Algoritmi e Strutture Dati – 22/08/2019

Esercizio A1

È facile notare che l'algoritmo Merge() proposto non è altro che una versione di Insertion Sort dove vengono specificati gli indici di inizio e di fine.

Il caso ottimo corrisponde, come nell'InsertionSort, al caso in cui il vettore sia già ordinato. In questo caso, tutte le volte che viene applicata Merge(), il costo è lineare e quindi l'equazione di ricorrenza è:

$$T(n) = \begin{cases} 2T(n/2) + n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Utilizzando il Master Theorem, il costo computazionale risultante è pari a $\Theta(n \log n)$.

Il caso pessimo corrisponde, come nell'InsertionSort, al caso in cui il vettore sia ordinato in senso opposto. In questo caso, tutte le volte che viene applicata Merge(), il costo è quadratico e quindi l'equazione di ricorrenza è:

$$T(n) = \begin{cases} 2T(n/2) + n^2 & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Utilizzando il Master Theorem, il costo computazionale risultante è pari a $\Theta(n^2)$.

Esercizio A2

Le regole di colorazione degli Alberi RedBlack sono le seguenti:

1. La radice è nera
2. Tutte le foglie sono nere
3. Entrambi i figli di un nodo rosso sono neri
4. Tutti i cammini semplici da un nodo u ad una delle foglie contenute nel sottoalbero radicato in u hanno lo stesso numero di nodi neri

Il codice seguente è lo scheletro della soluzione. Innanzitutto, viene verificata la proprietà (1), che è facile da controllare. Poi, si invoca una funzione ricorsiva blackHeight() che analizza le restanti proprietà degli alberi RB e ritorna l'altezza nera dell'albero, in caso le proprietà siano rispettate, oppure -1, se l'albero non rispetta le proprietà.

```
boolean isRedBlack(TREE T)
  if T.color == RED then
    return false
  else
    return (blackHeight(T) > 0)
```

La funziona ricorsiva che verifica le proprietà (2)-(4) sull'intero albero è la seguente.

- La proprietà (2) è verificata sui nodi **nil** e assumendo che essi siano nodi speciali, in cui sia comunque determinare il colore. Se questo è rosso, si ritorna -1, ad indicare che in un qualche punto dell'albero una delle proprietà non è rispettata.
- La proprietà (3) viene verificata su tutti i nodi rossi, controllando se il proprio genitore (se esiste) ha anch'esso colore rosso.
- La proprietà (4) viene vericata calcolando l'altezza nera dei rami sinistro e destro del nodo attuale. Se una di esse è negativa, le proprietà (2)-(4) non sono verificate nel sottoalbero relativo. Se sono diverse, la proprietà (4) non è rispettata e si ritorna -1. Altrimenti, si restuisce l'altezza nera del nodo attuale.

```
int blackHeight(TREE T)
    % Proprietà (2)
    if T == nil then
        return iif(T.color == RED, −1, 1)
    % Proprietà (3)
    if t.color == RED and t.parent ≠ nil and t.parent.color == RED then
        return −1
    % Proprietà (4)
    int bh_L = blackHeight(T.left)
    int bh_R = blackHeight(T.right)
    if bh_L < 0 or bh_R < 0 or bh_L ≠ bh_R then
        return −1
    else
        return bh_L + iif(t.color == BLACK, 1, 0)
```

La complessità è $\Theta(n)$, in quanto blackHeight() implementa una visita in profondità.

Esercizio A3

Se ordiniamo il grafo tramite un ordinamento topologico, otteniamo uno dei tanti ordinamenti possibili. Può capitare che un esame con una propedeuticità appaia prima di un esame senza propedeuticità. Tuttavia sappiamo che quando un esame viene estratto dallo stack, tutti gli esami che sono propedeutici ad esso sono stati già estratti.
Non siamo interessati alla distanza minima; invece, siamo interessati alla distanza massima a partire da tutti i nodi che non hanno archi entranti.
La distanza viene inizializzata a zero per tutti i nodi. Tutte le volte che viene estratto un nodo, la sua distanza massima effettiva è già stata calcolata (in quanto quando viene estratto, tutti gli esami propedeutici sono stati estratti). A questo punto, possiamo aggiornare la distanza di tutti i nodi adiacenti, incrementandola se necessario.
Al termine, ritorniamo la distanza massima trovata incrementata di uno, perché stiamo calcolando il numero di sessioni.

```
minSessions(GRAPH G)
    STACK S = topsort(G)
    int[] dist = new int[1 ... G.n]
    foreach u ∈ G.V() do
        dist[u] = 0
    while not S.isEmpty() do
        NODE u = S.pop()
        foreach v ∈ G.adj(u) do
            dist[v] = max(dist[v], dist[u] + 1)
    return max(dist) + 1
```

La complessità è quella di due visite in profondità, ovvero $O(m + n)$.

Esercizio B1

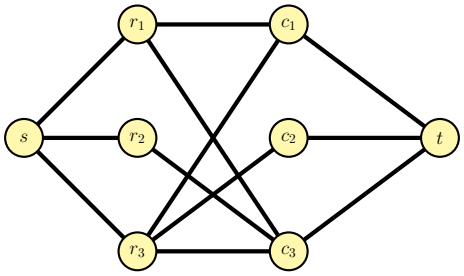
L'esercizio può essere risolto tramite una rete di flusso, con $|V| = 2n + 2$ nodi così suddivisi:

- Si aggiungano un nodo sorgente s e un nodo pozzo t;
- Si aggiungano n nodi riga $r_1, \ldots, r_n$;
- Si aggiungano n nodi colonna $c_1, \ldots, c_n$;

Gli archi sono così organizzati:

- Si collega il nodo sorgente ad ogni nodo riga r_i, con peso 1, ad indicare che può esserci al più una torre per riga;
- Si collega ogni nodo colonna c_j al nodo pozzo t, con peso 1, ad indicare che può esserci al più una torre per colonna;
- Si collega ogni nodo riga r_i ad ogni nodo colonna c_j tale per cui $M[i][j] =$ **true**, con peso 1;

Il numero totale di archi è quindi $|E| = 2n + O(n^2)$.
Il flusso massimo corrisponde al numero massimo di torri che possono essere piazzate. Infatti, se una torre viene piazzata su una casella, questo "consuma" sia la corrispondente riga che la corrispondente colonna.
Poichè possono essere piazzate al più n torri, il flusso massimo è limitato da n. Il costo computazionale è quindi $O((|V| + |E|) \cdot n)$, ovvero $O(n^3)$.
Nel disegno seguente, mostriamo la rete risultante dall'esempio del compito, omettendo il peso 1 su tutti gli archi.



Esercizio B2

Il problema può essere risolto tramite un algoritmo greedy. In forma "classica", si trova nel libro (3a edizione, es. 14.6, con soluzione testuale ma senza pseudocodice).
Si ordinano (separatamente) i due vettori *start* e *end*.
Si inizializzano a zero due variabili *bought* e *available*, che rappresentano rispettivamente il numero totale di skipass comprati e il numero di skipass attualmente disponibili, perchè non più utilizzati dai precedenti utilizzatori.
Si scorrono quindi i due vettori, in modalità simile alla funzione Merge(), ovvero confrontando il prossimo istante del vettore *start* e il prossimo istante del vettore *end*, cercando il più piccolo.

- Se il prossimo istante appartiene al vettore *start*, vuole dire che una persona vuole iniziare a sciare. Se esiste uno skipass già comprato e disponibile ($available > 0$), lo si prende diminuendo il numero di skipass disponibili; altrimenti se ne compra uno nuovo.
- Se il prossimo istante appartiene al vettore *end*, vuol dire che una persona ha smesso di sciare. Il suo skipass diventa disponibile
- Se $start[s] == end[e]$, vuole dire che una persona vuole smettere e una vuole iniziare; diamo precedenza alla prima, in modo che il suo skipass diventi disponibile

L'algoritmo termina quando sono stati analizzati tutti i tempi di inizio, il che avverrà ovviamente prima che vengano analizzati tutti i tempi di fine. A quel punto non sarà più necessario comprare nuovi skipass e si potrà restituire il valore *bought*.
È possibile provare (informalmente) la correttezza di questo algoritmo utilizzando il principio di scelta greedy. Si consideri l'intervallo con minor tempo di inizio. È ovviamente necessario comprare uno skipass per questo intervallo; per assurdo, assumiamo che esista una soluzione che richieda meno skipass ma non includa uno skipass per tale intervallo. Questa soluzione non può esistere, in quanto il primo sciatore si troverà senza skipass.
Dobbiamo quindi risolvere il sottoproblema contenente gli $n-1$ intervalli rimanenenti, tenendo conto che tutte le volte che si libera uno skipass, dobbiamo utilizzarlo per ridurre il numero di skipass comprati.

```
minSkipass(int[] start, int[] end, int n)
    sort(start)
    sort(end)
    int available = 0
    int bought = 0
    int s = 1
    int e = 1
    while s ≤ n do
        if start[s] < end[e] then
            if available > 0 then
                available = available − 1
            else
                bought = bought + 1
            s = s + 1
        else
            available = available + 1
            e = e + 1
    return bought
```

Il costo computazionale dell'algoritmo è dominato dall'ordinamento, ed è pari a $O(n \log n)$.

Esercizio B3

Questo problema può essere risolto in maniera efficiente utilizzando la programmazione dinamica. Sia $DP[h]$ il numero di alberi 1-bilanciati di altezza h.

- Se $h = 0$, stiamo parlando di un nodo singolo, con altezza 0; esiste un solo albero di questo tipo.
- Se $h = 1$, esistono tre casi possibili: una radice con solo figlio destro, una radice con solo figlio sinistro, una radice con entrambi i figli.
- Se $h > 1$, possono darsi tre casi:
  - Il figlio destro è radice di un sottoalbero di altezza $h - 1$, mentre il figlio sinistro è radice di un sottoalbero di altezza $h - 2$;
  - Il figlio sinistro è radice di un sottoalbero di altezza $h - 1$, mentre il figlio destro è radice di un sottoalbero di altezza $h - 2$;
  - Sia il figlio destro che sinistro sono radice di un sottoalbero di altezza $h - 1$.

  In tutti questi casi, i sottoalberi destro e sinistro devono essere 1-bilanciati essi stessi.

È quindi possibile esprimere la formula ricorsiva in questo modo:

$$DP[h] = \begin{cases} 1 & h = 0 \\ 3 & h = 1 \\ 2 \cdot DP[h-1] \cdot DP[h-2] + DP[h-1] \cdot DP[h-1] & h > 1 \end{cases}$$

Ovviamente, dobbiamo utilizzare programmazione dinamica per evitare di ricalcolare i casi già risolti. È possibile quindi tradurre il codice nel modo seguente:

```
int balanced(int h)
    int DP = new int[0...h]
    DP[0] = 1
    DP[1] = 3
    for i = 2 to h do
        DP[h] = 2 · DP[i − 1] · DP[i − 2] + DP[i − 1] · DP[i − 1]
    return DP[h]
```

La complessità è lineare nel numero di operazioni di lettura/scrittura del vettore DP; tenete conto tuttavia che il numero di alberi cresce più che esponenzialmente, quindi le operazioni di moltiplicazione e somma non hanno costo lineare in h.