A03G14Q14 : DAA ASSIGNMENT3

IV Semester B.Tech, Department of Studies in Information Technology,

Indian Institute of Technology Allahabad, Prayagraj, India

Abstract: In this paper, we have designed an algorithm that implements removal of all adjacent duplicates in String recursively.
We have discussed the time complexity of the algorithm by both Apriori and Apostriori analysis.

I. INTRODUCTION

Given a String, we are removing adjacent duplicate characters from String recursively. The output String should not have any adjacent duplicates.

Some Examples:

***Input** : azxxzy*
***Output**: ay*
First "azxxzy" is reduced to "azzy". The string "azzy" contains duplicates, so it is further reduced to "ay"

***Input**: acaaabbbacdddd*
***Output**: acac*

II. ALGORITHM DESCRIPTION

The following Algorithm demonstrate removal of duplicates in O(N) time:

- Start from the leftmost character and remove duplicates at left corner if there are any.
- The first character must be different from its adjacent now. Recur for Strings of length n-1(String without first character).
- Let the String obtained after reducing right substring of length n-1 be rem_str. There are three possible cases.

1. If first character of *rem_str* matches with the first character of original string, remove the first character from *rem_str*.
2. If remaining string becomes empty and last removed character is same as first character of original string. Return empty string.
3. Else, append the first character of the original string at the beginning of *rem_str*.

- Return *rem_str*.
- Print rem_str.

III. ALGORITHM AND ANALYSIS

Algorithm: DAA ASSIGNMENT3
Input **Str;**
Output result: **rem_str;**

```
ALGORITHM 1 func
function: remove(str,lastremoved)

step:1  if (str.length() == 0 || str.length() == 1)
          return str

step:2  if(str.charAt(0) == str.charAt(1))
        begin

step:3    lastremoved ← str.charAt(1)

step:4    while(str.length()>1&&str.charAt(0) == str.charAt(1))
          begin

step:5      str ← str.substring(1,str.length())

          end

step:6    str ← str.substring(1,str.length())

step:7    return remove(str,lastremoved)
        end
```

```
step:8   String remstr ←
         remove(str.substring(1,str.length(),lastremoved))
step:9   If(remstr.length()!= 0 && lastremoved ==
         str.charAt(0))
begin
    step:10 return remstr
end
step:11 return remstr
```

Algorithm 2 func

```
function main()
step12:  char lastremoved ← '\0'
step13:  str ← "acaaabbbacdddd"
step14:  remstr ←remove(str,lastremoved)
AlgEnds
```

III (a). Apriori Analysis

Table1: time complexity for Apriori analysis

	Time	FreqBest	Frequency
Step1	3	1	1
Step2	2	1	1
Step 3	1	1	1
Step 4	3	1	1
Step 5	2	1	1
Step 6	2	1	1
Step 7	T(n-1)	1	1
Step 8	T(n-1)	1	1
Step 9	3	1	1
Step 10	1	1	1
Step 11	1	1	1
Step 12	1	1	1
Step 13	1	1	1
Step 14	1	1	1

$T_\Omega \; \alpha \; 3*1+2*1+1*1+3*1+2*1+2*1+T(n-1)*1+ T(n-1)*1+3*1+1*1+1*1+1*1+1*1+1*1$

$T_\Omega \; \alpha \; 2*T(n-1)+21$ taking n=1
$\Omega(1)$

For N is length of string, lesser the length of string the algorithm takes minimum time for execution.

$T_\Omega \; \alpha \; 3*1+2*1+1*1+3*1+2*1+2*1+T(n-1)*1+ T(n-1)*1+3*1+1*1+1*1+1*1+1*1+1*1$

$T_\Omega \; \alpha \; 2*T(n-1)+21$

$T_o \; \alpha \; 2*T(n-1)+21$
$O(N)$

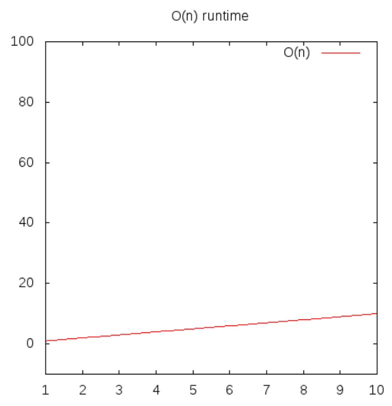

Figure1:Time complexity graph for apriori analysis.

IV. EXPERIMENTAL ANALYSIS AND PROFILING

Table2:Time complexity for Aposteriori analysis

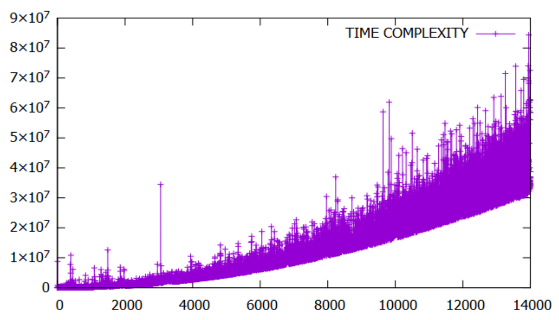

Figure2: Time complexity graph for aposteriori analysis.

In the above graph, the exponent 'N' is considered along x-axis and time along y-axis. The graph represents Time v/s N. as we increase the length of the string running time also increases.
The above graph shows that the Apostriori analysis is consistent with Apriori analysis.

CONCLUSION

From this paper we can conclude that the algorithm is an Ω(1)-O(N) time complexity algorithm is a linear function directly proportional to the length of string, the algorithm takes minimum time when the exponent N is small that is length of string is small and the algorithm gives best running time.

REFERENCES

[1]https://examples.javacodegeeks.com/recursion-java-example/

[2] https://www.javatpoint.com/program-to-find-the-duplicate-characters-in-a-string