ANALYSIS OF ALGORITHMS HOMEWORK 1
EXPERIMENT DESING

Ulaş Deniz IŞIK 150118887

The program doesn’t read inputs from the file instead it creates them randomly at each execution. The inputs for this project have different sizes that are 10 thousand, 100 thousand and 1 million, and also have different characteristics such as random, sorted and unsorted. The language used for this project is JavaScript. The code was executed on vscode. The code can be executed on browsers by simply deleting the first line of the code and uncomment the second code which assigns the window. Performance object to a variable so the performance property can be used.

The results can change drastically and this depends on where you execute the code on. For instance, when we executed the code on Google Chrome, the results we got were much better than what we got from vscode. The rate of change of the results are the same so that's not an obstacle for us to examine the performance differences of algorithms.

Here are the results we got when we used an unsorted array. We used different sizes so that we can make more accurate inferences and more importantly, we can see the effects of memory usage on the time complexity of algorithms.

Results For An Unsorted Array

	Input size n = 10k	**Input size n = 100k**	**Input size n = 1m**
Counting Sort	5.564 ms	8.112 ms	33.816 ms
Quick Sort	6.5 ms	21.645 ms	134.275 ms
Quick Sort (Median-of-3)	8.231 ms	29.099 ms	140.455 ms
Heap Sort	14.952 ms	42.536 ms	369.858 ms
Merge Sort	17.383 ms	1267.578 ms	468374.385 ms
Binary Insertion Sort	31.483 ms	2535.316 ms	266940.631 ms
Insertion Sort	32.582 ms	2925.296 ms	321638.120 ms

Results For A Sorted Array

	Input size n = 10k	**Input size n = 100k**	**Input size n = 1m**
Counting Sort	3.323 ms	6.436 ms	61.997 ms
Quick Sort	Stack Error!	Stack Error!	Stack Error!
Quick Sort (Median-of-3)	0.538 ms	28.945 ms	61.258 ms
Heap Sort	4.156 ms	40.287 ms	284.975 ms
Merge Sort	11.224 ms	1242.251 ms	300214.400 ms
Binary Insertion Sort	2.194 ms	17.701 ms	158.497 ms
Insertion Sort	0.844 ms	3.433 ms	10.315 ms

Results For A Reverse Sorted Array

	Input size n = 10k	Input size n = 100k	Input size n = 1m
Counting Sort	2.953 ms	5.333 ms	43.184 ms
Quick Sort	Stack Error!	Stack Error!	Stack Error!
Quick Sort (Median-of-3)	0.514 ms	17.215 ms	58.100 ms
Heap Sort	11.420 ms	32.991 ms	261.508 ms
Merge Sort	10.718 ms	1260.856 ms	404360.641 ms
Binary Insertion Sort	46.061 ms	5004.310 ms	628466.250 ms
Insertion Sort	53.994 ms	6073.795 ms	778042.252 ms

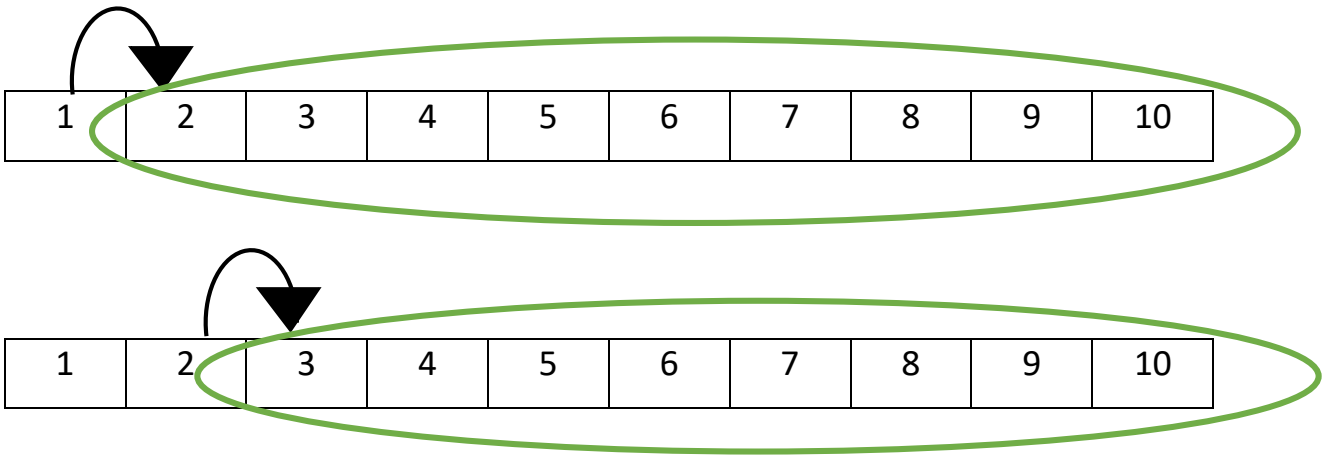
Counting Sort Algorithm

From the theoretical calculations, we know that the Counting Sort algorithm has O(n) time complexity and has O(m) space complexity where m is the table size that is used for counting the occurrences of the elements of the input array. We expected that it would be the most efficient algorithm before we did this experiment and the result approved that. Even we use a large array that has 100 million random integers, the sorting process is done in just 3.4 seconds and when that's impressive. It does the same thing for every kind of array (sorted, unsorted) and that makes it reliable. It gives the same results for all.

Quick Sort Algorithm

This was the most impressive algorithm for us. Despite all of the algorithms that have O(nlogn) time complexity, this is the second best algorithm that comes right after the Counting Sort. Theoretically, the time complexity of this algorithm in the worst case is O(n2). Most of the algorithm that runs less than O(nlogn) such as merge sort, use memory and when the input size is very large, memory usage makes the algorithm run much slower than Quick Sort.

The average time complexity of the algorithm is what we should concern since the array has random inputs. The average time complexity of this algorithm is in O(nlogn) and space complexity is O(logn), and which makes it more efficient than other algorithms that run less than O(nlogn) but use much more space than it. The worst case for a quick sort algorithm occurs when the array is sorted. But crucial thing is that when the array is sorted, and its size is greater than 10-20k, js gives a “Fatal Error!” that says “Stack size is exceeded”. We select the first item as a pivot and find its position (first index) and call the function again recursively with the element that comes right after it.



As you can see in the steps above, it will call the function as long as its length. We tried to see the maximum function call we can and we made a function that calls itself infinitely and counts the function call by simply increasing a variable inside the function by 1. The maximum function call is 15699 on vscode. On Google Chrome, the number is 13949. They were executed on the same computer. When the input is relatively large, it exceeds that number and gets a stack error.

Quick Sort (Median-of-3)

This algorithm is the same algorithm but it selects the pivot element in a different way so that when it encounters a sorted list, it won't take too much time to complete the process. In this example, we are not using a sorted array. So it is the same algorithm but makes more comparison than the other one and that makes it a little bit slower than the normal Quick Sort algorithm.

Heap Sort

Heap Sort algorithm has a time complexity O(nlogn) for the best, worst and average case. Space complexity is in O(1) so it's in place. With these informations, we could conclude that it would be the best of them but it is not. It is very efficient but it is 2 times slower than the quick sort algorithm. The main reason is even if our array is already sorted, it is going to swap all of the elements to order the array. But it comes to the quick sort, it almost doesn't do unnecessary element swaps. Swap is time consuming. That's why it is slower than the quick sort algorithm.

Merge Sort

It is a quite predictable and good algorithm. It gives the same results to arrays that have different characteristic(sorted, reverse sorted etc.). This makes it attractive for developers. The most important thing that we get from it is that space complexity/memory usage of an algorithm is extremely important for the speed of algorithms. Due to memory usage, it is slower than the quick sort and heap sort algorithm. Space complexity is in O(n) and when the size gets larger, algorithms start to slow rapidly. When the input size 10k, it just doubles the Quicksort's time and gives almost the same result with HeapSort. But when we have an array that has 1 million elements, results are enormously different:

When n is 10k:

Merge Sort: 17.383ms , Heapsort: 14.952ms and QucikSort: 6.5ms

It doubles QucikSort, and gives a close result to Heapsort

But when n is 1m:

Merge Sort: 468374.385 ms , Heapsort: 369.858 ms QucikSort : 134ms

It is 1266 times slower than the Heapsort and 3495 times slower than the QucikSort algorithm for n = 1m. The difference are rapidly increasing when input gets larger.

When we try to sort an array that has 100 million element, js gives us a fatal error that says “JavaScript heap out of memory”. It is because of the heap size limitation for node js. Allocated heap size for node js is between 1.4gb and 2gb in 64bit computers. You can increase this size by simply executing following code:

```
node --max-old-space-size=4096 index.js
```

this makes the heap size 4gb but you can increase it to any size you want and you can do it globally instead of a single js file. To sum up, it’s reliable and good but due to memory operations, it becomes slower than some algorithms that have the same time complexity.

Insertion Sort

It’s the slowest one among all of the 7 algorithms that are used in this project if we don’t count its result for the sorted array. The time complexity for this algorithm in the worst case is in O(n2). The best case is in O(n) where the array is already sorted. It just looks at the elements and skips them so that's why it is in O(n). Average time complexity is in (n2) and that means for both reverse sorted and random array, it will have approximately the same time complexity. If we look at the results, it spends 2 times more time sorting a reverse sorted array.

When n = 100k and the array is not sorted, it takes 2925ms to sort it. But when the array size is the same and its reverse sorted, it takes 6073ms to sort it. So the time differences are almost double even though theoretically they both in O(n2).

When the array is sorted, it is even more efficient than the Counting Sort algorithm. The main reason is while the Counting Sort algorithm makes a table of size m, where m is all elements that can be in the input array and passes the array 1 time to fill the table array and then makes a loop to create a sorted array.

“2n+m” for time complexity and “m” for space complexity. But insertion sort does not create a table of size m and passes elements for once.

Binary Insertion Sort

This is a better version of the insertion sort. The difference is that we use binary search to find the proper location to insert the selected item at each iteration. We reduce the comparisons for unsorted and reverse sorted arrays by doing this. But this increase the time when we have already sorted the array because it makes more comparison for selection pivot for the sorted array. That's why it takes more time to sort an already sorted array. İf we compare it with the counting sort algorithm, for the small size inputs, they gave the same results but when the input gets larger, it takes approximately 3 times more time to sort the array for binary insertion sort.