# Data Structures and Algorithms

Assignment 2 Report

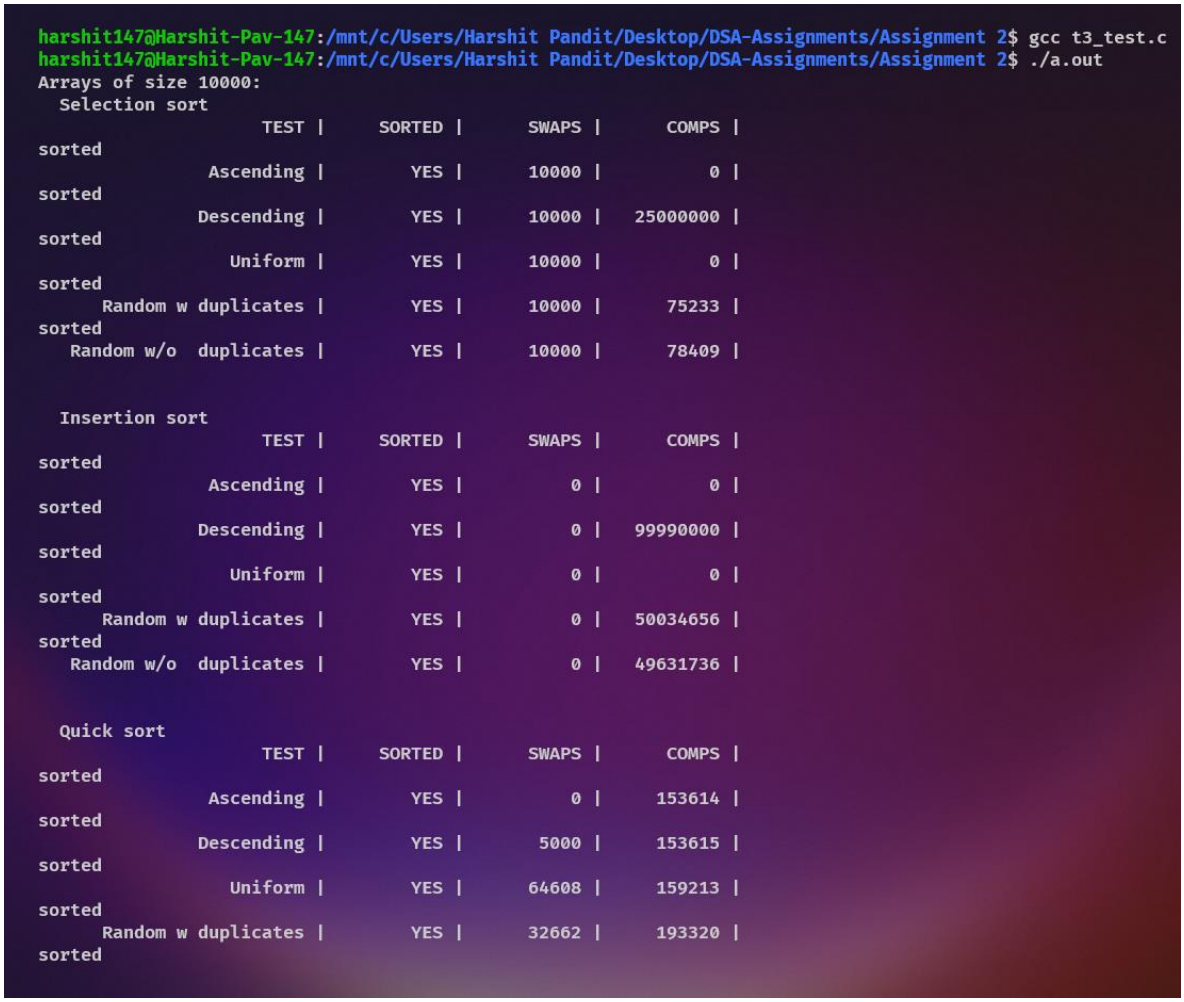
Harshit Pandit TCD 23364360

## Task 2- Quick sort function

My Quick Sort function takes the middle element as the pivot. In this way, even if the array is already sorted or it is in descending order, we have reduced the number of swap operations because there are higher chances that the chosen pivot element is already at its right place.

The partition function chooses a pivot, initializes the two index pointer variables, and runs an infinite loop. Inside the loop, we traverse the two pointers from the front and back until the conditions are satisfied. If the left and right pointers cross each other, we return the right pointer to choose as the new pivot. If they do not cross each other, both elements at their respective indices are swapped. The rest of the Quick Sort function remains similar with the trivial approach.

## Task 3- Comparison of Algorithms



```
harshit147@Harshit-Pav-147:/mnt/c/Users/Harshit Pandit/Desktop/DSA-Assignments/Assignment 2$ gcc t3_test.c
harshit147@Harshit-Pav-147:/mnt/c/Users/Harshit Pandit/Desktop/DSA-Assignments/Assignment 2$ ./a.out
Arrays of size 10000:
  Selection sort
                    TEST |     SORTED |      SWAPS |      COMPS |
sorted
                Ascending |        YES |      10000 |          0 |
sorted
               Descending |        YES |      10000 |   25000000 |
sorted
                  Uniform |        YES |      10000 |          0 |
sorted
       Random w duplicates |        YES |      10000 |      75233 |
sorted
    Random w/o  duplicates |        YES |      10000 |      78409 |


  Insertion sort
                    TEST |     SORTED |      SWAPS |      COMPS |
sorted
                Ascending |        YES |          0 |          0 |
sorted
               Descending |        YES |          0 |   99990000 |
sorted
                  Uniform |        YES |          0 |          0 |
sorted
       Random w duplicates |        YES |          0 |   50034656 |
sorted
    Random w/o  duplicates |        YES |          0 |   49631736 |


  Quick sort
                    TEST |     SORTED |      SWAPS |      COMPS |
sorted
                Ascending |        YES |          0 |     153614 |
sorted
               Descending |        YES |       5000 |     153615 |
sorted
                  Uniform |        YES |      64608 |     159213 |
sorted
       Random w duplicates |        YES |      32662 |     193320 |
sorted
```

The above screenshot is taken from Debian 12 Linux. I shall discuss each of the output tables for the sorting functions in each paragraph.

For the selection sort, the number of comparisons in the case of descending array is extremely high. That is because we are in nested loops that compare two elements even if the array is partially sorted. The number is 0 in ascending and uniformly filled array because the elements are already sorted or have the same value respectively.

For the insertion sort, the number of swaps in each case is 0. That is because insertion sort does not have any swap function to exchange two array elements. In this case also, the number of comparisons is very high because even though the array is partially sorted, we still compare it with the elements which are not already sorted.

Out of all the three methods, quick sort is the most optimized and fastest sorting function for this case. We see that in ascending array, the number of swaps is 0 and we are only comparing the elements. In descending array, the number of swaps is half the size of array because it simply reverses the array. In rest of the cases, the number of swaps is more than the size of array for random elements with or without duplicates. The number of comparisons in all the cases is approximately same.

## Task 4- Most Popular games

To get the top 10 most popular games of last 20 years as per in the given dataset, we would store each of the field in a struct, store all the structs in an array and then sort the array using an optimum sorting algorithm. For this case, Merge Sort was my personal choice instead of Quick Sort.

At the cost of space complexity, I choose Merge sort to sort each of the array elements by score because even at its worst case, its time complexity remains O(nlog(n)) whereas Quick Sort algorithm has $O(n^2)$. If we do choose the pivot element correctly from the array, the best case is still O(nlog(n)).

For larger number of elements in the array, Merge Sort is more efficient and optimum than Quick Sort if we look only in the terms of Time Complexity.

We enter the file from the command line into the program, the parser function gets the file pointer and parses it by each line. Each line is passed on to a function named ‘extractBufferData’ that extracts each of the field values into a struct. The same function then stores that struct into the array of structs declared globally.

Next, we see a utility function that copies the contents of a ‘src’ struct into a ‘dest’ struct which is the destination to copy the data to. This function will be useful in the sorting function.

The sorting function used is the Merge Sort algorithm. In the merge function we have arranged and added each of the structs to the main array as per their scores. We cannot simply copy one array element from the other because this might lead to wrong error allocations. This is the reason why I created a separate utility function that copies the struct data.

Then in the main function, we simply call all the functions in an order and lastly, print the top 10 most popular games of last 20 years. The screenshot of the output obtained is at the next page of this document. The code was tested and developed in Linux Debian 12.


Note: In the images of the command output, cc is my custom command that I created that enters the c file and the csv file name that we have to compile, and it directly gives the output. Under the hood, it still uses gcc compiler.