Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Institut für Theoretische Informatik
Peter Widmayer
Thomas Tschager
Antonis Thomas

16th March 2016

Datenstrukturen & Algorithmen Programming Exercise 3 FS 16

The exercise was to implement *Blum's algorithm* for median computation. We will describe a step by step implementation. Two class-wide variables are used: the variable n stores the number of elements while the elements itself are stored in the array v. The task is to find the median of the array v, i.e. the $\lceil n/2 \rceil$-th element in the sorted sequence.

```
static int n;
static int v[];
```

Furthermore, we define an auxiliary function `swap` that interchanges two elements in the array v.

```
static void swap(int i, int j) {
    int aux = v[i];
    v[i] = v[j];
    v[j] = aux;
}
```

Computing the median in each group

In the first step, the algorithm divides the elements into $\lfloor n/5 \rfloor$ groups of five elements and at most one group with $n \bmod 5$ elements. Now we need to find the median in each group. This is done by the function `posMedian5` that takes two parameters: a left index l (called `indexFirst` below) and a right index r (called `indexLast` below). Note that we set $r = l + 4$, except for the last group. The implementation performs an elementary sorting algorithm (here: bubble sort) to sort $v[l..r]$, and after sorting it returns the position of the median, which is $\lfloor (l + r)/2 \rfloor$.

```
static int posMedian5(int indexFirst, int indexLast) {
    int i, rep;
    /* Use bubble sort */
    for(rep = 0; rep < 4; rep++)
        for(i = indexFirst; i < indexLast-rep; i++)
            if(v[i] > v[i+1]) swap(i, i+1);
    /* Median is located at the middle position */
    return (indexFirst+indexLast)/2;
}
```

The partition function

We implement the partition function of quickselect in the following way. Once a pivot element p has been chosen, we interchange it with the last element of the sequence to be sorted. After that, we use the variable i to iterate from `left` to `right` $- 1$, and consider the element $v[i]$ in each iteration. We furthermore use a variable `storePos` with the property that $v[\texttt{left}..(\texttt{storePos}) - 1]$ contains all the elements that are smaller than p while $v[\texttt{storePos}..(i - 1)]$ stores only elements that have a value of at least p. If $v[i]$ is smaller than the pivot, it is swapped with $v[$`storePos`$]$ and `storePos` is incremented. At the end, the pivot element p on the right is swapped with $v[$`storePos`$]$ (which contains an element having a value of at least p). Then p is on the correct position: every element on the left side is smaller, while every element on the right side has a value of at least p.

For example, consider the array $[1, 9, 6, 5, 3, 7, 4]$ and use 5 as pivot element. First we swap the pivot with the element in the rightmost position, which results in $[1, 9, 6, 4, 3, 7, 5]$. Then, we iterate from the left to the right: we check if each element is smaller than the pivot, then we move it to the left side. After we moved all elements that are smaller than the pivot we obtain the array $[1, 4, 3, 9, 6, 7, 5]$. Finally we move the pivot element to the position after the last element being smaller than the pivot and obtain the array $[1, 4, 3, 5, 6, 7, 9]$.

```
static int partition(int left, int right, int pivotPos) {
    int storeIndex = left, i;
    int pivotVal = v[pivotPos];
    swap(pivotPos, right); // Move pivot to the end
    // Move element less than pivot at the beginning
    for(i = left; i < right; i++)
      if(v[i] < pivotVal)
          // Swap v[i] with v[storeIndex]
          swap(i, storeIndex++);
    // Move pivot on it's final position
    swap(storeIndex, right);
    return storeIndex; // Return new pivot position
}
```

Blum's algorithm

We can now implement the main algorithm to determine the position of the element which is located at the position k in the sorted sequence. Note that the first position has the index 0 while the last one has the index $n - 1$. Thus, to compute the median of the sequence, we have to use $k = \lfloor (n-1)/2 \rfloor$. The algorithm is implemented in function `medianPos`. Besides k, it takes three more parameters: `left` and `right` determine the left and the right indices of the sequence, while `display` is a boolean variable that determines whether the algorithm produces an output or not. Since an output is needed only in the first step of the algorithm, the variable `display` is initially set to `true` but it is set to `false` in every subsequent recursive call of the algorithm.

As previously stated, the array is divided into $\lceil n/5 \rceil$ groups. We sort the elements within every group to compute the medians $M_1, ..., M_{\lceil n/5 \rceil}$. After that, we move these medians to the beginning of $v[$`left`..`right`$]$, i.e. the content of $v[$`left` $+ i - 1]$ is swapped with the median M_i of the i-th group, $1 \leq i \leq \lceil n/5 \rceil$. Since the medians are now stored in a continuous subarray, we can recursively compute the *median-of-medians*, which is the median of the subarray $v[$`left`..$($`left` $+ \lceil n/5 \rceil - 1)]$. If the number of medians is at most 5, then the previously defined function `posMedian5` is used instead of Blum's algorithm to compute the median.

After this recursive call, we have computed the position of the median-of-medians. This element is used as the pivot element for the partition step. If k is exactly the position of the pivot element *after* the partition step, we are finished (since every element on the left has a smaller value than the pivot element while every element on the right has at least this value). If the pivot position is greater than k, we recurse on the left side of the pivot, otherwise on the right side.

```
static int medianPos(int first, int last, int k, boolean display) {
    int i, j, posMedian, pivotPos, nMedians = 1 + (last - first)/5;
    // Find medians of groups of 5
    for(i = first, j = 0; i <= last; i += 5, j++) {
        // Last group may have less than 5 elements
        posMedian = posMedian5(i, Math.min(i+4, last));

        // Display median of each group of 5 elements
        if(display) System.out.print(v[posMedian] + " ");

        // Store medians in the beginning of the array
        swap(first + j, posMedian);
    }
    // Find median of medians recursively
    // If we have less than 5 medians we can use posMedian5 to find the median
    if(nMedians > 5)
        pivotPos = medianPos(first, first+nMedians-1, first+(nMedians-1)/2, false);
    else
        pivotPos = posMedian5(first, first+nMedians-1);

    // Display median of medians
    if(display) System.out.print(v[pivotPos] + " ");

    // Partition with the median of medians as pivot
    pivotPos = partition(first, last, pivotPos);

    // Recurse on the left or right side of the pivot if necessary
    if(pivotPos == k) return pivotPos;
    else if(pivotPos > k) return medianPos(first, pivotPos-1, k, false);
    else return medianPos(pivotPos+1, last, k, false);
}
```

Main program

```
import java.util.Scanner;

public class Main2 {

    static int n;
    static int v[];

    static void swap(int i, int j){...}
    static int posMedian5(int indexFirst, int indexLast) {...}
    static int partition(int left, int right, int pivotPos) {...}
    static int medianPos(int first, int last, int k, boolean display) {...}

    public static void main(String[] args) {
        int test, ntest, i;
        Scanner sc = new Scanner(System.in);
        ntest = sc.nextInt();
        for(test = 1; test <= ntest; test++) {
            n = sc.nextInt();
            v = new int[n];
            for(i = 0; i < n; ++i)
                v[i] = sc.nextInt();

            // Compute the position of the median
            int posMedian = medianPos(0, n-1, (n-1)/2, true);
            System.out.println(v[posMedian]);
        }
    }

}
```