



# Assignment 6

## GROUP 16

Group members:

Gaurav N (IIT2019231)

Vinit W (IIT2019232)

Prince (IIT2019233)



# Abstract

For a given large binary string, the project aims at calculating the length of substring which is having maximum difference of number of 0s and number of 1s.



# Table of Contents

- Introduction
- Algorithm Design
- Algorithm Implementation
- Algorithm Analysis
- Experimental Study
- Conclusion



# Introduction

A string that consists of only 1's and 0's is said to be a binary string.

In our project, we tested various algorithms on different sample test cases for time complexity and space complexity. In our report, we tend to focus on establishing a rule or a relationship between the time and input data.

We have analysed three different approaches -

1. Brute force
2. Recursive 2D DP
3. Iterative 1D DP



# Algorithm Design - Brute force

## Approach

In brute force implementation, for every possible sub array of every possible size, we calculate difference of number of 0's and number of 1's. This can be done using combination of 3 nested loops. We keep track of the maximum value of difference and corresponding sub array size till that instant and print its value in the end.



# Algorithm Design - Brute force

## Algorithm

1. The outermost loop keeps track of current size of the sub array.
2. The middle loop will keep track of current starting index of the sub array.
3. The innermost loop will traverse the current sub array and count the number of 0's and number of 1's for it.
4. For every sub array, we find the difference of number of 0's and number of 1's. If this difference exceeds maximum value of difference till that instant, we assign current difference's value to maximum difference and update size accordingly.
5. Finally, we print the size of the sub array corresponding to the maximum value of difference.



## Algorithm Design- Recursive 2D DP

### Approach

In this approach, we will convert given binary string into integer array such that if  $\text{string}[i] = '0'$  then  $\text{arr}[i] = -1$  and if  $\text{string}[i] = '1'$  then  $\text{arr}[i] = 1$ .

Now, for each index we need to make decision whether to take it or skip it. So, we declare a 2D array to memorize result for that index.



# Algorithm Design - Recursive 2D DP

## Algorithm

1. The idea is to recursively find size of sub array corresponding to indices until base case (i.e. end of the binary string is reached).
2. Create a function that checks if the given binary string consists of all 1's. If it consists of all 1's, the answer will be 0.
3. Create a recursive function to calculate maximum difference of number of 0's and number of 1's corresponding to each index in the given binary string.
4. The base case of recursion is when end of the given binary string is reached.
5. Print the answer.





# Kadane's Algorithm

Given an array, Kadane's algorithm is able to find the maximum sum and size of a contiguous sub array in an array with a run time of  $O(n)$ . Kadane's algorithm uses the concept of dynamic programming.

# Algorithm Design - Iterative 1D DP

## Approach

In this approach, we convert all 0's in the binary string into 1's and all 1's into -1's. Now, all we have to do is to find out the size of subarray with maximum Using Kadane's Algorithm.

-1	1	1	-1	-1	...	1		-1
↑	↑	↑	↑	↑		↑		↑
1	0	0	1	1	...	0		1



# Algorithm Design - Iterative 1D DP

## Algorithm

1. In this implementation, we traverse the binary string from left to right.
2. We maintain four variables. They will signify values of current sum, maximum sum and current size, maximum size.
3. If the  $i$ th character of the binary string is 0 then we will add 1 to the current sum else we will decrement current sum by 1.
4. At any instant, if current sum becomes less than 0, we set current sum and current size to zero. At any instant, if current sum becomes greater than maximum sum, we set maximum sum to current sum and maximum size to current size.
5. Print the maximum size.



# Algorithm Implementation - Brute force

## Inputs:

N, string str

**procedure** *findMaxLength*(N, string str)

## Initialize:

size=0,count0s=0,count1s=0,maxLength=0,maxDifference=0, i,j,tempDifference

**for** size = 1; size < n; size++ **do**

**for** i = 0; i < n-size; i++ **do**

        count0s = 0

        count1a = 0

**for** j = i; j < size+i; j++ **do**

**if** str[j] == 0 **then**

                count0s++;

**end if**

**if** str[i] == 1 **then**

            count1s++;

**end if**

**end for**

    tempDifference = (count0s-count1s);

**if** tempDifference>=maxDifference  
    **then**

        diff\_max = temp;

        length\_max=size;

**end if**

**end for**

**end for**

**return** maxLength

**end procedure**



# Algorithm Implementation - Recursive 2D DP

```
Int dp[n+1][2];  
  
dp[0][0]=0,dp[0][1]=0;  
  
for(int i=0;i<n;i++){  
  
    (a[i]==1)?a[i]=-1:a[i]=1;  
  
}  
  
for(int i=1;i<=n;i++){  
  
    dp[i][0]=max(dp[i-1][1],dp[i-1][0]);  
  
    dp[i][1]=max(dp[i-1][1]+a[i-1],0);  
  
}  
  
cout<<max(dp[n][1],dp[n][0]);
```



## Algorithm Implementation - Iterative 1D DP

### Inputs:

N, string str

**procedure** *findMaxLength*(N, string str)

### Initialize:

summax = 0, sumcurrent = 0, sizemax = 0,

sizecurrent = 0, i = 0

**for** i = 0; i < n; i++ **do**

**if** str[i] == 0 **then**

        sumcurrent = sumcurrent + 1

**Else**

        sumcurrent = sumcurrent - 1

**end if**

sizecurrent = sizecurrent + 1

**if** sumcurrent < 0 **then**

    sumcurrent = 0

sizecurrent = 0

**end if**

**if** sumcurrent > summax **then**

    summax = sumcurrent

sizemax = sizecurrent

**end if**

**end for**

**return** sizemax

**end procedure**



## Algorithm Analysis - Brute force

In case of brute force algorithm, three nested loops are needed to for calculation of difference of number of 0's and number of 1's for all possible sub arrays, so the overall time complexity is  $O(n^3)$  while no extra space is required resulting into  $O(1)$  space complexity.

Time complexity:  $O(n^3)$

Space complexity:  $O(1)$



## Algorithm Analysis - iterative 2D DP

In case of recursive implementation, the concept of 2D dynamic programming is used. The height of the recursive tree is same as the length of the given binary string, so the time complexity is  $O(n)$ . The memorization for the DP will also be proportional to the length of the given binary string, resulting in  $O(n)$  space complexity.

Time complexity: $O(n)$

Space complexity: $O(n)$





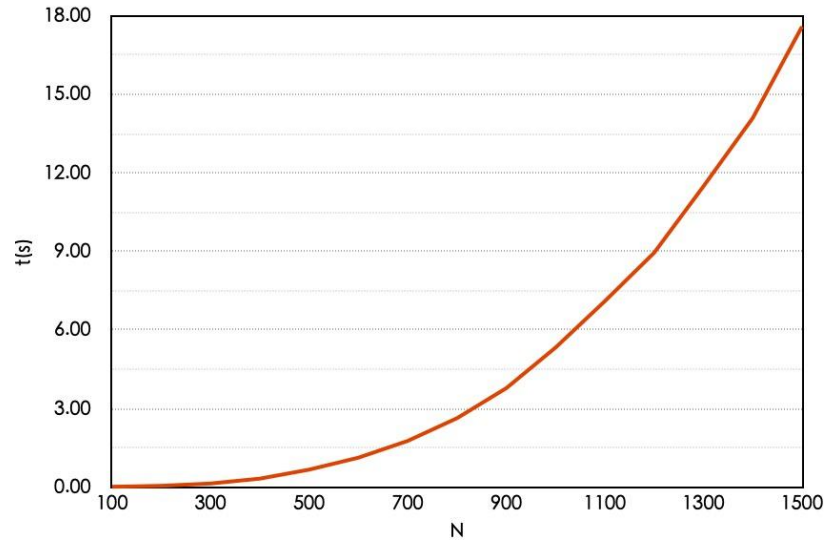
## Algorithm Analysis - Iterative 1D DP

In case of iterative implementation, the concept of 1D dynamic programming is used. One full array traversal is needed, so the time complexity is  $O(n)$ . In this implementation, we don't need to tabulate all the sub results, but only the recently derived ones. This results in  $O(1)$  space complexity.

Time complexity: $O(n)$

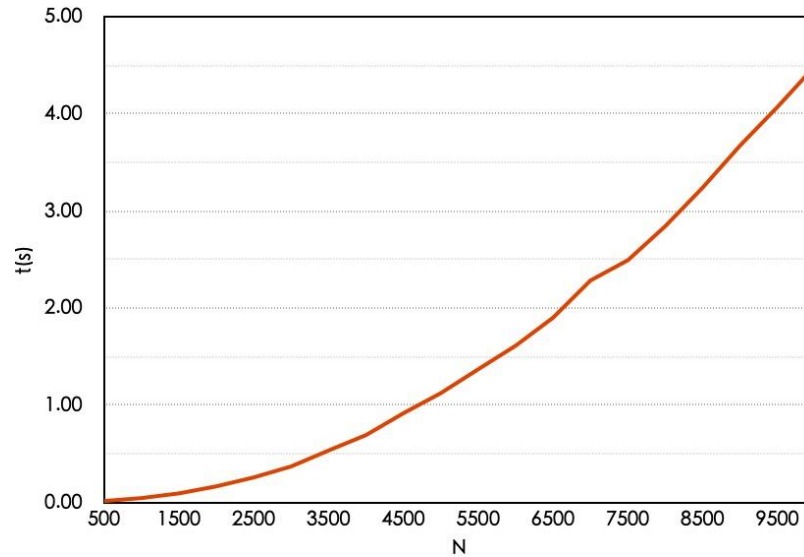
Space complexity: $O(1)$

# Experimental Data



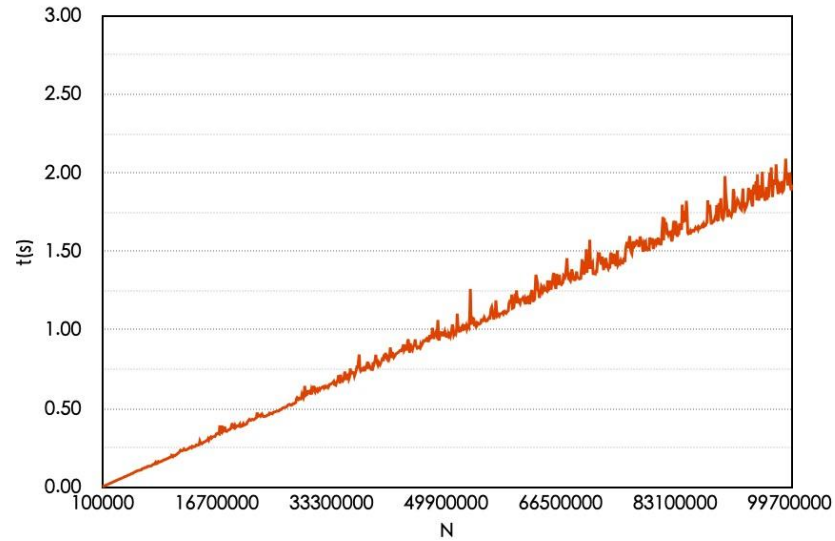
The graph for the first algorithm shows the non-linear behaviour as  $n$  increases, this behaviour is in fact of  $n^3$ .

# Experimental Data



The graph for the second algorithm shows the linear behaviour as  $n$  increases, this behaviour is in fact of  $n$ .

# Experimental Data



The graph for the third algorithm shows the linear behaviour as  $n$  increases, this behaviour is in fact of  $n$ .



## Conclusion

From the experimental result we concluded that the average running time of Dynamic Programming algorithm is best which can be observed from the mutual graphs of Brute Force algorithm and Dynamic Programming algorithm.



# Thank you!