Data Structures and Algorithms – Assignment 2

Emran Yasser Moustafa – 20332041

Part 1

For part one I had to create a series of functions to create arrays of integers with different characteristics. To create an ascending sorted array I just iterated down the array, setting the value of the array at that index to the value of the index. To create a descending array, I did the same however setting the array value to the size of the array minus the index minus one. To create a randomised array with no duplicates I first created an ascending sorted array. I then iterated through the array, swapping the value at that index with a random index in the array. To create a shuffle array with duplicates, I just iterated through the array, setting each value to a random number capped at the size of the array. I used the *rand()* function to get random numbers for this section.

Part 2

In this section of the assignment I implemented the three sorting algorithms requested. In some cases, I added functions inside the skeleton file to perform the sorting. This section of the assignment is quite self-explanatory.

Part 3

In this section of the assignment, I got this output from the test :

Arrays of size 10000:

Selection sort

TEST | SORTED | SWAPS | COMPS |

sorted

Ascending | YES | 0 | 49995000 |

sorted

Descending | YES | 5000 | 49995000 |

sorted

Uniform | YES | 0 | 49995000 |

sorted

Random w duplicates | YES | 9992 | 49995000 |
sorted
Random w/o duplicates | YES | 9990 | 49995000 |

Insertion sort

TEST | SORTED | SWAPS | COMPS |
sorted
Ascending | YES | 0 | 9999 |
sorted
Descending | YES | 49995000 | 49995000 |
sorted
Uniform | YES | 0 | 9999 |
sorted
Random w duplicates | YES | 25105960 | 25115950 |
sorted
Random w/o duplicates | YES | 24046202 | 24056193 |

Quick sort

TEST | SORTED | SWAPS | COMPS |
sorted
Ascending | YES | 9998 | 50004997 |
sorted
Descending | YES | 14998 | 50004997 |
sorted
Uniform | YES | 19997 | 50004997 |
sorted

Random w duplicates | YES | 35933 | 203260 |

sorted

Random w/o duplicates | YES | 34049 | 216576 |

The results here are as I expected. I will explain some points that I found interesting. Firstly, for sorted and uniform arrays, essentially arrays that do not “need” any elements to be swapped, both insertion sort and selection sort perform better than quick sort. This behaviour was expected. Quick sort, regardless of if the array is sorted or not, has to iterate through the array partitioning it as it goes. This is essentially unneeded computation. This is in contrast to selection sort and insertion sort which can just iterate through the array once and return the array.

However in most other cases quick sort outperforms the other sorting algorithms by many orders of magnitude. If we relate the number of swaps and comparisons to the compute time of each algorithm, this behaviour is completely expected. Quick sort has an average time complexity of O(n*logn) as compared to insertion sort’s and selection sort’s O(n^2). This explains the vastness of the disparity between the performance of both algorithms as compared to quick sort.

Part 4

I got the following output from my script for part 4 :

t4_ign.csv loaded!

Name : Project Gotham Racing 2

Score : 10

Name : Device 6

Score : 10

Name : Tony Hawk's Pro Skater 3

Score : 10

Name : Devil May Cry

Score : 10

Name : Devil May Cry 3: Dante's Awakening

Score : 10

Name : Tony Hawk's Pro Skater 2

Score : 10

Name : Tony Hawk's Pro Skater

Score : 10

Name : Diablo III

Score : 10

Name : Tony Hawk's Underground

Score : 10

Name : Digital Chocolate Cafe

Score : 10

Name : Tom Clancy's Splinter Cell Pandora Tomorrow

Score : 10

Name : Dell Magazines Crossword

Score : 10

Name : Tom Clancy's Splinter Cell Chaos Theory

Score : 10

Name : Tom Clancy's Splinter Cell

Score : 10

Name : Tom Clancy's Rainbow Six 3

Score : 10

Name : Tornado Mania

Score : 10

Name : Dragon Warrior III

Score : 10

Name : Driver

Score : 10

Name : Dark Souls III

Score : 10

Name : Elite Beat Agents

Score : 10

I used my csv reader from Assignment 0 to do most of the work for this section of the assignment. I iterated through the csv adding information from each column to a struct I called Game. This data structure stored two values; the name of the game and the score given to it by IGN. I created a list of pointers to these Game elements, 20,000 entries long. Each time a new entry was read from the csv it was stored in this array. And index counter was used to keep track of what entry we are on and to ensure elements were being added correctly to the array. One point to note, when adding elements to the list, I had to check if the entry existed already. The reason

for this is that there may be many releases of the same game on different platforms. To do this I just checked if the name of the entry I want to add is the same as the last three entries. If so, the element would be skipped.

I decided to use an altered version of my quick sort function for this exercise. The reason for this is that quicksort outperformed the other two sorting algorithms for sorting randomised shuffled arrays with duplicates. Also, the list is considered an in-place array and writing new data is not that difficult, making quick sort more suitable.

There was only one notable differences between the two quick sort implementations. When accessing the value at each index we had to follow that pointer and access it's score value instead of just looking at the value at a certain index of an array. The rest of the implementation is relatively straightforward.

To find the top 5 rated games over the last year I would perform a double sort on the list. I would first sort the list by year. I would then divide the list into tags representing the games from that year. This would be done by passing the start and end index of each section of the list. I would then pass these subarrays into my sorting function. For example, this would look like *quicksort(gameList[start_idx:end_idx], end_idx – start_idx)* in pseudocode, where the first parameter into quicksort is the array tag and the second parameter is the size of the tag. I would use a loop to do this for each of the 20 years, printing the top five games of the sorted output array.