**Introduction**

For our assignment, we had to create and compare three sorting algorithms. I chose:

1. Selection Sort
2. Insertion Sort
3. Quicksort

I first pick an easy algorithm. I pick selection sort as it is one of the most common sorting algorithms and a simple sorting method to implement.

I picked insertion sort because of how common in comes up in my course. It is supposed to be used instead of selection sort and supposed to be more efficient. I wanted to test if this was a true.

Finally, I picked quick sort as it is a very fast and popular sorting method. I wanted to compare its results to the other two sorting algorithms.

**Datasets**

We had to create various data types to use to test the algorithms on. I created a generator program which would create three different data types and put them into three separate text files appropriately named *reversed.txt, sorted.txt* and *unsorted.txt*. There is 1 number per line and the first line is the length of the array. I chose three data types:

1. **Unsorted array** – I chose this data type as usually an array that someone wishes to be sorted is unsorted. I used the rand() function to get a random number
2. **Sorted array** – I read that insertion sort was more efficient at sorting sorted arrays than quicksort. I found this hard to believe, so I tested it.
3. **Reversed array** – I read that insertion sort is very inefficient at sorting reversed arrays, so I wanted to compare its performance on reversed array compared to unsorted and sorted arrays.

**Algorithm Performance**

Selection Sort:

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **Unsorted** | **Sorted** | **Reversed** |
| **10,000** | 0:00:00.15 | 0:00:00.15 | 0:00:00.15 |
| **100,000** | 0:00:15.35 | 0:00:14.93 | 0:00:14.15 |
| **500,000** | 0:06:08.87 | 0:06:05.78 | 0:05:51.77 |
| **1,000,000** | 0:24:22.34 | 0:24:22.91 | 0:23:28.3 |
| **2,000,000** | 1:37:44.5 | 1:51:54.48 | 1:34:10.37 |

As N increases, the time taken increases significantly. From 500,000 to 1,000,000, the time taken is multiplied by approximately 4. The same is happens from 1,000,000 to 2,000,000, i.e. when N doubles. Unsorted and sorted stay similar on time taken with reversed around 1 minute faster for 1,000,000 integers.

Insertion Sort:

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **Unsorted** | **Sorted** | **Reversed** |
| **10,000** | 0:00:00.08 | 0:00:00 | 0:00:00.17 |
| **100,000** | 0:00:08.32 | 0:00:00 | 0:00:16.69 |
| **500,000** | 0:03:26.98 | 0:00:00 | 0:06:55.01 |
| **1,000,000** | 0:13:52.50 | 0:00:00 | 0:27:56.74 |
| **2,000,000** | 0:56:22.42 | 0:00:00.01 | 2:11:13.2 |

As N increases, the time taken increases, although at a much slower rate than selection sort as insertion sort. When N doubles from 1,000,000 to 2,000,000, the time taken increases by over 10 seconds.

Insertion sort is quite clearly very efficient at sorting sorted arrays as each integer only must be compared to one other integer in the array, the previous integer, for the algorithm to know that it is already in the correct place. No swaps are done.

Insertion sort seems to be quite slow at sorting reversed arrays. This is because every integer is compared and swapped with ***all*** previous integers in the array. For large arrays, this method of sorting would be foolish to use. The method took 2 hours 11 minutes and 13 seconds to sort 2,000,000 integers.

Quick Sort:

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | **Unsorted** | **Sorted** | **Reversed** |
| **10,000** | 0:00:00 | 0:00:00 | 0:00:00 |
| **100,000** | 0:00:00.02 | 0:00:00.01 | 0:00:00.01 |
| **500,000** | 0:00:00.13 | 0:00:00.09 | 0:00:00.09 |
| **1,000,000** | 0:00:00.3 | 0:00:00.23 | 0:00:00.22 |
| **2,000,000** | 0:00:00.89 | 0:00:00.83 | 0:00:01.22 |
| **5,000,000** | 0:00:03.11 | 0:00:03.43 | 0:00:05.06 |
| **10,000,000** | 0:00:09.43 | 0:00:12.23 | 0:00:17.48 |
| **20,000,000** | 0:00:32.55 | 0:00:45.22 | 0:01:06.01 |

As the name suggests, quick sort is quick. It was much quicker than the other two algorithms.

For unsorted arrays, quick sort was less than a second at sorting up to and including 2,000,000 integers. To make the time taken larger, I tested quicksort on much larger values of N than I did for the other two algorithms with quick sort taking just 32 seconds to sort 20,000,000 integers.

My quick sort algorithm initially didn’t work for sorted and reversed sort arrays. This was because initially I was always selecting the last integer of the array as the partition. This cause quicksort to hit the worst-case of O(n2). Through online research with websites like geeksforgeeks.org, I concluded that this could be fixed by using either the median or a random value as pivot. Having a random value as the pivot is supposed to be more efficient. To use a random value as pivot, I had to create a new function called rand\_pivot that swaps a random integer in the array with the last integer in the array.

Quicksort slowest data types at sorting are reversed arrays, followed by sorted arrays and fastest at sorting unsorted arrays.

Comparison:

Chart, line chart, histogram

Description automatically generatedChart, line chart

Description automatically generatedChart, line chart, histogram

Description automatically generated

In the above graphs, red represents selection sort, blue represents insertion sort and green represents quicksort.

For unsorted arrays, quicksort should be used and selection sort should be avoided. For sorted arrays, insertion sort should be used and selection sort should once again be avoided. For reversed arrays, quicksort should be used and insertion sort should be avoided.

**Negatives**

I made several mistakes while implementing the sorting methods. As I’ve already mentioned, I had to tweak my quicksort algorithm as it crashed for sorted arrays.

I tried to speed up my program by removing the tmp variable used for swapping variables. Having more variables can slow down a program. To swap elements without a tmp variable, I used operators. Here is how I tried swapping:

arr[prev] += arr[curr];

arr[curr] = arr[prev] – arr[curr];

arr[prev] -= arr[curr];

Although this did work at swapping the two integers, It slowed down the program significantly due to more operators being used. Thus, I reverted back to my original method of swapping using a tmp variable.

**Conclusion and Future Work**

To conclude, quick sort was the most efficient overall of the three sorting methods used. Insertion sort was the second most efficient overall but was a lot slower than quick sort overall but much more efficient than selection sort except for reversed arrays. Now I know which of the used sorting methods are best for sorted, unsorted and reversed arrays.

If I had more time I would’ve:

* Added a fourth data type, this data type would’ve been a nearly sorted array generator. I would be interesting to see how the various sorting methods would sort this. For example, insertion sort was very efficient at sorting sorted arrays. It would be interesting to compare insertion sorts time taken sorting unsorted and nearly sorted arrays.
* I would replace either selection sort or insertion sort with a sorting method that sorts in a very different way like radix sort or bucket sort.