CSCI 36200: Data Structures

Programming Assignment 3

Dimitrije Prosevski

Dr. Snehasis Mukhopadhyay

Due date: April 11, 2019

Contents

Project Objectives………………………………………….……….…………………………......2

Description of Algorithms.……………………………………………………………………......2

Plots……………………...….………………….…………………………………………............3

Comments on plots.……...….………………….…………………………………………............4

Comments & Conclusions….…………………………………………………………………......5

# **Project Objectives**

The objective of the project was to implement four sorting algorithms: insertion sort, quick sort, heap sort, and merge sort either recursively or iterative and compare their execution times/time complexities. Depending on the value of n, my initializeVec(vector<int> vec, int arr, int lim) function creates a vector. The size of the vector is passed through variable “int arr” where “arr” is the size of the current n that is in the in array of n values.

int nArray[repeats] = {100, 500, 1000, 2000, 5000, 8000, 10000};

Chrono library is used to measure the time of the execution of the function.

auto t1 = high\_resolution\_clock::now();

auto t2 = high\_resolution\_clock::now();

auto t = duration\_cast<microseconds>(t2 - t1);

t1 is the initial time, t2 is the end time, and t is the difference of them.

**Description of algorithms**

Insertion sort algorithm works iteratively by inserting the current number of the loop in the right spot of the data structure (vector in this case). Once the current number moves to the right spot, all the other numbers’ positions get shifted by 1 towards right. Once the loop is done, algorithm is sorted.

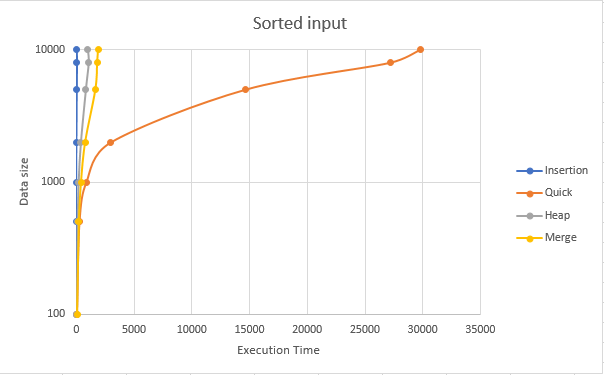
On each call of quick sort function, quick sort calls itself recursively if front < end. Partition gets incremented on each run and gets passed as front of the recursive call of quickSort() function. The algorithm works on divide and conquer principle. My implementation picks the first element as pivot and does the partition around the pivot. The algorithm puts smaller elements on the left of the pivot and greater elements on the right of the pivot. Recursively apply quickSort() to the left and to the right of the pivot.

Heap sort works as imitation of a binary tree as a vector in this case, where left child is (2\*i)+1 and right child is (2\*i)+2 and their parent (root node) is “i”. The code works as putting the greatest number as the root, then swaps the first element with the last of the heap, then reduces the size of heap range by 1. Then heapify() is called and by the recursive process the vector gets sorted.

My implementation of merge sort function has a divide and conquer recursive algorithm that divides the vector into subl and subr (sub-vector left and sub-vector right) and then recursively calls itself for the subl and subr splitting them in half until it gets individual elements. In the end it merges them together in increasing order.

**Plots**





**Comments on plots**

For the unsorted data we can see that the fastest execution times are in the following order:

1. Quick sort
2. Heap sort
3. Merge sort
4. Insertion sort

For the sorted data we can see that the fastest execution times are in the following order:

1. Insertion sort
2. Heap sort
3. Merge sort
4. Quick sort

|  |  |  |
| --- | --- | --- |
| **All the measures are in micro seconds** | | |
| **Insertion sort** | | |
| **n** | **Unsorted Exec Time** | **Sorted Exec Time** |
| 100 | 9 | 1 |
| 500 | 318 | 2 |
| 1000 | 507 | 4 |
| 2000 | 1925 | 6 |
| 5000 | 21201 | 12 |
| 8000 | 39221 | 15 |
| 10000 | 32601 | 12 |
|  |  |  |
| **Quick sort** | | |
| **n** | **Unsorted Exec Time** | **Sorted Exec Time** |
| 100 | 11 | 13 |
| 500 | 60 | 231 |
| 1000 | 127 | 838 |
| 2000 | 259 | 2979 |
| 5000 | 579 | 14635 |
| 8000 | 702 | 27192 |
| 10000 | 615 | 29793 |
|  |  |  |
| **Heap sort** | | |
| **n** | **Unsorted Exec Time** | **Sorted Exec Time** |
| 100 | 15 | 14 |
| 500 | 84 | 80 |
| 1000 | 171 | 163 |
| 2000 | 385 | 366 |
| 5000 | 831 | 808 |
| 8000 | 1012 | 1057 |
| 10000 | 978 | 995 |
|  |  |  |
| **Merge sort** | | |
| **n** | **Unsorted Exec Time** | **Sorted Exec Time** |
| 100 | 57 | 46 |
| 500 | 273 | 211 |
| 1000 | 515 | 397 |
| 2000 | 982 | 755 |
| 5000 | 2154 | 1651 |
| 8000 | 2624 | 1810 |
| 10000 | 2509 | 1891 |

All the sort functions worked similarly as discusses in the class. From the analyzing we can draw the conclusion that insertion sort would work the fastest for the smallest data sets while quick sort works the best for large data sets. Insertion set has by far the best time complexity when the data is already sorted, while quick sort takes longer when the data is sorted than unsorted. Heap sort did not seem to notice the different between sorted or unsorted. And finally, merge sort took a bit longer for unsorted input compared to sorted.

**Comments & Conclusion**

For small databases I would choose insertion sort as preferred algorithm. However, for the large databases I would choose heap sort. The reason is sometimes we can get the data where certain part of the data is sorted, then unsorted, and similarly unpredictable. As seen above, heap sort seems to deal with each scenario as the second fastest, while quick and insertion sort vary from first to last place.

**Comment:** I got my plot values from running the code on Tesla and plotting the data into excel.

I also prompted the results when n = 100; first unsorted vector data, then sorted vector data after the functions are ran; in order to demonstrate the correctness of each sorting algorithm.

Thank you.

**To run the program on Tesla:**

Go to my folder “**dprosev/362Data/Project3\_362**”

run “**make**”

then run “**./RunAll**”