Programming Assignment 1

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

Instructions For Submissions

- **Each group to have at most 2 members**. Although you can work individually, I encourage you to get a partner.
- One submission per group. **Mention the name of all members.**
- **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.

---

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

The 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`) [**20 points**]

  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`) [**15 points**]

  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`) [**50 points**]

  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 an additional 10 **bonus points**.

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) **[15 points]**

  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) **[10 points]**

  Implement the countInversions(int left, int right) method to count the number of inversions in an array. You may use the merge-sort code; you have to modify the base-case.

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 [10 points]

Once you get the correct output, use `TestTime` to get a running time analysis. I have also provided the numbers (for C++, JAVA, and C#). Your task is to analyze the second set of output, and **write a brief report on what you observe for the following:**

- How do the three sorting algorithms fare against each other? [**4 points**]
- How does the randomized selection algorithm fare against a radix sort based selection? Why do you think that although both are linear time algorithms, the latter turns out to be much slower in practice? [**4 points**]
- How does brute-force inversion counting fare against the merge-sort approach? [**2 points**]

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.

2.1 C++

2.1.1 Sorting Comparison Data

Length	MergeSort	QuickSort (median of 3)	QuickSort (randomized)	RadixSort
500000	124.252	146.538	66.409	58.737
650000	133.378	197.137	91.608	74.438
845000	177.408	261.867	125.692	96.021
1098500	227.205	331.342	155.467	126.37
1428050	306.9	479.368	244.107	173.668
1856465	422.92	625.535	291.686	215.672
2413404	549.229	832.814	376.646	279.286
3137425	737.468	1146.83	546.215	383.798
4078652	1026.29	1650.09	815.161	622.659
5302247	1233.76	1923.41	876.943	616.198
6892921	1904.16	2784.99	1426.58	921.307
8960797	2274.39	3334.91	1544.12	1176.93
11649036	2913.73	4361.6	2479.75	1755.76
15143746	3974.13	5782.47	2869.18	1943.02
19686869	5138.96	7528.37	3682.27	2704.85
25592929	6983.36	10428.5	5205.95	3663.11
33270807	10001.5	14173.8	6256.54	4377.5
43252049	12315.4	17886.1	8067.48	5733.41

```
MergeSort average time is: 2802.47
QuickSort (median of 3) average time is: 4104.2 millisecs
QuickSort (randomized) average time is: 1951.21 millisecs
RadixSort average time is: 1384.6 millisecs
```

2.1.2 Selection Comparison Data

Length	Selection via RadixSort	Randomized Selection
500000	97.366	18.3498
650000	95.845	22.0191
845000	119.198	33.0136
1098500	182.723	44.0556
1428050	200.916	60.0863
1856465	283.646	72.7877
2413404	344.308	92.5475
3137425	453.785	115.892
4078652	588.467	143.387
5302247	661.103	195.252
6892921	1017.63	211.189
8960797	1311.64	315.889
11649036	1966.91	474.657
15143746	2235.14	460.164
19686869	2725.57	727.39
25592929	3770.46	969.856
33270807	4559.16	1229.48
43252049	6496.4	1584.27

```
Selection using RadixSort average time is: 1650.27 millisecs
Selection using random pivot average time is: 412.401 millisecs
```

2.1.3 Inversion Counting Comparison Data

Length	BruteForce Inversion	MergeSort Inversion
10000	253.9610000	1.78
13000	422.81713000	2.057
16900	699.86316900	2.605
21970	1215.8321970	3.744
28561	2001.8728561	4.888
37129	3436.4237129	6.778
48267	6000.7548267	8.66
62747	9822.3262747	13.125
81571	16600.581571	17.04
106042	28544.1106042	28.57
137854	53573.9137854	35.379
179210	93286.8179210	38.292
232973	133755232973	43.705

```
BruteForce average time is: 26893.4 millisecs
MergeSort Inversion average time is: 15.8941 millisecs
```

2.2 JAVA

2.2.1 Sorting Comparison Data

Length	MergeSort	QuickSort (median of 3)	QuickSort (randomized)	RadixSort
500000	114.0	101.0	81.0	95.0
650000	119.0	93.0	108.0	73.0
845000	132.0	122.0	140.0	61.0
1098500	166.0	152.0	168.0	90.0
1428050	222.0	209.0	231.0	114.0
1856465	282.0	329.0	342.0	157.0
2413404	459.0	469.0	404.0	191.0
3137425	514.0	496.0	539.0	282.0
4078652	664.0	639.0	698.0	397.0
5302247	1468.0	847.0	1067.0	598.0
6892921	1459.0	1222.0	1310.0	708.0
8960797	1607.0	1424.0	1549.0	717.0
11649036	2170.0	1878.0	2277.0	970.0
15143746	2932.0	2622.0	2800.0	1598.0
19686869	3649.0	3339.0	3880.0	1900.0
25592929	5712.0	5752.0	6298.0	2368.0
33270807	6315.0	6995.0	7830.0	3267.0
43252049	8618.0	8229.0	9692.0	3976.0

```
MergeSort average time is: 2033.44 millisecs
QuickSort (median of 3) average time is: 1939.89 millisecs
QuickSort (randomized) average time is: 2189.67 millisecs
RadixSort average time is: 975.67 millisecs
```

2.2.2 Selection Comparison Data

Length	Selection via RadixSort	Randomized Selection
500000	42.0	8.92
650000	64.0	10.69
845000	79.0	12.85
1098500	104.0	16.31
1428050	139.0	21.21
1856465	195.0	29.79
2413404	242.0	33.93
3137425	391.0	44.29
4078652	361.0	55.40
5302247	450.0	64.67
6892921	451.0	72.27
8960797	656.0	106.06
11649036	1117.0	177.00
15143746	1562.0	194.50
19686869	1424.0	320.31
25592929	1867.0	311.53
33270807	3045.0	502.18
43252049	3376.0	655.35

```
Selection using RadixSort average time is: 945.02 millisecs
Selection using random pivot average time is: 160.70 millisecs
```

2.2.3 Inversion Counting Comparison Data

Length	BruteForce Inversion	MergeSort Inversion
10000	33.0	3.0
13000	49.0	1.0
16900	90.0	2.0
21970	138.0	2.0
28561	234.0	3.0
37129	403.0	3.0
48267	724.0	6.0
62747	1155.0	7.0
81571	1924.0	9.0
106042	3342.0	12.0
137854	5641.0	15.0
179210	9481.0	23.0
232973	16015.0	26.0

```
BruteForce average time is: 3017.62 millisecs
MergeSort Inversion average time is: 8.62 millisecs
```

2.3 C#

2.3.1 Sorting Comparison Data

Length	MergeSort	QuickSort (median of 3)	QuickSort (randomized)	RadixSort
500000	178	221	242	103
650000	221	289	303	128
845000	298	389	397	162
1098500	372	523	604	211
1428050	505	693	696	265
1856465	1149	975	971	421
2413404	922	1217	1241	451
3137425	1197	1565	1685	590
4078652	1574	2069	2195	837
5302247	2050	2763	2862	1061
6892921	3062	3669	4095	1526
8960797	3863	4907	5153	1798
11649036	4822	6689	6973	2400
15143746	6658	8923	8672	3113
19686869	8388	10975	11531	4020
25592929	11118	15037	15538	5437
33270807	14662	19558	20889	7907
43252049	19419	26479	27563	9759

```
MergeSort average time is: 4469.89 millisecs
QuickSort (median of 3) average time is: 5941.17 millisecs
QuickSort (randomized) average time is: 6200.56 millisecs
RadixSort average time is: 2232.72 millisecs
```

2.3.2 Selection Comparison Data

Length	Selection via RadixSort	Randomized Selection
500000	223	22.54
650000	136	33.69
845000	164	43.69
1098500	209	57.08
1428050	295	76.64
1856465	365	89.57
2413404	451	113.57
3137425	619	151.36
4078652	840	229.47
5302247	1249	316.60
6892921	1565	420.67
8960797	2577	476.31
11649036	2616	604.69
15143746	3189	720.94
19686869	4164	955.38
25592929	5961	1472.35
33270807	8407	1901.06
43252049	9618	2198.88

```
Selection using RadixSort average time is: 2600.91 millisecs
Selection using random pivot average time is: 602.33 millisecs
```

2.3.3 Inversion Counting Comparison Data

Length	BruteForce Inversion	MergeSort Inversion
10000	323	3
13000	519	3
16900	996	4
21970	1492	5
28561	2542	8
37129	4191	10
48267	7847	13
62747	12545	31
81571	21371	22
106042	35022	28
137854	59992	45
179210	99615	55
232973	178672	75

```
BruteForce average time is: 32702.08 millisecs
MergeSort Inversion average time is: 23.23 millisecs
```