Corso di laurea in Matematica
Insegnamento di Informatica generale
Lezioni in modalità mista o a distanza

Soluzioni Esercizi vari: Ricorrenza, Alberi, Ordinamenti Lineari

Giancarlo Bongiovanni
Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Queste dispense sono state realizzate sulla base delle slide preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Corso di laurea in Matematica
Insegnamento di Informatica generale
Lezioni in modalità mista o a distanza

Esercizi su Equazioni di Ricorrenza,

Giancarlo Bongiovanni
Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Queste dispense sono state realizzate sulla base delle slide
preparate da T. Calamoneri e G. Bongiovanni
per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Esercizio 8 – dallo pseudocodice all'equazione

```
Funzione Fun (n: intero)
1)        if n ≤ 1 return 1
2)        k ← 1
3)        while k ≤ n do
3)                k ← 2*k
4)        return (Fun(n DIV 2)))
```

ATTENZIONE!

Nota: DIV fa la divisione intera

1) Θ(1) – caso base
2) Θ(1) – parte al di fuori della chiamata ricorsiva
3) **Θ(log n)** – iterazione, parte al di fuori della chiamata ricorsiva
4) T(n/2) – una chiamata ricorsiva

Equazione:

$T(n) = T(n/2) + \Theta(\log n)$

$T(1) = \Theta(1)$

Soluzione (con metodo iterativo):

$$T(n) = \Theta(1)+\sum_{i=0}^{logn-1} \Theta(\log \frac{n}{2^i}) = \Theta(1) + \sum_{i=0}^{logn-1} \Theta(\log n - \log 2^i) =$$

$$\Theta(\log^2 n - \sum_{i=0}^{logn-1} i) = \Theta(\log^2 n - \frac{\log n(logn-1)}{2}) = \Theta(\log^2 n)$$

Esercizio 9.1 – soluzione dell'equazione

Equazione:

$T(n) = T(n/2) + T(n/4) + \Theta(n)$

$T(1) = \Theta(1)$

Prima idea: analogamente a quanto fatto per Fibonacci, possiamo maggiorare e minorare:

- $T(n) \leq 2T(n/2) + \Theta(n)$
- $T(n) \geq 2T(n/4) + \Theta(n)$

Questo ci consente di scrivere: (ad esempio usando il **metodo principale** o quello **iterativo**)

- $T(n) = O(n \log n)$
- $T(n) = \Omega(n)$

Però il primo risultato possiamo cercare di migliorarlo, col metodo di sostituzione.

Esercizio 9.1 – soluzione dell'equazione

Equazione:

$T(n) = T(n/2) + T(n/4) + cn$

$T(1) = d$

Ipotizziamo $T(n) = O(n)$, ossia $T(n) \leq kn$

Passo base:

$T(1) \leq k$, vera per $k \geq d$

Passo induttivo:

$T(n) \leq k\, n/2 + k\, n/4 + cn = \frac{3}{4} kn + cn = kn - \frac{1}{4} kn + cn \leq kn$

L'ultima disuguaglianza è vera se $cn - \frac{1}{4} kn \leq 0$, ossia $k \geq 4c$

Dunque

$T(n) = O(n)$ e quindi $T(n) = \Theta(n)$

Corso di laurea in Matematica
Insegnamento di Informatica generale
Lezioni in modalità mista o a distanza

Soluzioni degli esercizi proposti su Alberi

Giancarlo Bongiovanni
Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Queste dispense sono state realizzate sulla base delle slide preparate da T. Calamoneri e G. Bongiovanni
per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Esercizio 1 (1)

Dire quant'è la massima lunghezza di un vettore che è necessario allocare per poter memorizzare sempre un albero **di n nodi** con la rappresentazione posizionale

- Un albero ***di n nodi*** può avere altezza ***(n-1)***, nel caso sia un albero degenere (nel quale ogni nodo ha un solo figlio);
- come abbiamo visto, per memorizzare un albero di ***altezza h*** con la notazione posizionale è necessario predisporre un vettore di $2^{h+1} - 1$ elementi;
- dunque, è necessario un vettore di $2^{n} - 1$ elementi.

Esercizio 2 (1)

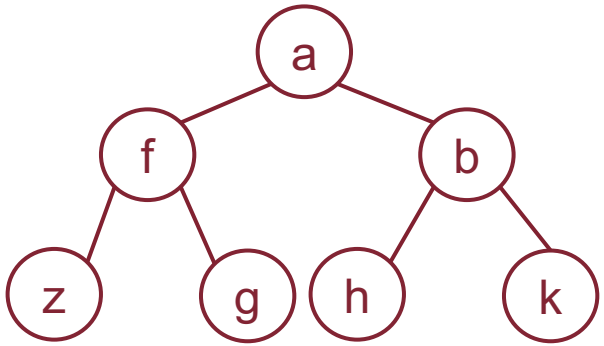
Dato un albero completo e non vuoto memorizzato mediante notazione posizionale, crearne uno identico memorizzato tramite vettore di padri. Calcolare il costo computazionale

IDEE

- Possiamo sfruttare il fatto che con la notazione posizionale sappiamo dove stanno i figli di ciascun nodo, quindi possiamo calcolare direttamente gli indici per il vettore dei padri.
- Basta un semplice ciclo che scorre contemporaneamente su tutti i vettori.
- Se però l'albero non è completo la gestione degli indici si complica un poco perché nel vettore dei padri non si prevede di norma di lasciare posizioni non occupate (lasciato come **esercizio :** ad esempio, ispirandosi a Counting sort, si potrebbe memorizzare quante posizioni vuote ci sono fino a *i*.).

Esercizio 2 (2)

Esempio



Notazione posizionale

1	2	3	4	5	6	7
a	f	b	z	g	h	k

Vettore A

Vettore dei padri

a	f	b	z	g	h	k
NULL	1	1	2	2	3	3

Vettore delle chiavi (C)

Vettore dei padri (P)

Esercizio 2 (3)

```
Funzione Converti (A,C,P:vettore;n:intero)
   C[1] ← A[1]
   P[1] ← NULL
   for (i = 2 to n) do
      C[i] ← A[i]
      P[i] ← ⌊i/2⌋
   return
```

Costo computazionale: $\Theta(n)$

Esercizio 3 (1)

Dato un albero binario non vuoto memorizzato tramite vettore dei padri, crearne uno identico memorizzato tramite notazione posizionale. Calcolare il costo computazionale

IDEE

- Prima di tutto bisogna identificare la radice, che è l'unico elemento con NULL nel vettore P (funzione **`Trovaradice()`**).
- Poi, per ogni nodo che viene sistemato, ossia copiato in posizione ***i***, bisogna identificare i suoi figli (con la funzione **`Trova_figli()`**) e copiarli nelle posizioni ***2i*** e ***2i+1***. Useremo la ricorsione per questo.
- Supponiamo che la funzione **`Trova_figli()`** restituisca un record a due campi **`{left,right}`** con gli indici di ciascuno dei due figli (oppure NULL per ogni figlio che non esiste). **Si assume** per convenzione che **il primo indice trovato** da **`Trova_figli()`** sia quello del figlio sinistro.

Esercizio 3 (2)

- La funzione **`Trova_radice(P:vettore)`** effettua una scansione del vettore P dei padri e restituisce l'indice dell'unico elemento NULL, che è quello relativo alla radice.
- La funzione **`Trova_figli(P:vettore;i:intero)`** cerca le posizioni dei figli del nodo collocato in posizione i. Effettua una scansione del vettore P da sinistra a destra:
  - Il primo indice h trovato (se esiste) per cui P[h] = i viene assegnato a **`left`**; se non viene trovato si assegna NULL a left;
  - Il secondo indice k trovato (se esiste) per cui P[k] = i viene assegnato a **`right`**; se non viene trovato si assegna NULL a right;
- La funzione **`SistemaNodo()`** che ora vedremo è ricorsiva:
  - Copia un nodo dal vettore delle chiavi (vettore dei padri, sottovettore B) al vettore posizionale (A);
  - Chiama ricorsivamente se stessa sui due figli del nodo appena sistemato.

Esercizio 3 (3)

Nel **`main()`** si avrà:

```
IndiceRadice ← Trova_radice(P)
SistemaNodo(A,C,P,1,Indiceradice)

Funzione SistemaNodo (A,C,P:vettore;indA,indC:intero)
   if (indA ≤ n) do
      A[indA] ← C[indC]
      {left,right} ← Trova_Figli(P,indB)
      if (left≠NULL) SistemaNodo(A,C,P,2*indA,left)
      if (right≠NULL) SistemaNodo(A,C,P,2*indA+1,right)
   return
```

- Vettore A: vettore risultato, con l'albero nella notazione posizionale
- Vettore C: chiavi dei nodi nella memorizzazione con vettore dei padri
- Vettore P: padri dei nodi nella memorizzazione con vettore dei padri
- indA: indice che si muove sul vettore A (memorizzazione posizionale)
- indB: indice che si muove sui vettori C e P (memorizzazione con vettore dei padri)

Esercizio 3 (4)

Il costo computazionale è dato dalla somma dei costi di **`Trova_radice()`** e **`SistemaNodo()`**.

1. **`Trova_radice()`** ha banalmente costo $\Theta(n)$
2. **`SistemaNodo()`** è di fatto una visita, nella quale *però il lavoro su ciascun nodo ha costo* $\Theta(n)$ perché bisogna sempre scandire l'intero vettore P per trovare i due figli di un nodo. Poiché la visita fa tale lavoro (sempre sull'intero vettore P) per ciascun nodo, il suo costo è $T(n) = \Theta(n^2)$

Quindi il costo computazionale è

$$T(n) = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

Esercizio 6

Scrivere lo pseudocodice della funzione **`Trova_Figli()`**

`P` è il sottovettore del vettore dei padri in cui sono memorizzati gli indici dei padri; **`ind`** è l'indice del nodo di cui si vogliono trovare i figli.

```
Funzione Trova_Figli(P: vettore; ind: intero)
   num_figli ← 0 /* serve per sapere se abbiamo già
                  * trovato il figlio sinistro */
   left ← NULL
   right ← NULL
   for (i=1 to n) do
      if (P[i] = ind)
         num_figli ← num_figli+1
         if (num_figli = 1) left ← i
                            else right ← i
   return {left,right}
```

Costo computazionale: $\Theta(n)$

Esercizio 7 (1)

In quell'esercizio, se usassimo un vettore ausiliario in cui memorizzare in fase di **pre-processing** i figli di ciascun nodo, come diventerebbe lo pseudo-codice? ed il costo computazionale?

Supponiamo di aver creato, con un pre-processing, due vettori **`LEFT[]`** e **`RIGHT[]`** in cui sono memorizzati gli indici dei due figli di ciascun nodo. Il codice diventerebbe:

```
Funzione Trova_Figli(LEFT,RIGHT: vettore; ind: intero)
   return {LEFT[ind],RIGHT[ind]}
```

Costo computazionale: $\Theta(1)$

NOTA: Poiché col pre-processing la funzione **`Trova_Figli()`** viene ad avere costo unitario, se la funzione **`SistemaNodo()`** dell'es. 3 utilizza questa nuova versione il suo costo scende da $\Theta(n^2)$ a $\Theta(n)$.

Esercizio 7 (2)

Il pre-processing può essere realizzato con costo lineare in n.

Reminder: **P[i]=k** significa che il nodo di indice k è padre del nodo di indice i.

```
Funzione PreProcess(P, LEFT,RIGHT: vettore)
   for (i= 1 to n) do /*inizializzazione */
      LEFT[i] ← NULL
      RIGHT[i] ← NULL
   for (i= 1 to n)
      if (P[i]≠ NULL)     //la radice non ha padre
         if (LEFT[P[i]] = NULL) LEFT[P[i]] ← i
         else RIGHT[P[i]] ← i
```

Costo computazionale: $\Theta(n)$.

Esercizio 4 (1)

Scrivere lo pseudocodice ITERATIVO della visita in preordine

IDEE

- Dobbiamo realizzarla con l'ausilio di una struttura d'appoggio, che in questo caso deve essere una **pila**.
- Nella pila inseriremo prima il figlio destro e poi quello sinistro, in modo che l'estrazione operi prima sul figlio sinistro e poi su quello destro.
- Supponiamo che, come per la coda utilizzata nella visita per livelli, l'implementazione della pila sia tale per cui si inseriscono ed estraggono puntatori a nodi dell'albero.

Esercizio 4 (2)

```
Funzione Visita_preorder_iterativa (t: albero)
   if (t = NULL) return
   top ← NULL /* crea pila vuota */
   Push(top, t)
   while (!PilaVuota(top))
      p ← Pop(top)
      visita nodo e operazioni conseguenti
      if(right[p] ≠ NULL) Push(top, right[p])
      if(left[p] ≠ NULL) Push(top, left[p])
   return
```

Il costo computazionale è $\Theta(n)$ perché per ognuno degli n nodi si effettuano:

- una Push, costo $\Theta(1)$;
- una Pop, costo $\Theta(1)$;
- un numero costante di operazioni elementari, $\Theta(1)$.

Esercizio 5

Calcolare il costo computazionale delle visite quando l'albero venga memorizzato tramite rappresentazione posizionale

Le visite si adattano facilmente alla rappresentazione posizionale, ad esempio la visita in ordine anticipato diviene:

```
Funzione Pre_Order_Posiz(P,i,n)
   visita il nodo corrispondente all'elemento P[i]
   if (2*i<=n) Pre_Order_Posiz(P,2*i,n)      //left
   if (2*i+1<=n) Pre_Order_Posiz(P, 2*i+1,n) //right
   return
```

La chiamata iniziale è: `Pre_Order_Posiz(P,1,n)`

Il costo computazionale resta $\Theta(n)$ come per la visita

Corso di laurea in Matematica
Insegnamento di Informatica generale
Lezioni in modalità mista o a distanza

Soluzioni degli esercizi proposti su Counting sort e Bucket sort

Giancarlo Bongiovanni
Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Queste dispense sono state realizzate sulla base delle slide preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Esercizio 1

Mostrare che il Counting sort è un algoritmo di ordinamento stabile

- Per il Counting sort di base la domanda non è pertinente, in quanto l'algoritmo di fatto non copia i valori del vettore originale su quello ordinato ma ricopia (più volte se necessario) un medesimo valore, quello contenuto nel vettore C, che non è attribuibile ad alcuno dei valori originari.
- Viceversa, la versione che gestisce i dati satellite è un algoritmo di ordinamento stabile:
  - elementi con uguale valore vengono scanditi, nel vettore da ordinare, da destra a sinistra e vengono ricopiati nel vettore ordinato, assieme ai loro dati satellite, in posizioni contigue da destra verso sinistra;
  - quindi l'ordinamento relativo fra elementi di uguale valore è preservato.

Esercizio 2

Qual è il tempo di esecuzione del Bucket sort nel caso peggiore? quale semplice modifica dell'algoritmo consente di conservare tempo medio lineare e costo Θ(n log n) nel caso peggiore?

- Il caso peggiore si verifica quando tutti gli n elementi cadono nello stesso bucket e risultano nel bucket in ordine decrescente: in tal caso l'ordinamento con Insertion sort costa $\Theta(n^2)$;
- basta ordinare i bucket con un algoritmo efficiente anche nel caso peggiore (Mergesort oppure Heapsort): così facendo il costo nel caso peggiore scende a Θ(n log n).
- Un tempo medio lineare (atteso) si verifica per le stesse ragioni già discusse per il Bucket sort standard: ci si aspetta un numero costante di elementi in ciascun bucket.

Esercizio 3 (1)

Il bucket sort può essere modificato in modo che l'ordinamento all'interno delle liste sia eseguito tramite counting sort. Affinché il costo dell'algoritmo sia lineare nel caso in cui tutti gli elementi da ordinare finiscano nello stesso bucket, quale ipotesi bisogna fare su k?

- L'i-esimo bucket contiene valori compresi fra (i-1)*k/n (escluso) e i*k/n (compreso). In altre parole, ogni bucket contiene valori in un intervallo di ampiezza k/n.
- Quindi se l'ampiezza k/n ≤ n, allora l'intervallo dei valori di ciascun bucket può essere mappato in un intervallo non più grande di [1..n] e quindi si può ordinare il bucket con Counting sort in tempo lineare in n anche nel caso in cui tutti gli n elementi siano contenuti nello stesso bucket.
- Il che corrisponde a dire che k ≤ n^2.

Esercizio 3 (2)

NOTE

- Così però **il costo di ogni singolo ordinamento** sarà comunque $\Theta(n)$, anche se ogni bucket contiene un numero costante di elementi, perché il Counting sort ha un costo che è $\Theta(n+h)$ dove n è il numero di elementi e h è l'ampiezza del range di valori (il vettore C si scandisce in ogni caso per intero).
- Quindi se $h = O(n)$, come nella soluzione proposta, il costo totale dell'ordinamento di tutti i bucket sale a $O(n^2)$ quando gli elementi da ordinare sono ben distribuiti nei bucket.
- Per avere un costo totale lineare anche in questo caso, si deve porre come condizione che $k/n = \Theta(1)$, ossia $k = \Theta(n)$.