The Imitation Game

1) Leggere degli interi da un file
2) Ordinare i numeri
3) Calcolare le occorrenze per ogni numero
4) Stampare le occorrente

Linguaggi:

- C
- Commodore Basic 2.0
- Turbo Pascal 3
- QBasic
- Java 1.0, 8.0
- Groovy 2.3
- JRuby 9.0.0.0
- Kotlin 1.0, 1.1
- Scala 2.12

Il codice esposto ha il solo scopo dimostrativo.

C (1973)

```c
#include <stdio.h>
#include <stdlib.h>

#define MAXSIZE  100
#define MAXVAL   100
#define LINESIZE 200

typedef int elem;
typedef int index;
typedef int vector[MAXSIZE];
typedef int occType[MAXVAL];

void loadData(char *filename, vector numbers, index *pTot);
index partition(vector arr, index left, index right);
void quickSort(vector arr, index left, index right);
void countOccurrences(vector numbers, index tot, occType occurrences
void printOccurrences(occType occurrences);

int main(int argc, char *argv[]) {
        vector numbers;
        index nval;
        occType occurrences;

        loadData("source", numbers, &nval);
        quickSort(numbers, 0, nval-1);
        countOccurrences(numbers, nval, occurrences);
        printOccurrences(occurrences);
}

void loadData(char *filename, vector numbers, index *pTot) {
        FILE *fp;
        char line[LINESIZE];

        fp = fopen(filename,"r");
        *pTot = 0;
        while (fgets(line, LINESIZE, fp) != NULL)
                numbers[(*pTot)++] = atoi(line);
        fclose(fp);
}

void quickSort(vector arr, index left, index right) {
        index index = partition(arr, left, right);
        if (left < index - 1)
            quickSort(arr, left, index - 1);
        if (index < right)
            quickSort(arr, index, right);
}

index partition(vector arr, index left, index right) {
        index i = left, j = right;
        elem pivot = arr[(left + right) / 2];

        while (i <= j) {
            while (arr[i] < pivot)
                i++;

            while (arr[j] > pivot)
                j--;

            if (i <= j) {
                elem tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
                i++;
                j--;
            }
        }
        return i;
}

void countOccurrences(vector numbers, index tot, occType occurrences) {
        index i;
        for (i=0; i<MAXVAL; ++i) occurrences[i] = 0;
        for (i=0; i<tot; ++i) occurrences[numbers[i]]++;
}

void printOccurrences(occType occurrences) {
        elem i;
        for (i=0; i<MAXVAL; ++i)
                if (occurrences[i] > 0)
                        printf("Value: %d, occurrences: %d\n", i, occurrences[i]);
}
```

Commodore Basic 2.0 (1977)

```
10 GOSUB 1000:REM READ FILE
20 GOSUB 2000:REM SORT ARRAY
30 GOSUB 3000:REM COUNT OCCURRENCES
40 GOSUB 4000:REM PRINT OCCURRENCES
999 END
1000 :
1010 REM READ FILE
1020 T=0:DIM V(100)
1030 OPEN 1,8,2,"SOURCE,S,R"
1040 IF ST AND 64 THEN 1090
1050 INPUT#1,X
1060 V(T)=X
1070 T=T+1
1080 GOTO 1040
1090 CLOSE 1
1100 RETURN
2000 :
2010 REM SORT ARRAY
2020 FOR I=0 TO T-1
2030 FOR J=I TO T-1
2040 IF V(I)>V(J) THEN S=V(I):V(I)=V(J):V(J)=S
2050 NEXT
2060 NEXT
2070 RETURN
3000 :
3010 REM COUNT OCCURRENCES
3020 DIM C(100)
3030 FOR I=0 TO T-1
3040 C(V(I))=C(V(I))+1
3050 NEXT
3060 RETURN
4000 :
4010 REM PRINT OCCURRENCES
4020 FOR I=0 TO 100
4030 IF C(I)>0 THEN PRINT"VAL=";I;"  COUNT=";C(I)
4040 NEXT
4050 RETURN
```

Turbo Pascal 3 (1986)

```
PROGRAM JUG201707;

CONST MAXSIZE = 100;
      MAXVAL = 100;

TYPE elem    = 0..MAXVAL;
     index   = 0..MAXSIZE;
     vector  = ARRAY [1..MAXSIZE] of elem;
     occType = ARRAY [elem] of integer;
     fname   = STRING[20];

VAR numbers     : vector;
    nval        : index;
    occurrences : occType;

PROCEDURE LoadData(filename:fname; VAR numbers:vector; VAR tot:index);
VAR f:TEXT;
BEGIN
  Assign(f, filename);
  Reset(f);
  tot := 0;
  WHILE NOT Eof(f) DO BEGIN
    tot := tot+1;
    READLN(f, numbers[tot]);
  END;
  Close(f)
END;

PROCEDURE Partition(VAR numbers:vector; l,u:index; VAR j,i:index);
VAR t,pivot    : elem;
    pivotIndex : index;
BEGIN
  pivotIndex := (l+u) div 2;
  pivot := numbers[pivotIndex];
  i := l;
  j := u;
  WHILE numbers[i] < pivot DO i := i+1;
  WHILE numbers[j] > pivot DO j := j-1;
  WHILE i < j-1 DO BEGIN
    t := numbers[i];
    numbers[i] := numbers[j];
    numbers[j] := t;
    i := i+1;
    j := j-1;
    WHILE numbers[i] < pivot DO i := i+1;
    WHILE numbers[j] > pivot DO j := j-1;
  END;
  IF i <= j THEN BEGIN
    IF i < j THEN BEGIN
      t := numbers[i];
      numbers[i] := numbers[j];
      numbers[j] := t;
    END;
    i := i+1;
    j := j-1;
  END;
END;

PROCEDURE QuickSort(VAR numbers:vector; low,high:index);
VAR lower,higher : index;
BEGIN
  IF low < high THEN BEGIN
    Partition(numbers,low,high,lower,higher);
    IF (lower-low) < (high-higher) THEN BEGIN
      QuickSort(numbers,low,lower);
      QuickSort(numbers,higher,high)
    END ELSE BEGIN
      QuickSort(numbers,higher,high);
      QuickSort(numbers,low,lower)
    END
  END
END;

PROCEDURE CountOccurrences(VAR numbers:vector; tot:index; VAR occurrences:occType);
VAR i:elem;
BEGIN
  FOR i := 1 TO tot DO occurrences[numbers[i]] := occurrences[numbers[i]]+1
END;

PROCEDURE PrintOccurrences(VAR occurrences:occType);
VAR i:elem;
BEGIN
  FOR i := 0 TO MAXVAL DO
    IF occurrences[i] > 0 THEN
      WriteLn('Value: ',i,', Occurrences: ',occurrences[i])
END;

BEGIN
  LoadData('source', numbers, nval);
  QuickSort(numbers, 1, nval);
  CountOccurrences(numbers, nval, occurrences);
  PrintOccurrences(occurrences);
END.
```

QBasic (1991)

```
OPTION BASE 0
DIM numbers(100) AS INTEGER
DIM tot AS INTEGER
DIM occurrences(100) AS INTEGER

DECLARE SUB LoadData (filename AS STRING, numbers() AS INTEGER, t AS INTEGER)
DECLARE SUB QuickSort (numbers() AS INTEGER, low AS INTEGER, high AS INTEGER)
DECLARE SUB Partition (numbers() AS INTEGER, low AS INTEGER, up AS INTEGER, j AS INTEGER, i AS INTE
DECLARE SUB CountOccurrences (numbers() AS INTEGER, t AS INTEGER, occurrences() AS INTEGER)
DECLARE SUB PrintOccurrences (occurrences() AS INTEGER)

LoadData "SOURCE", numbers(), tot
QuickSort numbers(), 0, tot - 1
CountOccurrences numbers(), tot, occurrences()
PrintOccurrences occurrences()

SUB LoadData (filename AS STRING, numbers() AS INTEGER, t AS INTEGER)
  f = FREEFILE
  OPEN filename FOR INPUT AS f
  t = 0
  DO WHILE NOT EOF(f)
    INPUT #f, x
    numbers(t) = x
    t = t + 1
  LOOP
  CLOSE f
END SUB

SUB Partition (numbers() AS INTEGER, low AS INTEGER, up AS INTEGER, j AS INTEGER, i AS INTEGER)
  DIM pivot AS INTEGER
  DIM pivotIndex AS INTEGER

  pivotIndex = (low + up) / 2
  pivot = numbers(pivotIndex)
  i = low
  j = up
  DO WHILE numbers(i) < pivot
    i = i + 1
  LOOP
  DO WHILE numbers(j) > pivot
    j = j - 1
  LOOP
  DO WHILE i < j - 1
    SWAP numbers(i), numbers(j)
    i = i + 1
    j = j - 1
    DO WHILE numbers(i) < pivot
      i = i + 1
    LOOP
    DO WHILE numbers(j) > pivot
      j = j - 1
    LOOP
  LOOP
  IF i <= j THEN
    IF i < j THEN SWAP numbers(i), numbers(j)
    i = i + 1
    j = j - 1
  END IF
END SUB

SUB QuickSort (numbers() AS INTEGER, low AS INTEGER, high AS INTEGER)
  DIM lower AS INTEGER
  DIM higher AS INTEGER
  IF (low < high) THEN
    Partition numbers(), low, high, lower, higher
    IF (lower - low) < (high - higher) THEN
      QuickSort numbers(), low, lower
      QuickSort numbers(), higher, high
    ELSE
      QuickSort numbers(), higher, high
      QuickSort numbers(), low, lower
    END IF
  END IF
END SUB

SUB CountOccurrences (numbers() AS INTEGER, t AS INTEGER, occurrences() AS INTEGER)
  DIM i AS INTEGER
  FOR i = 0 TO t - 1
    occurrences(numbers(i)) = occurrences(numbers(i)) + 1
  NEXT
END SUB

SUB PrintOccurrences (occurrences() AS INTEGER)
  DIM i AS INTEGER
  FOR i = LBOUND(occurrences) TO UBOUND(occurrences)
    IF occurrences(i) > 0 THEN PRINT "N."; i, "OCCURRENCES ="; occurrences(i)
  NEXT
END SUB

```

Java 1.0 (1995)

```java
public class Main {

    public static void main(String[] args) throws IOException {
        int[] numbers = loadFile("../source");

        quicksort(numbers, 0, numbers.length - 1);

        Dictionary occurrences = occurrences(numbers);

        Enumeration enumeration = occurrences.keys();
        while (enumeration.hasMoreElements()) {
            Object key = enumeration.nextElement();
            Object value = occurrences.get(key);
            System.out.println(key + "=" + value);
        }
    }

    private static int[] loadFile(String path) throws IOException {
        Vector intVector = new Vector();
        StringBuffer stringBuffer = new StringBuffer();
        InputStream inputStream = new FileInputStream(new File(path));
        while (inputStream.available() > 0) {
            int b = inputStream.read();
            if (b == '\n') {
                if (stringBuffer.length() > 0) {
                    intVector.addElement(stringBuffer.toString());
                }
                stringBuffer.setLength(0);
            } else {
                stringBuffer.append((char) b);
            }
        }

        int[] ints = new int[intVector.size()];
        for (int i = 0, max = ints.length; i < max; i++) {
            ints[i] = Integer.parseInt((String) intVector.elementAt(i));
        }
        return ints;
    }

    private static void quicksort(int arr[], int left, int right) {
        int index = partition(arr, left, right);
        if (left < index - 1)
            quicksort(arr, left, index - 1);
        if (index < right)
            quicksort(arr, index, right);
    }

    private static int partition(int arr[], int left, int right) {
        int i = left, j = right;
        int pivot = arr[(left + right) / 2];

        while (i <= j) {
            while (arr[i] < pivot)
                i++;

            while (arr[j] > pivot)
                j--;

            if (i <= j) {
                int tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
                i++;
                j--;
            }
        }
        return i;
    }

    private static Dictionary occurrences(int[] orderedNumbers) {
        Dictionary occurrences = new Hashtable();
        if (orderedNumbers.length > 0) {
            int i = 0;
            while (i < orderedNumbers.length) {
                int number = orderedNumbers[i];
                int count = 1;
                while (++i < orderedNumbers.length && number == orderedNumbers[i]) {
                    count++;
                }
                occurrences.put(new Integer(number), new Integer(count));
            }
        }
        return occurrences;
    }
}
```

Java 8 (2014)

```
14  public class Main {
15
16      public static void main(String... args) throws IOException {
17          final List<Integer> numbers = Files.lines( FileSystems.getDefault().getPath( "..", "source" ) )
18                  .map( Integer::parseInt )
19                  .collect( toList() );
20
21          final List<Integer> sorted = quicksort(numbers);
22
23          final Map<Integer, Long> occurrences = sorted.stream().collect( groupingBy( Function.identity(), counting() ) );
24
25          occurrences.entrySet().stream()
26                  .map( e -> format( "%d=%d", e.getKey(), e.getValue() ) )
27                  .forEach( System.out::println );
28      }
29
30      public static List<Integer> quicksort(List<Integer> numbers) {
31          if (numbers.size() < 2) { return numbers; }
32
33          final int pivot = numbers.get(0);
34          final Map<Boolean, List<Integer>> smallerThanPivot = numbers.subList(1, numbers.size()).stream()
35                  .collect( partitioningBy( x -> x <= pivot ) );
36
37          final List<Integer> result = new ArrayList<>(numbers.size());
38              result.addAll( quicksort( smallerThanPivot.get(true) ) );
39              result.add(pivot);
40              result.addAll( quicksort( smallerThanPivot.get(false) ) );
41
42          return result;
43          }
44  }
```

Groovy 2.3 (2014)

```
5   @groovy.transform.CompileStatic
6   <T extends Comparable<T>> List<T> quicksort(List<T> list) {
7       if (list.size() < 2) return list
8       def pivot = list[0]
9       def items = list.groupBy { it <=> pivot }.withDefault { [] }
10      quicksort(items[-1]) + items[0] + quicksort(items[1])
11  }
12
13  def numbers = Files.lines(FileSystems.getDefault().getPath("..", "source"))
14          .map { line -> Integer.valueOf(line) }
15          .collect(Collectors.toList())
16
17  def ordered = quicksort(numbers)
18
19  def occurrences = new LinkedHashMap().withDefault { 0 }
20  ordered.forEach { i ->
21      occurrences.put(i, 1 + occurrences[i])
22  }
23
24  occurrences.forEach { key, value ->
25      println(key + "=" + value)
26  }
```

JRuby 9.0.0.0 (2015)

```ruby
def quicksort(l)
  return l if (l.size < 2)
  p = l.shift
  (quicksort(l.select {|n| n < p}) << p << quicksort(l.select {|n| n >= p})).flatten
end

numbers = File.readlines('../source').map(&:to_i)
ordered = quicksort(numbers)
occurrences = ordered.group_by(&:itself).map {|k, v| {k => v.size}}.reduce(:merge)

puts occurrences.map {|n, o| "#{n}=#{o}"}
```

Kotlin 1.0 (2016)

```
3  fun main(vararg args: String) {
4      val numbers = File("../source")
5              .readLines()
6              .map { it.toInt() }
7
8      val ordered = numbers.quicksorted()
9
10     val occurrences = ordered
11             .fold(emptyMap<Int, Int>()) { acc, i ->
12                 val count = 1 + (acc[i] ?: 0)
13                 acc + mapOf(i to count)
14             }
15
16     occurrences.forEach(::println)
17 }
18
19 fun <T : Comparable<T>> List<T>.quicksorted(): List<T> =
20         when {
21             size < 2 -> this
22             else -> first().let { pivot ->
23                 val (smaller, greater) = drop(1).partition { it <= pivot }
24                 smaller.quicksorted() + pivot + greater.quicksorted()
25             }
26         }
```

Scala 2.12 (2016)

```
4  object Main extends App {
5
6    implicit class ExtendedList[T](t: List[T]) {
7
8      def occurrences: Map[T, Int] =
9        t.foldLeft(empty[T, Int]) { (a, v) => a + a.get(v).fold(v -> 1)(x => v -> (x + 1)) }
10
11     def quickSort(implicit f: T => Ordered[T]): List[T] = t match {
12       case Nil => Nil
13       case head :: tail =>
14         val (lower, upper) = tail.partition(_ < head)
15         lower.quickSort ++ List(head) ++ upper.quickSort
16     }
17   }
18
19   fromResource("source").getLines.map(_.toInt).toList
20     .quickSort
21     .occurrences
22     .foreach(x => println(s"${x._1}=${x._2}"))
23 }
```

Kotlin 1.1 (2017)

```
10  fun main(vararg args: String) = runBlocking {
11      val numbers = File("../source")
12              .readLines()
13              .map { it.toInt() }
14
15      val ordered = quicksortAsync(numbers).await()
16
17      val occurrences = ordered.stream()
18              .collect(groupingBy(identity<Int>(), counting()))
19
20      occurrences.forEach(::println)
21  }
22
23  suspend fun <T : Comparable<T>> quicksortAsync(list: List<T>): Deferred<List<T>> = async(CommonPool){
24      val pivot = list.firstOrNull() ?: return@async list
25      val (smaller, greater) = list.drop(1)
26              .partition { it <= pivot }.toList()
27              .map { quicksortAsync(it) }
28      return@async smaller.await() + pivot + greater.await()
29  }
```