Algoritmi e Strutture Dati
27 Agosto 2020

Note

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 seguente funzione ricorsiva che dato un array A e due indici $1 \leq p \leq r \leq A.length$ restituisce la somma degli elementi in $A[p,r]$

```
sum(A,p,r)
   if p < r
      q = (p+r)/2
      return sum(A,p,q) + sum(A,q+1,r)
   else
      return A[p]
```

Dimostrare induttivamente che la funzione è corretta, ovvero che $sum(A,p,r)$ ritorna $\Sigma_{i=p}^{r} A[i]$. Inoltre, determinare la ricorrenza che esprime la complessità della funzione e risolverla con il Master Theorem.

Soluzione: Per quanto riguarda la correttezza, procediamo per induzione sul numero n di elementi del sottoarray (ovvero $n = r-p+1$). Se $n = 1$ l'unico elemento del sottoarray è $A[p]$ e quindi correttamente la funzione ritorna $A[p]$. Se $n > 1$, il sottoarray viene diviso in due parti $A[p,q]$ e $A[q+1,r]$ di dimensione minore di n, sui quali viene richiamata la funzione. Per ipotesi induttiva $sum(A,p,q) = \Sigma_{i=p}^{q} A[i]$ e $sum(A,q+1,r) = \Sigma_{i=q+1}^{r} A[i]$. Quindi la funzione ritorna ritorna $\Sigma_{i=p}^{q} A[i] + \Sigma_{i=q+1}^{r} A[i] = \Sigma_{i=p}^{r} A[i]$, come desiderato.

Per quanto riguarda la ricorrenza, indicato ancora con n il numero degli elementi dell'array, nel caso ricorsivo, si effettuano operazioni in tempo costante e quindi si richiama la funzione su due sottoarray la cui dimensione è la metà di quella di partenza. Si ottiene quindi

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

Può essere risolta con il Master Theorem. Utilizzando la notazione del teorema, si deve confrontare la funzione $n^{\log_b a} = n^{\log_2 2} = n^1 = n$ con $f(n) = c$. Chiaramente $f(n) = O(n^{1-\epsilon})$ per ogni $0 < \epsilon < 1$ (infatti $\lim_{n\to\infty} c/n^{1-\epsilon} = 0$). Quindi siamo nel caso 1 del teorema e concludiamo $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$.

Domanda B (5 punti) (i) Si spieghi brevemente come funzionano le tabelle hash con open addressing. (ii) Data una tabella di dimensione m, la funzione di hash $h(k,i) = (k + 2 * i) \mod m$ è appropriata? Si motivi la risposta.

Soluzione: Per il funzionamento della tabelle hash con open addressing si rimanda al testo.

La funzione di hash indicata non è appropriata perché, diversamente da quanto richiesto per una funzione di hash, $h(k,0)$, $h(k,1)$, $\dots$, $h(k,m-1)$ non è, in generale, una permutazione di $0,\dots,m-1$. Ad esempio, se $m = 10$ e $k = 0$, la sequenza sarà $0,2,4,6,8,0,2,4,6,8$.

Esercizi

Esercizio 1 (9 punti) Realizzare un arricchimento degli alberi binari di ricerca che permetta di ottenere per ogni nodo x, in tempo costante, la media dei valori memorizzati nel sottoalbero radicato in x.

Indicare quali campi occorre aggiungere ai nodi. Fornire il codice per la funzione `avg(x)` che restituisce la media delle chiavi nel sottoalbero radicato in x e la procedura di inserimento di un nodo `insert(T,z)`.

Soluzione: L'idea è quella di arricchire la struttura facendo in modo che ogni nodo x, oltre ai campi usuali, abbia il campo x.`sum` che contiene la somma delle chiavi per i nodi del sottoalbero radicato in x e x.`size` che contiene il numero dei nodi in tale sottoalbero.

A questo punto è immediato realizzare la funzione `avg(x)` di tempo costante

```
avg(x)
  if x <> nil
     return x.sum/x.size
  else
     error
```

Invece per l'inserimento, si modifica la procedura standard, che discende l'albero dalla radice fino alla posizione in cui inserire il nuovo nodo, facendo in modo che tutti i nodi attraversati e che quindi conterranno nel sottoalbero il nuovo nodo, abbiano i campi `sum` e `size` aggiornati.

```
Insert(T,z)
   y = nil
   x = T.root

   while (x <> nil)
      y = x
      x.size++
      x.sum = x.sum + z.key
      if (z.key < x.key)
         x = x.left
      else
         x = x.right

   z.p = y
   if (y <> nil)
      if (z.key < y.key)
         y.left =  z
      else
         y.right = z

    z.sum = z.key
    z.size = 1
    z.left = z.right = nil
```

La complessità resta $O(h)$ dove h è l'altezza dell'albero.

Esercizio 2 (9 punti) Lungo una strada ci sono, in vari punti, n parcheggi liberi e n auto. Un posteggiatore ha il compito di parcheggiare tutte le auto, e lo vuole fare minimizzando lo spostamento totale da fare. Formalmente, dati n valori reali $p_1, p_2, \ldots, p_n$ e altri n valori reali $a_1, a_2, \ldots, a_n$, che rappresentano le posizioni lungo la strada rispettivamente di parcheggi e auto, si richiede di assegnare ad ogni auto a_i un parcheggio $p_{h(i)}$ minimizzando la quantità

$$\sum_{i=1}^{n} |a_i - p_{h(i)}|.$$

1. Si consideri il seguente algoritmo greedy. Si individui la coppia (auto, parcheggio) con la minima differenza. Si assegni quell'auto a quel parcheggio. Si ripeta con le auto e i parcheggi restanti fino a quando tutte le auto sono parcheggiate. Dimostrare che questo algoritmo non è corretto, esibendo un controesempio.

2. Si consideri il seguente algoritmo greedy. Si assuma che i valori $p_1, p_2, \ldots, p_n$ e $a_1, a_2, \ldots, a_n$ siano ordinati in modo non decrescente. Si produca l'assegnazione $(a_1, p_1), (a_2, p_2), \ldots, (a_n, p_n)$. Dimostrare la correttezza di questo algoritmo per il caso $n = 2$.

Soluzione:

1. Si consideri il seguente input:

   $$p_1 = 5, p_2 = 10 \quad \text{e} \quad a_1 = 9, a_2 = 14.$$

   L'algoritmo produce l'assegnazione $(a_1, p_2), (a_2, p_1)$, che ha costo $1 + 9 = 10$, mentre l'assegnazione $(a_1, p_1), (a_2, p_2)$ ha costo $4 + 4 = 8$.

2. Ci sono vari casi possibili:

   (a) Caso $a_1 \leq p_1 \leq p_2 \leq a_2$
   - l'assegnazione $(a_1, p_1), (a_2, p_2)$ ha costo $p_1 - a_1 + a_2 - p_2 = (a_2 - a_1) - (p_2 - p_1)$
   - l'assegnazione $(a_1, p_2), (a_2, p_1)$ ha costo $p_2 - a_1 + a_2 - p_1 = (a_2 - a_1) + (p_2 - p_1)$; siccome $p_2 - p_1 \geq 0$, questa assegnazione ha costo non inferiore rispetto alla precedente

   (b) Caso $a_1 \leq p_1 \leq a_2 \leq p_2$
   - l'assegnazione $(a_1, p_1), (a_2, p_2)$ ha costo $p_1 - a_1 + p_2 - a_2 = (p_2 - a_1) - (a_2 - p_1)$
   - l'assegnazione $(a_1, p_2), (a_2, p_1)$ ha costo $p_2 - a_1 + a_2 - p_1 = (p_2 - a_1) + (a_2 - p_1)$; siccome $a_2 - p_1 \geq 0$, questa assegnazione ha costo non inferiore rispetto alla precedente

   (c) Caso $a_1 \leq a_2 \leq p_1 \leq p_2$
   - l'assegnazione $(a_1, p_1), (a_2, p_2)$ ha costo $p_1 - a_1 + p_2 - a_2 = (p_2 - a_1) + (p_1 - a_2)$
   - l'assegnazione $(a_1, p_2), (a_2, p_1)$ ha costo $p_2 - a_1 + p_1 - a_2 = (p_2 - a_1) + (p_1 - a_2)$, uguale a quello precedente

   Tutti gli altri casi sono simmetrici e si dimostrano nella stessa maniera.