Assignment 3

Due: January 27, 2022

Your solutions must be typed (preferably typeset in LaTeX) and submitted as a PDF through Canvas before the beginning of class on the day its due.
When asked to provide an algorithm you need to give well formatted pseudocode, a description of how your code solves the problem, and a brief argument of its correctness.

Problem 1: Median of Medians The 'Median-of-medians' selection algorithm presented in class divides the input into groups of 5. Using a group of odd size helps keep things a little simpler (because otherwise the group medians are messier to define), but why the choice of 5?

(a) [10 points] Show that the same argument for linear worst-case time complexity works if we use groups of size 7 instead.

Answer: - In the question, it is given that the size of the group is 7. Therefore, the sub lists will be of the size [n/7] where each group consists of at most 7 elements.

Let S be the set of given n elements.
Let M be the list consisting of the medians from the lists mentioned above.
Let "m" define the median of the medians.

Therefore, we can say that in the list, the number of elements greater than "m" is at-least $4([n/14]) \geq 2n/7$.

Similarly, we can say that, the number of elements less than "m" is at least $\geq(2n/7)$.

Therefore, we can say that for the recurrence of the worst case complexity, subset size can be given as :-
$n-(2n/7) = (5n/7)$.

Therefore, the recurrence relation can be formulated as : -
$T(n) = T(\alpha n) + T(ßn)+an$

Replacing the values as
$\alpha = 1/7$
$ß = 5/7$

We get the recurrence relation as follow: -
$T(n) = T(n/7) + T(5n/7) + an$ for any $a > 0$

Now, we can see that for this relationship, we need to prove that "an" has an upper bound O(n).

For this, let us assume that for any large a, $T(n) \leq cn$
Therefore, the recurrence relation can be re-written as: -
$T(n) \leq T(n/7) + T(5n/7) + an$ -- (1)
Now, as we know that $T(n) \leq c(n)$ -- (2)

From (1) and (2), we can say that,
$T(n) \leq c'(n/7) + c'(5n/7) + an$
$T(n) \leq 6c'n/7 + an$
$T(n) \leq c'n$ iff $a>7c'$
Therefore, we can say that
$T(n) \in O(n)$

(b) [10 points] Show that groups of size 3 results in superlinear, $\omega(n)$, time complexity.

Answer: -
According to the question, we have,
Group size = 3
Sub-lists size = n/3 where n is the set of n elements with at most 3 elements each.

Let us assume that
M = list containing medians of different lists
"m" = median of medians
Now, from the above, we can say that n/3 elements would definitely be greater than "m" and n/3 less. But for the worst case, it turns out to be 2n/3 elements.

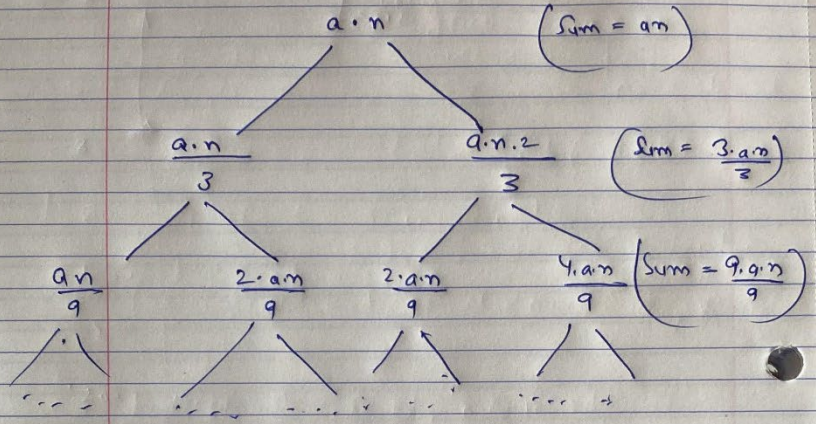
From this, we can say that the number of elements greater than "m" in the list is at least: - 2[n/6].

Hence, for the worst case complexity., the size of the subset will be:
n-(n/3) = 2n/3

Therefore, the recurrence relation can then be stated as follows: -
$T(n) = T(n/3) + T(2n/3) + an$ for all $a>0$.'

Now, we can say that the recurrence tree for this equation will be as follows: -

$T(n) \leq T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + an$



Sum at each level = an

Now, we will assume that the height of the tree is 'k'

we can say that

$T(n) = an + \ldots\ldots + an$ (k times)

$T(n) = a.n.$ (k times) $= a.n.k$

since, we know that $a.n.k$ for $n = 2^k$

$\Rightarrow K = \log_2(n)$

$\Rightarrow T(n) = an \log_3 n$ → final grouping

$\Rightarrow T(n) = O(n * \log n)$

Problem 2: Maximum Subarray Sum The Maximum Subarray Sum problem is the task of finding the contiguous subarray with largest sum in a given array of integers. Each number in the array could be positive, negative, or zero. For example: Given the array $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ the solution would be $[4, -1, 2, 1]$ with a sum of 6.

(a) [5 points] Give a brute force algorithm for this problem with complexity of $O(n^2)$.

Answer: -

Now, as we know that subarrays are arrays inside another array which only contains contiguous elements.
Now, for finding out the maximum subarray sum using brute force, we need to calculate the sum of each possible subarray and compare , then return the highest value.

For this, we have,

1) Let us define two variables namely currsum and the maxsum
   Currsum: -Represents the sum of the current subarray and it will be given a default value of the first element of the array.
   Maxsum: - maximum sum , before we start, we assume that the first element of the array provides the maximum sum.

   Let currsum = nums[0]

   Let maxsum = currsum

   Using brute force, we will need to loop through the array twice to calculate all sums of the arrays.

The first loop will be :-

```
For ( let i=0;i<nums.length;i++)

{


Currsum = nums[i];

If(currsum>maxsum)
{
```

Maxsum = currsum //to check whether maxsum is changed and if yes, then update it.

}

The second loop will be to check whether we have gone through each subwarry or not.

```
For( let j = i+1; j< nums.length; j++)
{
   Currsum = currsum + nums[j];
}
```

Once we have done with this loop, maxsum will hold our answer.

The entire algo can be written as follows: -

```
var
maxSubArray
= function
(nums) {

                    let currSum = nums[0]
                    let maxSum = currSum;


                    for (let i = 0; i < nums.length; i++) {
                       currSum = nums[i];
                       if (currSum > maxSum) {
                          maxSum = currSum
                       }
                       for (let j = i + 1; j < nums.length; j++) {
                          currSum = currSum + nums[j];
                          if (currSum > maxSum) {
                             maxSum = currSum
                          }
                       }
                    }

                    return maxSum;

                 };
```

Now, as know that there are two loops running for the entire length of the array (n), therefore, we can argue that the time complexity for this algorithm will be equal to O(n^2).

(b) [10 points] Give a divide and conquer algorithm for this problem with complexity $O(n \log n)$.

Answer: -

Using the divide and conquer approach for this problem, we have can divide it into two bigger steps as mentioned below: -

1) Divide the given array into two halves
2) Return the maximum of the following three scenarios
    a. Maximum subarray in the left half
    b. Maximum subarray in the right half
    c. Maximum subarray sum such that the subarray crosses the midpoint.

The idea behind this is approach is that find the maximum sum starting from mid point and ending at some point on left of mid, then find the maximum sum starting from mid + 1 and ending with some point on right of mid + 1. Finally, combine the two and return the maximum among left, right and combination of both

In the form of an algorithm, this can written as: -

```
findMaximumSum(int[] nums, int left, int right)
  {
     // If the array contains 0 or 1 element
     if (right == left) {
        return nums[left];
     }

     // Find the middle array element
     int mid = (left + right) / 2;

     // Find maximum subarray sum for the left subarray,
     // including the middle element

// Return maximum of following three
    //  possible cases:
    //  a) Maximum subarray sum in left half
   //    b) Maximum subarray sum in right half
    //   c) Maximum subarray sum such that the
  //      subarray crosses the midpoint

     int leftMax = Integer.MIN_VALUE;
     int sum = 0;
     for (int i = mid; i >= left; i--)
     {
```

```
            sum += nums[i];
            if (sum > leftMax) {
                leftMax = sum;
            }
        }

        // Find maximum subarray sum for the right subarray,
        // excluding the middle element
        int rightMax = Integer.MIN_VALUE;
        sum = 0;    // reset sum to 0
        for (int i = mid + 1; i <= right; i++)
        {
            sum += nums[i];
            if (sum > rightMax) {
                rightMax = sum;
            }
        }

        // Recursively find the maximum subarray sum for the left
        // and right subarray, and take maximum
        int maxLeftRight = Integer.max(findMaximumSum(nums, left, mid),
                                findMaximumSum(nums, mid + 1, right));

        // return the maximum of the three
        return Integer.max(maxLeftRight, leftMax + rightMax);
    }

    // Wrapper over findMaximumSum() function for the base case.
    public static int findMaximumSum(int[] nums)
    {
        // base case
        if (nums == null || nums.length == 0) {
            return 0;
        }

        return findMaximumSum(nums, 0, nums.length - 1);
    }
```

This approach takes O(n logn) time complexity as explained below: -
findMaximumSum() is a recursively calling method finding out the left and right subcases.
Therefore, the time complexity can be explained by the recurrence relation as mentioned below: -
$T(n) = 2T(n/2) + \Theta(n)$

Upon solving this recurrence relation, we get the time complexity as :- O(n logn).

(c) [10 points] Give a dynamic programming algorithm for this problem with complexity $O(n)$.

Answer: - Now for solving this problem using dynamic programming, we will follow the steps mentioned below: -

1) Initialize the variables: -
   Max_so_far = INT_MIN
   Max_ending_here =0

   We have to loop through each element of the array,

   a) Max_ending_here = Max_ending_here+a[i]
   (b) if(max_so_far < max_ending_here)

   max_so_far = max_ending_here

(c) if(max_ending_here < 0)

   max_ending_here = 0

return max_so_far

The basic approach to look for in the kadane's algorithm is to look for all the positive contiguous segments of the array (max_ending_here) and then we have to keep track of the sum of the contiguous segments of all the positive segments (max_So_far).

Then every time we get a positive -sum,. Then we have to compare it max_so_far and update it if it is greater than max_so_far.

There can be two approaches for algorithm which are defined below: -

1) The first approach does not use recursion

```
maximumSubarraySum(int[] arr) {

  int n = arr.length;

  int maxSum = Integer.MIN_VALUE;

  int currSum = 0;

  for (int i = 0; i <= n - 1; i++) {
```

```
    currSum += arr[i];

    if (currSum > maxSum) {

      maxSum = currSum;

    }

    if (currSum < 0) {

      currSum = 0;

    }

  }

  return maxSum;

}
```

Here, the time complexity is O(n) where n is the size of the array. This is because we have to iterate through the loop once and then we can find the required answer.

2) The second approach uses recursion for which the algorithm is mentioned below:-

```
maxSubArraySum(int a[])
    {
       int size = a.length;
       int max_so_far = Integer.MIN_VALUE, max_ending_here = 0;

       for (int i = 0; i < size; i++)
       {
          max_ending_here = max_ending_here + a[i];
```

```
        if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;
        if (max_ending_here < 0)
            max_ending_here = 0;
    }
    return max_so_far;
}
```

Here, also, we can see that the time complexity comes out to be O(n).