Algoritmi e Strutture Dati

CdL in Ingegneria Informatica

Università degli Studi del Sannio
Dipartimento di Ingegneria

A.A. 2023/2024

Esercizio 1

Sviluppare l'algoritmo BinaryInsertion che utilizzi la ricerca binaria per trovare il punto di inserimento j per l'entry a[i] e sposti tutti gli entry a[j],...,a[i-1] di una posizione a destra.

Il numero di confronti per ordinare un array di lunghezza N deve essere ~N lg N nel caso peggiore.

Esercizio 1

```
public class BinaryInsertion{

public static void sort(Comparable[] a) {
      int n = a.length;
      for (int i = 1; i < n; i++) {

         // ricerca binaria per determinare la posizione j nella quale inserire a[i]
         Comparable v = a[i];
         int lo = 0, hi = i;
         while (lo < hi) {
             int mid = lo + (hi - lo) / 2;
             if (less(v, a[mid])) hi = mid;
             else                 lo = mid + 1;
         }


         // shift di a[j], ..., a[i-1] a destra e inserimento di a[i] in posizione
         for (int j = i; j > lo; --j)
             a[j] = a[j-1];
         a[lo] = v;
      }
}

private static boolean less(Comparable v, Comparable w) {
    return v.compareTo(w) < 0;
}

}
```

Esercizio 2

Si supponga di avere un array di n interi A, ordinato in modo crescente, tale che

$$\forall\, i = 0, \dots, n-1 \; \mathrm{A[i]} \in \{1,2,3\}$$

Si implementi un algoritmo con complessità temporale O(log n) che calcoli le occorrenze del numero 2 all'interno dell' array A.

Esercizio 2

```java
public class FindElements{


public int numOfTwos(int[] A){
    int n = A.length-1;

    // Gli indici h e k rappresentano rispettivamente
    // l'indice dell'ultimo 1 e l'indice dell'ultimo 2
    int h, k;

    if (A[n] == 1){
        // Non ci sono 2
        return 0;
    }

    if (A[0] == 2){
        // Non ci sono 1
        h = 0;
    }else{
        // Uso la ricerca binaria per trovare l'indice h
        h = lastX(A,1,1,n);
    }
    if (A[n] == 2){
        k = n;
    }else{
        // Uso la ricerca binaria per trovare l'indice k
        k = lastX(A,2,0,n-1);
    }
    return (k - h);
}
```

Esercizio 2 (continua)

```
private int lastX(int[] A, int x, int i, int j){
   if (i > j){
        return -1;
   }
   m = (i+j)/2;
   if (A[m] == x && A[m+1] > x){
       return m;
   }
   if (A[m] > x){
       return lastX(A,x,i,m-1);
   }else{
       return lastX(A,x,m+1,j);
   }
}
}
```

Esercizio 3

Scrivere un programma per verificare se esistono almeno due valori distinti che compaiono lo stesso numero di volte in un dato array.

Esempio:
- dato in input [3, 5, 5, 3, 4, 7, 4], la risposta è SI, perché 3, 5 e 4 compaiono lo stesso numero di volte;

- dato [3, 4, 3, 3, 4], la risposta è NO.

Evitare soluzioni con complessità $O(n^2)$

Esercizio 3

```
public class SameValues {
  public boolean sameValues(Object array[]) {

      // Una tabella di hash per contare le occorrenze di ogni oggetto
      HashMap<Object,Integer> counts = new HashMap<Object,Integer>();
      for (Object o: array) {
          if (!counts.containsKey(o)) {
              counts.put(o,1);
          }else {
              int count = counts.get(o);
              counts.put(o,++count);
          }
      }
      return hasDuplicates(counts.values().toArray(new Integer[counts.values().size()]));
  }

  // il metodo helper hasDuplicates costruisce un hashSet con i valori
  // corrispondenti alle occorrenze dei vari oggetti
  private boolean hasDuplicates(Integer[] values) {
      Set<Integer> lump = new HashSet<Integer>();
      for (int i = 0; i<values.length; i++) {
          // appena si prova ad inserire un valore che è già presente nel Set
          // il metodo ritorna true: abbiamo trovato due oggetti che occorrono
          // lo stesso numero di volte
          if (lump.contains(values[i])) return true;
          lump.add(values[i]);
      }
      return false;
  }
}
```