Politecnico di Milano
Facoltà di Ingegneria dell'Informazione
Informatica 3
Proff. Ghezzi, Lanzi, Matera e Morzenti
Seconda prova in itinere
4 Luglio 2005

COGNOME E NOME (IN STAMPATELLO)

MATRICOLA

Risolvere i seguenti esercizi, scrivendo le risposte ed eventuali tracce di soluzione negli spazi disponibili.
Non consegnare altri fogli.

Spazio riservato ai docenti

Esercizio 1: Heap.

Quesito 1.
Si consideri l'algoritmo di costruzione di un max-heap a partire da n valori memorizzati in un array. Tale algoritmo è implementato dal metodo **buildheap()**di seguito riportato:

```
Public void buildheap()
 {  for (int i =n/2-1; i>=0; i--)
      siftdown(i);
}
```

A partire dal seguente array:

12	28	7	31	2	45	40	30	50	1

Si mostri il risultato delle varie ripetizioni di **siftdown(i)**, fino a ottenere un array che rappresenti un max-heap.

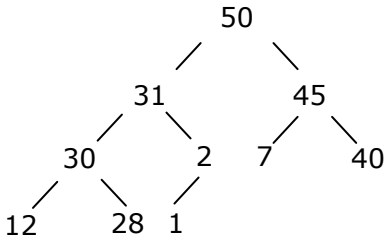
Risposta:

	12	28	7	31	2	45	40	30	50	1
Passo 1	12	28	7	31	2	45	40	30	50	1
Passo 2	12	28	7	50	2	45	40	30	31	1
Passo 3	12	28	45	50	2	7	40	30	31	1
Passo 4	12	50	45	31	2	7	40	30	28	1
Passo 5	50	31	45	30	2	7	40	12	28	1

Esercizio 1 (continua).

Quesito 2.
Si tracci il max-heap (un albero binario completo) corrispondente all'array ottenuto al punto precedente.

Risposta:



Esercizio 2: Tabelle Hash.

Quesito 1.
Data una tabella hash di lunghezza **m=11**, con funzione di hash **h(x)=x mod 11**, illustrare le operazioni sotto riportate, utilizzando *linear probing* e *quadratic probing*.
Rappresentare eventuali *tumbstone* (si utilizzi il simbolo T).
Nel caso di collisioni, elencare le posizioni analizzate nella *sequenza di probe*.

LINEAR PROBING

	0	1	2	3	4	5	6	7	8	9	10	Probe Sequence
Inserisci 11												
Inserisci 12												
Inserisci 23												
Inserisci 34												
Inserisci 56												
Inserisci 38												
Cancella 23												
Inserisci 78												

QUADRATIC PROBING

	0	1	2	3	4	5	6	7	8	9	10	Probe Sequence
Inserisci 11												
Inserisci 12												
Inserisci 23												
Inserisci 34												
Inserisci 56												
Inserisci 38												
Cancella 23												
Inserisci 78												

Risposta:

LINEAR PROBING

	0	1	2	3	4	5	6	7	8	9	10	Probe Sequence
Inserisci 11	11											
Inserisci 12	11	12										
Inserisci 23	11	12	23									1,2
Inserisci 34	11	12	23	34								1,2,3
Inserisci 56	11	12	23	34	56							1,2,3,4
Inserisci 38	11	12	23	34	56	38						
Cancella 23	11	12	T	34	56	38						1,2
Inserisci 78	11	12	78	34	56	38						1,2

QUADRATIC PROBING

	0	1	2	3	4	5	6	7	8	9	10	Probe Sequence
Inserisci 11	11											
Inserisci 12	11	12										
Inserisci 23	11	12	23									1,2
Inserisci 34	11	12	23			34						1,2,5
Inserisci 56	11	12	23			34					56	1,2,5,10
Inserisci 38	11	12	23			34	38				56	5,6
Cancella 23	11	12	T			34	38				56	1,2
Inserisci 78	11	12	78			34	38				56	1,2

Esercizio 2 (continua).

Quesito 2.
Osservando le due tabelle ottenute al punto precedente, si dica quale problema del linear probing e' risolto dal quadratic probing.

Risposta:
Il quadratic probing risolve il problema del clustering

Esercizio 3: Sorting.

Quesito 1.
In un array a contenente n elementi distinti, si dice che la coppia `(a[i], a[j])` costituisce un'inversione se `i<j` e `a[i]>a[j]`. Il numero di inversioni presenti in a si definisce come il numero di coppie `(a[i], a[j])`, con $0 \le i < j \le n-1$ che costituiscono un'inversione. Per esempio, per l'array contenente gli elementi `[3, 7, 2, 9, 5]`, il numero di inversioni in a è 4, corrispondente alle coppie `(3,2), (7,2), (7,5), (9,5)`.

Indicare la relazione tra la complessità di calcolo dell'algoritmo di ordinamento per inserzione e il numero di inversioni presenti nell'array che esso riceve in ingresso. Fornire una breve spiegazione.

Risposta:
Facendo riferimento al codice dell'algoritmo

```
static void inssort(Elem[] array) {
  for (int j=1; j<array.length; j++)
    for (int i=j; (i>0) && (array[i].key()<array[i-1].key()); i--)
      DSutil.swap(array, i, i-1);
}
```

si vede che la complessità è strettamente legata al numero delle inversioni. Infatti
- per ogni valore assunto dal contatore j del ciclo for più esterno, il corpo del ciclo for più interno viene eseguito un numero di volte pari al numero di inversioni (a[i], a[j])
- il numero di confronti tra elementi complessivamente effettuati (mediante la valutazione dell'espressione `array[i].key()<array[i-1].key()`) è pari al numero delle inversioni + (n-1)
- quindi, detti N_{inv} il numero delle inversioni presenti nell'array ed *n* il numero dei suoi elementi, si possono trovare tre costanti k_1, k_2 e k_3 tali che la complessità temporale dell'algoritmo inssort è $T(n)= k_1 \cdot N_{inv} + n \cdot k_2 + k_3$.

Ciò spiega anche perché l'algoritmo di ordinamento per inserzione è particolarmente efficiente su array "quasi ordinati", che cioè hanno un ridotto numero di inversioni.

Esercizio 3 (continua).

Quesito 2.
Scrivere un semplice algoritmo per contare il numero di inversioni presenti in un array, che abbia complessità $\Theta(n^2)$, fornendo una sintetica argomentazione del fatto che esso calcola effettivamente il numero delle inversioni e del fatto che è quadratico.

Risposta:

```
static int QinvCount(Elem[] array) {
  int c = 0;
  for (int i=0; i<array.length-1; i++)
    for (int j=i+1; (j<array.length); j++)
      if (a[i]>a[j]) c++;
  return c;
}
```

L'algoritmo ha complessità quadratica perché il ciclo for interno viene eseguito n-1 volte (con n=a.length) e il suo corpo viene ripetuto per un numero decrescente di volte, partendo da n-1 (la prima volta) e arrivando a 1 (l'ultima). Nel far ciò l'algoritmo considera tutte le coppie (a[i], a[j]) con $0<i<j\leq$a.length, e quindi effettivamente calcola il numero delle inversioni.

Esercizio 3 (continua).

Quesito 3.
Definire un algoritmo che calcoli il numero di inversioni in un array in tempo $\Theta(n \cdot \log n)$ nel caso pessimo, fornendo un'opportuna, sintetica spiegazione della sua correttezza e una valutazione della sua complessità.
Suggerimento: si applichi il metodo *divide et impera*, ispirandosi all'algoritmo di ordinamento per fusione (*mergesort*).

Risposta:

Si osserva innanzitutto che nell'algoritmo di mergesort, durante la fase di fusione, quando l'elemento da immettere nel segmento di array ordinato in corso di costruzione viene preso dal secondo dei due segmenti ordinati di partenza, esso scavalca (nel senso che viene messo a sinistra di) tutti gli elementi che ancora si trovano nel primo dei due segmenti, e che gli sono maggiori. Se *k* è il numero di tali elementi (*k* vale r1-i1+1 nella codifica sotto riportata), vengono così eliminate *k* inversioni. Si può quindi definire una versione dell'algoritmo di mergesort che nelle fasi di fusione tiene un conteggio delle inversioni eliminate in questo modo. Il valore restituito è evidentemente pari al numero delle inversioni complessive dell'array di partenza, perché alla fine l'array risulta ordinato. Inoltre la complessità dell'algoritmo è $\Theta(n \cdot \log n)$, perché il conteggio delle inversioni non altera la complessità asintotica dell'algoritmo di mergesort, preso come punto di partenza per la codifica del nuovo algoritmo.

```
static int mergesortCountInv(Elem[] array, Elem[] temp,
                     int l, int r) {
  if (l == r) return 0;
  int mid = (l+r)/2;
  int c1 = mergesortCountInv(array, temp, l, mid);
  int c2 = mergesort(array, temp, mid+1, r);
  return c1 + c2 + mergeCountInv(array, temp, l, mid, mid+1, r);
}


static int mergeCountInv(Elem[] array, Elem[] temp,
                  int l1, int r1, int l2, int r2) {
  for (int i=l1; i<=r2; i++)
    temp[i] = array[i];
  int countInv = 0;
  int i1 = l1; int i2 = l2;
  for (int curr=l1; curr<=r2; curr++) {
    if (i1 > r1)
      array[curr] = temp[i2++];
    else if (i2 > r2)
      array[curr] = temp[i1++];
    else if (temp[i1].key() < temp[i2].key())
      array[curr] = temp[i1++];
    else {
      //temp[i2] scavalca gli r1-i1+1 elementi
      //ancora presenti nel primo segmento
      countInv += r1-i1+1;
      array[curr] = temp[i2++];
    }
  }
  return countInv;
}
```