CS 212: Data Structures and Algorithms

Instructor: Drakhshan Bokhat

Lab Task 4: Implementation and analysis of Sorting Algorithms

Mapping CLOs: CLO3

1. **Sorting:**

To sort means to arrange a sequence of elements, like [a_0, a_1, a_2, . . ., a_n] in ascending or descending order. In practice sorting is used to arrange records based on a **key** – *the key is a field in the record.*

An algorithm that maps the given input/output pair is called a **sorting algorithm**

Input: An array A that contains n orderable elements A[0,1, 2,…,n-1]

Output: A sorted permutation of A, named B, such that B [0] ≤ B [1] ≤ B [2] … ≤B [n-1] or vice versa depending on the condition (ascending or descending).

There **are two classes of Comparison Based Sorting Algorithms**:

$O(n^2)$	**O(nlogn)**
Bubble Sort	Quick Sort
Insertion Sort	Merge Sort
Selection Sort	Heap Sort

The other type of sorting algorithms i.e., **Distributive Sorting Algorithms** include Radix Sort, Counting Sort and Bucket Sort.

Comparing different Sorting Algorithms:

Bubble Sort: Under best-case conditions (the list is already sorted), the bubble sort can approach a constant O (n) level of complexity. General-case is an abysmal O (n^2).

While the insertion, selection, and shell sorts also have O (n^2) complexities, they are significantly more efficient than bubble sort.

Insertion Sort: Like bubble sort, the insertion sort has a complexity of O (n^2). Although it has the same complexity, the *insertion sort is a little over twice as efficient as the bubble sort*.

Selection Sort: It yields *a 60% performance improvement over the bubble sort*, but the insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort.

Heap Sort: It is the slowest of the O (nlogn) sorting algorithms *but unlike merge and quick sort* it does not require massive recursion or multiple arrays to work.

Merge Sort: The merge sort is slightly faster than the heap sort for larger sets*, but* it requires twice the memory of the heap sort because of the second array.

Quick Sort: The quick sort is an in-place, divide-and-conquer, massively recursive sort. It can be said as the faster version of the merge sort. The efficiency of the algorithm is majorly impacted by which element is chosen as the pivot point. The worst-case efficiency of the quick sort is o(n^2) when the list is sorted and left most element is chosen as the pivot*. If the pivot point is chosen randomly, the quick sort has an algorithmic complexity of O (nlogn).*

Algorithm	**Time Complexity**		
	Best Case	**Average Case**	**Worst Case**
Selection Sort	O(n^2)	O(n^2)	O(n^2).
Insertion Sort	O(n)	O(n^2)	O(n^2)
Bubble Sort	O(n^2)	O(n^2)	O(n^2)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)

Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)

Big O Notation Revised:

We can group functions into classes by focusing on the fastest-growing term in the expression for the number of operations that they perform.

e.g., an algorithm that performs $n^2/2 – n/2$ operations is a $O(n^2)$-time or quadratic-time algorithm

• Common classes of algorithms:

Name [to slowest]	**Example Expression**	**Big O Notation**
Constant Time	1, 7, 10	O(1)
Logarithmic Time	3log10n, log2n+5	O(log n)
Linear Time	5n, 10n – 2log2n	O(n)
nlogn time	4nlog2n, nlog2n+n	O(nlogn)
Quadratic time	$2n^2 + 3n$, $n^2 – 1$	$O(n^2)$
Cubic time	$n^2 + 3n^3$, $5n^3 – 5$	$O(n^3)$
Exponential time	2^n, $5e^n + 2n^2$	$O(c^n)$
Factorial time	3n!, 5n + n!	O(n!)

You must fill in the table with execution time of the Sorting Algorithms. Results will vary for different input size and different machines [any two different systems/laptops].

To do:

- In the provided code, write a routine that generates random numbers and take it as an input.
  - For this you can use a built-in function ***random ()*** or ***rand ()***
  - Initially the number in array must be 1000 and keep increasing +1000 numbers for next five runs, *later start doubling the input size for next five runs. Lastly, test it for 250000 and 500000 numbers.* *[see if it crashes]*
  - You don't need to do anything except for incrementing the number of iterations in **for/while loop** that generates random numbers.

1000	*2000*	*3000*	*4000*	*5000*	*10000*	*20000*	*40000*	*80000*	*160000*	*250000*	*500000*

- For output, a corresponding output array must be sorted to see if the code (logic) worked properly.

✓ To measure the time taken by the function/routine we can use **clock () function** which is available as ***time.h*** / ***ctime.h.***

- *see the code snippet for reference*

```
#include <stdio.h>
#include <time.h>

void funcTest() { //Implementation }

int main() {      clock_t t;
t = clock();      funcTest();
//function call    t = clock() -
t;
   double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
printf("funcTest() took %f seconds to execute \n", time_taken);
return 0; }
```

Sorting Algorithm: [Merge Sort]

Input Size [N]	1000	2000	3000	4000	5000	10000	20000	40000	80000	160000	250000	500000
Execution Time [Machine A]	X_1ms	X_1ms	X_2ms	X_3ms	X_4ms	X_5ms	X_6ms	X_7ms	X_8ms	--	--	--
Execution Time [Machine B]	X_1ms	X_1ms	X_2ms	X_3ms	X_4ms	X_5ms	X_6ms	--	--	--	--	--