SAPIENZA
UNIVERSITÀ DI ROMA

Corso di laurea in Matematica

Esercizi svolti relativi all'insegnamento di

Informatica generale

Canale 1: Prof. Tiziana Calamoneri
Canale 2: Prof. Giancarlo Bongiovanni

Questi esercizi svolti si riferiscono al corso di Informatica Generale per il Corso di Laurea in Matematica, e vanno associati alle relative dispense, di cui vengono richiamati i numeri dei capitoli.

Essi rispecchiano piuttosto fedelmente il livello di dettaglio che viene richiesto durante l'esame scritto, ma allo stesso tempo, vogliono condurre lo studente verso il corretto modo di ragionare, per cui possono sembrare a volte prolissi.

Sono benvenuti esercizi svolti dagli studenti, che verranno uniformati a questi ed annessi al presente documento.

Gli studenti interessati a contribuire possono mandare il loro lavoro a

calamo@di.uniroma1.it ed a bongiovanni@di.uniroma1.it

Esercizio 1.

Riferimento ai capitoli: 2) Complessità asintotica

Si consideri il seguente pseudocodice:

```
1.    somma ← 0
2.    for I = 1 to n do
3.       somma ← somma + i
4.    return somma
```

e si calcoli la complessità asintotica nel caso peggiore.

Si dica, inoltre, se l'algoritmo descritto dallo pseudocodice dato è efficiente per il problema che esso si prefigge di risolvere.

Soluzione.

Come primo passo, dobbiamo riconoscere quale sia il parametro che guida la crescita della complessità. In casi come questo, in cui compare un elenco di n elementi, è immediato definire n come parametro.

Quindi la complessità sarà una funzione $T(n)$.

Secondo il modello RAM, una riga i può richiedere un tempo diverso da un'altra, ma ognuna impiega un tempo costante ci. Poiché la riga 2. è un ciclo che viene ripetuto n volte, si ha:

$$T(n) = c1 + c4 + n(c2 + c3) = an + b$$

per opportuni valori di a e b. Ne consegue che $T(n) = \Theta(n)$.

E' immediato riconoscere che lo pseudocodice descrive l'algoritmo di somma dei primi n interi. Tuttavia, esso non è l'algoritmo più efficiente.

Si consideri, infatti, il seguente pseudocodice:

```
1.    return n*(n+1)/2
```

la cui complessità è, ovviamente $T(n) = c1 = \Theta(1)$.

Esercizio tipo esame 2.

Riferimento ai capitoli: 5. Equazioni di ricorrenza

Si risolva la seguente equazione di ricorrenza con due metodi diversi, per ciascuno dettagliando il procedimento usato:

$T(n) = T(n/2) + \Theta(n^2)$ *se* $n > 1$

$T(1) = 0$

tenendo conto che n è una potenza di 2.

Soluzione.

Utilizziamo dapprima il metodo principale, che è il più rapido, e poi verifichiamo la soluzione trovata con un altro metodo.

Le costanti a e b del teorema principale valgono qui 1 e 2, rispettivamente, quindi $\log_b a = 0$ ed $n^{\log_b a} = 1$, per cui, ponendo $\varepsilon = 2$, siamo nel caso 3. del teorema, se vale che $af(n/b) \leq cf(n)$.

Poiché $f(n)$ è espressa tramite notazione asintotica, per verificare la disuguaglianza, dobbiamo eliminare tale notazione, e porre ad esempio $f(n) = \Theta(n) = hn^2 + k$.

Non è restrittivo supporre $h,k \geq 0$ visto che ci troviamo in presenza di una complessità e quindi non è sensato avere tempi di esecuzione negativi.

Ci chiediamo se esista una costante c tale che $h(n/2)^2 + k \leq c(hn^2 + k)$.

Risolvendo otteniamo: $hn^2(1/4 - c) + k(1 - c) \leq 0$ che è verificata ad esempio per $c = 1$.

Ne possiamo dedurre che $T(n) = \Theta(n^2)$.

Come secondo metodo usiamo quello iterativo.

Dall'equazione di ricorrenza deduciamo che:

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right)$$

$$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + \Theta\left(\left(\frac{n}{2^2}\right)^2\right)$$

e così via. Sostituendo:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n^2) =$$

$$= T\left(\frac{n}{2^2}\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta(n^2) =$$

$$= T\left(\frac{n}{2^3}\right) + \Theta\left(\left(\frac{n}{2^2}\right)^2\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta((n)^2) =$$

$$\dots$$

$$= T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} \Theta\left(\left(\frac{n}{2^i}\right)^2\right)$$

Continuiamo ad iterare fino al raggiungimento del caso base, cioè fino a quando $n/2^k = 1$, il che avviene se e solo se $k = \log n$.

L'equazione diventa così:

$$T(n) = T(1) + \sum_{i=0}^{\log n-1} \Theta\left(\left(\frac{n}{2^i}\right)^2\right) = \Theta(1) + n^2 \sum_{i=0}^{\log n-1} \Theta\left(\frac{1}{4^i}\right)$$

Ricordando che

$$\sum_{i=0}^{b} a^i = \frac{a^{b+1} - 1}{a - 1}$$

otteniamo infine:

$$T(n) = \Theta(1) + n^2\, \Theta\left(\frac{1-\left(\frac{1}{4}\right)^{\log n}}{1-\frac{1}{4}}\right) = \Theta(n^2).$$

Anche se forse superfluo, concludiamo con l'osservare che i due metodi ***devono*** portare allo stesso risultato o, quanto meno a risultati compatibili, altrimenti bisogna concludere che si è fatto un errore.

Esercizio tipo esame 3.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza

Dato un array *A* di *n* interi progettare un algoritmo che, usando il paradigma del "divide et impera" restituisce il massimo ed il minimo di *A* in tempo *O(n)*. Verificare tale complessità tramite l'impostazione e la risoluzione di un'equazione di ricorrenza.

Soluzione.

Cominciamo col fare qualche osservazione: il metodo del "divide et impera", incontrato spesso in molti algoritmi noti (ad esempio nel Merge Sort) si basa sull'idea di ridurre il problema dato, di dimensione *n*, a due o più problemi di dimensione inferiore; per sua natura è quindi un metodo basato sulla ricorsione. L'idea di produrre un algoritmo ricorsivo dovrebbe essere anche suggerita dal fatto che si chiede di calcolare la complessità tramite equazione di ricorrenza.

L'unica scelta che si deve fare è dunque quella di decidere come suddividere il problema in due sottoproblemi. Le uniche possibilità sensate sono le seguenti:

- calcolare ricorsivamente il massimo ed il minimo nei due sottovettori destro e sinistro di dimensione *n/2*, confrontare poi i due massimi ed i due minimi dando in output il massimo e minimo globali;
- calcolare ricorsivamente il massimo ed il minimo del sottovettore di dimensione *n-1* ottenuto eliminando il primo (o l'ultimo) elemento, confrontare poi il massimo ed il minimo ottenuti con l'elemento eliminato dando in output il massimo e minimo globali.

Per comprendere quale dei due approcci sia migliore, proviamo a calcolarne la complessità.

Per quanto riguarda il primo metodo, il tempo di esecuzione su un input di *n* elementi *T(n)* è pari al tempo di esecuzione dello stesso algoritmo su ciascuno dei due sottovettori di *n/2* elementi, più il tempo necessario per confrontare i due massimi ed i due minimi; il caso base si ha quando i sottovettori sono di dimensione 1, per cui il massimo ed il minimo coincidono, e l'unica operazione da fare è quella di confrontare tra loro massimi e minimi per dare in output i valori globali. L'equazione di ricorrenza è dunque:

$$T(n) = 2T(n/2) + \Theta(1)$$

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

Risolvendo questa equazione con il metodo iterativo (ma si può fare anche con il metodo dell'albero), si ottiene:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1) =$$

$$= 2\left(2T\left(\frac{n}{2^2}\right) + \Theta(1)\right) + \Theta(1) =$$

$$= 2^2\left(2T\left(\frac{n}{2^3}\right) + \Theta(1)\right) + 2\Theta(1) + \Theta(1) =$$

$$\dots$$

$$= 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2^i \Theta(1)$$

Procediamo fino a quando $n/2^k$ non sia uguale ad 1 e cioè fino a quando $k = \log n$. In tal caso, sostituendo nell'espressione di $T(n)$ e sfruttando il caso base si ha:

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

Studiamo ora il secondo approccio: il tempo di esecuzione dell'algoritmo $T(n)$ su un input di n elementi è pari al tempo di esecuzione dello stesso algoritmo su $n - 1$ elementi più il tempo per confrontare l'elemento rimasto con il massimo ed il minimo trovati; anche qui il passo base si ha quando $n = 1$. L'equazione di ricorrenza allora diventa:

$T(n) = T(n - 1) + \Theta(1)$

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

Anche questa equazione si può risolvere con il metodo iterativo ottenendo:

$T(n) = T(n - 1) + \Theta(1) = T(n - 2) + \Theta(1) + \Theta(1) = \dots = T(n - k) + k\Theta(1).$

Procediamo fino a quando $n - k = 1$ cioè fino a quando $k = n - 1$ e sostituiamo nell'equazione:

$$T(n) = T(1) + (n - 1)\,\Theta(1) = \Theta(n)$$

se si tiene conto del caso base.

Deduciamo dai precedenti ragionamenti che entrambi i metodi sono ugualmente corretti ed efficienti.

Esercizio tipo esame 4.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza; 6. Il problema dell'ordinamento

L'algoritmo Insertion-Sort può essere espresso come una procedura ricorsiva nel modo seguente:

- per ordinare $A[1 \dots n]$, si ordina in modo ricorsivo $A[1 \dots n-1]$ e poi si inserisce $A[n]$ nell'array ordinato $A[1 \dots n-1]$.

Si scriva una ricorrenza per il tempo di esecuzione di questa versione ricorsiva di insertion sort.

Soluzione.

Denotiamo con $T(n)$ il tempo impiegato nel caso peggiore dalla versione ricorsiva dell'algoritmo Insertion-Sort. Abbiamo che $T(n)$ è pari al tempo necessario per ordinare ricorsivamente un array di $n-1$ elementi (che è uguale a $T(n-1)$) più il tempo per inserire $A[n]$ (che nel caso peggiore è $\Theta(n)$).

Pertanto la ricorrenza diviene:

$$T(n) = T(n-1) + \Theta(n).$$

Per quanto riguarda il caso base, esso si ha ovviamente per $n = 1$ ed, in tal caso, il tempo di esecuzione è $\Theta(1)$.

La soluzione dell'equazione di ricorrenza può essere ottenuta con uno qualunque dei metodi studiati. Qui utilizziamo il metodo iterativo:

$$T(n) = T(n-1) + \Theta(n) =$$

$$= (T(n-2) + \Theta(n-1)) + \Theta(n) =$$

$$= ((T(n-3) + \Theta(n-2)) + \Theta(n-1)) + \Theta(n) =$$

$$= \dots =$$

$$= T(n-k) + \sum_{i=0}^{k-1} \Theta(n-i)$$

Raggiungiamo il caso base quando $n - k = 1$, cioè quando $k = n - 1$. In tal caso, l'equazione diventa:

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

Si consiglia il lettore di risolvere l'equazione di riscorrenza con altri metodi.

Esercizio tipo esame 5.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza; 6. Il problema dell'ordinamento

Dato un vettore di *n* numeri reali non nulli, progettare un algoritmo efficiente che posizioni tutti gli elementi negativi prima di tutti gli elementi positivi.

Dell'algoritmo progettato si dia:

a) la descrizione a parole
b) lo pseudocodice
c) la complessità computazionale.

Soluzione.

Sia *A* il vettore dato in input. Lo scopo dell'algoritmo che dobbiamo progettare è quello di separare i numeri positivi dai negativi, senza introdurre alcun ordine particolare.

Una funzione simile a quella che partiziona il vettore rispetto ad una soglia nel Quicksort potrebbe fare al caso nostro, con l'unica accortezza che la soglia qui deve essere 0. Non avremo difficoltà a gestire elementi del vettore nulli perché il testo ci assicura che non ce ne sono.

La descrizione a parole è dunque la seguente:

- scorri il vettore *A* da sinistra a destra con un indice *i* e da destra a sinistra con un indice *j*; ogni volta che *i* indica un elemento positivo e *j* un elemento negativo esegui uno scambio; prosegui fino a quando il vettore non sia stato completamente visionato.

Lo pseudocodice, del tutto analogo alla funzione Partiziona del libro di testo, è il seguente:

```
Funzione PartizionaPosNeg (A: vettore; i, j: intero)
1.  i ← 1
2.  j ← n
3.  while (i < j)
4.     while (A[j] > 0) and (i ≤ j)
5.        j ← j − 1
6.     while ((A[i] < 0)and(i ≤ j)
7.        i ← i + 1
8.     if (i < j)
9.        scambia A[i] e A[j]
```

Infine, la complessità dell'algoritmo è esattamente la stessa della funzione a cui ci siamo ispirati, ma ricalcoliamola qui per completezza:

- le linee 1 e 2 hanno complessità *O(1)*;
- i tre `while` alle linee 3, 4 e 6, complessivamente, fanno in modo che gli elementi del vettore vengano tutti esaminati esattamente una volta e le istruzioni all'interno del ciclo impiegano tempo costante.

Il tempo di esecuzione è indipendente dal vettore in input; se ne conclude che la complessità è *Θ(n)*.

La complessità spaziale è anch'essa lineare in *n*, visto che l'algoritmo utilizza il vettore di input *A* di lunghezza *n* e due contatori.

Questo esercizio si può anche risolvere osservando che separare gli elementi positivi dai negativi equivale ad ordinare *n* numeri interi nel range [0,1]; si può quindi applicare una modifica dell'algoritmo di counting sort, come segue.

Sia *A* l'array di input e *B* l'array di output, entrambi di lunghezza *n*.

```
Funzione SeparaPosNeg2(A, B)
   1.  pos ← n;
   2.  neg ← 1;
   3.  FOR i = 1 TO n DO
   4.     IF A[i]>0 THEN
   5.        B[pos] ← A[i];
   6.        pos ← pos-1;
   7.     ELSE
   8.        B[neg] ← A[i];
   9.        neg ← neg+1;
```

Si osservi che non abbiamo bisogno del terzo array *C*, essendo i dati semplicemente degli interi (senza dati satellite) e non essendoci richiesto di mantenere l'ordinamento dato in input. Ed infatti l'algoritmo proposto posiziona gli elementi positivi nell'ordine dato e gli elementi negativi nell'ordine inverso.

La complessità temporale di questo algoritmo è, ovviamente, lineare in *n*, visto che il suo pseudocodice è costituito da un ciclo FOR che esegue operazioni costanti ad ogni iterazione. La complessità spaziale è anch'essa lineare in *n*, visto che l'lagoritmo utilizza, oltre al vettore di input *A*, un vettore ausiliario *B* di lunghezza *n* e tre contatori.

Concludiamo osservando che anche un qualunque algoritmo di ordinamento avrebbe assolto allo scopo, facendo anzi più del richiesto, ma richiedendo anche una complessità di *Ω(n log n)*. Se ne deduce che una tale soluzione è senz'altro da scartare.

Esercizio tipo esame 6.

Riferimento ai capitoli: 6. Il problema dell'ordinamento

Dati due insiemi di interi $A = \{a_1, a_2, \ldots, a_n\}$ e $B = \{b_1, b_2, \ldots, b_m\}$, sia C la loro intersezione, ovvero l'insieme degli elementi contenuti sia in A che in B.

Si discuta, confrontandola, la complessità computazionale dei seguenti due approcci al problema:

- applicare un algoritmo "naive" che confronta elemento per elemento;
- applicare un algoritmo che prima ordina gli elementi di A e B.

Soluzione.

La soluzione "naive" è la prima e più semplice che viene in mente: per ogni elemento di A (cioè `FOR` un certo indice i che va da 1 ad n) si scorre B (cioè `FOR` un certo altro indice j che va da 1 ad m) per vedere se l'elemento a_i è presente o no in B. Ne segue che la complessità di questo approccio è $O(nm)$. Scriviamo O e non Θ perché, mentre il ciclo `FOR i` viene eseguito completamente, il ciclo `FOR j` si può interrompere appena l'elemento a_i è uguale all'elemento b_j. Pertanto, nel caso migliore, in cui tutti gli elementi di A sono uguali tra loro ed al primo elemento di B la complessità è $\Theta(n)$, ma nel caso peggiore, in cui l'intersezione è vuota, la complessità è $\Theta(nm)$.

Tra l'altro, l'analisi del caso migliore fa venire in mente che una trattazione a parte meriterebbe il caso della duplicazione degli elementi; esso si risolve facilmente ricordando che un insieme, per definizione, non ammette ripetizioni al suo interno, e quindi dobbiamo considerare gli a_i tutti distinti tra loro, così come i b_i. Ne discende che il caso migliore presentato non è ammissibile, ma questo non cambia la complessità del caso peggiore.

Consideriamo ora il secondo approccio.

Si ordinino gli insiemi A e B in modo crescente, e poi si confrontino i due vettori come segue:

- si pongano un indice i all'inizio di A e un indice j all'inizio di B;
- si confronti $A[i]$ con $B[j]$
    - se essi sono uguali l'elemento si inserisce in C e si incrementano sia i che j,
    - se $A[i]$ è minore si incrementa i,
    - se $B[j]$ è minore si incrementa j;
- si ripeta dal confronto finché non si raggiunga la fine di uno dei due vettori.

La complessità *T(n, m)* di questo approccio deve tener conto della complessità dell'ordinamento e della complessità del confronto, e pertanto è pari a

$$\Theta(n \log n) + \Theta(m \log m) + \Theta(max(n, m))$$

Se assumiamo, senza perdere di generalità, che $n \geq m$ si ha che

$$T(n, m) = \Theta(n \log n) + \Theta(n) = \Theta(n \log n)$$

Se possiamo fare alcune ipotesi sugli elementi di A e di B, in modo da poter applicare uno degli algoritmi di ordinamento lineari, la complessità si abbassa ulteriormente a $\Theta(n)$.

Esercizio tipo esame 7.

Riferimento ai capitoli: 4. Ricorsione; 5. Equazioni di ricorrenza; 6. Il problema dell'ordinamento

Si consideri lo pseudocodice del seguente algoritmo di ordinamento.

La funzione viene richiamata la prima volta con *i=1* e *j=n.*

```
Funzione UnAltroSort (A[], i, j)
1. if (A[i]>A[j])
2.    scambia(A[i], A[j])
3. if (i+1 ≥ j)
4.    return
5. k ←⌊ (j-i+1)/3⌋
6. UnAltroSort(A, i, j-k)  /* si ordinano i primi 2/3 del vettore
7. UnAltroSort(A, i+k, j) /* si ordinano gli ultimi 2/3 del vettore
8. UnAltroSort(A, i, j-k)  /* si ordinano di nuovo i primi 2/3 del vettore
```

1. si scriva l'equazione di ricorrenza che descrive la complessità computazionale della funzione nel caso peggiore, dandone giustificazione;
2. si risolva l'equazione trovata usando il teorema principale ed un altro metodo e, per ciascuno, si mostri il procedimento usato;
3. si confronti il tempo di esecuzione del caso peggiore di `UnAltroSort` con quello dell'InsertionSort e del MergeSort, facendo le opportune considerazioni sull'efficienza dell'algoritmo proposto.

Soluzione.

1. Innanzi tutto dobbiamo individuare il parametro dell'equazione di ricorrenza. In questo ci aiutano i commenti, i quali ci suggeriscono che le chiamate ricorsive vengono effettuate su un vettore di lunghezza pari ai 2/3 del vettore originario; pertanto è naturale assumere che il parametro sia proprio la lunghezza del vettore; usando i nomi delle variabili presenti nella funzione, tale parametro si può esprimere come $n=j-i+1$. Questo è in accordo con le linee 6, 7 ed 8, dove le lunghezze dei tre sottovettori sui quali viene richiamata la funzione sono in tutti e tre i casi all'incirca $2/3n$.

Si ricordi ora che un'equazione di ricorrenza si compone della parte relativa al caso generale e quella relativa al caso base.

Cominciamo quindi ad individuare le linee di codice relative al caso base. Esse sono, ovviamente, le righe 3 e 4, e quindi interessano il caso in cui $i+1 \geq j$. Poiché i e j rappresentano rispettivamente l'inizio e la fine del vettore in considerazione, siamo in un caso base tutte le volte che l'elemento successivo a quello di indice i si trova in corrispondenza di j oppure alla sua destra, quindi il caso base si ottiene quando $n \leq 2$ ed, in tal caso, la complessità è pari a $\Theta(1)$ poiché vengono eseguite le linee da 1 a 4, tutte operazioni di costo costante.

Nel caso generale, invece, la complessità è data dal contributo $\Theta(1)$ proveniente dalle linee 1-5 e dal contributo delle tre linee 6-8, ciascuna delle quali ha complessità $T(2/3\ n)$, per cui: $T(n)= \Theta(1)+3T(2/3\ n)$.

2. Questa equazione di ricorrenza si può risolvere tramite teorema principale, tramite metodo iterativo o metodo dell'albero, ed in tutti e tre i casi la soluzione risulta essere T(n)= $\Theta(n^{\log_{3/2} 3})$.

Poiché la soluzione non presenta particolari difficoltà, omettiamo qui i dettagli.

3. Confrontiamo ora la complessità della funzione data con quella dell'InsertonSort, $\Theta(n^2)$, e del MergeSort, $\Theta(n \log n)$. Per fare ciò, dobbiamo farci un'idea del valore di $\log_{3/2}3$. Osserviamo che la funzione logaritmica è una funzione monotona crescente, e quindi:

$1 = \log_{3/2}3/2 < 2 = \log_{3/2}9/4 < \log_{3/2}3$.

Pertanto $\log_{3/2}3$ è un valore strettamente maggiore di 2. Ne consegue che la complessità di `UnAltroSort` è peggiore sia di quella di MergeSort che di quella di InsertionSort.

Esercizio 8.

Riferimento ai capitoli: 6. Il problema dell'ordinamento

Dato un heap *H* di *n* elementi, descrivere a parole l'algoritmo che cancella un prefissato nodo *i* e riaggiusta l'heap risultante di *n – 1* elementi.

Valutare la complessità dell'algoritmo presentato.

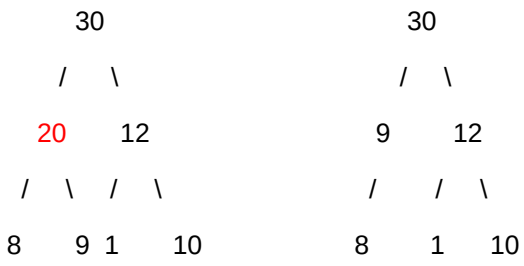
Soluzione.

Il problema si scomplone in due parti: la prima consiste nel cercare l'elemento da eliminare e l'altra nel cancellarlo e ripristinare l'heap.

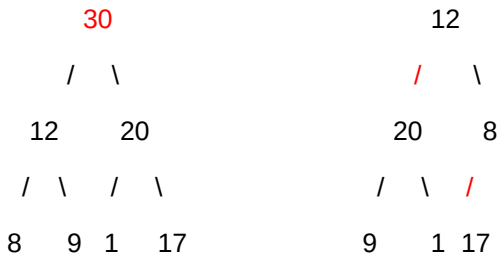
L'elemento *i* da eliminare può essere inteso come l'elemento di posizione *i* oppure l'elemento di chiave *i*. Nel primo caso l'individuazione dell'elemento richiede tempo costante (l'elemento è *H[i]*), nel secondo bisogna effettuare una ricerca, che può richiedere anche tempo lineare in *n*, cioè *O(n)*.

Una volta noto l'elemento, sia esso *H[x]*, è possibile eliminarlo apparentemente in molti modi:

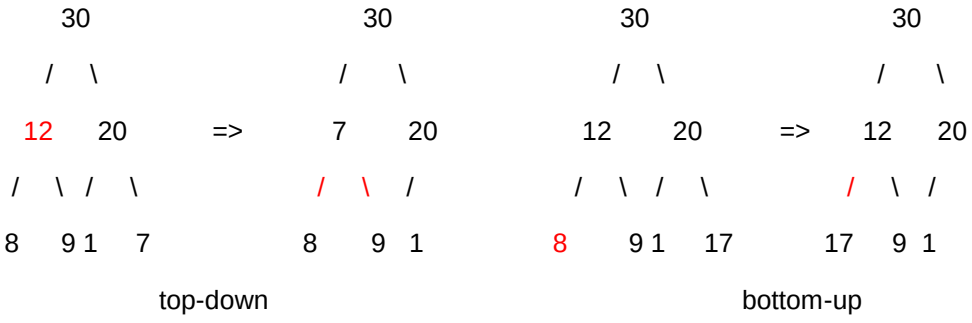
- si può portare verso le foglie l'elemento da cancellare scambiandolo con il maggiore dei suoi figli; una volta che l'elemento ha raggiunto le foglie, si elimina semplicemente; questo algoritmo e' errato perche' un heap è un albero binario completo o quasi completo e questo approccio può far perdere questa proprietà; si veda, ad esempio, nell'esempio qui sotto, cosa succede quando si tenta di eliminare il nodo di chiave 20 con questo approccio:



- si può shiftare tutti gli elementi *H[x+1], H[x+2], …, H[n]* di una posizione verso sinistra e poi riaggiustare l'heap, ma questa tecnica, seppure formalmente corretta risulta essere troppo costosa in termini di tempo, infatti lo shift a sinistra provoca un disallineamento dei figli rispetto ai padri per cui nessun nodo è garantito mantenere la proprietà dell'heap, che – per essere aggiustato – deve essere interamente ricostruito; si veda l'esempio qui sotto:



- si può scambiare H[x] con H[n], cioè con la foglia più a sinistra, eliminare tale foglia e riaggiustare l'heap risultante. Per fare ciò, bisogna confrontare il nuovo H[x] con il maggiore dei suoi figli; se H[x] risulta minore, si può applicare la funzione di aggiustamento dell'heap top-down; se H[x] sembra in posizione corretta rispetto ai suoi figli, non è ancora detto che l'heap sia corretto, bisogna infatti confrontarlo con il padre, se questo è minore di H[x] si deve procedere all'aggiustamento bottom-up:



Se escludiamo la ricerca del nodo, l'algoritmo di cancellazione e riaggiustamento ha una complessità pari al massimo tra la complessità dell'aggiustamento top-down e di quello bottom-up. Poiché entrambi si possono eseguire in tempo *O(log n)*, ne consegue che *O(log n)* è proprio la complessità dell'algoritmo presentato.

Esercizio tipo esame 9.

Riferimento ai capitoli: 7. Strutture dati fondamentali

Siano date *k* liste ordinate in cui sono memorizzati globalmente *n* elementi. Descrivere un algoritmo con tempo *O(n log k)* per fondere le *k* liste ordinate in un'unica lista ordinata.

Soluzione.

Si consideri, innanzi tutto, il classico algoritmo di fusione applicato a due liste:

- al generico passo *i*, sono già stati sistemati nella lista finale *i* elementi, e vi sono due puntatori alle teste (contenenti gli elementi più piccoli) delle due liste; si individua quale tra i due elementi puntati sia il minore, e questo viene inserito nella lista risultante in coda. Si passa quindi al passo *i+1*.

Il numero di passi complessivi è ovviamente pari ad *n*, cioè al numero complessivo di elementi.

E' naturale cercare di generalizzare questo algoritmo a *k* liste ordinate, purché si sia in grado di calcolare il minimo fra *k* elementi. Banalmente, questo si può fare in tempo $\Theta(k)$, producendo quindi un algoritmo di fusione di *k* liste di complessità $\Theta(nk)$.

Ma il tempo richiesto dal testo deve essere più basso, quindi dobbiamo trovare il modo di lavorare più efficientemente. Posto che non sembra possibile ridurre il numero di iterazioni, che dovrà comunque rimanere *n*, non resta che ragionare sulla ricerca del minimo tra *k* elementi.

Richiamiamo alla memoria il fatto che una struttura dati in grado di restituire efficientemente il valore minimo, tra quelli che contiene, è il min-heap. Allora, al generico passo *i*, i valori delle *k* teste delle *k* liste ordinate sono memorizzati in un min-heap, da cui si può estrarre il minimo in tempo *O(log k)*; una volta eliminato tale valore dalla struttura ed aggiornato il puntatore relativo alla testa della lista corrispondente, dovremo inserire nel min-heap il nuovo valore, ed anche questo impiega un tempo *O(log k)*.

Iterando questo procedimento per tutti gli *n* passi, si giunge ad una complessità di *O(n log k)*, che è quella richiesta.

Esercizio tipo esame 10.

Riferimento ai capitoli: 4. Equazioni di ricorrenza; 7. Strutture dati fondamentali

Sia dato un albero binario completo con *n* nodi, radicato al nodo puntato dal puntatore *r*. Calcolare la complessità computazionale della seguente funzione, in funzione di *n*:

```
link Funzione(link r)
{
    int fogliasx,fogliadx; link r1;
1.  if (!r->fsin) && (!r->fdes)
2.     return r
3.  else {
4.     r1=r;
5.     while (r1->fsin) do
6.        r1=r1->fsin;
7.     fogliasx=r1->dato;
8.     r1=r;
9.     while (r1->fdes) do
10.       r1=r1->fdes;
11.    fogliadx=r1->dato;
12.    if (fogliasx<fogliadx) return Funzione(r->fsin)
13.    else return Funzione(r->fdes);
14. }
}
```

Soluzione.

La funzione è ricorsiva e pertanto dobbiamo trovare un'equazione di ricorrenza in funzione di *n*; la complessità si ottiene sommando i contributi delle varie istruzioni:

- le linee 1 e 2 rappresentano il caso base, che costa $\Theta(1)$;
- le linee 4, 7, 8 e 11 costano $\Theta(1)$;
- i cicli while delle linee 5 e 9 vengono ripetuti un certo numero di volte indefinto tra 1 e l'altezza dell'albero, infatti il primo ciclo si interrompe quando il nodo non ha figlio sinistro ed il secondo ciclo quando il nodo non ha figlio destro;
- le linee 12 e 13 sono chiamate ricorsive e quindi la loro complessità si dovrà scrivere come T(…) con un opportuno valore tra le parentesi, ma quale?

Se indichiamo con *k* ed $n - 1 - k$ il numero dei nodi dei sottoalberi radicati ai figli sinistro e destro della radice, otteniamo:

$$T(n) = \Theta(1) + v_1\,\Theta(1) + v_2\,\Theta(1) + max(T(k), T(n-k))$$

dove v_1 e v_2 sono il numero di volte in cui vengono ripetuti i cicli delle linee 5 e 9.

Arrivati qui, non sappiamo come procedere, se non sostituendo a v_1 e v_2 l'altezza dell'albero, che ne è una limitazione superiore. L'altezza ha un valore indefinito tra $\log n$ ed n, e questo significa che dobbiamo anche qui sostituire ad *h* la sua limitazione superiore *n*. Viste tutte

queste approssimazioni che siamo costretti a fare, dovrebbe venire il dubbio di aver tralasciato qualcosa. Rileggendo con attenzione il testo, si osservi che l'albero in input è binario e ***completo***. Questa ipotesi, che abbiamo tralasciato finora, risolove numerosi problemi, infatti:

- tutti i nodi privi di figlio sinistro (ed anche di figlio destro) sono ad altezza *log n,* pertanto i due cicli alle linee 5 e 9 vengono ripetuti esattamente *log n* volte ciascuno;
- diventa facile calcolare max*(T(k), T(n – k))*: tanto il sottoalbero sinistro che quello destro contengono esattamente *(n –1)/2* nodi ciascuno.

L'equazione di ricorrenza diventa allora:

$T(n) = \Theta(1) + \Theta(\log n) + T(n/2) = \Theta(\log n) + T(n/2)$

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

Risolvendo per iterazione si ha:

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

$$= \Theta(\log n) + (\Theta(\log \tfrac{n}{2}) + T(\tfrac{n}{2^2})) =$$

$$= \ldots =$$

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

Si procede fino a quando $n/2^k = 1$ e cioè fino a quando $k = \log n$, ottenendo:

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

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

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

$$= \Theta\left(\sum_{i=0}^{\log n - 1} (\log n - i)\right) + T(1) =$$

$$= \Theta\left(\log n \sum_{i=0}^{\log n - 1} 1 - \sum_{i=0}^{\log n - 1} i\right) + T(1) =$$

$$= \Theta\left(\log^2 n + \frac{\log n \log(n-1)}{2}\right) + T(1) =$$

$$= \Theta\left(\log^2 n - \frac{\log^2 n}{2} + \frac{\log n}{2}\right) + T(1) = \Theta(\log^2 n)$$

Esercizio tipo esame 11.

Riferimento ai capitoli: 8. Dizionari

Siano dati due alberi binari di ricerca: B_1 con n_1 nodi ed altezza h_1, e B_2 con n_2 nodi ed altezza h_2. Assumendo che tutti gli elementi in B_1 siano minori di quelli in B_2, descrivere un algoritmo A che fonda gli alberi B_1 e B_2 in un unico albero binario di ricerca B di $n_1 + n_2$ nodi.

Determinare l'altezza dell'albero B e la complessità computazionale di A.

Soluzione.

La prima idea che si può provare a perseguire è quella di inserire nell'albero con il maggior numero di nodi (senza perdere di generalità sia esso B_1) i nodi dell'altro albero uno ad uno. Sapendo che l'operazione di inserimento in un albero binario di ricerca ha una complessità temporale dell'ordine dell'altezza dell'albero si ha, nel caso peggiore, che la complessità è:

$$O(h_1) + O(h_1+1) + O(h_1+2) + \ldots + O(h_1 + n_2 - 1) = n_2 O(h_1) + O(1 + 2 + \ldots + n_2 - 1)$$

Poiché nel caso peggiore $O(h_1) = O(n_1)$ ed $O(1 + 2 + \ldots + n_2) = O(n_2^2)$, si ottiene che la complessità di questo approccio è $O(n_1 n_2) + O(n_2^2) = O(n_1 n_2)$, essendo per ipotesi $n_1 > n_2$.

Tuttavia, questa soluzione non utilizza l'ipotesi che tutti gli elementi di B_1 siano minori di quelli di B_2. Per tentare di sfruttare questa ipotesi (è buona norma tenere presente che se un'ipotesi c'è allora serve a qualcosa!) possiamo pensare di appendere l'intero albero B_1 come figlio sinistro del minimo di B_2. Questo si può sempre fare facilmente perché il minimo di un albero binario di ricerca è, per definizione, il nodo più a sinistra che non possiede figlio sinistro, pertanto è sufficiente settare un puntatore sulla radice di B_2, scendere verso il figlio sinistro finché esso esista e, giunti al nodo che non ha figlio sinistro (che è il minimo), agganciargli a sinistra la radice di B_1. Osserviamo che questo procedimento è corretto perché un albero binario di ricerca ***non*** è necessariamente bilanciato, e quindi poco importa che l'altezza dell'albero risultante possa diventare $O(h_1 + h_2)$. La complessità di questo approccio è dominata dalla complessità della ricerca del massimo cioè da $O(h_1)$ ed è, pertanto, da preferirsi al primo metodo proposto.

Esercizio tipo esame 12.

Riferimento ai capitoli: 9. Grafi

Sia *G* un grafo con *n* nodi ed *m* spigoli. Dire se le seguenti affermazioni sono vere o false e giustificare la propria risposta:

1) tutte le foreste generate da differenti visite in profondità hanno lo stesso numero di alberi;
2) tutte le foreste generate da differenti visite in profondità hanno lo stesso numero di spigoli dell'albero e lo stesso numero di spigoli all'indietro.

Soluzione.

Si ricordi che, dato un grafo *G* con *k* componenti connesse, qualsiasi foresta ricoprente (che sia generata da una visita in profondità o no) è formata esattamente da *k* alberi, uno per ciascuna componente connessa. Da questa semplice osservazione si deduce che la risposta al primo quesito è affermativa.

Anche la risposta al secondo quesito è affermativa; infatti ogni albero con *a* nodi ha *a – 1* spigoli, ed ogni foresta di *k* alberi con *n* nodi ha *n – k* spigoli. Pertanto il numero di spigoli dell'albero è uguale per tutte le foreste (sia quelle generate da visita in profondità che le altre).

Per quanto riguarda gli spigoli all'indietro, si ricordi che una visita in profondità non produce spigoli di attraversamento, e quindi tutti gli spigoli non dell'albero sono all'indietro; ne consegue che tutte le foreste generate da differenti visite in profondità hanno *m – (n – k)* spigoli all'indietro.

Concludendo, mentre le proprietà descritte dalla prima affermazione e dalla parte della seconda affermazione riguardante gli spigoli dell'albero sono proprietà valide per tutti gli alberi ricoprenti, indipendentemente dal fatto che essi scaturiscano da una visita in profondità o no, la proprietà sugli archi all'indietro relativa alla seconda affermazione si basa pesantemente sulle proprietà degli alberi generati da visita in profondità.

Simboli

O

Ω

Θ

$\boldsymbol{\varepsilon}$

$\supseteq \subseteq \subset \supset \Rightarrow$

$\in$

$\leftarrow$