Algoritmi e Strutture Dati
8 Febbraio 2021

1. La leggibilità è un prerequisito: parti difficili da leggere potranno essere ignorate.
2. Quando si presenta un algoritmo è fondamentale spiegare l'idea e motivarne la correttezza.
3. L'efficienza e l'aderenza alla traccia sono criteri di valutazione delle soluzioni proposte.
4. Si consegna la scansione dei fogli di bella copia, e come ultima pagina un documento di identità.

Domande

Domanda A (8 punti) Si consideri la funzione ricorsiva `search(A,p,r,k)` che dato un array A, ordinato in modo crescente, un valore k e due indici p, q con $1 \leq p \leq r \leq A.length$ restituisce un indice i tale che $p \leq i \leq r$ e $A[i] = k$, se esiste, e altrimenti restituisce 0.

```
search(A,p,r,k)
   if p <= r
      q = (p+r)/2
      if A[q] = k
         return q
      elseif A[q] < k
         return search(A,q+1,r,k)
      else
         return search(A,p,q-1,k)
   else
      return 0
```

Dimostrare induttivamente che la funzione è corretta. Inoltre, determinare la ricorrenza che esprime la complessità della funzione e risolverla con il Master Theorem.

Soluzione: La prova di correttezza avviene per induzione sulla lunghezza $l = r - p + 1$ del sottoarray $A[p..r]$ di interesse. Se $l = 0$, ovvero $p > r$, la funzione ritorna correttamente 0, dato che non ci sono elementi nel sottoarray e quindi non ci possono essere elementi $= k$. Se invece $l > 0$ si calcola $q = \lfloor (p + r)/2 \rfloor$ e si distinguono tre casi:

- se $A[q] = k$ si ritorna correttamente k
- se $A[q] < k$, dato che l'array è ordinato certamente $A[j] \leq A[q] < k$ per $p \leq j \leq q$. Quindi il valore k può trovarsi solo nel sottoarray $A[q + 1, r]$. La chiamata $search(A, q + 1, r, k)$, dato che il sottoarray ha lunghezza minore di l, per ipotesi induttiva restituisce un indice i tale che $q + 1 \leq i \leq r$ e $A[i] = k$, se esiste, e altrimenti restituisce 0. Per l'osservazione precedente, questo è il valore corretto da restituire anche per $search(A, p, r, k)$ e si conclude.
- se $A[q] > k$ si ragiona in modo duale rispetto al caso precedente.

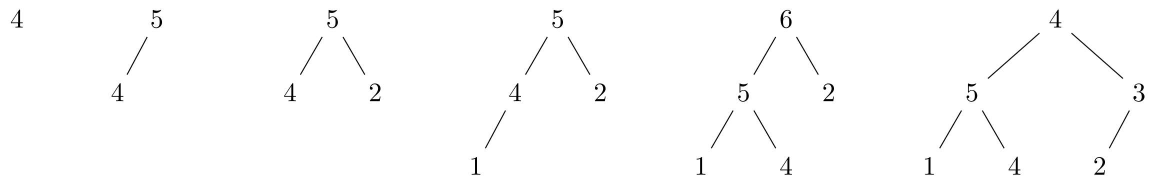
Per quanto riguarda la ricorrenza, ignorando i casi base, dato che la funzione ricorre su di un array la cui dimensione è la metà di quello originale, si ottiene:

$$T(n) = T(n/2) + c$$

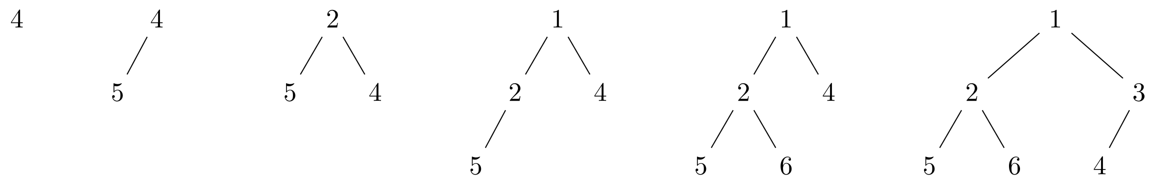
Rispetto allo schema standard del master theorem ho $a = 1$, $b = 2$ e $f(n) = c$. Ho dunque che $f(n) = 1 = \Theta(n^{\log_2 1} = n^0 = 1$. Pertanto si conclude che $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$.

Domanda B (5 punti) Si dia la definizione della struttura dati max-heap. Si supponga di inserire in un max-heap inizialmente vuoto, in successione, gli elementi 4, 5, 2, 1, 6, 3. Fornire la rappresentazione (ad albero) del max-heap così ottenuto. Si motivi la risposta mostrando la sequenza di max-heap ottenuti.

Soluzione: Per la definizione di max-heap e la procedura di inserzione di un elemento si rimanda al testo. La sequenza di max-heap ottenuta con l'inserimento degli elementi nell'ordine indicato è:



mentre invece la sequenza di min-heap è:



Esercizi

Esercizio 1 (9 punti) Realizzare una funzione `intersect(A1,A2,n)` che dati due array di interi `A1` e `A2`, organizzati a min-heap, con capacità n, restituisce un nuovo array `A`, ancora organizzato a min-heap, che contiene l'intersezione dei valori contenuti in `A1` e `A2`. Nel caso gli array contengano più occorrenze dello stesso valore v, l'intersezione mantiene il numero minimo di occorrenze di v (ad es. se `A1` contiene i valori `1,2,2,2` e `A2` contiene i valori `1,1,2,2` allora A conterrà `1,2,2`). Valutarne la complessità.

Soluzione: Per costruire l'intersezione, si procede nel modo seguente. Posso estrarre gli elementi di ciascun min-heap, in ordine crescente, semplicemente con una sequenza di estrazioni di minimo, ciascuna delle quali ha costo logaritmico. Se da $A1$ estraggo v_1 e da A_2 estraggo v_2, se $v_1 = v_2$, inserisco il valore comune nell'intersezione. Se invece $v_1 < v_2$, certamente v_1 non è nell'intersezione, dato che tutti gli elementi rimanenti in $A2$ sono $\geq v_2$, quindi posso scartare v_1, estrarre un nuovo elemento da $A1$ e continuare. Se $v_1 > v_2$ procedo dualmente.

In questo modo ottengo gli elementi dell'intersezione in ordine crescente. L'ultima osservazione è che posso inserirli in questo modo in A, che risulterà un array crescente, che è un caso particolare di min-heap.

Lo pseudocodice può essere il seguente:

```
intersect(A1,A2,n)
   allocate A[1..n]
   i=0                       // ultimo elemento occupato in A
   while (A1.heapsize > 0) and (A2.heapsize > 0)
         v1 = Min(A1)
         v2 = Min(A2)
         if (v1 = v2)
            i++
            A[i]=1
            ExtractMin(A1)
            ExtractMin(A2)
         elseif (v1 < v2)
            ExtractMin(A1)
         else
            ExtractMin(A2)

   A.heapsize=i
   return A
```

La complessità di ogni `ExtractMin` è $O(\log n)$ mentre `Min` ha costo costante. Ol numero di estrazioni è chiaramente limitato da $2n$ e questo porta a dedurre che il costo complessivo è $O(n \log n)$.

Esercizio 2 (8 punti) Per $n > 0$, siano dati due vettori a componenti intere $\mathbf{a}, \mathbf{b} \in \mathbf{Z}^n$. Si consideri la quantità $c(i,j)$, con $0 \leq i \leq j \leq n-1$, definita come segue:

$$c(i,j) = \begin{cases} a_i & \text{se } 0 < i \leq n-1 \text{ e } j = n-1, \\ b_j & \text{se } i = 0 \text{ e } 0 \leq j \leq n-1, \\ c(i-1,j-1) \cdot c(i,j+1) & 0 < i \leq j < n-1. \end{cases}$$

Si vuole calcolare la quantità $m = \min\{c(i,j) : 0 \leq i \leq j \leq n-1\}$.

1. Si fornisca il codice di un algoritmo iterativo bottom-up per il calcolo di m.
2. Si valuti la complessità esatta dell'algoritmo, associando costo unitario ai prodotti tra numeri interi e costo nullo a tutte le altre operazioni.

Soluzione:

1. Date le dipendenze tra gli indici nella ricorrenza, un modo corretto di riempire la tabella è attraverso una scansione in cui calcoliamo gli elementi in ordine crescente di indice di riga e, per ogni riga, in ordine decrescente di indice di colonna. Il codice è il seguente.

```
COMPUTE(a,b)
n <- length(a)
m = +infinito
for i=1 to n-1 do
   C[i,n-1] <- a_i
   m <- MIN(m,C[i,n-1])
for j=0 to n-1 do
   C[0,j] <- b_j
   m <- MIN(m,C[0,j])
for i=1 to n-2 do
   for j=n-2 downto i do
      C[i,j] <- C[i-1,j-1] * C[i,j+1]
      m <- MIN(m,C[i,j])
return m
```

2.

$$T(n) = \sum_{i=1}^{n-2} \sum_{j=i}^{n-2} 1 = \sum_{i=1}^{n-2} n - 1 - i = \sum_{k=1}^{n-2} k = (n-1)(n-2)/2.$$