

## STRAIGHTFORWARD VS SUFFIX TREE TRAVERSAL IMPLEMENTATION ANALYSIS & COMPARISON

***Based on Length of String and Length of Pattern.***

Construction of Suffix tree using Ukkonen's algorithm takes  $O(n)$  time where  $n$  is the length of the string and the traversal takes  $O(m)$  time for the length of the pattern.

```

//traverse for each character in the tree
public void traverseAfterBuild(LinkedHashMap patternHashMap,Node rootNode, List<Character> output) {
    //If root node is null, no traversal
    //empty string in tree
    if (rootNode == null) {
        return;
    }
    //if current position is not equal to -1
    if (rootNode.charPos != -1) {
        //StringBuilder builder = new StringBuilder(textToCheck.length);
        //from start to end
        for (int i = rootNode.start; i <= rootNode.finalPos.finalPos; i++) {
            output.add(textToCheck[i]);
        }
        // suffixHashMap.put(rootNode.charPos,builder.toString());

        for (int i = rootNode.start; i <= rootNode.finalPos.finalPos; i++) {
            output.remove( index: output.size() - 1);
        }
        return;
    }

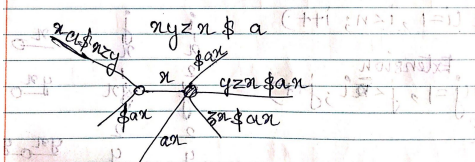
    for (int i = rootNode.start; i <= rootNode.finalPos.finalPos; i++) {
        output.add(textToCheck[i]);
    }
    for (Node node : rootNode.child) {
        traverseAfterBuild(patternHashMap,node, output);
    }
    for (int i = rootNode.start; i <= rootNode.finalPos.finalPos; i++) {
        output.remove( index: output.size() - 1);
    }
}
}

```

Rule 1: Walk till end, add new character

Rule 2: No path, create a new path

Rule 3: Path already exists, do nothing



Traversal of first test case in suffix tree-Step by step shown in below screenshots.

The first screenshot shows the initial state of the suffix tree traversal. The console output lists the following suffixes (lines 0-19):

```
0 tccpsivarsnfnvcrilpgtpeaicaltytgciilpgatcpgdyan
1 cpsivarsnfnvcrilpgtpeaicaltytgciilpgatcpgdyan
2 cpsivarsnfnvcrilpgtpeaicaltytgciilpgatcpgdyan
3 sivarsnfnvcrilpgtpeaicaltytgciilpgatcpgdyan
4 sivarsnfnvcrilpgtpeaicaltytgciilpgatcpgdyan
5 ivarsnfnvcrilpgtpeaicaltytgciilpgatcpgdyan
6 varsnfnvcrilpgtpeaicaltytgciilpgatcpgdyan
7 arsnfnvcrilpgtpeaicaltytgciilpgatcpgdyan
8 rsnfnvcrilpgtpeaicaltytgciilpgatcpgdyan
9 snfnvcrilpgtpeaicaltytgciilpgatcpgdyan
10 nfnvcrilpgtpeaicaltytgciilpgatcpgdyan
11 fnvcrilpgtpeaicaltytgciilpgatcpgdyan
12 fnvcrilpgtpeaicaltytgciilpgatcpgdyan
13 vcrilpgtpeaicaltytgciilpgatcpgdyan
14 crilpgtpeaicaltytgciilpgatcpgdyan
15 rilpgtpeaicaltytgciilpgatcpgdyan
16 lpgtpeaicaltytgciilpgatcpgdyan
17 lpgtpeaicaltytgciilpgatcpgdyan
18 tpeaicaltytgciilpgatcpgdyan
19 tpeaicaltytgciilpgatcpgdyan
```

The second screenshot shows the next step in the traversal. The console output lists the following suffixes (lines 20-39):

```
20 peaicaltytgciilpgatcpgdyan
21 eaicaltytgciilpgatcpgdyan
22 eaicaltytgciilpgatcpgdyan
23 icaltytgciilpgatcpgdyan
24 catytgciilpgatcpgdyan
25 atytgciilpgatcpgdyan
26 tgciiipgatcpgdyan
27 tgciiipgatcpgdyan
28 tgciiipgatcpgdyan
29 gciiipgatcpgdyan
30 cliipgatcpgdyan
31 iipgatcpgdyan
32 ipgatcpgdyan
33 pgatcpgdyan
34 pgatcpgdyan
35 atcpgdyan
36 atcpgdyan
37 cpgdyan
38 pgdyan
39 gdyan
```

***Straightforward implementation***

```

//Check if the given pattern matches
public boolean checkPattern(BufferedReader writer, FileUtils fileUtilsObj, char[] patternArr, char[] textArr) {
    for (int i = 0; i < textArr.length - patternArr.length + 1; i++) {
        for (int j = 0; j < patternArr.length; j++) {
            if (textArr[i + j] == patternArr[j]) {
                if (j == patternArr.length) {
                    return true;
                }
            }
        }
    }
    return false;
}

//Get the position of the substring
public int getPos(String text, String pattern) {
    int j=0;
    if (pattern.length() >= 1) {
        for (int i = 0; i < text.length(); i++) {
            if (text.charAt(i) == pattern.toString().charAt(j)) {
                j++;
                if (j == pattern.length()) {
                    return i - pattern.length() + 1;
                }
            } else {
                j = 0;
            }
        }
    }
    return -1;
}
}

```

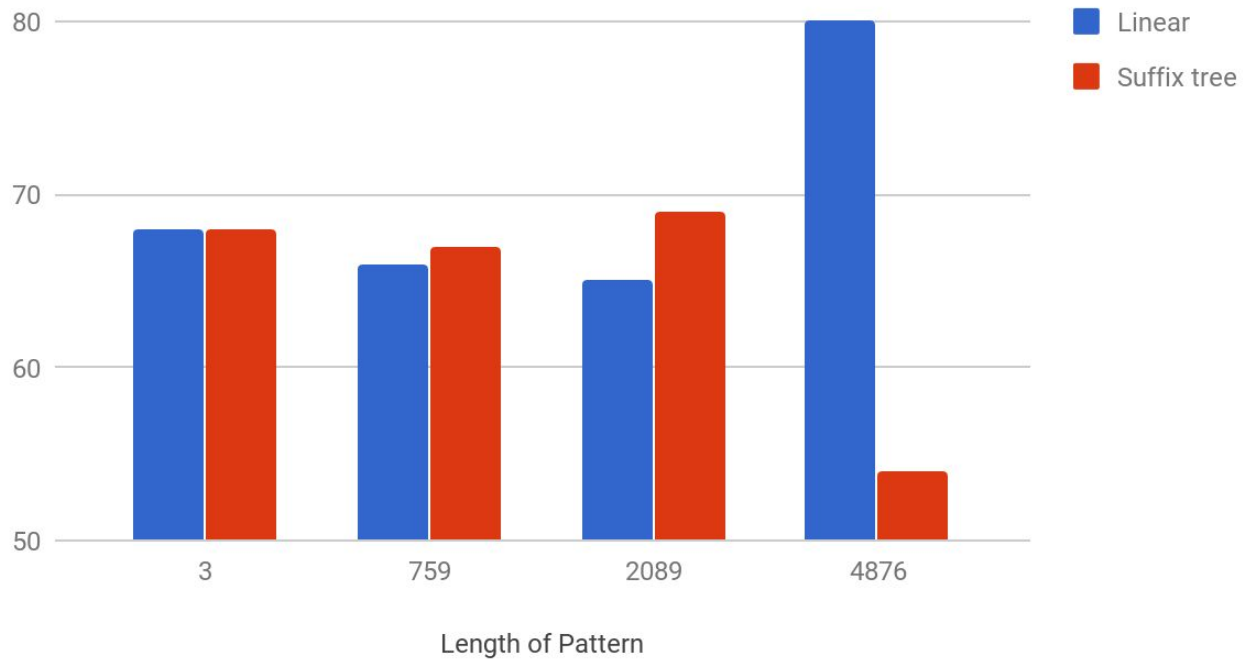
While the straightforward approach, takes  $O(n*m)$  time , $n$  being the length of the string and  $m$  being the length of the pattern, i.e., time taken is  $m*n$  time which isn't desirable as it takes too much time and space.

### ***Comparison of Straight forward approach Vs Suffix tree approach without including the time for tree construction***

Based on 5 testcases(2 given and 3 self experimentation)

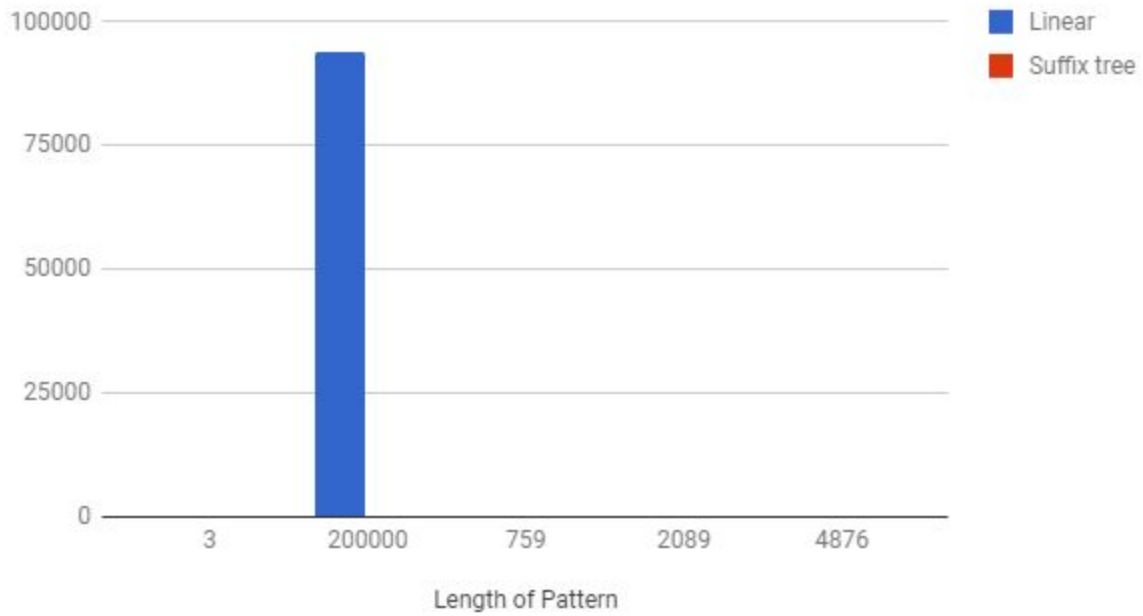
***Number of characters in given string and pattern and the time in ms.***

## Straightforward Vs Suffix tree

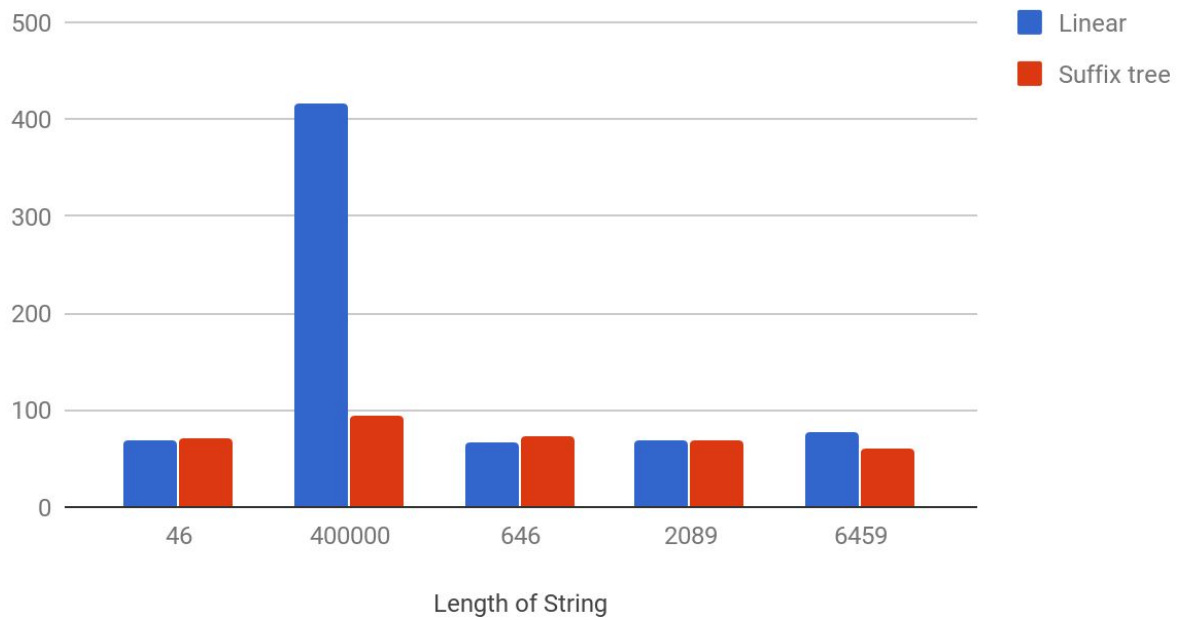


***WHEN THE LENGTH OF PATTERN IS 2,00,000 AND LENGTH OF STRING IS 4,00,000  
IT TAKES 94023ms,find the below graph for the same***

Straightforward Vs Suffix tree



Straightforward Vs Suffix tree



## References:

1. <http://www.geeksforgeeks.org/pattern-searching-set-8-suffix-tree-introduction/>
2. <http://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/>
3. <http://www.geeksforgeeks.org/suffix-tree-application-1-substring-check/>
4. <https://www.youtube.com/watch?v=aPRqocoBsFQ>
5. <http://www.geeksforgeeks.org/suffix-tree-application-1-substring-check/>
6. <https://web.stanford.edu/~mjkay/gusfield.pdf>