GeeksforGeeks

A computer science portal for geeks

IDE Q&A GeeksQuiz

Suffix Tree Application 3 – Longest Repeated Substring

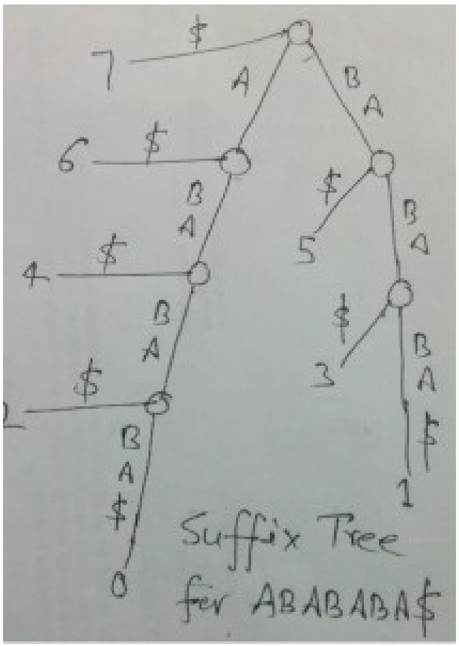
Given a text string, find Longest Repeated Substring in the text. If there are more than one Longest Repeated Substrings, get any one of them.

```
Longest Repeated Substring in GEEKSFORGEEKS is: GEEKS
Longest Repeated Substring in AAAAAAAAAA is: AAAAAAAAA
Longest Repeated Substring in ABCDEFG is: No repeated substring
Longest Repeated Substring in ABABABA is: ABABA
Longest Repeated Substring in ATCGATCGA is: ATCGA
Longest Repeated Substring in banana is: ana
Longest Repeated Substring in abcpqrabpqpq is: ab (pq is another LRS here)
```

This problem can be solved by different approaches with varying time and space complexities. Here we will discuss Suffix Tree approach (3rd Suffix Tree Application). Other approaches will be discussed soon.

As a prerequisite, we must know how to build a suffix tree in one or the other way.
Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:
Ukkonen's Suffix Tree Construction – Part 1
Ukkonen's Suffix Tree Construction – Part 2
Ukkonen's Suffix Tree Construction – Part 3
Ukkonen's Suffix Tree Construction – Part 4
Ukkonen's Suffix Tree Construction – Part 5
Ukkonen's Suffix Tree Construction – Part 6

Lets look at following figure:



This is suffix tree for string “ABABABA$”.
In this string, following substrings are repeated:
A, B, AB, BA, ABA, BAB, ABAB, BABA, ABABA
And Longest Repeated Substring is ABABA.
In a suffix tree, one node can’t have more than one outgoing edge starting with same character, and so if there are repeated substring in the text, they will share on same path and that path in suffix tree will go through one or more internal node(s) down the tree (below the point where substring ends on that path).
In above figure, we can see that

- Path with Substring “A” has three internal nodes down the tree
- Path with Substring “AB” has two internal nodes down the tree
- Path with Substring “ABA” has two internal nodes down the tree
- Path with Substring “ABAB” has one internal node down the tree
- Path with Substring “ABABA” has one internal node down the tree
- Path with Substring “B” has two internal nodes down the tree
- Path with Substring “BA” has two internal nodes down the tree
- Path with Substring “BAB” has one internal node down the tree
- Path with Substring “BABA” has one internal node down the tree

All above substrings are repeated.

Substrings ABABAB, ABABABA, BABAB, BABABA have no internal node down the tree (after the point where substring end on the path), and so these are not repeated.

Can you see how to find longest repeated substring ??
We can see in figure that, longest repeated substring will end at the internal node which is farthest from the root (i.e. deepest node in the tree), because length of substring is the path label length from root to that internal node.

So finding longest repeated substring boils down to finding the deepest node in suffix tree and then get the path label from root to that deepest internal node.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then find Longest Repeated Substring
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

    //pointer to other node via suffix link
    struct SuffixTreeNode *suffixLink;

    /*(start, end) interval specifies the edge, by which the
     node is connected to its parent node. Each edge will
     connect two nodes,  one parent and one child, and
     (start, end) interval of a given edge  will be stored
     in the child node. Lets say there are two nods A and B
     connected by an edge with indices (5, 8) then this
     indices (5, 8) will be stored in node B. */
    int start;
    int *end;

    /*for leaf nodes, it stores the index of suffix for
      the path  from root to leaf*/
    int suffixIndex;
};

typedef struct SuffixTreeNode Node;

char text[100]; //Input string
Node *root = NULL; //Pointer to root node

/*lastNewNode will point to newly created internal node,
  waiting for it's suffix link to be set, which might get
  a new suffix link (other than root) in next extension of
  same phase. lastNewNode will be set to NULL when last
  newly created internal node (if there is any) got it's
  suffix link reset to new internal node created in next
  extension of same phase. */
Node *lastNewNode = NULL;
Node *activeNode = NULL;

/*activeEdge is represeted as input string character
  index (not the character itself)*/
int activeEdge = -1;
int activeLength = 0;

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
```

```
    node->suffixLink = root;
    node->start = start;
    node->end = end;

    /*suffixIndex will be set to -1 by default and
      actual suffix index will be set later for leaves
      at the end of all phases*/
    node->suffixIndex = -1;
    return node;
}

int edgeLength(Node *n) {
    if(n == root)
        return 0;
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

void extendSuffixTree(int pos)
{
    /*Extension Rule 1, this takes care of extending all
    leaves created so far in tree*/
    leafEnd = pos;

    /*Increment remainingSuffixCount indicating that a
    new suffix added to the list of suffixes yet to be
    added in tree*/
    remainingSuffixCount++;

    /*set lastNewNode to NULL while starting a new phase,
     indicating there is no internal node waiting for
     it's suffix link reset in current phase*/
    lastNewNode = NULL;

    //Add all suffixes (yet to be added) one by one in tree
    while(remainingSuffixCount > 0) {

        if (activeLength == 0)
            activeEdge = pos; //APCFALZ

        // There is no outgoing edge starting with
        // activeEdge from activeNode
        if (activeNode->children[text[activeEdge]] == NULL)
        {
            //Extension Rule 2 (A new leaf edge gets created)
            activeNode->children[text[activeEdge]] =
                                          newNode(pos, &leafEnd);

            /*A new leaf edge is created in above line starting
             from  an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
```

```
             reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
            {
                //Start from next node (the new activeNode)
                continue;
            }
            /*Extension Rule 3 (current character being processed
              is already on the edge)*/
            if (text[next->start + activeLength] == text[pos])
            {
                //If a newly created node waiting for it's
                //suffix link to be set, then set suffix link
                //of that waiting node to curent active node
                if(lastNewNode != NULL && activeNode != root)
                {
                    lastNewNode->suffixLink = activeNode;
                    lastNewNode = NULL;
                }

                //APCFER3
                activeLength++;
                /*STOP all further processing in this phase
                and move on to next phase*/
                break;
            }

            /*We will be here when activePoint is in middle of
              the edge being traversed and current character
              being processed is not  on the edge (we fall off
              the tree). In this case, we add a new internal node
              and a new leaf edge going out of that new node. This
              is Extension Rule 2, where a new leaf edge and a new
            internal node get created*/
            splitEnd = (int*) malloc(sizeof(int));
            *splitEnd = next->start + activeLength - 1;

            //New internal node
            Node *split = newNode(next->start, splitEnd);
            activeNode->children[text[activeEdge]] = split;

            //New leaf coming out of new internal node
            split->children[text[pos]] = newNode(pos, &leafEnd);
            next->start += activeLength;
            split->children[text[next->start]] = next;

            /*We got a new internal node here. If there is any
              internal node created in last extensions of same
              phase which is still waiting for it's suffix link
              reset, do it now.*/
            if (lastNewNode != NULL)
            {
            /*suffixLink of lastNewNode points to current newly
              created internal node*/
                lastNewNode->suffixLink = split;
            }

            /*Make the current newly created internal node waiting
```

```
              for it's suffix link reset (which is pointing to root
              at present). If we come across any other internal node
              (existing or newly created) in next extension of same
              phase, when a new leaf edge gets added (i.e. when
              Extension Rule 2 applies is any of the next extension
              of same phase) at that point, suffixLink of this node
              will point to that internal node.*/
            lastNewNode = split;
        }

        /* One suffix got added in tree, decrement the count of
          suffixes yet to be added.*/
        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0) //APCFER2C1
        {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
        }
        else if (activeNode != root) //APCFER2C2
        {
            activeNode = activeNode->suffixLink;
        }
    }
}

void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}

//Print the suffix tree as well along with setting suffix index
//So tree will be printed in DFS manner
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
       // print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
           // if (leaf == 1 && n->start != -1)
             //   printf(" [%d]\n", n->suffixIndex);

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                  edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}
```

```
void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

/*Build the suffix tree and print the edge labels along with
suffixIndex. suffixIndex for leaf edges will be >= 0 and
for non-leaf edges will be -1*/
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

    /*Root is a special node with start and end indices as -1,
    as it has no parent from where an edge comes to root*/
    root = newNode(-1, rootEnd);

    activeNode = root; //First activeNode will be root
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}

void doTraversal(Node *n, int labelHeight, int* maxHeight,
int* substringStartIndex)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    if(n->suffixIndex == -1) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], labelHeight +
                                edgeLength(n->children[i]), maxHeight,
                                 substringStartIndex);
            }
        }
    }
    else if(n->suffixIndex > -1 &&
                (*maxHeight < labelHeight - edgeLength(n)))
    {
        *maxHeight = labelHeight - edgeLength(n);
        *substringStartIndex = n->suffixIndex;
    }
}

void getLongestRepeatedSubstring()
{
    int maxHeight = 0;
    int substringStartIndex = 0;
```

```
    doTraversal(root, 0, &maxHeight, &substringStartIndex);
//  printf("maxHeight %d, substringStartIndex %d\n", maxHeight,
//           substringStartIndex);
    printf("Longest Repeated Substring in %s is: ", text);
    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringStartIndex]);
    if(k == 0)
        printf("No repeated substring");
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AAAAAAAAAA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ABCDEFG$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ABABABA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "ATCGATCGA$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "banana$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "abcpqrabpqpq$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "pqrpqpqabab$");
    buildSuffixTree();
    getLongestRepeatedSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Run on IDE

Output:

```
Longest Repeated Substring in GEEKSFORGEEKS$ is: GEEKS
Longest Repeated Substring in AAAAAAAAAA$ is: AAAAAAAAA
Longest Repeated Substring in ABCDEFG$ is: No repeated substring
Longest Repeated Substring in ABABABA$ is: ABABA
Longest Repeated Substring in ATCGATCGA$ is: ATCGA
Longest Repeated Substring in banana$ is: ana
Longest Repeated Substring in abcpqrabpqpq$ is: ab
Longest Repeated Substring in pqrpqpqabab$ is: ab
```

In case of multiple LRS (As we see in last two test cases), this implementation prints the LRS which comes 1^{st} lexicographically.

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that finding deepest node will take O(N).
So it is linear in time and space.

Followup questions:

1. Find all repeated substrings in given text
2. Find all unique substrings in given text
3. Find all repeated substrings of a given length
4. Find all unique substrings of a given length
5. In case of multiple LRS in text, find the one which occurs most number of times

All these problems can be solved in linear time with few changes in above implementation.

We have published following more articles on suffix tree applications:

- Suffix Tree Application 1 – Substring Check
- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Generalized Suffix Tree 1
- Suffix Tree Application 5 – Longest Common Substring
- Suffix Tree Application 6 – Longest Palindromic Substring

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

0 Comments Category: Strings Tags: Pattern Searching

Related Questions:

- Shortest Superstring Problem
- Shortest Common Supersequence
- How to design a tiny URL or URL shortener?

- Remove spaces from a given string
- Online algorithm for checking palindrome in a stream
- Recursively print all sentences that can be formed from list of word lists
- Check if a given sequence of moves for a robot is circular or not
- Find the longest substring with k unique characters in a given string