Programming Assignment 1

Department of Computer Science, University of Wisconsin – Whitewater
Theory of Algorithms (CS 433)

Instructions For Submissions

- **This assignment is to be completed individually. If you are stuck with something, consider asking the instructor for help.**
- **Submit code and a brief report**. Submission is via Canvas as a single zip file. No need to include the algorithm description in the report.
- **Any function with a compilation error will receive a zero, regardless of how much it has been completed.**

---

1 Task 1: Coding & Correctness [110 (+10 bonus) points]

The first purpose of this exercise is to compare the sorting algorithms you have learnt so far, i.e., Quick Sort and Radix Sort. You are going to implement quick sort using two pivoting strategies – one using a random pivot, and the other using the median of three as pivot. For radix sort, you can either implement Approach 1 or Approach 2 in notes; the latter gets you some bonus points.

Your task is to implement the following:

- **Partition Algorithm** (`File: Partition`)

  Implement the `partition(int left, int right, int pivot)` method.

  You must implement the in-place partition method discussed in class.

- **Quick Sort** (`File: QuickSort`)

  Implement the function `quicksortRandom(int left, int right)`. For pivot generation, you call the function `generateRandomPivot(int left, int right)`.

  Implement the function `generateMedianOf3Pivot(int left, int right)`. For pivot generation, you call the function `generateMedianOf3Pivot(int left, int right)`.

- **Radix Sort** (`File: RadixSort`)

  Implement the following three functions:

  - `countSortOnDigits(int A[], int n, int digits[])`: This is described in notes, where you use counting sort to sort the array A based on a particular digit.
  - `radixSortNonNeg(int A[], int n)`: This is again described in notes, where you use radix sort to sort the array A, which contains only non-negative numbers.

- `radixSort()`: This to radix sort an array which contains both non-negative as well as negative integers. You may use Approach 1 (in notes) to complete this function.

  An implementation of Approach 2 will be awarded 5 **bonus points**. Avoiding the creation of extra arrays for the negative and non-negative integers in Approach 2 will be rewarded 10 **bonus points**.

  Regardless of the approach, this method will call the previous `radixSortNonNeg` method.

In the second part, you are going to implement the Selection algorithm for finding the k^{th} smallest number in an array. For the pivot, you are going to use a random one. Then, you will compare with a Radix Sort based selection strategy (essentially, sort and then return $A[k-1]$).

Your task is to implement the following:

- **Quick-Selection Algorithm** (`File: Selection`)

  Implement the `select(int left, int right, int k)` method. For pivot generation, you call the function `generateRandomPivot(int left, int right)`.

In the last and final part, you will count the number of inversions in an array using a Merge Sort kind of approach. We are going to compare this against a brute-force approach.

Your task is to implement the following:

- **Inversion Counting** (`File: InversionCounting`)

  Implement the `countInversions(int left, int right)` method to count the number of inversions in an array in $O(n \log n)$ time. You may use the merge-sort code as a guideline. (The merge-sort code has been written using generics to sort points.)

1.1 C++ Helpful Hints

For C++ programmers, remember to use DYNAMIC ALLOCATION for declaring any and all arrays/objects. DO NOT forget to clear memory using *delete* (for objects) and *delete*[ ] for arrays when using dynamic allocation.

1.2 Correctness Test

Once you complete the code, use `TestCorrectness` to test the correctness. You should get the following output:

```
Original array:                 [19, 1, 12, 100, 7, 8, 4, -10, 14, -1, 97, -1009, 4210]
MergeSorted array:              [-1009, -10, -1, 1, 4, 7, 8, 12, 14, 19, 97, 100, 4210]
QuickSorted (median of 3) array: [-1009, -10, -1, 1, 4, 7, 8, 12, 14, 19, 97, 100, 4210]
QuickSorted (random) array:     [-1009, -10, -1, 1, 4, 7, 8, 12, 14, 19, 97, 100, 4210]
RadixSorted array:              [-1009, -10, -1, 1, 4, 7, 8, 12, 14, 19, 97, 100, 4210]

1th smallest: -1009
2th smallest: -10
3th smallest: -1
4th smallest: 1
5th smallest: 4
```

```
6th smallest: 7
7th smallest: 8
8th smallest: 12
9th smallest: 14
10th smallest: 19
11th smallest: 97
12th smallest: 100
13th smallest: 4210

Array is: [19, 1, 12, 100, 7, 8, 4, -10, 14, -1, 97, -1009, 4210]
Number of inversions is: 42
```

2 Task 2: Report (40 points)

You don't have to write code for this part. Submit your answers as a separate text file along with rest of your code (everything zipped together).

2.1 Part 1 (30 points)

Check the methods `findClosestPair` and `findClosestPairHelper` in the **ClosestPairOfPoints** file. There are 8 parts in the methods – Part 0 through Part 7.

Explain briefly what each part achieves. Please explain the purpose of each part in light of the algorithm and do not make statements like "this part calls this function with these parameters", or "in this part, we run a for-loop from $j = 1$ to $j = 7$", etc.

2.2 Part 2 (10 points)

Additionally, run `TestTime` to get a running time analysis. Your task is to analyze the second set of output, and **write a brief report on what you observe for the following:**

1. Rank the three sorting algorithms from fastest to slowest (running time-wise).
2. How does the randomized quick-select algorithm compare against a radix sort based selection (running time-wise)? Why do you think that there is a noticeable difference in running time, although both are linear-time algorithms?
3. How does brute-force inversion counting compare against the merge-sort approach (running time-wise)? What are their respective complexities?
4. How does brute-force closest-pair-of-points algorithm compare against the divide & conquer approach (running time-wise)? What are their respective complexities?

You must analyze the time output with respect to the $O(\cdot)$ complexities in each case. You must analyze in accordance to the programming language that you have chosen to implement.