Algoritmi e Strutture Dati
31/05/2013

Esercizio 1

Sia (u, v) l'arco con peso $w - 1$. Questo arco deve assolutamente far parte dello minimum spanning tree. Tutti gli altri archi possono essere selezionati con una qualunque visita in profondità o in ampiezza, tanto hanno peso uguale.
L'algoritmo lavora nel modo seguente:

- si cerca l'arco (u, v) con peso $w - 1$; questo può essere scritto con codice di costo $O(m + n)$, qui sotto assumo semplicemente che questa coppia di nodi sia data in input
- si fa partire una visita a partire dal nodo u, assicurandosi di seguire sempre per primo l'arco u, v
- si prosegue la visita normalmente (qui implementata come una BFS).

```
erdos(GRAPH G, NODE u, NODE v, NODE[] p)
visited ← new boolean[1 ... G.n]
foreach x ∈ G.V() do
    visited[x] ← false
QUEUE S ← Queue()
S.enqueue(u)
S.enqueue(v)
p[u] ← nil
p[v] ← u
visited[u] ← true
visited[v] ← true
while not S.isEmpty() do
    NODE x ← S.dequeue()
    foreach y ∈ G.adj(x) do
        if not visited[y] then                    % Il nodo y non è già stato scoperto
            visited[y] ← true
            p[y] ← x
            S.enqueue(y)
```

Esercizio 2

E' sufficiente modificare l'algoritmo per le componenti connesse visto a lezione:

```
integer[] connectedK(GRAPH G, integer k)
integer[] visited ← new BOOLEAN[1 ... G.n]
foreach u ∈ G.V() do visited[u] ← false

return ccdfs(G, visited, s, k)

ccdfs(GRAPH G, boolean[] visited[], NODE u, integer k)
    if k = 0 then
        return false
    visited[u] ← true
    foreach v ∈ G.adj(u) do
        if not visited[v] then
            if not ccdfs(G, visited, v, k − 1) then
                return false
return true
```

La complessità è pari a $O(n + \min\{m, k^2\})$, in quanto è necessario pagare $O(n)$ per l'inizializzazione di *visited*, e poi visitare al più k nodi, ognuno dei quali può avere al più k archi (se ne avesse $k+1$, prima o poi questi verrebbero visitati, e l'algoritmo terminerebbe prima)

Esercizio 3

Il problema può essere risolto con tecnica greedy. Ordiniamo i segmenti in senso non decrescente rispetto alla lunghezza, in modo che il segmento 1 abbia lunghezza minima e il segmento n lunghezza massima. Procediamo quindi a tagliare prima il segmento più corto, poi quello successivo e così via finché possibile (cioè fino a quando la lunghezza residua ci consente di ottenere almeno un'altro segmento). Lo pseudocodice può essere descritto in questo modo

```
integer maxSegmenti(integer[] V, integer n, integer L)
    sort(V, n)
    integer i ← 1
    while i ≤ n and L ≥ V[i] do
        L ← L − V[i]
        i ← i + 1
    return i − 1
```

Notate che una soluzione ottima può essere descritta come un insieme S, sottoinsieme di $\{1, \ldots, n\}$, costituito dagli indici dei segmenti le cui richieste vengono soddisfatte, dopo che il vettore è stato ordinato. Ovviamente, dovrà risultare che

$$\sum_{i \in S} V[i] \leq L$$

Per dimostrare la correttezza, dobbiamo dimostrare le seguenti proprietà:

- **Sottostruttura ottima**: sia S una soluzione ottima per il problema, e sia $x \in S$. E' possibile vedere che $S - \{x\}$ è una soluzione ottima per uno sfilatino di lunghezza $L - V[x]$, grazie alla tecnica del taglia e cuci: per assurdo, se fosse possibile trovare un sottinsieme S' tale che $|S'| > |S - \{x\}|$ e $\sum_{y \in S'} V[y] \leq L - V[x]$, allora l'insieme $S' \cup \{x\}$ avrebbe cardinalità superiore a $|S|$, sempre rispettando la lunghezza dello sfilatino. Quindi S non sarebbe ottima, contraddizione.
- **Scelta greedy**: Dato un problema di dimensione n, vogliamo dimostrare che esiste una soluzione ottima che contiene il segmento più piccolo m (memorizzato nell'indice 1, quando il vettore è ordinato). Sia S una soluzione ottima; possono darsi due casi, $m \in S$ (nel qual caso la nostra tesi è banalmente dimostrata), oppure $m \notin S$. Sia x un qualunque elemento di S; sia $S' = S - \{x\} \cup \{m\}$. Ovviamente,
$$\sum_{i \in S'} V[i] \leq L$$
in quanto $V[m] \leq V[x]$ dato che m è il segmento più corto (o uno dei segmenti più corti), e $|S| = |S'|$, quindi S' ha la stessa cardinalità di S ed è ottimale.

L'operazione di ordinamento può essere fatta in tempo $O(n \log n)$. Il successivo ciclo while ha costo $O(n)$ nel caso peggiore. Il costo complessivo dell'algoritmo risulta quindi $O(n \log n)$.

Esercizio 4

I due problemi sono molto simili, ma questo non può essere risolto tramite la stessa tecnica greedy di cui sopra. Ad esempio, si consideri un insieme di monete da $1, 2, 2, 2$ centesimi ed un IMU da pagare da 6 centesimi. E' ovvio che la soluzione esiste, ed è data da $2, 2, 2$; ma scegliere sempre e comunque la moneta 1 porta all'impossibilità di pagare la cifra esatta.
Utilizziamo quindi la programmazione dinamica. Date le prime i monete e una tassa t da pagare, il numero massimo di monete $M[i, t]$ può essere calcolato nel modo seguente:

$$M[i,t] = \begin{cases} -\infty & i = 0 \wedge t > 0; \text{oppure} \\ -\infty & t < 0; \text{oppure} \\ 0 & t = 0; \text{oppure} \\ \max(M[i-1,t], M[i-1, t-V[i]] + 1) & i > 0 \end{cases}$$

La logica è la seguente: restituisco $-\infty$ se ho ancora resta da dare ($t > 0$), ma non ho più monete a disposizione ($i > 0$); oppure, restituisco $-\infty$ se il resto che devo dare è minore di zero (il che significa che in precedenza ho usato una moneta troppo grande); oppure,

restituisco 0 se non devo dare più alcun resto; oppure, restituisco il massimo fra due possibilità: non utilizzare la moneta i-esima oppure la utilizza, aumentando di 1 il numero di monete usate.
Il codice può essere scritto nel modo seguente:

```
integer maxResto(integer[] V, integer i, integer t, integer[][] M)
    if i = 0 and t > 0 then
        return −∞
    if t < 0 then
        return −∞
    if t == 0 then
        return 0
    if M[i, t] == ⊥ then
        M[i, t] = max(maxResto(V, i − 1, t, M), maxResto(V, i − 1, t − V[i], M) + 1)
    return M[i, t]
```

La chiamata iniziale sarà $\mathsf{maxResto}(V, n, T)$. La complessità risultante sarà pari a $O(nT)$