Alexandria University
Faculty of Engineering
Specialized Scientific Programs
Spring 2025

CSE227: Data Structures II
6th Term
Assignment 2
Deadline: Saturday 15th March 2025 12:30 pm

Assignment 2

Sorting Techniques Part 2

1. Lab Goal

- You need to understand:
    - Difference between sorting in place (no extra memory required) vs. An extra space
    - Different running times for each algorithm, best and worst-case running times

2. Quick Sort

2.1 Introduction

Quicksort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quicksort that pick pivot in different ways.

- Always pick the first element as pivot.
- Always pick the last element as the pivot
- Pick a random element as pivot.
- Pick the median as a pivot.

2.2 Requirements

You're required to implement the Quick Sort Algorithm using randomized partitioning which picks a random element as pivot.

Pseudo-code for the above procedures are explained in detail in lectures

3. Merge Sort

3.1 Introduction

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

3.2 Requirements

You're required to implement the Merge Sort Algorithm

Pseudo-code for the above procedures are explained in detail in lectures

4. Heap Sort

4.1 Introduction

The (binary) heap data structure is an array object that we can view as a nearly complete binary tree as shown in Figure 1. Each node of the tree corresponds to an element of the array.

The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

An array A that represents a heap is an object with two attributes: A.length, which (as usual) gives the number of elements in the array, and A.heap-size, which represents how many elements in the heap are stored within array A.

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a heap property.

In a max-heap, the max-heap property is that for every node i other than the root □ **A[parent[i]] ≥ A[i]**

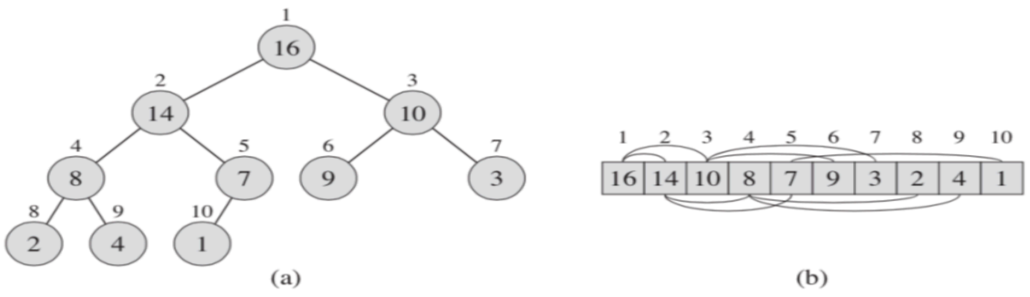


Figure 1

4.2 Requirements

You're required to implement the Heap Sort Algorithm including the following procedures.

- The **MAX-HEAPIFY** procedure, which runs in O (log n) time, is the key to maintaining the max-heap property.

- The **BUILD-MAX-HEAP** procedure, which runs in linear time, produces a max-heap from an unordered input array.

- The **HEAPSORT** procedure, which runs in O (n lg n) time, sorts an array in place.

Pseudo-code for the above procedures are explained in detail in Tutorials

5. Other Sorting Algorithms

You're required to include the algorithms implemented in Part 1 :

- O (n^2) sorting algorithms:
    - **Selection Sort.**
    - **Insertion sort.**
    - **Bubble Sort.**

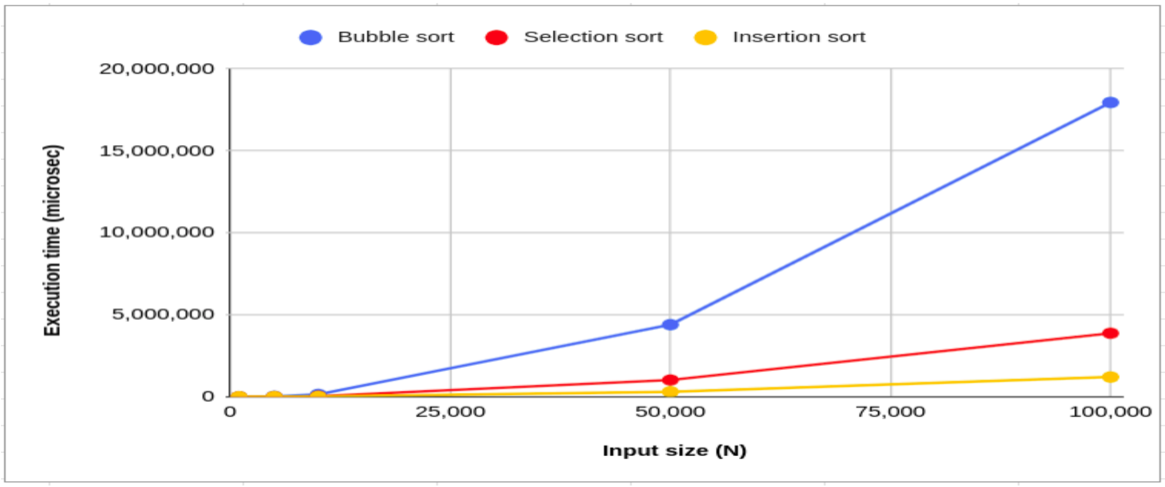
To test your implementation and analyze the running time performance, you will have to implement a function that generates a random array for a given size, sort this array with the sort algorithms you implemented, and print the time taken
Steps:

- ➔ Generate an Array of size X (1000 - 25,000 - 50,000 - 100,000 - ..etc)
  (Make sure you pass the same array to all algorithms with the same numbers)
- ➔ Sort it with your implementation of O($n \, log \, n$) algorithms
- ➔ Sort it with your implementation of O(n^2) algorithms
- ➔ Print:
    - ◆ Running time for Quick Sort is … ms[should be printed for each algorithm]

6. Graph

A graph is required between Time (milliseconds) vs Array Size for different sorting algorithms, generate random arrays of different sizes and calculate the time required to sort it using the algorithms you implemented above and plot the results to a graph as what is shown in the image, you can use Excel sheet to generate the graph.

(Example Graph)



7. Hybrid Merge and Insertion Sort Algorithm

In this Algorithm we need to reduce the overhead of merging when the array size is small, you are asked to Implement a sorting algorithm that can mix between Merge and Insertion, **Implemented Merge sort that expects** an extra parameter **THRESHOLD** that when the array size is less than or equal to this threshold the array should be sorted using Insertion Sort.

Example:

- **Given** an array of SIZE=50 and THRESHOLD=6
- Array will split into 2 arrays of size 25 - 25
- First Array will split into 2 arrays of size 12 - 13
- First Array will split into 2 arrays of size 6 - 6 ← NOTE
- Instead of continuing to break down the array and as the threshold is matched Insertion sort will be used and return sorted array

8. Find Kth smallest Element in Unsorted Array

Given an array and a number K where k is smaller than or equal to the size of the array, Implement a function to find the kth smallest element in the given array using the **Partition function** implemented in quicksort .

Example:

- Given an Array {3, 41,16, 25, 63, 52, 40} find the 3rd smallest Element in the array.
- **Output** 3rd smallest Element is 25.

Summary of Deliverables:

- Part 1:
  - Merge sort, Quick sort, and Heap sort Algorithms
  - Insertion, Bubble and Selection Sort Algorithms
  - Time comparison Report
- Part 2:
  - Hybrid Merge and Insertion Algorithm
  - K^{th} smallest Element
  -

9. Notes

- Implement your algorithms using (Python, Java, or C/C++) Preferably Python
- You should work in groups **of 3 members.**
- Discussion will have higher weight than implementation, so you should understand your implementation well to get discussion marks.
- Your file name should be <id1 #>_<id2 #>_<id3 #>.zip .
- Late submissions are not allowed unless there is a valid documented excuse.