Design and Analysis of Algorithms

Assignment 3
Group 16

Gaurav N*, Vinit W†and Prince‡
Department of Information Technology, Indian Institute of Information Technology
Allahabad
ID: *IIT2019231, †IIT2019232, ‡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.**

I. Introduction

A string that consists of only 1's ans 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 -

(i) Brute force
(ii) Recursive 2D DP
(iii) Iterative 1D DP

This report further contains -
II. Algorithm Design
III. Algorithm Implementation
IV. Algorithm Analysis
V. Experimental Study
VI. Conclusion

II. Algorithm Design

A) Brute force -
   a) 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.
   b) Algorithm:
      i) The outermost loop keeps track of current size of the sub array.
      ii) The middle loop will keep track of current starting index of the sub array.
      iii) The innermost loop will traverse the current sub array and count the number of 0's and number of 1's for it.
      iv) 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.
      v) Finally, we print the size of the sub array corresponding to the maximum value of difference.

B) Recursive 2D DP - Dynamic programming involves dividing a problem into similar sub-problems, and reusing the already calculated results for sub problems.
   a) Approach:
   In this approach, we will convert given binary string into integer array such that if $string[i]$ = '0' then $arr_{\mathrm{i}} = -1$ and if $string[i]$ = '1' then $arr_{\mathrm{i}} = 1$. 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.
   b) Algorithm:
      i) 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).
      ii) 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.
      iii) 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.
      iv) The base case of recursion is when end of the given binary string is reached.
      v) Print the answer.

C) Iterative 1D DP - 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.

a) 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 sub array with maximum Using Kadane's Algorithm.

b) Algorithm:

i) In this implementation, we traverse the binary string from left to right.

ii) We maintain four variables. They will signify values of current sum, maximum sum and current size, maximum size.

iii) 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.

iv) At any instant, if current sum becomes less than 0, we set current sum and current size to zero.

v) At any instant, if current sum becomes greater than maximum sum, we set maximum sum to current sum and maximum size to current size.

vi) Print maximum size.

III. Algorithm Implementation

Algorithm 1 Iterative 1D DP

Inputs:
N, *string str*
procedure findMaxLength(N, *string str*)
Initialize:
$sum_{\max} = 0$, $sum_{\text{current}} = 0$,$size_{\max} = 0$,
$size_{\text{current}} = 0$, $i = 0$
for $i = 0$; $i < n$; $i{+}{+}$ **do**
if $str_{\text{i}} == 0$ **then**
$sum_{\text{current}} = sum_{\text{current}} + 1$
else
$sum_{\text{current}} = sum_{\text{current}} - 1$
end if
$size_{\text{current}} = size_{\text{current}} + 1$
if $sum_{\text{current}} < 0$ **then**
$sum_{\text{current}} = 0$
$size_{\text{current}} = 0$
end if
if $sum_{\text{current}} > sum_{\max}$ **then**
$sum_{\max} = sum_{\text{current}}$
$size_{\max} = size_{\text{current}}$
end if
end for
return $size_{\max}$
end procedure

IV. Algorithm Analysis

A) 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)

.

B) Recursive 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)

.

C) 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)

.

V. Experimental Study

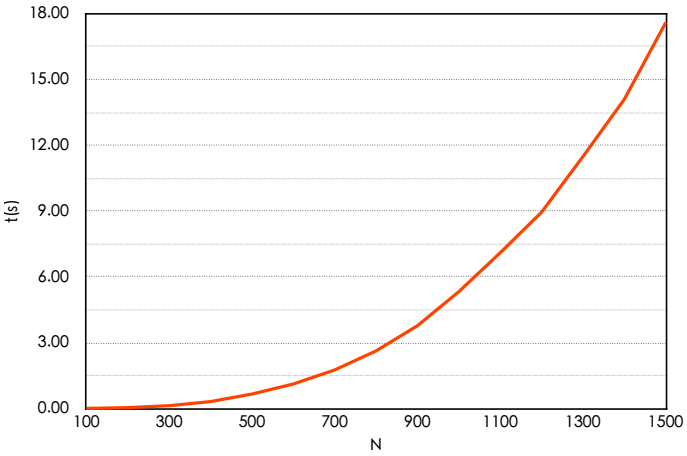

Fig. 1. N vs t for Brute force algorithm

Fig. 1. The graph for the first algorithm shows the non-linear behaviour as n increases, this behaviour is in fact of n^3.

Fig. 2. The graph for the second algorithm shows almost linear behaviour as n increases, this behaviour is in fact of n.

Fig. 3. The graph for the second algorithm shows the linear behaviour as n increases, this behaviour is in fact

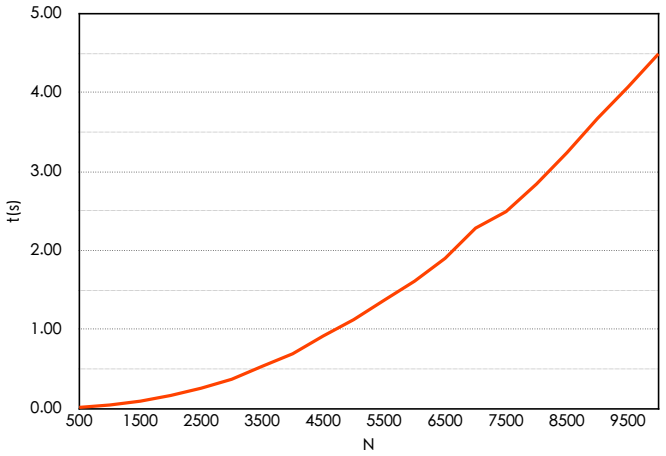

Fig. 2. N vs t for Recursive 2D DP algorithm

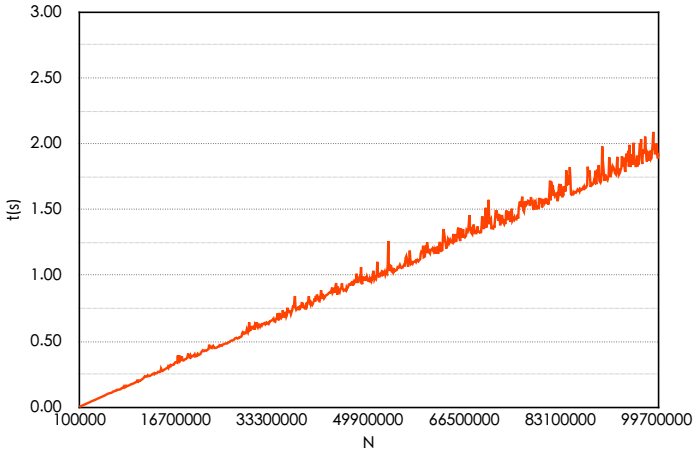

Fig. 3. N vs t for Iterative 1D DP algorithm

of n.

VI. Conclusion

From the mutual graphs of all three algorithms derived from experimental studies, we can conclude that the average running time of naive brute force implementation is worst of all three. The recursive 2D DP implementation takes significantly less time than plane brute force algorithm. By comparing the graphs of recursive and iterative algorithms, we can conclude that the iterative 1D DP algorithm is the fastest one.

One thing to note, as we have shown before in the algorithm analysis, the time complexity of recursive and iterative implementation both is the same i.e O(n). But after experimental analysis, we can see that the iterative algorithm performs significantly better than recursive one. This difference can be mapped to the fact that memorization can be slower than tabulation because of large number of recursive calls. Also, in the recursive implementation, we have to initialize memorization container of length n, while space needed in tabulation approach is constant.

One other disadvantage of recursive implementation is that for large binary strings, the recursion tree will be very deep, which will result into system running out of stack space and thereafter the program will crash.

References

[1] https://www.geeksforgeeks.org/maximum-difference-zeros-ones-binary-string/
[2] https://www.geeksforgeeks.org/maximum-difference-zeros-ones-binary-string-set-2-time/