DAA432C Assignment-02

Vikram Singh
IIT2019213
iit2019213@iiita.ac.in

Akshat Agrawal
IIT2019214
iit2019214@iiita.ac.in

Ravutla Ruthvik
IIT2019215
iit2019215@iiita.ac.in

Abstract— In this report, we design an algorithm to solve the problem that given a set of elements, we have to find the average of all elements using divide and conquer.

Keywords— Divide and conquer approach, array, time complexity,space complexity

I. INTRODUCTION

This paper discusses an algorithm that is designed to find the average of the given elements, using divide and conquer approach.

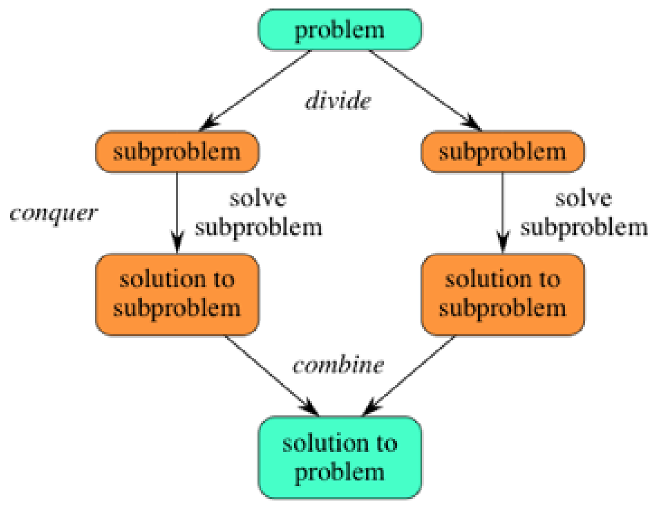
Divide and Conquer is an algorithm design paradigm in which we recursively break down a problem into two or more sub-problems of the same or related type, until these subproblems become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. The divide-and-conquer paradigm is often used to find an optimal solution of a problem in which we recursively simplify the problem by decreasing the constraints of a subproblem and selecting / dis-selecting a given subproblem based on some conditions.

The divide-and-conquer technique is the basis of efficient algorithms for many problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g., the Karatsuba algorithm),finding the closest pair of points.

The approach of divide and conquer generally follows three important aspects:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

Let us have a pictorial look of the image which makes the concept of Divide and Conquer Paradigm self explanatory.



Advantages of Divide and Conquer

The Divide and Conquer technique generally has the following mentioned advantages over Brute Force Algorithms.

Time Efficiency: This approach generally reduces the running time of the algorithm because the running time of an algorithm based on Brute Force is in general high order in nature. But when we are using the approach based on Divide and Conquer, this running time generally decreases and becomes logarithmic in nature. This is because the approach of Divide and Conquer keeps on dividing which makes it logarithmic in nature.

Space Efficiency: As no need to mention the algorithm, but still, the space complexity also reduces up to a much extent due to the nature of dividing and then merging up. Also it is worthy to note that some divide and conquer algorithms which are based on the recursive algorithms generally tend to occupy more space in the internal allotted stack of the process in RAM(Random Access Memory).

I. ALGORITHM DESIGN

The given problem can be solved using divide and conquer approach which is similar to binary search. Basically, we divide the given problem into smaller sub-problems and appropriately combine their solutions to get the solution to the main problem.

Approach: The idea is to first find the middle element of the array. Then we find the average of the elements to the left and right side of the middle element. Then we combine the two averages to get the final average of the array.

Algorithm

1) First, we find the middle element of the array.

2) Then we calculate the average of the left side of the array by recursively calling the same average function.

3) Repeat the previous step for the right side of the array.

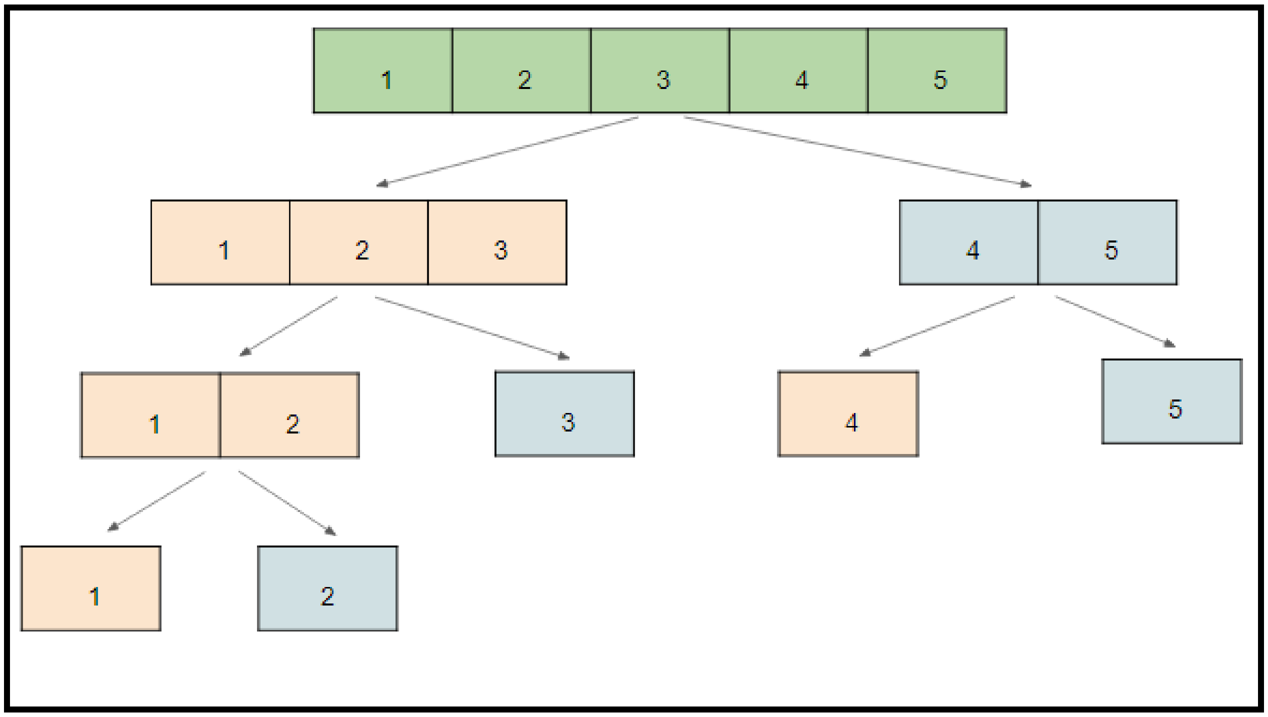
4) Then we combine the two averages according to their lengths to get the final average of the whole array.

5) The formula that we use to calculate the final average is:
((avg of left subarray) x (length of the subarray) + (avg of right subarray) x (length of the subarray)) / (length of the whole array).

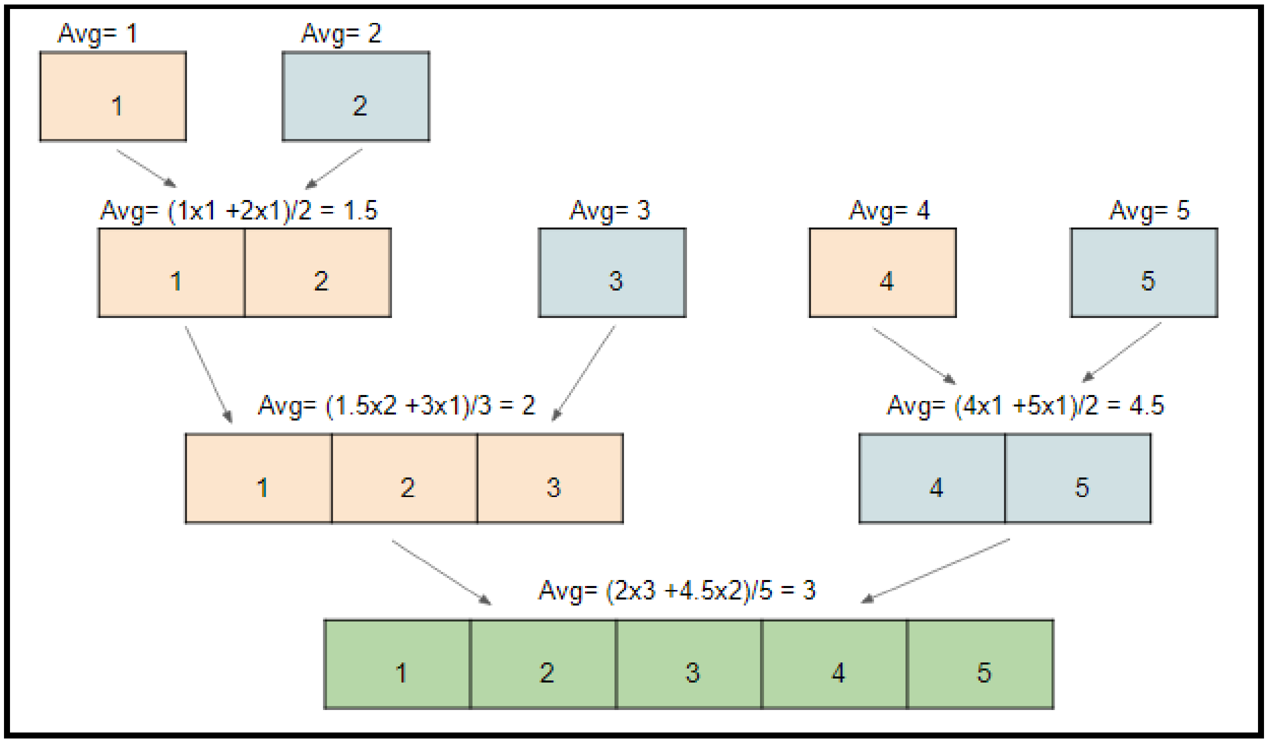
6) Using the above formula we can easily calculate the average of the whole array.

Pictorial representation:

For understanding the above algorithm in a better way, below is the pictorial representation of the approach:



(i) Dividing the input array



(ii) Combining the smaller subarrays to compute final average

III. PSEUDO CODE

Below is the pseudo code for the **find_average** function which contains the algorithm described above.
As this function is a recursive function it has the following base conditions which needs to be checked every time the function is called:

- If the value of the left index is greater than the value of right index , return 0.
- If the value of the left index equals the value of right index , return the value of arr[left].

If the value of the left index is greater than the value of right index , then we proceed further according to the above algorithm.

```
double find_average
(double arr[],int left,int right)
{
    if(left>right)
    return 0;

    if(left==right)
    return arr[left];
   //Base Conditions

    int mid = (left+right)/2;

double lavg=find_average(arr,left,mid);
double ravg=find_average(arr,mid+1,right);

   return
(lavg*(mid-left+1)+ravg*(right-mid)
)/(right-left+1);
}
```

IV. ALGORITHM ANALYSIS

For the above approach based on divide and conquer, the algorithm can be seen as recurring for 2 arrays of size n/2.

Time Complexity:

Following is recurrence for time complexity

$$T(n) = 2(n/2) + O(1)$$

The time complexity for such an recurrence relation can we found out using the

Masters Algorithm:

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta(n^k \log^p n)$.

Then, we have-
$a = 2$
$b = 2$
$k = 0$
$p = 0$

Now, $a = 2$ and $b^k = 2^0 = 1$.
Clearly, $a > b^k$.
So, we follow case-01 which is If $a > b^k$, then $T(n) = O(n^{\log b(a)})$

So, we have-
$T(n) = O((n^{\log b(a)})$
$T(n) = O(n^{\log 2(2)})$
$\mathbf{T(n) = O(n)}$

The solution of recurrence is **O(n)** using Master Method.

Time Complexity: ***O(n)***

Space Complexity:

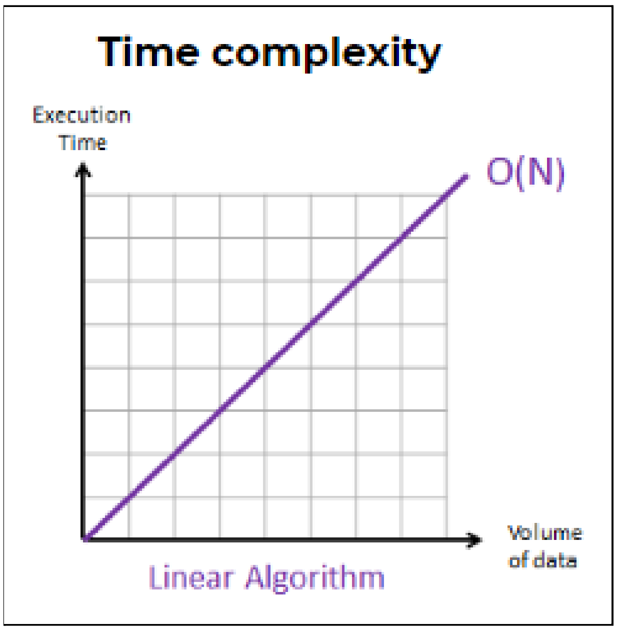
Since we input a 1-D array of size N, it will require **O(n) space.**

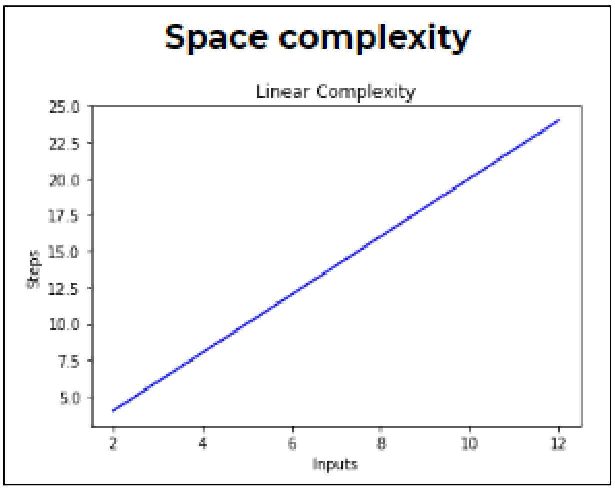
Also as the number of variables used in every function call are fixed, if we try to calculate the total number of function calls made (using the method used for calculating time complexity), we get **O(n)** space complexity.
Thus the overall space complexity of the algorithm is = $O(n) + O(n) \approx \mathbf{O(n)}$.

Space Complexity: ***O(n)***

V. PROFILING

Apriori analysis is performed prior to running it on a specific system. This analysis is a stage where a function is defined using some theoretical model. Hence, we determine the time and space complexity of an algorithm by just looking at the algorithm rather than running it on a particular system with a different memory,processor, and compiler.So,as we discussed under the heading complexity analysis we arrived at the conclusion that the average time complexity is O(n).
Also the space complexity that we arrived at was O(n).





Posteriori analysis of an algorithm means we perform analysis of an algorithm only after running it on a system. It directly depends on the system and changes from system to system. So for the A Posteriori analysis of the algorithm, we have run our code on the compiler and get values of the time by specifying the value of n and as a result we got different values of the time.

S. No	n	Time taken (in microseconds)
1	10	1
2	100	2
3	1000	20
4	10000	200
5	100000	2000
6	1000000	80000

VI. APPLICATIONS

Divide and Conquer is a wide variety of algorithmic paradigm which believes in the strategy of dividing the task and then applying an internal mechanism or algorithm in order to get the required answer. This programming paradigm has several predefined algorithms which work on the basis of Divide and Conquer approach.
Some of the applications of the Divide and Conquer Approach are:

1. Binary Search: It is a powerful programming algorithm based on the divide and conquer paradigm which works very efficiently and gives answers in generally time cost, which is in general logarithmic in nature.

2. Segment Trees: It is another data structure which uses three standard operations:insert,query and update. All these three operations are based on divide and conquer because for example for building a segment tree, we first need to create in a recursive bottom up divide and conquer based technique.

3. Strassen's Algorithm: It is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops. Strassen's algorithm multiplies two matrices in efficient time.

4. Quick Sort: It is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to the left side of pivot, and all greater elements move to the right side. Finally, the algorithm recursively sorts the subarrays on the left and right of the pivot element.

5. Merge Sort: is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

VII. CONCLUSION

We have used a divide and conquer approach with the motive to reduce the time and space complexity of the algorithm. But we find that the time complexity remained the same as found out using the brute force method i.e. O(n). So in this case, the divide and conquer algorithm is not any better than the brute force approach.

VIII. ACKNOWLEDGEMENT

We are very much grateful to our Course instructor Dr Rahul Kala and our mentor Mr. Md. Meraz, who have provided the great opportunity to do this wonderful work on the subject of Data Structure and Algorithm Analysis specifically on the programming paradigm of Divide and Conquer.

REFERENCES

[1] Introduction to Algorithms / Thomas H. Cormen . . . [et al.]. - 3rd edition.

[2] The Design and Analysis of Algorithms (Pearson) by A V Aho, J E Hopcroft, and J D Ullman

[3] Algorithm Design (Pearson) by J Kleinberg, and E Tard

[4]https://www.geeksforgeeks.org/auxiliary-space-recursive-functions/#:~:text=For%20example%20if%20we%20need,space%20required%20by%20a%20program

[5]https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/

APPENDIX

The full code for the above problem is given below:

```
#include <bits/stdc++.h>
using namespace std;

double find_average(double arr[], int left, int right)
{
    if (left > right)
        return 0;

    if (left == right)
        return arr[left];

    int mid = (left + right) / 2;
    double lavg = find_average(arr, left, mid);
    double ravg = find_average(arr, mid + 1, right);

    return (lavg * (mid - left + 1) + ravg * (right - mid)) / (right - left
+ 1);
}

int main()
{
    srand(time(0));
    int n;
    cout << "Enter the number of elements in the array:" << endl;
    cin >> n;
    cout << "Below is the randomly generated array of length " << n << ":"
<< endl;
    double arr[n];
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % 10;
    }
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "The Average of the elements present in the array is: " <<
find_average(arr, 0, n - 1);
}
```