BLG 335E – Analysis of Algorithm I, Fall 2017
Project 1 Report

Assignment Date: 27 Sep 2017 Wednesday

Due Date: 11 Oct 2017 Wednesday – 23:59

Kadir Emre Oto

150140032

Code Analysis

In this assignment, we are expected to implement insertion sort and merge sort algorithms and sort the hearthstone datasets using these algorithms. All codes are written in **kod.cpp** file.

Compilation Command: g++ kod.cpp -o kod -O2 -std=c++11
Running Command: ./kod -full -m hs-set-1M.txt out.txt

First of all, I assume that datasets have maximum 10 million card information. If any dataset which has more than 10 million card information, MAXN variable should be increased too. Otherwise most probably Segmentation Fault exception will be raised.

While designing the code, I tried to avoid using standard template library (**STL**) in C++ to increase efficiency, because operations in basic structures like char arrays is faster than complicated structures like std::string which provide many conveniences.

Also to speed-up the swapping operations in sorting, pointers of card structures are stored in Card Manager class and when swapping operation needed, just pointers are swapped and there will be no need for extra copying operation.

Some definitions of some important methods of Card Manager class in project are provided below:

```
bool compare(Card* left, Card* right, int proc):
```

Takes two pointers of Card object and procedure of comparison process as parameters. If procedure is 1, it means comparison will be done according to full sort, otherwise filter sort. The function returns true if the card given as first argument (left) is smaller than the second, else it returns false.

```
int strcompare(const char* left, const char* right):
```

Takes two pointers of char array and compares them ignoring the punctuations and returns -1 if left is smaller, 0 if they are equal or 1 if right is smaller.

```
void sortHelper(int algo, int proc);
```

Takes two integers, and calls the appropriate sorting function with parameters according to following conditions:
algo = {0: mergeSort, 1: insertionSort};
proc = {0: filterSort, 1: fullSort}

```
void mergeSort(Card** ar, int size, int proc);
```

Takes pointer of array of card pointers, size of array and sorting procedure and sorts all cards by using merge sort algorithm.

```
void insertionSort(Card** ar, int size, int proc);
```

Takes pointer of array of card pointers, size of array and sorting procedure and sorts all cards by using insertion sort algorithm.

Question 1 (10 Points)

Provide two graphs or tables (for full sort and filter sort) comparing runtimes of insertion sort and merge sort on three card collections provided with the assignment.

Merge Sort	Full Sort	Filter Sort
10K	0.004 s	0.001 s
100K	0.080 s	0.018 s
1M	1.215 s	0.243 s

Insertion Sort	Full Sort	Filter Sort
10K	0.145 s	0.036 s
100K	49.29 s	8.74 s
1M	-	-

Question 2 (10 Points)

Briefly analyze the computational performances of insertion sort and merge sort with respect to your sorting parameters. Did one of the algorithms always outperform the other? Or did one perform better than the other (or almost catch up with it) in specific scenario? If so, briefly discuss the reasons why?

Algorithm of insertion sort:

```
for (int i=1; i < size; i++)
   for (int j=i-1; 0 <= j && compare(ar[j+1], ar[j], proc); j--)
      std::swap(ar[j], ar[j+1]);
```

It is clear that the average and worst case complexity of algorithm is $\boldsymbol{n^2}$, because there are two nested loops and outer loop iterates all items in array and inner loop iterates until the current item comes to true position. However, let's consider the given array is sorted, in this case inner loop will be never executed because the current item is already in true place, so complexity will be $\boldsymbol{n}$.

Pseudocode of merge sort:

```
func mergeSort(array, size):
   if size is smaller or equal to 1:
      array should be sorted, do nothing
   else:
      mergeSort(array[0:size/2], size/2) # sort first half
      mergeSort(array[size/2:size], size-size/2) # sort second half

      merge this two parts
```

Merge sort is a divide and conquer algorithm, first it splits the array in half and sorts these parts separately, then merges sorted parts by traversing the array one time. Dividing process takes at most $\boldsymbol{log\ n}$ and merge process takes at most $\boldsymbol{n}$ time complexity. So overall complexity is $\boldsymbol{n * log\ n}$.

As a result, it can be said merge sort is outperforming insertion sort, but if the array is sorted or almost-sorted, insertions sort will be better.

Question 3 (10 Points)

Consider you had to sort cards by **RARITY**, or by **SET**, in filter sort. In what way are these two parameters similar to **TYPE**, but different from **NAME**? Could sorting by **RARITY** or **SET** in filter sort affect the performances of insertion sort and merge sort?

According to the given datasets, there are 1324 different names and names can be at most 25-characters length, but there are only 5 different **RARITY** and 17 different **SET**, also they can ben at most 9 and 19-characters length. Insertion sort works better if the array is sorted or almost sorted, thus insertion sort would accelerate much. Also merge sort would accelerate because the comparison operations take less time.

Question 4 (10 Points)

What is stable sorting? Are insertion sort and merge sort stable? What could go wrong in the full sort procedure if an unstable sorting algorithm is used?

A sorting algorithm can be called stable if and only if equivalent elements in a sequence are arranged by their initial position, after sorting. Insertion sort and merge sort are both stable, because they are comparison based sorting algorithms. If an unstable algorithm is used, most probably they generate different outputs.

Additional Approach

When I analyzed the given outputs, I realized that there are only 1324 unique lines in all datasets and all unique lines have unique names. It means we can count the same lines instead of adding the cards, in this manner we just have to sort 1324 cards and while saving stage, we can write lines to file as much as we counted. So the size of the inputs becomes unimportant. Despite this method is not wanted, I just coded this algorithm additionally. It does not take advantages for inputs that do not have repetitions.

Compilation Command: g++ superfast.cpp -o superfast -O2 -std=c++11
Running Command: ./superfast -full -m hs-set-1M.txt out.txt

Results:

Merge Sort	Full Sort	Filter Sort
10K	0.000 s	0.000 s
100K	0.000 s	0.000 s
1M	0.001 s	0.000 s

Insertion Sort	Full Sort	Filter Sort
10K	0.006 s	0.005 s
100K	0.007 s	0.006 s
1M	0.007 s	0.006 s