ESERCITAZIONI DI *INTRODUZIONE AGLI ALGORITMI*
(A.A. 08/09)

DISPENSA N. 4

1. Ricerca Binaria Ricorsiva

L'algoritmo Ricerca Binaria risolve il problema della ricerca di una chiave in un vettore. È un esempio di algoritmo che sfrutta una *ipotesi strutturale* sull'input: si suppone cioè che il vettore sia già ordinato (nel nostro esempio, in modo non decrescente). La seguente è una specifica ricorsiva.

```
algoritmo RicBin(array A, int k, sin, des){
  if (sin == des){
    if (k==A[sin]) then return sin;
    else return -1;
  }
  centro = (sin+des)/2;
  if (k <= A[centro]) then return RicBin(A,k,sin,centro);
  else return RicBin(A, k, centro+1, des);
}
```

Indichiamo con $|A|$ la dimensione del vettore A. La prima chiamata all'algoritmo è con $sin = 1$, $des = |A|$ (come precondizione dell'algoritmo richiediamo che $1 \leq sin \leq des \leq |A|$).

Analizziamo il codice. Abbiamo due casi a seconda che sia $(sin = des)$ o $(sin \neq des)$. Si osserva facilmente che sarà sempre $sin \leq des$, e dunque i due casi sono $(sin = des)$ o $(sin < des)$. Per come l'algoritmo è specificato nel codice qui sopra dovremmo, per essere precisi, scrivere la relazione di ricorrenza come funzione di tutti gli input, i.e., $T(A, k, sin, des)$. Per scrivere la relazione di ricorrenza in modo succinto e chiaro osserviamo che il parametro che ci interessa è la dimensione del segmento di vettore su cui l'algoritmo lavora, i.e., la dimensione della porzione contenuta tra sin e des. Alla prima chiamata l'algoritmo lavora su tutto il vettore in input, mentre ad ogni chiamata ricorsiva l'algoritmo lavora su un segmento di dimensione 1/2 la dimensione del segmento considerato nella chiamata precedente. Chiamiamo n la dimensione del segmento di vettore su cui l'algoritmo lavora (i.e., n è $des - sin + 1$), e riscriviamo i due casi dell'algoritmo come funzioni di n. Il caso $(sin = des)$ equivale al caso in cui la dimensione del segmento di vettore considerato è $n = 1$, mentre il caso $(sin < des)$ equivale al caso in cui la dimensione del vettore considerato è $n > 1$. Nel primo caso l'algoritmo fa operazioni con costo costante, mentre nel secondo caso fa delle operazioni di costo costante e una chiamata ricorsiva su un segmento del vettore di dimensione $n/2$.

Note preparate da Lorenzo Carlucci, carlucci@di.uniroma1.it.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ T(n/2) + c & \text{if } n > 1 \end{cases}$$

Per iterazione abbiamo

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/2^2) + c + c \\ &\dots \\ &= T(n/2^k) + k \cdot c \end{aligned}$$

Ricadiamo nel caso base quando $n/2^k = 1$, e dunque abbiamo

$$T(n) = T(n/2^{\log n}) + (\log n) \cdot c = T(1) + (\log n) \cdot c = O(1) + (\log n) \cdot c = O(\log n).$$

Per chi vuole applicare il Teorema Master: dobbiamo calcolare $\log_b a$, che in questo caso è $\log_2 1 = 0$, e confrontare $n^{\log_b a} = n^0$ con $f(n)$, che in questo caso è costante. Dunque abbiamo che $f(n)$ e $n^{\log_b a}$ sono asintoticamente uguali, i.e.,

$$f(n) = c = \Theta(1) = \Theta(n^0) = \Theta(n^{\log_b a}).$$

Dunque, per il Teorema Master, $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n)$.

2. Ricerca per confronti: limite assoluto

Analogamente a quanto fatto per il problema dell'ordinamento, è possibile dare un limite inferiore alla complessità di *qualunque* algoritmo che risolva per confronti il problema della ricerca.

Ci chiediamo cioè: dato un algoritmo $\mathcal{A}$ che risolve *per confronti* il problema della ricerca di una chiave k in un vettore V di n interi, quanti confronti sono necessari?

Richiediamo che $\mathcal{A}$ ritorni la posizione di k in V se k appare in V, e che segnali che k non appare in V se questo è il caso. Quante sono i casi tra cui $\mathcal{A}$ deve essere capace di distinguere? Sono $n + 1$: il caso in cui k non compare in V e gli n casi: $k = V[1]$, $k = V[2]$, $\dots$, $k = V[n]$. Dunque: per risolvere il problema della ricerca su un input di dimensione n, l'algoritmo $\mathcal{A}$ deve essere capace di distinguere tra $n + 1$ casi diversi.

Quanti casi si possono distinguere con 1 confronto? Un confronto tra due elementi a e b ha 3 risultati possibili: $a < b$, $a = b$, e $a > b$. Dunque con 1 confronto si possono distinguere al più tre casi diversi. Con 2 confronti successivi si possono distinguere 3^2 casi, con 3 confronti 3^3 casi, e così via. Con s confronti possiamo distinguere tra 3^s casi diversi.

Dato che $\mathcal{A}$ deve essere capace, su input di dimensione n, di distinguere tra $n+1$ casi diversi, se $\mathcal{A}$ fa x confronti, deve valere

$$n + 1 \leq 3^x,$$

i.e. $\log(n + 1) \leq x$. In altre parole, il numero di confronti necessari ad $\mathcal{A}$ per risolvere il problema della ricerca su un input di dimensione n è almeno $\log(n+1)$.

Per tanto, la complessità di un qualunque algoritmo che risolve per confronti il problema della ricerca è $\Omega(\log(n))$.

Osservazione 2.1. Nell'analisi qui sopra abbiamo fatto *due passi*. Il primo è quello di calcolare, in funzione della dimensione n dell'input, il numero $f(n)$ dei casi che un algoritmo deve poter distinguere per risolvere il problema in esame (questa valutazione dipende dal problema e nel caso della ricerca è $f(n) = n + 1$). In secondo luogo abbiamo calcolato, dato il numero $f(n)$ dei casi da distinguere, e dato lo strumento a nostra disposizione (i confronti, nel nostro caso con tre possibili risposte), il numero $g(f(n))$ di confronti necessari (questa valutazione dipende dallo strumento di decisione a disposizione dell'algoritmo, nel nostro caso $g(f(n)) = \log_3(n + 1)$).

3. Esercizio su algoritmi ricorsivi

Questo è l'esercizio 2 della prova del 05 Febbraio 2009. Consideriamo i seguenti algoritmi.

```
algoritmo Algo1(array T, indici i,j){
  k ← i
  while (k ≤ j) do{
    h ← i
    while (h ≤ j) do {
      ISTR
      h ← h + 1}
    k ← k + 1}
```

Nel codice qui sopra, ISTR denota un blocco di istruzioni di costo costante.

```
algoritmo Algo2(array T, indici i,j){
  if (i < j) then
    numEl ← j − i + 1
    Algo2(T, i, i + numEl/4)
    Algo2(T, j − numEl/4, j)
    Algo1(T,i,j)
}
```

Stimiamo la complessità di Algo2. Dato che Algo2 chiama Algo1, cominciamo a stimare la complessità di Algo1. Consideriamo i valori assunti da h (ciclo **while** interno) in dipendenza dei valori assunti da k (ciclo **while** esterno). Assumiamo, senza perdita di generalità in questo caso, che il costo costante del blocco $Istr$ sia uguale a 1.

k nel ciclo **while** esterno assume i valori $i, i + 1, \ldots, j$ $(= j - i + 1$ valori).

h nel ciclo **while** interno assume i valori seguenti:

- Per $(k = i)$: $h = i, i + 1, \ldots, j$ $(= j - i + 1$ valori)
- Per $(k = i + 1)$: $h = i, i + 1, \ldots, j$ $(= j - i + 1$ valori)

  ...
- Per $(k = j)$: $h = i, i + 1, \ldots, j$ $(= j - i + 1$ valori)

In conclusione, per ciascuna delle $j - i + 1$ iterazioni del ciclo **while** esterno, il ciclo **while** interno compie $j - i + 1$ iterazioni. La stima di Algo1 è dunque $T(n) = n^2$, dove n è la quantità che ci interessa per valutare il costo di una chiamata Algo1(T,i,j), i.e., la dimensione $j - i + 1$ della porzione di array T compresa tra i e j (la quantità chiamata *numEl* in Algo2).

Passiamo ora a stimare Algo2. Nel caso non base su un input di dimensione n (> 1) Algo2 effettua due chiamate ricorsive su input di dimensione $n/4$ e una chiamata ad Algo1 su input di dimensione n. Sappiamo già che il costo di Algo1 è quadratico. Possiamo scrivere così la relazione di ricorrenza per Algo2.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T(n/4) + n^2 & \text{if } n > 1 \end{cases}$$

Proviamo ad applicare il Teorema Master. Dobbiamo calcolare $\log_b a$, che in questo caso è $\log_4 2 = 1/\log_2 4 = 1/2$, e confrontare $n^{\log_b a} = n^{1/2} = \sqrt{n}$ con $f(n)$ che in questo caso è n^2. Abbiamo $f(n) = n^2 = \Omega(n^{\frac{1}{2}+\epsilon})$ per ogni $\epsilon > 0$ tale che $1/2 + \epsilon < 2$, e resta da verificare se, per un $c < 1$, per n abbastanza grande, vale $af(n/b) \leq cf(n)$. Abbiamo:

$$af(n/b) = 2f(n/4) \leq 2(n^2/4^2) = n^2/8.$$

Quindi basta scegliere $c = 1/8$ per soddisfare la condizione di regolarità. Siamo dunque nel terzo caso del Teorema Master, e possiamo concludere che $T(n) = f(n) = \Theta(n^2)$.

Esercizio 3.1. Cosa succederebbe se, nell'esempio qui sopra, avessimo come stima di Algo1 $f(n) = \Theta(n^2)$, anziché $f(n) = n^2$? Potremmo comunque applicare il terzo caso del Teorema Master? E se sapessimo solo $f(n) = O(n^2)$?

Per chi non ama il Teorema Master, risolviamo per iterazione.

$$\begin{aligned} T(n) &= 2T(n/4) + n^2 \\ &= 2(2T(n/4^2) + (n/4)^2) + n^2 \\ &= 2^2T(n/4^2) + 2(n/4)^2 + n^2 \\ &= 2^2(2T(n/4^3) + (n/4^2)^2) + 2(n/4)^2 + n^2 \\ &= 2^3T(n/4^3) + 2^2(n/4^2)^2 + 2(n/4)^2 + n^2 \\ &\dots \\ &= 2^kT(n/4^k) + 2^{k-1}(n/4^{k-1})^2 + 2^1(n/4^1)^2 + 2^0(n/4^0)^2 \\ &= 2^kT(n/4^k) + \sum_{i=0}^{k-1} 2^i(n/4^i)^2. \end{aligned}$$

Il caso base è raggiunto per $k = \log_4 n$ e dunque abbiamo

$$T(n) = 2^{\log_4 n}T(1) + \sum_{i=0}^{\log_4 n - 1} 2^i(n/4^i)^2.$$

Valutiamo la sommatoria a secondo termine come segue, ricordandoci che

$$\log_4 n = \log_{2^2} n = \frac{\log_2 n}{2}.$$

$$\begin{aligned}
\sum_{i=0}^{\log_4 n-1} 2^i(n/4^i)^2 &= n^2 \sum_{i=0}^{\log_4 n-1} 2^i/4^{2i} \\
&= n^2 \sum_{i=0}^{\log_4 n-1} 1/8^i \\
&= n^2 \left(\frac{1-\frac{1}{8^{\log_4 n}}}{1-\frac{1}{8}} \right) \\
&= n^2 \left(\frac{8}{7} \left(1 - \frac{1}{(2^3)^{\frac{\log_2 n}{2}}} \right) \right) \\
&= n^2 \left(\frac{8}{7} \left(1 - \frac{1}{n^{3/2}} \right) \right) \\
&= \frac{8n^2}{7} - \frac{8n^2}{7n^{3/2}} \\
&= \frac{8n^2}{7} - \frac{8\sqrt{n}}{7}
\end{aligned}$$

Concludendo,

$$T(n) = 2^{\log_4 n} T(1) + \sum_{i=0}^{\log_4 n-1} 2^i(n/4^i)^2 = O(\sqrt{n}) + \frac{8n^2}{7} - \frac{8\sqrt{n}}{7} = O(n^2).$$

4. Somma Massima

Consideriamo il seguente algoritmo.

```
int SM(array A, int n){
  max = 0;
  for i = 1 to n
    for j = i to n
      sum = 0
      for k = i to j
        sum = sum + A[k]
      if (sum > max) then max ← sum
  return max
}
```

L'algoritmo restituisce il valore del segmento di somma massima contenuta nell'array di interi A di dimensione n. (Possiamo assumere che l'array contenga almeno un elemento positivo e un elemento negativo, per evitare i casi banali).

Stimiamo la complessità della procedura.

Nel primo ciclo **for** la variabile i assume i valori $1, \ldots, n$. Il ciclo ha dunque n iterazioni. Valutiamo quante volte viene eseguito il corpo del secondo ciclo **for**, in dipendenza di i.

- Per $(i = 1)$, $j = 1, 2, \ldots, n$, $(= n - i + 1$ iterazioni),
- Per $(i = 2)$, $j = 2, 3, \ldots, n$ $(= n - i + 1$ iterazioni),

- ...
- Per $i = n$, $j = n$ $(= n - i + 1$ iterazioni).

Dunque il corpo del secondo ciclo **for** viene eseguito

$$n + (n-1) + \cdots + 1 = \sum_{i=1}^{n} = (n+1)n/2$$

volte, dando una limitazione inferiore quadratica al costo dell'algoritmo.

Vediamo ora quante volte viene eseguito il corpo del terzo ciclo **for**, in particolare la riga $sum = sum + A[k]$.

- Per $(i = 1$ e $j = 1)$, $k = 1$: $(= j - i + 1$ iterazioni),
- Per $(i = 1$ e $j = 2)$, $k = 1, 2$: $(= j - i + 1$ iterazioni),
- ...
- Per $(i = 1$ e $j = n)$, $k = 1, 2, \ldots, n$: $(= j - i + 1$ iterazioni),

* Per $(i = 2$ e $j = 2)$, $k = 2$: $(= j - i + 1$ iterazioni),
* Per $(i = 2$ e $j = 3)$, $k = 2, 3$: $(= j - i + 1$ iterazioni),
* ...
* Per $(i = 2$ e $j = n)$, $k = 2, \ldots, n$: $(= j - i + 1$ iterazioni),

...

- Per $(i = n - 1$ e $j = n - 1)$, $k = n - 1$: $(= j - i + 1$ iterazioni),
- Per $(i = n - 1$ e $j = n)$, $k = n - 1, n$: $(= j - i + 1$ iterazioni),
- ...
- Per $(i = n$ e $j = n)$, $k = n$: $(= j - i + 1$ iterazioni).

Dunque, all'iterazione j-esima del secondo ciclo **for** corrispondente alla iterazione i-esima del primo ciclo **for**, la variabile k del terzo ciclo **for** assume $j - i + 1$ valori. Complessivamente, il corpo del terzo ciclo **for** costa

$$\sum_{i=1}^{n} \sum_{j=i}^{n} (j - i + 1).$$

Osserviamo che

$$\sum_{j=i}^{n} (j - i + 1) = 1 + 2 + \cdots + (n - i + 1) = \sum_{h=1}^{n-i+1} h.$$

Usiamo la seguente maggiorazione banale per valutare questa somma dal basso.

$$\sum_{h=1}^{n-i+1} h = \frac{(n-i+1)(n-i+2)}{2} \geq \frac{(n-i+1)(n-i+1)}{2}.$$

Allora

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1) &\geq \sum_{i=1}^{n}\sum_{h=1}^{n-i+1} h \\ &\geq \sum_{i=1}^{n}\frac{(n-i+1)^2}{2} \\ &= \sum_{i=0}^{n-1}\frac{(n-i)^2}{2} \\ &= \frac{1}{2}\sum_{i=0}^{n-1}(n-i)^2.\end{aligned}$$

Osserviamo che, se $i \geq 2$, allora $i^2 \geq 2i$, e dunque dunque $(n-i)^2 = n^2-2i+i^2 \geq n^2$.

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i+1) &\geq \frac{1}{2}\sum_{i=0}^{n-1}(n-i)^2 \\ &\geq \frac{1}{2}\left(n^2+(n-1)^2+\sum_{i=2}^{n-1}(n-i)^2\right) \\ &\geq \frac{1}{2}\left(n^2+(n-1)^2+\sum_{i=2}^{n-1}n^2\right) \\ &\geq \frac{1}{2}(n^2+(n-1)^2+(n-2)n^2) \\ &\geq \frac{1}{2}(n^2+(n-1)^2+n^3-2n^2).\end{aligned}$$

Dunque l'algoritmo ha complessità almeno cubica, $\Omega(n^3)$.

Un algoritmo che risolve il problema in modo più efficiente (quadratico) è il seguente. L'algoritmo sfrutta il fatto che la somma del segmento $A[i], A[j+1]$ è uguale alla somma del segmento $A[i], A[j]$ più $A[j+1]$.

```
int SM2(array A, int n){
  max = 0;
  for i = 1 to n {
    sum = 0;
    for j = i to n {
      sum = sum + A[j]
      if (sum > max) then max = sum
    }
  }
return max
}
```

Esercizio 4.1. Stimare la complessità di $SM2$.