

Corso di Algoritmi e Strutture Dati—Informatica per il Management

Prova scritta del 01/07/2010

Note: Il tempo a disposizione per lo svolgimento della prova è di **due ore e 30 minuti**. Durante la prova non è possibile consultare libri o appunti, né interagire in alcun modo con gli altri studenti. Chi copia in tutto o in parte il compito verrà escluso dalla prova; l'esclusione potrà avvenire anche dopo il termine della prova stessa, e riguarderà entrambi gli "estremi" della copiatura. È consentita la consultazione del formulario allegato a questo compito. **Ove si richieda la descrizione di un algoritmo, è necessario fornire pseudo-codice oppure codice Java** (in quest'ultimo caso verranno ignorati eventuali errori minori di sintassi che non compromettano la comprensione dell'algoritmo né la sua correttezza). Tutte le risposte devono essere adeguatamente motivate.

Esercizio 1.

Si consideri la funzione $\text{Fun}(n)$, con $n \geq 1$ intero, definita dal seguente algoritmo ricorsivo:

```
algoritmo Fun(int n) → int
  if (n ≤ 2) then
    return n;
  else
    return Fun(n-1) - 2*Fun(n-2);
  endif
```

1. Determinare un limite inferiore sufficientemente accurato del tempo di esecuzione $T(n)$ [punti 5]
2. Determinare un limite superiore sufficientemente accurato del tempo di esecuzione $T(n)$ [punti 5]

Soluzione

Il tempo di esecuzione $T(n)$ può essere calcolato come soluzione della seguente equazione di ricorrenza:

$$T(n) = \begin{cases} c_1 & \text{se } n \leq 2 \\ T(n-1) + T(n-2) + c_2 & \text{altrimenti} \end{cases}$$

Questa equazione di ricorrenza è esattamente la stessa della funzione ricorsiva che calcola i numeri di Fibonacci. Si rimanda alle slides del corso (o al libro di testo) per la relativa stima di un limite inferiore e superiore.

Esercizio 2.

Si consideri un array $A[1..n]$ composto da $n \geq 1$ valori interi tutti distinti. Supponiamo che l'array sia ordinato in senso crescente ($A[1] < A[2] < \dots < A[n]$).

1. Scrivere un algoritmo efficiente che, dato in input l'array A , determina un indice i , se esiste, tale che $A[i] = i$. Nel caso esistano più indici che soddisfano la relazione precedente, è sufficiente restituirne uno qualsiasi. È richiesta la dimostrazione (a parole) della correttezza dell'algoritmo proposto. *(Suggerimento: considerare un approccio ricorsivo di tipo divide-et-impera)* [punti 7]
2. Determinare il costo computazionale dell'algoritmo di cui al punto 1. [punti 4]

Soluzione

È possibile utilizzare il seguente algoritmo ricorsivo, molto simile a quello della ricerca binaria (i e j rappresentano rispettivamente gli indici estremi del sottovettore $A[i..j]$ in cui effettuare la ricerca, all'inizio la funzione va invocata con $i=1, j=n$):

```
algoritmo cerca_i(Array A[1..n], int i, int j) → int
  if (i > j) then
    errore "Non esiste alcun i tale che A[i] = i"
```

```

endif
m := (i+j)/2; // posizione centrale nel sottovettore A[i..j]
if ( A[m] == m ) then
    return m;
elseif (A[m] > m ) then
    return cerca_i(A,i,m-1);
else
    return cerca_i(A,m+1,j);
endif

```

Si noti che i valori contenuti nel vettore potrebbero anche essere negativi, per cui si può anche avere il caso in cui $A[m] < m$.

Notiamo che se $A[m] < m$, l'indice i tale per cui $A[i]=i$ non potrà mai trovarsi nella prima metà $A[i..m-1]$, in quanto per ipotesi (il vettore contiene tutti interi distinti, ed è ordinato) si avrà che $A[k] < k$ per ogni $k=i, i+1, \dots, m-1$. Per rendersi conto di ciò, consideriamo $A[m-1]$. Poiché l'array è ordinato e non esistono duplicati, si ha che $A[m-1] < A[m]$. Ma poiché $A[m] < m$, si ha: $A[m-1] < A[m] < m$, da cui $A[m-1] < m-1$. Lo stesso ragionamento si può ripetere per $m-2, m-3$ eccetera. Il ragionamento simmetrico si applica al caso $A[m] > m$. L'algoritmo proposto è una semplice variante dell'algoritmo di ricerca binaria, e ha lo stesso costo computazionale $O(\log n)$.

Esercizio 3.

Consideriamo un grafo orientato $G=(V, E)$ i cui archi abbiano pesi non negativi. Denotiamo con $w(u,v)$ il peso dell'arco orientato (u,v) . Ricordiamo che l'algoritmo di Dijkstra per il calcolo dei cammini minimi da una singola sorgente $s \in V$ ha la seguente struttura generica:

```

algoritmo DijkstraGenerico(grafo G, vertice s) → albero
    inizializza D tale che  $D_{sv} = +\infty$  per  $v \neq s$ , e  $D_{ss} = 0$ 
    T := albero formato dal solo vertice s
    while (T ha meno di n nodi) do
        trova l'arco (u,v) incidente su T con  $D_{su} + w(u,v)$  minimo
         $D_{sv} := D_{su} + w(u,v)$ ;
        rendi u padre di v in T
    end while
return T;

```

1. Scrivere una versione dell'algoritmo di Dijkstra che non faccia uso di una coda di priorità ma di una semplice lista, in modo che ad ogni passo esamini sistematicamente gli archi incidenti per individuare quello che minimizza la distanza. [punti 7]
2. Determinare il costo computazionale della variante dell'algoritmo di Dijkstra descritta al punto 1. Specificare quale struttura dati viene usata per rappresentare il grafo. [punti 4]

Soluzione

Supponiamo che, per ogni vertice s , la distanza tra la sorgente e s sia indicata con l'attributo $s.d$; possiamo quindi scrivere la variante dell'algoritmo di Dijkstra come segue:

```

algoritmo DijkstraListe(grafo G, vertice s) → albero
    for each v in V do
        v.d =  $+\infty$ 
    endfor
    s.d = 0; // il nodo sorgente ha distanza zero da se stesso
    T := albero formato dal solo vertice s
    Lista L;
    L.insert(s); // inserisci s in L
    while (not L.empty()) do
        sia u il nodo di L con minimo valore di u.d // costo:  $O(n)$  nel caso peggiore

```

```

rimuovi u da L // costo: O(1) se L è una lista coppiamente concatenata
for each (u,v)∈E do
  if (v.d = +∞) then
    v.d = u.d + w(u,v);
    rendi u padre di v in T;
    L.insert(u); // u non era nella lista, inseriscilo. Costo O(1)
  else if (u.d + w(u,v) < v.d) then
    v.d := u.d + w(u,v);
    rendi u nuovo padre di v in T;
    // u era già nella lista, non va reinserito
  endif
endfor
endwhile
return T;

```

Si noti che l'algoritmo DijkstraListe() è leggermente più semplice da descrivere rispetto all'algoritmo di Dijkstra implementato con code di priorità visto a lezione. Infatti nel caso in cui $u.d + w(u,v) < v.d$, cioè nel caso in cui abbiamo scoperto un cammino più breve tra s e v che passa attraverso il nodo u , il nodo v non va reinserito nella struttura dati (nel nostro caso la lista), perché sicuramente vi è già contenuto.

Il costo di DijkstraListe() è $O(n^2)$, essendo n il numero di nodi del grafo ($n=|V|$). Notiamo infatti quanto segue:

1. Il ciclo while viene eseguito al più n volte. Questo perché ad ogni iterazione viene estratto un nodo dalla lista, e una volta estratto un nodo questo non viene più reinserito (questo è lo stesso identico ragionamento che abbiamo fatto per calcolare il costo computazionale dell'algoritmo di Dijkstra implementato tramite coda di priorità);
2. La ricerca del nodo u tale che $u.d$ sia minimo ha costo $O(n)$. Infatti la lista L conterrà al più n nodi (in quanto il grafo ha n nodi)
3. Il corpo del ciclo for viene eseguito $O(n)$ volte, in quanto un nodo può essere incidente al più a $n-1=O(n)$ altri nodi.