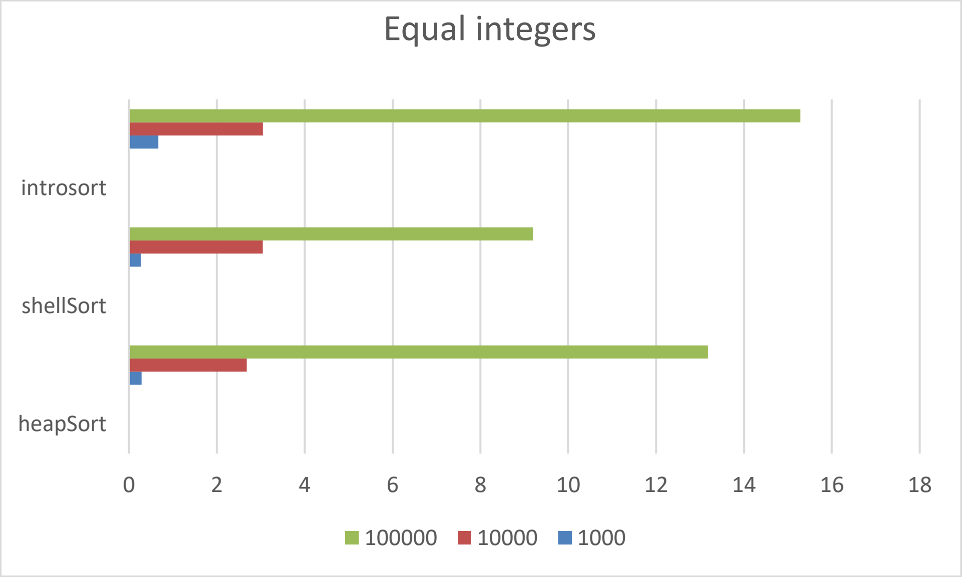
CME 2204 Assignment-1

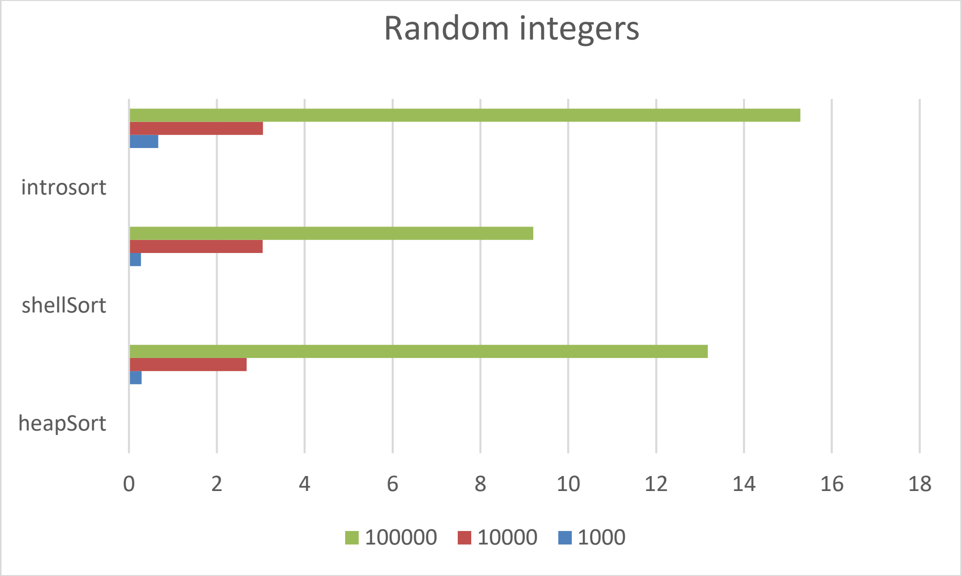
Comparison of Heapsort, Shellsort and Introsort

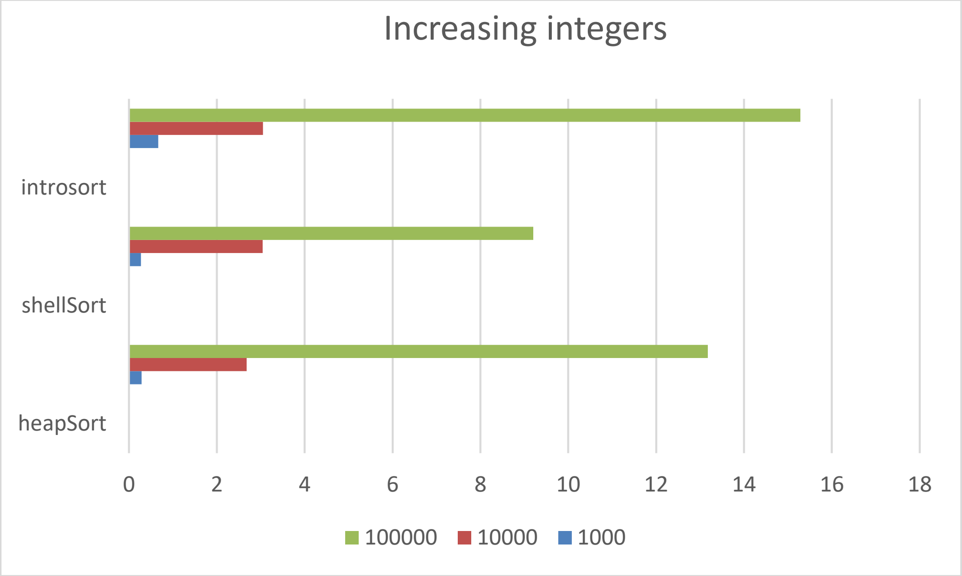
	EQUAL INTEGERS			RANDOM INTEGERS			INCREASING INTEGERS			DECREASING INTEGERS		
	1,000	10,000	100,000	1,000	10,000	100,000	1,000	10,000	100,000	1,000	10,000	100,000
heapSort	0.3685	0.5314	4.2154	0.3281	2.8305	23.4373	0.5726	3.5405	22.0663	0.2854	2.6772	13.1793
shellSort	0.924	2.8575	13.8883	0.2026	4.4124	22.0457	0.4138	3.645	11.987	0.2735	3.0397	9.2065
introsort	0.878	4.9255	14.7916	0.5183	2.8332	15.1046	0.8075	3.3197	18.7697	0.666	3.0503	15.2798

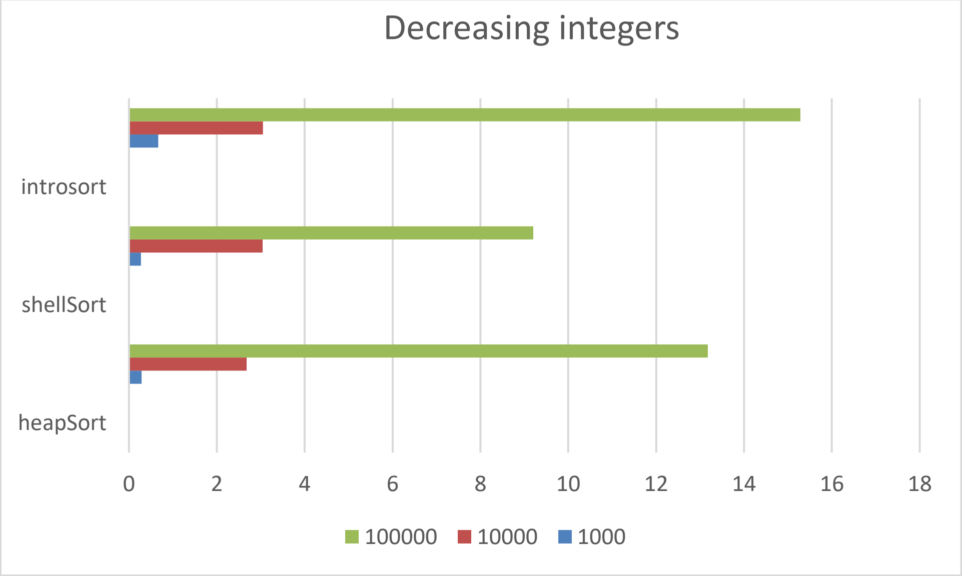
Report:

Heapsort→O(NlogN)
ShellSort→O(n^2)
IntroSort→O(NlogN)




Random integers
introsort
shellSort
heapSort
0
2
4
6
8
10
12
14
16
18
🟩 100000
🟥 10000
🟦 1000


Increasing integers
introsort
shellSort
heapSort
0
2
4
6
8
10
12
14
16
18
🟩 100000
🟥 10000
🟦 1000


Decreasing integers
introsort
shellSort
heapSort
0
2
4
6
8
10
12
14
16
18
🟩 100000
🟥 10000
🟦 1000

First of all, I would like to explain the algorithms a little bit. Let's start with Heapsort. Using the heap sort heapify structure, it first converts the given array into a heap, and then sorts it step by step by comparing the subheaps on the heap and taking the elements to root in order. Shellsort actually works with a simpler logic than the others. It works with 2 for loops and skips a certain part and ranks those parts among themselves. If it is already sequential, the shellsort's job becomes easier and performs much better. Briefly, it sorts the array by going piece by piece. Introsort, on the other hand, is not actually a sorting algorithm per se. It is a hybrid algorithm that includes quicksort, heapsort and insertionsort. It is a kind of selector that changes the algorithm according to the size of the data structure. If the number of elements in the input gets fewer, the Introsort performs Insertion sort for the input. After the number of data exceeds a certain number, if the recursion depth is high, then quicksort is used if the heapsort is low.
These 3 algorithms are an in-place algorithm, but it is not a stable sort.

Let's look at our first situation right away, as we can see in the equal integer part, heapsort works much faster than other algorithms. This is because equal integer is the best case of Heapsort. When we look at the 1k part, there are very small differences in the algorithms showing close performance. When we pass to the 10k state, the heapsort reveals itself and is ahead of the others. The 100k case is the case where the difference is most evident. The other two algorithms give a much worse result here compared to heapsort.

Let's move on to the random integer part. In this section, at first glance, shellsort seems to perform better in the case of 1k, but as the number of data increases, the introsort reveals itself. When the 10k state is passed, shellsort works slower than the other two algorithms. heapsort and introsort show close performance. In the case of 100k, the introsort is clearly evident. Of course, being a hybrid algorithm and choosing a different path according to the number of data provided the introsort to a great advantage.

When we switch to the increasing state, the algorithms exhibited almost the same performance in the 1k and 10k states. Since there are already small data, the difference is not clearly understood. But looking at the 100k situation, it's clear that shellsort performed the best here. The reason for this is that the data that is already given in a ready-made order is the best case of the shellsort. Shellsort compares certain parts by skipping them, for example, if the jump amount is 3, it compares the 1st and 4th elements, but since it is already sequential, it works very quickly and reaches no immediate results.

When the last situation, decreasing, is considered, almost the same performances are obtained as in the previous situation. Again, shellsort performs better.Looking at the 1k situation, heapsort and shellsort work at almost the same speed, while introsort lags behind a bit. When we moved to the 10k part, heapsort worked faster than the other two algorithms with a very small difference. In this part, shellsort and introsort showed close performance. Looking at the 100k part, shellsort performed much better than the other two algorithms. introsort and heapsort performed close to each other here. Frankly, I ran the code many times in this part and encountered different results each time, but this was the situation I encountered the most.

Scenario: We aim to place students at universities according to their central exam grades and department preferences. If there are millions of students in the exam, which sorting algorithm would you use to do this placement task faster?

Answer: When we look at the situation here, the part that most resembles the data we have is the random 100k situation. When this part is examined, the algorithm that should be used is the introsort. Because as the number of data increases, introsort analyzes the data within itself and chooses a path accordingly. If we have millions of data and the recursion depth is high, the heapsort algorithm will use it. But if it is low, it will work with the quicksort algorithm by changing the method immediately. This situation and the dynamics it contains make it more flexible and logical to use different methods according to the size and depth of the data.

References:

https://www.javatpoint.com/heap-sort
https://www.javatpoint.com/shell-sort
https://www.geeksforgeeks.org/introsort-or-introspective-sort/