Algoritmi e Strutture Dati - Prima provetta
03/05/12

Esercizio 1

Utilizzando il master theorem, è facile vedere che $T(n) = \Theta(\sqrt{n} \log n)$.
Dimostriamolo per sostituzione, partendo da $T(n) = O(\sqrt{n} \log n)$.
Coinvolgendo il logaritmo, il caso base è fra quelli problematici:

$$T(1) = 1 \not\leq c \log 1 = 0$$

Per questo motivo, consideriamo i valori i compresi fra 2 e 7, estremi inclusi; $\lfloor i/4 \rfloor$ in questo caso è pari a 0 o 1; scriviamo quindi

$$T(i) = 2T(\lfloor i/4 \rfloor) + \sqrt{i} = 2 + \sqrt{i} \leq c\sqrt{i} \log i \qquad \forall i : 2 \leq i \leq 7$$

da cui si ottiene:

$$c \geq \frac{2 + \sqrt{i}}{\sqrt{i} \log i} \qquad \forall i : 2 \leq i \leq 7$$

Per $i = 8$, $\lfloor i/4 \rfloor$ è pari a 2 e rientra nei casi base già risolti. Possiamo quindi fermarci a 7.
Nel passo induttivo, dobbiamo dimostrare che $T(n) \leq c\sqrt{n} \log n$ e supponiamo che la relazione $T(n') \leq c\sqrt{n'} \log n'$ sia già stata dimostrata per $2 \leq n' < n$.

$$\begin{aligned}
T(n) &\leq 2c\sqrt{\lfloor n/4 \rfloor} \log \lfloor n/4 \rfloor + \sqrt{n} \\
&\leq 2c\sqrt{n/4} \log n/4 + \sqrt{n} \\
&= c\sqrt{n} \log n/4 + \sqrt{n} \\
&= c\sqrt{n}(\log n - \log 4) + \sqrt{n} \\
&= c\sqrt{n} \log n - 2c\sqrt{n} + \sqrt{n} \leq c\sqrt{n} \log n
\end{aligned}$$

L'ultima disequazione è soddisfatta se $c \geq 1/2$. Poiché questa disequazione per c e tutte quelle derivanti dal caso base sono di tipo $\geq$, è sufficiente prendere il valore più alto fra questi valori come valore per c.
Consideriamo ora $T(n) = \Omega(\sqrt{n} \log n)$. Il caso base è simile a quello precedente:

$$T(i) = 2T(\lfloor i/4 \rfloor) + \sqrt{i} = 2 + \sqrt{i} \geq c\sqrt{i} \log i \qquad \forall i : 2 \leq i \leq 7$$

da cui si ottiene:

$$c \leq \frac{2 + \sqrt{i}}{\sqrt{i} \log i} \qquad \forall i : 2 \leq i \leq 7$$

Nel passo induttivo, dobbiamo dimostrare che $T(n) \geq c\sqrt{n} \log n$ e supponiamo che la relazione $T(n') \geq c\sqrt{n'} \log n'$ sia già stata dimostrata per $2 \leq n' < n$.

$$\begin{aligned}
T(n) &\geq 2c\sqrt{\lfloor n/4 \rfloor} \log \lfloor n/4 \rfloor + \sqrt{n} \\
&\approx 2c\sqrt{n/4} \log n/4 + \sqrt{n} \\
&= c\sqrt{n} \log n/4 + \sqrt{n} \\
&= c\sqrt{n}(\log n - \log 4) + \sqrt{n} \\
&= \sqrt{n}(c \log n - 2c + 1) \geq c\sqrt{n} \log n
\end{aligned}$$

L'ultima disequazione è vera per $c \leq 1/2$, il che è compatibile con gli altri estremi trovati per i casi base.

Esercizio 2

In generale, per evitare di dover fare una visita a partire da tutti i nodi per vedere se i cammini arrivano a v, si può utilizzare il grafo trasposto (costruibile in tempo $O(m + n)$, vedi libro e appunti) e considerare una visita che parta da v.

Si effettua una visita a partire dal nodo v sul grafo trasposto G^T (in tempo $O(m+n)$), utilizzando per esempio l'algoritmo che abbiamo scritto per identificare le componenti connesse; v è di tipo "roma" se e solo se tutti i nodi sono stati visitati a partire da v.

```
isRoma(GRAPH G^T, NODE v)
  boolean[] id ← new integer[1 ... G^T.n]
  foreach u ∈ G^T.V() do
      id[u] ← 0
  ccdfs(G^T, 1, v, id)
  foreach u ∈ G^T.V() do
      if id[u] = 0 then
          return false
  return true
```

Un modo alternativo si basa sul seguente concetto: un nodo può raggiungere v se ha un arco con il nodo v, oppure se può raggiungere un nodo che può raggiungere v. In altre parole, è possibile far partire una visita in profondità da tutti i nodi non visitati, e verificare se è possibile raggiungere v direttamente o indirettamente. Se non è possibile, v non è di tipo Roma. Poichè ogni nodo può essere visitato al massimo una volta, il costo computazionale è ancora $O(m+n)$.

```
isRoma(GRAPH G, NODE v)
  integer status ← new integer[1 ... G.N]
  foreach v ∈ G.V() − {r} do
      status[v] ← UNVISITED
  status[r] ← REACH-ROMA
  foreach v ∈ G.V() − {r} do
      if status[v] = UNVISITED then
          visitRoma(G, v, status)
      if status[v] = VISITED then
          return false
  return true
```

```
visitRoma(GRAPH G, NODE v, integer[] status)
  status[v] ← VISITED
  foreach w ∈ G.adj(v) do
      if status[w] = UNVISITED then
          visitRoma(G, w, status)
      if status[w] = REACH-ROMA then
          status[v] ← REACH-ROMA
```

Per la seconda parte, è ovviamente possibile ripetere la procedura isRoma() a partire da ogni nodo, con un costo computazionale $O(n(m+n)) = O(mn)$; ma è comunque possibile risolvere il problema in $O(m+n)$, sempre operando sul grafo trasposto.
Si effettui una visita in profondità toccando tutti i nodi del grafo trasposto, utilizzando il meccanismo di discovery/finish time. Sia v l'ultimo nodo ad essere chiuso. Si utilizzi ora la procedura roma(G^T, v) definita sopra; se otteniamo **true**, allora esiste un nodo Roma. Altrimenti, non esiste alcun nodo Roma in G. La dimostrazione è per assurdo. Supponiamo che esista un nodo w Roma; possono darsi due casi:

- se w è stato scoperto prima di v, allora v è un discendente di w e deve essere stato chiuso prima di w, assurdo;
- se v è stato scoperto prima di w, allora possono darsi due casi:
  - w è un discendente di v; ma allora anche v è Roma, perchè v può raggiungere w e da esso tutti gli altri nodi; assurdo.
  - w non è un discendente di v; non esiste quindi un cammino di da v a w, e quindi v viene chiuso prima di w, assurdo.

Per scrivere il codice, utilizziamo la procedura topsort() definita nei lucidi.

```
Roma(GRAPH G^T)
    STACK S ← topsort(G^T)
    NODE v ← S.pop()
    return isRoma(G^T, v)
```

La procedura risultante è $O(m+n)$.

Esercizio 3

Si ordini il vettore e si considerino le somme degli elementi i e $n-i+1$, con $1 \leq i \leq n/2$. Se sono tutti uguali, si ritorna **true**, altrimenti si ritorna **false**. Il costo dell'algoritmo è dominato dall'ordinamento, ed è quindi $O(n \log n)$.
Dimostrazione: supponiamo per assurdo che esista un insieme di coppie che rispetti le condizioni per restituire **true**, in cui l'elemento maggiore M sia associato ad un elemento M' diverso dal minore m ($m < M'$). Quindi il minore m è associato ad un elemento m' diverso dal massimo M ($m' < M$). Allora $m + m' < M + M'$, il che contraddice l'ipotesi che tale insieme di coppie rispetti le condizioni per restituire **true**.
L'algoritmo consiste quindi nel scegliere il minore e il maggiore, e confrontarli con i secondi minori e maggiori, i terzo minori e maggiori, e così via.

```
checkPairs(integer[] A, integer n)
    sort(A, n)
    integer s ← A[1] + A[n]
    for i ← 2 to n/2 − 1 do
        if A[i] + A[n − i + 1] ≠ s then
            return false
    return true
```

Una soluzione più efficiente, in tempo $O(n)$, è la seguente: si calcola

$$S = \frac{\sum_{i=1}^{n} A[i]}{n/2}$$

S rappresenta il valore che deve avere la somma delle coppie. A questo punto si può utilizzare una hash table inserendo tutti i valori come chiavi. Per ogni valore $A[i]$, si cerca $S - A[i]$; se uno dei valori è assente, allora il vettore non può essere partizionato nel modo richiesto. Assumendo che la tabella hash sia ben dimensionata, il costo di ogni operazione su di essa è $O(1)$ e quindi il costo totale é O(n).

```
checkPairs(integer[] A, integer n)
    integer S ← 0
    for i ← 1 to n do S ← S + A[i]
    S ← S/(n/2)
    HASH H ← Hash()
    for i ← 1 to n do
        H.insert(A[i], A[i])
    for i ← 1 to n do
        if H.lookup(S − A[i]) = nil then
            return false
    return true
```

Una soluzione migliore ancora si basa sulla seguente affermazione: un il vettore A è suddivisibile in coppie se e solo se:

$$\sum_{i=1}^{n} A[i] = n/2 \cdot (\mathit{min} + \mathit{max})$$

dove *min* e *max* sono il valore minimo e massimo contenuti nel vettore.
La proprietà si dimostra per induzione su n.

- Caso base: Per $n = 2$,
$$A[1] + A[2] = 1 \cdot (\mathit{min} + \mathit{max})$$

- Passo induttivo: supponiamo la proprietà sia vera per i numeri pari inferiori a n e dimostriamolo per il numero pari n. Si costruisca un vettore A' rimuovendo da A il minimo e il massimo. A' ha $n-2$ elementi e quindi ha dimensione pari. Per induzione, sappiamo che:
$$\sum_{i=1}^{n-2} A'[i] = (n-2)/2 \cdot (\mathit{min}' + \mathit{max}')$$
da cui si ottiene:
$$\sum_{i=1}^{n} A[i] = \sum_{i=1}^{n-2} A'[i] + \mathit{min} + \mathit{max} = (n-2)/2 \cdot (\mathit{min}' + \mathit{max}') + (\mathit{min} + \mathit{max})$$
Quest'ultima espressione è pari a $n/2 \cdot (\mathit{min} + \mathit{max})$ se e solo se $\mathit{min}' + \mathit{max}' = \mathit{min} + \mathit{max}$, la stessa condizione che abbiamo utilizzato all'inizio.

Possiamo quindi utilizzare il codice seguente, che può essere eseguito in tempo $O(n)$:

```
checkPairs(integer[] A, integer n)
  integer S ← 0
  for i ← 1 to n do
    S ← S + A[i]
  return min(A, n) + max(A, n) · (n/2) = S
```

Esercizio 4

Da un certo punto di vista, il problema è simile a quello del resto, in cui però non è richiesto di dare il resto esatta, ma il maggior resto possibile. In realtà, il problema è più simile a quello chiamo SubsetSum.
Il massimo valore $X[i, w]$ che può essere ottenuto con i primi $i \leq n$ valori e tale per cui $X[i, w]$ è minore di w è pari a:

$$X[i, w] = \begin{cases} 0 & i = 0 \lor w = 0 \\ -\infty & i > 0 \land w < 0 \\ \max\{X[i-1, w], X[i, w - V[i]] + V[i] & \text{altrimenti} \end{cases}$$

La soluzione al problema si trova in $X[n, W]$; è possibile calcolare X tramite il seguente algoritmo basato su memoization:

```
integer sum(integer[] V, integer i, integer w, integer[][] X)
  if i = 0 or w = 0 then
    return 0
  if w < 0 then
    return −∞
  if X[i, w] = ⊥ then
    X[i, w] ← max{sum(X, i − 1, w), sum(X, i, w − V[i]) + V[i]}
  return X[i, w]
```

E' possibile ottenere x tramite il seguente algoritmo:

```
integer[] getsum(integer[] V, integer n, integer W, integer[][] X)
    integer[] x = new integer[1...n]
    for j ← 1 to n do x[i] ← 0
    integer i ← n
    while i > 0 do
        if X[i, W − V[i]] + V[i] > X[i − 1, W] then
            x[i] ← x[i] + 1
            W ← W − V[i]
        else
            i ← i − 1
    return x
```

La complessità è $O(nW)$.