Algoritmi e Strutture Dati - 21/01/2019

Esercizio A1

Andando per tentativi, proviamo con $\Theta(n\sqrt{n})$. È facile vedere che la ricorrenza è $\Omega(n\sqrt{n})$, per via della sua componente non ricorsiva. Proviamo quindi a dimostrare che $T(n) = O(n\sqrt{n})$.

- Caso base: $T(n) = 1 \leq cn\sqrt{n}$, per tutti i valori di n compresi fra 1 e 9, ovvero:

$$c \geq \frac{1}{n\sqrt{n}}, \forall n : 1 \leq n \leq 9$$

  I valori $\frac{1}{n\sqrt{n}}$ sono minori o uguali di 1, per $1 \leq n \leq 9$; quindi tutte queste disequazioni sono soddisfatte da $c \geq 1$.

- Ipotesi induttiva: $T(k) \leq ck\sqrt{k}$, per $k < n$

- Passo induttivo:

$$\begin{aligned}
T(n) &= 4T(\lfloor n/4 \rfloor) + 9T(\lfloor n/9 \rfloor) + n\sqrt{n} \\
&\leq 4c\lfloor n/4 \rfloor^{1.5} + 9c\lfloor n/9 \rfloor^{1.5} + n\sqrt{n} \\
&\leq \frac{4}{8}cn\sqrt{n} + \frac{9}{27}cn\sqrt{n} + n\sqrt{n} \\
&= \frac{1}{2}cn\sqrt{n} + \frac{1}{3}cn\sqrt{n} + n\sqrt{n} \\
&\leq 5/6cn\sqrt{n} + n\sqrt{n} \leq cn\sqrt{n}
\end{aligned}$$

  L'ultima disequazione è vera per $c \geq 6$.

Abbiamo quindi dimostrato che $T(n) = \Theta(n\sqrt{n})$, con $m = 1$ e $c \geq 6$.

Esercizio A2

La soluzione si basa (ovviamente) sulla ricerca binaria, ma bisogna prestare particolare attenzione ai casi particolari e agli indici. In particolare, se il valore v è più alto di tutti i valori presenti nel vettore, il valore cercato non esiste e bisogna restituire -1.

- **Caso base**: Si consideri un vettore costituito da un elemento solo.
  - Se tale vettore contiene un valore più piccolo o uguale a v, significa che non esiste un elemento più grande di v nel vettore e dobbiamo restituire -1.
  - Se tale vettore contiene un valore più grande di v, è anche il più piccolo con questa proprietà presente nel vettore e quindi è il valore da restituire.
- **Passo ricorsivo**: Si considera l'elemento mediano in posizione $m = \lfloor (i + j)/2 \rfloor$.
  - Se v è maggiore o uguale a $A[m]$, allora il valore cercato, se esiste, si trova in $A[m + 1 \ldots j]$.
  - Altrimenti, se v è più minore $A[m]$il valore esiste sicuramente, e si trova in $A[i \ldots m]$ (essendo potenzialmente $A[m]$ stesso).

Ricalcando la ricerca binaria, l'algoritmo ha una complessità pari a $O(\log n)$, come richiesto.

```
int ceilRec(int [ ] A, int i, int j, int v)
    if i == j then
        if v ≥ A[i] then
            return −1
        else
            return A[i]
    else
        int m = ⌊(i + j)/2⌋
        if v ≥ A[m] then
            return ceilRec(A, m + 1, j, v)
        else
            return ceilRec(A, i, m, v)
```

```
int ceil(int [ ] A, int n, int v)
    return ceilRec(A, 1, n, v)
```

Esercizio A3

L'esercizio si risolve tramite una visita in ampiezza dell'albero binario, e ed è molto simile al problema di calcolare la larghezza dell'albero radicato (Problema 1.1 del blocco di esercizi sugli Alberi e Problema 5.4 del libro). La soluzione fornita per quell'esercizio deve essere modificata per due aspetti:

- L'albero che prendiamo in input è binario, non generale, quindi va modificata la parte che inserisce nodi in coda.
- Invece di calcolare la larghezza, calcoliamo la somma delle brillantezze, che poi va confrontata con la brillantezza della radice.

Il costo dell'algoritmo è ovviamente $\Theta(n)$.

```
boolean isChristmassy(TREE t)
    int count = 1                              % # nodi nel livello corrente da visitare; radice
    int brillianceSoFar = 0                    % Brillantezza del livello attuale
    boolean christmassy = true                 % Vero se l'albero rispetta la definizione
    QUEUE Q = Queue()
    Q.enqueue(t)
    while not Q.isEmpty() and christmassy do
        TREE u = Q.dequeue()
        if u.left() ≠ nil then
            Q.enqueue(u.left())
        if u.right() ≠ nil then
            Q.enqueue(u.right())
        brillianceSoFar = brillianceSoFar + u.brilliance
        count = count − 1
        if count == 0 then                     % Finito livello
            count = Q.size()
            if brillianceSoFar ≠ t.brilliance then
                christmassy = false
            brillianceSoFar = 0
    return christmassy
```

Esercizio B1

Potrebbe sembrare uno dei classici esercizi basati su reti di flusso. In effetti, è possibile risolvere l'esercizio in questo modo, ma non è il metodo più efficiente. Il grafo risultante avrebbe $|V| = 2 + n + 6 = n + 8$ nodi e $|E| = n + 6 + 2n = 3n + 6$ archi. Una visita di tale grafo avrebbe costo $O(|V| + |E|)$, ovvero un costo pari a $O(n)$. Il flusso totale $|f^*|$ ha valore n. Quindi, secondo il limite di Ford e Fulkerson, il costo totale dell'algoritmo è $O(n^2)$.
È possibile tuttavia risolvere il problema con costo $O(n)$, utilizzando un approccio greedy.
Innanzitutto, si contano le richieste per ogni taglia, utilizzando un vettore di appoggio $\mathit{richieste}[\,]$. Questo passo potrebbe essere interpretato come un ordinamento dei partecipanti utilizzando Counting Sort; è più efficiente che ordinare i partecipanti tramite un algoritmo basato su confronti, che avrebbe costo $O(n \log n)$.
Per ogni taglia i, partendo dalla 1, si calcola quante magliette rimangono, sottraendo $\mathit{richieste}[i]$ da $T[i]$. Se tale valore è negativo, le magliette non sono sufficienti per la particolare taglia, e bisogna utilizzare magliette della taglia successiva, sottraendo le magliette che mancano da $T[i + 1]$. Se $T[i + 1]$ diventa negativo, non ci sono magliette a sufficienza per soddisfare le persone con taglia i-esima, e si ritorna **false**. Altrimenti, si passa alla taglia successiva. Se tutte le taglie sono soddisfatte, si ritorna **true**.
Come sottoprodotto di questo approccio, il vettore T contiene il numero di magliette rimaste, per ogni taglia.
La correttezza dell'algoritmo può essere provata ragionando su ogni taglia. Le magliette di taglia 1 servono a persone di taglia 1, e quindi tanto vale assegnarle tutte a loro. Se non bastano, l'unico modo è utilizzare magliette di taglia 2. Se non bastano ancora, non c'è nulla da fare. Una volta risolte le persone di taglia 1, si fa lo stesso ragionamento sulla taglia 2, e così via.
L'algoritmo ha costo lineare $\Theta(n)$.

```
boolean isDoable(int[] T, int[] P, int n)
    int[] richieste = new int[1 ... 5]
    for i = 1 to 5 do
        richieste[i] = 0
    for j = 1 to n do
        richieste[P[j]] = richieste[P[j]] + 1
    int i = 1
    boolean doable = true
    while doable and i ≤ 5 do
        T[i] = T[i] − richieste[i]
        if T[i] < 0 then
            T[i + 1] = T[i + 1] + T[i]        % Poichè T[i] è negativo, questo corrisponde ad una sottrazione
            if T[i + 1] < 0 then
                doable = false
    return doable
```

Soluzione alternativa È possibile ordinare il vettore P, e poi assegnare una maglietta alla volta, sottraendo la maglietta $P[j]$ o $P[j] + 1$ a seconda della disponibilità. Il costo di questo algoritmo è $O(n \log n)$.

Esercizio B2

L'esercizio è identico all'esercizio 4 del 5/6/2014. La soluzione corrisponde alla funzione partition(H, n, k) descritta in quel compito. Sarebbe stato completamente accettabile citare la soluzione e passare al prossimo esercizio.

Esercizio B3

Richiedendo di elencare tutte le possibili somme, è necessario utilizzare la tecnica di backtrack. Scriviamo una funzione wrapper che chiama la funzione allSumsRec(**int**[ ] A, **int**[ ] S, **int** target, **int** i), dove S è uno stack contenente i valori che formano la somma, target è il valore della somma da ottenere, i è l'indice dell'elemento di A che stiamo considerando.
Ad ogni chiamata, ci sono due possibilità: inseriamo $A[i]$ nello stack, lo sottriamo da target senza modificare l'indice i perché il valore può essere riutilizzato; oppure non inseriamo $A[i]$ nello stack e passiamo al prossimo indice $(i - 1)$. Se si raggiunge un target negativo ($\mathit{target} < 0$) oppure si finiscono i valori ($i = 0$), si compie un'operazione di backtrack. Se target raggiunge il valore 0, si stampa il contenuto dello stack.

Il costo è pari a $O(v \cdot 2^{v+n})$, in quanto è necessario $O(v)$ per stampare ed ogni passo si provano due possibilità: o si diminuisce il valore o si dimuisce il numero di elementi. Ovviamente gran parte dell'albero viene potato, quindi questo è un limite superiore.

```
allSumsRec(int[] A, int[] S, int target, int i)
    if target == 0 then
        print(S)
    else if target > 0 and i > 0 then
        allSumsRec(A, S, target, i − 1)
        S.push(A[i])
        allSumsRec(A, S, target − A[i], i)
        S.pop()
```

```
allSums(int[] A, int n, int v)
    STACK S = Stack()
    int[] S = new int[1 . . . v]
    allSumsRec(A, S, v, n)
```