# Programming Assignment 2 (Binary Search Applications, Selection Sort, and Insertion Sort)

Department of Computer Science, University of Wisconsin – Whitewater
Data Structure (CS 223)

## 1 Part 1: Applications of Binary Search

Complete the functions in the BinarySearchApplications.cpp/BinarySearchApplications.java file.

Details of the functions to be implemented are provided in the following subsections.

Following is the expected output.

```
*** Counting the number of occurrences of key ***

Number of occurrences of 1 is 2
Number of occurrences of 14 is 6
Number of occurrences of 39 is 3
Number of occurrences of 7 is 1
Number of occurrences of 100 is 0
Number of occurrences of -88 is 0
Number of occurrences of 16 is 0

*** Finding Predecessor ***

Predecessor of 1 is 1
Predecessor of 0 is not defined.
Predecessor of 39 is 39
Predecessor of 47 is 39
Predecessor of 36 is 27
Predecessor of 12 is 9
Predecessor of 6 is 3
```

### 1.1 Counting the number of occurrences of key in a sorted array

In certain applications, we are interested in counting the number of times *key* appears in a sorted array. The technique to solve such problems is to determine:

- minIndex: the minimum index where *key* appears
- maxIndex: the maximum index where *key* appears

The number of occurrences is given by $(maxIndex - minIndex + 1)$.

Hence, our task is to find both the minimum and maximum positions where a key occurs. We seek to solve this using **Binary Search**. To this end, complete the following functions:

- **int** minIndexBinarySearch(**int** array[ ], **int** arrayLength, int key): returns the minimum index where *key* appears. If *key* does not appear, then returns $-1$.
- **int** maxIndexBinarySearch(**int** array[ ], **int** arrayLength, int key): returns the maximum index where *key* appears. If *key* does not appear, then returns $-1$.
- **int** countNumberOfKeys(**int** array[ ], **int** arrayLength, int key): Returns 0 if *key* is not the in the array, else it returns the number of occurrences of *key*.

**Caution:** Your code should have complexity $O(\log n)$, where $n = arrayLength$. If your code ends up scanning the entire array (has a complexity $O(n)$), you will be awarded partial credit, even if you get the correct output.

**Algorithm for finding the minimum index**

- The main idea is to use binary search with a slight modification. Declare a variable called $minIndex$ along with $left$ and $right$. Initially, $minIndex = -1$.
- Now, when you find *key* at index $mid$, do not return mid, but set $minIndex = mid$, $right = mid - 1$, and continue.
- Finally, after the while loop expires return $minIndex$ (instead of $-1$).

**Algorithm for finding the maximum index**

Use a variable $maxIndex$ (instead of $minIndex$). Algorithm remains the same as above, just that when you find *key* at $mid$, we set $maxIndex = mid$ and $left = mid + 1$. Finally, return $maxIndex$.

**Algorithm to count number of occurrences of *key***

Use the above two algorithms to get $minIndex$ and $maxIndex$. Then, use the formula to count and return the number of occurrences.

### 1.2 The Predecessor Problem

Given a set of numbers, the predecessor of a number $x$ is the highest number in the set that is less than or equal to $x$. Thus, if I have the set $\{6, 9, 10, 13, 22, 31, 34, 88\}$, then the predecessor of 31 is 31 itself, whereas the predecessor of 30 is 22, and the predecessor of 5 is not defined.

The predecessor problem has remarkable applications in *network routing*, where we send information from one computer to another, making email and other uses of the internet possible. Another application is *nearest-neighbor search*, akin to locating restaurants on Google Maps, where it returns the closest match to a cuisine of your choice.

Our task is to find predecessor of a number in an array using a **Binary Search approach**. To this end, complete the following function:

- **int** findPredecessor(**int** A[ ], **int** arrayLen, **int** key): returns a position in the array $A$ where the predecessor of *key* lies. Needless to say that the array $A$ is sorted in ascending order. If the predecessor of *key* is not defined, return $-1$.

**Caution:** You MUST use a binary search approach. Thus, the complexity should be $O(\log n)$. If your code ends up scanning the entire array (has a complexity $O(n)$), you will be awarded partial credit, even if your code is correct.

**Algorithm for finding the predecessor index**

- The main idea is to use binary search with a slight modification. Declare a variable called $predIndex$ along with $left$ and $right$. Initially, $predIndex = -1$.
- Now, when $A[mid] < key$, then $mid$ is a better estimate of your predecessor index; so, set $pred = mid$, $left = mid + 1$, and continue. Rest remains unchanged withing the while loop.
- Finally, after the while loop expires return $predIndex$ (instead of $-1$).

## 2 Part 2: Selection Sort and Insertion Sort

Complete the *selectionSort* and *insertionSort* functions of the Sorting.cpp/Sorting.java file. If your code is correct, you should get the following output.

```
Original Array: [13, 17, 8, 14, 1]
Selection Sorted Array: [1, 8, 13, 14, 17]

Original Array: [-13, -17, -8, -14, -1, -20]
Insertion Sorted Array: [-20, -17, -14, -13, -8, -1]
```