COST: array dei costi di dimensione n
    FOR i <- 1 TO n DO
        T[i] <- n+1                     /* n+1 rappresenta il nodo z */
        COST[i] <- p[i]
    REPEAT n volte DO
        imin <- -1
        FOR i <- 1 TO n DO              /* Trova il nodo di costo minimo */
            IF COST[i] ≠ -1 AND (imin = -1 OR COST[i] < COST[imin]) THEN
                imin <- i
        COST[imin] <- -1                /* I nodi estratti sono marcati con -1 */
        FOR i <- 1 TO n DO
```

```
            IF c[imin, i] < COST[i] THEN
                T[i] <- imin
                COST[i] <- c[imin, i]
    RETURN T
```

Le case i in cui bisogna costruire un pozzo sono quelle per cui $T[i] = n+1$. La Complessità è chiaramente $O(n^2)$.

Esercizio [museo] (max 8) In un museo c'è un lungo corridoio rettilineo in cui sono esposti n quadri nelle posizioni $0 \leq q_1 < q_2 < q_3 < \ldots < q_n$. Un custode può sorvegliare i quadri che si trovano ad al più distanza 5 dalla sua posizione. Vogliamo sorvegliare tutti i quadri con un numero minimo di custodi. Dare lo pseudo-codice di un algoritmo che trova le posizioni di un numero minimo di custodi che sorvegliano gli n quadri. La complessità dell'algoritmo deve essere $O(n)$. Dare la dimostrazione di correttezza dell'algoritmo proposto.

Un algoritmo greedy naturale consiste nel considerare i quadri in ordine di posizione crescente e se ci sono quadri non ancora sorvegliati, posizionare un nuovo custode in modo tale che l'estremo minore del suo intervallo di sorveglianza (di lunghezza 10) cada nella posizione del primo quadro non ancora sorvegliato:

```
MUSEO(q: array delle posizioni degli n quadri)
    POS <- lista vuota delle posizioni dei custodi
    i = 1
    WHILE i <= n DO
        p <- q[i]+5                        /* Posizione custode */
        WHILE i <= n AND q[i] <= p+5 DO    /* Trova il primo quadro non sorvegliato */
            i <- i + 1
        POS.append(p)
    RETURN POS
```

Ad ogni iterazione del `WHILE` esterno almeno un nuovo quadro viene sorvegliato e l'indice `i` è incrementato di almeno 1 nel `WHILE` interno. Ne segue che sono eseguite al più n iterazioni del `WHILE` esterno e la complessità totale è $O(n)$. Per dimostrare la correttezza seguiamo lo schema generale per gli algoritmi greedy. Sia POS_h la lista `POS` al passo h e sia k il numero di iterazioni del `WHILE` esterno. Per completare la dimostrazione di correttezza è sufficiente dimostrare:

Per ogni $h = 1, 2, \ldots, k$, esiste una soluzione ottima POS^* che estende POS_h.
Dimostrazione. Per $h = 0$, POS_0 è una lista vuota e quindi è estesa da una qualsiasi soluzione ottima. Supponiamo che sia vero per h e dimostriamolo per $h + 1$. Sia POS^* una soluzione ottima che per ipotesi induttiva estende POS_h. Se POS^* estende anche POS_{h+1}, abbiamo fatto. Altrimenti, la posizione p del custode scelta al passo $h + 1$ non è contenuta in POS^*. Il quadro i che è il primo a non essere sorvegliato al passo $h + 1$ deve essere sorvegliato da un qualche custode in POS^*. Sia p^* la posizione di un tale custode. Siccome $p = q[i] + 5$, deve essere $p^* \leq p$ (perchè altrimenti un custode in p^* non potrebbe sorvegliare il quadro i). Chiaramente p^* non può essere in POS_h perchè il quadro i non è sorvegliato dalle posizioni in POS_h. Allora, possiamo sostituire p^* con p in POS^* definendo $POS^@ = (POS^* - p^*) + p$. Tutti i quadri con posizione $\geq q[i]$ sorvegliati dalla posizione p^* in POS^* sono anche sorvegliati dalla posizione p. Quindi $POS^@$ è una soluzione ottima che estende POS_{h+1}.