# California State University, Fresno

# DEPARTMENT OF COMPUTER SCIENCE

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Class: | **Algorithms & Data Structures** | | | Semester: | **Fall 2021** |
|  | | | | | |
| Points |  | Document author: | **Ojas Deshmukh** | | |
|  | Author’s email: | **ojas2424@mail.fresnostate.edu email** | | |
| Laboratory number: | **Lab 3** | | |
|  | | | | | |

**1. Statement of Objectives**

In this lab we will be looking at tow sorting algorithms. First one being Merge sort and the other being Selection sort. We will look at the run time for both the sorting algorithms and compare their efficiency three different cases. We will test them on half sorted array, sorted array and reverse sorted array. We will check the time taken to execute these three cases on two different array sizes. one being regular small array and other significantly big.

**2. Experimental Procedure**

We will start with coding the algorithms, first will be merge sort.

Working of **Merge sort** can be explained simply in a few lines. First, we divide the array or list into sub arrays of individual elements, or in simpler terms ‘divide till we get sub arrays of one element each’. Then we simultaneously keep forming subarrays of sorted elements. These subarrays then keep on merging with each other and get sorted at the same time. For example, let us say we have an array (5, 3, 7, 4, 6, 1). 1st step will divide into sub arrays {(5-7), (4-1)} 🡪 {(5, 3), (7), (4,6), (1)} 🡪 {(5), (3), (7), (4), (6), (1)}

then we merge + sort these subarrays {(3, 5), (7), (4), (1,6)} 🡪 {(3, 5, 7), (1, 4, 6)} 🡪 {(1, 3, 4, 5, 6, 7)}

In our algorithm written in lab 3 ‘cpp’ file we have merge function with time complexity of order O(NLogN) since we have while loop checking for the conditions while merging the subarrays.

Let us check **Selection sort**- This is also an easy to explain algorithm. For every position in the array, we scan for the lowest element present to take the place. For example, array = {46 22 11 21 10} {10 22 11 21 46} {10 11 22 21 46} {10 11 21 22 46}. Time complexity order becomes O(N2), because we have two loops, one nested inside the other, where one iterates for n number of times and other for (n-1) times for every position in the array therefore n2.

We call both these algorithms for half sorted, sorted and reverse sorted array. We run it for different values varying significantly in sizes. For int main, to call functions easily we have narrowed both down to two parameter functions, for this, we have created helper functions. We will check both for 20, 200, 20000, 200k, and 2M.

**3. Analysis**

When checked for experimental results, it becomes clear that for larger arrays merge sort outperforms selection sort. When checked for three array categories, we found out that individually for both algorithms, sorted array takes the longest to execute. When we tried to perform selection sort on 2,000,000 and 200,000 array size, we could not complete the execution or get any result, whereas merge sort completed the execution in microseconds, as per our expectations from NlogN compared to N2. Though only outputs for 20,000 and 20 are shown here, that is because it was taking too long for selection sort to execute.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**4. Encountered Problems**

For smaller arrays selection sort did better than merge sort but those were very small arrays, such as 20, 200, 2000 etc elements. We can expect that as merge might take a little longer in this case and selections sort tends to be straight forward. But for larger data sets merge sort is better.

**5. Conclusions**

Merge sort is a standard NlogN algorithm which takes average time in every case. Selection can be taken under consideration for small data sets and can be useful there. Since selection sort did not even complete execution for 5 mins when we left it to run its course for array size of 200,000.