3D5A Data Structures & Algorithms
Assignment 2 - Sorting Algorithms

Alex Dunphy - 12303659 – 15/11/2016

Abstract:

In this report we designed and evaluated two sorting methods, selection sort and quicksort. Selection sort being a more simplistic method and quicksort a more complex method. Through analysis it was found that the quicksort method in terms of time complexity was more efficient.

Introduction:

Sorting algorithms are used commonly throughout computing, this is due to the fact that arrays are often desired in an order, i.e. alphabetically or numerically, this is for two reasons. The first of these is when an ordered set of data is required eg. A list of student names. The second case is so as to optimize another method, these other methods that would benefit from an ordered set of data would be search or merge functions. Search methods such as binary search trees require a sorted list of data as part of their method of operation.

Elementary sorts are the most basic sorting algorithms; however, this simplicity does result in a less efficient algorithm. There are four elementary sorts and they are as follows: selection sort, insertion sort, shell sort and bubble sort.

More complex sorting algorithms exist and are found to be more efficient than the previously mentioned algorithms.

Method:

As part of this assignment two sorting algorithms were designed and evaluated, the first of the two was the selection sort, an elementary sorting algorithm. This algorithm as mentioned previously is a simplistic method but has drawbacks. The second of the two algorithms was a more complex algorithm which is commonly known as quicksort.

Selection sort operates by cycling through the array of data for each element and switching the position of the investigated element with the lowest value found after it in the array. Through this operation the array is ordered over the multiple cycles. In practice this technique is achieved using a for loop nested within another for loop.

```
for (int k = 0; k < size-1; k++) {
        int current_pos = k;
        int min_val = k;
        for (int i = k; i < size; i++) {
                probe++;
                if (array[i] < array[min_val]) {
                        min_val = i;
                }
        }
        swap(array, k, min_val);
}
```

Listing.1

The above listing is the body taken from the selection sort function. The outer for loop is the current position in the array, going from position 0 to the size -2. The loop only goes to size-2 due to the fact that the array will be ordered before reaching the last position of the array. The nested for loop (line 21) cycles from the current position in the outer loop to the end of the array. At each point the program checks whether the element in the position of the current value of i is less than the element in the min_val position of the array (line 23). Min_val is a variable which is set initially to the value of k (current position in the outer loop) and in the case that the if statement (line 23) is satisfied, min_val is given the value of i. Once the nested for loop finishes the value of k (current position of outer loop) and min_val are swapped using the swap function. The outer for loop then cycles to the next value and the process is repeated until the for loop finishes.

The swap function is used in both sorting algorithms. It is a void function which given a pointer to an array and two values, switches them. A temp variable is used in order to switch the two variables.

The quicksort algorithm unlike the selection sort orders the array of data by dividing the array in two sections and partially ordering the two smaller sections, i.e. all values smaller than the pivot point are shifted to one side. This is done recursively until the smaller sections are comprised of only one value. This results in the array being fully ordered. This process is completed using two functions, shown in listing.3 and listing.4.

```
60 void quicksort(int*array, int first, int last) {
61     if (first < last) {
62             int pivot_index = partition(array, first, last);
63             quicksort(array, first, pivot_index - 1);
64             quicksort(array, pivot_index + 1, last);
65     }
66 }
```

Listing.3

Listing.3 is the quicksort function, this function is used to split the array in to two parts, this is done recursively (lines 63 & 64). This loop of the function calling itself is stopped by the if statement at line 61. This if statement requires the value of first to be less than last, this prevents the program from entering the loop when the divided parts of the array have less than 2 elements. The pivot-index, the point at which the array is divided, is assigned the value given from the partition function (listing.4).

```
47 int x = array[last];
48     int i = first - 1;
49     for (int j = first; j < last; j++) {
50             probe++;
51             if (array[j] <= x) {
52                     i++;
53                     swap(array, i, j);
54             }
55     }
56     swap(array, i + 1, last);
57     return i + 1;
```

Listing.4

Listing.4 shows the body of the partition function. The for loop (line 47) cycles from the variable first to the variable last, at each point an if statement is used to check whether the value of the current position of the for loop in the array is less than that of last. If the statement is satisfied the value of the current position in the array will be shifted to the left using the swap function(line53). This is done as at the end of the for loop the variable last will be used as the pivot point which will be returned to the quicksort function and as mentioned previously we desire all values less than the pivot point to be on one side of the pivot point.

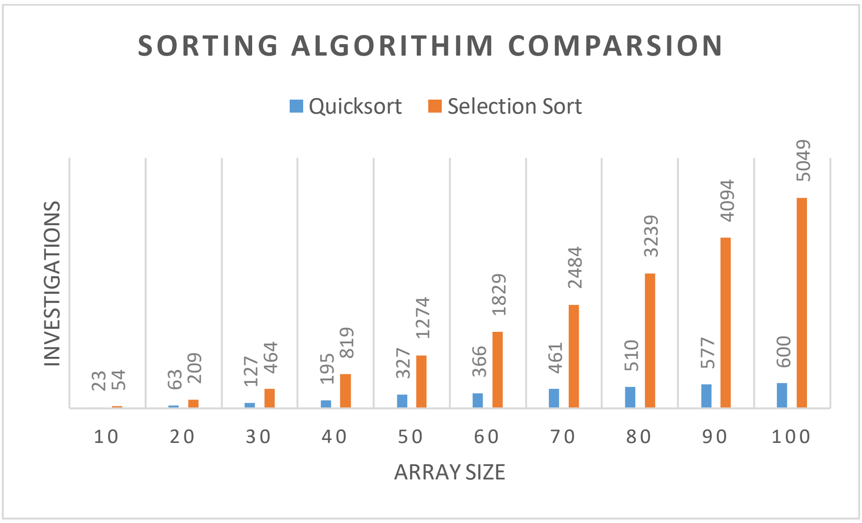
Test Results:

Through testing and research, it was found that selection sort was the less effective algorithm in terms of time complexity. This can be seen in table.1 in both the increased number of probes and the best and average time complexity.

Algorithim Name	Array Size	Probe Count	Best	Average	Worst
Selection Sort	20	209	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quicksort	20	63	O(n log n)	O(n log n)	$O(n^2)$

Table.1 – Big(O) values [1]

Selection sorts n^2 time complexity is due to the nested for loop, for each value in the outer loop the program had to cycle through the complete array. Quicksort's best and average time complexity are when the tree of smaller sections is balanced or roughly balanced meaning the program only has to navigate half the array. The worst case scenario is when the tree has all values on one side, meaning the program has to cycle through every element.



The above chart displays the number of investigations/probes used by each algorithm for varying arrays sizes. This chart allows us to see the effectiveness of the quicksort algorithm over the selection sort, especially for larger array sizes.

Conclusion:

From this report we can see how both quicksort and selection sort algorithms operate, and how the difference in their method of operation results in a significant change in the time complexity.

Bibliography:

[1] - *https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort*