CSE 4810: Algorithm Engineering Lab
Lab - 3

Md Farhan Ishmam (180041120)

Group - CSE 1A

February 12, 2023

Task 1

The merge sort algorithm works by dividing the array into two equal partitions until the array is of a single element. Then two single-element arrays are merged into a sorted two-element array. Afterward, the sorted two-element arrays are merged and the process is continued till we get the sorted original array.

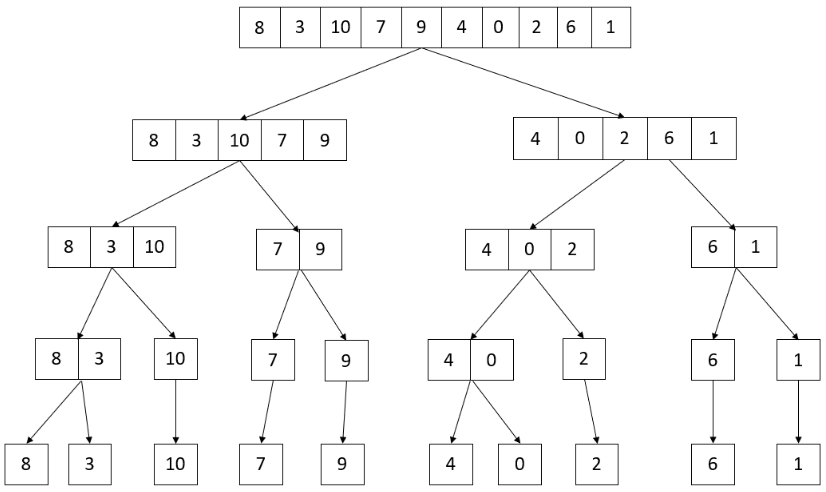

Figure 1: Dividing the given array. The depth of the tree is $O(log_2(n))$

The key principle that improves merge sort is that merging two sorted arrays into a single sorted array takes $O(n)$ time. With a tree depth of $O(log_2(n))$, merge sort gets an overall complexity of $O(nlog_2(n))$. If the arrays are merged in-place then merge sort has a space complexity of $O(n)$. Figure-1 and 2 help us visualize the process of dividing and merging an array while figure-3 visualizes the step of merging two sorted arrays.

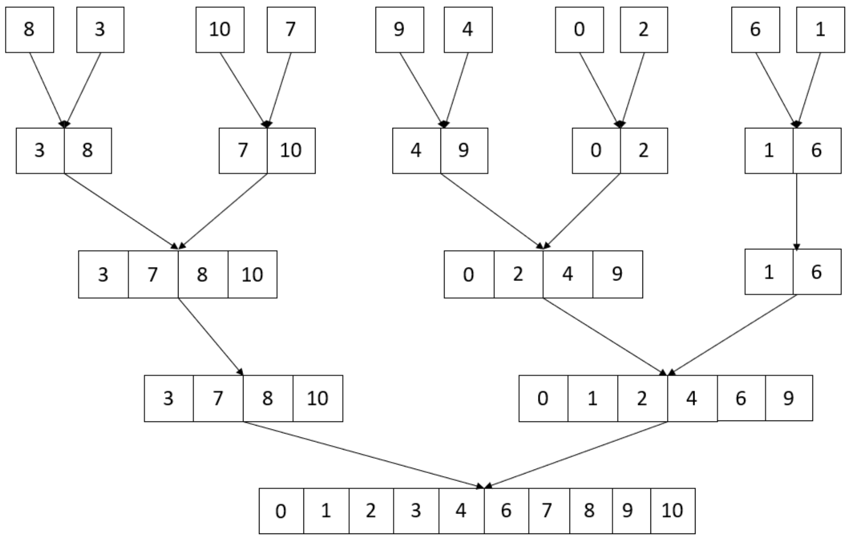

Figure 2: Merging the given array. The depth of the tree is $O(log_2(n))$.

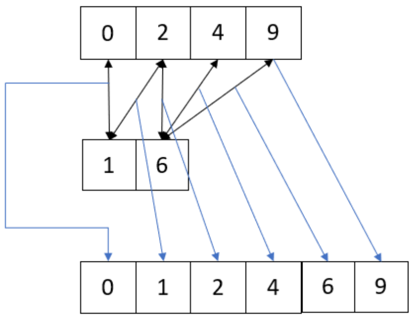

Figure 3: The process of merging two sorted arrays: [0,2,4,9] and [1,6]. The black arrows indicate the element-wise comparison while the blue arrows indicate the result of the comparison in the output array. Initially, the leftmost elements are compared - 0 and 1. The result 0 is then stored in the first element of the output array. The pointer of the first array and the output array is moved by 1. Then the comparison of 2 and 1 is made. The process is continued until there are no elements in any of one of the arrays i.e. the pointer reaches the end of one of the arrays. The remaining elements are then appended at the end of the output array.

Task 2

No, if a problem can be solved by Divide and Conquer techniques, Dynamic Programming (DP) is not guaranteed to apply. Divide and Conquer algorithm surprisingly resembles DP algorithms as both of them deal with dividing a larger problem into multiple sub-problems. However, Dynamic Programming can be thought of as an extension of Divide and Conquer problems. The Divide and Conquer problems need **overlapping sub-problems** and **optimal substructure** in order to be approached as a Dynamic Programming Problem.

Overlapping sub-problem means the same sub-problem that will occur multiple times in a recursion tree. If we find overlapping sub-problems, we can simply solve that sub-problem once and store it in the memory. This technique is called Memoization or Tabulation depending on how we approach the problem. For example - while computing the Fibonacci numbers, we need to calculate the second Fibonacci value multiple times as the next two values depend on that value. But instead of calculating

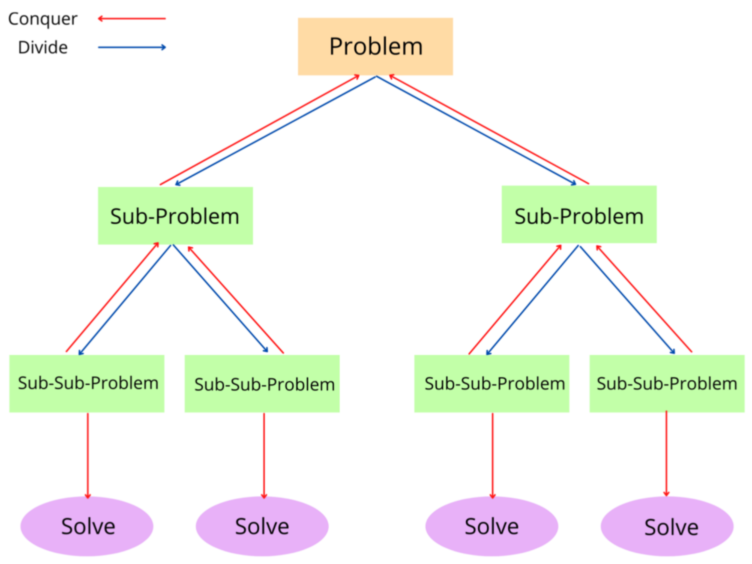

Figure 4: General Divide and Conquer approach

these values multiple times, we can simply store these values in an array, and retrieve the value from the array when needed.

Another aspect of a DP problem is having optimal substructure i.e. combining the best results from the sub-problems gives us the best result from the merged problem. Let's look at the rod-cutting problem where if we have the best value for a rod of length 1, then combining them gives us a value for rod length 2. Again, we can have another value for rod length 2 without dividing it into two rods of length 1. Therefore, the best value of a rod with length 2 is the maximum between the best values of two rods of length and the value of a rod with length 2. This form of problem formulation is called optimal substructure.

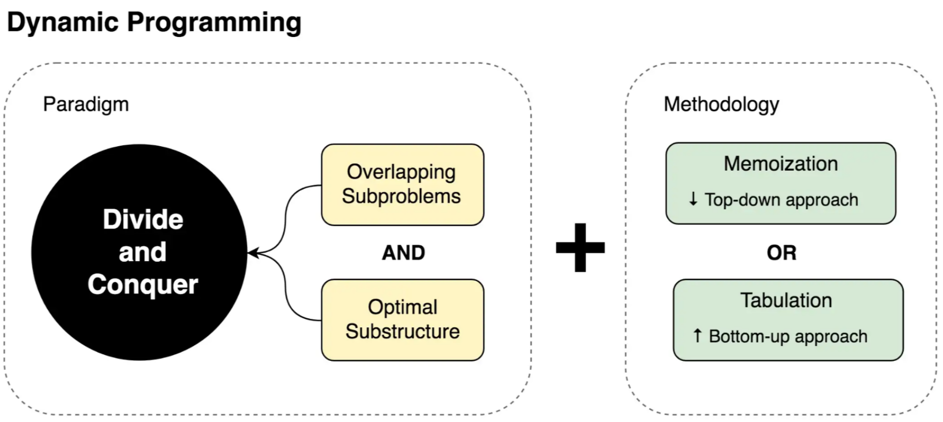

Figure 5: The Dynamic Programming (DP) paradigm and methodology - DP is an extension of Divide and Conquer algorithms and uses Memoization and Tabulation

In order to solve a Divide and Conquer problem as a DP problem our problem needs to satisfy the above two conditions - overlapping sub-problems and optimal substructure. Otherwise, DP will not be applicable to those problems.

Task 3

Idea of the Naive Algorithm

1. The algorithm is similar to the merge step of the merge sort algorithm. But instead of merging two sorted arrays, we merge multiple sorted arrays.

2. Initially, we need a create an output array of size $N * k$.

3. We fix a pointer at the start of every array. For each array, we compare the pointed element and store the minimum in the output array.

4. The pointer of the array with the minimum value and the pointer of the output array is incremented by 1.

5. Repeat step 4-5 until the pointer of $(N - 1)$ arrays reaches the end of the arrays.

6. For the remaining elements of our last array, we append them at the end of our output array and return the output array.

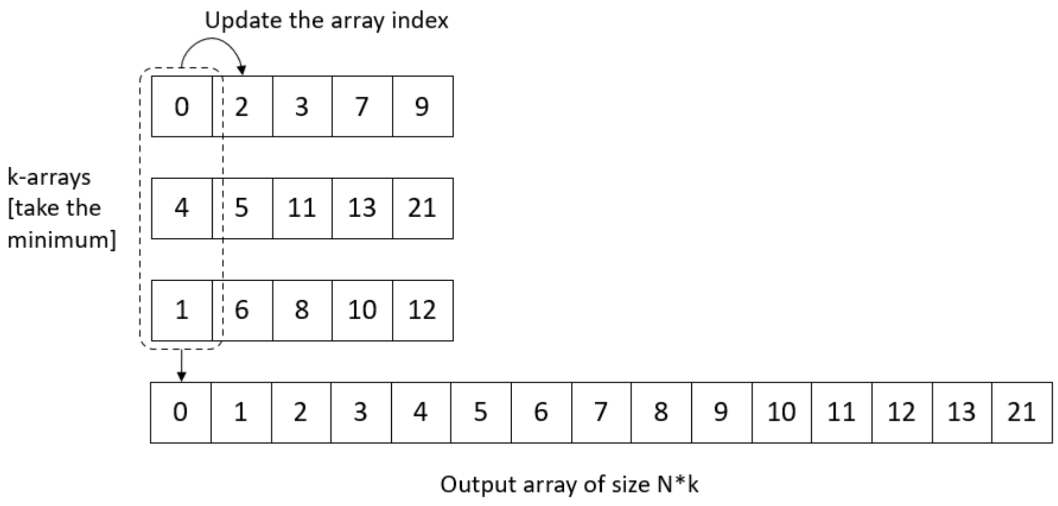

Figure 6: Naive Approach for merging k sorted arrays

Time Complexity of the Naive Algorithm

To find the minimum value it takes $O(k)$ in each step. The algorithm makes a total of $O(N * k)$ comparisons as the algorithm needs to take the minimum value for every element in the k arrays and in total, there are $N * k$ elements. So, the overall time complexity is $O(N * k^2)$.

Idea of the Efficient Algorithm

1. The idea of the algorithm is similar to the merging step in Figure-2 where we merge every 2 array.

2. Step-1 is repeated until all the arrays are merged into a single array and then return that array.

Pseudocode of the Efficient Algorithm

```
merge2array (array a, array b):
    array temp = []
    while a and b are non-empty:
        if a[0] > b[0]:
            append b[0] to temp
            remove b[0] from b
        else
            append a[0] to temp
            remove a[0] from a
    while a is non-empty:
        append a[0] to temp
        remove a[0] from a
    while b is non-empty:
        append b[0] to temp
        remove b[0] from b
    return temp

mergeKarray (array L):
    //L is a list of arrays of size N to be merged
    if length(L) == 1: //There is only one array in the list of arrays
        return array L[0]

    new_list = [] //stores a list of arrays

    for every two array i, j:
        merged_array = merge2array(i, j)
        append merged_array to new_list

    mergeKarray(new_list)
```

Time Complexity of the Efficient Algorithm

The merge2array() function is simply the merge operation done in the merge sort and takes $O(N)$ time. However, the merge operation is done for every 2 arrays. Hence, the outer for loop would take $O(N * \frac{k}{2}) = O(N * k)$ time complexity as it needs to merge $\frac{k}{2}$. The recursion tree has a depth of $log_2(k)$ and hence the overall time complexity of the algorithm is $O(N * k * log_2(k))$

Task 4

Idea of the Naive Algorithm

1. For a number x we iterate over all the numbers at the right side of x, and compare the number with x
2. If the encountered number is smaller than x, we increment the count by 1
3. Step - 1, 2 are repeated for all the numbers in the array

Psuedocode of the Naive Algorithm

```
count_smaller(array arr):
    count_arary = []
    for i in range(n):
        count = 0
        for j = i+1...n:
            if arr[i]>arr[j]:
                count++
        count_array[i] = count
    return count_array
```

Complexity of the Naive Algorithm

The for-loops in the pseudocode iterate over all the numbers. Both for loops have an algorithmic complexity of $O(n)$ making the overall complexity of the algorithm $O(n^2)$

Idea of the Efficient Algorithm

1. We copy the input array to another array and sort it using merge sort.
2. For every element of the input array, we search it in the sorted array using binary search. Binary search gives us the index of the number in the sorted array and the index is stored in the output array.
3. The searched number is removed from the sorted array.
4. Repeat step - 2, 3 for all the elements of the input array and then return the output array.

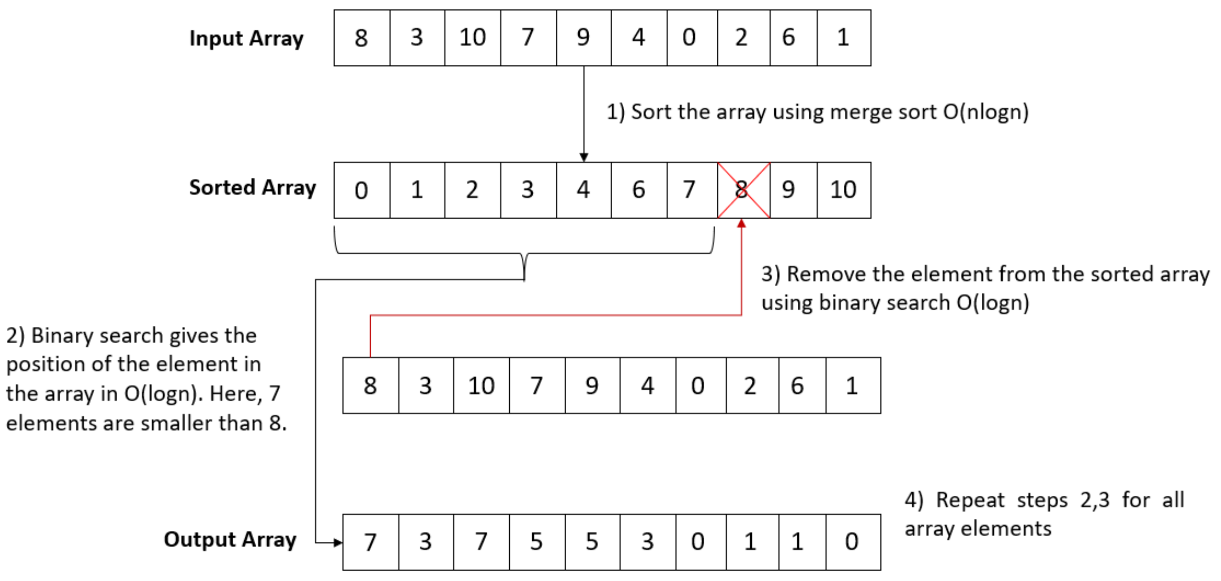
Input Array
8 3 10 7 9 4 0 2 6 1
1) Sort the array using merge sort O(nlogn)
Sorted Array
0 1 2 3 4 6 7 8 9 10
3) Remove the element from the sorted array using binary search O(logn)
2) Binary search gives the position of the element in the array in O(logn). Here, 7 elements are smaller than 8.
8 3 10 7 9 4 0 2 6 1
4) Repeat steps 2,3 for all array elements
Output Array
7 3 7 5 5 3 0 1 1 0

Figure 7: Efficient algorithm for counting the smaller numbers at the right side of a number

Psuedocode of the Efficient Algorithm

```
count_smaller(array arr):
    sorted_arr = mergeSort(arr)
    output_arr = []
    len = length(arr)
    for i = 0...len-1:
        output_arr[i] = binary_search(sorted_arr, arr[i])
        //Binary search function takes the array and the element to be searched as input
    return output_arr
```

Complexity of the Efficient Algorithm

The algorithm first uses merge sort which takes $O(nlogn)$ time complexity and then uses binary search for n elements making the complexity. The complexity of binary search for one element is $logn$ and for n elements, it is $nlogn$. Hence the overall time complexity is $O(nlogn)$.

Task 5

Idea of the Naive Algorithm

1. We have two sorted arrays and we merge them using the merge function of merge sort.
2. If the number of elements of the merged array is odd, the middle index is returned as the median.
3. If the number of elements of the merged array is even, the average of the two middle indices is returned as the median.

Pseudocode of the Naive Algorithm

```
find_median_2array_naive(array a, array b):
    merged_array = merge(a, b)
    len = length(merged_array)
    if len is odd:
        //We assume the index of the array starts from 0
        return merged_array[(len-1)/2]
    else:
        return (merged_array[len/2] + merged_array[len/2 - 1])/2
```

Complexity of the Naive Algorithm

The merge function can merge two sorted arrays in linear time. If the size of one array is N and the size of another array is M, then the complexity for merging is $O(min(M, N))$. To find the median it takes constant or $O(1)$ time. The overall complexity of the algorithm is $O(min(M, N))$.

Idea of the Efficient Algorithm

1. We find the median of the input arrays and compare them.
2. If the median of the first array is smaller than the median of the second array, then we reduce the search space by taking the right half of the first array and the left half of the second array.
3. If the median of the second array is smaller than the median of the first array, then we reduce the search space by taking the right half of the second array and the left half of the first array.
4. Step 1-3 are repeated until the resulting array is of length 1 or 2.
5. If the length of the last array is 1, the only element of the array is returned as the median.
6. If the length of the last array is 2, the average of the two elements is returned as the median.

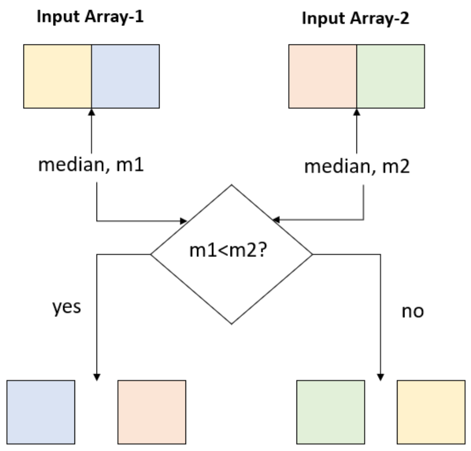

Figure 8: Efficient algorithm for finding the median for two sorted arrays

Pseudocode of the Efficient Algorithm

```
find_median(array a):
    len = length(a)
    if len is odd:
        //We assume the index of the array starts from 0
        return a[(len-1)/2]
    else:
        return (a[len/2] + a[len/2 - 1])/2

find_median_2array(array a, array b):
    l1 = length(a)
    l2 = length(b)
    if (l1 == 1 or 2) or(l2 == 1 or 2):
        return find_median_2array_naive(a, b)

    m1 = find_median(a)
    m2 = find_median(b)

    if (m1<m2):
        find_median_2array(a[m1:], b[:m2])

    else (m1>=m2):
        find_median_2array(a[:m1], b[m2:])
```

Complexity of the Efficient Algorithm

The algorithm uses recursion which divides the search space by two in each call making the depth of the recursion tree $O(log(N + M))$. Comparing the median takes constant time, $O(1)$, and the base case of finding the median between arrays of length 2 also takes constant time. Finding the median in a single sorted array also takes constant time. Overall, the time complexity of the algorithm is $O(log(N + M))$.