Sorting – Assignment 2

3D5

Name: Oisín Cullen |Student number: 20332292 | Date: 30/10/2022

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Task 1 -Function Generation

In this task I was required to create functions which would fill an array of X size in certain orders. The first way I had to fill the code was in **ascending order.**

To do this I used a for loop which would go from 0 to size of the array defined by the user before hand iterating the nth iteration of the loop into that position each time.
Descending order was similar but the for loop was adjusted so that the biggest value would be placed at the start instead of the end.

For the **uniform array**, a "random" number was generated by the rand() function. The size of the value was kept in the bounds of the size of the array. The value that was randomly generated was then inserted into the whole array

For the **random array with repeating variables,** I again used the rand function to generate variables in the same size of the scope. A for loop would iterate through the array inserting a random variable each time.

For the random array with non-repeating variables .I first called on the fill ascending function so that each unique value was stored in the array, once stored I looped through the whole array and used the rand function to move the values around in random positions so that no repeating variable was present but their position was completely randomized.

Task 2

Selection sort works very simply: In short A for loop iterates over the array from the start to one before the end. Inside this for loop is another loop which looks for the smallest value from the position of the first loop to the end. If the smallest value is found the values are swapped, this repeats through each iteration of the array.

Insertion sort works by inserting the variable into an array of already sorted variables. A for loop sweeps through from left to right. Inside this loop another loop is running from that inde-1 down to the end of the array if the item is bigger swap otherwise the loop is broken and the outer loop continues.

Quicksort works by breaking the array down to small enough parts so that the problems are elementary enough that recursion occurs . How my quick sort function works is that once quicksort is called the bounds of the array is defined and called into a new function called quick_sort2 . Quicksort2 calls on another function called partition. Partition will return an index position to split the array. Once a value is Quicksort2 than calls on itself again but this time splitting it between whatever value partition returned and repeats the process further splitting the array the process stops when the index of the left side is greater than the index position of the right, meaning it cannot be broken down any further.

Partition does two tasks:

- Swaps values of variables and sorts them in order
- Returns the next position of the array to split to continue the loop

Partition achieves this is by first defining a value in the array which will split the array. This variable is called the pivot(piv) pivot is a value in the array The last value in the array was chosen this was for ease of implementation so I could implement the movement of that variable easily and so that everything worked, However this may be why quicksort doesn't perform as effective as it could have done in task 3(see task 3).

Task 3.

Task 2's code and task 3 codes are the same. What's important for this task is to note and compare the effective ness of each sorting algorithm compared to each other and the possible arrays they have to sort. How we measured how effective these assignments were was by noting the number of swaps and comparisons that occurred. A tracker was placed in every algorithm which measured these respectively.

Results:

```
oisincullen@pop-os:~/Documents/Code/Assingment2$ gcc -Wall t1_skeleton.c t2_skeleton.c t3_test
bash: ./t: No such file or directory
oisincullen@pop-os:~/Documents/Code/Assingment2$ ./t3
Arrays of size 10000:
  Selection sort
                    TEST |     SORTED |      SWAPS |      COMPS |
sorted
               Ascending |        YES |          0 |   49995000 |
sorted
              Descending |        YES |   49995000 |   49995000 |
sorted
                 Uniform |        YES |          0 |   49995000 |
sorted
      Random w duplicates |        YES |   18145701 |   49995000 |
sorted
   Random w/o  duplicates |        YES |   23921434 |   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 |   25123608 |   25133602 |
sorted
   Random w/o  duplicates |        YES |   23643689 |   23653681 |


  Quick sort
                    TEST |     SORTED |      SWAPS |      COMPS |
sorted
               Ascending |        YES |      10000 |   50015000 |
sorted
              Descending |        YES |       9999 |   50000002 |
sorted
                 Uniform |        YES |      60518 |     131036 |
sorted
      Random w duplicates |        YES |      31800 |     190327 |
sorted
   Random w/o  duplicates |        YES |      31393 |     184043 |
```

Above are the results from testing each sorting algorithm. From analyzing the results provided it is clear that certain operations do favor certain algorithms. For Quicksort which uses divide and conquer techniques it faired much better at randomly entered arrays, whilst searching algorithms dealt better with ordered arrays.

Selection sort performed arguably the worst out of each algorithm of size 10,000 this is because it has a large amount of comparisons and a large amount of swaps. On every test it used the maximum possible amount of comparisons it could have done with its complexity being (o(n^2)(loop inside a loop).Therefor this outcome was expected.

In my opinion Insertion sort worked incredibly well. With regards to its complexity, it performed its best and worst cases, performing (o(n)) on the first and third array whilst

performing worst on task 2(($o(n^2)$), the most intensive array and performing in between its best and worst for both random arrays.

Quick Sort was interesting. When it dealt with the hardest array for every other algorithm to deal with, descending, it performed the worst out of every other algorithm performing close to ($o(n^2)$. The same can be said for ascending order where it performed the worst out of all generated arrays.

Conclusions

Whilst I expected Quicksort to work badly for descending order, I found it interesting that it did not perform well for ascending order. One reason of this I suspect is how the partition function chooses its pivot variable. Since the value at the end of the array is chosen as the pivot variable it means that high has to cycle left and right to find no value being out of place since nothing is found low equals to high and the value is returned dividing the array further down for it to realize that no changes need to be done.

I found the number of swaps interesting with Quick always staying consistently low but for the insertion algorithms if a swap happens at all it happens in a large magnitude. To conclude I do believe quicksort was the most effective algorithm of this array size however it did have clear downsides. Insertion sort proves that it does have some useful use cases and selection sort showed its one of the worst sorting algorithms to use.

Task 4

Task 4 uses the same algorithm for quicksort used in task 2 and 3 however the function is altered to deal with struct arrays instead of the regular integer array. I decided to use quick sort as it would be most effective for this task. I sorted the array with respect to the top rating and then printed the last 10 values in the array as the quick sort algorithm still sorted from ascending order.

How I would sort each top 5 games of every year would be by using a quicksort algorithm to first sort each value by year. Once sorted I would loop through the array and use an array of 20 variables which would take note of each index where that respective year starts in the array. Once the years are sorted, I would use I would then use quick sort again to sort in order of descending value the top 10 games, however I would only call quick sort from where that year starts and end, essentially breaking the array into a smaller array of each year but by doing so keeping the stable.

I would then print the top 5 of every year by going to where the year starts in the array and loop through it 5 times and repeat for all 20 years.