Rajshahi University of Engineering & Technology Department of Computer Science of Engineering

**EXPERIMENT NO**: 01

**NAME OF EXPERIMENT**: Complexity Analysis of Bubble Sort, Selection Sort, and Insertion Sort Algorithm and searching algorithm (Linear search and Binary search)

**SUBMITTED TO:**

**(Rizoan Toufiq)**

**(**Assistant Professor**)**

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING RAJSHAHI UNIVERSITY OF ENGINEERING & TECHNOLOGY**

**SUBMITTED BY:**

**NAME: Md Tanbeer Jubaer**

**ROLL NO.: 1903071**

**GROUP:**

**DATE OF EXP.: /2022**

**DATE OF SUB. :/2022**

**SERIES: 19**

**MACHINE CONFIGURATION:**

Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.21 GHz, 16 GB of RAM

1 TB of Hard disk

**Finding Maximum and Minimum**

Finding the largest element from a list of n random numbers and the smallest number from a list of n random numbers.

**Straight Forward Method:**

In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

**Algorithm: Max-Min-Element (numbers[])**

max := numbers[1]

min := numbers[1]

for i = 2 to n do

if numbers[i] > max then

max := numbers[i]

if numbers[i] < min then

min := numbers[i]

return (max, min)

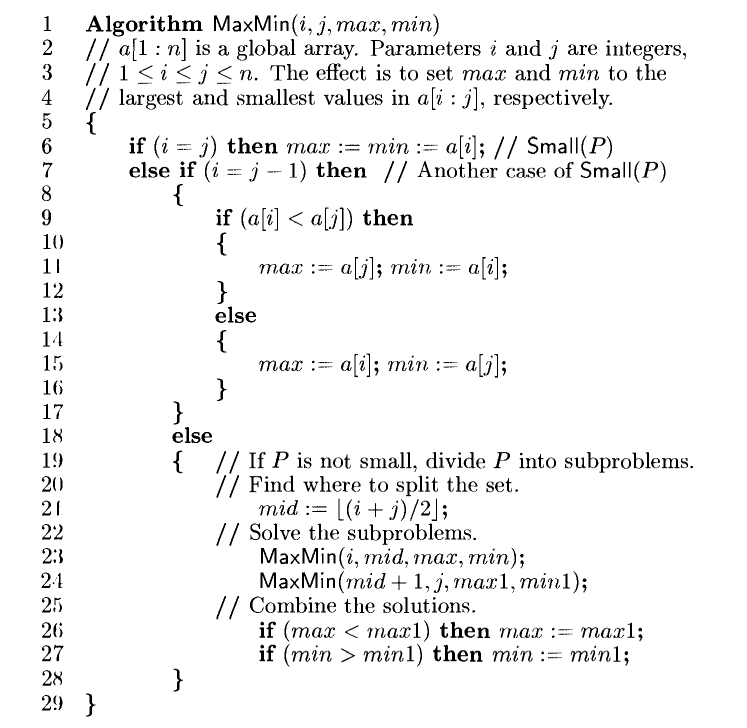
**Analysis**

The number of comparison in Naive method is 2n - 2.

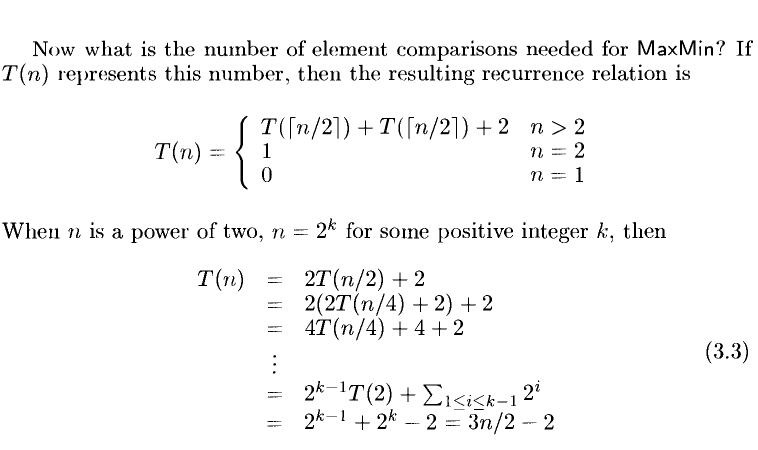
The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

**Divide and Conquer Approach**

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.



**ANALYSIS:**



Compared to Naïve method, in divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation both of the approaches are represented by **O(n)**.

Quick Sort

It is used on the principle of divide-and-conquer. Quick sort is an algorithm of choice in many situations as it is not difficult to implement. It is a good general purpose sort and it consumes relatively fewer resources during execution.

Quick sort works by partitioning a given array *A[p ... r]* into two non-empty sub array *A[p ... q]* and *A[q+1 ... r]* such that every key in *A[p ... q]* is less than or equal to every key in *A[q+1 ... r]*.

**Algorithm: Quick-Sort (A, p, r)**

if p < r then

q Partition (A, p, r)

Quick-Sort (A, p, q)

Quick-Sort (A, q + r, r)

**Procedure: Partition (A, p, r)**

x ← A[p]

i ← p-1

j ← r+1

while TRUE do

Repeat j ← j - 1

until A[j] ≤ x

Repeat i← i+1

until A[i] ≥ x

if i < j then

exchange A[i] ↔ A[j]

else

return j

**ANALYSIS:**

The worst case complexity of Quick-Sort algorithm is ***O(n2)***. However using this technique, in

average cases generally we get the output in ***O(n log n)*** time.

# Merge Sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being Ο(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

**Algorithm: Merge-Sort (numbers[], p, r)**

if p < r then

q = ⌊(p + r) / 2⌋

Merge-Sort (numbers[], p, q)

Merge-Sort (numbers[], q + 1, r)

Merge (numbers[], p, q, r)

**Function: Merge (numbers[], p, q, r)**

n1 = q – p + 1

n2 = r – q

declare leftnums[1…n1 + 1] and rightnums[1…n2 + 1] temporary arrays

for i = 1 to n1

leftnums[i] = numbers[p + i - 1]

for j = 1 to n2

rightnums[j] = numbers[q+ j]

leftnums[n1 + 1] = ∞

rightnums[n2 + 1] = ∞

i = 1

j = 1

for k = p to r

if leftnums[i] ≤ rightnums[j]

numbers[k] = leftnums[i]

i = i + 1

else

numbers[k] = rightnums[j]

j = j + 1

## **Analysis:**

Let us consider, the running time of Merge-Sort as ***T(n)***. Hence,

*T*(*n*)={*cifn*⩽12*xT*(*n*2)+*dxnotherwise* where *c* and *d* are constants

Therefore, using this recurrence relation,

*T*(*n*)=2*iT*(*n*2*i*)+*i*.*d*.*n*

As, *i*=*logn*,*T*(*n*)=2*lognT*(*n*2*logn*)+*logn*.*d*.*n* = *c*.*n*+*d*.*n*.*logn*.

Therefore, *T*(*n*)=*O*(*nlogn*)

**DATA TABLE AND GRAPH:**

|  |  |  |
| --- | --- | --- |
| **Small data** | **Required Time (milli seconds)** | |
| **No. of Data** | **Brute Force method** | **Max/Min** |
| 1000 | 3 | 17 |
| 2500 | 8 | 46 |
| 5000 | 22 | 92 |
| 7500 | 40 | 105 |
| 10000 | 52 | 173 |
| 12500 | 44 | 245 |
| 15000 | 49 | 286 |
| 17500 | 96 | 313 |
| 20000 | 66 | 282 |
| 22500 | 105 | 355 |

|  |  |  |
| --- | --- | --- |
| **Large data** | **Required Time (milli seconds)** | |
| **No. of Data(Lakh)** | **Brute Force method** | **Max/Min** |
| 1 | 0.632 | 1.076 |
| 5 | 17.691 | 5.517 |
| 10 | 11.096 | 11.044 |
| 15 | 12.63 | 16.88 |
| 20 | 24.066 | 38.77 |
| 25 | 12.999 | 40.273 |
| 30 | 15.624 | 50.835 |
| 35 | 17.71 | 40.519 |
| 40 | 95.231 | 61.508 |
| 45 | 166.551 | 49.765 |
| 50 | 100.403 | 77.422 |
| 55 | 102.394 | 75.481 |
| 60 | 197.932 | 75.388 |

|  |  |  |
| --- | --- | --- |
|  | **Required Time (milli seconds)** | |
| **No. of Data** | **Quick Sort** | **Merge Sort** |
| 1000 | 0.155 | 0.221 |
| 2500 | 0.511 | 0.531 |
| 5000 | 0.812 | 1.308 |
| 7500 | 1.23 | 2.069 |
| 10000 | 2.126 | 2.862 |
| 12500 | 2.093 | 3.529 |
| 15000 | 3.034 | 4.149 |
| 17500 | 3.368 | 5.11 |
| 20000 | 3.643 | 5.891 |

**DISCUSSION:**

Today we have analyzed the time complexity of maximum and minimum and sorting algorithms. We have used time function to calculate the time between the required function to be executed. We have measured the time in milli seconds or micro seconds. If we see the graphical output we can see the difference between them. In straight forward method the time is taken less than divide and conquer method when the data was in smaller range. But when we took a large num of data’s then divide and conquer show its best. In between merge sort and quick sort we see that quick sort is showing slightly better result.

**Questions::**

1. What is the reason that divide and conquer in max min took extra time when input data was smaller?
2. Why quick sort is better than merge sort?

**Answer ::**

1. **I think** the divide and conquer approach uses stack this is why in terms of smaller data it behaves slight slow. Because here we need to call the function recursively and it stays on stack the additional time for calling the function may cause this inconvenience. But When the amount of data is very large it shows it’s efficiency.
2. I think There are certain reasons due to which quicksort is better especially in case of arrays:
   1. **Auxiliary Space** : Mergesort uses extra space, quicksort requires little space and exhibits good cache locality. Quick sort is an in-place sorting algorithm. In-place sorting means no additional storage space is needed to perform sorting. Merge sort requires a temporary array to merge the sorted arrays and hence it is not in-place giving Quick sort the advantage of space.
   2. **Worst Cases** : The worst case of quicksort O(n2) can be avoided by using randomized quicksort. It can be easily avoided with high probability by choosing the right pivot. Obtaining an average case behavior by choosing right pivot element makes it improvise the performance and becoming as efficient as Merge sort.
   3. **Locality of reference** : Quicksort in particular exhibits good cache locality and this makes it faster than merge sort in many cases like in virtual memory environment. Merge sort is better for large data structures: Mergesort is a stable sort, unlike quicksort and heapsort, and can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media such as disk storage or network attached storage.