**CS102 - Algorithms and Programming II**

**Lab Programming Assignment 6**

**Fall 2021**

| **ATTENTION:**   1. Compress all Java program source files (.java) into a single zip file. 2. The name of the zip file should follow the below convention:   **CS102\_Sec1\_Asgn6\_YourSurname\_YourName.zip**   1. Replace the variables “YourSurname” and “YourName” with your actual surname and name. 2. You may ask questions on Moodle. 3. Complete **all** of the assignment during the lab session and upload the above zip file to Moodle by the deadline. |
| --- |

**The work must be done individually. Codesharing is strictly forbidden. We are using sophisticated tools to check the code similarities. The** [**Honor Code**](https://docs.google.com/document/d/1v_3ltpV_lClLsROXrMbojyuv4KrFQAm1uoz3SdC-7es/edit) **specifies what you can and cannot do. Breaking the rules will result in disciplinary action.**

**You are allowed to use the skeleton code we have given to you.**

**The size of the lab document shall not frighten you. The task is short, but we have provided a lot of explanations.**

**Q1 Searching [50 pts.].**

In this part of the lab, you will extend the Binary Search algorithm, a well-known divide-and-conquer algorithm. Each time you divide the input into two and recurse on one of the halves. Apparently, this approach only halves the problem. A simple extension of this algorithm called Ternary Search divides the problem into three pieces. By following a similar pattern, we want you to create a **recursive** Generic Search algorithm that can divide the problem into **k** parts each time and recurse into appropriate sub-problem at each step.

To begin with, we shall now give you the pseudo-codes for Binary and Ternary Search algorithms. You will assume that the input array is sorted in increasing order. In addition, both methods return the index of the target value if it is present in the array. Otherwise, you should return -1.

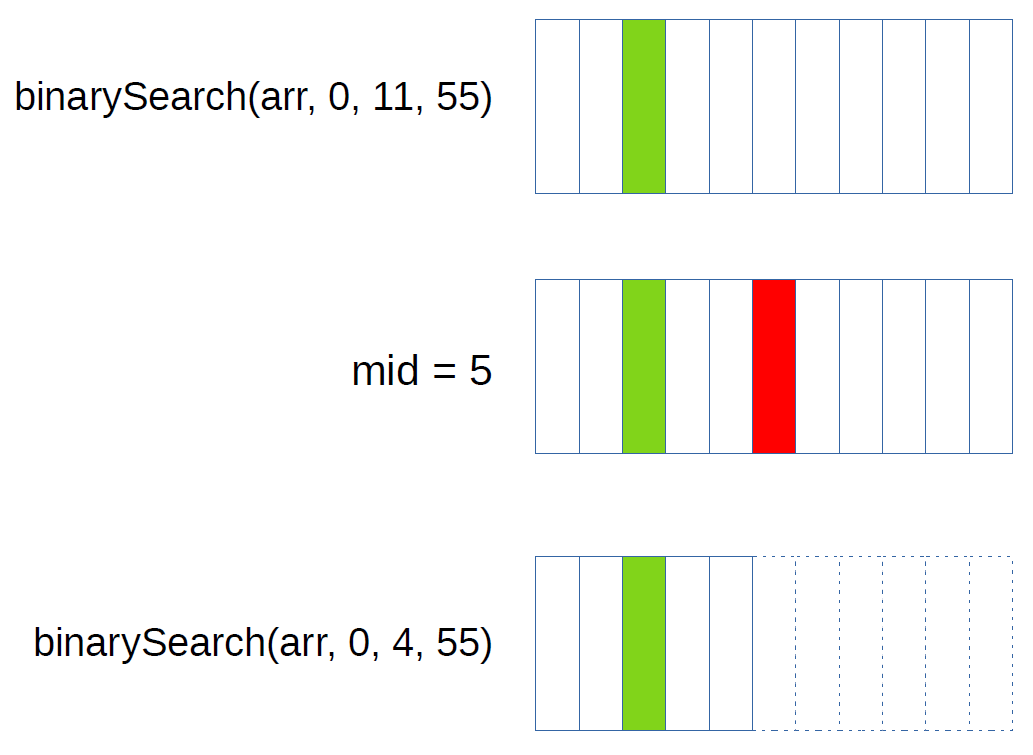
Binary Search Algorithm (It is a pseudo-code, not a Java code):

| int binarySearch(int[] arr, int low, int high, int target):  if low > high: return -1  mid = low + (high - low) / 2  if arr[mid] == target: return mid  if target < arr[mid]: return “binary search to left”  return “binary search to right” |
| --- |

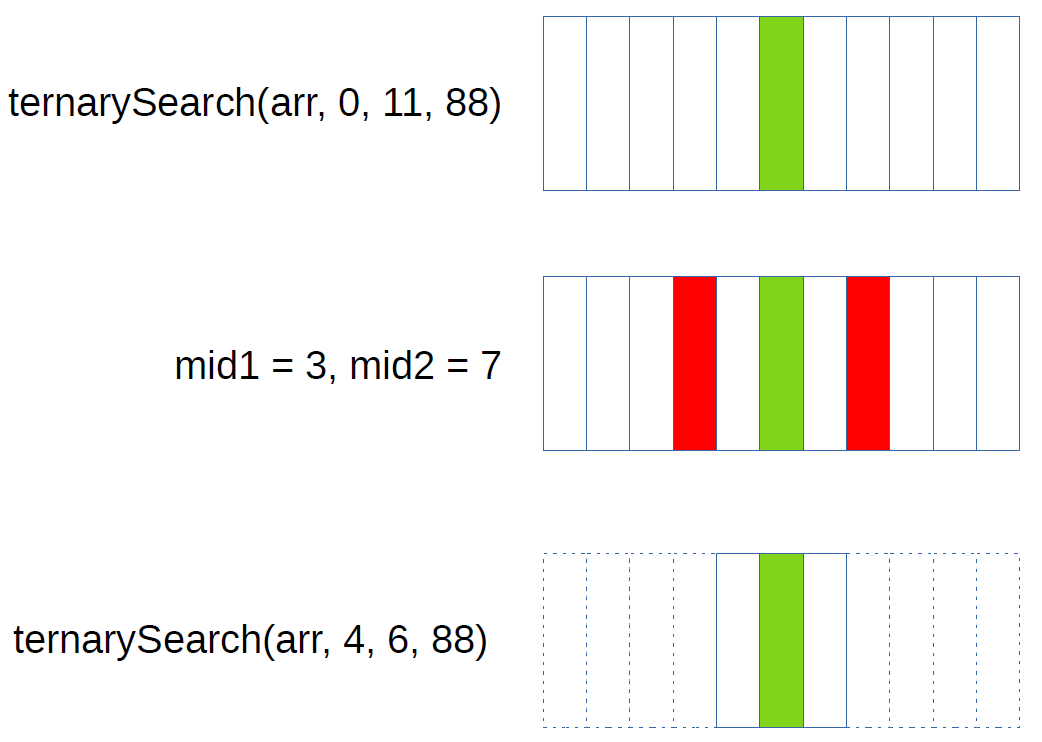
Ternary Search Algorithm (It is a pseudo-code, not a Java code):

| int ternarySearch(int[] arr, int low, int high, int target):  if low > high: return -1  mid1 = low + (high - low) / 3  mid2 = low + 2 \* (high - low) / 3  if arr[mid1] == target: return mid1  if arr[mid2] == target: return mid2  if target < arr[mid1]: return “ternary search to left”  if target < arr[mid2]: return “ternary search to middle”  return “ternary search to right” |
| --- |

To hopefully make it clear for you, we have figures briefly explaining the both algorithms. Figure 1 and 2 shows a single step during the execution of each algorithm. In Figure 1, assume that binarySearch(arr, 0, 11, 55) is executed. We calculate the mid value using the calculation in the pseudo-code. Now, we know that the green block is to the left of the red block, so we need to recurse to the left, which is what’s happening in the last step of Figure 1. Since we guarantee that the target is on the left, we can safely ignore the right sub-problem. In Figure 2, we have a similar approach. This time we need to calculate 2 midpoints (mid1 and mid2) to divide the input into 3 areas (it is where the name “ternary” comes from). Since the target block is between two midpoints, we need to recurse on that region, corresponding to “ternary search to middle” in the above pseudo-code.



*Figure 1: Illustration of the Binary Search algorithm. The green block is the target value, while the red is the middle point. Dashed blocks emphasize that we are no longer interested in those regions.*



*Figure 2: Illustration of the Ternary Search algorithm. The green block is the target value, while the red ones are the middle points. Dashed blocks emphasize that we are no longer interested in those regions.*

First, implement these two methods by using pseudo-codes. You will probably see the pattern there. Do **NOT** copy code from elsewhere. One more thing, you should create new custom Exception classes (i.e., InputNotSortedException, KIsLessThan2Exception), throw them inside your method and handle them in the main method. Then, please go ahead and implement the below method (assume that the input array is sorted in increasing order and **k** is greater than or equal to 2).

| int genericSearch(int[] arr, int low, int high, int target, **int k**) |
| --- |

After you have implemented the function and made sure that it is correct, please do some runtime experiments. That means you should measure the time to execute the genericSearch method. Conduct experiments with various **k** values and large array sizes. The motivation here is to find out which one of the **k** values seems to be faster. Experimenting with different array sizes will be helpful to observe some patterns.

**Q2 Sorting [50 pts.]**

In this part, you will implement a hybrid sorting method. This method will combine Quick Sort, Merge Sort, and Insertion Sort algorithms. You can use the Quick, Merge, and Insertion Sort codes we have given to you. Specifically, Quick and Merge Sort should be called in an alternated order. The problem size should be approximately halved each time (Note the quickSort cannot always divide the problem evenly due to the pivot). If the number of elements becomes less than 15, it should run Insertion Sort right away. *Note that it is not acceptable if your Insertion Sort sorts the whole array, it should only sort the sub-problem, nothing more*.

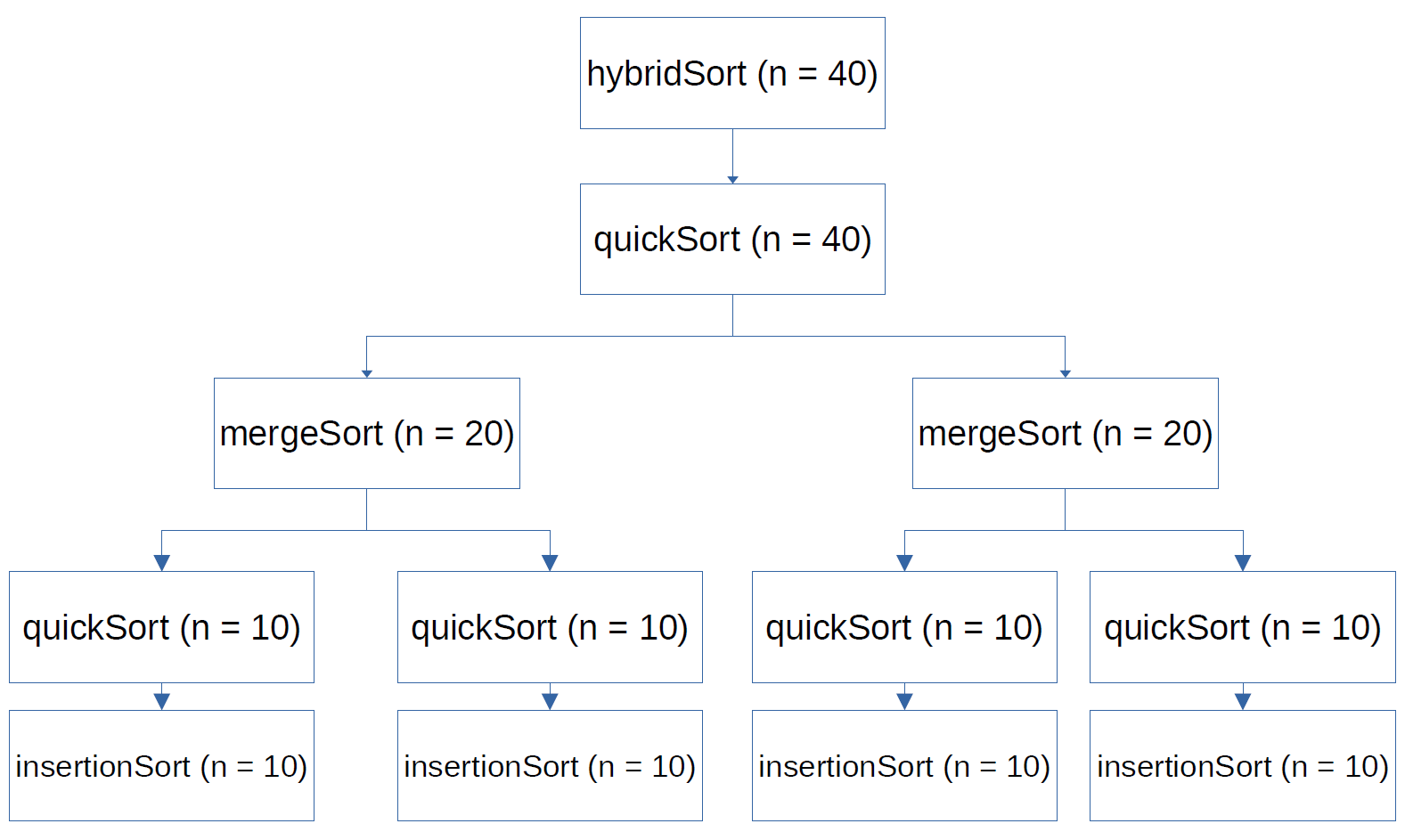
Let’s give you more detail. When your method is called, it should directly call Quick Sort. Afterward, inside Quick Sort, you should call Merge Sort, and similarly, inside the Merge Sort, you should call Quick Sort, and so on. This alternating execution should continue until the problem size is reduced to 15 elements, at which point, you should directly call Insertion Sort. To achieve this, you should modify the sorting codes we have provided. Think about a way to achieve such an execution pattern.

Let’s illustrate what you need to do once again. You will have the following hybridSort method that will call the quickSort method with appropriate parameters. You will modify quickSort, mergeSort, and insertionSort to have an execution scheme similar to Figure 3. That figure visualizes an example execution flow for an array of size 40. The problem size reduces as follows: 40 → 20 → 10. In the figure, we assume that quickSort divides the problem into two, but this is not necessarily correct all the time. That is not a problem if your quickSort does not partition the set evenly. Moreover, as the input size increases, that tree will be bigger and bigger. For instance, if the input size were 80, the problem size would diminish as: 80 → 40 → 20 → 10.

Please define the following method and make necessary modifications to quickSort, mergeSort, and insertionSort methods:

| void hybridSort(int[] arr):  quickSort(...) |
| --- |

After implementing the method and ensuring it is correct, please do some runtime experiments. That means you should measure the time it takes to execute the hybridSort, quickSort, and mergeSort methods. Conduct experiments with various large array sizes and compare the 3 sorting methods. The motivation here is to determine which one of the sorting methods seems to allow the fastest execution. Experimenting with different array sizes will be helpful to observe some patterns.



*Figure 3: Execution flow of hybridSort to sort a list with 40 elements. Note that the problem size is halved each time quickSort or mergeSort is called. In this execution, quickSort is assumed to partition the input into halves perfectly, but this may not always be correct.*

**NOTES**:

* Please reuse the available methods as much as possible instead of repeating the same code in different methods.
* Please comment your code according to the documentation and commenting conventions used in the textbook.