Algoritmi e Strutture Dati **Anno Accademico 2020–2021**

Primo Appello
8 Giugno 2021

SECONDA PARTE

ATTENZIONE: L'uso di **break** e **continue** NON è consentito

Esercizio 1. (max 10 punti). Si consideri il seguente problema:

INPUT: Array A **ordinato** di n interi il cui valore può essere solo 0 oppure 1.
OUTPUT: Il numero di occorrenze del numero 1 in A

Si scriva lo pseudo-codice di un algoritmo basato sulla tecnica **Divide&Impera** che richieda tempo $O(\log n)$ nel caso peggiore. Scrivere l'equazione di ricorrenza e risolverla per mostrare che il costo computazionale dell'algoritmo sia effettivamente quello richiesto.

Esercizio 2. (max 11 punti) Si consideri il seguente problema:

INPUT: Grafo $G = (V, E)$, una funzione di peso positiva sui **nodi** $w : V \to \mathbb{N}$, due nodi $x, y \in V$ e un valore $k \in \mathbb{N}$.
OUTPUT: VERO se esiste un cammino in G da x a y che passi solo per nodi di peso minore o uguale a k, FALSO altrimenti.

Descrivere brevemente l'algoritmo a parole, scrivere lo pseudo-codice ad alto livello e studiarne il costo computazionale.

Domanda teorica (in alternativa ad uno dei due esercizi) (max 4 punti)

Dato un MIN-HEAP memorizzato in un array $H[0..n-1]$ che contiene $n \geq 4$ chiavi intere **tutte diverse** tra loro...

1. (1 punto) ... in che posizioni di H può comparire il secondo elemento più piccolo?
2. (1 e $1/2$ punto) ... per $k = 3, 4$, in che posizioni di H può comparire il k-esimo elemento più piccolo?
3. (1 e $1/2$ punti) ... in che posizioni di H può comparire l'elemento massimo?

SOLUZIONI SECONDA PARTE

SOLUZIONE Esercizio 1.

Ci sono almeno due modi diversi per trovare la soluzione, entrambi usano l'idea della ricerca binaria sfruttando il fatto che l'array sia ordinato. In questo modo il tempo è logaritmico nella dimensione dell'input.

In entrambi i casi, come nella ricerca binaria, scriviamo una funzione ricorsiva (per utilizzare la tecnica Divide&Impera) che prende un input, oltre che A, due indici i e j che indicano la porzione dell'array su cui lavorare in una generica chiamata ricorsiva (con i estremo di sinistra e j estremo di destra).

VERSIONE 1

Scriviamo una funzione che, data una porzione dell'array A dall'indice i all'indice j (compresi) restituisce il numero di occorrenze del numero 1, calcolando la somma degli uno nella metà di destra più quelli nella metà di sinistra. Attenzione, si deve chiamare la funzione ricorsivamente su una metà sola.

- **Caso base**: $i > j$, la porzione dell'array è vuota, quindi si restituisce zero (non ci possono più essere altri uni).
- **Caso ricorsivo:** $i \leq j$. Si considera l'elemento in posizione centrale k di $A[i..j]$: se in $A[k]$ abbiamo uno zero, allora gli uni si possono trovare solo nella porzione $A[k+1..j]$, ammesso che ce ne siano. Quindi si chiama la funzione ricorsivamente su questa porzione e si restituisce il risultato della chiamata. Altrimenti, se $A[k] = 1$, allora tutti gli elementi in $A[k..j]$ contengono uni (e sono esattamente $j-k+1$) e in più potrebbero esserci altri uni nella parte di sinistra. Quindi si restituisce la somma di $j-k+1$ e del risultato della chiamata ricorsiva sulla porzione di sinistra.

```
Occorrenze(A,i,j)
   if i > j then return 0
   k = ⌊ (i+j) /2 ⌋
   if A[k] = 0
      then return Occorrenze(A,k+1,j)
      else return (j-k+1) + Occorrenze(A,i,k-1)
```

Chiamata principale: `Occorrenze(A,0,n-1)`.

L'equazione di ricorrenza si ricava dal codice (e non si cerca di scrivere l'equazione per avere come risultato $O(\log n)$): nel caso base il lavoro da fare è costante, nel caso ricorsivo si paga il costo della chiamata ricorsiva su un array che è la metà di quello di partenza, più un lavoro costante (per calcolare k e fare il confronto nel test dell'if, più un eventuale numero costante di operazioni aritmetiche). Quindi, quello che abbiamo è

$$T(n) = \begin{cases} O(1) & \text{se } n = 0, \quad \text{caso base della ricorsione} \\ T(\frac{n}{2}) + O(1) & \text{se } n > 0, \quad \text{formula ricorsiva} \end{cases}$$

e abbiamo che: $a = 1, b = 2$ e $d = 0$ (perché $n^0 = 1$) e, per il master theorem $\log_b a = \log_2 1 = 0 = d$ e quindi $T(n) = O(n^d \log n) = O(\log n)$ (che è esattamente il conto che abbiamo fatto per la ricerca binaria).

VERSIONE 2

Trovare l'indice del primo uno cercando una coppia di indici consecutivi per cui si ha uno zero preceduto da un uno. L'indice u del primo uno può essere poi usato per calcolare il risultato richiesto: da u a $n-1$ ci sono esattamente $(n-1)-u+1 = n-u$ uni.

A differenza di quanto fatto a lezione, per calcolare l'elemento centrale, prendiamo la parte intera superiore (invece che inferiore). Nel caso in cui la sequenza abbia un numero dispari di elementi non cambia niente, ma nel caso in cui sia pari, in questo modo si seleziona il primo elemento della porzione di destra (invece che l'ultimo della parte di sinistra) e quando abbiamo due elementi nella porzione $A[i..j]$ da analizzare, andremo a selezionare il secondo elemento e siamo sicuri che l'elemento precedente esista.

In questa versione è necessario fare attenzione a quanti elementi costituiscono la sottosequenza che stiamo considerando. Infatti, poiché ispezioneremo due elementi, è necessario considerare a parte i casi in cui la porzione di A da analizzare contenga un solo elemento (può succedere quando partiamo da una sequenza lunga tre elementi e la chiamata ricorsiva viene fatta su un solo elemento), o nessuno. Questo per evitare di andare ad accedere ad indici illeciti o che si trovano al di fuori di $A[i..j]$. Questo fa sì che ci siano più casi base.

- **Casi base:** (1) se $i > j$ allora non ci sono uni nella sequenza e si ritorna zero; (2) se $i = j$ vuol dire che la sottosequenza che stiamo analizzando ha un solo elemento. Se questo elemento è zero restituiamo zero, altrimenti in i abbiamo trovato il primo uno e si restituisce $n-i$; (3) sia k l'indice dell'elemento a metà della sequenza $A[i..j]$, se $A[k] = 1$ e l'elemento precedente è uno zero ($A[k-1] = 0$), allora è stato trovato il primo uno e si restituisce $n-k$.
- **Caso ricorsivo**: k è l'indice dell'elemento a metà della sequenza $A[i..j]$ e abbiamo che: $i < j$ AND ($A[k] = 0$ OR $A[k-1] = 1$). Se $A[k] = A[k-1] = 1$ (in realtà basta testare $A[k]$), allora il primo uno si trova a sinistra dell'indice k, altrimenti (se $A[k] = 0$), allora il primo uno si trova a destra di k. A seconda del caso si chiama la funzione ricorsivamente sulla metà appropriata, restituendo il risultato della chiamata ricorsiva.

```
Occorrenze(A,i,j,n)
   if i > j then return 0 // non ci sono elementi (tantomeno uni)
   if i = j AND A[i] = 1 then return n-i // c'e' un elemento solo, il primo 1
   if i = j AND A[i] = 0 then return 0 // c'e' un elemento solo uguale a zero
   // abbiamo almeno due elementi
   k = ⌈ (i+j) /2 ⌉
   // se ci sono esattamente due elementi, k e' l'indice di quello a destra
   // quindi k-1 e' un indice che cade in A[i..j]
   if A[k] = 1 AND A[k-1] = 0 then return n-k // trovato il primo uno
   if A[k] = 1
      then return Occorrenze(A,i,k-1,n) // il primo uno sta a sinistra.
                                        // Dopo la ricorsione, rimane almeno un elemento,
                                        // se erano esattamente due, rimane quello di sinistra
                                        // e alla prossima chiamata termina
   // in posizione k c'e' uno zero
   return Occorrenze(A,k+1,j,n) // potrebbe non rimane neanche un elemento,
                                // se questo e' il caso,
                                //alla prossima chiamata termina
```

Chiamata principale: `Occorrenze(A,0,n-1,n)`.

Osservazione: per comodità passiamo anche n, ma potrebbe essere sostituito nel codice da `length(A)` e non essere passato come parametro.

Per quello che riguarda l'equazione di ricorrenza, si procede in modo analogo all'altra versione:

$$T(n) \leq \begin{cases} O(1) & \text{se } n = 1, \quad \text{caso base della ricorsione} \\ T(\frac{n}{2}) + O(1) & \text{se } n > 0, \quad \text{formula ricorsiva} \end{cases}$$

Ci sono due differenze: mettiamo un $\leq$ perché può succedere che $n > 1$ ma il costo è costante, e il caso base della ricorsione è uno.

ERRORI PIÙ COMUNI:

- Chiamare la funzione ricorivamente su entrambe le metà dell'array (il costo diventa lineare)
- Dimenticare il return alle chiamate ricorsive
- Fare una scansione dell'array utilizzando la ricorsione invece che l'iterazione (è lineare anche così)
- Dimenticare la chiamata principale
- Sbagliare ad impostare o risolvere l'equazione di ricorrenza (o non provarci neanche)

SOLUZIONE Esercizio 2.

Ci sono vari modi di affrontare il problema, oltre al fatto che si può utilizzare una variante sia della BFS che della DFS. In queste soluzioni ne metto solo due versioni, una che usa la DFS e una la BFS.

VERSIONE 1

Modifichiamo una visita DFS del grafo che parte da x. Inizializziamo, per ogni nodo $v \in V$ con peso maggiore di k `visited` a TRUE. In questo modo questi nodi non verranno presi in considerazione durante la visita (pensando "erroneamente" di esserci già passati). Facciamo attenzione che y deve essere raggiungibile anche se $w(y) > k$, quindi inizializziamo `visited[y]` a False. Alla fine della visita si restituisce il valore di `visited[y]`. Poiché siamo interessati ai nodi raggiungibili da x non è necessario ripartire con la visita da altri nodi una volta finita quella partita da x.

```
Cammino(G=(V,E), x, y)
  for all v ∈ V do
      if w(v) > k
         then visited[v] := True
         else visited[v] := False
  visited[y] := False
  DFS-visit(G,x)
  return visited[y]
```

La procedura `DFS-visit` è quella vista a lezione, volendo la si può modificare per interrompere la visita appena si incontra y (provate a farlo per **ESERCIZIO**).

Il costo computazionale è quello della DFS, perché le modifiche portano solo l'aggiunta di un costo costante. Quindi, $O(|V| + |E|)$.

VERSIONE 2

Modifichiamo la BFS nella fase di esplorazione dei vicini del nodo estratto dalla coda: (1) se siamo arrivati in y abbiamo scoperto che il cammino esiste, quindi interrompiamo la visita e restituiamo True (non ci interessa controllare il peso di y); (2) prima di mettere un vicino in coda controlliamo non solo che `visited` sia False, ma anche che il suo peso sia minore o uguale di k. In questo modo in coda ci saranno solo nodi che possono essere considerati nel cammino da x a y. Se arriviamo alla fine della visita (cioè a svuotare la coda) e non abbiamo incontrato y, allora non esiste il cammino cercato e restituiamo False.

```
Cammino(G=(V,E), x, y)
  for all v ∈ V do
      visited[v] := False
  visited[x] := true
  Q := new_queue()
  enqueue(Q,x)
  while NOT is_empty_queue(Q) do
        u := dequeue(Q)
        for all (u,v) ∈ E do
                if v = y then return True
                if visited[v] = False AND w(v) ≤ k
                   then
                     visited[v] := True
                     enqueue(Q,v)
  return False
```

Il costo computazionale è quello della BFS, perché le modifiche portano solo l'aggiunta di un costo costante. Quindi, $O(|V| + |E|)$.

ERRORI PIÙ COMUNI:

- Sbagliare a fare la visita del grafo
- Terminare la visita (restituendo Falso) appena si arriva in un nodo v con $w(v) > k$. La visita deve continuare, perché potrebbe esserci un'altra strada.
- Restituire Falso se $w(y) > k$
- Sbagliare a riportare il costo della visita del grafo (non quella dell'esercizio, ma quella vista a lezione)

SOLUZIONE Domanda teorica.

1. Il secondo elemento più piccolo può è uno dei due figli della radice, quindi può apparire in posizione 1 o 2.
2. Dato un nodo v dell'heap, i nodi sul cammino dalla radice a v contengono chiavi minori di v (per definizione), quindi, il terzo elemento più piccolo non può trovarsi a livello maggiore di due, così come il quarto a livello maggiore di tre. Entrambi possono, però, stare in uno qualunque dei livelli a partire dal livello uno a quello massimo, a seconda di come sono memorizzate le chiavi nell'heap.
   $k = 3$: il terzo elemento più piccolo può essere uno dei figli della radice (posizioni 1 o 2), oppure un nipote della

radice (posizioni 3, 4, 5 o 6). Esempio del primo caso: $H =< 3, \mathbf{4}, 5, 6, 7, 8, 9, ... >$. Esempio del secondo caso $H =< 3, 4, 7, \mathbf{5}, 6, 8, 9, ... >$.
$k = 4$: il quarto elemento più piccolo può essere uno dei figli della radice (posizioni 1 o 2), oppure un nipote della radice (posizioni 3, 4, 5 o 6), oppure un bis-nipote della radice (posizioni nell'intervallo $[7..14]$). Esempio del primo caso: $H =< 3, 4, \mathbf{6}, 5, 7, 8, 9, ... >$. Esempio del secondo caso $H =< 3, 4, 5, \mathbf{6}, 7, 8, 9, ... >$. Esempio del terzo caso: $H =< 3, 4, 11, 5, 7, 12, 13, \mathbf{6}, 8, 9, 10, ... >$

3. Il massimo può apparire in una qualunque delle foglie, che sono memorizzate in celle consecutive di H dopo l'ultimo nodo interno, che non è altro che il padre dell'ultima foglia. Quest'ultima si trova in posizione $DimHeap$, quindi le posizioni possibili sono da $Padre(DimHeap) + 1$ a $n - 1$.

Secondo Appello
29 Giugno 2021

SECONDA PARTE

ATTENZIONE: L'uso di **break** e **continue** NON è consentito

Esercizio 1. (max 12 punti)

Sia dato un grafo non orientato connesso $G = (V, E)$ che rappresenta una rete stradale e tre nodi (città) $u, v, w \in V$ di questa rete in cui abitano tre amici. I tre amici vogliono incontrasi in un nodo (città) $z \in V$ minimizzando la distanza che devono percorrere (misurata come somma del numero minimo di archi che ognuno di essi deve attraversare per raggiungere z partendo da u, v e w).

Descrivere a parole e fornire lo pseudocodice di un algoritmo che prende in input G, u, v, w e restituisce z.
Discutere il costo computazionale dell'algoritmo proposto.

Esercizio 2. (max 9 punti)

Si assuma di avere a disposizione una procedura `delete_node(p)` che, preso in input un puntatore p ad un nodo di un albero, elimini il nodo dalla memoria. Si consideri il seguente problema:

INPUT: un albero binario T realizzato con nodi e puntatori (campi del nodo: chiave, puntatore al figlio sinistro e puntatore al figlio destro)
OUTPUT: eliminare tutti i nodi dell'albero.

Fornire lo pseudocodice di una procedura ricorsiva per il problema. Scrivere a parole il caso base e il caso ricorsivo. Discutere il costo computazionale della procedura proposta.

Domanda teorica (in alternativa ad uno dei due esercizi) (max 5 punti) - **Soluzioni sui lucidi delle lezioni**
Dato un albero binario completo e perfettamente bilanciato T con n nodi e altezza h:
1. (3 punti) dimostrare per induzione che T ha esattamente 2^h foglie;
2. (2 punti) (sfruttando il risultato precedente) dimostrare che vale $n = 2^{h+1} - 1$.

SOLUZIONI SECONDA PARTE

SOLUZIONE Esercizio 1.

L'esercizio si può risolvere utilizzando diversi algoritmi visti a lezione per calcolare i cammini minimi. A seconda della scelta, il costo computazionale varia. In particolare, le alternative sono tra: BFS (soluzione più efficiente, perché si deve contare solo il numero di archi dei cammini e non abbiamo una funzione di costo arbitraria sugli archi), Dijkstra (soluzione meno efficiente della prima, perché l'utilizzo della coda con priorità comporta un dispendio maggiore di tempo, che in questo caso poteva essere evitato), Floyd-Warshall (soluzione ancora più costosa, perché esegue anche computazioni che non sono necessarie in questo caso).

Poiché il testo dell'esercizio non pone vincoli, tutte le alternative potevano essere accettabili, a patto di stabilire il corretto costo computazionale.

Versione 1

L'algoritmo funziona in due passi:

1. usando la BFS (o Dijkstra modificato impostando $c(e) = 1$ per ogni arco $e \in E$), calcolare le distanze di tutti i nodi di G da ognuno dei tre nodi u, v e w. In questa soluzione, a questo scopo, invochiamo la BFS tre volte con sorgenti u, v e w. Salviamo le distanze calcolate in tre array delle distanze distinte.
2. per ogni nodo del grafo, utilizzando i tre array delle distanze, calcolare la somma delle distanze da quel nodo ai tre nodi u, v e w. Restituire l'identificativo del nodo che minimizza tale somma.

Poiché il codice della BFS vista a lezione non restituisce l'array delle distanze e poiché non abbiamo bisogno del vettore dei pardi, riscriviamo anche la BFS con queste modifiche.

```
PuntoDIncontro(G=(V,E),u,v,w)
   DistU := BFS(G,u)
   DistV := BFS(G,v)
   DistW := BFS(G,w)
   z := 0 // 0 non e' l'identificativo di un nodo del grafo, alla fine sara' il risultato
   min := +∞ // inizializzazione minimo
   for i = 1 to n do // ciclo su tutti i nodi di G
       dist := DistU[i] + DistV[i] + DistW[i]
       if dist < min
          then // aggiornamento del minimo
            min := dist
            z := i
   return z


BFS(G=(V,E),s)
  for all x ∈ V do
      dist[x] := -1
  dist[s] := 0
  Q := new_queue()
  enqueue(Q, s)
```

```
while NOT is_empty_queue(Q) do
   x := dequeue(Q)
   for all (x,y) ∈ E do
       if dist[y] = -1
          then
            dist[y] := dist[x] + 1
            enqueue(Q,y)
return dist[]
```

Il costo computazionale è dato dal costo di tre chiamate della BFS ($O(3(|V| + |E|)) \in O(|V| + |E|)$, perché il numero delle invocazioni della BFS è costante), PIÙ il costo del ciclo su tutti i nodi. Quest'ultimo viene eseguito esattamente n volte e, ad ogni iterazione, il lavoro è costante. Quindi, il ciclo costa $O(|V|)$ e in totale abbiamo $O(|V| + |E|) + O(|V|) \in O(|V| + |E|)$.

Versione 2 - per esercizio

Calcolare (utilizzando Floyd-Warshall, oppure Dijkstra per ogni nodo) le distanze tra tutte le coppie di nodi di G. Per ogni nodo del grafo, calcolare la somma delle distanze da quel nodo ai tre nodi u, v e w. Restituire l'identificativo del nodo che minimizza tale somma.

ERRORI PIÙ COMUNI:

- Assumere che il nodo z sia dato in input, calcolare le distanze di u, v e w da z per poi sommarle e restituire z (che era già noto prima di fare tutto questo lavoro).
- Sbagliare a determinare il nodo z calcolando una funzione diversa da quella richiesta
- Usare BFS/Dijkstra assumendo che l'algoritmo restituisca l'array delle distanze. Restituisce il vettore dei padri dell'albero dei cammini minimi.
- Usare Dijkstra senza passare la funzione di costo sugli archi c ma usandola nello pseudo-codice
- Invocare più volte la BFS o Dijkstra utilizzando un solo array delle distanze che è una variabile globale: il contenuto delle prime chiamate verrà perso perché sovrascritto ripetutamente. Rimarrà solo il contenuto relativo all'ultima chiamata.
- Sbagliare (malamente) a riportare/modificare il codice della BFS o di Dijkstra
- Sbagliare a riportare il costo computazionale della BFS o di Dikstra

SOLUZIONE Esercizio 2.

La soluzione all'esercizio è una semplice visita DFS in post-ordine in cui, al ritorno della ricorsione, si elimina il nodo radice dell'albero su cui è stata fatta la chiamata. Infatti, per poter utilizzare i puntatori ai figli, non è possibile prima eliminare i nodi e successivamente accedere ai campi con i puntatori (che sono stati cancellati).

Esistono almeno tre versioni leggermente diverse per risolvere il problema, a seconda del caso base considerato. Tutte le versioni hanno lo stesso costo computazionale asintotico, pari a $O(n)$, dove n è il numero di nodi dell'albero, assumendo che la procedura `delete_node(t)` abbia costo costante. Ovviamente, il numero di chiamate ricorsive ESATTE può cambiare a seconda della scelta del caso base.

ESERCIZIO: è possibile risolvere il problema anche con una visita in pre-order, ma è necessario adottare qualche accorgimento. Scrivere una versione della procedura che risolva il problema utilizzando una visita in pre-ordine.

Versione 1

- **Caso base:** albero vuoto, non si deve fare niente perché non c'è niente da cancellare
- **Caso ricorsivo:** albero non vuoto: si chiama la procedura ricorsivamente su entrambi i sottoalberi (nel caso in cui uno o entrambi siano vuoti la procedura funziona correttamente) e poi si cancella la radice dell'albero.

```
CancellaAlbero(t)
   if t ≠ NIL
      then
        CancellaAlbero(t.left)
        CancellaAlbero(t.right)
        delete_node(t)
```

Chiamata principale `CancellaAlbero(T)`.

Versione 2

- **Caso base:** l'albero è una foglia: si cancella la foglia
- **Caso ricorsivo:** l'albero non è una foglia, ci sono almeno due nodi nell'albero: si chiama la procedura ricorsivamente sui sottoalberi non vuoti (almeno uno) e poi si cancella la radice dell'albero.

```
CancellaAlbero(t)
   if t.left = NIL AND t.right = NIL
      then
        delete_node(t)
      else
        if t.left ≠ NIL
           then CancellaAlbero(t.left)
        if t.right ≠ NIL
           then CancellaAlbero(t.right)
        delete_node(t)
```

Per gestire anche il caso un cui l'albero T passato in input sia vuoto, è necessario scrivere una procedura principale che chiama `CancellaAlbero` dopo aver controllato che T sia diverso da NIL.

```
RimuoviAlbero(T)
   if T ≠ NIL
      then
      CancellaAlbero(T)
```

Versione 3

- **Casi base:** (1) l'albero è vuoto, non si deve fare niente perché non c'è niente da cancellare; (2) l'albero è una foglia: si cancella la foglia
- **Caso ricorsivo:** l'albero non è vuoto e ci sono almeno due nodi nell'albero: si chiama la procedura ricorsivamente su entrambi i sottoalberi (nel caso in cui uno dei due sia vuoto o una foglia, la procedura funziona correttamente) e poi si cancella la radice dell'albero.

```
CancellaAlbero(t)
   if t ≠ NIL
      then
        if t.left = NIL AND t.right = NIL
           then
             delete_node(t)
           else
             CancellaAlbero(t.left)
             CancellaAlbero(t.right)
             delete_node(t)
```

Chiamata principale `CancellaAlbero(T)`.

ERRORI PIÙ COMUNI:

- Non cancellare il nodo nel caso ricorsivo. Se il caso base è l'albero vuoto, non si cancella nessun nodo. Se il caso base è la foglia, si cancellano solo le foglie dell'allbero.
- Usare una DFS in pre-ordine invece che in post-ordine (senza nessun accorgimento).
- Stabilire che il caso base si ha quando la procedura viene invocata su una foglia, ma non trattare il caso il cui un nodo interno abbia un solo figlio. In questo caso la procedura viene chiamata su un albero vuoto, si prova ad accedere ai puntatori dei figli che, ovviamente, non esistono e si ha un errore.
- Scrivere una funzione invece che una procedura, come richiesto dalla traccia.
- Dimenticare la chiamata principale
- Nella descrizione dei casi: (1) non indicare quali siano i casi base e/o quello ricorsivo; (2) non spiegare cosa si deve fare nel caso ricorsivo, ma dire solo che il caso ricorsivo si ha quando il nodo ha almeno un figlio; (3) nel caso ricorsivo dire che: "si scende fino alle foglie e poi si torna su", questo è quello che succede eseguendo tutte le chiamate, ma nel caso ricorsivo si deve spiegare cosa fare nel caso in cui non ci si trovi nel caso base, ovvero spiegare a parole quello che si troverà nel codice (e codice ed esecuzione del codice sono due cose diverse).
- Sbagliare ad indicare il costo computazionale della DFS, che non dipende dall'altezza dell'albero, ma dal numero di nodi.
- Dare una descrizione dei casi base/ricorsivo diversa da quello che viene fatto nello pseudo-codice.
- Non utilizzare la rappresentazione dell'albero con puntatori

Terzo Appello
14 Luglio 2021

SECONDA PARTE

ATTENZIONE: L'uso di **break** e **continue** NON è consentito

Esercizio 1. (max 10 punti) Si consideri il seguente problema:

INPUT: Un grafo $G = (V, E)$ indiretto e connesso (con $V = \{1, 2, \ldots, n\}$), una funzione di pesi positiva $c : E \to R^+$ sugli archi di G, e un sottoinsieme $E' \subseteq E$ di archi disgiunti (ovvero che non condividono nessun estremo).
OUTPUT: Un albero di copertura che includa gli archi in E' e minimizzi la somma dei costi degli archi non in E'.

Stabilire e descrivere quale struttura dati si intende utilizzare per rappresentare l'albero di copertura e progettare un algoritmo per risolvere il problema.

Descrivere brevemente l'algoritmo a parole e fornire lo pseudo-codice che sia coerente con la scelta fatta per la rappresentazione dell'albero di copertura.

Discutere il costo computazionale dell'algoritmo.

Esercizio 2. (max 11 punti)

Progettare un algoritmo ricorsivo basato sulla tecnica **Divide et Impera** che calcola il secondo elemento più grande di un array A di naturali (non necessariamente ordinato), senza modificarlo o copiarlo in un altro array.

Descrivere il caso base e il caso ricorsivo, e fornire lo pseudocodice per l'algoritmo. Scrivere la relazione di ricorrenza che descrive il costo computazionale nel caso peggiore e risolverla.

Domanda teorica (in alternativa ad uno dei due esercizi) (max 5 punti)

(2 punti) Descrivere un algoritmo per la costruzione di un Minimo Albero di Copertura per un grafo indiretto $G = (V, E)$ e (3 punti) spiegare perché è corretto.

SOLUZIONI SECONDA PARTE

SOLUZIONE Esercizio 1.

L'idea alla base della soluzione è che gli archi in E' vadano considerati tutti come archi del minimum spanning tree (MST), come da richiesta, e a questi poi vada aggiunga una selezione degli altri archi (ovvero quelli in $E \setminus E'$) che vadano a completare il MST e la cui somma totale sia la minima tra tutte le selezioni possibili. Osserviamo che gli archi in E' sono tutti disgiunti (ovvero non condividono estremi) e quindi non possono formare cicli.

Per per scegliere gli archi di $E \setminus E'$ si possono usare entrambi gli algoritmi visti a lezione. Tutte le soluzioni hanno lo stesso costo computazionale dell'algoritmo scelto per la costruzione del MST. Per **esercizio**, argomentate questa ultima affermazione sul costo coputazionale.

VERSIONE 1

Per far selezionare tutti gli archi in E' basta assegnargli un costo minore di tutti gli altri. Poiché i pesi della funzione data in input sono positivi, basta scegliere un qualunque numero negativo, per esempio -1. Poi si può invocare un algoritmo standard per il MST e restituire il risultato di questa invocazione.

Poiché gli archi di E' sono disgiunti verranno sicuramente selezionati dagli algoritmi per MST. **Esercizio:** Provate a pensare al perché (la soluzione alla fine).

Se si usa l'algoritmo di Kruskal la struttura dati per rappresentare l'albero è una lista di archi, se si usa Prim, invece, l'array dei padri.

```
MinAlberoCoperturaK(G = (V,E), c, E')
   for all e ∈ E' do
       c(e) := -1
   return Kruskal(G,c)
```

oppure

```
MinAlberoCoperturaP(G = (V,E), c, E')
   for all e ∈ E' do
       c(e) := -1
   return Prim(G,c)
```

Osservazione: invece di modificare la funzione c se ne può definire una nuova (che viene poi passata a Kruskal/Prim al posto di c) che è identica a c per tutto gli archi in $E \setminus E'$ e uguale a -1 per tutti gli archi in E'.

VERSIONE 2 Modificare l'algoritmo di Kruskal

Poichè gli archi di E' devono stare nell'albero, basta modificare l'algoritmo di Kruskal facendogli selezionare subito quegli archi (che come detto prima non formeranno cicli) e poi usare l'algoritmo standard per selezionare gli altri. L'unica accortezza sta nell'inizializzare opportunamente la variabile `count` che conta il numero di archi che sono già stati selezionati per tenere conto di quelli di E' (ovvero, il numero di archi che devono essere selezionati in $E \setminus E'$ sarà $|V| - 1 - |E'|$ e non $|V| - 1$).

L'algoritmo di Kruscal memorizza gli archi selezionati per il MST in una lista di archi, quindi questa lista sarà l'output dell'algoritmo.

Per selezionare subito gli archi di E' si deve fare esattamente quello che fa Kruskal per inserire un arco nel MST con ognuno degli archi di E'. Gli archi di E' devono essere inseriti nella lista degli archi selezionati e si deve aggiornare la struttura dati Disjoint Set per evitare che, successivamente, si vadano a formare dei cicli.

Per migliorare la leggibilità si può scrivere una procedura che, dato un arco, genera un nuovo nodo lista, lo inizializza opportunamente e lo inserisce in testa alla lista degli archi selezionati.

In rosso le differenze con l'algoritmo di Kruskal visto a lezione (a parte l'intestazione della funzione).

```
MinAlberoCoperturaK(G = (V,E), c, E')
   S := make_set(V)
   T := new_list()
   for all (u,v) ∈ E' do
       InsertNode(T,(u,v))
       union(S,u,v)
   ordina gli archi di E\E' per ordine di peso crescente
   count := |E'|
   while count < n-1 do
      scegli il prossimo arco (u,v) ∈ E\E' in base all'ordinamento
      if find-set(S,u) ≠ find-set(S,v)
         then
           InsertNode(T,(u,v))
           union(S,u,v)
           count := count + 1
   return T


InsertNode(L,e)
  p := new_list_node()
  p.val := e
  p.next := L
  L := p
```

Osservazione: l'algoritmo funziona correttamente anche se, dopo aver selezionato gli archi di E', si lavora con tutti gli archi di E e non solo con quelli di $E \setminus E'$, soltanto sarà, nel caso peggiore, meno efficiente. Infatti, quando nel ciclo verrà selezionato un arco di E', gli estremi risulteranno già nella stessa componente connessa e l'arco non verrà selezionato nuovamente. Quindi, nello pseudocodice precedente, rispetto alla versione vista a lezione, rimarrebbe solo il for evidenziato in rosso.

VERSIONE 3 - Modificare l'algoritmo di Prim

Scegliendo l'algoritmo di Prim le cose sono un pò più complicate perché gli archi di E' non si possono selezionare a priori, ma devono essere selezionati nel momento in cui si arriva in uno dei loro estremi.

Quindi, se viene estratto dalla coda un nodo u che è l'estremo di un arco (u, v) che appartiene a E', facciamo in modo che all'iterazione successiva esca dalla coda il nodo v che è l'altro estremo dell'arco (in questo modo anche v andrà a far parte del MST e sarà collegato attraverso l'arco (u, v)).

Per far questo, impostiamo u come padre di v e modifichiamo la priorità di v a un valore negativo (per esempio -1). Poichè i costi associati agli altri nodi sono necessariamente positivi (v è unico perché gli archi in E' sono disgiunti),

all'iterazione successiva il nodo v avrà prioprità più grande di tutti e verrà estratto dalla coda. In altre parole, è come se stessimo imbrogliando facendo finta che il costo dell'arco (u, v) sia sempre stato negativo.

La struttura dati utilizzata da Prim per rappresentare il MST è l'array dei padri, quindi anche il nostro algoritmo restituirà l'albero di copertura minima sotto forma di array dei padri. Scegliamo di iniziare la costruzione dell'albero dal nodo 1 (ogni altra scelta va bene).

In rosso le differenze con l'algoritmo di Prim visto a lezione (a parte l'intestazione della funzione).

```
MinAlberoCoperturaP(G = (V,E), c, E')
   for all v ∈ V do
       cost[v] := +∞
       prev[v] := 0
       S[v] := 0
   cost[1] := 0
   Q := make_priority_queue({ (v,cost[v]) | v ∈ V })
   while NOT is_empty_queue(Q) do
      u := DeQueue(Q)
      S[u] := 1
      for all (u,v)∈ E do
          if S[v]=0 AND (u,v) ∈ E' // ERRATA CORRIGE: prima era E \ E' ed era sbagliato
             then
               cost[v] := -1
               prev[v] := u
               Decrease_Priority(Q,v,cost[v])
             else
               if S[v]=0 AND cost[v] > c(u,v)
                  then
                    cost[v] := c(u,v)
                    prev[v] := u
                    Decrease_Priority(Q,v,cost[v])
   return prev[v]
```

Soluzione esercizio lasciato nella versione 1: supponiamo che uno degli archi (u, v) di E' non venga selezionato per il MST. Se aggiungiamo (u, v) all'albero otteniamo un ciclo. Togliamo un arco qualunque di questo ciclo che non sia in E' (almeno uno deve esistere perché gli archi di E' non hanno estremi in comune): il risultato è ancora un albero di copertura, ma di costo inferiore a prima, perchè abbiamo tolto un arco di costo non negativo e l'abbiamo sostituito con un arco di costo negativo. Fate un piccolo esempio per visualizzare il ragionamento.

ERRORI PIÙ COMUNI:

- Confondere l'albero dei cammini minimi con l'albero minimo di copertura, quindi usare Dijkstra invece di Prim.
- Usare la BFS per determinare l'albero di copertura minima
- Confondere la struttura dati che memorizza l'albero con la Disjoint Set.
- Risolvere il problema invocando Prim o Kruskal su G, questo non garantisce che gli archi di E' vengano selezionati tutti.

SOLUZIONE Esercizio 2.

Ci sono almeno due modi di approcciare il problema utilizzando la tecnica Divide&Impera, che si differenziano nel modo in cui viene suddiviso il problema in sottoproblemi. Per entrambi i casi assumiamo che, nel caso in cui non ci sia un secondo massimo (e questo può succedere solo quando tutti i valori sono uguali, o c'è un elemento solo), il risultato sia -1 (ovvero un valore che non può comparire nell'array).

Versione 1

Scriviamo una funzione `Secondo` che, su input una sottosequenza $K[i..j]$ di un array K, restituisce una coppia di valori (p, s) tali che p è il massimo valore nella sottosequenza e s il secondo. La funzione sfrutta la tecnica Divide&Impera, è ricorsiva e nel passo Divide partiziona la sottosequenza in due sottosequenze di lunghezza la metà di quella di partenza. Il problema dell'esercizio si risolve invocando la funzione `Secondo` su $A[0..n-1]$ e dando in output il secondo valore della coppia restituita da questa invocazione.

In particolare, la funzione `Secondo` prende in input un array K e due indici i, j nel range $[0..n-1]$ (dove n è la dimensione dell'array):

> **Casi base:** (1) la porzione $K[i..j]$ è vuota (ovvero $i > j$), si restituisce la coppia $(-1, -1)$ ad indicare che non esiste né il massino e né il secondo massimo; (2) la porzione $K[i..j]$ contiene un solo elemento (ovvero $i = j$), si restituisce la coppia $(K[i], -1)$ ad indicare che esiste solo il massimo.
>
> **Caso ricorsivo:** $K[i..j]$ contiene almeno due elementi ($j > i$). (*Divide:*) calcola l'indice centrale m (nel solito modo) della porzione di array $K[i..j]$. (*Impera:*) risolve ricorsivamente il problema sulle sottosequenze $K[i..m]$ e $K[m+1..j]$ e ottiene due coppie (p_1, s_1) e (p_2, s_2) dall'esecuzione delle due chiamate, rispettivamente. (*Combina:*) per calcolare la coppia da restituire si devono calcolare il massimo e il secondo massimo tra i valori p_1, p_2, s_1, s_2, facendo attenzione al caso in cui ci siano valori uguali[1]:
>
> - se $p_1 > p_2$, allora il massimo per la sottosequenza $K[i..j]$ è p_1 e il secondo massimo è il valore più grande tra s_1 e p_2;
> - se $p_2 > p_1$, analogamente a prima, abbiamo che il massimo per la sottosequenza $K[i..j]$ è p_2 e il secondo massimo è il valore più grande tra s_2 e p_1;
> - se $p_1 = p_2$, allora il massimo per la sottosequenza $K[i..j]$ è p_1 (o p_2, tanto sono uguali) e il secondo massimo è il valore più grande tra s_1 e s_2 (funziona anche nel caso in cui siano uguali)
>
> Provare con un piccolo esempio. Funziona anche quando uno o due valori della coppia sono -1.

```
TrovaSecondoMassimo(A)
  (p,s) := Secondo(A)
  return s


Secondo(K,i,j)
  if j < i then return (-1, -1)
  if i = j then return (K[i], -1)
  m := ⌈ (i+j) /2 ⌉
  (p1, s1) := Secondo(K,i,m)
  (p2, s2) := Secondo(K,m+1,j)
```

[1] Nella correzione non sono state penalizzate soluzioni che funzionavano solo nel caso in cui i valori nell'array fossero tutti diversi, ma sono state premiate quelle che lo funzionavano anche in quel caso.

```
  if p1 > p2 then return (p1, max{s1,p2})
  if p2 > p2 then return (p2, max{p1,s2})
  return (p1, max{s1,s2})
```

Per l'equazione di ricorrenza: visto che la funzione viene chiamata due volte su sottosequenze che hanno dimensione la metà di quella di partenza e che il lavoro di Divide e Combina è costante (così come il caso base), abbiamo:

$$T(n) = \begin{cases} O(1) & \text{se } n < 2 \\ 2T\left(n/2\right) + O(1) & \text{se } n \geq 2 \end{cases}$$

Da cui $T(n) \in O(n)$ usando il Master Theorem.

Versione 2

Scriviamo una funzione `TrovaSecondoMassimo` che *simula* una scansione iterativa dell'array. Ovvero, la ricorsione viene invocata su una sottosequenza dell'array che è più piccola di un elemento rispetto a quella originale.

La funzione prende in input un array K, la dimensione dell'array originale $n = length(K)$, l'indice del primo elemento nella sottosequenza corrente (e lavorerà sulla sottosequenza $K[i..n-1]$), e due valori m_1 e m_2 (che rappresentano il primo e il secondo massimo della sottosequenza $K[0..i-1]$). Restituisce il secondo massimo tra i valori nella sottosequenza $K[i..n-1]$, m_1 e m_2.

> **Caso base:** la sottosequenza $K[i..n-1]$ è vuota (ovvero $i > n-1$), non ci sono più elementi da esaminare, quindi si restituisce m_2 (che è il secondo massimo tra i valori della sottosequenza $K[i..n-1]$, che è vuota, e m_1 e m_2).
>
> **Caso ricorsivo:** la sottosequenza $K[i..n-1]$ contiene almeno un elemento (ovvero $i \leq n-1$). (*Divide*:) Si effettua una chiamata ricorsiva sulla sottosequenza $K[i+1..n-1]$. Per stabilire i parametri della chiamata al sottoproblema si confronta l'elemento in posizione i con i valori m_1 e m_2 dati in input:
>
> - Se $A[i] > m_1$, allora $A[i]$ è il massimo tra i valori in $K[0..i]$ e m_1 diventa il secondo massimo.
> - Se $m_1 > A[i] > m_2$, allora m_1 è ancora il massimo tra i valori in $K[0..i]$, ma $A[i]$ diventa il secondo massimo.
> - Altrimenti (comprende i casi: $A[i] < m_2, A[i] = m_1$ e $A[i] = m_2$), la situazione non è cambiata, il massimo e il secondo massimo in $K[0..i]$ sono ancora m_1 e m_2.
>
> *Impera:* risolve ricorsivamente il problema sulla sottosequenza $K[i+1..n-1]$; *Combina:* restituire il valore restituito dalla chiamata ricorsiva.

```
TrovaSecondoMassimo(K,i,len,m1,m2)
  if i < len
     then
       if K[i] > m1
          then return Secondo(K,i+1,len,K[i],m1)
       if m1 > K[i] > m2
          then return Secondo (K,i+1,len,m1,K[i])
          else return Secondo(K,i+1,len,m1,m2)
  return m2
```

Chiamata principale `TrovaSecondoMassimo(A,0,n,A[0],-1)`.

$$T(n) = \begin{cases} O(1) & \text{se } n \leq 1 \\ T(n-1) + O(1) & \text{se } n \geq 2 \end{cases}$$

In questo caso non si può usare il Master Theorem perché la chiamata ricorsiva viene invocata su una dimensione che non è una frazione costante della dimensione originale. Per cui, per risolverla si deve procedere con il metodo iterativo (fare riferimento alla lezione sulla ricorsione o le dispense). Il risultato è anche in questo caso $T(n) \in O(n)$.

ERRORI PIÙ COMUNI:

- Dividere l'array in due parti, cercare il massimo in entrambe, restituire il minimo dei massimi. Questo valore potrebbe non essere il secondo massimo dell'array, perché il massimo e il secondo massimo potrebbero essere nella stessa metà. Esempio: A = <1,2,3,4,5,6,7,8>, il massimo della metà di sinistra è 4, il massimo della metà di destra è 8, il minimo tra i due numeri è 4, ma il secondo massimo dell'array A è 7 e non 4.
- Calcolare il secondo massimo delle metà di destra e di sinistra, restituire il minimo dei due. Potrebbe essere un numero molto lontano dal secondo minimo, soprattuto se fatto ricorsivamente. Esempio (senza ricorsione): A = <1,2,3,4,5,6,7,8>, il secondo massimo della metà di sinistra è 3, il secondo massimo della metà di destra è 7, il minimo dei due è 3, ma il secondo massimo dell'array A è 7 e non 3.
- Scrivere un algoritmo iterativo
- Non utilizzare la tecnica Divide&Impera correttamente
- Assumere che l'array sia già ordinato in partenza
- Dare una descrizione dei casi base/ricorsivo diversa da quello che viene fatto nello pseudo-codice
- Dimenticare la chiamata principale
- Nella descrizione dei casi: (1) non indicare quali siano i casi base e/o quello ricorsivo; (2) non spiegare cosa si deve fare nel caso base e/o ricorsivo.
- Invocare la funzione sulle due metà facendo precedere l'invocazione dall'istruzione `return`: in questo caso la seconda invocazione non verrà mai eseguita, perché il flusso di esecuzione termina alla fine della prima chiamata.

Quarto Appello
9 Settembre 2021

SECONDA PARTE - ATTENZIONE: L'uso di **break** e **continue** NON è consentito

Esercizio 1. (max 9 punti) Si consideri il seguente problema:

INPUT: Due grafi **diretti** $G_1 = (V, E_1)$ e $G_2 = (V, E_2)$, con $V = \{1, 2, \ldots, n\}$.
OUTPUT: Il (nuovo) grafo $G = (V, E)$ **unione** di G_1 e G_2, ovvero tale che $E = E_1 \cup E_2$.

Fare un esempio di input e output del problema con $n = 5$.

Scrivere lo pseudocodice di un algoritmo per risolvere il problema supponendo che i grafi (sia di input che di output) siano realizzati tramite **liste di adiacenza**, senza sovrascrivere o modificare i grafi originali, e senza utilizzare le primitive viste a lezione per le liste.

Discutere il costo computazionale dell'algoritmo.

Esercizio 2. (max 12 punti)

La **vista sinistra** di un albero binario è data dalla lista dei nodi visibili quando l'albero viene osservato dal lato sinistro, ordinata per livello crescente (vedere esempio in Figura 1).

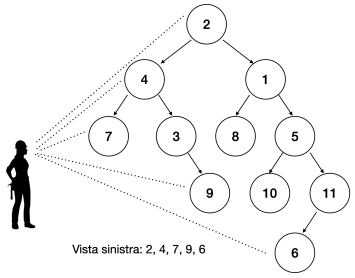

Figura 1: Esempio di vista sinistra di albero binario

Si consideri il seguente problema:

INPUT: un albero binario T realizzato con nodi e puntatori (campi *key, left, right*, con significato standard).
OUTPUT: la stampa delle chiavi dei nodi della vista sinistra dell'albero.

Descrivere brevemente a parole un algoritmo per risolvere il problema e fornirne lo pseudocodice. Studiare il costo computazionale dell'algoritmo proposto.

Domanda teorica (in alternativa ad uno dei due esercizi) (max 5 punti)

(1 punto) Definire il problema di *String Matching* visto a lezione, (3 punti) spiegare come funziona *l'algoritmo di forza bruta* per risolvere il problema e (1 punto) discutere il costo computazionale dell'algoritmo.

SOLUZIONI SECONDA PARTE

SOLUZIONE Esercizio 1.

Per tutte le versioni è comodo avere una primitiva che inserisce un nuovo nodo in testa ad una lista, dati il puntatore alla lista in cui inserirlo e il valore che deve essere memorizzato dal nodo. Questa è una variante dell'inserzione in testa che abbiamo definito a lezione (in cui si dava il puntatore al nodo da inserire invece che la chiave del nodo da inserire). Scriviamo la primitiva una volta per tutte le versioni:

```
InserisciInTesta(L,k)
  n := new_list_node()
  n.val := k
  n.next := L
  L := n
```

Per la valutazione dei costi computazionali assumeremo che $|V| = n$, $|E_1| = m_1$ e $|E_2| = m_2$.

Versione 1

Per la lista di adiacenza di ogni nodo i (con $i \in [1..n]$) l'algoritmo lavora in due passi:

1. copia $G_1[i]$ nel nuovo grafo G, inizialmente vuoto;
2. fa il merge di $G[i]$ con $G_2[i]$ facendo attenzione a non inserire due volte lo stesso arco.

Il primo passo viene implementato con un ciclo sulla lista $G_1[i]$ in cui, ad ogni iterazione, si aggiunge un nuovo nodo in testa a $G[i]$. Il secondo passo viene implementato con due cicli annidati: il più esterno scorre la lista $G_2[i]$ e, per ogni nodo, controlla se il valore è già presente in $G[i]$ o no. Questo controllo viene implementato con un ciclo interno in cui si scorre $G[i]$.

```
Unione(G1, G2)
  G[1..n] new array di liste
  for i = 1 to n do
      Copy(G, G1, i)
      Merge(G ,G2, i)
  return G


Copy(G,G1,i)
  p := G1[i]
  while p ≠ NIL do
    InserisciInTesta(G[i], p.val)
    p := p.next


Merge(G1,G2,i)
  if G2[i] = NIL then return // se la seconda lista e' vuota, non si deve fare niente
  if G1[i] = NIL // se la prima lista e' vuota, basta copiare la seconda lista
     then
```

```
    Copy(G1,G2,i)
    return
// le liste non sono vuote, quindi si fa il merge delle liste evitando i duplicati
p:= G2[i]
while p ≠ NIL do
  tmp := G1[i]
  trovato := FALSE
  while tmp ≠ NIL OR NOT trovato do
    if tmp.val = p.val
       then trovato := TRUE
       else tmp := tmp.next
  if tmp = NIL then InserisciInTesta(G1[i], p.val)
  p := p.next
```

Costo computazionale. Osserviamo che:

- Un'esecuzione della procedura `Copy` scorre le liste di adiacenza di un nodo di un grafo (quello passato come secondo parametro) e esecuzioni diverse scorrono liste di adiacenza di nodi diversi, quindi tutte le esecuzioni di `Copy` scorrono esattamente una volta la lista di ogni nodo del grafo. Per ogni nodo delle liste di adiacenza (che corrisponde ad un arco nel grafo corrispondente) `Copy` lavora per un tempo costante, quindi tutte le esecuzioni di `Copy` con secondo parametro G_1 costano $O(m_1)$.
- La procedura `Merge` ha come caso migliore quello in cui la lista da aggiungere è vuota. Negli altri casi:
  - se la lista a cui aggiungere nodi è vuota, si paga il costo di `Copy`, che dipende dal numero di nodi nella lista da aggiungere;
  - altrimenti, si scorre la lista da aggiungere e, nodo per nodo, si decide se inserirlo nella lista finale scorrendo la lista corrispondente dell'altro grafo. Il costo di questa operazione dipende dal numero di archi uscenti nei nodi dei due grafi. In particolare, dato il nodo i, sia $deg_1(i)$ (rispettivamente $deg_2(i)$) il grado del nodo in G_1 (rispettivamente G_2), allora il costo dipende linearmente da $deg_1(i) \cdot deg_2(i)$.

  Quindi, nel caso peggiore, tutte le esecuzioni di `Merge` costano $O(\sum_i (deg_1(i) \cdot deg_2(i)))$. Questa quantità è difficile da valutare esattamente e può variare molto da grafo a grafo, però nel caso peggiore, le liste di adiacenza possono essere lunghe $n-1$ nodi (grafo completo), quindi abbiamo che $deg_1(i) \cdot deg_2(i) \in O(n^2)$ e la sommatoria è una somma di n termini, per cui

$$O(\sum_i (deg_1(i) \cdot deg_2(i))) \in O(n^3).$$

Questi costi devono essere sommati tra di loro, quindi abbiamo che in totale il costo computazionale nel caso peggiore è:

$$O(m_1) + O(n^3) \in O(n^3).$$

Versione 2

Per la lista di adiacenza di ogni nodo i (con $i \in [1..n]$) l'algoritmo lavora in due passi:

1. copia $G_1[i]$ nel nuovo grafo G, inizialmente vuoto, e ricorda in un array gli identificativi dei nodi estremi degli archi uscenti da i. Poiché gli identificativi dei nodi sono gli interi tra 1 e n, questa operazione può essere implementata semplicemente indicizzando l'array con gli identificativi dei nodi.
2. fa il merge di $G[i]$ con $G_2[i]$ facendo attenzione a non inserire due volte lo stesso arco. Questo controllo viene fatto usando l'array del passo 1.

```
Unione(G1, G2)
  G[1..n] new array di liste
  K[1..n] new array di booleani - globale
  for i = 1 to n do
      for i=1 to n do
          K[i] := 0
      Copy(G, G1, i)
      Merge(G, G2, i)
  return G


Copy(Gr1, Gr2, i)
  p := Gr2[i]
  while p ≠ NIL do
    InserisciInTesta(Gr1[i], p.val)
    K[p.val] := 1
    p := p.next


Merge(Gr1, Gr2, i)
  if Gr2[i] = NIL then return // se la seconda lista e' vuota, non si deve fare niente
  if Gr1[i] = NIL // se la prima lista e' vuota, basta copiare la seconda lista
     then
       Copy(Gr1, Gr2,i)
       return
  // le liste non sono vuote, quindi si fa il merge delle liste evitando i duplicati
  p:= Gr2[i]
  while p ≠ NIL do
    if K[p.val] = 0
       then // l'arco non e' gia' presente e lo si inserisce in testa, altrimenti no
         InserisciInTesta(Gr1[i], p.val)
    p := p.next
```

Costo computazionale. Osserviamo che:

- L'inizializzazione dell'array `K` costa $O(n)$ e viene eseguita esattamente n volte, per un costo totale di $O(n^2)$.
- Un'esecuzione della procedura `Copy` scorre le liste di adiacenza di un nodo di un grafo (quello passato come secondo parametro) e esecuzioni diverse scorrono liste di adiacenza di nodi diversi, quindi tutte le esecuzioni di `Copy` scorrono esattamente una volta la lista di ogni nodo del grafo. Per ogni nodo delle liste di adiacenza (che corrisponde ad un arco nel grafo corrispondente) `Copy` lavora per un tempo costante, quindi tutte le esecuzioni di `Copy` con secondo parametro G_1 costano $O(m_1)$.

- La procedura `Merge` ha come caso migliore quello in cui la lista da aggiungere è vuota. Negli altri casi:
  - se la lista a cui aggiungere nodi è vuota, si paga il costo di `Copy`, che dipende dal numero di nodi nella lista da aggiungere;
  - altrimenti, si scorre la lista da aggiungere e, nodo per nodo, si decide se inserirlo nella lista finale. Il costo di questa operazione dipende, nuovamente, dal numero di nodi presenti nella lista da aggiungere (perché controllare se un nodo è si fa in tempo costante).

  Quindi, nel caso peggiore, tutte le esecuzioni di `Merge` costano $O(m_2)$.

Questi costi devono essere sommati tra di loro, quindi abbiamo che in totale il costo computazionale è:

$$O(n^2) + O(m_1) + O(m_2) = O(n^2 + m_1 + m_2).$$

Osserviamo che questa versione ha anche un costo computazionale in spazio che non è costante, ma $\Theta(n)$.

Versione 3

Per ogni nodo i (con $i \in [1..n]$) l'algoritmo lavora in due passi:

1. Scorre le liste di adiacenza di i in G_1 e G_2 e ricorda in un array gli identificativi dei nodi estremi degli archi uscenti da i (ovviamente se lo stesso estremo compare due volte non è rilevante, basta ricordarlo una volta sola).
2. Genera la lista di adiacenza in G scorrendo l'array e creando un nodo solo per gli identificativi che risultano essere stati ricordati al passo 1. Durante la scansione l'array viene reinizializzato in modo che possa essere riutilizzato per il nodo successivo.

```
Unione(G1, G2)
  G[1..n] new array di liste
  K[1..n] new array di booleani - globale
  for i=1 to n do
      K[i] := 0
  for i = 1 to n do
      Edges(G1,i)
      Edges(G2,i)
      AdjacentList(G,i)
  return G


Edges(Gr,i)
  p := Gr[i]
  while p ≠ NIL do
    K[p.val] := 1
    p := p.next



AdjacentList(Gr,i)
  for i = 1 to n do
```

```
if K[p.val] = 1
then
  InserisciInTesta(Gr[i], p.val)
  K[p.val] := 0
```

Costo computazionale. Osserviamo che:

- L'inizializzazione dell'array K costa $O(n)$ e viene eseguita esattamente una volta.
- Un'esecuzione della procedura `Edges` scorre la lista di adiacenza di un nodo di uno dei due grafi di input ed esecuzioni diverse scorrono liste di adiacenza di nodi diversi, quindi tutte le esecuzioni di `Edges` scorrono esattamente una volta la lista di ogni nodo nei due grafi di input. Per ogni nodo delle liste di adiacenza (che corrisponde ad un arco nel grafo corrispondente) `Edges` lavora per un tempo costante, quindi tutte le esecuzioni di `Edges` costano $O(m_1 + m_2)$.
- La procedura `AdjiacentList` esegue un ciclo for n volte e, ad ogni iterazione, effettua un numero di operazioni costanti, quindi il suo costo totale è $O(n)$. Poiché viene ripetuta esattamente n volte all'interno del ciclo di `Unione( , )`, il costo totale di tutte le sue esecuzioni è $O(n^2)$.

Questi costi devono essere sommati tra di loro, quindi abbiamo che il costo computazionale totale è

$$O(n) + O(m_1 + m_2) + O(n^2) = O(n^2 + m_1 + m_2).$$

Osserviamo che questa versione ha anche un costo computazionale in spazio che non è costante, ma $\Theta(n)$.

ERRORI PIÙ COMUNI:

- Non usare la rappresentazione con liste di adiacenza
- Non fare avanzare il/i puntatore/i che scorrono le liste
- Non creare un nuovi nodi da inserire nel nuovo grafo, ma modificare i puntatori delle liste di adiacenza dei grafi in input
- Non gestire correttamente l'avanzamento nelle liste
- Qualcuno ha assunto che le liste di adiacenza del grafo fossero ordinate. Non era esplicitato nella traccia, ma è un'assunzione ragionevole (che dipende ovviamente da come viene implementato il grafo con le liste di adiacenza). Quello che non si può fare, però, è costruire il nuovo grafo senza ordinare le liste di adiacenza. Questo perché il grafo unione è una struttura dati dello stesso tipo di quelle date in ingresso e, quindi, una proprietà che vale per gli ultimi deve valere anche per il primo. Supponiamo, per esempio, che il grafo unione venga successivamente unito con un altro grafo: se le sue liste di adiacenza non sono ordinate, la seconda unione non andrà a buon fine e si genererà in output un grafo che non è l'unione dei due dati in input.

SOLUZIONE Esercizio 2.

Il problema si può risolvere con una visita dell'albero, sia BFS che DFS. La BFS forse è una scelta più naturale perché lavora per livelli e, per risolvere il problema, è necessario stampare un nodo per livello.

In tutte le soluzioni proposte il costo computazionale è quello della visita dell'albero.

Soluzioni con DFS

La DFS deve essere in preordine perché la radice sta ad un livello inferiore dei figli e deve essere stampata prima. Inoltre, si deve prima scendere nel sottoalbero sinistro e poi in quello destro. È importante scendere sempre anche in quello destro perché quel sottoalbero potrebbe essere più alto di quello sinistro e, quindi, avere dei nodi che sono visibili da sinistra (come il nodo 6 nell'esempio. Se non si scendesse mai nel sottoalbero destro della radice, non verrebbe mai trovato e quindi mai stampato).

Versione 1

Effettuiamo una DFS dell'albero "portandoci dietro" il livello dell'ultimo nodo che è stato stampato. Scriviamo una funzione ricorsiva `VistaSinistra` che prende in **input**:

- il puntatore alla radice di un sottoalbero t,
- il livello ℓ in T di t
- il livello s dell'ultimo nodo di T che è stato stampato prima dell'inizio della visita del sottoalbero radicato in t.

In **output** la funzione restituisce il livello in T dell'ultimo nodo del sottoalbero radicato in t che è stato stampato.

> **Caso base:** l'albero t a livello ℓ in T è una foglia: se ℓ è maggiore di s (osservazione, può essere solo $\ell = s+1$), allora la foglia t si vede da sinistra, si stampa il suo valore, e si restituisce il livello di t, ovvero ℓ.

Caso ricorsivo: l'albero t a livello ℓ in T non è una foglia: si effettua lo stesso controllo (ed eventuali azioni) del caso base, in più si chiama la visita ricorsivamente sugli eventuali figli. La prima chiamata viene fatta sul *figlio sinistro*, che ha livello in T pari a $\ell + 1$ e livello dell'ultimo nodo stampato uguale a ℓ se è stato stampato il valore di t, s altrimenti. Il risultato di questa chiamata viene memorizzato e viene utilizzato per definire l'ultimo parametro della chiamata al figlio destro, se il figlio destro non è vuoto, altrimenti viene restituito come risultato della chiamata ricorsiva. L'eventuale chiamata al *figlio destro* viene invocata con secondo parametro $\ell + 1$ (come nel caso della chiamata al figlio sinistro), mentre l'altezza dell'ultimo nodo stampato è dato dal risultato della chiamata sul figlio sinistro (se è stata fatta), altrimenti dall'altezza di t se il valore di t è stato stampato, altrimenti dal valore del parametro di input s. Il risultato della chiamata ricorsiva al figlio destro (se effettuata) viene restituito come risultato della chiamata su t.

```
Principale(T)
  if T = NIL
     then
       print "albero vuoto"
       return
  v := VistaSinistra(T,0,-1)

VistaSinistra(t,ℓ,s)
   if ℓ > s
      then
        print t.val
        s := ℓ
   k := s
   if t.left ≠ NIL then k := VistaSinistra(t.left, ℓ+1, s)
   if t.right ≠ NIL then return VistaSinistra(t.right, ℓ+1,k)
   return k
```

Osservazione: la funzione `VistaSinistra` non viene mai invocata su un albero vuoto. Il controllo su T viene fatto da una procedura principale che invoca `VistaSinistra` solo nel caso in cui T non sia nullo. Inoltre, il valore restituito dalla chiamata principale di `VistaSinistra` nella procedura principale non viene utilizzato perché non richiesto (ma la funzione lo calcola lo stesso e dovrebbe essere uguale all'altezza dell'albero più uno (perché?)).

Versione 2

Effettuiamo una DFS dell'albero con l'ausilio di un array globale che viene utilizzato per ricordare per quale livello dell'albero è già stato stampato il nodo più a sinistra. L'array deve essere dimensionato in modo da non "perdere" nessun livello, che nel caso peggiore sono esattamente tanti quanti i nodi dell'albero (nel caso in cui l'albero sia degenere e sia una catena di n nodi). In alternativa, si può calcolare l'altezza esatta dell'albero con la funzione vista durante il corso.

Quindi avremo un array globale `V` tale che: `V[i] = 1` se il nodo più a sinistra del livello `i` dell'albero è già stato stampato, `V[i] = 0` altrimenti, per $i = 1, 2, \ldots, n$. L'array deve essere, ovviamente, inizializzato con tutte le celle a zero.

Effettuando una visita DFS in pre-ordine e scendendo sempre nel sottoalbero di sinistra prima che in quello di destra, quando si incontra un nodo su un livello ancora non visitato, questo nodo sarà proprio quello da stampare (ovvero il più a sinistra del livello). Quindi, la visita DFS standard deve essere modificata in modo che ogni nodo visitato sappia a che livello dell'albero si trovi.

Modifichiamo, quindi la DFS, in modo che prenda in input non solo il puntatore t alla radice di un sottoalbero, ma anche un intero che è il livello ℓ della radice del sottoalbero radicato in t nell'albero. La prima chiamata verrà fatta sulla radice dell'albero con livello zero.

> **Caso base:** l'albero t a livello ℓ è una foglia: se `V[`ℓ`] = 1` non si deve fare niente, altrimenti si stampa $t.val$ e si aggiorna `V[`ℓ`]` a uno;
>
> **Caso ricorsivo:** l'albero t a livello ℓ non è una foglia: si effettua lo stesso controllo (ed eventuali azioni) del caso base, in più si chiama la visita ricorsivamente sugli eventuali figli (il sinistro per prima), ricordandosi di incrementare il livello di uno.

Una chiamata principale controlla che l'albero T non sia vuoto, inizializza l'array `V` globale, e invoca la chiamata principale della DFS ricorsiva.

```
Principale(T)
  if T = NIL
     then
       print "albero vuoto"
       return
  V[0..n-1] new array di booleani - varibabile globale
  for i=1 to n do
      V[i] := 0
  VistaSinistra(T,0)

VistaSinistra(t,ℓ)
  if V[ℓ] = 0
     then
       print t.val
       V[ℓ] := 1
  if t.left ≠ NIL then VistaSinistra(t.left,ℓ+1)
  if t.right ≠ NIL then VistaSinistra(t.right,ℓ+1)
```

Osservazione: il costo dell'inizializzazione dell'array `V` non aumenta il costo computazionale asintotico della visita DFS, perché è un costo additivo pari a $O(n)$.

Soluzioni con BFS

Bisogna stampare solo un nodo per livello e questo nodo è il primo nodo di ogni livello. Quindi bisogna modificare la BFS vista a lezione per riuscire ad accorgersi di quando si estrae dalla coda il primo nodo di un livello. Questo può essere fatto in almeno due modi diversi.

Versione 1

Per accorgersi di quando si estrae il primo nodo di un nuovo livello dalla coda, modifichiamo la visita BFS vista a lezione utilizzando la coda per ricordare una coppia di valori e non solo i nodi dell'albero.

In particolare, le coppie del tipo (u, ℓ) che mettiamo in coda contengono un nodo dell'albero u e il suo livello ℓ nell'albero. Utilizziamo una variabile per ricordare il livello correte (ovvero quello al quale siamo arrivati durante la visita) e, ad ogni iterazione, quando estraiamo una coppia dalla coda controlliamo il suo livello:

- se è uguale a quello corrente non dobbiamo stampare il valore memorizzato nel nodo, ma solo procedere ad inserire i suoi eventuali figli in coda (facendo attenzione a impostare correttamente il loro livello, che deve essere uno in più del livello del nodo che stiamo analizzando);
- se il livello del nodo è diverso da quello corrente (può essere solo uno in più del livello corrente), allora si procede a stampare il valore memorizzato nel nodo e poi a inserire gli eventuali figli in coda, come nell'altro caso.

```
VistaSinistra(T)
   if T = NIL
      then
        print "albero vuoto"
        return
   Q := new_queue()
   enqueue(Q,(T,0))
   current-level := -1
   while NOT empty_queue(Q) do
       (u,ℓ) := dequeue(Q)
       if ℓ > current-level
          then // u e' il primo nodo di un nuovo livello
            print u.val
            current-value := ℓ
       if u.left ≠ NIL
          then enqueue(Q,(t.left, ℓ+1))
       if u.right ≠ NIL
          then enqueue(Q,(t.right, ℓ+1))
```

Versione 2

Per accorgersi di quando si estrae il primo nodo di un nuovo livello dalla coda, modifichiamo la visita BFS vista a lezione andando ad inserire in coda un nodo vuoto alla fine di ogni livello (basta inserire un puntatore NIL).

Quando si estrae dalla coda un nodo vuoto significa che abbiamo finito di esaminare un livello e quindi:

- il nodo successivo che esce dalla coda deve essere stampato (ammesso che esista e il nodo nullo non fosse quello che indica la fine dell'ultimo livello dell'albero);
- va inserito un nodo nullo in coda, per segnalare la fine del livello successivo (ovvero quello che contiene i figli dei nodi al livello che abbiamo appena finito di analizzare).

```
VistaSinistra(T)
   if T = NIL
      then
        print "albero vuoto"
        return
   Q := new_queue()
   enqueue(Q,T)
   enqueue(Q,NIL) // il livello della radice contiene solo la radice
   while NOT empty_queue(Q) do
       u := dequeue(Q)
       if u = NIL
          then // u indica la fine di un livello e l'inizio del successivo
          if empty_queue(Q)
             then return // siamo arrivati alla fine della visita
             else // prendiamo il prossimo nodo in coda, che e' il primo del prossimo livello
               u := dequeue(Q)
               print u.val
               enqueue(Q, NIL) // si segnala la fine del prossimo livello
       // se siamo arrivati qua, u e' un puntatore ad un nodo non nullo
       if u.left ≠ NIL
          then enqueue(Q,t.left)
       if u.right ≠ NIL
          then enqueue(Q,t.right)
```

Osservazione: dopo l'analisi del nodo u si mettono in coda solo i figli non nulli, quindi quando viene estratto dalla coda un puntatore nullo, questo individua senza ambiguità la fine di un livello e l'inizio di quello successivo.

ERRORI PIÙ COMUNI:

- Mescolare la BFS con la DFS, usando sia la ricorsione che l'interazione.
- Usando la strategia DFS, scendere a sinistra fino a quando possibile e, solo quando questo non è più possibile, scendere a destra. Ci potrebbero essere, nei sottoalberi di destra "scartati" durante la discesa a sinistra, dei nodi a profondità maggiore che non verrebbero stampati (nella figura: se non scendessi mai nel sottoalbero di destra della radice, non arriverei mai a stampare il nodo 6).
- Utilizzando la strategia BFS, non implementare un meccanismo che permetta di stabilire quando inizia un nuovo livello, o implementarne uno sbagliato.
- Assumere che l'altezza dell'albero sia $\log n$, non è un albero completo e perfettamente bilanciato. L'altezza, nel caso peggiore, può essere uguale a $n - 1$, se l'albero è degenere ed è una catena di n nodi.

Quinto Appello
26 Gennaio 2022

SECONDA PARTE

ATTENZIONE: L'uso di **break** e **continue** NON è consentito

Esercizio 1.

Si consideri il seguente problema:
INPUT: un albero binario T con n nodi, implementato con nodi e puntatori (campi del nodo *key, left, right*, con significato standard), che memorizza numeri interi nel campo *key*.
OUTPUT: modificare l'albero T in modo che tutte le chiavi negative sono state spostate verso il fondo dell'albero, senza modificare la struttura dell'albero. Ovvero, un nodo con chiave negativa non può avere figli con chiavi positive o zero.

1. Si fornisca un esempio di input e output del problema con un albero di almeno 7 nodi.
2. Si descriva brevemente a parole e di fornisca lo pseudocodice di un algoritmo per il problema che usi la ricorsione.
3. Si studi il costo computazionale dell'algoritmo.

Esercizio 2.

Si consideri il seguente problema:
INPUT: un array A di dimensione n che memorizza valori interi.
OUTPUT: modificare A in modo che tutti i valori negativi si trovino a sinstra (ovvero ad indici minori) dei valori positivi e degli zero.

1. Descrivere brevemente a parole e fornire lo pseudocodice di un algoritmo che risolva il problema. L'algoritmo proposto **deve** avere tempo lineare $O(n)$ e usare spazio aggiuntivo costante $O(1)$.
2. Mostrare un esempio di esecuzione **dell'algoritmo proposto** in cui A inizialmente abbia sia valori negativi che positivi mescolati tra di loro e tale che $n \geq 8$, elencando gli stati intermedi dell'array dopo ogni scambio di elementi.

Domanda teorica (in alternativa ad uno dei due esercizi). (max 5 punti)
Enunciare il problema della *Longest Increasing Subsequence* e presentare l'algoritmo di programmazione dinamica visto a lezione (non è strettamente necessario lo pseudocodice, non è sufficiente lo pseudocodice senza spiegazioni). Spiegare in che modo in questo algoritmo le soluzioni a sottoproblemi più piccoli vengono utilizzate per costruire la soluzione di un sottoproblema più grande.

SOLUZIONI SECONDA PARTE

NON ho avuto tempo allora per scrivere delle soluzioni dettagliate in tempo utile per l'appello successivo e anche adesso se mi fermo a scriverle ritardo l'uscita delle soluzioni. Quando avrò tempo per aggiornare il file verrete informati.

Vi scrivo quali sono le idee utili a risolvere gli esercizi e quali gli errori più comuni e più gravi.

SOLUZIONE Esercizio 1.

Il problema si risolve con una visita DFS in POST ordine: prima si mettono "a posto" i sottoalberi dei figli, POI ci si occupa della chiave nel padre. Se quest'ultima è positiva non si deve fare niente, altrimenti la si deve far scendere nel sottoalbero.

Per trovare il posto giusto per la chiave del padre si scende nel suo sottoalbero (già modificato) scambiando ripetutamente la chiave con quella di uno dei figli, se positiva. Ci si ferma quando si arriva in una foglia, oppure se entrambi i figli sono negativi. Anche questa operazione si fa con una DFS.

Il costo è $O(n^2)$ perché chiamiamo una DFS per ogni nodo dell'albero (sul suo sottoalbero).

ERRORI PIÙ COMUNI:

- Utilizzare una visita DFS in PRE-ORDINE. Scambiare semplicemente la chiave del padre con il figlio non funziona. La chiave potrebbe dover scendere di più nell'albero. Provate a fare un esempio di un albero in cui questo approccio non funziona. Suggerimento: padre e figlio hanno chiavi entrambi negative, nel sottoalbero del figlio c'è una chiave positiva. Anche se sbagliato, l'approccio di utilizzare una DFS è corretto.
- Utilizzare un algoritmo non ricorsivo ma iterativo. Questo approccio è sbagliato perché la traccia chiedeva esplicitamente di utilizzare la ricorsione.
- Non mettere l'esempio.

SOLUZIONE Esercizio 2.

Per risolvere il problema basta modificare la procedura `Partition` di `Quicksort` facendo usare come pivot il valore zero (e separare i valori strettamente minori del pivot da quelli maggiore uguale) invece che il valore che si trova in una delle celle dell'array.

Si può usare una qualunque versione di `Partition` (es, quella vista a lezione, o quella che scorre l'array con due indici a partire dai due estremi).

ERRORI PIÙ COMUNI:

- Utilizzare una variante di `CountingSort` o un array di appoggio: la soluzione doveva avere costo aggiuntivo in spazio costante. Quanto spazio aggiuntivo richiede `CountingSort`? Questo errore è grave.
- L'esempio non è coerente con l'algoritmo descritto, o non c'è, o non presenta gli stati intermedi.

Sesto Appello
18 Febbraio 2022

SECONDA PARTE

ATTENZIONE: L'uso di **break** e **continue** NON è consentito

Esercizio 1. (max 11 punti) Si consideri il seguente problema:

INPUT: Un DAG $G = (V, E)$ e una sorgente $s \in V$.
OUTPUT: Il numero di pozzi disitinti che si possono raggiungere partendo da s.

Fare un esempio con un DAG di almeno 8 nodi, indicando chiaramente l'input e l'output del problema.

Descrivere brevemente a parole e fornire lo pseudocodice di un algoritmo per il problema.

Studiare il costo computazionale in tempo e spazio dell'algoritmo proposto.

Esercizio 2. (max 10 punti) Si consideri il seguente problema:

INPUT: Un array A di n numeri naturali distinti e ordinato (in ordine crescente).
OUTPUT: L'indice del più piccolo numero mancante nella sequanza di numeri ordinati, oppure -1 se il numero mancante non esiste.

Fare un esempio di input e output con $n \geq 8$.

Descrivere brevemente a parole e fornire lo pseudocodice di un algoritmo per il problema di costo computazionale in tempo $O(\log n)$.

Domanda teorica (in alternativa ad uno dei due esercizi) (max 5 punti)

Presentare la struttura dati DISJOINT-SET e l'implementazione della primitiva UNION con l'euristica del rango. Spiegare quale problema risolve l'euristica del rango rispetto all'implementazione che non utilizza questa euristica.

SOLUZIONI SECONDA PARTE

SOLUZIONE Esercizio 1.

Facendo una visita (BFS o DFS) nel DAG partendo dalla sorgente s si raggiungono tutti i nodi raggiungibili da s, tra cui i pozzi che si devono contare. Quindi, la modifica importante da fare alla visita è rendersi conto quando si arriva in un nodo del DAG che non ha archi uscenti e capire come contare questi nodi. Osserviamo che un pozzo sarà sicuramente una foglia dell'albero della visita (perché da quel nodo non è possibile continuare la visita verso altri nodi), ma che non tutte le foglie saranno necessariamente pozzi (semplicemente, tutti i nodi raggiungibili dal nodo foglia potrebbero essere già stati raggiunti dalla visita precedentemente).

Nel caso in cui si utilizzi la DFS è necessario modificare il codice (visto a lezione) della funzione principale in modo da non invocare la `DFS_visit` su tutti nodi del grafo non ancora visitati ma solo a partire da s. Inoltre, essendo una visita ricorsiva, è necessario fare attenzione alla gestione del conteggio (o si utilizza un contatore globale, che però non deve essere reinizializzato ad ogni chiamata ricorsiva (!), oppure si utilizza un approccio simile a quello adottato quando si contano, per esempio, le foglie di un albero: si scrive una funzione che restituisce il numero di pozzi raggiunti da un certo nodo e, per ogni nodo, si sommano i risultati delle chiamate dei nodi raggiungibili con archi uscenti).

In ogni caso deve essere modificato l'output.

Attenzione: nel caso un cui la sorgente s sia anche un pozzo, il risultato deve essere 1.

VERSIONE BFS: La funzione non è ricorsiva, non è necessario indicare la chiamata principale

```
ContaPozziBFS(G=(V,E),s)
   for all v ∈ V do
       visited[v] := F
   visited[s] := T
   Q := new_queue()
   Enqueue(Q,s)
   count := 0 // conta il numero di pozzi raggiungibili da s
   while NOT is_empty_queue(Q) do
         u := Dequeue(Q)
         pozzo := T // rimane vera se u e' un pozzo
         for all (u,v) ∈ E do // entra nel ciclo for solo se u ha archi uscenti
             pozzo := F // u non e' un pozzo
             if NOT visited[v]
                then visited[v] := T
                     Enqueue(Q,v)
         if pozzo then count := count + 1 // se u e' un pozzo incrementa il contatore
   return count
```

Funziona anche se la sorgente è un pozzo.

VERSIONE DFS: La funzione principale non è ricorsiva, non è necessario indicare la chiamata principale. La funzione `DFS_Visit` è ricorsiva, ma la chiamata principale si trova nella funzione principale.

Versione senza contatore globale:
la `DFS_Visit` è una funzione che viene invocata su un nodo `v` ed esplora la porzione di DAG raggiungibile da `v` che non sia già stata visitata in precedenza. Restituisce il numero di pozzi raggiunti durante questa esplorazione. Utilizza un contatore locale che alla fine sarà uguale alla somma del numero di pozzi raggiunti da ogni vicino di `v` raggiungibile seguendo un arco uscente da `v`.

```
ContaPozziDFS(G=(V,E),s)
   for all v ∈ V do
       visited[v] := F
   return DFS_Visit(G,s) // visita invocata solo a partire da s

DFS_Visit(G,v)
   visited[v] := T
   count := 0 // conta i pozzi raggiunti da v - non e' un contatore globale
   pozzo := T // rimane vera se v e' un pozzo
   for all (v,w) ∈ E do // entra nel ciclo for solo se v non e' un pozzo
       pozzo := F // v non e' un pozzo
       if NOT visited[w] then count := count + DFS_Visit(G,w)
              // somma il numero di pozzi raggiunti da w
   if pozzo
      then return 1
      else return count
```

Funziona anche se la sorgente è un pozzo.

Esercizio di stile: riscrivere `DFS_visit` in modo da non dover fare il controllo finale sulla variabile `pozzo` (devono essere inizializzati e aggiornati diversamente count e pozzo).

```
DFS_Visit(G,v)
   visited[v] := T
   count := 1 // conta i pozzi raggiunti da v - non e' un contatore globale
   pozzo := 0 // rimane 0 se v e' un pozzo
   for all (v,w) ∈ E do // entra nel ciclo for solo se v non e' un pozzo
       pozzo := 1 // v non e' un pozzo
       if NOT visited[w] then count := count + DFS_Visit(G,w)
              // somma il numero di pozzi raggiunti da w
   return (count - pozzo) // e' uguale a 1 se v e' un pozzo, a (count -1) altrimenti
```

Versione con contatore globale:
la `DFS_Visit` è una procedura che viene invocata su un nodo `v` ed esplora la porzione di DAG raggiungibile da `v` che non sia già stata visitata in precedenza. Quando viene invocata su un pozzo incrementa un contatore globale, altrimenti viene invocata ricorsivamente sui vicini raggiungibili seguendo un arco uscente da `v`.

```
ContaPozziDFS(G=(V,E),s)
   for all v ∈ V do
       visited[v] := F
   count := 0 // /e' un contatore globale
   DFS_Visit(G,s) // visita invocata solo a partire da s
   return count

DFS_Visit(G,v)
   visited[v] := T
   pozzo := T // rimane vera se v e' un pozzo
   for all (v,w) ∈ E do // entra nel ciclo for solo se v non e' un pozzo
       pozzo := F // v non e' un pozzo
       if NOT visited[w] then DFS_Visit(G,w) // cerca pozzi a partire da w
   if pozzo then count := count +1
```

ERRORI PIÙ COMUNI:

- Contare tutti i pozzi del DAG: alcuni di quei pozzi possono non essere raggiungibili dalla sorgente data in input e non devono essere contati. Questo succede se si fa ri-partire la DFS fino a quando non ci sono più nodi non visitati.
- Fare una visita (BFS o DFS) partendo dalla sorgente data in input e contare le foglie dell'albero. Le foglie non sono necessariamente pozzi. Se non hanno figli nell'albero della visita non vuol dire che non abbiano archi uscenti nel DAG, ma solo che tutti i nodi raggiungibili da quel nodo sono già stati visitati precedentemente.
- Fare una visita (BFS o DFS) partendo dalla sorgente data in input e contare tutti i nodi che, alla fine della visita, non sono `prev` di nessun altro nodo. Sicuramente si contano i pozzi, ma si contano anche molti altri nodi: le foglie dell'albero di copertura risultante dalla visita (non sono necessariamente pozzi) e tutti i nodi che non sono raggiungibili dalla sorgente data in input.
- Sbagliare a scrivere il codice della visita del grafo.
- Utilizzare uno degli algoritmi per la costruzione dello spanning tree e contare le foglie dell'albero. Si ripresenta, infatti, il problema delle foglie già evidenziato nella soluzione che utilizzava una visita del grafo. Inoltre, gli algoritmi di PRIM e KRUSKAL sono progettati per grafi non diretti. Se utilizzati in grafo diretto non garantiscono di trovare il minimo albero di copertura (che in questo caso non sarebbe un problema). Nel caso dell'algoritmo di KRUSKAL però non si ha neanche la garanzia di ottenere un albero (gli archi vengono scelti in base al peso e ignorando la direzione) e si includono tutti i nodi del DAG senza escludere quelli non raggiungibili da s.

SOLUZIONE Esercizio 2.

Scriviamo una funzione principale che controlla se esiste un elemento mancante nell'array: nel caso un cui non ci sia allora restituisce subito -1, altrimenti lo va a cercare (essendo sicuro di trovarlo).

Se gli n elementi nell'array sono n interi positivi consevutivi, allora necessariamente avremo

$$A[0] + n - 1 = A[n-1],$$

perché, per ogni $i = 0, \ldots, n-1$, abbiamo che $A[i] = A[0] + i$, indipendentemente da quale sia il valore $A[0]$. Esempio: dato $A =< 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 >$ $(n = 11)$ abbiamo $A[0] + 10 = 3 + 10 = 13 = A[10]$ e, per esempio, $A[5] = 8 = 3 + 5 = A[0] + 5$.

Altrimenti (se l'elemento mancante c'è), per cercare un elemento con una determinata proprietà in tempo logaritmico in un array ordinato, si può procedere utilizzando un approccio simile a quello della ricerca binaria, dividendo sempre a metà lo spazio di ricerca.

Ad ogni chiamata ricorsiva lavoriamo su una porzione di array $A[i..j]$ che va da un indice i ad un indice j con $i \leq j$.

Caso Base: se $i = j$ abbiamo trovato l'elemento che stiamo cercando (e quindi la posizione di quello che manca) e restituiamo i;

Caso Ricorsivo: calcoliamo l'indice centrale k nello stesso modo della ricerca binaria e, per decidere se il primo elemento mancante si trova nella metà di destra o di sinistra, testiamo l'elemento in posizione k:

- Se $A[0] + k = A[k]$, allora tutti i valori fino all'indice k compreso sono consecutivi e l'elemento mancante deve essere cercato a destra, ovvero in $A[k + 1..j]$;
- altrimenti, il primo elemento mancante si trova a sinitra e deve essere cercato in $A[i..k]$ (potrebbe essere anche in k)

```
FirstMissing(A)
   if A[0] + n − 1 = A[n − 1]
      then return −1
      else return FindMissing(A,0,n-1)

FindMissing(A,i,j)
   if i = j then return i
   k = ⌊(i+j)/2⌋
   if A[0] + k = A[k]
      then return FindMissing(A,k+1,j)
      else return FindMissing(A,i,k)
```

Anche in questo caso non è necessario indicare la chiamata principale per `FindMissing` perché questa si trova già nella funzione principale.

Varianti:

- Il test su $A[k]$ può essere fatto anche utilizzando $A[i]$ invece che $A[0]$. Provate a scrivere la formula corretta.
- Si può aggiungere un controllo per stabilire se k sia l'indice cercato (senza dover aspettare di arrivare ad avere un solo elemento): deve valere che $A[0] + k < A[k]$ e che tutti i valori fino a $k - 1$ devono essere consecutivi (quindi deve valere anche $A[0] + (k - 1) = A[k - 1]$). In questo caso bisogna sempre fare attenzione ed essere sicuri che esita l'elemento all'indice $k - 1$ (ovvero che $k - 1 \geq 0$).
- Si può non fare il controllo preventivo su A. In questo caso si devono considerare due casi base, uno che corrisponde al caso in cui ci sia l'elemento mancante e uno al caso in cui non ci sia.

ERRORI PIÙ COMUNI:

- Cercare l'elemento mancante con una scansione linerare (o peggio quadratica) dell'array. Grave dichiarare che la soluzione è logaritmica.

- Non testare l'esistenza dell'elemento mancante prima di cercarlo e avere come caso base unicamente quello in cui, quando non ci sono più elementi dell'array da analizzare (tipicamente quando $i > j$), si restituisce -1. Se non si prevedere un caso base in cui si resitituisce, sotto certe condizioni, l'indice di un elemento dell'array, non sarà mai possibile resitituire qualcosa di diverso da -1.

- Utilizzare un approccio "simile" alla ricerca binaria, ma invocare la ricorsione su entrambe le metà dell'array e non solo su una delle due metà (es, resitiure il minimo tra i risultati dell'invocazione ricorsiva sulla metà di destra e sulla metà di sinistra). In questo caso, il costo non è più logaritmico ma lineare.

- Assumere che il primo elemento della sequenza memorizzata nell'array sia zero (o uno). Il primo numero può essere un intero qualsiasi.

- Decidere quale metà analizzare confrontando l'elemento centrale solo con i valori agli indici vicini, ovvero confrontare $A[k]$ con $A[k-1]$ e/o $A[k+1]$. Infatti, quando elemento centrale è "a posto" rispetto ai suoi vicini, l'elemento mancante potrebbe essere sia alla sua destra che alla sua sinistra, e non ci sono abbastanza informazioni per decidere su quale metà continuare con la ricorsione.
  Per esempio, dato $A =< 4, 5, 6, 7, 8, 10, 11 >$: il primo k è 3, abbiamo che $A[3]$ è a posto (infatti $A[2] = A[3] + 1 = A[4] + 2$) e che l'elemento mancante è a destra. Prendiamo adesso $A =< 3, 4, 6, 7, 8, 10, 11 >$. Il risultato del test di prima su $A[k] = A[3]$ è esattamente lo stesso di prima e ci dice, come prima, che l'elemento centrale è al suo posto, ma questa volta il primo elemento mancante si trova a sinistra di quello centrale. Quindi il test fatto sull'elemento centrale NON aiuta a decidere su quale metà invocare la ricorsione.