Algoritmi e Strutture Dati – Soluzione 07/02/2022

Esercizio A1 – Ricorrenza – Punti ≥ 9

Andando per tentativi, proviamo con $\Theta(n\sqrt[3]{n})$. È facile vedere che la ricorrenza è $\Omega(n\sqrt[3]{n})$, per via della sua componente non ricorsiva. Proviamo quindi a dimostrare che $T(n) = O(n\sqrt[3]{n})$.

- Caso base: $T(n) = 1 \leq cn\sqrt[3]{n}$, per tutti i valori di n compresi fra 1 e 27, ovvero:

$$c \geq \frac{1}{n\sqrt[3]{n}}, \forall n : 1 \leq n \leq 27$$

  I valori $\frac{1}{n\sqrt[3]{n}}$ sono minori o uguali di 1, per $1 \leq n \leq 27$; quindi tutte queste disequazioni sono soddisfatte da $c \geq 1$.

- Ipotesi induttiva: $T(k) \leq ck\sqrt[3]{k}$, per $k < n$

- Passo induttivo:

$$\begin{aligned} T(n) &= 12T(\lfloor n/8 \rfloor) + 18T(\lfloor n/27 \rfloor) + n\sqrt[3]{n} \\ &\leq 12c\lfloor n/8 \rfloor\sqrt{\lfloor n/8 \rfloor} + 18c\lfloor n/27 \rfloor\sqrt{\lfloor n/27 \rfloor} + n\sqrt[3]{n} \\ &\leq \frac{12}{16}cn\sqrt[3]{n} + \frac{18}{81}cn\sqrt[3]{n} + n\sqrt[3]{n} \\ &= \frac{3}{4}cn\sqrt[3]{n} + \frac{2}{9}cn\sqrt[3]{n} + n\sqrt[3]{n} \\ &\leq 35/36cn\sqrt[3]{n} + n\sqrt[3]{n} \leq cn\sqrt[3]{n} \end{aligned}$$

  L'ultima disequazione è vera per $c \geq 36$.

Abbiamo quindi dimostrato che $T(n) = \Theta(n\sqrt[3]{n})$, con $m = 1$ e $c \geq 36$.

Esercizio A2 – Gazprom – Punti ≥ 9

Il problema non richiede altro che individuare le componenti connesse e sommare le loro capacità, tramite i seguenti passi:

- si richiama l'algoritmo per il calcolo delle componenti connesse su grafo non orientato, che associa un identificatore di componente a ognuno dei nodi;
- si alloca un vettore *count* di dimensione n (numero massimo di componenti possibili e si inizializza a zero;
- si sommano quindi le capacità di ognuna delle componenti scorrendo tutti i nodi e sommando la capacità del nodo alla componente a esso associata;
- infine, si restituisce il massimo del vettore *count*, cioè la componente di capacità massima.

La complessità è $\Theta(m + n)$.

```
int gazprom(GRAPH G, int[] C)
    int[] id = cc(G)
    int k = max(G.size)
    int[] count = new int[1...k] = {0}
    foreach u ∈ G.V() do
        count[id[u]] = count[id[u]] + C[u]
    return max(count)
```

Esercizio A3 – Alberi pasciuti – Punti ≥ 12

Modifichiamo l'algoritmo che abbiamo proposto durante le esercitazioni per calcolare la "larghezza" di un albero generico, tenendo conto del fatto che stiamo trattando alberi binari e non alberi generali.
Tutte le volte che un livello viene completato, nella coda ci sono tutti e soli i nodi del livello successivo. Quindi, è possibile valutare se la dimensione del livello rispetta la regola degli alberi pasciuti. Se ciò non accade, si può immediatamente restituire **false**. Se l'albero intero viene visitato senza che questo accada, si restituisce **true** in quanto l'albero è pasciuto.
La complessità dell'algoritmo proposto è $O(n)$.
Si noti che assumiamo che l'albero sia non vuoto; tale albero ha almeno un nodo, cioè la radice a livello 0. Il livello 0 deve avere almeno $2^{-1} = 1/2$ nodi, condizione sempre vera se l'albero è non vuoto. Quindi il controllo non viene effettuato sul livello della radice.

```
boolean isPlump(Tree T)
    int count = 1                                  % # nodi nel livello corrente da visitare; radice
    int level = 0                                  % Livello corrente
    Queue Q = Queue()
    Q.enqueue(T)
    while not Q.isEmpty() do
        Tree u = Q.dequeue()
        if u.left ≠ nil then
            Q.enqueue(u.left)
        if u.right ≠ nil then
            Q.enqueue(u.right)
        count = count − 1
        if count == 0 then                         % Nuovo livello
            count = Q.size()
            level = level + 1
            if count > 0 and count < 2^(level−1) then
                return false
    return true
```

Soluzioni alternative Alcuni hanno usato l'approccio seguente:

- calcola l'altezza h dell'albero, utilizzando la funzione vista a lezione;
- dichiara un vettore *count* di interi con indici $0 \dots h-1$, inizializzato a zero;
- effettua una visita in profondità, passando il livello ℓ del nodo nella chiamata e incrementando il numero di nodi al livello ℓ: $count[\ell] = count[\ell] + 1$;
- al termine della visita, verifica che il requisito degli alberi pasciuti sia rispettato

L'algoritmo è corretto e la complessità è $\Theta(n)$, quindi prende 100%. Tuttavia, si noti che l'algoritmo proposto è $O(n)$, perché può terminare prima di completare l'intera visita.
In alcuni casi, si è usato lo stesso approccio ma con una tabella hash al posto del vettore (quindi, senza calcolare l'altezza dell'albero a priori). Anche questa soluzione prende 100%.

Sbagliando si impara Ho visto un certo numero di soluzioni scritte nel modo seguente:

```
boolean isPlump(Tree T)
    return isPlump(T, 0))
```

```
boolean isPlumpRec(Tree T, int ℓ)
    int size = iif(T.left ≠ nil, 1, 0) + iif(T.right ≠ nil, 1, 0)
    if size < 2^(ℓ−1) then
        return false
    return isPlumpRec(T.left, ℓ + 1) and isPlumpRec(T.right, ℓ + 1)
```

Questo algoritmo è ovviamente sbagliato, perché in realtà non fa altro che verificare se il numero di figli di un nodo rispetta la regola (e la risposta sarà probabilmente no, in quando dal livello 3 in poi il numero di figli deve essere almeno 4, impossibile in un albero binario).

Esercizio B1 – Bit consecutivi – Punti ≥ 8

Ancora? Sì, ancora. Il problema si può risolvere tramite backtrack. La funzione ricorsiva, oltre al parametro i che identifica il prossimo bit da riempire, prende i parametri i_0 e i_1 che rappresentano il numero di bit 0 e 1 consecutivi che si possono scegliere, rispettivamente; e i parametri n_0, n_1 necessari per re-inizializzare questi valori quando si sceglie il bit opposto. All'inizio, $i = n$, $i_0 = n_0$ e $i_1 = n_1$.
Quando $i = 0$, tutti i bit sono stati scelti e si può stampare. Altrimenti, se $i_0 > 0$, si può ancora scegliere 0, si chiama ricorsivamente la funzione $(bRec)()$ con un bit consecutivo in meno per i_0 e re-inizializzando i_1 a n_1, perché abbiamo possiamo ricomonciare con un'eventuale sequenza di 1. In modo simile, se $i_1 > 0$ è possibile chiamare ricorsivamente $(bRec)()$ con un bit consecutivo in meno per i_1 e re-inizializzando i_0 a n_0.

La complessità è $O(n \cdot 2^n)$; infatti, nel caso in cui $n_0 = n_1 = n$, verranno stampate tutte le 2^n stringhe di n bit.

```
binary(int n, int n_0, int n_1)
    int[] S = new int[1...n]
    bRec(S, n, n_0, n_1, n_0, n_1)
```

```
bRec(int[] S, int i, int i_0, int i_1, int n_0, int n_1)
    if i == 0 then
        print(S)
    if i_0 > 0 then
        S[i] = 0
        bRec(S, i − 1, i_0 − 1, n_1, n_0, n_1)
    if i_1 > 0 then
        S[i] = 1
        bRec(S, i − 1, n_0, i_1 − 1, n_0, n_1)
```

Esercizio B2 – Discordville – Punti ≥ 11

Il problema, guarda un po', può essere risolto con programmazione dinamica in tempo lineare.
Sia $DP[i]$ la quantità massima di donazioni che possono essere ottenute considerando le sole prime i case. Ovviamente, la casa in posizione n può donare senza preclusioni, in quanto non ha nessuno dopo di lui. Quando si considera la posizione i-esima, è possibile seguire due scelte:

- se si prende la donazione della casa i-esima, poi si prende il massimo che si può ottenere con le prime $i - N[i] - 1$ case; se questo indice è minore di zero, si prende il valore $D[i]$ e basta;
- se non si prende la donazione della casa i-sima, si prende il massimo che si può ottenere con le prime $i - 1$ case.

Il valore massimo che stiamo cercando si trova in posizione $O[n]$.
Questo può essere espresso dalla seguente equazione ricorsiva:

$$DP[i] = \begin{cases} D[1] & i = 1 \\ \max\{DP[i-1], D[i]\} & i > 1 \wedge i - N[i] - 1 \leq 0 \\ \max\{DP[i-1], D[i] + DP[i - N[i] - 1]\} & i > 1 \wedge i - N[i] - 1 > 0 \end{cases}$$

Questo può essere tradotto nel seguente algoritmo basato su programmazione dinamica:

```
int discordville(int[] D, int[] N, int n)
    int[] DP = new int[1...n]
    DP[1] = D[1]
    for i = 2 to n do
        int prec = iif(i − N[i] − 1 > 0, DP[i − N[i] − 1], 0)
        DP[i] = max(DP[i − 1], D[i] + prec)
    return DP[n]
```

Esercizio B3 – Math ways – Punti ≥ 11

È possibile utilizzare la programmazione dinamica per risolvere il problema.
Sia $DP[i]$ il numero di modi per ottenere n a partire da i. Ovviamente, nel caso $i = n$, esiste un solo modo - non fare nulla. Altrimenti, $DP[i]$ è pari alla somma di $DP[i+1]$, $DP[2i]$, $DP[i^2]$. Se $i > n$, abbiamo sforato e ovviamente il numero di modi è pari a 0.
La formula ricorsiva per calcolare DP è quindi:

$$DP[i] = \begin{cases} 0 & i > n \\ 1 & i = n \\ DP[i+1] + DP[2i] + DP[i^2] & \text{altrimenti} \end{cases}$$

Scriviamo il codice in maniera bottom-up:

```
int mathways(int n, int k)
    int[] DP = new int[k...n]
    DP[n] = 1
    for i = n − 1 downto k do
        DP[i] = DP[i + 1]
        if 2 · i ≤ n then
            DP[i] = DP[i] + DP[2 · i]
        if i² ≤ n then
            DP[i] = DP[i] + DP[i²]
    return DP[k]
```

Il costo dell'algoritmo proposto è $\Theta(n)$.
Si noti che la richiesta che k sia maggiore di 3 è per non aver problemi nel distinguere $2k$ e k^2 quando $k = 2$. Ovviamente l'algoritmo proposto funziona anche nel caso $k = 1, 2$ se assumiamo che $2 \cdot 2$ e 2^2 siano due operazioni diverse. Altrimenti, è sufficiente inserire un caso particolare nella soluzione proposta.