

# STRING ALGORITHMS

## CHAPTER

# 15



### 15.1 Introduction

To understand the importance of string algorithms let us consider the case of entering the URL (Uniform Resource Locator) in any browser (say, Internet Explorer, Firefox, or Google Chrome). You will observe that after typing the prefix of the URL, a list of all possible URLs is displayed. That means, the browsers are doing some internal processing and giving us the list of matching URLs. This technique is sometimes called *auto-completion*.

Similarly, consider the case of entering the directory name in the command line interface (in both *Windows* and *UNIX*). After typing the prefix of the directory name, if we press the *tab* button, we get a list of all matched directory names available. This is another example of auto completion.

In order to support these kinds of operations, we need a data structure which stores the string data efficiently. In this chapter, we will look at the data structures that are useful for implementing string algorithms.

We start our discussion with the basic problem of strings: given a string, how do we search a substring (pattern)? This is called a *string matching* problem. After discussing various string matching algorithms, we will look at different data structures for storing strings.

### 15.2 String Matching Algorithms

In this section, we concentrate on checking whether a pattern  $P$  is a substring of another string  $T$  ( $T$  stands for text) or not. Since we are trying to check a fixed string  $P$ , sometimes these algorithms are called *exact string matching* algorithms. To simplify our discussion, let us assume that the length of given text  $T$  is  $n$  and the length of the pattern  $P$  which we are trying to match has the length  $m$ . That means,  $T$  has the characters from 0 to  $n-1$  ( $T[0 \dots n-1]$ ) and  $P$  has the characters from 0 to  $m-1$  ( $P[0 \dots m-1]$ ). This algorithm is implemented in C++ as `strstr()`.

In the subsequent sections, we start with the brute force method and gradually move towards better algorithms.

- Brute Force Method
- Robin-Karp String Matching Algorithm
- String Matching with Finite Automata
- KMP Algorithm
- Boyce-Moore Algorithm
- Suffix Trees

### 15.3 Brute Force Method

In this method, for each possible position in the text  $T$  we check whether the pattern  $P$  matches or not. Since the length of  $T$  is  $n$ , we have  $n - m + 1$  possible choices for comparisons. This is because we do not need to check the last  $m - 1$  locations of  $T$  as the pattern length is  $m$ . The following algorithm searches for the first occurrence of a pattern string  $P$  in a text string  $T$ .

**Algorithm**

```
def strstrBruteForce(str, pattern):
    if not pattern: return 0
    for i in range(len(str)-len(pattern)+1):
        stri = i; patterni = 0
        while stri < len(str) and patterni < len(pattern) and str[stri] == pattern[patterni]:
            stri += 1
            patterni += 1
        if patterni == len(pattern): return i
    return -1

print strstrBruteForce("xxxxyzabcdabedfabcd", "abc")
```

Time Complexity:  $O((n - m + 1) \times m) \approx O(n \times m)$ . Space Complexity:  $O(1)$ .

## 15.4 Robin-Karp String Matching Algorithm

In this method, we will use the hashing technique and instead of checking for each possible position in  $T$ , we check only if the hashing of  $P$  and the hashing of  $m$  characters of  $T$  give the same result.

Initially, apply the hash function to the first  $m$  characters of  $T$  and check whether this result and  $P$ 's hashing result is the same or not. If they are not the same, then go to the next character of  $T$  and again apply the hash function to  $m$  characters (by starting at the second character). If they are the same then we compare those  $m$  characters of  $T$  with  $P$ .

### Selecting Hash Function

At each step, since we are finding the hash of  $m$  characters of  $T$ , we need an efficient hash function. If the hash function takes  $O(m)$  complexity in every step, then the total complexity is  $O(n \times m)$ . This is worse than the brute force method because first we are applying the hash function and also comparing.

Our objective is to select a hash function which takes  $O(1)$  complexity for finding the hash of  $m$  characters of  $T$  every time. Only then can we reduce the total complexity of the algorithm. If the hash function is not good (worst case), the complexity of the Robin-Karp algorithm is  $O((n - m + 1) \times m) \approx O(n \times m)$ . If we select a good hash function, the complexity of the Robin-Karp algorithm complexity is  $O(m + n)$ . Now let us see how to select a hash function which can compute the hash of  $m$  characters of  $T$  at each step in  $O(1)$ .

For simplicity, let's assume that the characters used in string  $T$  are only integers. That means, all characters in  $T \in \{0, 1, 2, \dots, 9\}$ . Since all of them are integers, we can view a string of  $m$  consecutive characters as decimal numbers. For example, string '61815' corresponds to the number 61815. With the above assumption, the pattern  $P$  is also a decimal value, and let us assume that the decimal value of  $P$  is  $p$ . For the given text  $T[0..n-1]$ , let  $t(i)$  denote the decimal value of length- $m$  substring  $T[i..i+m-1]$  for  $i = 0, 1, \dots, n-m-1$ . So,  $t(i) == p$  if and only if  $T[i..i+m-1] == P[0..m-1]$ .

We can compute  $p$  in  $O(m)$  time using Horner's Rule as:

$$p = P[m-1] + 10(P[m-2] + 10(P[m-3] + \dots + 10(P[1] + 10P[0])\dots))$$

The code for the above assumption is:

```
value = 0
for i in range(0, m-1):
    value = value * 10
    value = value + P[i]
```

We can compute all  $t(i)$ , for  $i = 0, 1, \dots, n-m-1$  values in a total of  $O(n)$  time. The value of  $t(0)$  can be similarly computed from  $T[0..m-1]$  in  $O(m)$  time. To compute the remaining values  $t(0), t(1), \dots, t(n-m-1)$ , understand that  $t(i+1)$  can be computed from  $t(i)$  in constant time.

$$t(i+1) = 10 * (t(i) - 10^{m-1} * T[i]) + T[i+m-1]$$

For example, if  $T = "123456"$  and  $m = 3$

$$\begin{aligned} t(0) &= 123 \\ t(1) &= 10 * (123 - 100 * 1) + 4 = 234 \end{aligned}$$

**Step by Step explanation**

First : remove the first digit :  $123 - 100 * 1 = 23$

Second: Multiply by 10 to shift it :  $23 * 10 = 230$

Third: Add last digit :  $230 + 4 = 234$

The algorithm runs by comparing,  $t(i)$  with  $p$ . When  $t(i) == p$ , then we have found the substring  $P$  in  $T$ , starting from position  $i$ .

```
def RobinKarp(text, pattern):
    if pattern == None or text == None:
        return -1
    if pattern == "" or text == "":
        return -1
    if len(pattern) > len(text):
        return -1

    hashText = Hash(text, len(pattern))
    hashPattern = Hash(pattern, len(pattern))
    hashPattern.update()
    for i in range(len(text)-len(pattern)+1):
        if hashText.hashValue() == hashPattern.hashValue():
            if hashText.text() == pattern:
                return i
        hashText.update()
    return -1

class Hash:
    def __init__(self, text, size):
        self.str = text
        self.hash = 0
        for i in xrange(0, size):
            self.hash += ord(self.str[i])

        self.init = 0
        self.end = size

    def update(self):
        if self.end <= len(self.str) - 1:
            self.hash -= ord(self.str[self.init])
            self.hash += ord(self.str[self.end])
            self.init += 1
            self.end += 1

    def hashValue(self):
        return self.hash

    def text(self):
        return self.str[self.init:self.end]

print RobinKarp("3141592653589793", "26")
```

## 15.5 String Matching with Finite Automata

In this method we use the finite automata which is the concept of the Theory of Computation (ToC). Before looking at the algorithm, first let us look at the definition of finite automata.

### Finite Automata

A finite automaton  $F$  is a 5-tuple  $(Q, q_0, A, \Sigma, \delta)$ , where

- $Q$  is a finite set of states
- $q_0 \in Q$  is the start state
- $A \subseteq Q$  is a set of accepting states
- $\Sigma$  is a finite input alphabet
- $\delta$  is the transition function that gives the next state for a given current state and input

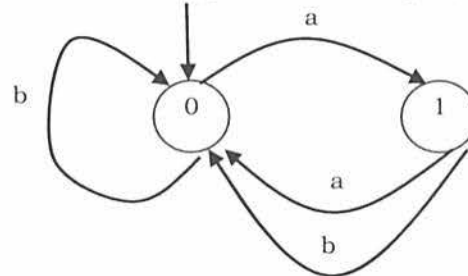
## How does Finite Automata Work?

- The finite automaton  $F$  begins in state  $q_0$
- Reads characters from  $\Sigma$  one at a time
- If  $F$  is in state  $q$  and reads input character  $a$ ,  $F$  moves to state  $\delta(q, a)$
- At the end, if its state is in  $A$ , then we say,  $F$  accepted the input string read so far
- If the input string is not accepted it is called the rejected string

**Example:** Let us assume that  $Q = \{0,1\}$ ,  $q_0 = 0$ ,  $A = \{1\}$ ,  $\Sigma = \{a,b\}$ .  $\delta(q, a)$  as shown in the transition table/diagram. This accepts strings that end in an odd number of  $a$ 's; e.g.,  $abbaaa$  is accepted,  $aa$  is rejected.

Input		
State	a	b
0	1	0
1	0	0

Transition Function/Table



## Important Notes for Constructing the Finite Automata

For building the automata, first we start with the initial state. The FA will be in state  $k$  if  $k$  characters of the pattern have been matched. If the next text character is equal to the pattern character  $c$ , we have matched  $k + 1$  characters and the FA enters state  $k + 1$ . If the next text character is not equal to the pattern character, then the FA goes to a state  $0, 1, 2, \dots, \text{or } k$ , depending on how many initial pattern characters match the text characters ending with  $c$ .

## Matching Algorithm

Now, let us concentrate on the matching algorithm.

- For a given pattern  $P[0..m-1]$ , first we need to build a finite automaton  $F$ 
  - The state set is  $Q = \{0, 1, 2, \dots, m\}$
  - The start state is 0
  - The only accepting state is  $m$
  - Time to build  $F$  can be large if  $\Sigma$  is large
- Scan the text string  $T[0..n-1]$  to find all occurrences of the pattern  $P[0..m-1]$
- String matching is efficient:  $\Theta(n)$ 
  - Each character is examined exactly once
  - Constant time for each character
  - But the time to compute  $\delta$  (transition function) is  $O(m|\Sigma|)$ . This is because  $\delta$  has  $O(m|\Sigma|)$  entries. If we assume  $|\Sigma|$  is constant then the complexity becomes  $O(m)$ .

### Algorithm:

```

# Input: Pattern string P[0..m-1],  $\delta$  and  $F$ 
# Goal: All valid shifts displayed
def FiniteAutomataStringMatcher(P, m, F,  $\delta$ ):
    q = 0
    for i in range(0, m):
        q =  $\delta(q, T[i])$ 
        if (q == m):
            print("Pattern occurs with shift: ", i-m)
  
```

Time Complexity:  $O(m)$ .

## 15.6 KMP Algorithm

As before, let us assume that  $T$  is the string to be searched and  $P$  is the pattern to be matched. This algorithm was presented by Knuth, Morris and Pratt. It takes  $O(n)$  time complexity for searching a pattern. To get  $O(n)$

time complexity, it avoids the comparisons with elements of  $T$  that were previously involved in comparison with some element of the pattern  $P$ .

The algorithm uses a table and in general we call it *prefix function* or *prefix table* or *fail function*  $F$ . First we will see how to fill this table and later how to search for a pattern using this table. The prefix function  $F$  for a pattern stores the knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern  $P$ . It means that this table can be used for avoiding backtracking on the string  $TT$ .

## Prefix Table

```
def prefixTable(pattern):
    m = len(pattern)
    F = [0] * m
    k = 0
    for q in range(1, m):
        while k > 0 and pattern[k] != pattern[q]:
            k = F[k - 1]
        if pattern[k] == pattern[q]:
            k = k + 1
        F[q] = k
    return F
```

As an example, assume that  $P = ababaca$ . For this pattern, let us follow the step-by-step instructions for filling the prefix table  $F$ . Initially:  $m = \text{length}[P] = 7$ ,  $F[0] = 0$  and  $F[1] = 0$ .

**Step 1:**  $i = 1, j = 0, F[1] = 0$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0					

**Step 2:**  $i = 2, j = 0, F[2] = 1$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1				

**Step 3:**  $i = 3, j = 1, F[3] = 2$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2			

**Step 4:**  $i = 4, j = 2, F[4] = 3$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3		

**Step 5:**  $i = 5, j = 3, F[5] = 1$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3	0	

**Step 6:**  $i = 6, j = 1, F[6] = 1$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3	0	1

At this step the filling of the prefix table is complete.

## Matching Algorithm

The KMP algorithm takes pattern  $P$ , string  $T$  and prefix function  $F$  as input, and finds a match of  $P$  in  $T$ .

```
def KMP(text, pattern):
    n = len(text)
    m = len(pattern)
    F = prefixTable(pattern)
    q = 0
```



```

for i in range(n):
    while q > 0 and pattern[q] != text[i]:
        q = F[q - 1]
    if pattern[q] == text[i]:
        q = q + 1
    if q == m:
        return i - m + 1
return -1

print KMP("bacbabababacaca", "ababaca")

```

Time Complexity:  $O(m + n)$ , where  $m$  is the length of the pattern and  $n$  is the length of the text to be searched.  
 Space Complexity:  $O(m)$ .

Now, to understand the process let us go through an example. Assume that  $T = bacbabababacaca$  &  $P = ababaca$ . Since we have already filled the prefix table, let us use it and go to the matching algorithm. Initially:  $n = \text{size of } T = 15$ ;  $m = \text{size of } P = 7$ .

**Step 1:**  $i = 0, j = 0$ , comparing  $P[0]$  with  $T[0]$ .  $P[0]$  does not match with  $T[0]$ .  $P$  will be shifted one position to the right.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P	a	b	a	b	a	c	a								

**Step 2:**  $i = 1, j = 0$ , comparing  $P[0]$  with  $T[1]$ .  $P[0]$  matches with  $T[1]$ . Since there is a match,  $P$  is not shifted.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P		a	b	a	b	a	c	a							

**Step 3:**  $i = 2, j = 1$ , comparing  $P[1]$  with  $T[2]$ .  $P[1]$  does not match with  $T[2]$ . Backtracking on  $P$ , comparing  $P[0]$  and  $T[2]$ .

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P		a	b	a	b	a	c	a							

**Step 4:**  $i = 3, j = 0$ , comparing  $P[0]$  with  $T[3]$ .  $P[0]$  does not match with  $T[3]$ .

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P				a	b	a	b	a	c	a					

**Step 5:**  $i = 4, j = 0$ , comparing  $P[0]$  with  $T[4]$ .  $P[0]$  matches with  $T[4]$ .

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

**Step 6:**  $i = 5, j = 1$ , comparing  $P[1]$  with  $T[5]$ .  $P[1]$  matches with  $T[5]$ .

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				


**Step 7:**  $i = 6, j = 2$ , comparing  $P[2]$  with  $T[6]$ .  $P[2]$  matches with  $T[6]$ .

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

**Step 8:**  $i = 7, j = 3$ , comparing  $P[3]$  with  $T[7]$ .  $P[3]$  matches with  $T[7]$ .


T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

**Step 9:**  $i = 8, j = 4$ , comparing  $P[4]$  with  $T[8]$ .  $P[4]$  matches with  $T[8]$ .




T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

**Step 10:**  $i = 9, j = 5$ , comparing  $P[5]$  with  $T[9]$ .  $P[5]$  does not match with  $T[9]$ . Backtracking on  $P$ , comparing  $P[4]$  with  $T[9]$  because after mismatch  $j = F[4] = 3$ .




T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

Comparing  $P[3]$  with  $T[9]$ .




T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P							a	b	a	b	a	c	a		

**Step 11:**  $i = 10, j = 4$ , comparing  $P[4]$  with  $T[10]$ .  $P[4]$  matches with  $T[10]$ .




T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P							a	b	a	b	a	c	a		

**Step 12:**  $i = 11, j = 5$ , comparing  $P[5]$  with  $T[11]$ .  $P[5]$  matches with  $T[11]$ .



T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P							a	b	a	b	a	c	a		

**Step 13:**  $i = 12, j = 6$ , comparing  $P[6]$  with  $T[12]$ .  $P[6]$  matches with  $T[12]$ .



T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P							a	b	a	b	a	c	a		

Pattern  $P$  has been found to completely occur in string  $T$ . The total number of shifts that took place for the match to be found are:  $i - m = 13 - 7 = 6$  shifts.

#### Notes:

- KMP performs the comparisons from left to right
- KMP algorithm needs a preprocessing (prefix function) which takes  $O(m)$  space and time complexity
- Searching takes  $O(n + m)$  time complexity (does not depend on alphabet size)

## 15.7 Boyce-Moore Algorithm

Like the KMP algorithm, this also does some pre-processing and we call it *last function*. The algorithm scans the characters of the pattern from right to left beginning with the rightmost character. During the testing of a possible placement of pattern  $P$  in  $T$ , a mismatch is handled as follows: Let us assume that the current character being matched is  $T[i] = c$  and the corresponding pattern character is  $P[j]$ . If  $c$  is not contained anywhere in  $P$ , then shift the pattern  $P$  completely past  $T[i]$ . Otherwise, shift  $P$  until an occurrence of character  $c$  in  $P$  gets aligned with  $T[i]$ . This technique avoids needless comparisons by shifting the pattern relative to the text.

The *last function* takes  $O(m + |\Sigma|)$  time and the actual search takes  $O(nm)$  time. Therefore the worst case running time of the Boyer-Moore algorithm is  $O(nm + |\Sigma|)$ . This indicates that the worst-case running time is quadratic, in the case of  $n == m$ , the same as the brute force algorithm.

- The Boyer-Moore algorithm is very fast on the large alphabet (relative to the length of the pattern).
- For the small alphabet, Boyce-Moore is not preferable.
- For binary strings, the KMP algorithm is recommended.
- For the very shortest patterns, the brute force algorithm is better.

## 15.8 Data Structures for Storing Strings

If we have a set of strings (for example, all the words in the dictionary) and a word which we want to search in that set, in order to perform the search operation faster, we need an efficient way of storing the strings. To store sets of strings we can use any of the following data structures.

- Hashing Tables
- Binary Search Trees
- Tries
- Ternary Search Trees

## 15.9 Hash Tables for Strings

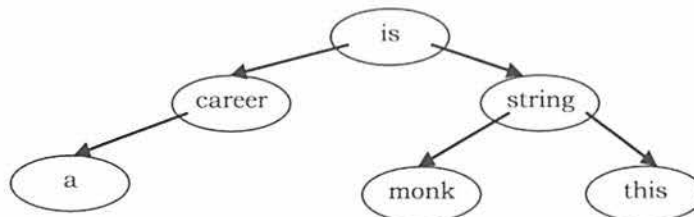
As seen in the *Hashing* chapter, we can use hash tables for storing the integers or strings. In this case, the keys are nothing but the strings. The problem with hash table implementation is that we lose the ordering information – after applying the hash function, we do not know where it will map to. As a result, some queries take more time. For example, to find all the words starting with the letter “K”, with hash table representation we need to scan the complete hash table. This is because the hash function takes the complete key, performs hash on it, and we do not know the location of each word.

## 15.10 Binary Search Trees for Strings

In this representation, every node is used for sorting the strings alphabetically. This is possible because the strings have a natural ordering: *A* comes before *B*, which comes before *C*, and so on. This is because words can be ordered and we can use a Binary Search Tree (BST) to store and retrieve them. For example, let us assume that we want to store the following strings using BSTs:

*this is a career monk string*

For the given string there are many ways of representing them in BST. One such possibility is shown in the tree below.



### Issues with Binary Search Tree Representation

This method is good in terms of storage efficiency. But the disadvantage of this representation is that, at every node, the search operation performs the complete match of the given key with the node data, and as a result the time complexity of the search operation increases. So, from this we can say that BST representation of strings is good in terms of storage but not in terms of time.

## 15.11 Tries

Now, let us see the alternative representation that reduces the time complexity of the search operation. The name *trie* is taken from the word *re*“trie”.

### What is a Trie?

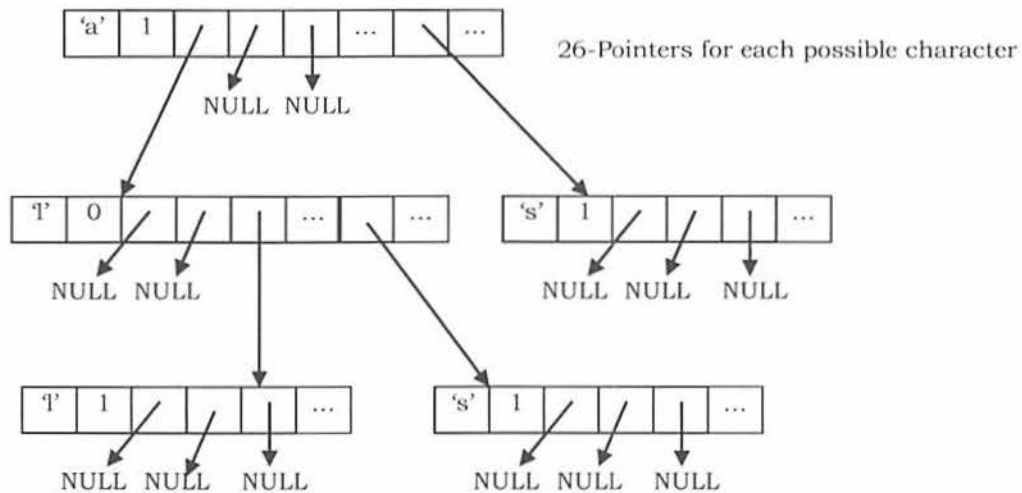
A *trie* is a tree and each node in it contains the number of pointers equal to the number of characters of the alphabet. For example, if we assume that all the strings are formed with English alphabet characters “a” to “z” then each node of the trie contains 26 pointers. A trie data structure can be declared as:

```

class Node(object):
    def __init__(self):
        self.children={}#contains a map with child characters as keys and their Node as values
  
```



Suppose we want to store the strings "a", "all", "als", and "as": *trie* for these strings will look like:



## Why Tries?

The tries can insert and find strings in  $O(L)$  time (where  $L$  represents the length of a single word). This is much faster than hash table and binary search tree representations.

## Trie Declaration

The structure of the `TrieNode` has data (char), is\_End\_Of\_String (boolean), and has a collection of child nodes (Collection of `TrieNodes`). It also has one more method called `subNode(char)`. This method takes a character as argument and will return the child node of that character type if that is present. The basic element - `TrieNode` of a TRIE data structure looks like this:

```
class Node(object):
    def __init__(self):
        self.children={}#contains a map with child characters as keys and their Node as values
class Trie(object):
    def __init__(self):
        self.root = Node()
        self.root.data = "/"
```

Now that we have defined our `TrieNode`, let's go ahead and look at the other operations of TRIE. Fortunately, the TRIE data structure is simple to implement since it has two major methods: `insert()` and `search()`. Let's look at the elementary implementation of both these methods.

## Inserting a String in Trie

To insert a string, we just need to start at the root node and follow the corresponding path (path from root indicates the prefix of the given string). Once we reach the NULL pointer, we just need to create a skew of tail nodes for the remaining characters of the given string.

```
def addWord(self,word):
    currentNode = self.root
    i = 0
    #print "adding word "+ word+" to trie "
    for c in word:
        #print "adding character " + c
        try:
            currentNode = currentNode.children[c]
            #print "character "+c+" exists"
        except:
            self.createSubTree(word[i:len(word)],currentNode)
            break
    i = i + 1
```

Time Complexity:  $O(L)$ , where  $L$  is the length of the string to be inserted.

**Note:** For real dictionary implementation, we may need a few more checks such as checking whether the given string is already there in the dictionary or not.

## Searching a String in Trie

The same is the case with the search operation: we just need to start at the root and follow the pointers. The time complexity of the search operation is equal to the length of the given string that want to search.

```
def getWordList(self, startingCharacters):
    startNode = self.root
    for c in startingCharacters:
        try:
            startNode = startNode.children[c]
        except:
            return []
    nodestack=[]
    for child in startNode.children:
        nodestack.append(startNode.children[child])
    words=[]
    currentWord=""
    while len(nodestack) != 0:
        currentNode = nodestack.pop()
        currentWord += currentNode.data
        if len (currentNode.children) == 0:
            words.append(startingCharacters+currentWord)
            currentWord = ""
        for n in currentNode.children:
            temp = currentNode.children[n]
            nodestack.append(temp)
    return words
```

Time Complexity:  $O(L)$ , where  $L$  is the length of the string to be searched.

## Issues with Tries Representation

The main disadvantage of tries is that they need lot of memory for storing the strings. As we have seen above, for each node we have too many node pointers. In many cases, the occupancy of each node is less. The final conclusion regarding tries data structure is that they are faster but require huge memory for storing the strings.

**Note:** There are some improved tries representations called *trie compression techniques*. But, even with those techniques we can reduce the memory only at the leaves and not at the internal nodes.

## 15.12 Ternary Search Trees

This representation was initially provided by Jon Bentley and Sedgewick. A ternary search tree takes the advantages of binary search trees and tries. That means it combines the memory efficiency of BSTs and the time efficiency of tries.

### Ternary Search Trees Declaration

```
class TSTNode:
    def __init__(self, x):
        self.data = x
        self.left = None
        self.eq = None
        self.right = None
```

The Ternary Search Tree (TST) uses three pointers:

- The *left* pointer points to the TST containing all the strings which are alphabetically less than *data*.
- The *right* pointer points to the TST containing all the strings which are alphabetically greater than *data*.
- The *eq* pointer points to the TST containing all the strings which are alphabetically equal to *data*. That means, if we want to search for a string, and if the current character of the input string and the *data* of current node in TST are the same, then we need to proceed to the next character in the input string and search it in the subtree which is pointed by *eq*.

### Operation Method of TST

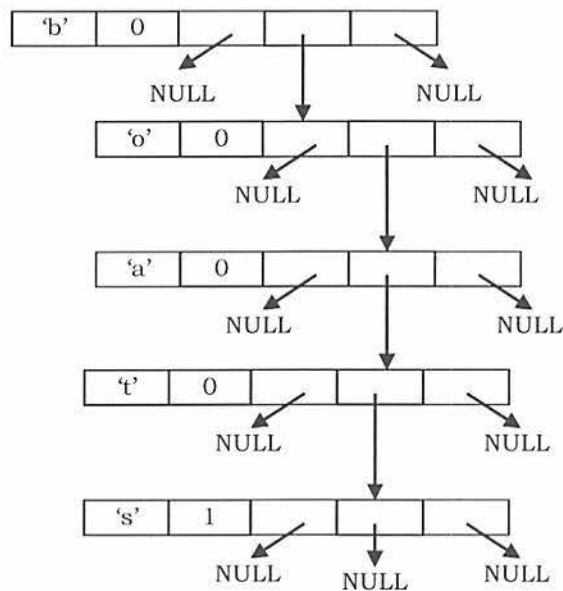
Let's make the operation method of class TST.

```
class TST:
    def __init__(self, x = None):
        self.root = Node (None) # header
        self.leaf = x
```

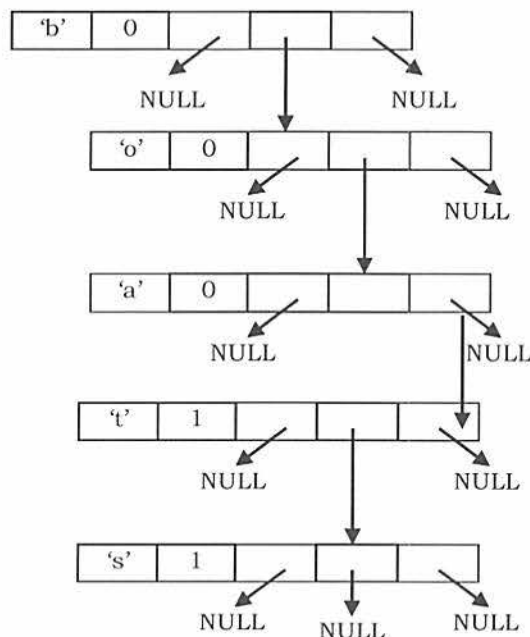
The instance variable `root` of TST will store the header. Data in this section is a dummy. The actual data will continue to add to the root of the child. Instance variable `leaf` stores the data representing the termination. `leaf` is passed as an argument when calling the TST. It will be `None` if it is omitted.

### Inserting strings in Ternary Search Tree

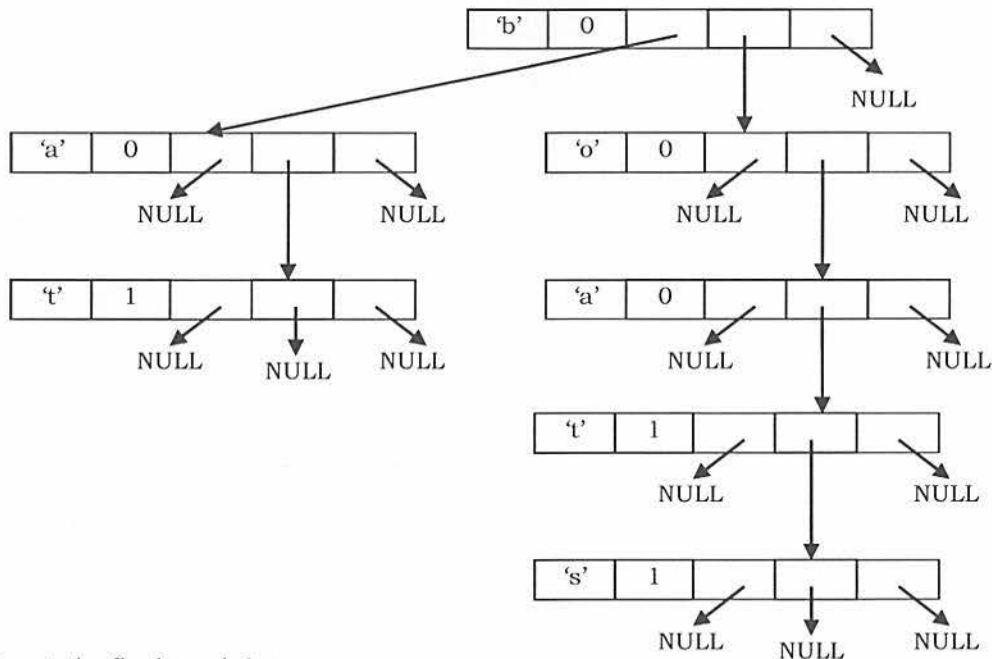
For simplicity let us assume that we want to store the following words in TST (also assume the same order): *boats*, *boat*, *bat* and *bats*. Initially, let us start with the *boats* string.



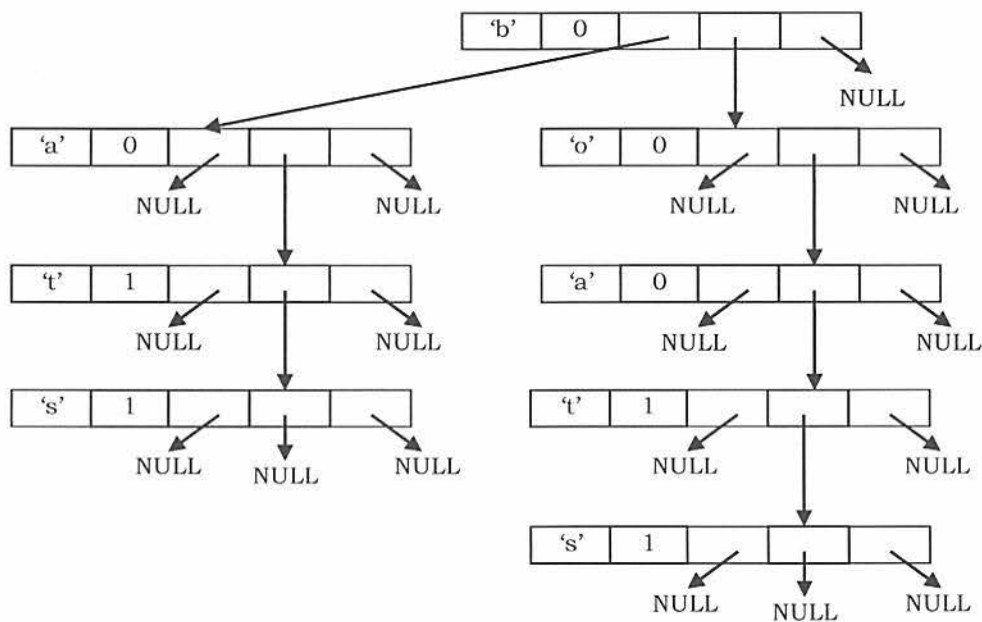
Now if we want to insert the string *boat*, then the TST becomes [the only change is setting the *is\_End\_Of\_String* flag of "t" node to 1]:



Now, let us insert the next string: *bat*



Now, let us insert the final word: *bats*.



Based on these examples, we can write the insertion algorithm as below. We will combine the insertion operation of BST and tries.

```

# Insert
def _insert (node, x):
    if node is None: return x
    elif x.data == node.data: return node
    elif x.data < node.data:
        node.left = _insert (node.left, x)
    else:
        node.right = _insert (node.right, x)
    return node

class TST:
    def __init__ (self, x = None):

```

```

self.root = TSTNode (None) # header
self.leaf = x

# Insert
def insert (self, seq):
    node = self.root
    for x in seq:
        child = _search (node.eq, x)
        if not child:
            child = TSTNode (x)
            node.eq = _insert (node.eq, child)
        node = child
    # Check leaf
    if not _search (node.eq, self.leaf):
        node.eq = _insert (node.eq, TSTNode (self.leaf))

```

Time Complexity:  $O(L)$ , where  $L$  is the length of the string to be inserted.

## Searching in Ternary Search Tree

If after inserting the words we want to search for them, then we have to follow the same rules as that of binary search. The only difference is, in case of match we should check for the remaining characters (in *eq* subtree) instead of return. Also, like BSTs we will see both recursive and non-recursive versions of the search method.

```

# Search
def _search (node, x):
    while node:
        if node.data == x: return node
        if x < node.data:
            node = node.left
        else:
            node = node.right
    return None

class TST:
    # Search
    def _search (node, x):
        while node:
            if node.data == x: return node
            if x < node.data:
                node = node.left
            else:
                node = node.right
        return None

```

Time Complexity:  $O(L)$ , where  $L$  is the length of the string to be searched.

## Displaying All Words of Ternary Search Tree

If we want to print all the strings of TST we can use the following algorithm. If we want to print them in sorted order, we need to follow the inorder traversal of TST.

```

# Traverse
def _traverse (node, leaf):
    if node:
        for x in _traverse (node.left, leaf):
            yield x
        if node.data == leaf:
            yield []
        else:
            for x in _traverse (node.eq, leaf):
                yield [node.data] + x
            for x in _traverse (node.right, leaf):
                yield x

class TST:
    def __init__ (self, x = None):
        self.root = TSTNode (None) # header

```



```

        self.leaf = x
    # Traverse
    def traverse (self):
        for x in _traverse (self.root.eq, self.leaf):
            yield x

```

### Full Implementation

```

class TSTNode:
    def __init__ (self, x):
        self.data = x
        self.left = None
        self.eq = None
        self.right = None
def _search (node, x):
    while node:
        if node.data == x: return node
        if x < node.data:
            node = node.left
        else:
            node = node.right
    return None
def _insert (node, x):
    if node is None: return x
    elif x.data == node.data: return node
    elif x.data < node.data:
        node.left = _insert (node.left, x)
    else:
        node.right = _insert (node.right, x)
    return node
# Find the minimum value
def _searchMin (node):
    if node.left is None: return node.data
    return _searchMin (node.left)
# Delete the minimum value
def _deleteMin (node):
    if node.left is None: return node.right
    node.left = _deleteMin (node.left)
    return node
def _delete (node, x):
    if node:
        if x == node.data:
            if node.left is None:
                return node.right
            elif node.right is None:
                return node.left
            else:
                node.data = _searchMin (node.right)
                node.right = _deleteMin (node.right)
        elif x < node.data:
            node.left = _delete (node.left, x)
        else:
            node.right = _delete (node.right, x)
    return node
def _traverse (node, leaf):
    if node:
        for x in _traverse (node.left, leaf):
            yield x
        if node.data == leaf:
            yield []
        else:

```

```

        for x in _traverse (node.eq, leaf):
            yield [node.data] + x
        for x in _traverse (node.right, leaf):
            yield x
##### Ternary Search Tree #####
class TST:
    def __init__ (self, x = None):
        self.root = TSTNode (None) # header
        self.leaf = x
    def search (self, seq):
        node = self.root
        for x in seq:
            node = _search (node.eq, x)
            if not node: return False
        # Check leaf
        return _search (node.eq, self.leaf) is not None
    def insert (self, seq):
        node = self.root
        for x in seq:
            child = _search (node.eq, x)
            if not child:
                child = TSTNode (x)
                node.eq = _insert (node.eq, child)
            node = child
        # Check leaf
        if not _search (node.eq, self.leaf):
            node.eq = _insert (node.eq, TSTNode (self.leaf))
    def delete (self, seq):
        node = self.root
        for x in seq:
            node = _search (node.eq, x)
            if not node: return False
        # Delete leaf
        if _search (node.eq, self.leaf):
            node.eq = _delete (node.eq, self.leaf)
            return True
        return False
    def traverse (self):
        for x in _traverse (self.root.eq, self.leaf):
            yield x
    # The data with a common prefix
    def commonPrefix (self, seq):
        node = self.root
        buff = []
        for x in seq:
            buff.append (x)
            node = _search (node.eq, x)
            if not node: return
        for x in _traverse (node.eq, self.leaf):
            yield buff + x
if __name__ == '__main__':
    # Suffix trie
    def makeTST (seq):
        a = TST ()
        for x in xrange (len (seq)):
            a.insert (seq [x:])
        return a
    s = makeTST ('abcabbca')
    for x in s.traverse ():
        print x

```

```

for x in ['a', 'bc']:
    print x
    for y in s.commonPrefix (x):
        print y
print s.delete ('a')
print s.delete ('ca')
print s.delete ('bca')
for x in s.traverse ():
    print x
s = makeTST ([0,1,2,0,1,1,2,0])
for x in s.traverse ():
    print x

```

## 15.13 Comparing BSTs, Tries and TSTs

- Hash table and BST implementation stores complete the string at each node. As a result they take more time for searching. But they are memory efficient.
- TSTs can grow and shrink dynamically but hash tables resize only based on load factor.
- TSTs allow partial search whereas BSTs and hash tables do not support it.
- TSTs can display the words in sorted order, but in hash tables we cannot get the sorted order.
- Tries perform search operations very fast but they take huge memory for storing the string.
- TSTs combine the advantages of BSTs and Tries. That means they combine the memory efficiency of BSTs and the time efficiency of tries

## 15.14 Suffix Trees

Suffix trees are an important data structure for strings. With suffix trees we can answer the queries very fast. But this requires some preprocessing and construction of a suffix tree. Even though the construction of a suffix tree is complicated, it solves many other string-related problems in linear time.

**Note:** Suffix trees use a tree (suffix tree) for one string, whereas Hash tables, BSTs, Tries and TSTs store a set of strings. That means, a suffix tree answers the queries related to one string.

Let us see the terminology we use for this representation.

### Prefix and Suffix

Given a string  $T = T_1T_2 \dots T_n$ , the *prefix* of  $T$  is a string  $T_1 \dots T_i$  where  $i$  can take values from 1 to  $n$ . For example, if  $T = \text{banana}$ , then the prefixes of  $T$  are:  $b, ba, ban, bana, banan, banana$ .

Similarly, given a string  $T = T_1T_2 \dots T_n$ , the *suffix* of  $T$  is a string  $T_i \dots T_n$  where  $i$  can take values from  $n$  to 1. For example, if  $T = \text{banana}$ , then the suffixes of  $T$  are:  $a, na, ana, nana, anana, banana$ .

### Observation

From the above example, we can easily see that for a given text  $T$  and pattern  $P$ , the exact string matching problem can also be defined as:

- Find a suffix of  $T$  such that  $P$  is a prefix of this suffix or
- Find a prefix of  $T$  such that  $P$  is a suffix of this prefix.

**Example:** Let the text to be searched be  $T = \text{accbkkbac}$  and the pattern be  $P = \text{kbb}$ . For this example,  $P$  is a prefix of the suffix  $\text{kbbac}$  and also a suffix of the prefix  $\text{accbkkb}$ .

### What is a Suffix Tree?

In simple terms, the suffix tree for text  $T$  is a Trie-like data structure that represents the suffixes of  $T$ . The definition of suffix trees can be given as: A suffix tree for a  $n$  character string  $T[1 \dots n]$  is a rooted tree with the following properties.

- A suffix tree will contain  $n$  leaves which are numbered from 1 to  $n$
- Each internal node (except root) should have at least 2 children
- Each edge in a tree is labeled by a nonempty substring of  $T$
- No two edges of a node (children edges) begin with the same character
- The paths from the root to the leaves represent all the suffixes of  $T$

## The Construction of Suffix Trees

### Algorithm

1. Let  $S$  be the set of all suffixes of  $T$ . Append  $\$$  to each of the suffixes.
2. Sort the suffixes in  $S$  based on their first character.
3. For each group  $S_c$  ( $c \in \Sigma$ ):
  - (i) If  $S_c$  group has only one element, then create a leaf node.
  - (ii) Otherwise, find the longest common prefix of the suffixes in  $S_c$  group, create an internal node, and recursively continue with Step 2,  $S$  being the set of remaining suffixes from  $S_c$  after splitting off the longest common prefix.

For better understanding, let us go through an example. Let the given text be  $T = \text{tatat}$ . For this string, give a number to each of the suffixes.

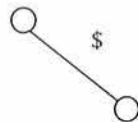
Index	Suffix
1	$\$$
2	$t\$$
3	$at\$$
4	$tat\$$
5	$atat\$$
6	$tatat\$$

Now, sort the suffixes based on their initial characters.

Index	Suffix
1	$\$$
3	$at\$$
5	$atat\$$
2	$t\$$
4	$tat\$$
6	$tatat\$$

} Group  $S_1$  based on  $a$   
 } Group  $S_2$  based on  $a$   
 } Group  $S_3$  based on  $t$

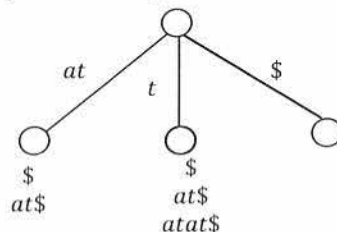
In the three groups, the first group has only one element. So, as per the algorithm, create a leaf node for it, as shown below.



Now, for  $S_2$  and  $S_3$  (as they have more than one element), let us find the longest prefix in the group, and the result is shown below.

Group	Indexes for this group	Longest Prefix of Group Suffixes
$S_2$	3, 5	$at$
$S_3$	2, 4, 6	$t$

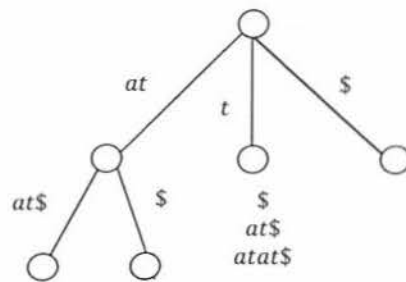
For  $S_2$  and  $S_3$ , create internal nodes, and the edge contains the longest common prefix of those groups.



Now we have to remove the longest common prefix from the  $S_2$  and  $S_3$  group elements.

Group	Indexes for this group	Longest Prefix of Group Suffixes	Resultant Suffixes
$S_2$	3, 5	$at$	$\$, at\$$
$S_3$	2, 4, 6	$t$	$\$, at\$, atat\$$

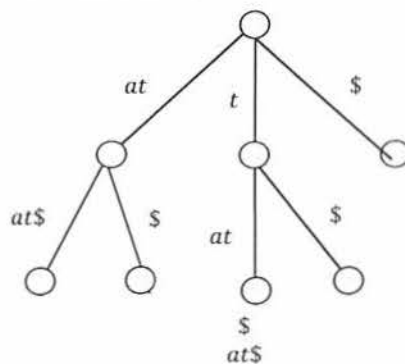
Our next step is solving  $S_2$  and  $S_3$  recursively. First let us take  $S_2$ . In this group, if we sort them based on their first character, it is easy to see that the first group contains only one element  $\$$ , and the second group also contains only one element,  $at\$$ . Since both groups have only one element, we can directly create leaf nodes for them.



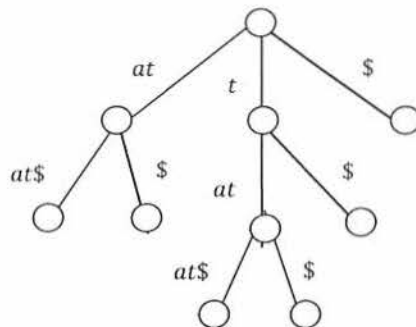
At this step, both  $S_1$  and  $S_2$  elements are done and the only remaining group is  $S_3$ . As similar to earlier steps, in the  $S_3$  group, if we sort them based on their first character, it is easy to see that there is only one element in the first group and it is \$. For  $S_3$  remaining elements, remove the longest common prefix.

Group	Indexes for this group	Longest Prefix of Group Suffixes	Resultant Suffixes
$S_3$	4, 6	$at$	$at$, at$$

In the  $S_3$  second group, there are two elements: \$ and  $at$$ . We can directly add the leaf nodes for the first group element \$. Let us add  $S_3$  subtree as shown below.



Now,  $S_3$  contains two elements. If we sort them based on their first character, it is easy to see that there are only two elements and among them one is \$ and other is  $at$$ . We can directly add the leaf nodes for them. Let us add  $S_3$  subtree as shown below.



Since there are no more elements, this is the completion of the construction of the suffix tree for string  $T = atat$ . The time-complexity of the construction of a suffix tree using the above algorithm is  $O(n^2)$  where  $n$  is the length of the input string because there are  $n$  distinct suffixes. The longest has length  $n$ , the second longest has length  $n - 1$ , and so on.

#### Note:

- There are  $O(n)$  algorithms for constructing suffix trees.
- To improve the complexity, we can use indices instead of string for branches.

## Applications of Suffix Trees

All the problems below (but not limited to these) on strings can be solved with suffix trees very efficiently (for algorithms refer to *Problems* section).

- **Exact String Matching:** Given a text  $T$  and a pattern  $P$ , how do we check whether  $P$  appears in  $T$  or not?
- **Longest Repeated Substring:** Given a text  $T$  how do we find the substring of  $T$  that is the maximum repeated substring?



- **Longest Palindrome:** Given a text  $T$  how do we find the substring of  $T$  that is the longest palindrome of  $T$ ?
- **Longest Common Substring:** Given two strings, how do we find the longest common substring?
- **Longest Common Prefix:** Given two strings  $X[i \dots n]$  and  $Y[j \dots m]$ , how do we find the longest common prefix?
- How do we search for a regular expression in given text  $T$ ?
- Given a text  $T$  and a pattern  $P$ , how do we find the first occurrence of  $P$  in  $T$ ?

## 15.15 String Algorithms: Problems & Solutions

**Problem-1** Given a paragraph of words, give an algorithm for finding the word which appears the maximum number of times. If the paragraph is scrolled down (some words disappear from the first frame, some words still appear, and some are new words), give the maximum occurring word. Thus, it should be dynamic.

**Solution:** For this problem we can use a combination of priority queues and tries. We start by creating a trie in which we insert a word as it appears, and at every leaf of trie. Its node contains that word along with a pointer that points to the node in the heap [priority queue] which we also create. This heap contains nodes whose structure contains a *counter*. This is its frequency and also a pointer to that leaf of trie, which contains that word so that there is no need to store the word twice.

Whenever a new word comes up, we find it in trie. If it is already there, we increase the frequency of that node in the heap corresponding to that word, and we call it heapify. This is done so that at any point of time we can get the word of maximum frequency. While scrolling, when a word goes out of scope, we decrement the counter in heap. If the new frequency is still greater than zero, heapify the heap to incorporate the modification. If the new frequency is zero, delete the node from heap and delete it from trie.

**Problem-2** Given two strings, how can we find the longest common substring?

**Solution:** Let us assume that the given two strings are  $T_1$  and  $T_2$ . The longest common substring of two strings,  $T_1$  and  $T_2$ , can be found by building a generalized suffix tree for  $T_1$  and  $T_2$ . That means we need to build a single suffix tree for both the strings. Each node is marked to indicate if it represents a suffix of  $T_1$  or  $T_2$  or both. This indicates that we need to use different marker symbols for both the strings (for example, we can use \$ for the first string and # for the second symbol). After constructing the common suffix tree, the deepest node marked for both  $T_1$  and  $T_2$  represents the longest common substring.

**Another way of doing this is:** We can build a suffix tree for the string  $T_1\$T_2\#$ . This is equivalent to building a common suffix tree for both the strings.

Time Complexity:  $O(m + n)$ , where  $m$  and  $n$  are the lengths of input strings  $T_1$  and  $T_2$ .

**Problem-3** **Longest Palindrome:** Given a text  $T$  how do we find the substring of  $T$  which is the longest palindrome of  $T$ ?

**Solution:** The longest palindrome of  $T[1..n]$  can be found in  $O(n)$  time. The algorithm is: first build a suffix tree for  $T\$reverse(T)\#$  or build a generalized suffix tree for  $T$  and  $reverse(T)$ . After building the suffix tree, find the deepest node marked with both \$ and #. Basically it means find the longest common substring.

**Problem-4** Given a string (word), give an algorithm for finding the next word in the dictionary.

**Solution:** Let us assume that we are using Trie for storing the dictionary words. To find the next word in Tries we can follow a simple approach as shown below. Starting from the rightmost character, increment the characters one by one. Once we reach Z, move to the next character on the left side. Whenever we increment, check if the word with the incremented character exists in the dictionary or not. If it exists, then return the word, otherwise increment again. If we use *TST*, then we can find the inorder successor for the current word.

**Problem-5** Give an algorithm for reversing a string.

**Solution:**

```
# If the str is editable
def ReversingString(str):
    s = list(str)
    end = len(str)-1
    start = 0
    while (start<end):
        temp = s[start]
        s[start] = s[end]
        s[end] = temp
        start += 1
        end -= 1
    return "".join(s)
```

```

str = "CareerMonk Publications."
print ReversingString(str)

# Alternative Implementation
def reverse(str):
    r = ""
    for c in str:
        r = c + r
    return r
str = "CareerMonk Publications."
print reverse(str)

```

Time Complexity:  $O(n)$ , where  $n$  is the length of the given string. Space Complexity:  $O(n)$ .

**Problem-6** Can we reverse the string without using any temporary variable?

**Solution: Yes**, we can use XOR logic for swapping the variables.

```

def ReversingString(str):
    s = list(str)
    end = len(str)-1
    start = 0
    while (start<end):
        s[start], s[end] = s[end], s[start]
        start += 1
        end -= 1

    return "".join(s)
str = "CareerMonk Publications."
print ReversingString(str)

# Alternative Implementation
str = "CareerMonk Publications."
print "".join(str[c] for c in xrange(len(str) - 1, -1, -1))

```

Probably the easiest and close to the fastest way to reverse a string is to use Python's extended slice syntax. This allows you to specify a start, stop and step value to use when creating a slice. The syntax is: [start:stop:step].

```

str = "CareerMonk Publications."
print str[::-1]

```

If start is omitted it defaults to 0 and if stop is omitted it defaults to the length of the string. A step of -1 tells Python to start counting by 1 from the stop until it reaches the start.

When working with large strings, or when you just don't want to reverse the whole string at once, you can use the reversed() built-in. reversed() returns an iterator and is arguably the most Pythonic way to reverse a string.

```

str = "CareerMonk Publications."
print "".join(reversed(str))

```

Time Complexity:  $O(\frac{n}{2}) \approx O(n)$ , where  $n$  is the length of the given string. Space Complexity:  $O(1)$ .

**Problem-7** a text and a pattern, give an algorithm for matching the pattern in the text. Assume ? (single character matcher) and \* (multi character matcher) are the wild card characters.

**Solution: Brute Force Method.** For efficient method, refer to the theory section.

```

def wildcardMatch(inputString, pattern):
    if len(pattern) == 0:
        return len(inputString) == 0
    # inputString can be empty
    if pattern[0] == '?':
        return len(inputString) > 0 and wildcardMatch(inputString[1:], pattern[1:])
    elif pattern[0] == '*':
        # match nothing or
        # match one and continue, AB* = A*
        return wildcardMatch(inputString, pattern[1:]) or \
            (len(inputString) > 0 and wildcardMatch(inputString[1:], pattern))
    else:
        return len(inputString) > 0 and inputString[0] == pattern[0] and \
            wildcardMatch(inputString[1:], pattern[1:])

    return 0

```

```

print wildcardMatch("cc", "c")
print wildcardMatch("cc", "cc")
print wildcardMatch("ccc", "cc")
print wildcardMatch("cc", "*")
print wildcardMatch("cc", "a*")
print wildcardMatch("ab", "?*")
print wildcardMatch("cca", "c*a*b")

```

Time Complexity:  $O(mn)$ , where  $m$  is the length of the text and  $n$  is the length of the pattern.

Space Complexity:  $O(1)$ .

**Problem-8** Give an algorithm for reversing words in a sentence.

**Example:** Input: "This is a Career Monk String", Output: "String Monk Career a is This"

**Solution:** Start from the beginning and keep on reversing the words. The below implementation assumes that " " (space) is the delimiter for words in given sentence.

```

# @param s, a string
# @return a string
def reverseWordsInSentence(self, s):
    result = []
    inWord = False
    for i in range(0, len(s)):
        if (s[i] == ' ' or s[i] == '\t') and inWord:
            inWord = False
            result.insert(0, s[start:i])
            result.insert(0, ' ')
        elif not (s[i] == ' ' or s[i] == '\t') and inWord:
            inWord = True
            start = i
    if inWord:
        result.insert(0, s[start:len(s)])
        result.insert(0, ' ')
    if len(result) > 0:
        result.pop(0)
    return "".join(result)

```

Time Complexity:  $O(2n) \approx O(n)$ , where  $n$  is the length of the string. Space Complexity:  $O(1)$ .

**Problem-9 Permutations of a string [anagrams]:** Give an algorithm for printing all possible permutations of the characters in a string. Unlike combinations, two permutations are considered distinct if they contain the same characters but in a different order. For simplicity assume that each occurrence of a repeated character is a distinct character. That is, if the input is "aaa", the output should be six repetitions of "aaa". The permutations may be output in any order.

**Solution:** The solution is reached by generating  $n!$  strings, each of length  $n$ , where  $n$  is the length of the input string. A generator function that generates all permutations of the input elements. If the input contains duplicates, then some permutations may be visited with multiplicity greater than one.

Our recursive algorithm requires two pieces of information, the elements that have not yet been permuted and the partial permutation built up so far. We thus phrase this function as a wrapper around a recursive function with extra parameters.

```

def permutations(elems):
    for perm in recursivePermutations(elems, []):
        print perm

```

A helper function to recursively generate permutations. The function takes in two arguments, the elements to permute and the partial permutation created so far, and then produces all permutations that start with the given sequence and end with some permutations of the unpermuted elements.

```

def recursivePermutations(elems, soFar):
    # Base case: If there are no more elements to permute, then the answer will
    # be the permutation we have created so far.
    if len(elems) == 0:
        yield soFar
    # Otherwise, try extending the permutation we have so far by each of the
    # elements we have yet to permute.
    else:

```

```

    for i in range(0, len(elems)):
        # Extend the current permutation by the ith element, then remove
        # the ith element from the set of elements we have not yet
        # permuted. We then iterate across all the permutations that have
        # been generated this way and hand each one back to the caller.
        for perm in recursivePermutations(elems[0:i] + elems[i+1:], soFar + [elems[i]]):
            yield perm
# Permutations by iteration
def permutationByIteration(elems):
    level=[elems[0]]
    for i in range(1,len(elems)):
        nList=[]
        for item in level:
            nList.append(item+elems[i])
            for j in range(len(item)):
                nList.append(item[0:j]+elems[i]+item[j:])
        level=nList
    return nList

```

**Problem-10 Combinations of a String:** Unlike permutations, two combinations are considered to be the same if they contain the same characters, but may be in a different order. Give an algorithm that prints all possible combinations of the characters in a string. For example, "ac" and "ab" are different combinations from the input string "abc", but "ab" is the same as "ba".

**Solution:** The solution is achieved by generating  $n!/r!(n-r)!$  strings, each of length between 1 and  $n$  where  $n$  is the length of the given input string.

**Algorithm:**

- For each of the input characters
  - a. Put the current character in output string and print it.
  - b. If there are any remaining characters, generate combinations with those remaining characters.

```

def combinationByRecursion(elems, s, idx, li):
    for i in range(idx, len(elems)):
        s+=elems[i]
        li.append(s)
        #print s, idx
        combinationByRecursion(elems, s, i+1, li)
        s=s[0:-1]
def combinationByIteration(elems):
    level=[""]
    for i in range(len(elems)):
        nList=[]
        for item in level:
            nList.append(item+elems[i])
        level+=nList
    return level[1:]
res=[]
combinationByRecursion('abc', "", 0, res)
print combinationByIteration('abc')
print combinationByIteration('abc')

```

**Problem-11** Given a string "ABCCBCBA", give an algorithm for recursively removing the adjacent characters if they are the same. For example, ABCCBCBA --> ABBCBA-->ACBA

**Solution:** First we need to check if we have a character pair; if yes, then cancel it. Now check for next character and previous element. Keep canceling the characters until we either reach the start of the array, reach the end of the array, or don't find a pair.

```

def removeAdjacentRepeats(nums):
    i = 1
    while i < len(nums):
        if nums[i] == nums[i-1]:
            nums.pop(i)
            i -= 1
        i += 1

```



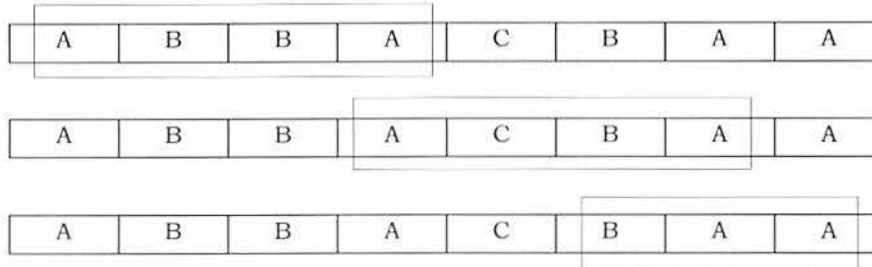
```

return nums
nums=["A","B","C","C","C","C","B","A"]
print removeAdjacent(nums)

```

**Problem-12** Given a set of characters *CHARS* and a input string *INPUT*, find the minimum window in *str* which will contain all the characters in *CHARS* in complexity  $O(n)$ . For example, *INPUT* = *ABBACBAA* and *CHARS* = *AAB* has the minimum window *BAA*.

**Solution:** This algorithm is based on the sliding window approach. In this approach, we start from the beginning of the array and move to the right. As soon as we have a window which has all the required elements, try sliding the window as far right as possible with all the required elements. If the current window length is less than the minimum length found until now, update the minimum length. For example, if the input array is *ABBACBAA* and the minimum window should cover characters *AAB*, then the sliding window will move like this:



**Algorithm:** The input is the given array and chars is the array of characters that need to be found.

- 1 Make an integer array shouldfind[] of len 256. The  $i^{th}$  element of this array will have the count of how many times we need to find the element of ASCII value  $i$ .
- 2 Make another array hasfound of 256 elements, which will have the count of the required elements found until now.
- 3 Count  $\leq 0$
- 4 While input[i]
  - a. If input[i] element is not to be found  $\rightarrow$  continue
  - b. If input[i] element is required  $\Rightarrow$  increase count by 1.
  - c. If count is length of chars[] array, slide the window as much right as possible.
  - d. If current window length is less than min length found until now, update min length.

```

from collections import defaultdict
def smallestWindow(INPUT, CHARS):
    assert CHARS != ""
    disctionary = defaultdict(int)
    nneg = [0] # number of negative entries in dictionary
    def incr(c):
        disctionary[c] += 1
        if disctionary[c] == 0:
            nneg[0] -= 1
    def decr(c):
        if disctionary[c] == 0:
            nneg[0] += 1
        disctionary[c] -= 1
    for c in CHARS:
        decr(c)
    minLength = len(INPUT) + 1
    j = 0
    for i in xrange(len(INPUT)):
        while nneg[0] > 0:
            if j >= len(INPUT):
                return minLength
            incr(INPUT[j])
            j += 1
        minLength = min(minLength, j - i)
        decr(INPUT[i])
    return minLength
print smallestWindow("ADOBECODEBANC","ABC")

```

Complexity: If we walk through the code,  $i$  and  $j$  can traverse at most  $n$  steps (where  $n$  is the input size) in the worst case, adding to a total of  $2n$  times. Therefore, time complexity is  $O(n)$ .



**Problem-13** Given two strings *str1* and *str2*, write a function that prints all interleavings of the given two strings. We may assume that all characters in both strings are different. Example: Input: *str1* = "AB", *str2* = "CD" and Output: ABCD ACBD ACDB CABD CADB CDAB. An interleaved string of given two strings preserves the order of characters in individual strings. For example, in all the interleavings of above first example, 'A' comes before 'B' and 'C' comes before 'D'.

**Solution:** Let the length of *str1* be *m* and the length of *str2* be *n*. Let us assume that all characters in *str1* and *str2* are different. Let  $\text{Count}(m, n)$  be the count of all interleaved strings in such strings. The value of  $\text{Count}(m, n)$  can be written as following.

$$\begin{aligned}\text{Count}(m, n) &= \text{Count}(m-1, n) + \text{Count}(m, n-1) \\ \text{Count}(1, 0) &= 1 \text{ and } \text{Count}(0, 1) = 1\end{aligned}$$

To print all interleavings, we can first fix the first character of *str1*[0..m-1] in output string, and recursively call for *str1*[1..m-1] and *str2*[0..n-1]. And then we can fix the first character of *str2*[0..n-1] and recursively call for *str1*[0..m-1] and *str2*[1..n-1].

On other words, this problem can be reduced to that of creating all unique permutations of a particular list. Say *m* and *n* are the lengths of the strings *str1* and *str2*, respectively. Then construct a list like this:

[0] \* *str1* + [1] \* *str2*

There exists a one-to-one correspondence (a bijection) from the unique permutations of this list to all the possible interleavings of the two strings *str1* and *str2*. The idea is to let each value of the permutation specify which string to take the next character from.

```
def PrintInterleavings(str1, str2):
    perms = []
    if len(str1) + len(str2) == 1:
        return [str1 or str2]
    if str1:
        for item in PrintInterleavings(str1[1:], str2):
            perms.append(str1[0] + item)
    if str2:
        for item in PrintInterleavings(str1, str2[1:]):
            perms.append(str2[0] + item)
    return perms
print PrintInterleavings("AB", "CD")
```

**Problem-14** Given a matrix with size  $n \times n$  containing random integers. Give an algorithm which checks whether rows match with a column(s) or not. For example, if  $i^{th}$  row matches with  $j^{th}$  column, and  $i^{th}$  row contains the elements - [2,6,5,8,9]. Then  $j^{th}$  column would also contain the elements - [2,6,5,8,9].

**Solution:** We can build a trie for the data in the columns (rows would also work). Then we can compare the rows with the trie. This would allow us to exit as soon as the beginning of a row does not match any column (backtracking). Also this would let us check a row against all columns in one pass.

If we do not want to waste memory for empty pointers then we can further improve the solution by constructing a suffix tree.

**Problem-15** How do you replace all spaces in a string with '%20'. Assume string has sufficient space at end of string to hold additional characters.

**Solution:**

```
class ReplacableString:
    def __init__(self, inputString):
        self.inputString = inputString
    def replacer(self, to_replace, replacer):
        for i in xrange(len(self.inputString)):
            if to_replace == self.inputString[i:len(to_replace)]:
                self.inputString = self.inputString[:i] + replacer + self.inputString[i+len(to_replace):]
    def __str__(self):
        return str(self.inputString)
input = ReplacableString("This is eth string")
input.replacer(" ", "%20")
print input
```

Time Complexity:  $O(n)$ . Space Complexity:  $O(1)$ . Here, we do not have to worry on the space needed for extra characters. We have to see how much extra space is needed for filling that.

**Important note:** Python provides a simple way to encode URLs.

```
import urllib
input_url = urllib.quote ( 'http://www.CareerMonk.com/example one.html' )
```

In this example, Python loads the urllib module, then takes the string and normalizes the URL by replacing the unreadable blank space in the URL between "example one.html" with the special character "%20".

**Problem-16** Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'. A region is captured by flipping all 'O's into 'X's in that surrounded region .

Sample Input:	Output:
X X X X	X X X X
X O O X	X X X X
X X O X	X X X X
X O X X	X O X X

**Solution:** We use backtracking to identify the elements not surrounded by 'X' and we mark those with a temporal symbol ('\$'). The elements not surrounded by 'X' means that exists a path of elements 'O' to a border. So we start the backtracking algorithm with the borders. The last thing is replacing the temporal element by 'O' and the rest elements to 'X'.

```
class CamptureRegions:
    # @param board, a 2D array
    # Capture all regions by modifying the input board in-place.
    # Do not return any value.
    def solve(self, board):
        if len(board)==0:
            return
        for row in range(0,len(board)):
            self.mark(board,row,0)
            self.mark(board,row,len(board[0])-1)
        for col in range(0, len(board[0])):
            self.mark(board, 0, col)
            self.mark(board, len(board)-1, col)
        for row in range(0,len(board)):
            for col in range(0, len(board[0])):
                if board[row][col]=='$':
                    board[row][col] = 'O'
                else:
                    board[row][col] = 'X'
    def mark(self, board, row, col):
        stack = []
        nCols= len(board[0])
        stack.append(row*nCols+col)
        while len(stack)>0:
            position = stack. pop()
            row = position // nCols
            col = position % nCols
            if board[row][col] != 'O':
                continue
            board[row][col] = '$'
            if row>0:
                stack.append((row-1)*nCols+col)
            if row< len(board)- 1:
                stack.append((row+1)*nCols+col)
            if col>0:
                stack.append(row*nCols+col-1)
            if col < nCols-1:
                stack.append(row*nCols+col+1)
```

**Problem-17** If h is any hashing function and is used to hash  $n$  keys in to a table of size  $m$ , where  $n \leq m$ , the expected number of collisions involving a particular key  $X$  is :

A) less than 1. B) less than  $n$ . C) less than  $m$ . D) less than  $n/2$ .

**Solution:** A. Hash function should distribute the elements uniformly