Algoritmi e Strutture Dati

CdL in Ingegneria Informatica

Università degli Studi del Sannio
Dipartimento di Ingegneria

A.A. 2020/2021

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 non decrescente, 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

```
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;
  }
}
```

Esercizio 4

Implementare una versione iterativa della DFS usando uno stack.

Esercizio 4

```
public class IterativeDFS {
  private boolean[] marked;

  public void dfsUsingStack(Graph G, int v){
      marked = new boolean[G.V()];

      for (int i = 0; i<G.V();i++) {
          marked[i] = false;
      }

      Stack<Integer> stack = new Stack<Integer>();
      stack.push(v);

      while(!stack.isEmpty()) {
          int vertex = stack.pop();
          if (!marked[vertex]){
              System.out.println(vertex);
              marked[vertex] = true;
          }

          for (int w: G.adj[vertex]) {
              if (!marked[w]) {
                  stack.push(w);
              }
          }
      }
  }
}
```

Esercizio 5

Implementare un algoritmo per eseguire lo swap di due variabili di tipo intero, senza utilizzare variabili temporanee.

Esercizio 5

```
public class Swap {

  public void swap(Integer a, Integer b) {
      a = a + b;

      // Dal momento che a = (a + b) -> b = (a + b) - b = a
      b = a - b;

      // Dal momento a = (a + b) e b = a -> a = (a + b) - a = b
      a = a - b;
  }

}
```

Esercizio 6

Implementare un algoritmo che data una Stringa verifichi che essa sia palindroma. L'algoritmo deve avere complessità temporale sublineare.

Esercizio 6

```
public class Palindroma {

  public static boolean palindroma(String s) {
      if (s.length() < 2)
          return true;
      if (s.charAt(0) == s.charAt(s.length() - 1))
          return palindroma(s.substring(1, s.length() - 1));
      else
          return false;
  }

}
```

Esercizio 7

Implementare un algoritmo che date due Stringhe verifichi che la seconda stringa sia un anagramma della prima. La complessità temporale dell'algoritmo deve essere minore di $O(n^2)$

Esempio: "calendario" e "locandiera"

Esercizio 7

```
public class Anagram {

public static boolean isAnagram(String word, String anagram){

    if (word.length != anagram.length){
        return false;
    }
    char[] charFromWord = word.toCharArray();
    char[] charFromAnagram = anagram.toCharArray();

    // I due array vengono ordinati
    Arrays.sort(charFromWord);
    Arrays.sort(charFromAnagram);

    // verifico che i due array ordinati contengano gli stessi
    // caratteri nelle stesse posizioni
    for (int i = 0; i<charFromWord.length; i++){
          if (charFromWord[i] != charFromAnagram[i]){
                return false;
          }
    }

    return true;

}

}
```

Esercizio 8

Dato un array di n interi, A tale che

$$\forall\, i = 0, \dots, n-1 \; \mathrm{A[i]} \in \{0,1, \dots, k\} \; con \; k < n$$

Implementare un algoritmo con complessità temporale minore di O(n log n) che ordini l'array A.

Esercizio 8

```
public class CountingSort{

public static void countingSort(int[] input, int k) {
  // Definisco un array di contatori con k+1 posizioni
  int counter[] = new int[k + 1];

  // Conteggio le occorrenze di ogni elemento dell'array
  for (int i : input) {
      // Utilizzo l'elemento dell'array orignario
      // come indice per incrementare il conteggio
      counter[i]++;
  }

  // ordino l'array utilizzando le occorrenze di ogni elemento
  int n = 0;
  for (int i = 0; i < counter.length; i++) {
      while (counter[i] > 0) {
          input[n++] = i;
          counter[i]--;
      }
  }
}

}
```

Esercizio 9

Implementare un algoritmo che ricevuto in ingresso un array di interi di n elementi e un intero x, restituisca:

- **true** nel caso in cui all'interno dell'array esistano due elementi la cui somma sia pari a x;
- **false** altrimenti.

L'algoritmo deve avere complessità temporale O(n).

Esercizio 9

```
class Main{
  public boolean void findPair(int[] A, int x)
  {
      // utilizzo una tabella di hash
      Map<Integer, Integer> map = new HashMap<Integer, Integer>();


      for (int i = 0; i < A.length; i++)
      {
          // verifico se la coppia (A[i], x-A[i]) esiste
          if (map.containsKey(x - A[i]))
          {
              System.out.println("La somma degli elementi di posizione " +
                                 map.get(x - A[i]) + " e " + i +" è "+ x);
              return true;
          }

          // memorizzo l'indice dell'elemento corrente nell'array
          map.put(A[i], i);
      }

      System.out.println("Non esistono coppie la cui somma sia "+x);
      return false;
  }

}
```

Esercizio 10

Implementare un algoritmo di complessità temporale inferiore a $O(n^2)$ che dato un array di interi trova il massimo prodotto di due elementi nell'array.

Esempio: considerato l'array {5,4,-10,-3,1} il massimo prodotto è dato dalla coppia di elementi (-10,-3).

Esercizio 10

```java
class Main
{
  public static void findMaximumProduct(int[] A)
  {
      // n è la cardinalità dell' array
      int n = A.length;

      // ordino l'array i
      Arrays.sort(A);

      // il massimo prodotto può essere dato dalle seguenti coppie:
      // 1. i primi due elementi (ad esempio quando questi sono entrambi < 0) o
      // 2. gli ultimi due elementi (che sono i maggiori in un array ordinato)

      if ((A[0] * A[1]) > (A[n - 1] * A[n - 2])) {
          System.out.print("(" + A[0] + ',' + A[1] + ')');
      } else {
          System.out.print("(" + A[n - 1] + ',' + A[n - 2] + ')');
      }
  }

}
```

Esercizio 11

Dato un array in cui tutti gli elementi, eccetto due (che si sono scambiati di posto), siano ordinati in maniera crescente, ordinare l'array in un tempo lineare. Assumere che non ci siano valori duplicati nell'array.

Esempi: [3,8,6,7,5,9] OR [3,5,6,9,8,7]

Esercizio 11

```java
class Main
{
  private static void sortArray(int[] arr)
  {
      int x = -1, y = -1;
      int prev = arr[0];

      // analizzo ogni coppia di elementi adiacenti
      for (int i = 1; i < arr.length; i++)
      {
          // se l'elemento precedente è maggiore dell'elemento corrente
          if (prev > arr[i])
          {
              // primo elemento del conflitto
              if (x == -1) {
                  x = i - 1;
                  y = i;
              }
              else {
                  // secondo elemento del conflitto
                  y = i;
              }
          }
          prev = arr[i];
      }
      // swap degli elementi presenti agli indici x e y
      swap(arr, x, y);
  }

  private static void swap(int[] a, int i, int j) {
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
  }
}
```

Esercizio 12

Implementare un algoritmo di complessità minore di O(n) che dato un array A di n interi, trovi un elemento di picco. Un elemento A[i] è di picco se non è minore dei suoi elementi contigui:

A[i-1] <= A[i] <= A[i+1] per 0 < i < n-1

A[i-1] <= A[i] se i = n-1

A[i] >= A[i+1] se i = 0

Esempi:

nell' array {8,9,10,2,5,6} l'elemento di picco è 10

nell' array {8,9,10,12,15] l'elemento di picco è 15

nell' array {10,8,6,5,3,2} l'elemento di picco è 10

Esercizio 12

```
class Main
{
 // Funzione ricorsiva per la ricerca dell'elemento di picco
 public static int findPeakElement(int[] A, int left, int right)
 {
     // accedo all'elemento centrale
     int mid = (left + right) / 2;

     // verifico se l'elemento di posizione mid sia maggiore degli elementi contigui
     if ((mid == 0 || A[mid - 1] <= A[mid]) &&
             (mid == A.length - 1 || A[mid + 1] <= A[mid])) {
         return mid;
     }

     // Se l'elemento a sinistra di mid è maggiore dell'elemento di posizione mid,
     // vado a cercare il picco ricorsivamente nel sottoarray di sinistra.
     if (mid - 1 >= 0 && A[mid - 1] > A[mid]) {
         return findPeakElement(A, left, mid - 1);
     }

     // Viceversa, se l'elemento a destra di mid è maggiore dell'elemento di posizione mid,
     // vado a cercare il picco ricorsivamente nel sottoarray di destra.
     return findPeakElement(A, mid + 1, right);
 }

}
```

Esercizio 13

Dato un albero binario T, ed un nodo n appartenente a T, si definisce *imbalance* di n, la differenza in valore assoluto tra il numero di foglie nei sotto-alberi sinistro e destro di n. Scrivere un algoritmo per il calcolo dell'imbalance e valutarne la complessità.

Esercizio 13

```java
class Main{
  static class Node{
      Object val;
      Node sinistro,destro;
  }

  public static int calcolaImbalance(Node n) {
      int leftCount = contaFoglie(n.sinistro);
      int rightCount = contaFoglie(n.destro);
      return Math.abs(leftCount - rightCount);
  }

  private static int contaFoglie(Node n) {
      int count = 0;
      Stack<Node> stack = new Stack<Node>();
      stack.add(n);

      // DFS iterativa
      while (!stack.isEmpty()) {
          Node node = stack.pop();
          if (node.destro!=null) {
              stack.push(node.destro);
          }
          if (node.sinistro!=null) {
              stack.push(node.sinistro);
          }

          // quando mi trovo in una foglia aggiorno il contatore
          if (node.sinistro == null && node.destro == null) {
              count++;
          }

      }
      return count;
  }
}
```

Esercizio 14

Implementare un algoritmo efficiente per verificare che due alberi binari siano identici oppure no. Due alberi binari sono identici se hanno uguale struttura e il contenuto di ogni nodo corrispondente è lo stesso.

Esercizio 14

```
class Node
{
  Object value;
  Node left = null, right = null;

  Node(Object value) {
      this.value = value;
  }
}

class Main
{
  // Funzione ricorsiva per verificare che due alberi binari siano identici
  public static boolean isIdentical(Node x, Node y)
  {
      // se i due alberi sono entrambi vuoti, ritorno true
      if (x == null && y == null) {
          return true;
      }

      // se entrambi gli alberi non sono vuoti e i valori nei nodi radice corrispondono,
      // uso la ricorsione per i loro sottoalberi destro e sinistro
      return (x != null && y != null) && (x.value.equals(y.value)) &&
                      isIdentical(x.left, y.left) &&
                      isIdentical(x.right, y.right);
  }
}
```

Esercizio 15

Dati due arrays di interi di dimensione m, implementare un algoritmo che in modo efficiente stampi tutti gli elementi contenuti nei due arrays in ordine crescente. Calcolarne la complessità.

Esempio: [1,4,5] AND [2,10,6]
Output: 1,2,4,5,6,10

Esercizio 15

```
class Main{

 public static void stampaOrdinata(int a[], int b[]){
   PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
   int m = a.length;

   for (int i = 0; i < m; i++){
       pq.add(a[i]);
       pq.add(b[i]);
   }

   while (!pq.isEmpty()){
           System.out.println(pq.poll());
   }
 }
}
```

Esercizio 16

Data una stringa s, implementare un algoritmo che inverta i caratteri della stringa, che completi l'esecuzione in tempo sublineare.

Esempio: Reverse me
Output: em esreveR

Esercizio 16

```
class Main
{

  public static void reverse(char[] chars, int i, int j){
      if (i < j){
          // scambio i caratteri nelle posizioni i e j
          swap(chars, i, j);
          // uso la ricorsione incrementando i e j
          reverse(chars, i + 1, j - 1);
      }
  }

  // Metodo wrapper
  public static String reverse(String str){
      char[] chars = str.toCharArray();
      reverse(chars, 0, str.length() - 1);
      return new String(chars);
  }

  // scambia i cratteri i e j nella stringa
  private static void swap(char[] A, int i, int j) {
      char ch = A[i];
      A[i] = A[j];
      A[j] = ch;
  }

}
```