Suffix Tree Application 6 - Longest Palindromic Substring - GeeksforGeeks

Suffix Tree Application 6 – Longest Palindromic Substring

Given a string, find the longest substring which is palindrome.

We have already discussed Naïve [O(n^3)], quadratic [O(n^2)] and linear [O(n)] approaches in Set 1, Set 2 and Manacher’s Algorithm.
In this article, we will discuss another linear time approach based on suffix tree.
If given string is S, then approach is following:

- Reverse the string S (say reversed string is R)
- Get Longest Common Substring of S and R **given that LCS in S and R must be from same position in S**

Can you see why we say that **LCS in R and S must be from same position in S** ?

Let’s look at following examples:

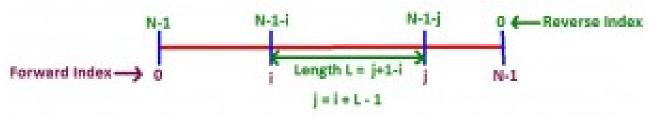
- For S = *xababayz* and R = *zyababax*, LCS and LPS both are ababa (SAME)
- For S = *abacdfgdcaba* and R = *abacdgfdcaba*, LCS is *abacd* and LPS is *aba* (DIFFERENT)
- For S = *pqrqpabcdfgdcba* and R = *abcdgfdcbapqrqp*, LCS and LPS both are *pqrqp* (SAME)
- For S = *pqqpabcdfghfdcba* and R = *abcdfhgfdcbapqqp*, LCS is *abcdf* and LPS is *pqqp* (DIFFERENT)

We can see that LCS and LPS are not same always. When they are different ?
When S has a reversed copy of a non-palindromic substring in it which is of same or longer length than LPS in S, then LCS and LPS will be different.
In 2nd example above (S = *abacdfgdcaba*), for substring *abacd*, there exists a reverse copy *dcaba* in S, which is of longer length than LPS *aba* and so LPS and LCS are different here. Same is the scenario in 4th example.

To handle this scenario we say that LPS in S is same as LCS in S and R **given that LCS in R and S must be from same position in S**.
If we look at 2nd example again, substring *aba* in R comes from exactly same position in S as substring *aba* in S which is ZERO (0th index) and so this is LPS.

The Position Constraint:

```
                         (Click to see it clearly)
```



We will refer string S index as forward index (S_i) and string R index as reverse index (R_i).
Based on above figure, a character with index i (forward index) in a string S of length N, will be at index N-1-i (reverse index) in it's reversed string R.
If we take a substring of length L in string S with starting index i and ending index j (j = i+L-1), then in it's reversed string R, the reversed substring of the same will start at index N-1-j and will end at index N-1-i.
If there is a common substring of length L at indices S_i (forward index) and R_i (reverse index) in S and R, then these will come from same position in S if $\mathbf{R_i = (N - 1) - (S_i + L - 1)}$ where N is string length.

So to find LPS of string S, we find longest common string of S and R where both substrings satisfy above constraint, i.e. if substring in S is at index S_i, then same substring should be in R at index $(N - 1) - (S_i + L - 1)$. If this is not the case, then this substring is not LPS candidate.

Naive [O($N*M^2$)] and Dynamic Programming [O(N*M)] approaches to find LCS of two strings are already discussed here which can be extended to add position constraint to give LPS of a given string.

Now we will discuss suffix tree approach which is nothing but an extension to Suffix Tree LCS approach where we will add the position constraint.

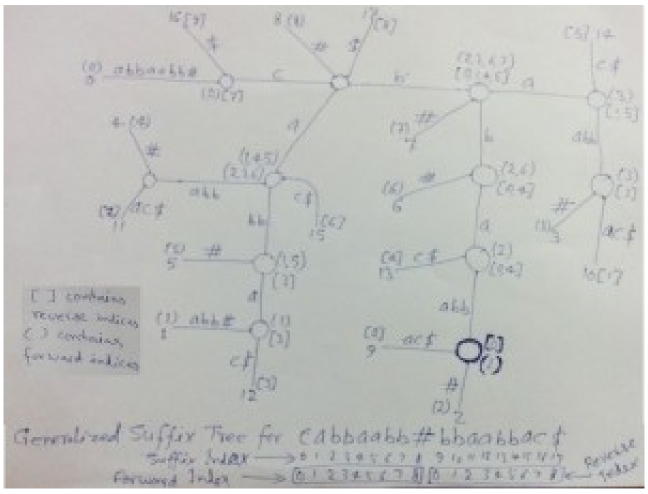
While finding LCS of two strings X and Y, we just take deepest node marked as XY (i.e. the node which has suffixes from both strings as it's children).
While finding LPS of string S, we will again find LCS of S and R with a condition that the common substring should satisfy the position constraint (the common substring should come from same position in S). To verify position constraint, we need to know all forward and reverse indices on each internal node (i.e. the suffix indices of all leaf children below the internal nodes).

In Generalized Suffix Tree of *S#R$*, a substring on the path from root to an internal node is a common substring if the internal node has suffixes from both strings S and R. The index of the common substring in S and R can be found by looking at suffix index at respective leaf node.
If string *S#* is of length N then:

- If suffix index of a leaf is less than N, then that suffix belongs to S and same suffix index will become forward index of all ancestor nodes
- If suffix index of a leaf is greater than N, then that suffix belongs to R and reverse index for all ancestor nodes will be **N – suffix index**

Let's take string S = *cabbaabb*. The figure below is Generalized Suffix Tree for *cabbaabb#bbaabbac$* where we have shown forward and reverse indices of all children suffixes on all internal nodes (except root).
Forward indices are in Parentheses () and reverse indices are in square bracket [].

(Click to see it clearly)



In above figure, all leaf nodes will have one forward or reverse index depending on which string (S or R) they belong to. Then children’s forward or reverse indices propagate to the parent.

Look at the figure to understand what would be the forward or reverse index on a leaf with a given suffix index. At the bottom of figure, it is shown that leaves with suffix indices from 0 to 8 will get same values (0 to 8) as their forward index in S and leaves with suffix indices 9 to 17 will get reverse index in R from 0 to 8.

For example, the highlighted internal node has two children with suffix indices 2 and 9. Leaf with suffix index 2 is from position 2 in S and so it’s forward index is 2 and shown in (). Leaf with suffix index 9 is from position 0 in R and so it’s reverse index is 0 and shown in []. These indices propagate to parent and the parent has one leaf with suffix index 14 for which reverse index is 4. So on this parent node forward index is (2) and reverse index is [0,4]. And in same way, we should be able to understand the how forward and reverse indices are calculated on all nodes.

In above figure, all internal nodes have suffixes from both strings S and R, i.e. all of them represent a common substring on the path from root to themselves. Now we need to find deepest node satisfying position constraint. For this, we need to check if there is a forward index S_i on a node, then there must be a reverse index R_i with value (N – 2) – (S_i + L – 1) where N is length of string *S#* and L is node depth (or substring length). If yes, then consider this node as a LPS candidate, else ignore it. In above figure, deepest node is highlighted which represents LPS as bbaabb.

We have not shown forward and reverse indices on root node in figure. Because root node itself doesn’t represent any common substring (In code implementation also, forward and reverse indices will not be calculated on root node)

How to implement this apprach to find LPS? Here are the things that we need:

- We need to know forward and reverse indices on each node.
- For a given forward index S_i on an internal node, we need know if reverse index **R_i = (N – 2) – (S_i +**

L – 1) also present on same node.

- Keep track of deepest internal node satisfying above condition.

One way to do above is:
While DFS on suffix tree, we can store forward and reverse indices on each node in some way (storage will help to avoid repeated traversals on tree when we need to know forward and reverse indices on a node). Later on, we can do another DFS to look for nodes satisfying position constraint. For position constraint check, we need to search in list of indices.
What data structure is suitable here to do all these in quickest way ?

- If we store indices in array, it will require linear search which will make overall approach non-linear in time.
- If we store indices in tree (set in C++, TreeSet in Java), we may use binary search but still overall approach will be non-linear in time.
- If we store indices in hash function based set (unordered_set in C++, HashSet in Java), it will provide a constant search on average and this will make overall approach linear in time. *A hash function based set may take more space depending on values being stored.*

We will use two unordered_set (one for forward and other from reverse indices) in our implementation, added as a member variable in SuffixTreeNode structure.

```
// A C++ program to implement Ukkonen's Suffix Tree Construction// Here we
build generalized suffix tree for given string S// and it's reverse R, then
we find  // longest palindromic substring of given string S#include
<stdio.h>#include <string.h>#include <stdlib.h>#include <iostream>#include
<unordered_set>#define MAX_CHAR 256
using namespace std;
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;
    //To store indices of children suffixes in given string
    unordered_set<int> *forwardIndices;
    //To store indices of children suffixes in reversed string
    unordered_set<int> *reverseIndices;
};
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
int size1 = 0; //Size of 1st string
int reverseIndex; //Index of a suffix in reversed string
unordered_set<int>::iterator forwardIndex;
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;
    node->forwardIndices = new unordered_set<int>;
    node->reverseIndices = new unordered_set<int>;
    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])
            {
                //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 && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}//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(n != root)
            {
                //Add chldren's suffix indices in parent
                n->forwardIndices->insert(
                    n->children[i]->forwardIndices->begin(),
                    n->children[i]->forwardIndices->end());
                n->reverseIndices->insert(
                    n->children[i]->reverseIndices->begin(),
                    n->children[i]->reverseIndices->end());
            }
        }
    }
    if (leaf == 1)
    {
        for(i= n->start; i<= *(n->end); i++)
        {
            if(text[i] == '#')
            {
                n->end = (int*) malloc(sizeof(int));
                *(n->end) = i;
            }
        }
        n->suffixIndex = size - labelHeight;
        if(n->suffixIndex < size1) //Suffix of Given String
            n->forwardIndices->insert(n->suffixIndex);
        else //Suffix of Reversed String
            n->reverseIndices->insert(n->suffixIndex - size1);
        //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 withsuffixIndex.
suffixIndex for leaf edges will be >= 0 andfor 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;
    int ret = -1;
    if(n->suffixIndex < 0) //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);
                if(*maxHeight < labelHeight
                    && n->forwardIndices->size() > 0 &&
                    n->reverseIndices->size() > 0)
                {
                    for (forwardIndex=n->forwardIndices->begin();
                            forwardIndex!=n->forwardIndices->end();
                            ++forwardIndex)
                    {
                        reverseIndex = (size1 - 2) -
                            (*forwardIndex + labelHeight - 1);
                        //If reverse suffix comes from
                        //SAME position in given string
                        //Keep track of deepest node
                        if(n->reverseIndices->find(reverseIndex) !=
                            n->reverseIndices->end())
                        {
                            *maxHeight = labelHeight;
                            *substringStartIndex = *(n->end) -
                                labelHeight + 1;
                            break;
                        }
                    }
                }
            }
        }
    }
}
void getLongestPalindromicSubstring()
{
    int maxHeight = 0;
    int substringStartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringStartIndex);
    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringStartIndex]);
    if(k == 0)
        printf("No palindromic substring");
    else
        printf(", of length: %d",maxHeight);
    printf("\n");
}// driver program to test above functions
int main(int argc, char *argv[])
```

```
{
    size1 = 9;
    printf("Longest Palindromic Substring in cabbaabb is: ");
    strcpy(text, "cabbaabb#bbaabbac$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    size1 = 17;
    printf("Longest Palindromic Substring in forgeeksskeegfor is: ");
    strcpy(text, "forgeeksskeegfor#rofgeeksskeegrof$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    size1 = 6;
    printf("Longest Palindromic Substring in abcde is: ");
    strcpy(text, "abcde#edcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    size1 = 7;
    printf("Longest Palindromic Substring in abcdae is: ");
    strcpy(text, "abcdae#eadcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    size1 = 6;
    printf("Longest Palindromic Substring in abacd is: ");
    strcpy(text, "abacd#dcaba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    size1 = 6;
    printf("Longest Palindromic Substring in abcdc is: ");
    strcpy(text, "abcdc#cdcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    size1 = 13;
    printf("Longest Palindromic Substring in abacdfgdcaba is: ");
    strcpy(text, "abacdfgdcaba#abacdgfdcaba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    size1 = 15;
    printf("Longest Palindromic Substring in xyabacdfgdcaba is: ");
    strcpy(text, "xyabacdfgdcaba#abacdgfdcabayx$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    size1 = 9;
    printf("Longest Palindromic Substring in xababayz is: ");
    strcpy(text, "xababayz#zyababax$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    size1 = 6;
    printf("Longest Palindromic Substring in xabax is: ");
    strcpy(text, "xabax#xabax$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    return 0;
}
```

Output:

```
Longest Palindromic Substring in cabbaabb is: bbaabb, of length: 6
Longest Palindromic Substring in forgeeksskeegfor is: geeksskeeg, of
length: 10
Longest Palindromic Substring in abcde is: a, of length: 1
Longest Palindromic Substring in abcdae is: a, of length: 1
Longest Palindromic Substring in abacd is: aba, of length: 3
Longest Palindromic Substring in abcdc is: cdc, of length: 3
Longest Palindromic Substring in abacdfgdcaba is: aba, of length: 3
Longest Palindromic Substring in xyabacdfgdcaba is: aba, of length: 3
Longest Palindromic Substring in xababayz is: ababa, of length: 5
Longest Palindromic Substring in xabax is: xabax, of length: 5
```

Followup:

Detect ALL palindromes in a given string.
e.g. For string abcddcbefgf, all possible palindromes are a, b, c, d, e, f, g, dd, fgf, cddc, bcddcb.

We have published following more articles on suffix tree applications:

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