```
char word[1024];
void DisplayAllWords(struct TSTNode *root) {
    if(!root)
        return;
    DisplayAllWords(root→left);
    word[i] = root→data;
    if(root→is_End_Of_String) {
            word[i] = '\0';
            printf("%c", word);
    }
    i++;
    DisplayAllWords(root→eq);

    i--;
    DisplayAllWords(root→right);
}
```

Finding the Length of the Largest Word in TST

This is similar to finding the height of the BST and can be found as:

```
int MaxLengthOfLargestWordInTST(struct TSTNode *root) {
    if(!root)
        return 0;
    return Max(MaxLengthOfLargestWordInTST(root→left),
            MaxLengthOfLargestWordInTST(root→eq)+1,
            MaxLengthOfLargestWordInTST(root→right)));
}
```

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

Similarly, given a string $T = T_1T_2 \ldots T_n$, the *suffix* of T is a string $T_i \ldots T_n$ where i can take values from n to 1. For example, if $T = banana$, 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 $T = acebkkbac$ and the pattern be $P = kkb$. For this example, P is a prefix of the suffix *kkbac* and also a suffix of the prefix *acebkkb.*

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 $T[1 \ldots n]$ 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 ($c \in \sum$):
   (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* = *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$*

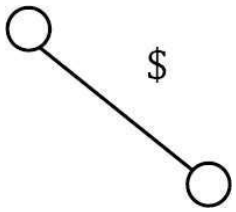
Now, sort the suffixes based on their initial characters.

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

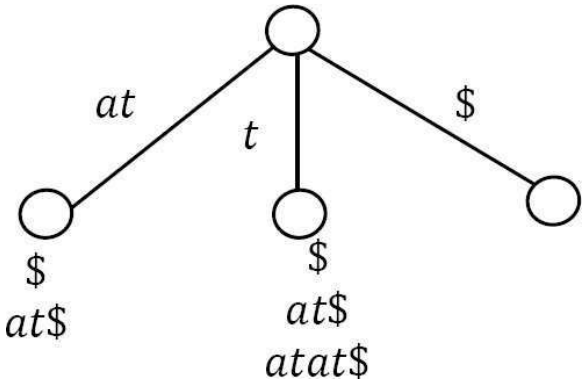
node for it, as shown below.



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

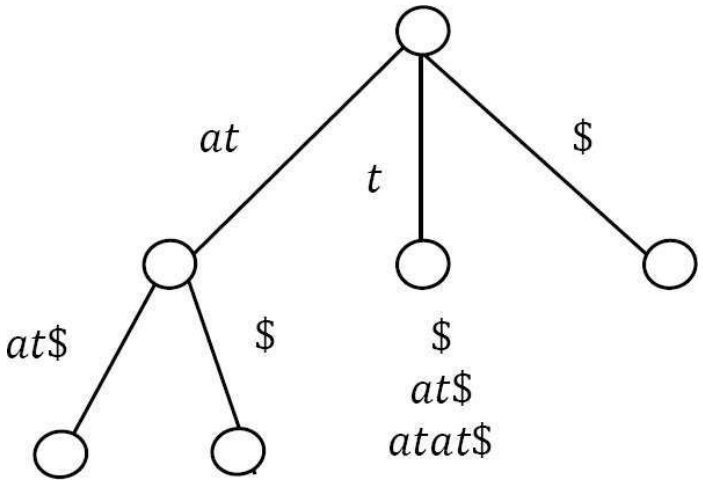
For S_2 and S_3, create internal nodes, and the edge contains the longest common prefix of those groups.



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	$\$, at\$, atat\$$

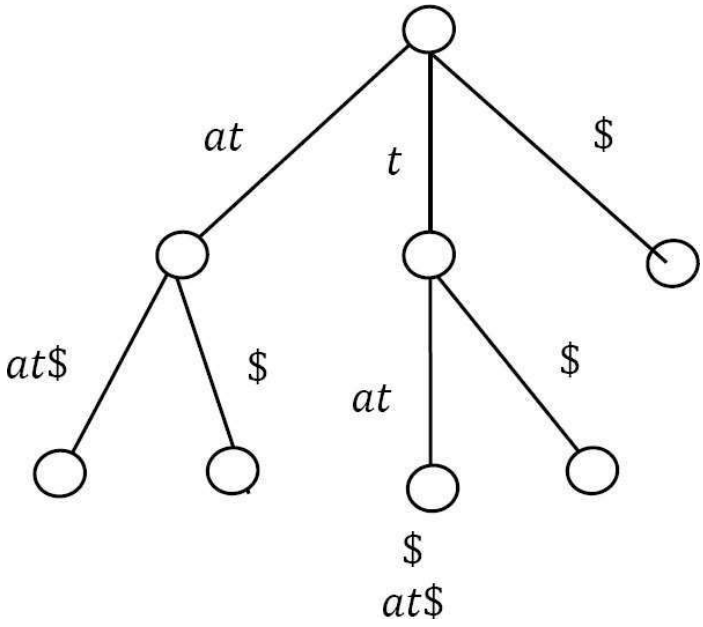
Out next step is solving S_2 and S_3 recursively. First let us take S_2. In this group, if we sort them based on their first character, it is easy to see that the first group contains only one element $, and the second group also contains only one element, *at*$. Since both groups have only one element,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



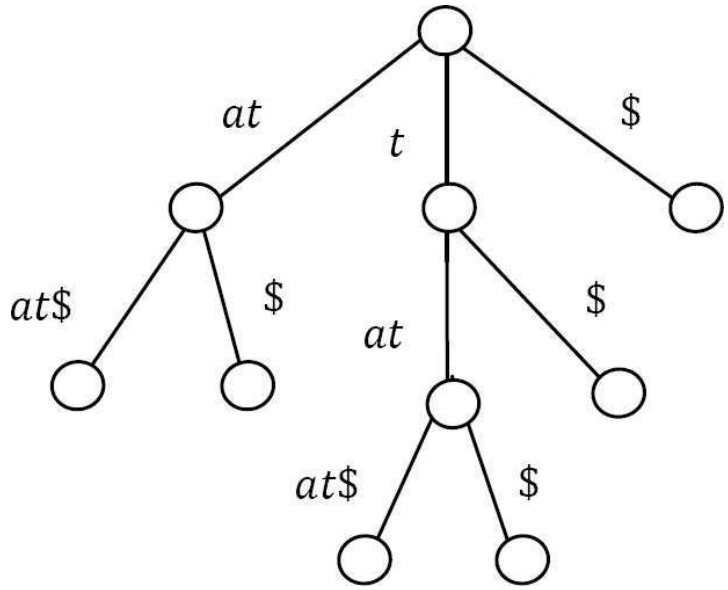
At this step, both S_1 and S_2 elements are done and the only remaining group is S_3. As similar to earlier steps, in the S_3 group, if we sort them based on their first character, it is easy to see that there is only one element in the first group and it is \$. For S_3 remaining elements, remove the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

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

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 $T = tatat$. 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 $X[i \ldots n]$ and $Y[j \ldots m]$,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
char *ReversingString(char str[]) {
        char temp, start, end;
        if(str == NULL || *str == '\0')
                return str;
        for (end = 0; str[end]; end++);
        end--;
        for (start = 0; start < end; start++, end--) {
                temp = str[start]; str[start] = str[end]; str[end] = temp;
        }
        return str;
}
```

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

Problem-6 If the string is not editable, how do we create a string that is the reverse of the given string?

Solution: If the string is not editable, then we need to create an array and return the pointer of that.

```
//If str is a const string (not editable)
char* ReversingString(char* str) {
        int start, end, len;
        char temp, *ptr=NULL;
        len=strlen(str);
        ptr=malloc(sizeof(char)*(len+1));
        ptr=strcpy(ptr,str);
        for (start=0, end=len-1; start<=end; start++, end--) {        //Swapping
                temp=ptr[start]; ptr[start]=ptr[end]; ptr[end]=temp;
        }
        return ptr;
}
```

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

Problem-7 Can we reverse the string without using any temporary variable?

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

```
char* ReversingString(char *str) {
        int start = 0, end= strlen(str)-1;
        while( start<end ) {
                str[start] ^= str[end];     str[end]  ^=  str[start];    str[start] ^= str[end];
                ++start;
                --end;
        }
        return str;
}
```

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

Problem-8 Given 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.

```
int PatternMatching(char *text, char *pattern) {
    if(*pattern == 0)
        return 1;
    if(*text == 0)
        return *p == 0;
    if('?' == *pattern)
        return PatternMatching(text+1,pattern+1) || PatternMatching(text,pattern+1);
    if('*' == *pattern)
        return PatternMatching(text+1,pattern) || PatternMatching(text,pattern+1);
    if(*text == *pattern)
        return PatternMatching(text+1,pattern+1);
    return -1;
}
```

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-9 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.

```
void ReverseWordsInSentences(char *text) {
    int wordStart, wordEnd, length;
    length = strlen(text);
    ReversingString(text, 0, length-1);
    for(wordStart = wordEnd = 0; wordEnd < length; wordEnd ++) {
        if(text[wordEnd] ! = ' ') {
            wordStart = wordEnd;
            while (text[wordEnd] != ' ' && wordEnd < length)
                wordEnd ++;
            wordEnd--;
            ReversingString(text, wordStart, wordEnd); //Found current word, reverse it now.
        }
    }
}
void ReversingString(char text[], int start, int end) {
    for (char temp; start < end; start++, end--) {
        temp = str[end];
        str[end] = str[start];
        strstart] = temp;
    }
}
```

Time Complexity: O(2*n*) ≈ O(*n*), where *n* is the length of the string. Space Complexity: O(1).

Problem-10 **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.

```
void Permutations(int depth, char *permutation, int *used, char *original) {
    int length = strlen(original);
    if(depth == length)
        printf("%s",permutation);
    else {
        for (int i = 0; i < length; i++) {
            if(!used[i]) {
                used[i] = 1;
                permutation[depth] = original[i];
                Permutations(depth + 1, permutation, used, original);
                used[i] = 0;
            }
        }
    }
}
```

Problem-11 Combinations 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.

```
void Combinations(int depth, char *combination, int start, char *original) {
    int length = strlen(original);
    for (int i = start; i < length; i++) {
        combination[depth] = original[i];
        combination[depth +1] = '\0';
        printf("%s", combination);
        if(i < length -1)
            Combinations(depth + 1, combination, start + 1, original);
    }
}
```

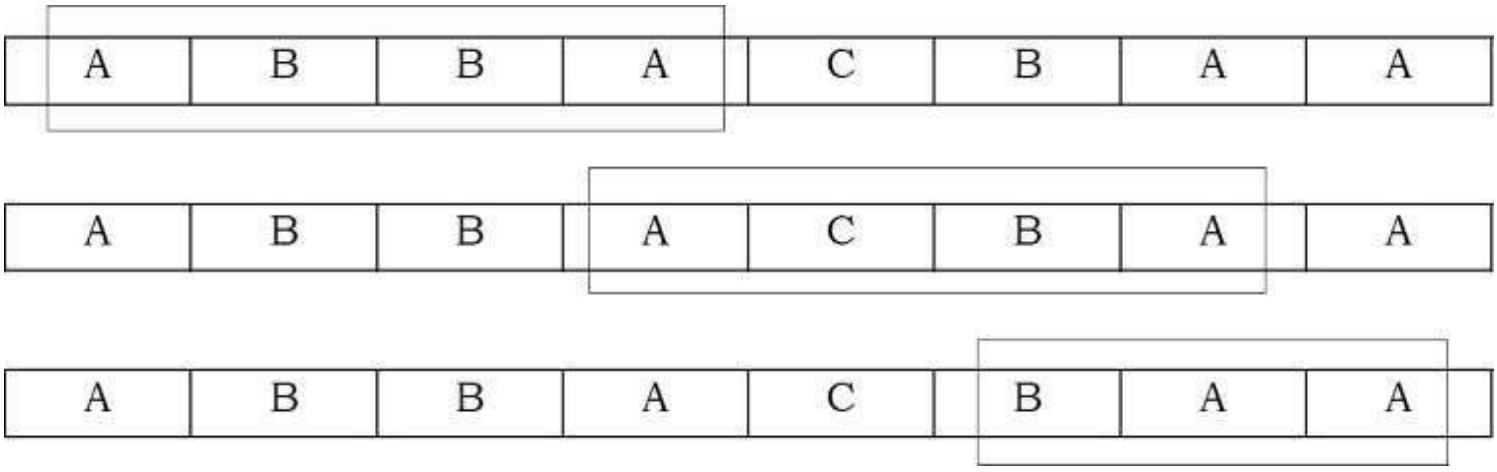
Problem-12 Given a string "ABCCBCBA", give an algorithm for recursively removing the adjacent characters if they are the same. For example, ABCCBCBA nnnnnn> ABBCBA->ACBA

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.

```
void RremoveAdjacentPairs(char* str) {
    int len = strlen(str), i, j = 0;
    for (i=1; i <= len; i++) {
        while ((str[i] == str[j]) && (j >= 0)){        //Cancel pairs
            i++;
            j--;
        }
        str[++j] = str[i];
    }
    return;
}
```

Problem-13 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 <= 0
4 While input[i]
    a. If input[i] element is not to be found → continue
    b. If input[i] element is required => 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.

```
#define MAX 256
void MinLengthWindow(char input[], char chars[]) {
        int shouldfind[MAX] = {0,}, hasfound[MAX] = {0,};
        int j=0, cnt = 0, start=0, finish, minwindow = INT_MAX;
        int charlen = strlen(chars), iplen = strlen(input);
        for (int i=0; i< charlen; i++)
                shouldfind[chars[i]] += 1;
        finish = iplen;
        for (int i=0; i< iplen; i++) {
                if(!shouldfind[input[i]])
                        continue;
                hasfound[input[i]] += 1;
                if(shouldfind[input[i]] >= hasfound[input[i]])
                        cnt++;
                if(cnt == charlen) {
                        while (shouldfind[input[j]] == 0 || hasfound[input[j]] > shouldfind[input[j]])          {
                                if(hasfound[input[j]] > shouldfind[input[j]])
                                        hasfound[input[j]]--;
                                j++;
                        }
                        if(minwindow > (i - j +1)) {
                                minwindow = i - j +1;
                                finish = i;
                                start = j;
                        }
                }
        }
        printf("Start:%d and Finish: %d", start, finish);
}
```

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 2*n* times. Therefore, time complexity is O(*n*).

Problem-14 We are given a 2D array of characters and a character pattern. Give an algorithm to find if the pattern is present in the 2D array. The pattern can be in any order (all 8 neighbors to be considered) but we can't use the same character twice while matching. Return 1 if match is found, 0 if not. For example: Find "MICROSOFT" in the below matrix.

A	C	P	**R**	C
X	**S**	**O**	P	**C**
V	**O**	V	N	**I**
W	G	**F**	**M**	N
Q	A	**T**	I	T

Solution: Manually finding the solution of this problem is relatively intuitive; we just need to describe an algorithm for it. Ironically, describing the algorithm is not the easy part.

How do we do it manually? First we match the first element, and when it is matched we match the second element in the 8 neighbors of the first match. We do this process recursively, and when the last character of the input pattern matches, return true.

During the above process, take care not to use any cell in the 2D array twice. For this purpose, you mark every visited cell with some sign. If your pattern matching fails at some point, start matching from the beginning (of the pattern) in the remaining cells. When returning, you unmark the visited cells.

Let's convert the above intuitive method into an algorithm. Since we are doing similar checks for pattern matching every time, a recursive solution is what we need. In a recursive solution, we need to check if the substring passed is matched in the given matrix or not. The condition is not to use the already used cell, and to find the already used cell, we need to add another 2D array to the function (or we can use an unused bit in the input array itself.) Also, we need the current position of the input matrix from where we need to start. Since we need to pass a lot more information than is actually given, we should be having a wrapper function to initialize the extra information to be passed.

Algorithm:

```
If we are past the last character in the pattern
        Return true
If we get a used cell again
        Return false if we got past the 2D matrix
        Return false
If searching for first element and cell doesn't match
        FindMatch with next cell in row-first order (or column-first order)
```

Otherwise if character matches
    mark this cell as used
    res = FindMatch with next position of pattern in 8 neighbors
    mark this cell as unused
    Return res
Otherwise
    Return false

```
#define MAX 100
boolean FindMatch_wrapper(char mat[MAX][MAX], char *pat, int nrow, int ncol) {
      if(strlen(pat) > nrow*ncol) return false;
      int used[MAX][MAX] = {{0,},};
      return FindMatch(mat, pat, used, 0, 0, nrow, ncol, 0);
}
//level: index till which pattern is matched & x, y: current position in 2D array
boolean FindMatch(char mat[MAX][MAX], char *pat, int used[MAX][MAX],
                                        int x, int y, int nrow, int ncol, int level) {
   if(level == strlen(pat)) //pattern matched
      return true;
   if(nrow == x || ncol == y)   return false;
   if(used[x][y])        return false;
   if(mat[x][y] != pat[level] && level == 0) {
      if(x < (nrow - 1))
         return FindMatch(mat, pat, used, x+1, y, nrow, ncol, level); //next element in same row
      else if(y < (ncol - 1))
         return FindMatch(mat, pat, used, 0, y+1, nrow, ncol, level); //first element from same column
      else  return false;
   }
   else if(mat[x][y] == pat[level]) {
      boolean res;
      used[x][y] = 1;       //marking this cell as used
      //finding subpattern in 8 neighbors
      res = (x > 0  ? FindMatch(mat, pat, used, x-1, y, nrow, ncol, level+1) :   false) ||
         (res = x < (nrow - 1)  ? FindMatch(mat, pat, used, x+1, y, nrow, ncol, level+1) :   false) ||
         (res = y > 0 ? FindMatch(mat, pat, used, x, y-1, nrow, ncol, level+1) :   false) ||
         (res = y < (ncol - 1)  ? FindMatch(mat, pat, used, x, y+1, nrow, ncol, level+1) :   false) ||
          (res = x < (nrow - 1)&&(y < ncol -1)?FindMatch(mat, pat, used, x+1, y+1, nrow, ncol, level+1) :  false) ||
         (res = x < (nrow - 1) && y > 0  ?  FindMatch(mat, pat, used, x+1, y-1, nrow, ncol, level+1) : false) ||
         (res = x > 0 && y < (ncol - 1) ?    FindMatch(mat, pat, used, x-1, y+1, nrow, ncol, level+1) : false) ||
         (res = x > 0 && y > 0 ? FindMatch(mat, pat, used, x-1, y-1, nrow, ncol, level+1) : false);
      used[x][y] = 0;           //marking this cell as unused
      return res;
   }
   else  return false;
}
```

Problem-15 Given two strings str1 and *str*2, 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: *str*1 = "AB", *str*2 = "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 *str*1 be *m* and the length of *str*2 be *n*. Let us assume that all characters in *str*1 and *str*2 are different. Let Count(*m*,*n*) be the count of all interleaved strings in such strings. The value of Count(*m*,*n*) can be written as following.

```
Count(m, n) = Count(m-1, n) + Count(m, n-1)
Count(1, 0) = 1 and Count(1, 0) = 1
```

To print all interleavings, we can first fix the first character of strl[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].

```
void PrintInterleavings(char *str1, char *str2, char *iStr, int m, int n, int i){
   // Base case: If all characters of str1 & str2 have been included in output string,
   // then print the output string
   if ( m==0 && n ==0 )
      printf("%s\n", iStr) ;
   // If some characters of str1 are left to be included, then include the
   // first character from the remaining characters and recur for rest
   if ( m != 0 )   {
      iStr[i] = str1[0];
      PrintInterleavings(str1 + 1, str2, iStr, m-1, n, i+1);
   }
   // If some characters of str2 are left to be included, then include the
   // first character from the remaining characters and recur for rest
   if ( n != 0 ) {
      iStr[i] = str2[0];
      PrintInterleavings(str1, str2+1, iStr, m, n-1, i+1);
   }
}
// Allocates memory for output string and uses PrintInterleavings() for printing all interleaving's
void Print(char *str1, char *str2, int m, int n){
  // allocate memory for the output string
  char *iStr= (char*)malloc((m+n+1)*sizeof(char));
        // Set the terminator for the output string
  iStr[m+n] = '\0';
  // print all interleaving's using PrintInterleavings()
  PrintInterleavings(str1, str2, iStr, m, n, 0);
  free(iStr);
}
```

Problem-16 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;''[1] 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-17 Write a method to replace all spaces in a string with '%20'. Assume string has sufficient space at end of string to hold additional characters.

Solution: Find the number of spaces. Then, starting from end (assuming string has enough space), replace the characters. Starting from end reduces the overwrites.

```
void encodeSpaceWithString(char* A){
    char *space = "%20";
    int stringLength = strlen(A);
    if(stringLength ==0){
        return;
    }
    int i, numberOfSpaces = 0;
    for(i = 0; i < stringLength; i++){
        if(A[i] == ' ' || A[i] == '\t'){
            numberOfSpaces ++;
        }
    }
    if(!numberOfSpaces)
        return;
    int newLength = len + numberOfSpaces * 2;
    A[newLength] = '\0';
    for(i = stringLength-1; i >= 0; i--){
        if(A[i] == ' ' || A[i] == '\t'){
            A[newLength--] = '0';
            A[newLength--] = '2';
            A[newLength--] = '%';
        }
        else{
            A[newLength--] = A[i];
        }
    }
}
```

Time Complexity: O(*n*). Space Complexity: O(1). Here, we do not have to worry about the space needed for extra characters.

Problem-18 **Running length encoding:** Write an algorithm to compress the given string by using the count of repeated characters and if new corn-pressed string length is not smaller than the original string then return the original string.

Solution:

With extra space of O(2):

```
string CompressString(string inputStr){
        char last = inputStr.at(0);
        int size = 0, count = 1;
        char temp[2];
        string str;
        for (int i = 1; i < inputStr.length(); i++){
                if(last == inputStr.at(i))
                        count ++;
                else{
                        itoa(count, temp, 10);
                        str += last;
                        str += temp;
                        last = inputStr.at(i);
                        count = 1;
                }
        }
        str = str + last + temp;
        // If the compresed string size is greater than input string, return input string
        if(str.length() >= inputStr.length())
                return inputStr;
        else return str;
}
```

Time Complexity: O(*n*). Space Complexity: O(1), but it uses a temporary array of size two.

Without extra space (*inplace*):

```
char CompressString(char *inputStr, char currentChar, int lengthIndex, int& countChar, int& index){
	if(lengthIndex == -1)
		return currentChar;
	char lastChar = CompressString(inputStr, inputStr[lengthIndex], lengthIndex-1, countChar, index);
	if(lastChar == currentChar)
		countChar++;
	else {
		inputStr[index++] = lastChar;
		for(int i =0; i< NumToString(countChar).length(); i++)
			inputStr[index++] = NumToString(countChar).at(i);
		countChar = 1;
	}
	return currentChar;
}
```

Time Complexity: O(*n*). Space Complexity: O(1).