String

1. Return maximum occurring character in the input string

Write an efficient C function to return maximum occurring character in the input string e.g., if input string is “test string” then function should return ‘t’.

Algorithm:

```
Input string = “test”
1: Construct character count array from the input string.
  count['e'] = 1
  count['s'] = 1
  count['t'] = 2

2: Return the index of maximum value in count array (returns ‘t’).
```

Implementation:

```c
#include <stdio.h>
#include <stdlib.h>
#define NO_OF_CHARS 256

int *getCharCountArray(char *);
char getIndexOfMax(int *, int);

/* Returns the maximum occurring character in
 the input string */
char getMaxOccuringChar(char *str)
{
  int *count = getCharCountArray(str);
  return getIndexOfMax(count, NO_OF_CHARS);
}

/* Returns an array of size 256 containg count
   of characters in the passed char array */
int *getCharCountArray(char *str)
{
  int *count = (int *)calloc(NO_OF_CHARS, sizeof(int));
  int i;

  for (i = 0; *(str+i);  i++)
    count[*(str+i)]++;

  return count;
}

char getIndexOfMax(int ar[], int ar_size)
{
  int i;
  int max_index = 0;

  for(i = 1; i < ar_size; i++)
    if(ar[i] > ar[max_index])
      max_index = i;

  /* free memory allocated to count */
  free(ar);
  ar = NULL;

  return max_index;
}

int main()
{
  char str[] = "sample string";
  printf("%c", getMaxOccuringChar(str));

  getchar();
  return 0;
}
```

Time Complexity: O(n)

Notes:
If more than one character have the same and maximum count then function returns only the first character in alphabetical order. For example if input string is “test sample” then function will return only ‘e’.

2. Remove all duplicates from the input string.

Below are the different methods to remove duplicates in a string.

METHOD 1 (Use Sorting)

Algorithm:

```
1) Sort the elements.
2) Now in a loop, remove duplicates by comparing the
    current character with previous character.
3)  Remove extra characters at the end of the resultant string.
```

Example:

```
Input string:  geeksforgeeks
1) Sort the characters
   eeeefggkkosss
2) Remove duplicates
    efgkosgkkosss
3) Remove extra characters
     efgkos
```

Note that, this method doesn't keep the original order of the input string. For example, if we are to remove duplicates for geeksforgeeks and keep the order of characters same, then output should be geksfor, but above function returns efgkos. We can modify this method by storing the original order. METHOD 2 keeps the order same.

Implementation:

```
# include <stdio.h>
# include <stdlib.h>

/* Function to remove duplicates in a sorted array */
char *removeDupsSorted(char *str);

/* Utitlity function to sort array A[] */
void quickSort(char A[], int si, int ei);

/* Function removes duplicate characters from the string
   This function work in-place and fills null characters
   in the extra space left */
char *removeDups(char *str)
{
  int len = strlen(str);
  quickSort(str, 0, len-1);
  return removeDupsSorted(str);
```

```
}

/* Function to remove duplicates in a sorted array */
char *removeDupsSorted(char *str)
{
  int res_ind = 1, ip_ind = 1;

  /* In place removal of duplicate characters*/
  while(*(str + ip_ind))
  {
    if(*(str + ip_ind) != *(str + ip_ind - 1))
    {
      *(str + res_ind) = *(str + ip_ind);
      res_ind++;
    }
    ip_ind++;
  }

  /* After above step string is stringiittg.
     Removing extra iittg after string*/
  *(str + res_ind) = '\0';

  return str;
}

/* Driver program to test removeDups */
int main()
{
  char str[] = "eeeefggkkosss";
  printf("%s", removeDups(str));
  getchar();
  return 0;
}

/* FOLLOWING FUNCTIONS ARE ONLY FOR SORTING
    PURPOSE */
void exchange(char *a, char *b)
{
  char temp;
  temp = *a;
  *a   = *b;
  *b   = temp;
}

int partition(char A[], int si, int ei)
{
  char x = A[ei];
  int i = (si - 1);
  int j;

  for (j = si; j <= ei - 1; j++)
  {
    if(A[j] <= x)
    {
      i++;
      exchange(&A[i], &A[j]);
    }
  }
  exchange (&A[i + 1], &A[ei]);
  return (i + 1);
}

/* Implementation of Quick Sort
```

```
/* Implementation of Quick Sort
A[] --> Array to be sorted
si  --> Starting index
ei  --> Ending index
*/
void quickSort(char A[], int si, int ei)
{
  int pi;    /* Partitioning index */
  if(si < ei)
  {
    pi = partition(A, si, ei);
    quickSort(A, si, pi - 1);
    quickSort(A, pi + 1, ei);
  }
}
```

Time Complexity: O(nlogn) If we use some nlogn sorting algorithm instead of quicksort.

METHOD 2 (Use Hashing)

Algorithm:

```
1: Initialize:
    str   =  "test string" /* input string */
    ip_ind =  0          /* index to  keep track of location of next
                            character in input string */
    res_ind  =  0         /* index to  keep track of location of
                           next character in the resultant string */
    bin_hash[0..255] = {0,0, ….} /* Binary hash to see if character is
                                    already processed or not */
2: Do following for each character *(str + ip_ind) in input string:
              (a) if bin_hash is not set for *(str + ip_ind) then
                   // if program sees the character *(str + ip_ind) first time
                         (i)  Set bin_hash for *(str + ip_ind)
                         (ii)  Move *(str  + ip_ind) to the resultant string.
                              This is done in-place.
                         (iii) res_ind++
              (b) ip_ind++
  /* String obtained after this step is "te sringng" */
3: Remove extra characters at the end of the resultant string.
  /*  String obtained after this step is "te sring" */
```

Implementation:

```c
# include <stdio.h>
# include <stdlib.h>
# define NO_OF_CHARS 256
# define bool int

/* Function removes duplicate characters from the string
   This function work in-place and fills null characters
   in the extra space left */
char *removeDups(char *str)
{
  bool bin_hash[NO_OF_CHARS] = {0};
  int ip_ind = 0, res_ind = 0;
  char temp;

  /* In place removal of duplicate characters*/
  while(*(str + ip_ind))
  {
    temp = *(str + ip_ind);
    if(bin_hash[temp] == 0)
    {
        bin_hash[temp] = 1;
        *(str + res_ind) = *(str + ip_ind);
        res_ind++;
    }
    ip_ind++;
  }

  /* After above step string is stringiittg.
     Removing extra iittg after string*/
  *(str+res_ind) = '\0';

  return str;
}

/* Driver program to test removeDups */
int main()
{
    char str[] = "geeksforgeeks";
    printf("%s", removeDups(str));
    getchar();
    return 0;
}
```

Time Complexity: O(n)

NOTES:

* It is assumed that number of possible characters in input string are 256. NO_OF_CHARS should be changed accordingly.
* calloc is used instead of malloc for memory allocations of counting array (count) to initialize allocated memory to '\0'. malloc() followed by memset() could also be used.
* Above algorithm also works for an integer array inputs if range of the integers in array is given. Example problem is to find maximum occurring number in an input array given that the input array contain integers only between 1000 to 1100

3. Print all the duplicates in the input string.

Write an efficient C program to print all the duplicates and their counts in the input string

Algorithm: Let input string be “geeksforgeeks”
1: Construct character count array from the input string.

count[‘e’] = 4
count[‘g’] = 2
count[‘k’] = 2
……

2: Print all the indexes from the constructed array which have value greater than 0.

Solution

```
# include <stdio.h>
# include <stdlib.h>
# define NO_OF_CHARS 256

/* Fills count array with frequency of characters */
void fillCharCounts(char *str, int *count)
{
   int i;
   for (i = 0; *(str+i);  i++)
      count[*(str+i)]++;
}
```

```
/* Print duplicates present in the passed string */
void printDups(char *str)
{
  // Create an array of size 256 and fill count of every character in
  int *count = (int *)calloc(NO_OF_CHARS, sizeof(int));
  fillCharCounts(str, count);

  // Print characters having count more than 0
  int i;
  for (i = 0; i < NO_OF_CHARS; i++)
    if(count[i] > 1)
        printf("%c,  count = %d \n", i,  count[i]);

  free(count);
}
```

```
/* Driver program to test to pront printDups*/
int main()
{
    char str[] = "test string";
    printDups(str);
    getchar();
    return 0;
}
```

Output:

```
s,  count = 2
t,  count = 3
```

Time Complexity: O(n)

4. Remove characters from the first string which are present in the second string

Write an efficient C function that takes two strings as arguments and removes the characters from first string which are present in second string (mask string).

Algorithm: Let first input string be"test string" and the string which has characters to be removed from first string be "mask"
1: Initialize:
res_ind = 0 /* index to keep track of processing of each character in i/p string */
ip_ind = 0 /* index to keep track of processing of each character in the resultant string */

2: Construct count array from mask_str. Count array would be:
(We can use Boolean array here instead of int count array because we don't need count, we need to know only if character is present in mask string)
count['a'] = 1
count['k'] = 1
count['m'] = 1
count['s'] = 1

3: Process each character of the input string and if count of that character is 0 then only add the character to the resultant string.
str = "tet tringng" // 's' has been removed because 's' was present in mask_str but we we have got two extra characters "ng"
ip_ind = 11
res_ind = 9

4: Put a '\0′ at the end of the string?

Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#define NO_OF_CHARS 256

/* Returns an array of size 256 containg count
   of characters in the passed char array */
int *getCharCountArray(char *str)
{
   int *count = (int *)calloc(sizeof(int), NO_OF_CHARS);
   int i;
   for (i = 0; *(str+i);  i++)
      count[*(str+i)]++;
   return count;
}

/* removeDirtyChars takes two string as arguments: First
string (str)  is the one from where function removes dirty
characters. Second  string is the string which contain all
dirty characters which need to be removed  from first string */
char *removeDirtyChars(char *str, char *mask_str)
{
  int *count  = getCharCountArray(mask_str);
  int ip_ind  = 0, res_ind = 0;
  char temp;
  while(*(str + ip_ind))
  {
    temp = *(str + ip_ind);
    if(count[temp] == 0)
    {
      *(str + res_ind) = *(str + ip_ind);
      res_ind++;
    }
    ip_ind++;
  }

  /* After above step string is ngring.
    Removing extra "iittg" after string*/
  *(str+res_ind) = '\0';

  return str;
}

/* Driver program to test getCharCountArray*/
int main()
{
    char mask_str[]  = "mask";
    char str[]       = "geeksforgeeks";
    printf("%s", removeDirtyChars(str, mask_str));
    getchar();
    return 0;
}
```

Time Complexity: O(m+n) Where m is the length of mask string and n is the length of the input string.

5. A Program to check if strings are rotations of each other or not

Given a string s1 and a string s2, write a snippet to say whether s2 is a rotation of s1 using only one call to strstr routine?
(eg given s1 = ABCD and s2 = CDAB, return true, given s1 = ABCD, and s2 = ACBD , return false)

<?php the_content('Read on...'); ?>

Algorithm: areRotations(str1, str2)

```
1. Create a temp string and store concatenation of str1 to
   str1 in temp.
                      temp = str1.str1
2. If str2 is a substring of temp then str1 and str2 are
   rotations of each other.

Example:
                 str1 = "ABACD"
                 str2 = "CDABA"

 temp = str1.str1 = "ABACDABACD"
 Since str2 is a substring of temp, str1 and str2 are
 rotations of each other.
```

Implementation:

```
# include <stdio.h>
# include <string.h>
# include <stdlib.h>

/* Function checks if passed strings (str1 and str2)
   are rotations of each other */
int areRotations(char *str1, char *str2)
{
  int size1   = strlen(str1);
  int size2   = strlen(str2);
  char *temp;
  void *ptr;

  /* Check if sizes of two strings are same */
  if (size1 != size2)
     return 0;

  /* Create a temp string with value str1.str1 */
  temp   = (char *)malloc(sizeof(char)*(size1*2 + 1));
  temp[0] = '\0';
  strcat(temp, str1);
  strcat(temp, str1);

  /* Now check if str2 is a substring of temp */
  ptr = strstr(temp, str2);

  free(temp); // Free dynamically allocated memory

  /* strstr returns NULL if the second string is NOT a
    substring of first string */
  if (ptr != NULL)
    return 1;
  else
    return 0;
}

/* Driver program to test areRotations */
int main()
{
    char *str1 = "AACD";
    char *str2 = "ACDA";

    if (areRotations(str1, str2))
       printf("Strings are rotations of each other");
    else
       printf("Strings are not rotations of each other");

    getchar();
    return 0;
}
```

Output:

```
Strings are rotations of each other
```

Library Functions Used:

strstr:

strstr finds a sub-string within a string.

Prototype: char * strstr(const char *s1, const char *s2);

See

http://www.lix.polytechnique.fr/Labo/Leo.Liberti/public/computing/prog/c/C/MAN/strstr.htm for more details

strcat:
strncat concatenate two strings
Prototype: char *strcat(char *dest, const char *src);
See
http://www.lix.polytechnique.fr/Labo/Leo.Liberti/public/computing/prog/c/C/MAN/strcat.htm for more details

Time Complexity: Time complexity of this problem depends on the implementation of strstr function.
If implementation of strstr is done using KMP matcher then complexity of the above program is (-)(n1 + n2) where n1 and n2 are lengths of strings. KMP matcher takes (-)(n) time to find a substrng in a string of length n where length of substring is assumed to be smaller than the string.

6. Print reverse of a string using recursion

Write a recursive C function to print reverse of a given string.

Program:

```
# include <stdio.h>

/* Function to print reverse of the passed string */
void reverse(char *str)
{
   if(*str)
   {
       reverse(str+1);
       printf("%c", *str);
   }
}

/* Driver program to test above function */
int main()
{
   char a[] = "Geeks for Geeks";
   reverse(a);
   getchar();
   return 0;
}
```

Explanation: Recursive function (reverse) takes string pointer (str) as input and calls itself with next location to passed pointer (str+1). Recursion continues this way, when

pointer reaches '\0', all functions accumulated in stack print char at passed location (str) and return one by one.

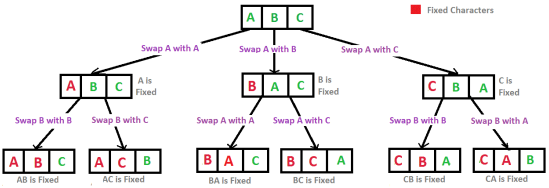
Time Complexity: O(n)

7. Write a C program to print all permutations of a given string

A permutation, also called an "arrangement number" or "order," is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself. A string of length n has n! permutation.
Source: Mathword(http://mathworld.wolfram.com/Permutation.html)

Below are the permutations of string ABC.
ABC, ACB, BAC, BCA, CAB, CBA

Here is a solution using backtracking.



Recursion Tree for Permutations of String "ABC"

```
# include <stdio.h>

/* Function to swap values at two pointers */
void swap (char *x, char *y)
{
    char temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

/* Function to print permutations of string
   This function takes three parameters:
   1. String
   2. Starting index of the string
   3. Ending index of the string. */
void permute(char *a, int i, int n)
{
   int j;
   if (i == n)
     printf("%s\n", a);
   else
   {
        for (j = i; j <= n; j++)
       {
          swap((a+i), (a+j));
          permute(a, i+1, n);
          swap((a+i), (a+j)); //backtrack
       }
   }
}

/* Driver program to test above functions */
int main()
{
   char a[] = "ABC";
   permute(a, 0, 2);
   getchar();
   return 0;
}
```

Output:

```
ABC
ACB
BAC
BCA
CBA
CAB
```

Algorithm Paradigm: Backtracking
Time Complexity: O(n*n!)

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

8. Divide a string in N equal parts

Difficulty Level: Rookie

Question:
Write a program to print N equal parts of a given string.

Solution:
1) Get the size of the string using string function strlen() (present in string.h)
2) Get size of a part.

```
    part_size = string_length/n
```

3) Loop through the input string. In loop, if index becomes multiple of part_size then put a part separator("\n")

Implementation:

```
#include<stdio.h>
#include<string.h>

/* Function to print n equal parts of str*/
void divideString(char *str, int n)
{
   int str_size = strlen(str);
   int i;
   int part_size;

   /*Check if string can be divided in n equal parts */
   if(str_size%n != 0)
   {
     printf("Invalid Input: String size is not divisible by n");
     return;
   }

   /* Calculate the size of parts to find the division points*/
   part_size = str_size/n;
   for(i = 0; i< str_size; i++)
   {
     if(i%part_size == 0)
        printf("\n"); /* newline separator for different parts */
     printf("%c", str[i]);
   }
}

int main()
{
    /*length od string is 28*/
    char *str = "a_simple_divide_string_quest";

    /*Print 4 equal parts of the string */
    divideString(str, 4);

    getchar();
    return 0;
}
```

In above solution, we are simply printing the N equal parts of the string. If we want individual parts to be stored then we need to allocate part_size + 1 memory for all N parts (1 extra for string termination character '\0'), and store the addresses of the parts in an array of character pointers.

Asked by Jason.

9. Given a string, find its first non-repeating character

Given a string, find the first non-repeating character in it. For example, if the input string is "GeeksforGeeks", then output should be 'f' and if input string is "GeeksQuiz", then output should be 'G'.

We can use string characters as index and build a count array. Following is the algorithm.

```
1) Scan the string from left to right and construct the count array.
2) Again, scan the string from left to right and check for count of each
 character, if you find an element who's count is 1, return it.
```

Example:

```
Input string: str = geeksforgeeks
1: Construct character count array from the input string.
   ....
  count['e'] = 4
  count['f'] = 1
  count['g'] = 2
  count['k'] = 2
  ……
2: Get the first character who's count is 1 ('f').
```

Implementation:

```
#include<stdlib.h>
#include<stdio.h>
#define NO_OF_CHARS 256

/* Returns an array of size 256 containg count
   of characters in the passed char array */
int *getCharCountArray(char *str)
{
   int *count = (int *)calloc(sizeof(int), NO_OF_CHARS);
   int i;
   for (i = 0; *(str+i);  i++)
      count[*(str+i)]++;
   return count;
}

/* The function returns index of first non-repeating
   character in a string. If all characters are repeating
   then returns -1 */
int firstNonRepeating(char *str)
{
  int *count = getCharCountArray(str);
  int index = -1, i;

  for (i = 0; *(str+i);  i++)
  {
    if (count[*(str+i)] == 1)
    {
      index = i;
      break;
    }
  }

  free(count); // To avoid memory leak
  return index;
}

/* Driver program to test above function */
int main()
{
  char str[] = "geeksforgeeks";
  int index =  firstNonRepeating(str);
  if (index == -1)
    printf("Either all characters are repeating or string is empty");
  else
   printf("First non-repeating character is %c", str[index]);
  getchar();
  return 0;
}
```

Output:

```
First non-repeating character is f
```

Can we do it by traversing the string only once?

The above approach takes O(n) time, but in practice it can be improved. The first part of the algorithm runs through the string to construct the count array (in O(n) time). This is reasonable. But the second part about running through the string again just to find the first non-repeater is not good in practice. In real situations, your string is expected to be

much larger than your alphabet. Take DNA sequences for example: they could be millions of letters long with an alphabet of just 4 letters. What happens if the non-repeater is at the end of the string? Then we would have to scan for a long time (again).
We can augment the count array by storing not just counts but also the index of the first time you encountered the character e.g. (3, 26) for 'a' meaning that 'a' got counted 3 times and the first time it was seen is at position 26. So when it comes to finding the first non-repeater, we just have to scan the count array, instead of the string. Thanks to Ben for suggesting this approach.

Following is C implementation of the extended approach that traverses the input string only once.

```
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#define NO_OF_CHARS 256

// Structure to store count of a character and index of the first
// occurrence in the input string
struct countIndex {
   int count;
   int index;
};

/* Returns an array of above structure type. The size of
   array is NO_OF_CHARS */
struct countIndex *getCharCountArray(char *str)
{
   struct countIndex *count =
        (struct countIndex *)calloc(sizeof(countIndex), NO_OF_CHARS);
   int i;
   for (i = 0; *(str+i);  i++)
   {
      (count[*(str+i)].count)++;

      // If it's first occurrence, then store the index
      if (count[*(str+i)].count == 1)
         count[*(str+i)].index = i;
   }
   return count;
}

/* The function returns index of the first non-repeating
    character in a string. If all characters are repeating
    then reurns INT_MAX */
int firstNonRepeating(char *str)
{
  struct countIndex *count = getCharCountArray(str);
  int result = INT_MAX, i;

  for (i = 0; i < NO_OF_CHARS;  i++)
  {
    // If this character occurs only once and appears
    // before the current result, then update the result
    if (count[i].count == 1 && result > count[i].index)
       result = count[i].index;
  }

  free(count); // To avoid memory leak
```

```
  return result;
}

/* Driver program to test above function */
int main()
{
  char str[] = "geeksforgeeks";
  int index =  firstNonRepeating(str);
  if (index == INT_MAX)
    printf("Either all characters are repeating or string is empty");
  else
   printf("First non-repeating character is %c", str[index]);
  getchar();
  return 0;
}
```

Output:

```
First non-repeating character is f
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

10. Print list items containing all characters of a given word

There is a list of items. Given a specific word, e.g., “sun”, print out all the items in list which contain all the characters of “sum”

For example if the given word is “sun” and the items are “sunday”, “geeksforgeeks”, “utensils”, “”just” and “sss”, then the program should print “sunday” and “utensils”.

Algorithm: Thanks to geek4u for suggesting this algorithm.

```
1) Initialize a binary map:
        map[256] = {0, 0, ..}
2) Set values in map[] for the given word "sun"
        map['s'] = 1,  map['u'] = 1,  map['n'] = 1
3) Store length of the word "sun":
        len = 3 for "sun"
4) Pick words (or items)one by one from the list
    a) set count = 0;
    b) For each character ch of the picked word
         if(map['ch'] is set)
            increment count and unset map['ch']
    c) If count becomes equal to len (3 for "sun"),
           print the currently picked word.
```

```
    d) Set values in map[] for next list item
          map['s'] = 1,  map['u'] = 1,  map['n'] = 1
```

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# define NO_OF_CHARS 256

/* prints list items having all caharacters of word */
void print(char *list[], char *word, int list_size)
{
  /*Since calloc is used, map[] is initialized as 0 */
  int *map = (int *)calloc(sizeof(int), NO_OF_CHARS);
  int i, j, count, word_size;

  /*Set the values in map */
  for (i = 0; *(word+i); i++)
      map[*(word + i)] = 1;

  /* Get the length of given word */
  word_size = strlen(word);

  /* Check each item of list if has all characters
     of word*/
  for (i = 0; i < list_size; i++)
  {
    for(j = 0, count = 0; *(list[i] + j); j++)
    {
      if(map[*(list[i] + j)])
      {
         count++;

          /* unset the bit so that strings like
            sss not printed*/
         map[*(list[i] + j)] = 0;
      }
    }
    if(count == word_size)
      printf("\n %s", list[i]);

    /*Set the values in map for next item*/
    for (j = 0; *(word+j); j++)
      map[*(word + j)] = 1;
  }
}

/* Driver program to test to pront printDups*/
int main()
{
  char str[] = "sun";
  char *list[] = {"geeksforgeeks", "unsorted", "sunday", "just", "sss"
  print(list, str, 5);
  getchar();
  return 0;
}
```

Asked by Joey

Time Complexity: O(n + m) where n is total number of characters in the list of items. And m = (number of items in list) * (number of characters in the given word)

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem

11. Reverse words in a given string

Example: Let the input string be “i like this program very much”. The function should change the string to “much very program this like i”

Algorithm:

```
1) Reverse the individual words, we get the below string.
     "i ekil siht margorp yrev hcum"
2) Reverse the whole string from start to end and you get the desired output.
     "much very program this like i"
```

```
#include<stdio.h>

/* function prototype for utility function to
  reverse a string from begin to end  */
void reverse(char *begin, char *end);

/*Function to reverse words*/
void reverseWords(char *s)
{
  char *word_begin = s;
  char *temp = s; /* temp is for word boundry */

  /*STEP 1 of the above algorithm */
  while( *temp )
  {
    temp++;
    if (*temp == '\0')
    {
      reverse(word_begin, temp-1);
    }
    else if(*temp == ' ')
    {
      reverse(word_begin, temp-1);
      word_begin = temp+1;
    }
  } /* End of while */

   /*STEP 2 of the above algorithm */
  reverse(s, temp-1);
}

/* UTILITY FUNCTIONS */
/*Function to reverse any sequence starting with pointer
  begin and ending with pointer end  */
void reverse(char *begin, char *end)
{
  char temp;
  while (begin < end)
  {
    temp = *begin;
    *begin++ = *end;
    *end-- = temp;
  }
}

/* Driver function to test above functions */
int main()
{
  char s[] = "i like this program very much";
  char *temp = s;
  reverseWords(s);
  printf("%s", s);
  getchar();
  return 0;
}
```

The above code doesn't handle the cases when the string starts with space. The following version handles this specific case and doesn't make unnecessary calls to reverse function in the case of multiple space in between. Thanks to rka143 for providing this version.

```
void reverseWords(char *s)
{
    char *word_begin = NULL;
    char *temp = s; /* temp is for word boundry */

    /*STEP 1 of the above algorithm */
    while( *temp )
    {
        /*This condition is to make sure that the string start with
          valid character (not space) only*/
        if (( word_begin == NULL ) && (*temp != ' ') )
        {
            word_begin=temp;
        }
        if(word_begin && ((*(temp+1) == ' ') || (*(temp+1) == '\0')))
        {
            reverse(word_begin, temp);
            word_begin = NULL;
        }
        temp++;
    } /* End of while */

    /*STEP 2 of the above algorithm */
    reverse(s, temp-1);
}
```

Time Complexity: O(n)

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem.

12. Run Length Encoding

Given an input string, write a function that returns the Run Length Encoded string for the input string.

For example, if the input string is “wwwwaaadexxxxxx”, then the function should return “w4a3d1e1x6″.

Algorithm:
a) Pick the first character from source string.
b) Append the picked character to the destination string.
c) Count the number of subsequent occurrences of the picked character and append the count to destination string.
d) Pick the next character and repeat steps b) c) and d) if end of string is NOT reached.

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define MAX_RLEN 50

/* Returns the Run Length Encoded string for the
   source string src */
char *encode(char *src)
{
  int rLen;
  char count[MAX_RLEN];
  int len = strlen(src);

  /* If all characters in the source string are different,
    then size of destination string would be twice of input string.
    For example if the src is "abcd", then dest would be "a1b1c1d1"
    For other inputs, size would be less than twice.  */
  char *dest = (char *)malloc(sizeof(char)*(len*2 + 1));

  int i, j = 0, k;

  /* traverse the input string one by one */
  for(i = 0; i < len; i++)
  {

    /* Copy the first occurrence of the new character */
    dest[j++] = src[i];

    /* Count the number of occurrences of the new character */
    rLen = 1;
    while(i + 1 < len && src[i] == src[i+1])
    {
      rLen++;
      i++;
    }

    /* Store rLen in a character array count[] */
    sprintf(count, "%d", rLen);

    /* Copy the count[] to destination */
    for(k = 0; *(count+k); k++, j++)
    {
      dest[j] = count[k];
    }
  }

  /*terminate the destination string */
  dest[j] = '\0';
  return dest;
}

/*driver program to test above function */
int main()
{
  char str[] = "geeksforgeeks";
  char *res = encode(str);
  printf("%s", res);
  getchar();
}
```

Time Complexity: O(n)

References:
http://en.wikipedia.org/wiki/Run-length_encoding

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

13. Find the smallest window in a string containing all characters of another string

Given two strings string1 and string2, find the smallest substring in string1 containing all characters of string2 efficiently.

For Example:

Input string1: “this is a test string”
Input string2: “tist”
Output string: “t stri”

Method 1 (Brute force solution)
a) Generate all substrings of string1 (“this is a test string”)
b) For each substring, check whether the substring contains all characters of string2 (“tist”)
c) Finally print the smallest substring containing all characters of string2.

Method 2 (Efficient Solution)
1) Build a count array count[] for string 2. The count array stores counts of characters.
count[‘i’] = 1
count[‘t’] = 2
count[‘s’] = 1

2) Scan the string1 from left to right until we find all the characters of string2. To check if all the characters are there, use count[] built in step 1. So we have substring “this is a t” containing all characters of string2. Note that the first and last characters of the substring must be present in string2. Store the length of this substring as min_len.

3) Now move forward in string1 and keep adding characters to the substring “this is a t”. Whenever a character is added, check if the added character matches the left most character of substring. If matches, then add the new character to the right side of substring and remove the leftmost character and all other extra characters after left most character. After removing the extra characters, get the length of this substring and compare with min_len and update min_len accordingly.

Basically we add ‘e’ to the substring “this is a t”, then add ‘s’ and then’t’. ‘t’ matches the

left most character, so remove ‘t’ and ‘h’ from the left side of the substring. So our current substring becomes “is a test”. Compare length of it with min_len and update min_len.
Again add characters to current substring “is a test”. So our string becomes “is a test str”. When we add ‘i’, we remove leftmost extra characters, so current substring becomes “t stri”. Again, compare length of it with min_len and update min_len. Finally add ‘n’ and ‘g’. Adding these characters doesn’t decrease min_len, so the smallest window remains “t stri”.

4) Return min_len.

Please write comments if you find the above algorithms incorrect, or find other ways to solve the same problem.

Source: http://geeksforgeeks.org/forum/topic/find-smallest-substring-containing-all-characters-of-a-given-word

14. Searching for Patterns | Set 1 (Naive Pattern Searching)

Given a text *txt[0..n-1]* and a pattern *pat[0..m-1]*, write a function *search(char pat[], char txt[])* that prints all occurrences of *pat[]* in *txt[]*. You may assume that *n* > *m*.

Examples:
1) Input:

```
txt[] =  "THIS IS A TEST TEXT"
pat[] = "TEST"
```

Output:

```
Pattern found at index 10
```

2) Input:

```
txt[] =  "AABAACAADAABAAABAA"
pat[] = "AABA"
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are

used to show the search results.

Naive Pattern Searching:
Slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.

```
#include<stdio.h>
#include<string.h>
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++)
    {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
        {
            if (txt[i+j] != pat[j])
                break;
        }
        if (j == M)  // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
        {
           printf("Pattern found at index %d \n", i);
        }
    }
}

/* Driver program to test above function */
int main()
{
   char *txt = "AABAACAADAABAAABAA";
   char *pat = "AABA";
   search(pat, txt);
   getchar();
   return 0;
}
```

What is the best case?
The best case occurs when the first character of the pattern is not present in text at all.

```
txt[]  = "AABCCAADDEE"
pat[] = "FAA"
```

The number of comparisons in best case is O(n).

What is the worst case ?
The worst case of Naive Pattern Searching occurs in following scenarios.
1) When all characters of the text and pattern are same.

```
txt[] = "AAAAAAAAAAAAAAAAAA"
pat[] = "AAAAA".
```

2) Worst case also occurs when only the last character is different.

```
txt[] = "AAAAAAAAAAAAAAAAB"
pat[] = "AAAAB"
```

Number of comparisons in worst case is O(m*(n-m+1)). Although strings which have repeated characters are not likely to appear in English text, they may well occur in other applications (for example, in binary texts). The KMP matching algorithm improves the worst case to O(n). We will be covering KMP in the next post. Also, we will be writing more posts to cover all pattern searching algorithms and data structures.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

15. Searching for Patterns | Set 2 (KMP Algorithm)

Given a text *txt[0..n-1]* and a pattern *pat[0..m-1]*, write a function *search(char pat[], char txt[])* that prints all occurrences of *pat[]* in *txt[]*. You may assume that *n* > *m*.

Examples:
1) Input:

```
txt[] =  "THIS IS A TEST TEXT"
pat[] = "TEST"
```

Output:

```
 Pattern found at index 10
```

2) Input:

```
txt[] =  "AABAACAADAABAAABAA"
pat[] = "AABA"
```

Output:

```
 Pattern found at index 0
 Pattern found at index 9
 Pattern found at index 13
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed Naive pattern searching algorithm in the previous post. The worst case complexity of Naive algorithm is O(m(n-m+1)). Time complexity of KMP algorithm is O(n) in worst case.

KMP (Knuth Morris Pratt) Pattern Searching

The Naive pattern searching algorithm doesn't work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

```
txt[] = "AAAAAAAAAAAAAAAAAB"
pat[] = "AAAAB"

txt[] = "ABABABCABABABCABABABC"
pat[] =  "ABABAC" (not a worst case, but a bad case for Naive)
```

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text (since they matched the pattern characters prior to the mismatch). We take advantage of this information to avoid matching the characters that we know will anyway match.
KMP algorithm does some preprocessing over the pattern pat[] and constructs an auxiliary array lps[] of size m (same as size of pattern). Here **name lps indicates longest proper prefix which is also suffix.**. For each sub-pattern pat[0…i] where i = 0 to m-1, lps[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

```
lps[i] = the longest proper prefix of pat[0..i]
           which is also a suffix of pat[0..i].
```

Examples:
For the pattern "AABAACAABAA", lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]
For the pattern "ABCDE", lps[] is [0, 0, 0, 0, 0]
For the pattern "AAAAA", lps[] is [0, 1, 2, 3, 4]
For the pattern "AAABAAA", lps[] is [0, 1, 2, 0, 1, 2, 3]
For the pattern "AAACAAAAAC", lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

Searching Algorithm:

Unlike the Naive algo where we slide the pattern by one, we use a value from lps[] to decide the next sliding position. Let us see how we do that. When we compare pat[j] with txt[i] and see a mismatch, we know that characters pat[0..j-1] match with txt[i-j+1…i-1], and we also know that lps[j-1] characters of pat[0…j-1] are both proper prefix and suffix which means we do not need to match these lps[j-1] characters with txt[i-j…i-1] because we know that these characters will anyway match. See KMPSearch() in the below code for details.

Preprocessing Algorithm:

In the preprocessing part, we calculate values in lps[]. To do that, we keep track of the length of the longest prefix suffix value (we use len variable for this purpose) for the previous index. We initialize lps[0] and len as 0. If pat[len] and pat[i] match, we increment

len by 1 and assign the incremented value to lps[i]. If pat[i] and pat[len] do not match and len is not 0, we update len to lps[len-1]. See computeLPSArray () in the below code for details.

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void computeLPSArray(char *pat, int M, int *lps);

void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix suffix values fo
    int *lps = (int *)malloc(sizeof(int)*M);
    int j  = 0;  // index for pat[]

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

    int i = 0;  // index for txt[]
    while (i < N)
    {
      if (pat[j] == txt[i])
      {
        j++;
        i++;
      }

      if (j == M)
      {
        printf("Found pattern at index %d \n", i-j);
        j = lps[j-1];
      }

      // mismatch after j matches
      else if (i < N && pat[j] != txt[i])
      {
        // Do not match lps[0..lps[j-1]] characters,
        // they will match anyway
        if (j != 0)
         j = lps[j-1];
        else
         i = i+1;
      }
    }
    free(lps); // to avoid memory leak
}

void computeLPSArray(char *pat, int M, int *lps)
{
    int len = 0;  // lenght of the previous longest prefix suffix
    int i;

    lps[0] = 0; // lps[0] is always 0
    i = 1;

    // the loop calculates lps[i] for i = 1 to M-1
    while (i < M)
    {
```

```
    {
       if (pat[i] == pat[len])
       {
         len++;
         lps[i] = len;
         i++;
       }
       else // (pat[i] != pat[len])
       {
         if (len != 0)
         {
           // This is tricky. Consider the example AAACAAAA and i = 7.
           len = lps[len-1];

           // Also, note that we do not increment i here
         }
         else // if (len == 0)
         {
           lps[i] = 0;
           i++;
         }
       }
    }
}

// Driver program to test above function
int main()
{
   char *txt = "ABABDABACDABABCABAB";
   char *pat = "ABABCABAB";
   KMPSearch(pat, txt);
   return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

16. Searching for Patterns | Set 3 (Rabin-Karp Algorithm)

Given a text *txt[0..n-1]* and a pattern *pat[0..m-1]*, write a function *search(char pat[], char txt[])* that prints all occurrences of *pat[]* in *txt[]*. You may assume that n > m.

Examples:
1) Input:

```
txt[] =  "THIS IS A TEST TEXT"
pat[] = "TEST"
```

Output:

Pattern found at index 10

2) Input:

```
txt[] =  "AABAACAADAABAAABAA"
pat[] = "AABA"
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

The Naive String Matching algorithm slides the pattern one by one. After each slide, it one by one checks characters at the current shift and if all characters match then prints the match.
Like the Naive Algorithm, Rabin-Karp algorithm also slides the pattern one by one. But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for following strings.

1) Pattern itself.
2) All the substrings of text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which has following property.
Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say *hash(txt[s+1 .. s+m])* must be efficiently computable from *hash(txt[s .. s+m-1])* and *txt[s+m]* i.e., *hash(txt[s+1 .. s+m])= rehash(txt[s+m], hash(txt[s .. s+m-1])* and rehash must be O(1) operation.

The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is numeric value of a string. For example, if all possible characters are from 1 to 10, the numeric value of “122” will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value. Rehashing is done using the following formula.

*hash(txt[s+1 .. s+m]) = d (hash(txt[s .. s+m-1]) – txt[s]*h) + txt[s + m]) mod q*

hash(txt[s .. s+m-1]) : Hash value at shift *s*.
hash(txt[s+1 .. s+m]) : Hash value at next shift (or shift *s*+1)
d: Number of characters in the alphabet
q: A prime number
h: d^(m-1)

```c
/* Following program is a C implementation of the Rabin Karp Algorithm
   given in the CLRS book */

#include<stdio.h>
#include<string.h>

// d is the number of characters in input alphabet
#define d 256

/*  pat  -> pattern
    txt  -> text
    q    -> A prime number
*/
void search(char *pat, char *txt, int q)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0;  // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;

    // The value of h would be "pow(d, M-1)%q"
    for (i = 0; i < M-1; i++)
        h = (h*d)%q;

    // Calculate the hash value of pattern and first window of text
    for (i = 0; i < M; i++)
    {
        p = (d*p + pat[i])%q;
        t = (d*t + txt[i])%q;
    }

    // Slide the pattern over text one by one
    for (i = 0; i <= N - M; i++)
    {

        // Check the hash values of current window of text and pattern
        // If the hash values match then only check for characters on
        if ( p == t )
        {
            /* Check for characters one by one */
            for (j = 0; j < M; j++)
            {
                if (txt[i+j] != pat[j])
                    break;
            }
            if (j == M)  // if p == t and pat[0...M-1] = txt[i, i+1, .
            {
                printf("Pattern found at index %d \n", i);
            }
        }

        // Calculate hash value for next window of text: Remove leadin
        // add trailing digit
        if ( i < N-M )
        {
            t = (d*(t - txt[i]*h) + txt[i+M])%q;

            // We might get negative value of t, converting it to posi
            if(t < 0)
              t = (t + q);
        }
```

```
        j
    }
}

/* Driver program to test above function */
int main()
{
    char *txt = "GEEKS FOR GEEKS";
    char *pat = "GEEK";
    int q = 101;  // A prime number
    search(pat, txt, q);
    getchar();
    return 0;
}
```

The average and best case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm). Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the hash values of all the substrings of txt[] match with hash value of pat[]. For example pat[] = “AAA” and txt[] = “AAAAAAA”.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

http://net.pku.edu.cn/~course/cs101/2007/resource/Intro2Algorithm/book6/chap34.htm

http://www.cs.princeton.edu/courses/archive/fall04/cos226/lectures/string.4up.pdf

http://en.wikipedia.org/wiki/Rabin-Karp_string_search_algorithm

Related Posts:
Searching for Patterns | Set 1 (Naive Pattern Searching)
Searching for Patterns | Set 2 (KMP Algorithm)

17. Searching for Patterns | Set 4 (A Naive Pattern Searching Question)

Question: We have discussed Naive String matching algorithm here. Consider a situation where all characters of pattern are different. Can we modify the original Naive String Matching algorithm so that it works better for these types of patterns. If we can, then what are the changes to original algorithm?

Solution: In the original Naive String matching algorithm , we always slide the pattern by 1. When all characters of pattern are different, we can slide the pattern by more than 1. Let us see how can we do this. When a mismatch occurs after j matches, we know that the first character of pattern will not match the j matched characters because all

characters of pattern are different. So we can always slide the pattern by j without missing any valid shifts. Following is the modified code that is optimized for the special patterns.

```
#include<stdio.h>
#include<string.h>

/* A modified Naive Pettern Searching algorithn that is optimized
   for the cases when all characters of pattern are different */
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i = 0;

    while(i <= N - M)
    {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
        {
            if (txt[i+j] != pat[j])
                break;
        }
        if (j == M)  // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
        {
           printf("Pattern found at index %d \n", i);
           i = i + M;
        }
        else if (j == 0)
        {
           i = i + 1;
        }
        else
        {
           i = i + j; // slide the pattern by j
        }
    }
}

/* Driver program to test above function */
int main()
{
   char *txt = "ABCEABCDABCEABCD";
   char *pat = "ABCD";
   search(pat, txt);
   getchar();
   return 0;
}
```

Output:

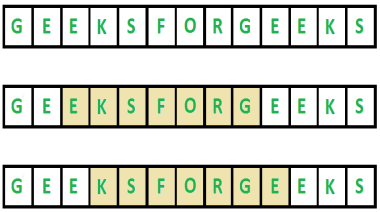
Pattern found at index 4

Pattern found at index 12

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

18. Length of the longest substring without repeating characters

Given a string, find the length of the longest substring without repeating characters. For example, the longest substrings without repeating characters for “ABDEFGABEF” are “BDEFGA” and “DEFGAB”, with length 6. For “BBBB” the longest substring is “B”, with length 1. For “GEEKSFORGEEKS”, there are two longest substrings shown in the below diagrams, with length 7.



The desired time complexity is O(n) where n is the length of the string.

Method 1 (Simple)
We can consider all substrings one by one and check for each substring whether it contains all unique characters or not. There will be n*(n+1)/2 substrings. Whether a substirng contains all unique characters or not can be checked in linear time by scanning it from left to right and keeping a map of visited characters. Time complexity of this solution would be O(n^3).

Method 2 (Linear Time)
Let us talk about the linear time solution now. This solution uses extra space to store the last indexes of already visited characters. The idea is to scan the string from left to right, keep track of the maximum length Non-Repeating Character Substring (NRCS) seen so far. Let the maximum length be max_len. When we traverse the string, we also keep track of length of the current NRCS using cur_len variable. For every new character, we look for it in already processed part of the string (A temp array called visited[] is used for this purpose). If it is not present, then we increase the cur_len by 1. If present, then there are two cases:

a) The previous instance of character is not part of current NRCS (The NRCS which is under process). In this case, we need to simply increase cur_len by 1.
b) If the previous instance is part of the current NRCS, then our current NRCS changes. It becomes the substring staring from the next character of previous instance to currently scanned character. We also need to compare cur_len and max_len, before changing current NRCS (or changing cur_len).

Implementation

```
#include<stdlib.h>
#include<stdio.h>
#define NO_OF_CHARS 256

int min(int a, int b);

int longestUniqueSubsttr(char *str)
{
    int n = strlen(str);
    int cur_len = 1;  // To store the lenght of current substring
    int max_len = 1;  // To store the result
    int prev_index;  // To store the previous index
    int i;
    int *visited = (int *)malloc(sizeof(int)*NO_OF_CHARS);

    /* Initialize the visited array as -1, -1 is used to indicate that
       character has not been visited yet. */
    for (i = 0; i < NO_OF_CHARS;  i++)
        visited[i] = -1;

    /* Mark first character as visited by storing the index of first
       character in visited array. */
    visited[str[0]] = 0;

    /* Start from the second character. First character is already pro
       (cur_len and max_len are initialized as 1, and visited[str[0]]
    for (i = 1; i < n; i++)
    {
        prev_index =  visited[str[i]];

        /* If the currentt character is not present in the already pro
           substring or it is not part of the current NRCS, then do cu
        if (prev_index == -1 || i - cur_len > prev_index)
            cur_len++;

        /* If the current character is present in currently considered
           then update NRCS to start from the next character of previo
        else
        {
            /* Also, when we are changing the NRCS, we should also che
              length of the previous NRCS was greater than max_len or
            if (cur_len > max_len)
                max_len = cur_len;

            cur_len = i - prev_index;
        }

        visited[str[i]] = i; // update the index of current character
    }

    // Compare the length of last NRCS with max_len and update max_len
    if (cur_len > max_len)
        max_len = cur_len;


    free(visited); // free memory allocated for visited

    return max_len;
}

/* A utility function to get the minimum of two integers */
int min(int a, int b)
```

```
{
    return (a>b)?b:a;
}

/* Driver program to test above function */
int main()
{
    char str[] = "ABDEFGABEF";
    printf("The input string is %s \n", str);
    int len =  longestUniqueSubsttr(str);
    printf("The length of the longest non-repeating character substrin

    getchar();
    return 0;
}
```

Output

```
The input string is ABDEFGABEF
The length of the longest non-repeating character substring is 6
```

Time Complexity: O(n + d) where n is length of the input string and d is number of characters in input string alphabet. For example, if string consists of lowercase English characters then value of d is 26.
Auxiliary Space: O(d)
Algorithmic Paradigm: Dynamic Programming

As an exercise, try the modified version of the above problem where you need to print the maximum length NRCS also (the above program only prints length of it).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

19. Print all permutations with repetition of characters

Given a string of length n, print all permutation of the given string. Repetition of characters is allowed. Print these permutations in lexicographically sorted order Examples:

```
Input: AB
Ouput: All permutations of AB with repetition are:
      AA
      AB
      BA
      BB

Input: ABC
```

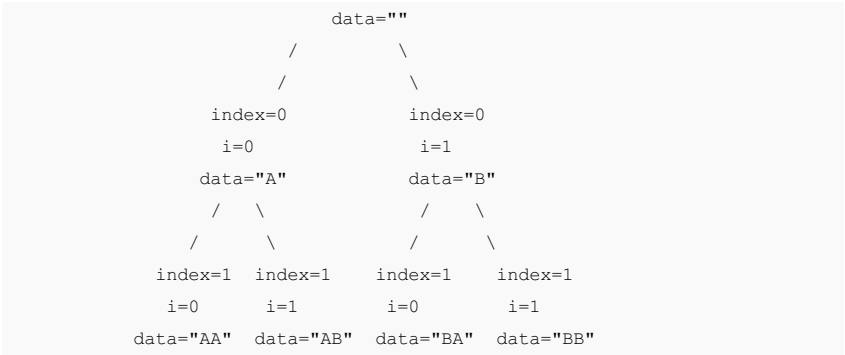
```
Output: All permutations of ABC with repetition are:
       AAA
       AAB
       AAC
       ABA
       ...
       ...
       CCB
       CCC
```

For an input string of size n, there will be n^n permutations with repetition allowed. The idea is to fix the first character at first index and recursively call for other subsequent indexes. Once all permutations starting with the first character are printed, fix the second character at first index. Continue these steps till last character. Thanks to PsychoCoder for providing following C implementation.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

/* Following function is used by the library qsort() function to sort
  array of chars */
int compare (const void * a, const void * b);

/* The main function that recursively prints all repeated permutations
  the given string. It uses data[] to store all permutations one by on
void allLexicographicRecur (char *str, char* data, int last, int index
{
    int i, len = strlen(str);

    // One by one fix all characters at the given index and recur for
    // subsequent indexes
    for ( i=0; i<len; i++ )
    {
        // Fix the ith character at index and if this is not the last
        // then recursively call for higher indexes
        data[index] = str[i] ;

        // If this is the last index then print the string stored in d
        if (index == last)
            printf("%s\n", data);
        else // Recur for higher indexes
            allLexicographicRecur (str, data, last, index+1);
    }
}

/* This function sorts input string, allocate memory for data (needed
  allLexicographicRecur()) and calls allLexicographicRecur() for print
  permutations */
void allLexicographic(char *str)
{
    int len = strlen (str) ;

    // Create a temp array that will be used by allLexicographicRecur(
    char *data = (char *) malloc (sizeof(char) * (len + 1)) ;
    data[len] = '\0';
```

```
    // Sort the input string so that we get all output strings in
    // lexicographically sorted order
    qsort(str, len, sizeof(char), compare);

    // Now print all permutaions
    allLexicographicRecur (str, data, len-1, 0);

    // Free data to avoid memory leak
    free(data);
}

// Needed for library function qsort()
int compare (const void * a, const void * b)
{
    return ( *(char *)a - *(char *)b );
}

// Driver program to test above functions
int main()
{
    char str[] = "ABC";
    printf("All permutations with repetition of %s are: \n", str);
    allLexicographic(str);
    getchar();
    return 0;
}
```

Following is recursion tree for input string “AB”. The purpose of recursion tree is to help in understanding the above implementation as it shows values of different variables.



In the above implementation, it is assumed that all characters of the input string are different. The implementation can be easily modified to handle the repeated characters. We have to add a preprocessing step to find unique characters (before calling allLexicographicRecur()).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

20. Print all interleavings of given two strings

Given two strings str1 and str2, write a function that prints all interleavings of the given two strings. You may assume that all characters in both strings are different

Example:

```
Input: str1 = "AB",  str2 = "CD"
Output:
    ABCD
    ACBD
    ACDB
    CABD
    CADB
    CDAB

Input: str1 = "AB",  str2 = "C"
Output:
    ABC
    ACB
    CAB
```

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’.

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 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(0, 1) = 1
```

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]. Thanks to akash01 for providing following C implementation.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// The main function that recursively prints all interleavings.  The v
// iStr is used to store all interleavings (or output strings) one by
// i is used to pass next available place in iStr
void printIlsRecur (char *str1, char *str2, char *iStr, int m, int n,
{
    // Base case: If all characters of str1 and str2 have been include
    // 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 includ
    // first character from the remaining characters and recur for res
    if ( m != 0 )
    {
        iStr[i] = str1[0];
        printIlsRecur (str1 + 1, str2, iStr, m-1, n, i+1);
    }

    // If some characters of str2 are left to be included, then includ
    // first character from the remaining characters and recur for res
    if ( n != 0 )
    {
        iStr[i] = str2[0];
        printIlsRecur (str1, str2+1, iStr, m, n-1, i+1);
    }
}

// Allocates memory for output string and uses printIlsRecur()
// for printing all interleavings
void printIls (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 interleavings using printIlsRecur()
   printIlsRecur (str1, str2, iStr, m, n, 0);

   // free memory to avoid memory leak
   free(iStr);
}

// Driver program to test above functions
int main()
{
    char *str1 = "AB";
    char *str2 = "CD";
    printIls (str1, str2, strlen(str1), strlen(str2));
    return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

21. Check whether a given string is an interleaving of two other given strings

Given three strings A, B and C. Write a function that checks whether C is an interleaving of A and B.
C is said to be interleaving A and B, if it contains all characters of A and B and order of all characters in individual strings is preserved. See previous post for examples.

Solution:
Pick each character of C one by one and match it with the first character in A. If it doesn't match then match it with first character of B. If it doesn't even match first character of B, then return false. If the character matches with first character of A, then repeat the above process from second character of C, second character of A and first character of B. If first character of C matches with the first character of B (and doesn't match the first character of A), then repeat the above process from the second character of C, first character of A and second character of B. If all characters of C match either with a character of A or a character of B and length of C is sum of lengths of A and B, then C is an interleaving A and B.

Thanks to venkat for suggesting following C implementation.

```
#include<stdio.h>

// Returns true if C is an interleaving of A and B, otherwise
// returns false
bool isInterleaved (char *A, char *B, char *C)
{
    // Iterate through all characters of C.
    while (*C != 0)
    {
        // Match first character of C with first character of A,
        // If matches them move A to next
        if (*A == *C)
            A++;

        // Else Match first character of C with first character of B,
        // If matches them move B to next
        else if (*B == *C)
            B++;

        // If doesn't match with either A or B, then return false
        else
            return false;

        // Move C to next for next iteration
        C++;
    }

    // If A or B still have some characters, then length of C is small
    // than sum of lengths of A and B, so return false
    if (*A || *B)
        return false;

    return true;
}

// Driver program to test above functions
int main()
{
    char *A = "AB";
    char *B = "CD";
    char *C = "ACBG";
    if (isInterleaved(A, B, C) == true)
        printf("%s is interleaved of %s and %s", C, A, B);
    else
        printf("%s is not interleaved of %s and %s", C, A, B);

    return 0;
}
```

Output:

```
ACBG is not interleaved of AB and CD
```

Time Complexity: O(m+n) where m and n are the lengths of strings A and B respectively.

Note that the above approach doesn't work if A and B have some characters in common. For example, if string A = "AAB", string B = "AAC" and string C = "AACAAB", then the above method will return false. We have discussed here an extended solution that handles common characters.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

22. Check whether two strings are anagram of each other

Write a function to check whether two given strings are anagram of each other or not. An anagram of a string is another string that contains same characters, only the order of characters can be different. For example, “abcd” and “dabc” are anagram of each other.

Method 1 (Use Sorting)
1) Sort both strings
2) Compare the sorted strings

```
#include <stdio.h>
#include <string.h>

/* Fucntion prototype for srting a given string using quick sort */
void quickSort(char *arr, int si, int ei);

/* function to check whether two strings are anagram of each other */
bool areAnagram(char *str1, char *str2)
{
    // Get lenghts of both strings
    int n1 = strlen(str1);
    int n2 = strlen(str2);

    // If lenght of both strings is not same, then they cannot
    // be anagram
    if (n1 != n2)
      return false;

    // Sort both strings
    quickSort (str1, 0, n1 - 1);
    quickSort (str2, 0, n2 - 1);

    // Compare sorted strings
    for (int i = 0; i < n1;  i++)
       if (str1[i] != str2[i])
         return false;

    return true;
}

// Following functions (exchange and partition are needed for quickSor
void exchange(char *a, char *b)
{
    char temp;
    temp = *a;
    *a   = *b;
    *b   = temp;
}

int partition(char A[], int si, int ei)
```

```
int partition(char A[], int si, int ei)
{
    char x = A[ei];
    int i = (si - 1);
    int j;

    for (j = si; j <= ei - 1; j++)
    {
        if(A[j] <= x)
        {
            i++;
            exchange(&A[i], &A[j]);
        }
    }
    exchange (&A[i + 1], &A[ei]);
    return (i + 1);
}

/* Implementation of Quick Sort
A[] --> Array to be sorted
si  --> Starting index
ei  --> Ending index
*/
void quickSort(char A[], int si, int ei)
{
    int pi;    /* Partitioning index */
    if(si < ei)
    {
        pi = partition(A, si, ei);
        quickSort(A, si, pi - 1);
        quickSort(A, pi + 1, ei);
    }
}

/* Driver program to test to pront printDups*/
int main()
{
    char str1[] = "test";
    char str2[] = "ttew";
    if ( areAnagram(str1, str2) )
      printf("The two strings are anagram of each other");
    else
      printf("The two strings are not anagram of each other");

    return 0;
}
```

Output:

```
The two strings are not anagram of each other
```

Time Complexity: Time complexity of this method depends upon the sorting technique used. In the above implementation, quickSort is used which may be O(n^2) in worst case. If we use a O(nLogn) sorting algorithm like merge sort, then the complexity becomes O(nLogn)

Method 2 (Count characters)
This method assumes that the set of possible characters in both strings is small. In the following implementation, it is assumed that the characters are stored using 8 bit and

there can be 256 possible characters.
1) Create count arrays of size 256 for both strings. Initialize all values in count arrays as 0.
2) Iterate through every character of both strings and increment the count of character in the corresponding count arrays.
3) Compare count arrays. If both count arrays are same, then return true.

```
# include <stdio.h>
# define NO_OF_CHARS 256

/* function to check whether two strings are anagram of each other */
bool areAnagram(char *str1, char *str2)
{
    // Create two count arrays and initialize all values as 0
    int count1[NO_OF_CHARS] = {0};
    int count2[NO_OF_CHARS] = {0};
    int i;

    // For each character in input strings, increment count in
    // the corresponding count array
    for (i = 0; str1[i] && str2[i];  i++)
    {
        count1[str1[i]]++;
        count2[str2[i]]++;
    }

    // If both strings are of different length. Removing this condition
    // will make the program fail for strings like "aaca" and "aca"
    if (str1[i] || str2[i])
      return false;

    // Compare count arrays
    for (i = 0; i < NO_OF_CHARS; i++)
        if (count1[i] != count2[i])
            return false;

    return true;
}

/* Driver program to test to pront printDups*/
int main()
{
    char str1[] = "geeksforgeeks";
    char str2[] = "forgeeksgeeks";
    if ( areAnagram(str1, str2) )
      printf("The two strings are anagram of each other");
    else
      printf("The two strings are not anagram of each other");

    return 0;
}
```

Output:

```
The two strings are anagram of each other
```

The above implementation can be further to use only one count array instead of two. We can increment the value in count array for characters in str1 and decrement for

characters in str2. Finally, if all count values are 0, then the two strings are anagram of each other. Thanks to Ace for suggesting this optimization.

```
bool areAnagram(char *str1, char *str2)
{
    // Create a count array and initialize all values as 0
    int count[NO_OF_CHARS] = {0};
    int i;

    // For each character in input strings, increment count in
    // the corresponding count array
    for (i = 0; str1[i] && str2[i];  i++)
    {
        count[str1[i]]++;
        count[str2[i]]--;
    }

    // If both strings are of different length. Removing this conditio
    // will make the program fail for strings like "aaca" and "aca"
    if (str1[i] || str2[i])
      return false;

    // See if there is any non-zero value in count array
    for (i = 0; i < NO_OF_CHARS; i++)
        if (count[i])
            return false;
     return true;
}
```

If the possible set of characters contains only English alphabets, then we can reduce the size of arrays to 52 and use *str[i] – 'A'* as an index for count arrays. This will further optimize this method.

Time Complexity: O(n)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

23. Searching for Patterns | Set 5 (Finite Automata)

Given a text *txt[0..n-1]* and a pattern *pat[0..m-1]*, write a function *search(char pat[], char txt[])* that prints all occurrences of *pat[]* in *txt[]*. You may assume that n > m.

Examples:
1) Input:

```
txt[] =  "THIS IS A TEST TEXT"
pat[] = "TEST"
```

Output:

Pattern found at index 10

2) Input:

```
txt[] =  "AABAACAADAABAAABAA"
pat[] = "AABA"
```

Output:

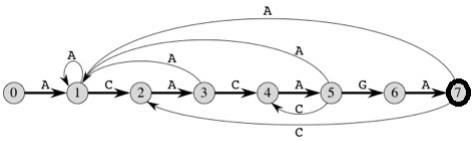
```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed the following algorithms in the previous posts:

Naive Algorithm
KMP Algorithm
Rabin Karp Algorithm

In this post, we will discuss Finite Automata (FA) based pattern searching algorithm. In FA based algorithm, we preprocess the pattern and build a 2D array that represents a Finite Automata. Construction of the FA is the main tricky part of this algorithm. Once the FA is built, the searching is simple. In search, we simply need to start from the first state of the automata and first character of the text. At every step, we consider next character of text, look for the next state in the built FA and move to new state. If we reach final state, then pattern is found in text. Time complexity of the search prcess is O(n).
Before we discuss FA construction, let us take a look at the following FA for pattern ACACAGA.



	character			
state	A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

The abvoe diagrams represent graphical and tabular representations of pattern

ACACAGA.

Number of states in FA will be M+1 where M is length of the pattern. The main thing to construct FA is to get the next state from the current state for every possible character. Given a character x and a state k, we can get the next state by considering the string “pat[0..k-1]x” which is basically concatenation of pattern characters pat[0], pat[1] … pat[k-1] and the character x. The idea is to get length of the longest prefix of the given pattern such that the prefix is also suffix of “pat[0..k-1]x”. The value of length gives us the next state. For example, let us see how to get the next state from current state 5 and character ‘C’ in the above diagram. We need to consider the string, “pat[0..5]C” which is “ACACAC”. The lenght of the longest prefix of the pattern such that the prefix is suffix of “ACACAC”is 4 (“ACAC”). So the next state (from state 5) is 4 for character ‘C’.

In the following code, computeTF() constructs the FA. The time complexity of the computeTF() is O(m^3*NO_OF_CHARS) where m is length of the pattern and NO_OF_CHARS is size of alphabet (total number of possible characters in pattern and text). The implementation tries all possible prefixes starting from the longest possible that can be a suffix of “pat[0..k-1]x”. There are better implementations to construct FA in O(m*NO_OF_CHARS) (Hint: we can use something like lps array construction in KMP algorithm). We have covered the better implementation in our next post on pattern searching.

```
#include<stdio.h>
#include<string.h>
#define NO_OF_CHARS 256

int getNextState(char *pat, int M, int state, int x)
{
    // If the character c is same as next character in pattern,
    // then simply increment state
    if (state < M && x == pat[state])
        return state+1;

    int ns, i;  // ns stores the result which is next state

    // ns finally contains the longest prefix which is also suffix
    // in "pat[0..state-1]c"

    // Start from the largest possible value and stop when you find
    // a prefix which is also suffix
    for (ns = state; ns > 0; ns--)
    {
        if(pat[ns-1] == x)
        {
            for(i = 0; i < ns-1; i++)
            {
                if (pat[i] != pat[state-ns+1+i])
                    break;
            }
            if (i == ns-1)
                return ns;
        }
    }

    return 0;
}
```

```
}

/* This function builds the TF table which represents Finite Automata
   given pattern  */
void computeTF(char *pat, int M, int TF[][NO_OF_CHARS])
{
    int state, x;
    for (state = 0; state <= M; ++state)
        for (x = 0; x < NO_OF_CHARS; ++x)
           TF[state][x] = getNextState(pat, M,  state, x);
}

/* Prints all occurrences of pat in txt */
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    int TF[M+1][NO_OF_CHARS];

    computeTF(pat, M, TF);

    // Process txt over FA.
    int i, state=0;
    for (i = 0; i < N; i++)
    {
       state = TF[state][txt[i]];
       if (state == M)
       {
         printf ("\n patterb found at index %d", i-M+1);
       }
    }
}

// Driver program to test above function
int main()
{
   char *txt = "AABAACAADAABAAABAA";
   char *pat = "AABA";
   search(pat, txt);
   return 0;
}
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

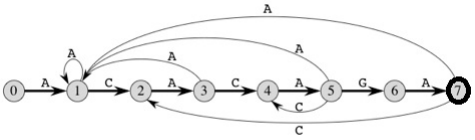
References:

Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

24. Pattern Searching | Set 6 (Efficient Construction of Finite Automata)

In the previous post, we discussed Finite Automata based pattern searching algorithm. The FA (Finite Automata) construction method discussed in previous post takes O((m^3)*NO_OF_CHARS) time. FA can be constructed in O(m*NO_OF_CHARS) time. In this post, we will discuss the O(m*NO_OF_CHARS) algorithm for FA construction. The idea is similar to lps (longest prefix suffix) array construction discussed in the KMP algorithm. We use previously filled rows to fill a new row.



state	character A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

The abvoe diagrams represent graphical and tabular representations of pattern ACACAGA.

Algorithm:
1) Fill the first row. All entries in first row are always 0 except the entry for pat[0] character. For pat[0] character, we always need to go to state 1.
2) Initialize lps as 0. lps for the first index is always 0.
3) Do following for rows at index i = 1 to M. (M is the length of the pattern)
…..a) Copy the entries from the row at index equal to lps.
…..b) Update the entry for pat[i] character to i+1.
…..c) Update lps “lps = TF[lps][pat[i]]” where TF is the 2D array which is being constructed.

Implementation
Following is C implementation for the above algorithm.

```
#include<stdio.h>
#include<string.h>
#define NO_OF_CHARS 256

/* This function builds the TF table which represents Finite Automata
   given pattern  */
void computeTransFun(char *pat, int M, int TF[][NO_OF_CHARS])
{
```

```
    int i, lps = 0, x;

    // Fill entries in first row
    for (x =0; x < NO_OF_CHARS; x++)
       TF[0][x] = 0;
    TF[0][pat[0]] = 1;

    // Fill entries in other rows
    for (i = 1; i<= M; i++)
    {
        // Copy values from row at index lps
        for (x = 0; x < NO_OF_CHARS; x++)
            TF[i][x] = TF[lps][x];

        // Update the entry corresponding to this character
        TF[i][pat[i]] = i + 1;

        // Update lps for next row to be filled
        if (i < M)
          lps = TF[lps][pat[i]];
    }
}

/* Prints all occurrences of pat in txt */
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    int TF[M+1][NO_OF_CHARS];

    computeTransFun(pat, M, TF);

    // process text over FA.
    int i, j=0;
    for (i = 0; i < N; i++)
    {
       j = TF[j][txt[i]];
       if (j == M)
       {
         printf ("\n pattern found at index %d", i-M+1);
       }
    }
}

/* Driver program to test above function */
int main()
{
    char *txt = "GEEKS FOR GEEKS";
    char *pat = "GEEKS";
    search(pat, txt);
    getchar();
    return 0;
}
```

Output:

```
pattern found at index 0
pattern found at index 10
```

Time Complexity for FA construction is O(M*NO_OF_CHARS). The code for search is same as the previous post and time complexity for it is O(n).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

25. Pattern Searching | Set 7 (Boyer Moore Algorithm – Bad Character Heuristic)

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

Examples:

```
1) Input:

  txt[] =  "THIS IS A TEST TEXT"
  pat[] = "TEST"

Output:

Pattern found at index 10

2) Input:

  txt[] =  "AABAACAADAABAAABAA"
  pat[] = "AABA"

Output:

   Pattern found at index 0
   Pattern found at index 9
   Pattern found at index 13
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed the following algorithms in the previous posts:

Naive Algorithm
KMP Algorithm
Rabin Karp Algorithm

Finite Automata based Algorithm

In this post, we will discuss Boyer Moore pattern searching algorithm. Like KMP and Finite Automata algorithms, Boyer Moore algorithm also preprocesses the pattern.
Boyer Moore is a combination of following two approaches.
1) Bad Character Heuristic
2) Good Suffix Heuristic
Both of the above heuristics can also be used independently to search a pattern in a text. Let us first understand how two independent approaches work together in the Boyer Moore algorithm. If we take a look at the Naive algorithm, it slides the pattern over the text one by one. KMP algorithm does preprocessing over the pattern so that the pattern can be shifted by more than one. The Boyer Moore algorithm does preprocessing for the same reason. It preporcesses the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by max of the slides suggested by the two heuristics. So it uses best of the two heuristics at every step. Unlike the previous pattern searching algorithms, Boyer Moore algorithm starts matching from the last character of the pattern.

In this post, we will discuss bad character heuristic, and discuss Good Suffix heuristic in the next post.

The idea of bad character heuristic is simple. The character of the text which doesn't match with the current character of pattern is called the Bad Character. Whenever a character doesn't match, we slide the pattern in such a way that aligns the bad character with the last occurrence of it in pattern. We preprocess the pattern and store the last occurrence of every possible character in an array of size equal to alphabet size. If the character is not present at all, then it may result in a shift by m (length of pattern). Therefore, the bad character heuristic takes O(n/m) time in the best case.

```
/* Program for Bad Character Heuristic of Boyer Moore String Matching A

# include <limits.h>
# include <string.h>
# include <stdio.h>

# define NO_OF_CHARS 256

// A utility function to get maximum of two integers
int max (int a, int b) { return (a > b)? a: b; }

// The preprocessing function for Boyer Moore's bad character heuristi
void badCharHeuristic( char *str, int size, int badchar[NO_OF_CHARS])
{
    int i;

    // Initialize all occurrences as -1
    for (i = 0; i < NO_OF_CHARS; i++)
         badchar[i] = -1;

    // Fill the actual value of last occurrence of a character
    for (i = 0; i < size; i++)
         badchar[(int) str[i]] = i;
}
```

```
/* A pattern searching function that uses Bad Character Heuristic of
   Boyer Moore Algorithm */
void search( char *txt,  char *pat)
{
    int m = strlen(pat);
    int n = strlen(txt);

    int badchar[NO_OF_CHARS];

    /* Fill the bad character array by calling the preprocessing
       function badCharHeuristic() for given pattern */
    badCharHeuristic(pat, m, badchar);

    int s = 0;  // s is shift of the pattern with respect to text
    while(s <= (n - m))
    {
        int j = m-1;

        /* Keep reducing index j of pattern while characters of
           pattern and text are matching at this shift s */
        while(j >= 0 && pat[j] == txt[s+j])
            j--;

        /* If the pattern is present at current shift, then index j
           will become -1 after the above loop */
        if (j < 0)
        {
            printf("\n pattern occurs at shift = %d", s);

            /* Shift the pattern so that the next character in text
               aligns with the last occurrence of it in pattern.
               The condition s+m < n is necessary for the case when
               pattern occurs at the end of text */
            s += (s+m < n)? m-badchar[txt[s+m]] : 1;

        }

        else
            /* Shift the pattern so that the bad character in text
               aligns with the last occurrence of it in pattern. The
               max function is used to make sure that we get a positive
               shift. We may get a negative shift if the last occurrence
               of bad character in pattern is on the right side of the
               current character. */
            s += max(1, j - badchar[txt[s+j]]);
    }
}

/* Driver program to test above funtion */
int main()
{
    char txt[] = "ABAAABCD";
    char pat[] = "ABC";
    search(txt, pat);
    return 0;
}
```

Output:

```
pattern occurs at shift = 4
```

The Bad Character Heuristic may take O(mn) time in worst case. The worst case occurs when all characters of the text and pattern are same. For example, txt[] = “AAAAAAAAAAAAAAAAAA” and pat[] = “AAAAA”.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

26. Dynamic Programming | Set 17 (Palindrome Partitioning)

Given a string, a partitioning of the string is a *palindrome partitioning* if every substring of the partition is a palindrome. For example, “aba|b|bbabb|a|b|aba” is a palindrome partitioning of “ababbbabbababa”. Determine the fewest cuts needed for palindrome partitioning of a given string. For example, minimum 3 cuts are needed for “ababbbabbababa”. The three cuts are “a|babbbab|b|ababa”. If a string is palindrome, then minimum 0 cuts are needed. If a string of length n containing all different characters, then minimum n-1 cuts are needed.

Solution
This problem is a variation of Matrix Chain Multiplication problem. If the string is palindrome, then we simply return 0. Else, like the Matrix Chain Multiplication problem, we try making cuts at all possible places, recursively calculate the cost for each cut and return the minimum value.

Let the given string be str and minPalPartion() be the function that returns the fewest cuts needed for palindrome partitioning. following is the optimal substructure property.

```
// i is the starting index and j is the ending index. i must be passed as 0 and j a
minPalPartion(str, i, j) = 0 if i == j. // When string is of length 1.
minPalPartion(str, i, j) = 0 if str[i..j] is palindrome.

// If none of the above conditions is true, then minPalPartion(str, i, j) can be
// calculated recursively using the following formula.
minPalPartion(str, i, j) = Min { minPalPartion(str, i, k) + 1 +
                                 minPalPartion(str, k+1, j) }
                           where k varies from i to j-1
```

Following is Dynamic Programming solution. It stores the solutions to subproblems in two arrays P[][] and C[][], and reuses the calculated values.

```
// Dynamic Programming Solution for Palindrome Partitioning Problem
#include <stdio.h>
#include <string.h>
#include <limits.h>
```

```
// A utility function to get minimum of two integers
int min (int a, int b) { return (a < b)? a : b; }

// Returns the minimum number of cuts needed to partition a string
// such that every part is a palindrome
int minPalPartion(char *str)
{
    // Get the length of the string
    int n = strlen(str);

    /* Create two arrays to build the solution in bottom up manner
       C[i][j] = Minimum number of cuts needed for palindrome partitio
                 of substring str[i..j]
       P[i][j] = true if substring str[i..j] is palindrome, else false
       Note that C[i][j] is 0 if P[i][j] is true */
    int C[n][n];
    bool P[n][n];

    int i, j, k, L; // different looping variables

    // Every substring of length 1 is a palindrome
    for (i=0; i<n; i++)
    {
        P[i][i] = true;
        C[i][i] = 0;
    }

    /* L is substring length. Build the solution in bottom up manner b
       considering all substrings of length starting from 2 to n.
       The loop structure is same as Matrx Chain Multiplication proble
       See http://www.geeksforgeeks.org/archives/15553 )*/
    for (L=2; L<=n; L++)
    {
        // For substring of length L, set different possible starting
        for (i=0; i<n-L+1; i++)
        {
            j = i+L-1; // Set ending index

            // If L is 2, then we just need to compare two characters.
            // need to check two corner characters and value of P[i+1]
            if (L == 2)
                P[i][j] = (str[i] == str[j]);
            else
                P[i][j] = (str[i] == str[j]) && P[i+1][j-1];

            // IF str[i..j] is palindrome, then C[i][j] is 0
            if (P[i][j] == true)
                C[i][j] = 0;
            else
            {
                // Make a cut at every possible localtion starting fro
                // and get the minimum cost cut.
                C[i][j] = INT_MAX;
                for (k=i; k<=j-1; k++)
                    C[i][j] = min (C[i][j], C[i][k] + C[k+1][j]+1);
            }
        }
    }

    // Return the min cut value for complete string. i.e., str[0..n-1]
    return C[0][n-1];
}
```

```
// Driver program to test above function
int main()
{
    char str[] = "ababbbabbababa";
    printf("Min cuts needed for Palindrome Partitioning is %d",
            minPalPartion(str));
    return 0;
}
```

Output:

```
Min cuts needed for Palindrome Partitioning is 3
```

Time Complexity: $O(n^3)$

An optimization to above approach

In above approach, we can calculating minimum cut while finding all palindromic substring. If we finding all palindromic substring 1st and then we calculate minimum cut, time complexity will reduce to $O(n^2)$.

Thanks for **Vivek** for suggesting this optimization.

```
// Dynamic Programming Solution for Palindrome Partitioning Problem
#include <stdio.h>
#include <string.h>
#include <limits.h>

// A utility function to get minimum of two integers
int min (int a, int b) { return (a < b)? a : b; }

// Returns the minimum number of cuts needed to partition a string
// such that every part is a palindrome
int minPalPartion(char *str)
{
    // Get the length of the string
    int n = strlen(str);

    /* Create two arrays to build the solution in bottom up manner
       C[i] = Minimum number of cuts needed for palindrome partitionin
              of substring str[0..i]
       P[i][j] = true if substring str[i..j] is palindrome, else false
       Note that C[i] is 0 if P[0][i] is true */
    int C[n];
    bool P[n][n];

    int i, j, k, L; // different looping variables

    // Every substring of length 1 is a palindrome
    for (i=0; i<n; i++)
    {
        P[i][i] = true;
    }

    /* L is substring length. Build the solution in bottom up manner b
       considering all substrings of length starting from 2 to n. */
    for (L=2; L<=n; L++)
    {
        // For substring of length L, set different possible starting
```

```
        for (i=0; i<n-L+1; i++)
        {
            j = i+L-1; // Set ending index

            // If L is 2, then we just need to compare two characters.
            // need to check two corner characters and value of P[i+1]
            if (L == 2)
                P[i][j] = (str[i] == str[j]);
            else
                P[i][j] = (str[i] == str[j]) && P[i+1][j-1];
        }
    }

    for (i=0; i<n; i++)
    {
        if (P[0][i] == true)
            C[i] = 0;
        else
        {
            C[i] = INT_MAX;
            for(j=0;j<i;j++)
            {
                if(P[j+1][i] == true && 1+C[j]<C[i])
                    C[i]=1+C[j];
            }
        }
    }

    // Return the min cut value for complete string. i.e., str[0..n-1]
    return C[n-1];
}

// Driver program to test above function
int main()
{
    char str[] = "ababbbabbababa";
    printf("Min cuts needed for Palindrome Partitioning is %d",
            minPalPartion(str));
    return 0;
}
```

Output:

```
Min cuts needed for Palindrome Partitioning is 3
```

Time Complexity: $O(n^2)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

27. Lexicographic rank of a string

Given a string, find its rank among all its permutations sorted lexicographically. For example, rank of “abc” is 1, rank of “acb” is 2, and rank of “cba” is 6.

For simplicity, let us assume that the string does not contain any duplicated characters.

One simple solution is to initialize rank as 1, generate all permutations in lexicographic order. After generating a permutation, check if the generated permutation is same as given string, if same, then return rank, if not, then increment the rank by 1. The time complexity of this solution will be exponential in worst case. Following is an efficient solution.

Let the given string be “STRING”. In the input string, ‘S’ is the first character. There are total 6 characters and 4 of them are smaller than ‘S’. So there can be 4 * 5! smaller strings where first character is smaller than ‘S’, like following

R X X X X X
I X X X X X
N X X X X X
G X X X X X

Now let us Fix S’ and find the smaller strings staring with ‘S’.

Repeat the same process for T, rank is 4*5! + 4*4! +…

Now fix T and repeat the same process for R, rank is 4*5! + 4*4! + 3*3! +…

Now fix R and repeat the same process for I, rank is 4*5! + 4*4! + 3*3! + 1*2! +…

Now fix I and repeat the same process for N, rank is 4*5! + 4*4! + 3*3! + 1*2! + 1*1! +…

Now fix N and repeat the same process for G, rank is 4*5! + 4*4 + 3*3! + 1*2! + 1*1! + 0*0!

Rank = 4*5! + 4*4! + 3*3! + 1*2! + 1*1! + 0*0! = 597

Since the value of rank starts from 1, the final rank = 1 + 597 = 598

```
#include <stdio.h>
#include <string.h>

// A utility function to find factorial of n
int fact(int n)
{
    return (n <= 1)? 1 :n * fact(n-1);
}

// A utility function to count smaller characters on right
// of arr[low]
int findSmallerInRight(char* str, int low, int high)
{
    int countRight = 0, i;

    for (i = low+1; i <= high; ++i)
        if (str[i] < str[low])
            ++countRight;

    return countRight;
}

// A function to find rank of a string in all permutations
// of characters
int findRank (char* str)
{
    int len = strlen(str);
    int mul = fact(len);
    int rank = 1;
    int countRight;

    int i;
    for (i = 0; i < len; ++i)
    {
        mul /= len - i;

        // count number of chars smaller than str[i]
        // fron str[i+1] to str[len-1]
        countRight = findSmallerInRight(str, i, len-1);

        rank += countRight * mul ;
    }

    return rank;
}

// Driver program to test above function
int main()
{
    char str[] = "string";
    printf ("%d", findRank(str));
    return 0;
}
```

Output

```
598
```

The time complexity of the above solution is O(n^2). We can reduce the time complexity to O(n) by creating an auxiliary array of size 256. See following code.

```
// A O(n) solution for finding rank of string
#include <stdio.h>
#include <string.h>
#define MAX_CHAR 256

// A utility function to find factorial of n
int fact(int n)
{
    return (n <= 1)? 1 :n * fact(n-1);
}

// Construct a count array where value at every index
// contains count of smaller characters in whole string
void populateAndIncreaseCount (int* count, char* str)
{
    int i;

    for( i = 0; str[i]; ++i )
        ++count[ str[i] ];

    for( i = 1; i < 256; ++i )
        count[i] += count[i-1];
}

// Removes a character ch from count[] array
// constructed by populateAndIncreaseCount()
void updatecount (int* count, char ch)
{
    int i;
    for( i = ch; i < MAX_CHAR; ++i )
        --count[i];
}

// A function to find rank of a string in all permutations
// of characters
int findRank (char* str)
{
    int len = strlen(str);
    int mul = fact(len);
    int rank = 1, i;
    int count[MAX_CHAR] = {0};  // all elements of count[] are initial

    // Populate the count array such that count[i] contains count of
    // characters which are present in str and are smaller than i
    populateAndIncreaseCount( count, str );

    for (i = 0; i < len; ++i)
    {
        mul /= len - i;

        // count number of chars smaller than str[i]
        // fron str[i+1] to str[len-1]
        rank += count[ str[i] - 1] * mul;

        // Reduce count of characters greater than str[i]
        updatecount (count, str[i]);
    }

    return rank;
}

// Driver program to test above function
int main()
```

```
int main()
{
    char str[] = "string";
    printf ("%d", findRank(str));
    return 0;
}
```

The above programs don't work for duplicate characters. To make them work for duplicate characters, find all the characters that are smaller (include equal this time also), do the same as above but, this time divide the rank so formed by p! where p is the count of occurrences of the repeating character.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

28. Print all permutations in sorted (lexicographic) order

Given a string, print all permutations of it in sorted order. For example, if the input string is "ABC", then output should be "ABC, ACB, BAC, BCA, CAB, CBA".

We have discussed a program to print all permutations in this post, but here we must print the permutations in increasing order.

Following are the steps to print the permutations lexicographic-ally

1. Sort the given string in non-decreasing order and print it. The first permutation is always the string sorted in non-decreasing order.

2. Start generating next higher permutation. Do it until next higher permutation is not possible. If we reach a permutation where all characters are sorted in non-increasing order, then that permutation is the last permutation.

Steps to generate the next higher permutation:
1. Take the previously printed permutation and find the rightmost character in it, which is smaller than its next character. Let us call this character as 'first character'.

2. Now find the ceiling of the 'first character'. Ceiling is the smallest character on right of 'first character', which is greater than 'first character'. Let us call the ceil character as 'second character'.

3. Swap the two characters found in above 2 steps.

4. Sort the substring (in non-decreasing order) after the original index of 'first character'.

Let us consider the string "ABCDEF". Let previously printed permutation be "DCFEBA". The next permutation in sorted order should be "DEABCF". Let us understand above

steps to find next permutation. The ‘first character’ will be ‘C’. The ‘second character’ will be ‘E’. After swapping these two, we get “DEFCBA”. The final step is to sort the substring after the character original index of ‘first character’. Finally, we get “DEABCF”.

Following is C++ implementation of the algorithm.

```
// Program to print all permutations of a string in sorted order.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Following function is needed for library function qsort(). Refer
   http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int compare (const void *a, const void * b)
{  return ( *(char *)a - *(char *)b ); }

// A utility function two swap two characters a and b
void swap (char* a, char* b)
{
    char t = *a;
    *a = *b;
    *b = t;
}

// This function finds the index of the smallest character
// which is greater than 'first' and is present in str[l..h]
int findCeil (char str[], char first, int l, int h)
{
    // initialize index of ceiling element
    int ceilIndex = l;

    // Now iterate through rest of the elements and find
    // the smallest character greater than 'first'
    for (int i = l+1; i <= h; i++)
      if (str[i] > first && str[i] < str[ceilIndex])
            ceilIndex = i;

    return ceilIndex;
}

// Print all permutations of str in sorted order
void sortedPermutations ( char str[] )
{
    // Get size of string
    int size = strlen(str);

    // Sort the string in increasing order
    qsort( str, size, sizeof( str[0] ), compare );

    // Print permutations one by one
    bool isFinished = false;
    while ( ! isFinished )
    {
        // print this permutation
        printf ("%s \n", str);

        // Find the rightmost character which is smaller than its next
        // character. Let us call it 'first char'
        int i;
        for ( i = size - 2; i >= 0; --i )
           if (str[i] < str[i+1])
```

```
            break;

        // If there is no such chracter, all are sorted in decreasing
        // means we just printed the last permutation and we are done.
        if ( i == -1 )
            isFinished = true;
        else
        {
            // Find the ceil of 'first char' in right of first charact
            // Ceil of a character is the smallest character greater t
            int ceilIndex = findCeil( str, str[i], i + 1, size - 1 );

            // Swap first and second characters
            swap( &str[i], &str[ceilIndex] );

            // Sort the string on right of 'first char'
            qsort( str + i + 1, size - i - 1, sizeof(str[0]), compare
        }
    }
}

// Driver program to test above function
int main()
{
    char str[] = "ABCD";
    sortedPermutations( str );
    return 0;
}
```

Output:

```
ABCD
ABDC
....
....
DCAB
DCBA
```

The upper bound on time complexity of the above program is O(n^2 x n!). We can optimize step 4 of the above algorithm for finding next permutation. Instead of sorting the subarray after the ‘first character’, we can reverse the subarray, because the subarray we get after swapping is always sorted in non-increasing order. This optimization makes the time complexity as O(n x n!). See following optimized code.

```
// An optimized version that uses reverse instead of sort for
// finding the next permutation

// A utility function to reverse a string str[l..h]
void reverse(char str[], int l, int h)
{
   while (l < h)
   {
       swap(&str[l], &str[h]);
       l++;
       h--;
   }
}
```

```
// Print all permutations of str in sorted order
void sortedPermutations ( char str[] )
{
    // Get size of string
    int size = strlen(str);

    // Sort the string in increasing order
    qsort( str, size, sizeof( str[0] ), compare );

    // Print permutations one by one
    bool isFinished = false;
    while ( ! isFinished )
    {
        // print this permutation
        printf ("%s \n", str);

        // Find the rightmost character which is smaller than its next
        // character. Let us call it 'first char'
        int i;
        for ( i = size - 2; i >= 0; --i )
           if (str[i] < str[i+1])
              break;

        // If there is no such chracter, all are sorted in decreasing
        // means we just printed the last permutation and we are done.
        if ( i == -1 )
            isFinished = true;
        else
        {
            // Find the ceil of 'first char' in right of first charact
            // Ceil of a character is the smallest character greater t
            int ceilIndex = findCeil( str, str[i], i + 1, size - 1 );

            // Swap first and second characters
            swap( &str[i], &str[ceilIndex] );

            // reverse the string on right of 'first char'
            reverse( str, i + 1, size - 1 );
        }
    }
}
```

The above programs print duplicate permutation when characters are repeated. We can avoid it by keeping track of the previous permutation. While printing, if the current permutation is same as previous permutation, we won't print it.

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

29. Longest Palindromic Substring | Set 1

Given a string, find the longest substring which is palindrome. For example, if the given string is “forgeeksskeegfor”, the output should be “geeksskeeg”.

Method 1 (Brute Force)
The simple approach is to check each substring whether the substring is a palindrome or not. We can run three loops, the outer two loops pick all substrings one by one by fixing the corner characters, the inner loop checks whether the picked substring is palindrome or not.

Time complexity: O (n^3)
Auxiliary complexity: O (1)

Method 2 (Dynamic Programming)
The time complexity can be reduced by storing results of subproblems. The idea is similar to this post. We maintain a boolean table[n][n] that is filled in bottom up manner. The value of table[i][j] is true, if the substring is palindrome, otherwise false. To calculate table[i][j], we first check the value of table[i+1][j-1], if the value is true and str[i] is same as str[j], then we make table[i][j] true. Otherwise, the value of table[i][j] is made false.

```
// A dynamic programming solution for longest palindr.
// This code is adopted from following link
// http://www.leetcode.com/2011/11/longest-palindromic-substring-part-
```

```
#include <stdio.h>
#include <string.h>

// A utility function to print a substring str[low..high]
void printSubStr( char* str, int low, int high )
{
    for( int i = low; i <= high; ++i )
        printf("%c", str[i]);
}
```

```
// This function prints the longest palindrome substring
// of str[].
// It also returns the length of the longest palindrome
int longestPalSubstr( char *str )
{
    int n = strlen( str ); // get length of input string

    // table[i][j] will be false if substring str[i..j]
    // is not palindrome.
    // Else table[i][j] will be true
```

```
    bool table[n][n];
    memset(table, 0, sizeof(table));

    // All substrings of length 1 are palindromes
    int maxLength = 1;
    for (int i = 0; i < n; ++i)
        table[i][i] = true;

    // check for sub-string of length 2.
    int start = 0;
    for (int i = 0; i < n-1; ++i)
    {
        if (str[i] == str[i+1])
        {
            table[i][i+1] = true;
            start = i;
            maxLength = 2;
        }
    }

    // Check for lengths greater than 2. k is length
    // of substring
    for (int k = 3; k <= n; ++k)
    {
        // Fix the starting index
        for (int i = 0; i < n-k+1 ; ++i)
        {
            // Get the ending index of substring from
            // starting index i and length k
            int j = i + k - 1;

            // checking for sub-string from ith index to
            // jth index iff str[i+1] to str[j-1] is a
            // palindrome
            if (table[i+1][j-1] && str[i] == str[j])
            {
                table[i][j] = true;

                if (k > maxLength)
                {
                    start = i;
                    maxLength = k;
                }
            }
        }
    }

    printf("Longest palindrome substring is: ");
    printSubStr( str, start, start + maxLength - 1 );

    return maxLength; // return length of LPS
}

// Driver program to test above functions
int main()
{
    char str[] = "forgeeksskeegfor";
    printf("\nLength is: %d\n", longestPalSubstr( str ) );
    return 0;
}
```

Output:

```
Longest palindrome substring is: geeksskeeg
Length is: 10
```

Time complexity: O (n^2)
Auxiliary Space: O (n^2)

We will soon be adding more optimized methods as separate posts.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

30. An in-place algorithm for String Transformation

Given a string, move all even positioned elements to end of string. While moving elements, keep the relative order of all even positioned and odd positioned elements same. For example, if the given string is “a1b2c3d4e5f6g7h8i9j1k2l3m4″, convert it to “abcdefghijklm1234567891234″ in-place and in O(n) time complexity.

Below are the steps:

1. Cut out the largest prefix sub-string of size of the form 3^k + 1. In this step, we find the largest non-negative integer k such that 3^k+1 is smaller than or equal to n (length of string)

2. Apply cycle leader iteration algorithm (it has been discussed below), starting with index 1, 3, 9…… to this sub-string. Cycle leader iteration algorithm moves all the items of this sub-string to their correct positions, i.e. all the alphabets are shifted to the left half of the sub-string and all the digits are shifted to the right half of this sub-string.

3. Process the remaining sub-string recursively using steps#1 and #2.

4. Now, we only need to join the processed sub-strings together. Start from any end (say from left), pick two sub-strings and apply the below steps:

….**4.1** Reverse the second half of first sub-string.
….**4.2** Reverse the first half of second sub-string.
….**4.3** Reverse the second half of first sub-string and first half of second sub-string together.

5. Repeat step#4 until all sub-strings are joined. It is similar to k-way merging where first sub-string is joined with second. The resultant is merged with third and so on.

Let us understand it with an example:

Please note that we have used values like 10, 11 12 in the below example. Consider these values as single characters only. These values are used for better readability.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
a 1 b 2 c 3 d 4 e 5 f  6  g  7  h  8  i  9  j  10 k  11 l  12 m  13
```

After breaking into size of the form 3^k + 1, two sub-strings are formed of size 10 each. The third sub-string is formed of size 4 and the fourth sub-string is formed of size 2.

```
0 1 2 3 4 5 6 7 8 9
a 1 b 2 c 3 d 4 e 5

10 11 12 13 14 15 16 17 18 19
f  6  g  7  h  8  i  9  j  10

20 21 22 23
k  11 l  12

24 25
m  13
```

After applying cycle leader iteration algorithm to first sub-string:

```
0 1 2 3 4 5 6 7 8 9
a b c d e 1 2 3 4 5

10 11 12 13 14 15 16 17 18 19
f  6  g  7  h  8  i  9  j  10

20 21 22 23
k  11 l  12

24 25
m  13
```

After applying cycle leader iteration algorithm to second sub-string:

```
0 1 2 3 4 5 6 7 8 9
a b c d e 1 2 3 4 5

10 11 12 13 14 15 16 17 18 19
f  g  h  i  j  6  7  8  9  10

20 21 22 23
k  11 l  12
```

```
24 25
m 13
```

After applying cycle leader iteration algorithm to third sub-string:

```
0 1 2 3 4 5 6 7 8 9
a b c d e 1 2 3 4 5

10 11 12 13 14 15 16 17 18 19
f  g  h  i  j  6  7  8  9  10

20 21 22 23
k  l  11 12

24 25
m  13
```

After applying cycle leader iteration algorithm to fourth sub-string:

```
0 1 2 3 4 5 6 7 8 9
a b c d e 1 2 3 4 5

10 11 12 13 14 15 16 17 18 19
f  g  h  i  j  6  7  8  9  10

20 21 22 23
k  l  11 12

24 25
m  13
```

Joining first sub-string and second sub-string:

1. Second half of first sub-string and first half of second sub-string reversed.

```
0 1 2 3 4 5 6 7 8 9
a b c d e 5 4 3 2 1            <--------- First Sub-string

10 11 12 13 14 15 16 17 18 19
j  i  h  g  f  6  7  8  9  10  <--------- Second Sub-string

20 21 22 23
k  l  11 12

24 25
m  13
```

2. Second half of first sub-string and first half of second sub-string reversed together(They are merged, i.e. there are only three sub-strings now).

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
a b c d e f g h i j 1  2  3  4  5  6  7  8  9  10

20 21 22 23
k  l  11 12

24 25
m  13
```

Joining first sub-string and second sub-string:
1. Second half of first sub-string and first half of second sub-string reversed.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
a b c d e f g h i j 10 9  8  7  6  5  4  3  2  1 <--------- First Sub-string

20 21 22 23
l  k  11 12                                      <--------- Second Sub-string

24 25
m  13
```

2. Second half of first sub-string and first half of second sub-string reversed together(They are merged, i.e. there are only two sub-strings now).

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
a b c d e f g h i j k  l  1  2  3  4  5  6  7  8  9  10 11 12

24 25
m  13
```

Joining first sub-string and second sub-string:
1. Second half of first sub-string and first half of second sub-string reversed.

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
a b c d e f g h i j k  l  12 11 10 9  8  7  6  5  4  3  2  1 <----- First Sub-strin

24 25
m  13   <----- Second Sub-string
```

2. Second half of first sub-string and first half of second sub-string reversed together(They are merged, i.e. there is only one sub-string now).

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

```
a b c d e f g h i j k  l  m  1  2  3  4  5  6  7  8  9  10 11 12 13
```

Since all sub-strings have been joined together, we are done.

How does cycle leader iteration algorithm work?

Let us understand it with an example:

```
Input:
0 1 2 3 4 5 6 7 8 9
a 1 b 2 c 3 d 4 e 5

Output:
0 1 2 3 4 5 6 7 8 9
a b c d e 1 2 3 4 5

Old index    New index
0  0
1  5
2  1
3  6
4  2
5  7
6  3
7  8
8  4
9  9
```

Let len be the length of the string. If we observe carefully, we find that the new index is given by below formula:

```
if( oldIndex is odd )
 newIndex = len / 2 + oldIndex / 2;
else
        newIndex = oldIndex / 2;
```

So, the problem reduces to shifting the elements to new indexes based on the above formula.

Cycle leader iteration algorithm will be applied starting from the indices of the form 3^k, starting with k = 0.

Below are the steps:

1. Find new position for item at position i. Before putting this item at new position, keep the back-up of element at new position. Now, put the item at new position.

2. Repeat step#1 for new position until a cycle is completed, i.e. until the procedure comes back to the starting position.

3. Apply cycle leader iteration algorithm to the next index of the form 3^k. Repeat this step until 3^k < len.

Consider input array of size 28:

The first cycle leader iteration, starting with index 1:

1->14->7->17->22->11->19->23->25->26->13->20->10->5->16->8->4->2->1

The second cycle leader iteration, starting with index 3:

3->15->21->24->12->6->3

The third cycle leader iteration, starting with index 9:

9->18->9

Based on the above algorithm, below is the code:

```
#include <stdio.h>
#include <string.h>
#include <math.h>

// A utility function to swap characters
void swap ( char* a, char* b )
{
    char t = *a;
    *a = *b;
    *b = t;
}

// A utility function to reverse string str[low..high]
void reverse ( char* str, int low, int high )
{
    while ( low < high )
    {
        swap( &str[low], &str[high] );
        ++low;
        --high;
    }
}

// Cycle leader algorithm to move all even positioned elements
// at the end.
void cycleLeader ( char* str, int shift, int len )
{
    int j;
    char item;

    for (int i = 1; i < len; i *= 3 )
    {
        j = i;

        item = str[j + shift];
        do
        {
```

```
        {
            // odd index
            if ( j & 1 )
                j = len / 2 + j / 2;
            // even index
            else
                j /= 2;

            // keep the back-up of element at new position
            swap (&str[j + shift], &item);
        }
        while ( j != i );
    }
}

// The main function to transform a string. This function mainly uses
// cycleLeader() to transform
void moveNumberToSecondHalf( char* str )
{
    int k, lenFirst;

    int lenRemaining = strlen( str );
    int shift = 0;

    while ( lenRemaining )
    {
        k = 0;

        // Step 1: Find the largest prefix subarray of the form 3^k + 1
        while ( pow( 3, k ) + 1 <= lenRemaining )
            k++;
        lenFirst = pow( 3, k - 1 ) + 1;
        lenRemaining -= lenFirst;

        // Step 2: Apply cycle leader algorithm for the largest subarray
        cycleLeader ( str, shift, lenFirst );

        // Step 4.1: Reverse the second half of first subarray
        reverse ( str, shift / 2, shift - 1 );

        // Step 4.2: Reverse the first half of second sub-string.
        reverse ( str, shift, shift + lenFirst / 2 - 1 );

        // Step 4.3 Reverse the second half of first sub-string and fi
        // half of second sub-string together
        reverse ( str, shift / 2, shift + lenFirst / 2 - 1 );

        // Increase the length of first subarray
        shift += lenFirst;
    }
}

// Driver program to test above function
int main()
{
    char str[] = "a1b2c3d4e5f6g7";
    moveNumberToSecondHalf( str );
    printf( "%s", str );
    return 0;
}
```

Click here to see various test cases.

Notes:
1. If the array size is already in the form 3^k + 1, We can directly apply cycle leader iteration algorithm. There is no need of joining.

2. Cycle leader iteration algorithm is only applicable to arrays of size of the form 3^k + 1.

How is the time complexity O(n) ?
Each item in a cycle is shifted at most once. Thus time complexity of the cycle leader algorithm is O(n). The time complexity of the reverse operation is O(n). We will soon update the mathematical proof of the time complexity of the algorithm.

Exercise:
Given string in the form "abcdefg1234567", convert it to "a1b2c3d4e5f6g7" in-place and in O(n) time complexity.

References:
A Simple In-Place Algorithm for In-Shu?e.

__Aashish Barnwal.Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

31. Longest Palindromic Substring | Set 2

Given a string, find the longest substring which is palindrome. For example, if the given string is "forgeeksskeegfor", the output should be "geeksskeeg".

We have discussed dynamic programming solution in the previous post. The time complexity of the Dynamic Programming based solution is O(n^2) and it requires O(n^2) extra space. We can find the longest palindrome substring in (n^2) time with O(1) extra space. The idea is to generate all even length and odd length palindromes and keep track of the longest palindrome seen so far.

Step to generate odd length palindrome:
Fix a centre and expand in both directions for longer palindromes.

Step to generate even length palindrome
Fix two centre (low and high) and expand in both directions for longer palindromes.

```
// A O(n^2) time and O(1) space program to find the longest palindromi
#include <stdio.h>
#include <string.h>

// A utility function to print a substring str[low, high]
```

```
// A utility function to print a substring str[low..high]
void printSubStr(char* str, int low, int high)
{
    for( int i = low; i <= high; ++i )
        printf("%c", str[i]);
}

// This function prints the longest palindrome substring (LPS)
// of str[]. It also returns the length of the longest palindrome
int longestPalSubstr(char *str)
{
    int maxLength = 1;  // The result (length of LPS)

    int start = 0;
    int len = strlen(str);

    int low, high;

    // One by one consider every character as center point of
    // even and length palindromes
    for (int i = 1; i < len; ++i)
    {
        // Find the longest even length palindrome with center points
        // as i-1 and i.
        low = i - 1;
        high = i;
        while (low >= 0 && high < len && str[low] == str[high])
        {
            if (high - low + 1 > maxLength)
            {
                start = low;
                maxLength = high - low + 1;
            }
            --low;
            ++high;
        }

        // Find the longest odd length palindrome with center
        // point as i
        low = i - 1;
        high = i + 1;
        while (low >= 0 && high < len && str[low] == str[high])
        {
            if (high - low + 1 > maxLength)
            {
                start = low;
                maxLength = high - low + 1;
            }
            --low;
            ++high;
        }
    }

    printf("Longest palindrome substring is: ");
    printSubStr(str, start, start + maxLength - 1);

    return maxLength;
}

// Driver program to test above functions
int main()
{
    char str[] = "forgeeksskeegfor";
```

```
    printf("\nLength is: %d\n", longestPalSubstr( str ) );
    return 0;
}
```

Output:

```
Longest palindrome substring is: geeksskeeg
Length is: 10
```

Time complexity: O (n^2) where n is the length of input string.
Auxiliary Space: O (1)

We will soon be adding more optimized method as separate post.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

32. Given a sequence of words, print all anagrams together | Set 1

Given an array of words, print all anagrams together. For example, if the given array is {“cat”, “dog”, “tac”, “god”, “act”}, then output may be “cat tac act dog god”.

A **simple method** is to create a Hash Table. Calculate the hash value of each word in such a way that all anagrams have the same hash value. Populate the Hash Table with these hash values. Finally, print those words together with same hash values. A simple hashing mechanism can be modulo sum of all characters. With modulo sum, two non-anagram words may have same hash value. This can be handled by matching individual characters.

Following is **another method** to print all anagrams together. Take two auxiliary arrays, index array and word array. Populate the word array with the given sequence of words. Sort each individual word of the word array. Finally, sort the word array and keep track of the corresponding indices. After sorting, all the anagrams cluster together. Use the index array to print the strings from the original array of strings.

Let us understand the steps with following input Sequence of Words:

```
"cat", "dog", "tac", "god", "act"
```

1) Create two auxiliary arrays index[] and words[]. Copy all given words to words[] and store the original indexes in index[]

```
index[]:  0   1   2   3   4
words[]: cat dog tac god act
```

2) Sort individual words in words[]. Index array doesn't change.

```
index[]:   0    1    2    3    4
words[]:  act  dgo  act  dgo  act
```

3) Sort the words array. Compare individual words using strcmp() to sort

```
index:     0    2    4    1    3
words[]:  act  act  act  dgo  dgo
```

4) All anagrams come together. But words are changed in words array. To print the original words, take index from the index array and use it in the original array. We get

```
"cat tac act dog god"
```

Following is C implementation of the above algorithm. In the following program, an array of structure "Word" is used to store both index and word arrays. DupArray is another structure that stores array of structure "Word".

```
// A program to print all anagarms together
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// structure for each word of duplicate array
struct Word
{
    char* str;  // to store word itself
    int index; // index of the word in the original array
};

// structure to represent duplicate array.
struct DupArray
{
    struct Word* array;  // Array of words
    int size;   // Size of array
};

// Create a DupArray object that contains an array of Words
struct DupArray* createDupArray(char* str[], int size)
{
    // Allocate memory for dupArray and all members of it
    struct DupArray* dupArray =
              (struct DupArray*) malloc( sizeof(struct DupArray) );
    dupArray->size = size;
    dupArray->array =
              (struct Word*) malloc( dupArray->size * sizeof(struct Wo

    // One by one copy words from the given wordArray to dupArray
    int i;
    for (i = 0; i < size; ++i)
    {
        dupArray->array[i].index = i;
        dupArray->array[i].str = (char*) malloc( strlen(str[i]) + 1 );
        strcpy(dupArray->array[i].str, str[i] );
    }

    return dupArray;
```

```
}

// Compare two characters. Used in qsort() for sorting an array of cha
int compChar(const void* a, const void* b)
{
    return *(char*)a - *(char*)b;
}

// Compare two words. Used in qsort() for sorting an array of words
int compStr(const void* a, const void* b)
{
    struct Word* a1 = (struct Word *)a;
    struct Word* b1 = (struct Word *)b;
    return strcmp(a1->str, b1->str);
}

// Given a list of words in wordArr[],
void printAnagramsTogether(char* wordArr[], int size)
{
    // Step 1: Create a copy of all words present in given wordArr.
    // The copy will also have orignal indexes of words
    struct DupArray* dupArray = createDupArray(wordArr, size);

    // Step 2: Iterate through all words in dupArray and sort individu
    int i;
    for (i = 0; i < size; ++i)
       qsort(dupArray->array[i].str,
             strlen(dupArray->array[i].str), sizeof(char), compChar);

    // Step 3: Now sort the array of words in dupArray
    qsort(dupArray->array, size, sizeof(dupArray->array[0]), compStr);

    // Step 4: Now all words in dupArray are together, but these words
    // changed. Use the index member of word struct to get the corresp
    // original word
    for (i = 0; i < size; ++i)
        printf("%s ", wordArr[dupArray->array[i].index]);
}

// Driver program to test above functions
int main()
{
    char* wordArr[] = {"cat", "dog", "tac", "god", "act"};
    int size = sizeof(wordArr) / sizeof(wordArr[0]);
    printAnagramsTogether(wordArr, size);
    return 0;
}
```

Output:

```
act tac cat god dog
```

Time Complexity: Let there be N words and each word has maximum M characters. The upper bound is O(NMLogM + MNLogN).
Step 2 takes O(NMLogM) time. Sorting a word takes maximum O(MLogM) time. So sorting N words takes O(NMLogM) time. step 3 takes O(MNLogN) Sorting array of words takes NLogN comparisons. A comparison may take maximum O(M) time. So time to sort array of words will be O(MNLogN).

We will soon be publishing more efficient methods to solve this problem. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

33. Given a sequence of words, print all anagrams together | Set 2

Given an array of words, print all anagrams together. For example, if the given array is {“cat”, “dog”, “tac”, “god”, “act”}, then output may be “cat tac act dog god”.

We have discussed two different methods in the previous post. In this post, a more efficient solution is discussed.

Trie data structure can be used for a more efficient solution. Insert the sorted order of each word in the trie. Since all the anagrams will end at the same leaf node. We can start a linked list at the leaf nodes where each node represents the index of the original array of words. Finally, traverse the Trie. While traversing the Trie, traverse each linked list one line at a time. Following are the detailed steps.

1) Create an empty Trie
2) One by one take all words of input sequence. Do following for each word
…**a)** Copy the word to a buffer.
…**b)** Sort the buffer
…**c)** Insert the sorted buffer and index of this word to Trie. Each leaf node of Trie is head of a Index list. The Index list stores index of words in original sequence. If sorted buffe is already present, we insert index of this word to the index list.
3) Traverse Trie. While traversing, if you reach a leaf node, traverse the index list. And print all words using the index obtained from Index list.

```
// An efficient program to print all anagrams together
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define NO_OF_CHARS 26

// Structure to represent list node for indexes of words in
// the given sequence. The list nodes are used to connect
// anagrams at leaf nodes of Trie
struct IndexNode
{
    int index;
    struct IndexNode* next;
};

// Structure to represent a Trie Node
struct TrieNode
{
```

```
{
    bool isEnd;  // indicates end of word
    struct TrieNode* child[NO_OF_CHARS]; // 26 slots each for 'a' to '
    struct IndexNode* head; // head of the index list
};


// A utility function to create a new Trie node
struct TrieNode* newTrieNode()
{
    struct TrieNode* temp = new TrieNode;
    temp->isEnd = 0;
    temp->head = NULL;
    for (int i = 0; i < NO_OF_CHARS; ++i)
        temp->child[i] = NULL;
    return temp;
}

/* Following function is needed for library function qsort(). Refer
   http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int compare(const void* a, const void* b)
{  return *(char*)a - *(char*)b; }

/* A utility function to create a new linked list node */
struct IndexNode* newIndexNode(int index)
{
    struct IndexNode* temp = new IndexNode;
    temp->index = index;
    temp->next = NULL;
    return temp;
}

// A utility function to insert a word to Trie
void insert(struct TrieNode** root, char* word, int index)
{
    // Base case
    if (*root == NULL)
        *root = newTrieNode();

    if (*word != '\0')
        insert( &( (*root)->child[tolower(*word) - 'a'] ), word+1, ind
    else  // If end of the word reached
    {
        // Insert index of this word to end of index linked list
        if ((*root)->isEnd)
        {
            IndexNode* pCrawl = (*root)->head;
            while( pCrawl->next )
                pCrawl = pCrawl->next;
            pCrawl->next = newIndexNode(index);
        }
        else  // If Index list is empty
        {
            (*root)->isEnd = 1;
            (*root)->head = newIndexNode(index);
        }
    }
}

// This function traverses the built trie. When a leaf node is reached
// all words connected at that leaf node are anagrams. So it traverses
// the list at leaf node and uses stored index to print original words
void printAnagramsUtil(struct TrieNode* root, char *wordArr[])
```

```
{
    if (root == NULL)
        return;

    // If a lead node is reached, print all anagrams using the indexes
    // stored in index linked list
    if (root->isEnd)
    {
        // traverse the list
        IndexNode* pCrawl = root->head;
        while (pCrawl != NULL)
        {
            printf( "%s \n", wordArr[ pCrawl->index ] );
            pCrawl = pCrawl->next;
        }
    }

    for (int i = 0; i < NO_OF_CHARS; ++i)
        printAnagramsUtil(root->child[i], wordArr);
}

// The main function that prints all anagrams together. wordArr[] is i
// sequence of words.
void printAnagramsTogether(char* wordArr[], int size)
{
    // Create an empty Trie
    struct TrieNode* root = NULL;

    // Iterate through all input words
    for (int i = 0; i < size; ++i)
    {
        // Create a buffer for this word and copy the word to buffer
        int len = strlen(wordArr[i]);
        char *buffer = new char[len+1];
        strcpy(buffer, wordArr[i]);

        // Sort the buffer
        qsort( (void*)buffer, strlen(buffer), sizeof(char), compare );

        // Insert the sorted buffer and its original index to Trie
        insert(&root,  buffer, i);
    }

    // Traverse the built Trie and print all anagrms together
    printAnagramsUtil(root, wordArr);
}


// Driver program to test above functions
int main()
{
    char* wordArr[] = {"cat", "dog", "tac", "god", "act", "gdo"};
    int size = sizeof(wordArr) / sizeof(wordArr[0]);
    printAnagramsTogether(wordArr, size);
    return 0;
}
```

Output:

```
cat
```

```
tac
act
dog
god
gdo
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

34. Count words in a given string

Given a string, count number of words in it. The words are separated by following characters: space (‘ ‘) or new line (‘\n’) or tab (‘\t’) or a combination of these.

There can be many solutions to this problem. Following is a simple and interesting solution.
The idea is to maintain two states: IN and OUT. The state OUT indicates that a separator is seen. State IN indicates that a word character is seen. We increment word count when previous state is OUT and next character is a word character.

```
/* Program to count no of words from given input string. */
#include <stdio.h>

#define OUT 0
#define IN  1

// returns number of words in str
unsigned countWords(char *str)
{
    int state = OUT;
    unsigned wc = 0;  // word count

    // Scan all characters one by one
    while (*str)
    {
        // If next character is a separator, set the state as OUT
        if (*str == ' ' || *str == '\n' || *str == '\t')
            state = OUT;

        // If next character is not a word separator and state is OUT,
        // then set the state as IN and increment word count
        else if (state == OUT)
        {
            state = IN;
            ++wc;
        }

        // Move to next character
        ++str;
    }

    return wc;
}

// Driver program to tes above functions
int main(void)
{
    char str[] = "One two          three\n  four\nfive  ";
    printf("No of words: %u\n", countWords(str));
    return 0;
}
```

Output:

```
No of words: 5
```

Time complexity: O(n)

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

35. String matching where one string contains wildcard characters

Given two strings where first string may contain wild card characters and second string is a normal string. Write a function that returns true if the two strings match. The following are allowed wild card characters in first string.

```
* --> Matches with 0 or more instances of any character or set of characters.
? --> Matches with any one character.
```

For example, “g*ks” matches with “geeks” match. And string “ge?ks*” matches with “geeksforgeeks” (note ‘*’ at the end of first string). But “g*k” doesn’t match with “gee” as character ‘k’ is not present in second string.

```c
// A C program to match wild card characters
#include <stdio.h>
#include <stdbool.h>

// The main function that checks if two given strings match. The first
// string may contain wildcard characters
bool match(char *first, char * second)
{
    // If we reach at the end of both strings, we are done
    if (*first == '\0' && *second == '\0')
        return true;

    // Make sure that the characters after '*' are present in second st
    // This function assumes that the first string will not contain two
    // consecutive '*'
    if (*first == '*' && *(first+1) != '\0' && *second == '\0')
        return false;

    // If the first string contains '?', or current characters of both
    // strings match
    if (*first == '?' || *first == *second)
        return match(first+1, second+1);

    // If there is *, then there are two possibilities
    // a) We consider current character of second string
    // b) We ignore current character of second string.
    if (*first == '*')
        return match(first+1, second) || match(first, second+1);
    return false;
}

// A function to run test cases
void test(char *first, char *second)
{  match(first, second)? puts("Yes"): puts("No"); }

// Driver program to test above functions
int main()
{
    test("g*ks", "geeks"); // Yes
    test("ge?ks*", "geeksforgeeks"); // Yes
    test("g*k", "gee");  // No because 'k' is not in second
    test("*pqrs", "pqrst"); // No because 't' is not in first
    test("abc*bcd", "abcdhghgbcd"); // Yes
    test("abc*c?d", "abcd"); // No because second must have 2 instances
    test("*c*d", "abcd"); // Yes
    test("*?c*d", "abcd"); // Yes
    return 0;
}
```

Output:

```
Yes
Yes
No
No
Yes
No
Yes
```

```
Yes
```

Exercise
1) In the above solution, all non-wild characters of first string must be there is second string and all characters of second string must match with either a normal character or wildcard character of first string. Extend the above solution to work like other pattern searching solutions where the first string is pattern and second string is text and we should print all occurrences of first string in second.

2) Write a pattern searching function where the meaning of '?' is same, but '*' means 0 or more occurrences of the character just before '*'. For example, if first string is 'a*b', then it matches with 'aaab', but doesn't match with 'abb'.

This article is compiled by Vishal Chaudhary and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

36. Write your own atoi()

The atoi() function takes a string (which represents an integer) as an argument and returns its value.

Following is a simple implementation. We initialize result as 0. We start from the first character and update result for every character.

```
// A simple C++ program for implementation of atoi
#include <stdio.h>

// A simple atoi() function
int myAtoi(char *str)
{
    int res = 0; // Initialize result

    // Iterate through all characters of input string and update resul
    for (int i = 0; str[i] != '\0'; ++i)
        res = res*10 + str[i] - '0';

    // return result.
    return res;
}

// Driver program to test above function
int main()
{
    char str[] = "89789";
    int val = myAtoi(str);
    printf ("%d ", val);
    return 0;
}
```

Output:

```
89789
```

The above function doesn't handle negative numbers. **Following is a simple extension to handle negative numbers**.

```
// A C++ program for implementation of atoi
#include <stdio.h>

// A simple atoi() function
int myAtoi(char *str)
{
    int res = 0;  // Initialize result
    int sign = 1;  // Initialize sign as positive
    int i = 0;  // Initialize index of first digit

    // If number is negative, then update sign
    if (str[0] == '-')
    {
        sign = -1;
        i++;  // Also update index of first digit
    }

    // Iterate through all digits and update the result
    for (; str[i] != '\0'; ++i)
        res = res*10 + str[i] - '0';

    // Return result with sign
    return sign*res;
}

// Driver program to test above function
int main()
{
    char str[] = "-123";
    int val = myAtoi(str);
    printf ("%d ", val);
    return 0;
}
```

Output:

```
-123
```

The above implementation doesn't handle errors. What if *str* is NULL or *str* contains non-numeric characters. **Following implementation handles errors**.

```
// A simple C++ program for implementation of atoi

#include <stdio.h>

// A utility function to check whether x is numeric
bool isNumericChar(char x)
{
    return (x >= '0' && x <= '9')? true: false;
}

// A simple atoi() function. If the given string contains
// any invalid character, then this function returns 0
int myAtoi(char *str)
{
    if (*str == NULL)
       return 0;

    int res = 0;  // Initialize result
    int sign = 1;  // Initialize sign as positive
    int i = 0;  // Initialize index of first digit

    // If number is negative, then update sign
    if (str[0] == '-')
    {
        sign = -1;
        i++;  // Also update index of first digit
    }

    // Iterate through all digits of input string and update result
    for (; str[i] != '\0'; ++i)
    {
        if (isNumericChar(str[i]) == false)
            return 0; // You may add some lines to write error message
                      // to error stream
        res = res*10 + str[i] - '0';
    }

    // Return result with sign
    return sign*res;
}

// Driver program to test above function
int main()
{
    char str[] = "-134";
    int val = myAtoi(str);
    printf("%d ", val);
    return 0;
}
```

Time Complexity: O(n) where n is the number of characters in input string.

Exercise
Write your won atof() that takes a string (which represents an floating point value) as an argument and returns its value as double.

This article is compiled by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.
If you like GeeksforGeeks and would like to contribute, you can also write an article and

mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

37. Dynamic Programming | Set 29 (Longest Common Substring)

Given two strings ‘X’ and ‘Y’, find the length of the longest common substring. For example, if the given strings are “GeeksforGeeks” and “GeeksQuiz”, the output should be 5 as longest common substring is “Geeks”

Let m and n be the lengths of first and second strings respectively.

A **simple solution** is to one by one consider all substrings of first string and for every substring check if it is a substring in second string. Keep track of the maximum length substring. There will be O(m^2) substrings and we can find whether a string is subsring on another string in O(n) time (See this). So overall time complexity of this method would be O(n * m^2)

Dynamic Programming can be used to find the longest common substring in O(m*n) time. The idea is to find length of the longest common suffix for all substrings of both strings and store these lengths in a table.

```
The longest common suffix has following optimal substructure property
   LCSuff(X, Y, m, n) = LCSuff(X, Y, m-1, n-1) + 1 if X[m-1] = Y[n-1]
                        0  Otherwise (if X[m-1] != Y[n-1])

The maximum length Longest Common Suffix is the longest common substring.
   LCSubStr(X, Y, m, n)  = Max(LCSuff(X, Y, i, j)) where 1 <= i <= m
                                                   and 1 <= j <= n
```

Following is C++ implementation of the above solution.

```
/* Dynamic Programming solution to find length of the longest common s
#include<iostream>
#include<string.h>
using namespace std;

// A utility function to find maximum of two integers
int max(int a, int b)
{   return (a > b)? a : b; }

/* Returns length of longest common substring of X[0..m-1] and Y[0..n-
int LCSubStr(char *X, char *Y, int m, int n)
{
    // Create a table to store lengths of longest common suffixes of
    // substrings.   Notethat LCSuff[i][j] contains length of longest
    // common suffix of X[0..i-1] and Y[0..j-1]. The first row and
    // first column entries have no logical meaning, they are used onl
    // for simplicity of program
    int LCSuff[m+1][n+1];
    int result = 0;  // To store length of the longest common substrin

    /* Following steps build LCSuff[m+1][n+1] in bottom up fashion. */
    for (int i=0; i<=m; i++)
    {
        for (int j=0; j<=n; j++)
        {
            if (i == 0 || j == 0)
                LCSuff[i][j] = 0;

            else if (X[i-1] == Y[j-1])
            {
                LCSuff[i][j] = LCSuff[i-1][j-1] + 1;
                result = max(result, LCSuff[i][j]);
            }
            else LCSuff[i][j] = 0;
        }
    }
    return result;
}

/* Driver program to test above function */
int main()
{
    char X[] = "OldSite:GeeksforGeeks.org";
    char Y[] = "NewSite:GeeksQuiz.com";

    int m = strlen(X);
    int n = strlen(Y);

    cout << "Length of Longest Common Substring is " << LCSubStr(X, Y,
    return 0;
}
```

Output:

```
Length of Longest Common Substring is 10
```

Time Complexity: O(m*n)
Auxiliary Space: O(m*n)

References: http://en.wikipedia.org/wiki/Longest_common_substring_problem

The longest substring can also be solved in O(n+m) time using Suffix Tree. We will be covering Suffix Tree based solution in a separate post.

Exercise: The above solution prints only length of the longest common substring. Extend the solution to print the substring also.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

38. Remove “b” and “ac” from a given string

Given a string, eliminate all “b” and “ac” in the string, you have to replace them in-place, and you are only allowed to iterate over the string once. (Source Google Interview Question)

Examples:

```
acbac   ==>  ""
aaac    ==>  aa
ababac  ==>   aa
bbbbd   ==>   d
```

The two conditions are:
1. Filtering of all ‘b’ and ‘ac’ should be in single pass
2. No extra space allowed.

The approach is to use two index variables i and j. We move forward in string using ‘i’ and add characters using index j except ‘b’ and ‘ac’. The trick here is how to track ‘a’ before ‘c’. An interesting approach is to use a two state machine. The state is maintained to TWO when previous character is ‘a’, otherwise state is ONE.
1) If state is ONE, then do NOT copy the current character to output if one of the following conditions is true
…**a)** Current character is ‘b’ (We need to remove ‘b’)
…**b)** Current character is ‘a’ (Next character may be ‘c’)
2) If state is TWO and current character is not ‘c’, we first need to make sure that we copy the previous character ‘a’. Then we check the current character, if current character is not ‘b’ and not ‘a’, then we copy it to output.

```
// A C++ program to remove "b" and 'ac' from input string
#include <iostream>
using namespace std;
#define ONE 1
#define TWO 2
```

```
// The main function that removes occurences of "a" and "bc" in input
void stringFilter(char *str)
{
    // state is initially ONE (The previous character is not a)
    int state = ONE;

    // i and j are index variables, i is used to read next character o
    // string, j is used for indexes of output string (modified input
    int j = 0;

    // Process all characters of input string one by one
    for (int i = 0; str[i] != '\0'; i++)
    {
        /* If state is ONE, then do NOT copy the current character to
           one of the following conditions is true
           ...a) Current character is 'b' (We need to remove 'b')
           ...b) Current character is 'a' (Next character may be 'c')
        if (state == ONE && str[i] != 'a' && str[i] != 'b')
        {
            str[j] = str[i];
            j++;
        }

        // If state is TWO and current character is not 'c' (otherwise
        // we ignore both previous and current characters)
        if (state == TWO && str[i] != 'c')
        {
            // First copy the previous 'a'
            str[j] = 'a';
            j++;

            // Then copy the current character if it is not 'a' and 'b
            if (str[i] != 'a' && str[i] != 'b')
            {
                str[j] = str[i];
                j++;
            }
        }

        // Change state according to current character
        state = (str[i] == 'a')? TWO: ONE;
    }

    // If last character was 'a', copy it to output
    if (state == TWO)
    {
        str[j] = 'a';
        j++;
    }

    // Set the string terminator
    str[j] = '\0';
}

/* Driver program to check above functions */
int main()
{
    char str1[] = "ad";
    stringFilter(str1);
    cout << str1 << endl;

    char str2[] = "acbac";
    stringFilter(str2);
```

```
    stringFilter(str2);
    cout << str2 << endl;

    char str3[] = "aaac";
    stringFilter(str3);
    cout << str3 << endl;

    char str4[] = "react";
    stringFilter(str4);
    cout << str4 << endl;

    char str5[] = "aa";
    stringFilter(str5);
    cout << str5 << endl;

    char str6[] = "abababaac";
    stringFilter(str6);
    cout << str6 << endl;

    return 0;
}
```

Output:

```
ad


aa
ret
aa
aaa
```

An extension of above problem where we don't want "ac" in output at all:
The above code looks fine and seems to handle all cases, but what if input string is "aacacc", the above code produces output as "ac" which looks correct as it removes consecutive occurrences of 'a' and 'c'. What if the requirement is to not have an "ac" in output string at all. Can we modify the above program to produce output as empty string for input "aacacc" and produce output as "d" when input is "abcaaccd"? It turns out that it can also be done with given restrictions. The idea is simple. We need to add following lines inside for loop of the above program.

```
if (j>1 && str[j-2] == 'a' && str[j-1] =='c')
  j = j-2;
```

See this for different test cases of modified program.

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

39. Dynamic Programming | Set 33 (Find if a string is interleaved of

two other strings)

Given three strings A, B and C. Write a function that checks whether C is an interleaving of A and B. C is said to be interleaving A and B, if it contains all characters of A and B and order of all characters in individual strings is preserved.

We have discussed a simple solution of this problem here. The simple solution doesn't work if strings A and B have some common characters. For example A = "XXY", string B = "XXZ" and string C = "XXZXXXY". To handle all cases, two possibilities need to be considered.

a) If first character of C matches with first character of A, we move one character ahead in A and C and recursively check.

b) If first character of C matches with first character of B, we move one character ahead in B and C and recursively check.

If any of the above two cases is true, we return true, else false. Following is simple recursive implementation of this approach (Thanks to Frederic for suggesting this)

```
// A simple recursive function to check whether C is an interleaving o
bool isInterleaved(char *A, char *B, char *C)
{
    // Base Case: If all strings are empty
    if (!(*A || *B || *C))
        return true;

    // If C is empty and any of the two strings is not empty
    if (*C == '\0')
        return false;

    // If any of the above mentioned two possibilities is true,
    // then return true, otherwise false
    return ( (*C == *A) && isInterleaved(A+1, B, C+1))
           || ((*C == *B) && isInterleaved(A, B+1, C+1));
}
```

Dynamic Programming

The worst case time complexity of recursive solution is $O(2^n)$. The above recursive solution certainly has many overlapping subproblems. For example, if wee consider A = "XXX", B = "XXX" and C = "XXXXXX" and draw recursion tree, there will be many overlapping subproblems.
Therefore, like other typical Dynamic Programming problems, we can solve it by creating a table and store results of subproblems in bottom up manner. Thanks to Abhinav Ramana for suggesting this method and implementation.

```
// A Dynamic Programming based program to check whether a string C is
// an interleaving of two other strings A and B.
#include <iostream>
#include <string.h>
using namespace std;
```

```
// The main function that returns true if C is
// an interleaving of A and B, otherwise false.
bool isInterleaved(char* A, char* B, char* C)
{
    // Find lengths of the two strings
    int M = strlen(A), N = strlen(B);

    // Let us create a 2D table to store solutions of
    // subproblems.  C[i][j] will be true if C[0..i+j-1]
    // is an interleaving of A[0..i-1] and B[0..j-1].
    bool IL[M+1][N+1];

    memset(IL, 0, sizeof(IL)); // Initialize all values as false.

    // C can be an interleaving of A and B only of sum
    // of lengths of A & B is equal to length of C.
    if ((M+N) != strlen(C))
       return false;

    // Process all characters of A and B
    for (int i=0; i<=M; ++i)
    {
        for (int j=0; j<=N; ++j)
        {
            // two empty strings have an empty string
            // as interleaving
            if (i==0 && j==0)
                IL[i][j] = true;

            // A is empty
            else if (i==0 && B[j-1]==C[j-1])
                IL[i][j] = IL[i][j-1];

            // B is empty
            else if (j==0 && A[i-1]==C[i-1])
                IL[i][j] = IL[i-1][j];

            // Current character of C matches with current character o
            // but doesn't match with current character of B
            else if(A[i-1]==C[i+j-1] && B[j-1]!=C[i+j-1])
                IL[i][j] = IL[i-1][j];

            // Current character of C matches with current character o
            // but doesn't match with current character of A
            else if (A[i-1]!=C[i+j-1] && B[j-1]==C[i+j-1])
                IL[i][j] = IL[i][j-1];

            // Current character of C matches with that of both A and
            else if (A[i-1]==C[i+j-1] && B[j-1]==C[i+j-1])
                IL[i][j]=(IL[i-1][j] || IL[i][j-1]) ;
        }
    }

    return IL[M][N];
}

// A function to run test cases
void test(char *A, char *B, char *C)
{
    if (isInterleaved(A, B, C))
        cout << C <<" is interleaved of " << A <<" and " << B << endl;
    else
```

```
        cout << C <<" is not interleaved of " << A <<" and " << B << en
}

// Driver program to test above functions
int main()
{
    test("XXY", "XXZ", "XXZXXXY");
    test("XY" ,"WZ" ,"WZXY");
    test ("XY", "X", "XXY");
    test ("YX", "X", "XXY");
    test ("XXY", "XXZ", "XXXXZY");
    return 0;
}
```

Output:

```
XXZXXXY is not interleaved of XXY and XXZ
WZXY is interleaved of XY and WZ
XXY is interleaved of XY and X
XXY is not interleaved of YX and X
XXXXZY is interleaved of XXY and XXZ
```

See this for more test cases.

Time Complexity: O(MN)
Auxiliary Space: O(MN)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

40. Find the first non-repeating character from a stream of characters

Given a stream of characters, find the first non-repeating character from stream. You need to tell the first non-repeating character in O(1) time at any moment.

If we follow the first approach discussed here, then we need to store the stream so that we can traverse it one more time to find the first non-repeating character at any moment. If we use extended approach discussed in the same post, we need to go through the count array every time first non-repeating element is queried. We can find the first non-repeating character from stream at any moment without traversing any array.

We strongly recommend you to minimize the browser and try it yourself first.

The idea is to use a DLL (**D**oubly **L**inked **L**ist) to efficiently get the first non-repeating character from a stream. The DLL contains all non-repeating characters in order, i.e., the

head of DLL contains first non-repeating character, the second node contains the second non-repeating and so on.
We also maintain two arrays: one array is to maintain characters that are already visited two or more times, we call it repeated[], the other array is array of pointers to linked list nodes, we call it inDLL[]. The size of both arrays is equal to alphabet size which is typically 256.

```
1) Create an empty DLL. Also create two arrays inDLL[] and repeated[] of size 256.
   inDLL is an array of pointers to DLL nodes. repeated[] is a boolean array,
   repeated[x] is true if x is repeated two or more times, otherwise false.
   inDLL[x] contains pointer to a DLL node if character x is present in DLL,
   otherwise NULL.

2) Initialize all entries of inDLL[] as NULL and repeated[] as false.

3) To get the first non-repeating character, return character at head of DLL.

4) Following are steps to process a new character 'x' in stream.
  a) If repeated[x] is true, ignore this character (x is already repeated two
      or more times in the stream)
  b) If repeated[x] is false and inDLL[x] is NULL (x is seen first time)
     Append x to DLL and store address of new DLL node in inDLL[x].
  c) If repeated[x] is false and inDLL[x] is not NULL (x is seen second time)
     Get DLL node of x using inDLL[x] and remove the node. Also, mark inDLL[x]
     as NULL and repeated[x] as true.
```

Note that appending a new node to DLL is O(1) operation if we maintain tail pointer. Removing a node from DLL is also O(1). So both operations, addition of new character and finding first non-repeating character take O(1) time.

```
// A C++ program to find first non-repeating character from a stream o
#include <iostream>
#define MAX_CHAR 256
using namespace std;

// A linked list node
struct node
{
    char a;
    struct node *next, *prev;
};

// A utility function to append a character x at the end of DLL.
// Note that the function may change head and tail pointers, that
// is why pointers to these pointers are passed.
void appendNode(struct node **head_ref, struct node **tail_ref, char x
{
    struct node *temp = new node;
    temp->a = x;
    temp->prev = temp->next = NULL;

    if (*head_ref == NULL)
```

```
    {
        *head_ref = *tail_ref = temp;
        return;
    }
    (*tail_ref)->next = temp;
    temp->prev = *tail_ref;
    *tail_ref = temp;
}

// A utility function to remove a node 'temp' fromt DLL. Note that the
// function may change head and tail pointers, that is why pointers to
// these pointers are passed.
void removeNode(struct node **head_ref, struct node **tail_ref,
                struct node *temp)
{
    if (*head_ref == NULL)
        return;

    if (*head_ref == temp)
        *head_ref = (*head_ref)->next;
    if (*tail_ref == temp)
        *tail_ref = (*tail_ref)->prev;
    if (temp->next != NULL)
        temp->next->prev = temp->prev;
    if (temp->prev != NULL)
        temp->prev->next = temp->next;

    delete(temp);
}

void findFirstNonRepeating()
{
    // inDLL[x] contains pointer to a DLL node if x is present in DLL.
    // If x is not present, then inDLL[x] is NULL
    struct node *inDLL[MAX_CHAR];

    // repeated[x] is true if x is repeated two or more times. If x is
    // not seen so far or x is seen only once. then repeated[x] is fal
    bool repeated[MAX_CHAR];

    // Initialize the above two arrays
    struct node *head = NULL, *tail = NULL;
    for (int i = 0; i < MAX_CHAR; i++)
    {
        inDLL[i] = NULL;
        repeated[i] = false;
    }

    // Let us consider following stream and see the process
    char stream[] = "geeksforgeeksandgeeksquizfor";
    for (int i = 0; stream[i]; i++)
    {
        char x = stream[i];
        cout << "Reading " << x << " from stream \n";

        // We process this character only if it has not occurred or oc
        // only once. repeated[x] is true if x is repeated twice or mo
        if (!repeated[x])
        {
            // If the character is not in DLL, then add this at the en
            if (inDLL[x] == NULL)
            {
                appendNode(&head, &tail, stream[i]);
```

```
                appendNode(&head, &tail, stream[i]);
                inDLL[x] = tail;
            }
            else // Otherwise remove this caharacter from DLL
            {
                removeNode(&head, &tail, inDLL[x]);
                inDLL[x] = NULL;
                repeated[x] = true;  // Also mark it as repeated
            }
        }

        // Print the current first non-repeating character from stream
        if (head != NULL)
            cout << "First non-repeating character so far is " << head
    }
}

/* Driver program to test above function */
int main()
{
    findFirstNonRepeating();
    return 0;
}
```

Output:

```
Reading g from stream
First non-repeating character so far is g
Reading e from stream
First non-repeating character so far is g
Reading e from stream
First non-repeating character so far is g
Reading k from stream
First non-repeating character so far is g
Reading s from stream
First non-repeating character so far is g
Reading f from stream
First non-repeating character so far is g
Reading o from stream
First non-repeating character so far is g
Reading r from stream
First non-repeating character so far is g
Reading g from stream
First non-repeating character so far is k
Reading e from stream
First non-repeating character so far is k
Reading e from stream
First non-repeating character so far is k
Reading k from stream
First non-repeating character so far is s
Reading s from stream
First non-repeating character so far is f
```

```
Reading a from stream
First non-repeating character so far is f
Reading n from stream
First non-repeating character so far is f
Reading d from stream
First non-repeating character so far is f
Reading g from stream
First non-repeating character so far is f
Reading e from stream
First non-repeating character so far is f
Reading e from stream
First non-repeating character so far is f
Reading k from stream
First non-repeating character so far is f
Reading s from stream
First non-repeating character so far is f
Reading q from stream
First non-repeating character so far is f
Reading u from stream
First non-repeating character so far is f
Reading i from stream
First non-repeating character so far is f
Reading z from stream
First non-repeating character so far is f
Reading f from stream
First non-repeating character so far is o
Reading o from stream
First non-repeating character so far is r
Reading r from stream
First non-repeating character so far is a
```

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

41. Recursively remove all adjacent duplicates

Given a string, recursively remove adjacent duplicate characters from string. The output string should not have any adjacent duplicates. See following examples.

```
Input:  azxxzy
Output: ay
```

```
First "azxxzy" is reduced to "azzy". The string "azzy" contains duplicates,
so it is further reduced to "ay".

Input: geeksforgeeg
Output: gksfor
First "geeksforgeeg" is reduced to "gksforgg". The string "gksforgg" contains
duplicates, so it is further reduced to "gksfor".

Input: caaabbbaacdddd
Output: Empty String

Input: acaaabbbacdddd
Output: acac
```

A simple approach would be to run the input string through multiple passes. In every pass remove all adjacent duplicates from left to right. Stop running passes when there are no duplicates. The worst time complexity of this method would be O(n^2).

We can remove all duplicates in O(n) time.
1) Start from the leftmost character and remove duplicates at left corner if there are any.
2) The first character must be different from its adjacent now. Recur for string of length n-1 (string without first character).
3) Let the string obtained after reducing right substring of length n-1 be *rem_str*. There are three possible cases
……..**a)** If first character of *rem_str* matches with the first character of original string, remove the first character from *rem_str*.
……..**b)** Else if the last removed character in recursive calls is same as the first character of the original string. Ignore the first character of original string and return *rem_str*.
……..**c)** Else, append the first character of the original string at the beginning of *rem_str*.
4) Return *rem_str*.

Following is C++ implementation of the above algorithm.

```
#include <iostream>
#include <string.h>
using namespace std;

// Recursively removes adjacent duplicates from str and returns new
// string. las_removed is a pointer to last_removed character
char* removeUtil(char *str, char *last_removed)
{
    // If length of string is 1 or 0
    if (str[0] == '\0' || str[1] == '\0')
        return str;

    // Remove leftmost same characters and recur for remaining string
    if (str[0] == str[1])
    {
        *last_removed = str[0];
        while (str[1] && str[0] == str[1])
```

```
            str++;
        str++;
        return removeUtil(str, last_removed);
    }

    // At this point, the first character is definiotely different fro
    // adjacent. Ignore first character and recursively remove charact
    // remaining string
    char* rem_str = removeUtil(str+1, last_removed);

    // Check if the first character of the rem_string matches with the
    // first character of the original string
    if (rem_str[0] && rem_str[0] == str[0])
    {
        *last_removed = str[0];
        return (rem_str+1); // Remove first character
    }

    // If remaining string becomes empty and last removed character
    // is same as first character of original string.  This is needed
    // for a string like "acbbcddc"
    if (rem_str[0] == '\0' && *last_removed == str[0])
         return rem_str;

    // If the two first characters of str and rem_str don't match, app
    // first character of str before the first character of rem_str.
    rem_str--;
    rem_str[0] = str[0];
    return rem_str;
}

char *remove(char *str)
{
    char last_removed = '\0';
    return removeUtil(str, &last_removed);
}

// Driver program to test above functions
int main()
{
    char str1[] = "geeksforgeeg";
    cout << remove(str1) << endl;

    char str2[] = "azxxxzy";
    cout << remove(str2) << endl;

    char str3[] = "caaabbbaac";
    cout << remove(str3) << endl;

    char str4[] = "gghhg";
    cout << remove(str4) << endl;

    char str5[] = "aaaacddddcappp";
    cout << remove(str5) << endl;

    char str6[] = "aaaaaaaaaa";
    cout << remove(str6) << endl;

    char str7[] = "qpaaaaadaaaaadprq";
    cout << remove(str7) << endl;

    char str8[] = "acaaabbbacdddd";
```

```
    cout << remove(str8) << endl;

    char str9[] = "acbbcddc";
    cout << remove(str9) << endl;

    return 0;
}
```

Output:

```
gksfor
ay

g
a

qrq
acac
a
```

Time Complexity: The time complexity of the solution can be written as T(n) = T(n-k) + O(k) where n is length of the input string and k is the number of first characters which are same. Solution of the recurrence is O(n)

Thanks to **Prachi Bodke** for suggesting this problem and initial solution. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

42. Rearrange a string so that all same characters become d distance away

Given a string and a positive integer d. Some characters may be repeated in the given string. Rearrange characters of the given string such that the same characters become d distance away from each other. Note that there can be many possible rearrangements, the output should be one of the possible rearrangements. If no such arrangement is possible, that should also be reported.
Expected time complexity is O(n) where n is length of input string.

```
Examples:
Input:  "abb", d = 2
Output: "bab"

Input:  "aacbbc", d = 3
Output: "abcabc"
```

```
Input: "geeksforgeeks", d = 3
Output: egkegkesfesor

Input:  "aaa",  d = 2
Output: Cannot be rearranged
```

We strongly recommend to minimize the browser and try this yourself first.
Hint: Alphabet size may be assumed as constant (256) and extra space may be used.

Solution: The idea is to count frequencies of all characters and consider the most frequent character first and place all occurrences of it as close as possible. After the most frequent character is placed, repeat the same process for remaining characters.

1) Let the given string be str and size of string be n

2) Traverse str, store all characters and their frequencies in a Max Heap MH. The value of frequency decides the order in MH, i.e., the most frequent character is at the root of MH.

3) Make all characters of str as '\0'.

4) Do following while MH is not empty.
…**a)** Extract the Most frequent character. Let the extracted character be x and its frequency be f.
…**b)** Find the first available position in str, i.e., find the first '\0' in str.
…**c)** Let the first position be p. Fill x at p, p+d,.. p+(f-1)d

Following is C++ implementation of above algorithm.

```
// Rearrange a string so that all same characters become at least d
// distance away
#include <iostream>
#include <cstring>
#include <cstdlib>
#define MAX 256
using namespace std;

// A structure to store a character 'c' and its frequency 'f'
// in input string
struct charFreq {
    char c;
    int f;
};

// A utility function to swap two charFreq items.
void swap(charFreq *x, charFreq *y) {
    charFreq z = *x;
    *x = *y;
    *y = z;
}

// A utility function to maxheapify the node freq[i] of a heap
// stored in freq[]
```

```
void maxHeapify(charFreq freq[], int i, int heap_size)
{
    int l = i*2 + 1;
    int r = i*2 + 2;
    int largest = i;
    if (l < heap_size && freq[l].f > freq[i].f)
        largest = l;
    if (r < heap_size && freq[r].f > freq[largest].f)
        largest = r;
    if (largest != i)
    {
        swap(&freq[i], &freq[largest]);
        maxHeapify(freq, largest, heap_size);
    }
}

// A utility function to convert the array freq[] to a max heap
void buildHeap(charFreq freq[], int n)
{
    int i = (n - 1)/2;
    while (i >= 0)
    {
        maxHeapify(freq, i, n);
        i--;
    }
}

// A utility function to remove the max item or root from max heap
charFreq extractMax(charFreq freq[], int heap_size)
{
    charFreq root = freq[0];
    if (heap_size > 1)
    {
        freq[0] = freq[heap_size-1];
        maxHeapify(freq, 0, heap_size-1);
    }
    return root;
}
```

```
// The main function that rearranges input string 'str' such that
// two same characters become d distance away
void rearrange(char str[], int d)
{
    // Find length of input string
    int n = strlen(str);

    // Create an array to store all characters and their
    // frequencies in str[]
    charFreq freq[MAX] = {{0, 0}};

    int m = 0; // To store count of distinct characters in str[]

    // Traverse the input string and store frequencies of all
    // characters in freq[] array.
    for (int i = 0; i < n; i++)
    {
        char x = str[i];

        // If this character has occurred first time, increment m
        if (freq[x].c == 0)
            freq[x].c = x, m++;

        (freq[x].f)++;
```

```
        str[i] = '\0';  // This change is used later
    }

    // Build a max heap of all characters
    buildHeap(freq, MAX);

    // Now one by one extract all distinct characters from max heap
    // and put them back in str[] with the d distance constraint
    for (int i = 0; i < m; i++)
    {
        charFreq x = extractMax(freq, MAX-i);

        // Find the first available position in str[]
        int p = i;
        while (str[p] != '\0')
            p++;

        // Fill x.c at p, p+d, p+2d, .. p+(f-1)d
        for (int k = 0; k < x.f; k++)
        {
            // If the index goes beyond size, then string cannot
            // be rearranged.
            if (p + d*k >= n)
            {
                cout << "Cannot be rearranged";
                exit(0);
            }
            str[p + d*k] = x.c;
        }
    }
}

// Driver program to test above functions
int main()
{
    char str[] = "aabbcc";
    rearrange(str, 3);
    cout << str;
}
```

Output:

```
abcabc
```

Algorithmic Paradigm: Greedy Algorithm

Time Complexity: Time complexity of above implementation is O(n + mLog(MAX)). Here n is the length of str, m is count of distinct characters in str[] and MAX is maximum possible different characters. MAX is typically 256 (a constant) and m is smaller than MAX. So the time complexity can be considered as O(n).

More Analysis:
The above code can be optimized to store only m characters in heap, we have kept it this way to keep the code simple. So the time complexity can be improved to O(n + mLogm). It doesn't much matter through as MAX is a constant.

Also, the above algorithm can be implemented using a O(mLogm) sorting algorithm. The

first steps of above algorithm remain same. Instead of building a heap, we can sort the freq[] array in non-increasing order of frequencies and then consider all characters one by one from sorted array.

We will soon be covering an extended version where same characters should be moved at least d distance away.

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

43. Suffix Array | Set 2 (nLogn Algorithm)

A suffix array is a sorted array of all suffixes of a given string. The definition is similar to Suffix Tree which is compressed trie of all suffixes of the given text.

```
Let the given string be "banana".

0 banana                          5 a
1 anana     Sort the Suffixes     3 ana
2 nana      ----------------->    1 anana
3 ana        alphabetically       0 banana
4 na                              4 na
5 a                               2 nana

The suffix array for "banana" is {5, 3, 1, 0, 4, 2}
```

We have discussed **Naive algorithm** for construction of suffix array. The Naive algorithm is to consider all suffixes, sort them using a O(nLogn) sorting algorithm and while sorting, maintain original indexes. Time complexity of the Naive algorithm is $O(n^2Logn)$ where n is the number of characters in the input string.

In this post, a **O(nLogn) algorithm** for suffix array construction is discussed. Let us first discuss a O(n * Logn * Logn) algorithm for simplicity. The idea is to use the fact that strings that are to be sorted are suffixes of a single string.
We first sort all suffixes according to first character, then according to first 2 characters, then first 4 characters and so on while the number of characters to be considered is smaller than 2n. The important point is, if we have sorted suffixes according to first 2^i characters, then we can sort suffixes according to first 2^{i+1} characters in O(nLogn) time using a nLogn sorting algorithm like Merge Sort. This is possible as two suffixes can be compared in O(1) time (we need to compare only two values, see the below example and code).

The sort function is called O(Logn) times (Note that we increase number of characters to be considered in powers of 2). Therefore overall time complexity becomes O(nLognLogn). See http://www.stanford.edu/class/cs97si/suffix-array.pdf for more details.

Let us build suffix array the example string “banana” using above algorithm.

Sort according to first two characters Assign a rank to all suffixes using ASCII value of first character. A simple way to assign rank is to do “str[i] – ‘a’” for ith suffix of strp[]

Index	Suffix	Rank
0	banana	1
1	anana	0
2	nana	13
3	ana	0
4	na	13
5	a	0

For every character, we also store rank of next adjacent character, i.e., the rank of character at str[i + 1] (This is needed to sort the suffixes according to first 2 characters). If a character is last character, we store next rank as -1

Index	Suffix	Rank	Next Rank
0	banana	1	0
1	anana	0	13
2	nana	13	0
3	ana	0	13
4	na	13	0
5	a	0	-1

Sort all Suffixes according to rank and adjacent rank. Rank is considered as first digit or MSD, and adjacent rank is considered as second digit.

Index	Suffix	Rank	Next Rank
5	a	0	-1
1	anana	0	13
3	ana	0	13
0	banana	1	0
2	nana	13	0
4	na	13	0

Sort according to first four character
Assign new ranks to all suffixes. To assign new ranks, we consider the sorted suffixes one by one. Assign 0 as new rank to first suffix. For assigning ranks to remaining suffixes, we consider rank pair of suffix just before the current suffix. If previous rank pair of a suffix is same as previous rank of suffix just before it, then assign it same rank. Otherwise assign rank of previous suffix plus one.

```
Index       Suffix          Rank
  5          a               0     [Assign 0 to first]
  1          anana           1     (0, 13) is different from previous
  3          ana             1     (0, 13) is same as previous
  0          banana          2     (1, 0) is different from previous
  2          nana            3     (13, 0) is different from previous
  4          na              3     (13, 0) is same as previous
```

For every suffix str[i], also store rank of next suffix at str[i + 2]. If there is no next suffix at i + 2, we store next rank as -1

```
Index       Suffix          Rank        Next Rank
  5          a               0             -1
  1          anana           1              1
  3          ana             1              0
  0          banana          2              3
  2          nana            3              3
  4          na              3              -1
```

Sort all Suffixes according to rank and next rank.

```
Index       Suffix          Rank        Next Rank
  5          a               0             -1
  3          ana             1              0
  1          anana           1              1
  0          banana          2              3
  4          na              3             -1
  2          nana            3              3
```

```cpp
// C++ program for building suffix array of a given text
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index; // To store original index
    int rank[2]; // To store ranks and next rank pair
};

// A comparison function used by sort() to compare two suffixes
// Compares two pairs, returns 1 if first pair is smaller
int cmp(struct suffix a, struct suffix b)
{
    return (a.rank[0] == b.rank[0])? (a.rank[1] < b.rank[1] ?1: 0):
               (a.rank[0] < b.rank[0] ?1: 0);
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
int *buildSuffixArray(char *txt, int n)
{
```

```
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabatically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].rank[0] = txt[i] - 'a';
        suffixes[i].rank[1] = ((i+1) < n)? (txt[i + 1] - 'a'): -1;
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // At his point, all suffixes are sorted according to first
    // 2 characters.  Let us sort suffixes according to first 4
    // characters, then first 8 and so on
    int ind[n];  // This array is needed to get the index in suffixes[
                 // from original index.  This mapping is needed to ge
                 // next suffix.
    for (int k = 4; k < 2*n; k = k*2)
    {
        // Assigning rank and index values to first suffix
        int rank = 0;
        int prev_rank = suffixes[0].rank[0];
        suffixes[0].rank[0] = rank;
        ind[suffixes[0].index] = 0;

        // Assigning rank to suffixes
        for (int i = 1; i < n; i++)
        {
            // If first rank and next ranks are same as that of previo
            // suffix in array, assign the same new rank to this suffi
            if (suffixes[i].rank[0] == prev_rank &&
                    suffixes[i].rank[1] == suffixes[i-1].rank[1])
            {
                prev_rank = suffixes[i].rank[0];
                suffixes[i].rank[0] = rank;
            }
            else // Otherwise increment rank and assign
            {
                prev_rank = suffixes[i].rank[0];
                suffixes[i].rank[0] = ++rank;
            }
            ind[suffixes[i].index] = i;
        }

        // Assign next rank to every suffix
        for (int i = 0; i < n; i++)
        {
            int nextindex = suffixes[i].index + k/2;
            suffixes[i].rank[1] = (nextindex < n)?
                                  suffixes[ind[nextindex]].rank[0]: -1
        }

        // Sort the suffixes according to first k characters
        sort(suffixes, suffixes+n, cmp);
    }
```

```
    // Store indexes of all sorted suffixes in the suffix array
    int *suffixArr = new int[n];
    for (int i = 0; i < n; i++)
        suffixArr[i] = suffixes[i].index;

    // Return the suffix array
    return  suffixArr;
}

// A utility function to print an array of given size
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    char txt[] = "banana";
    int n = strlen(txt);
    int *suffixArr = buildSuffixArray(txt,  n);
    cout << "Following is suffix array for " << txt << endl;
    printArr(suffixArr, n);
    return 0;
}
```

Output:

```
Following is suffix array for banana
5 3 1 0 4 2
```

Note that the above algorithm uses standard sort function and therefore time complexity is O(nLognLogn). We can use Radix Sort here to reduce the time complexity to O(nLogn).

Please note that suffx arrays can be constructed in O(n) time also. We will soon be discussing O(n) algorithms.

References:
http://www.stanford.edu/class/cs97si/suffix-array.pdf
http://www.cbcb.umd.edu/confcour/Fall2012/lec14b.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

44. Printing Longest Common Subsequence

Given two sequences, print the longest subsequence present in both of them.

Examples:
LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.
LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

We have discussed Longest Common Subsequence (LCS) problem in a previous post. The function discussed there was mainly to find the length of LCS. To find length of LCS, a 2D table L[][] was constructed. In this post, the function to construct and print LCS is discussed.

Following is detailed algorithm to print the LCS. It uses the same 2D table L[][].

1) Construct L[m+1][n+1] using the steps discussed in previous post.

2) The value L[m][n] contains length of LCS. Create a character array lcs[] of length equal to the length of lcs plus 1 (one extra to store \0).

2) Traverse the 2D array starting from L[m][n]. Do following for every cell L[i][j]
…..**a)** If characters (in X and Y) corresponding to L[i][j] are same (Or X[i-1] == Y[j-1]), then include this character as part of LCS.
…..**b)** Else compare values of L[i-1][j] and L[i][j-1] and go in direction of greater value.

The following table (taken from Wiki) shows steps (highlighted) followed by the above algorithm.

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U
0	Ø	**0**	0	0	0	0	0	0	0
1	X	0	0	0	0	0	0	1	1
2	M	0	**1**	1	1	1	1	1	1
3	J	0	1	1	**2**	2	2	2	2
4	Y	0	1	1	**2**	2	2	2	2
5	A	0	1	1	2	**3**	3	3	3
6	U	0	1	1	2	3	3	3	**4**
7	Z	0	1	2	2	3	3	3	4

Following is C++ implementation of above approach.

```
/* Dynamic Programming implementation of LCS problem */
#include<iostream>
#include<cstring>
#include<cstdlib>
using namespace std;

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
void lcs( char *X, char *Y, int m, int n )
{
   int L[m+1][n+1];

   /* Following steps build L[m+1][n+1] in bottom up fashion. Note
      that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] *
```

```
    for (int i=0; i<=m; i++)
    {
      for (int j=0; j<=n; j++)
      {
        if (i == 0 || j == 0)
          L[i][j] = 0;
        else if (X[i-1] == Y[j-1])
          L[i][j] = L[i-1][j-1] + 1;
        else
          L[i][j] = max(L[i-1][j], L[i][j-1]);
      }
    }

    // Following code is used to print LCS
    int index = L[m][n];

    // Create a character array to store the lcs string
    char lcs[index+1];
    lcs[index] = '\0'; // Set the terminating character

    // Start from the right-most-bottom-most corner and
    // one by one store characters in lcs[]
    int i = m, j = n;
    while (i > 0 && j > 0)
    {
       // If current character in X[] and Y are same, then
       // current character is part of LCS
       if (X[i-1] == Y[j-1])
       {
           lcs[index-1] = X[i-1]; // Put current character in result
           i--; j--; index--;     // reduce values of i, j and index
       }

       // If not same, then find the larger of two and
       // go in the direction of larger value
       else if (L[i-1][j] > L[i][j-1])
          i--;
       else
          j--;
    }

    // Print the lcs
    cout << "LCS of " << X << " and " << Y << " is " << lcs;
}

/* Driver program to test above function */
int main()
{
  char X[] = "AGGTAB";
  char Y[] = "GXTXAYB";
  int m = strlen(X);
  int n = strlen(Y);
  lcs(X, Y, m, n);
  return 0;
}
```

Output:

```
LCS of AGGTAB and GXTXAYB is GTAB
```

References:
http://en.wikipedia.org/wiki/Longest_common_subsequence_problem

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

45. Print all possible words from phone digits

Before advent of QWERTY keyboards, texts and numbers were placed on the same key. For example 2 has “ABC” if we wanted to write anything starting with ‘A’ we need to type key 2 once. If we wanted to type ‘B’, press key 2 twice and thrice for typing ‘C’. below is picture of such keypad.

Given a keypad as shown in diagram, and a n digit number, list all words which are possible by pressing these numbers.
For example if input number is 234, possible words which can be formed are (Alphabetical order):
adg adh adi aeg aeh aei afg afh afi bdg bdh bdi beg beh bei bfg bfh bfi cdg cdh cdi ceg ceh cei cfg cfh cfi

Let’s do some calculations first. How many words are possible with seven digits with each digit representing n letters? For first digit we have at most four choices, and for each choice for first letter, we have at most four choices for second digit and so on. So it’s simple maths it will be $O(4^n)$. Since keys 0 and 1 don’t have any corresponding alphabet and many characters have 3 characters, 4^n would be the upper bound of number of words and not the minimum words.

Now let’s do some examples.
For number above 234. Do you see any pattern? Yes, we notice that the last character always either G,H or I and whenever it resets its value from I to G, the digit at the left of it gets changed.
Similarly whenever the second last alphabet resets its value, the third last alphabet gets changes and so on. First character resets only once when we have generated all words. This can be looked from other end also. That is to say whenever character at position i changes, character at position i+1 goes through all possible characters and it creates ripple effect till we reach at end.
Since 0 and 1 don’t have any characters associated with them. we should break as there will no iteration for these digits.
Let’s take the second approach as it will be easy to implement it using recursion. We go till the end and come back one by one. Perfect condition for recursion. let’s search for base case.

When we reach at the last character, we print the word with all possible characters for last digit and return. Simple base case.
When we reach at the last character, we print the word with all possible characters for last digit and return. Simple base case.

Following is C implementation of recursive approach to print all possible word corresponding to a n digit input number. Note that input number is represented as an array to simplify the code.

```
#include <stdio.h>
#include <string.h>

// hashTable[i] stores all characters that correspond to digit i in ph
const char hashTable[10][5] = {"", "", "abc", "def", "ghi", "jkl",
                               "mno", "pqrs", "tuv", "wxyz"};

// A recursive function to print all possible words that can be obtain
// by input number[] of size n.  The output words are one by one store
// in output[]
void  printWordsUtil(int number[], int curr_digit, char output[], int
{
    // Base case, if current output word is prepared
    int i;
    if (curr_digit == n)
    {
        printf("%s ", output);
        return ;
    }

    // Try all 3 possible characters for current digir in number[]
    // and recur for remaining digits
    for (i=0; i<strlen(hashTable[number[curr_digit]]); i++)
    {
        output[curr_digit] = hashTable[number[curr_digit]][i];
        printWordsUtil(number, curr_digit+1, output, n);
        if (number[curr_digit] == 0 || number[curr_digit] == 1)
            return;
    }
}

// A wrapper over printWordsUtil().  It creates an output array and
// calls printWordsUtil()
void printWords(int number[], int n)
{
    char result[n+1];
    result[n] ='\0';
    printWordsUtil(number, 0, result, n);
}

//Driver program
int main(void)
{
    int number[] = {2, 3, 4};
    int n = sizeof(number)/sizeof(number[0]);
    printWords(number, n);
    return 0;
}
```

Output:

```
adg adh adi aeg aeh aei afg afh afi bdg
bdh bdi beg beh bei bfg bfh bfi cdg cdh
cdi ceg ceh cei cfg cfh cfi
Process returned 0 (0x0)   execution time : 0.025 s
Press any key to continue.
```

Time Complexity: Time complexity of above code is $O(4^n)$ where n is number of digits in input number.

Reference:
Buy Programming Interviews Exposed: Secrets to Landing Your Next Job 3rd Edition from Flipkart.com

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

46. Check if a given string is a rotation of a palindrome

Given a string, check if it is a rotation of a palindrome. For example your function should return true for “aab” as it is a rotation of “aba”.

Examples:

```
Input:  str = "aaaad"
Output: 1
// "aaaad" is a rotation of a palindrome "aadaa"

Input:  str = "abcd"
Output: 0
// "abcd" is not a rotation of any palindrome.
```

A **Simple Solution** is to take the input string, try every possible rotation of it and return true if a rotation is a palindrome. If no rotation is palindrome, then return false. Following is C++ implementation of this approach.

```
#include<iostream>
#include<string>
using namespace std;

// A utility function to check if a string str is palindrome
bool isPalindrome(string str)
{
    // Start from leftmost and rightmost corners of str
    int l = 0;
    int h = str.length() - 1;

    // Keep comparing characters while they are same
    while (h > l)
        if (str[l++] != str[h--])
            return false;

    // If we reach here, then all characters were matching
    return true;
}

// Function to chech if a given string is a rotation of a
// palindrome.
bool isRotationOfPalindrome(string str)
{
   // If string iteself is palindrome
   if (isPalindrome(str))
         return true;

   // Now try all rotations one by one
   int n = str.length();
   for (int i = 0; i < n-1; i++)
   {
       string str1 = str.substr(i+1, n-i-1);
       string str2 = str.substr(0, i+1);

       // Check if this rotation is palindrome
       if (isPalindrome(str1.append(str2)))
          return true;
   }
   return false;
}

// Driver program to test above function
int main()
{
    cout << isRotationOfPalindrome("aab") << endl;
    cout << isRotationOfPalindrome("abcde") << endl;
    cout << isRotationOfPalindrome("aaaad") << endl;
    return 0;
}
```

Output:

```
1
0
1
```

Time Complexity: $O(n^2)$
Auxiliary Space: O(n) for storing rotations.

Note that the above algorithm can be optimized to work in O(1) extra space as we can rotate a string in O(n) time and O(1) extra space.

An **Optimized Solution** can work in O(n) time. The idea is similar to this post. Following are steps.
1) Let the given string be 'str' and length of string be 'n'. Create a temporary string 'temp' which is of size 2n and contains str followed by str again. For example, let str be "aab", temp would be "aabaab".
2) Now the problem reduces to find a palindromic substring of length n in str. If there is palindromic substring of length n, then return true, else return false.
We can find whether there is a palindromic substring of size n or not in O(n) time (See Longest palindromic substring)

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

47. Count Possible Decodings of a given Digit Sequence

Let 1 represent 'A', 2 represents 'B', etc. Given a digit sequence, count the number of possible decodings of the given digit sequence.

Examples:

```
Input:  digits[] = "121"
Output: 3
// The possible decodings are "ABA", "AU", "LA"

Input: digits[] = "1234"
Output: 3
// The possible decodings are "ABCD", "LCD", "AWD"
```

An empty digit sequence is considered to have one decoding. It may be assumed that the input contains valid digits from 0 to 9 and there are no leading 0's, no extra trailing 0's and no two or more consecutive 0's.

We strongly recommend to minimize the browser and try this yourself first.

This problem is recursive and can be broken in sub-problems. We start from end of the given digit sequence. We initialize the total count of decodings as 0. We recur for two subproblems.
1) If the last digit is non-zero, recur for remaining (n-1) digits and add the result to total count.
2) If the last two digits form a valid character (or smaller than 27), recur for remaining (n-

2) digits and add the result to total count.

Following is C++ implementation of the above approach.

```
// A naive recursive C++ implementation to count number of decodings
// that can be formed from a given digit sequence
#include <iostream>
#include <cstring>
using namespace std;

// Given a digit sequence of length n, returns count of possible
// decodings by replacing 1 with A, 2 woth B, ... 26 with Z
int countDecoding(char *digits, int n)
{
    // base cases
    if (n == 0 || n == 1)
        return 1;

    int count = 0;  // Initialize count

    // If the last digit is not 0, then last digit must add to
    // the number of words
    if (digits[n-1] > '0')
        count =  countDecoding(digits, n-1);

    // If the last two digits form a number smaller than or equal to 2
    // then consider last two digits and recur
    if (digits[n-2] < '2' || (digits[n-2] == '2' && digits[n-1] < '7'))
        count +=  countDecoding(digits, n-2);

    return count;
}

// Driver program to test above function
int main()
{
    char digits[] = "1234";
    int n = strlen(digits);
    cout << "Count is " << countDecoding(digits, n);
    return 0;
}
```

Output:

```
Count is 3
```

The time complexity of above the code is exponential. If we take a closer look at the above program, we can observe that the recursive solution is similar to Fibonacci Numbers. Therefore, we can optimize the above solution to work in O(n) time using Dynamic Programming. Following is C++ implementation for the same.

```
// A Dynamic Programming based C++ implementation to count decodings
#include <iostream>
#include <cstring>
using namespace std;

// A Dynamic Programming based function to count decodings
int countDecodingDP(char *digits, int n)
{
    int count[n+1]; // A table to store results of subproblems
    count[0] = 1;
    count[1] = 1;

    for (int i = 2; i <= n; i++)
    {
        count[i] = 0;

        // If the last digit is not 0, then last digit must add to
        // the number of words
        if (digits[i-1] > '0')
            count[i] = count[i-1];

        // If second last digit is smaller than 2 and last digit is
        // smaller than 7, then last two digits form a valid character
        if (digits[i-2] < '2' || (digits[i-2] == '2' && digits[i-1] <
            count[i] += count[i-2];
    }
    return count[n];
}

// Driver program to test above function
int main()
{
    char digits[] = "1234";
    int n = strlen(digits);
    cout << "Count is " << countDecodingDP(digits, n);
    return 0;
}
```

Output:

```
Count is 3
```

Time Complexity of the above solution is O(n) and it requires O(n) auxiliary space. We can reduce auxiliary space to O(1) by using space optimized version discussed in the Fibonacci Number Post.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

48. Find Excel column name from a given column number

MS Excel columns has a pattern like A, B, C, … ,Z, AA, AB, AC,…. ,AZ, BA, BB, … ZZ,

AAA, AAB ….. etc. In other words, column 1 is named as “A”, column 2 as “B”, column 27 as “AA”.

Given a column number, find its corresponding Excel column name. Following are more examples.

```
Input          Output
 26             Z
 51             AY
 52             AZ
 80             CB
 676            YZ
 702            ZZ
 705            AAC
```

We strongly recommend to minimize the browser and try this yourself first.

Thanks to Mrigank Dembla for suggesting the below solution in a comment.

Suppose we have a number n, let’s say 28. so corresponding to it we need to print the column name. We need to take remainder with 26.

If remainder with 26 comes out to be 0 (meaning 26, 52 and so on) then we put ‘Z’ in the output string and new n becomes n/26 -1 because here we are considering 26 to be ‘Z’ while in actual it’s 25th with respect to ‘A’.

Similarly if the remainder comes out to be non zero. (like 1, 2, 3 and so on) then we need to just insert the char accordingly in the string and do n = n/26.

Finally we reverse the string and print.

Example:
n = 700

Remainder (n%26) is 24. So we put ‘X’ in output string and n becomes n/26 which is 26.

Remainder (26%26) is 0. So we put ‘Z’ in output string and n becomes n/26 -1 which is 0.

Following is C++ implementation of above approach.

```
#include<iostream>
#include<cstring>
#define MAX 50
using namespace std;

// Function to print Excel column name for a given column number
void printString(int n)
{
    char str[MAX];  // To store result (Excel column name)
    int i = 0;  // To store current index in str which is result

    while (n>0)
    {
        // Find remainder
        int rem = n%26;

        // If remainder is 0, then a 'Z' must be there in output
        if (rem==0)
        {
            str[i++] = 'Z';
            n = (n/26)-1;
        }
        else // If remainder is non-zero
        {
            str[i++] = (rem-1) + 'A';
            n = n/26;
        }
    }
    str[i] = '\0';

    // Reverse the string and print result
    cout << strrev(str) << endl;

    return;
}

// Driver program to test above function
int main()
{
    printString(26);
    printString(51);
    printString(52);
    printString(80);
    printString(676);
    printString(702);
    printString(705);
    return 0;
}
```

Output

```
Z
AY
AZ
CB
YZ
ZZ
AAC
```

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

49. Anagram Substring Search (Or Search for all permutations)

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] and its permutations (or anagrams) in txt[]. You may assume that n > m.
Expected time complexity is O(n)

Examples:

```
1) Input:  txt[] = "BACDGABCDA"  pat[] = "ABCD"
   Output:   Found at Index 0
             Found at Index 5
             Found at Index 6
2) Input: txt[] =  "AAABABAA" pat[] = "AABA"
   Output:   Found at Index 0
             Found at Index 1
             Found at Index 4
```

This problem is slightly different from standard pattern searching problem, here we need to search for anagrams as well. Therefore, we cannot directly apply standard pattern searching algorithms like KMP, Rabin Karp, Boyer Moore, etc.

A simple idea is to modify Rabin Karp Algorithm. For example we can keep the hash value as sum of ASCII values of all characters under modulo of a big prime number. For every character of text, we can add the current character to hash value and subtract the first character of previous window. This solution looks good, but like standard Rabin Karp, the worst case time complexity of this solution is O(mn). The worst case occurs when all hash values match and we one by one match all characters.

We can achieve O(n) time complexity under the assumption that alphabet size is fixed which is typically true as we have maximum 256 possible characters in ASCII. The idea is to use two count arrays:

1) The first count array store frequencies of characters in pattern.
2) The second count array stores frequencies of characters in current window of text.

The important thing to note is, time complexity to compare two count arrays is O(1) as the number of elements in them are fixed (independent of pattern and text sizes). Following are steps of this algorithm.
1) Store counts of frequencies of pattern in first count array *countP[]*. Also store counts

of frequencies of characters in first window of text in array *countTW[]*.

2) Now run a loop from i = M to N-1. Do following in loop.
…..a) If the two count arrays are identical, we found an occurrence.
…..b) Increment count of current character of text in countTW[]
…..c) Decrement count of first character in previous window in countWT[]

3) The last window is not checked by above loop, so explicitly check it.

Following is C++ implementation of above algorithm.

```
// C++ program to search all anagrams of a pattern in a text
#include<iostream>
#include<cstring>
#define MAX 256
using namespace std;

// This function returns true if contents of arr1[] and arr2[]
// are same, otherwise false.
bool compare(char arr1[], char arr2[])
{
    for (int i=0; i<MAX; i++)
        if (arr1[i] != arr2[i])
            return false;
    return true;
}

// This function search for all permutations of pat[] in txt[]
void search(char *pat, char *txt)
{
    int M = strlen(pat), N = strlen(txt);

    // countP[]:  Store count of all characters of pattern
    // countTW[]: Store count of current window of text
    char countP[MAX] = {0}, countTW[MAX] = {0};
    for (int i = 0; i < M; i++)
    {
        (countP[pat[i]])++;
        (countTW[txt[i]])++;
    }

    // Traverse through remaining characters of pattern
    for (int i = M; i < N; i++)
    {
        // Compare counts of current window of text with
        // counts of pattern[]
        if (compare(countP, countTW))
            cout << "Found at Index " << (i - M) << endl;

        // Add current character to current window
        (countTW[txt[i]])++;

        // Remove the first character of previous window
        countTW[txt[i-M]]--;
    }

    // Check for the last window in text
    if (compare(countP, countTW))
        cout << "Found at Index " << (N - M) << endl;
}

/* Driver program to test above function */
int main()
{
    char txt[] = "BACDGABCDA";
    char pat[] = "ABCD";
    search(pat, txt);
    return 0;
}
```

Output:

```
Found at Index 0
```

```
Found at Index 5
Found at Index 6
```

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

50. Given a sorted dictionary of an alien language, find order of characters

Given a sorted dictionary (array of words) of an alien language, find order of characters in the language.

Examples:

```
Input:  words[] = {"baa", "abcd", "abca", "cab", "cad"}
Output: Order of characters is 'b', 'd', 'a', 'c'
Note that words are sorted and in the given language "baa"
comes before "abcd", therefore 'b' is before 'a' in output.
Similarly we can find other orders.

Input:  words[] = {"caa", "aaa", "aab"}
Output: Order of characters is 'c', 'a', 'b'
```

We strongly recommend to minimize the browser and try this yourself first.

The idea is to create a graph of characters and then find topological sorting of the created graph. Following are the detailed steps.

1) Create a graph *g* with number of vertices equal to the size of alphabet in the given alien language. For example, if the alphabet size is 5, then there can be 5 characters in words. Initially there are no edges in graph.

2) Do following for every pair of adjacent words in given sorted array.
…..a) Let the current pair of words be *word1* and *word2*. One by one compare characters of both words and find the first mismatching characters.
…..b) Create an edge in *g* from mismatching character of *word1* to that of *word2*.

3) Print topological sorting of the above created graph.

Following is C++ implementation of the above algorithm.

```
// A C++ program to order of characters in an alien language
#include<iostream>
#include <list>
#include <stack>
```

```
#include <cstring>
using namespace std;

// Class to represent a graph
class Graph
{
    int V;    // No. of vertices'

    // Pointer to an array containing adjacency listsList
    list<int> *adj;

    // A function used by topologicalSort
    void topologicalSortUtil(int v, bool visited[], stack<int> &Stack)
public:
    Graph(int V);   // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints a Topological Sort of the complete graph
    void topologicalSort();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

// A recursive function used by topologicalSort
void Graph::topologicalSortUtil(int v, bool visited[], stack<int> &Sta
{
    // Mark the current node as visited.
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            topologicalSortUtil(*i, visited, Stack);

    // Push current vertex to stack which stores result
    Stack.push(v);
}

// The function to do Topological Sort. It uses recursive topologicalS
void Graph::topologicalSort()
{
    stack<int> Stack;

    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store Topological Sort
    // starting from all vertices one by one
    for (int i = 0; i < V; i++)
```

```
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            topologicalSortUtil(i, visited, Stack);

    // Print contents of stack
    while (Stack.empty() == false)
    {
        cout << (char) ('a' + Stack.top()) << " ";
        Stack.pop();
    }
}

int min(int x, int y)
{
    return (x < y)? x : y;
}

// This function fidns and prints order of characer from a sorted
// array of words. n is size of words[].  alpha is set of possible
// alphabets.
// For simplicity, this function is written in a way that only
// first 'alpha' characters can be there in words array.  For
// example if alpha is 7, then words[] should have only 'a', 'b',
// 'c' 'd', 'e', 'f', 'g'
void printOrder(string words[], int n, int alpha)
{
    // Create a graph with 'aplha' edges
    Graph g(alpha);

    // Process all adjacent pairs of words and create a graph
    for (int i = 0; i < n-1; i++)
    {
        // Take the current two words and find the first mismatching
        // character
        string word1 = words[i], word2 = words[i+1];
        for (int j = 0; j < min(word1.length(), word2.length()); j++)
        {
            // If we find a mismatching character, then add an edge
            // from character of word1 to that of word2
            if (word1[j] != word2[j])
            {
                g.addEdge(word1[j]-'a', word2[j]-'a');
                break;
            }
        }
    }

    // Print topological sort of the above created graph
    g.topologicalSort();
}

// Driver program to test above functions
int main()
{
    string words[] = {"caa", "aaa", "aab"};
    printOrder(words, 3, 3);
    return 0;
}
```

Output:

```
c a b
```

Time Complexity: The first step to create a graph takes O(n + alhpa) time where n is number of given words and alpha is number of characters in given alphabet. The second step is also topological sorting. Note that there would be *alpha* vertices and at-most (n-1) edges in the graph. The time complexity of topological sorting is O(V+E) which is O(n + aplha) here. So overall time complexity is O(n + aplha) + O(n + aplha) which is O(n + aplha).

Exercise:
The above code doesn't work when the input is not valid. For example {"aba", "bba", "aaa"} is not valid, because from first two words, we can deduce 'a' should appear before 'b', but from last two words, we can deduce 'b' should appear before 'a' which is not possible. Extend the above program to handle invalid inputs and generate the output as "Not valid".

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

51. Given an array of strings, find if the strings can be chained to form a circle

Given an array of strings, find if the given strings can be chained to form a circle. A string X can be put before another string Y in circle if the last character of X is same as first character of Y.

Examples:

```
Input: arr[] = {"geek", "king"}
Output: Yes, the given strings can be chained.
Note that the last character of first string is same
as first character of second string and vice versa is
also true.

Input: arr[] = {"for", "geek", "rig", "kaf"}
Output: Yes, the given strings can be chained.
The strings can be chained as "for", "rig", "geek"
and "kaf"

Input: arr[] = {"aab", "bac", "aaa", "cda"}
Output: Yes, the given strings can be chained.
The strings can be chained as "aaa", "aab", "bac"
```

```
and "cda"

Input: arr[] = {"aaa", "bbb", "baa", "aab"};
Output: Yes, the given strings can be chained.
The strings can be chained as "aaa", "aab", "bbb"
and "baa"

Input: arr[] = {"aaa"};
Output: Yes

Input: arr[] = {"aaa", "bbb"};
Output: No
```

We strongly recommend to minimize the browser and try this yourself first.

The idea is to create a directed graph of all characters and then find if their is an eulerian circuit in the graph or not. If there is an eulerian circuit, then chain can be formed, otherwise not.
Note that a directed graph has eulerian circuit only if in degree and out degree of every vertex is same, and all non-zero degree vertices form a single strongly connected component.

Following are detailed steps of the algorithm.

1) Create a directed graph g with number of vertices equal to the size of alphabet. We have created a graph with 26 vertices in the below program.

2) Do following for every string in the given array of strings.
…..a) Add an edge from first character to last character of the given graph.

3) If the created graph has eulerian circuit, then return true, else return false.

Following is C++ implementation of the above algorithm.

```
// A C++ program to check if a given directed graph is Eulerian or not
#include<iostream>
#include <list>
#define CHARS 26
using namespace std;

// A class that represents an undirected graph
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
    int *in;
public:
    // Constructor and destructor
    Graph(int V);
    ~Graph()   { delete [] adj; delete [] in; }

    // function to add an edge to graph
    void addEdge(int v, int w) { adj[v].push_back(w);  (in[w])++; }
```

```
    // Method to check if this graph is Eulerian or not
    bool isEulerianCycle();

    // Method to check if all non-zero degree vertices are connected
    bool isSC();

    // Function to do DFS starting from v. Used in isConnected();
    void DFSUtil(int v, bool visited[]);

    Graph getTranspose();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
    in = new int[V];
    for (int i = 0; i < V; i++)
       in[i] = 0;
}

/* This function returns true if the directed graph has an eulerian
   cycle, otherwise returns false  */
bool Graph::isEulerianCycle()
{
    // Check if all non-zero degree vertices are connected
    if (isSC() == false)
        return false;

    // Check if in degree and out degree of every vertex is same
    for (int i = 0; i < V; i++)
        if (adj[i].size() != in[i])
            return false;

    return true;
}

// A recursive function to do DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);
}

// Function that returns reverse (or transpose) of this graph
// This function is needed in isSC()
Graph Graph::getTranspose()
{
    Graph g(V);
    for (int v = 0; v < V; v++)
    {
        // Recur for all the vertices adjacent to this vertex
        list<int>::iterator i;
        for(i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            g.adj[*i].push_back(v);
```

```
            g.adj[u].push_back(v);
            (g.in[v])++;
        }
    }
    return g;
}

// This function returns true if all non-zero degree vertices of
// graph are strongly connected. Please refer
// http://www.geeksforgeeks.org/connectivity-in-a-directed-graph/
bool Graph::isSC()
{
    // Mark all the vertices as not visited (For first DFS)
    bool visited[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Find the first vertex with non-zero degree
    int n;
    for (n = 0; n < V; n++)
        if (adj[n].size() > 0)
          break;

    // Do DFS traversal starting from first non zero degree vertex.
    DFSUtil(n, visited);

     // If DFS traversal doesn't visit all vertices, then return false
    for (int i = 0; i < V; i++)
        if (adj[i].size() > 0 && visited[i] == false)
              return false;

    // Create a reversed graph
    Graph gr = getTranspose();

    // Mark all the vertices as not visited (For second DFS)
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Do DFS for reversed graph starting from first vertex.
    // Staring Vertex must be same starting point of first DFS
    gr.DFSUtil(n, visited);

    // If all vertices are not visited in second DFS, then
    // return false
    for (int i = 0; i < V; i++)
        if (adj[i].size() > 0 && visited[i] == false)
             return false;

    return true;
}

// This function takes an of strings and returns true
// if the given array of strings can be chained to
// form cycle
bool canBeChained(string arr[], int n)
{
    // Create a graph with 'aplha' edges
    Graph g(CHARS);

    // Create an edge from first character to last character
    // of every string
    for (int i = 0; i < n; i++)
    {
```

```
    {
        string s = arr[i];
        g.addEdge(s[0]-'a', s[s.length()-1]-'a');
    }

    // The given array of strings can be chained if there
    // is an eulerian cycle in the created graph
    return g.isEulerianCycle();
}

// Driver program to test above functions
int main()
{
    string arr1[] =  {"for", "geek", "rig", "kaf"};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);
    canBeChained(arr1, n1)?  cout << "Can be chained \n" :
                             cout << "Can't be chained \n";

    string arr2[] =  {"aab", "abb"};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);
    canBeChained(arr2, n2)?  cout << "Can be chained \n" :
                             cout << "Can't be chained \n";

    return 0;
}
```

Output:

```
Can be chained
Can't be chained
```

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

52. Pattern Searching using a Trie of all Suffixes

Problem Statement: Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

As discussed in the previous post, we discussed that there are two ways efficiently solve the above problem.

1) Preprocess Pattern: KMP Algorithm, Rabin Karp Algorithm, Finite Automata, Boyer Moore Algorithm.

2) Preoprocess Text: Suffix Tree

The best possible time complexity achieved by first (preprocssing pattern) is O(n) and by second (preprocessing text) is O(m) where m and n are lengths of pattern and text

respectively.

Note that the second way does the searching only in O(m) time and it is preferred when text doesn't doesn't change very frequently and there are many search queries. We have discussed Suffix Tree (A compressed Trie of all suffixes of Text) .

Implementation of Suffix Tree may be time consuming for problems to be coded in a technical interview or programming contexts. In this post simple implementation of a Standard Trie of all Suffixes is discussed. The implementation is close to suffix tree, the only thing is, it's a simple Trie instead of compressed Trie.

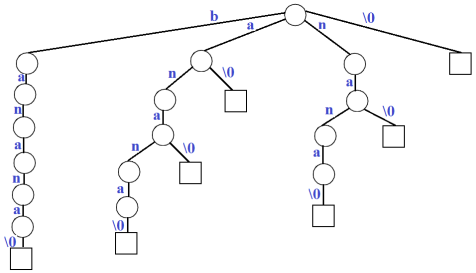
As discussed in Suffix Tree post, the idea is, every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. So if we build a Trie of all suffixes, we can find the pattern in O(m) time where m is pattern length.

Building a Trie of Suffixes
1) Generate all suffixes of given text.
2) Consider all suffixes as individual words and build a trie.

Let us consider an example text "banana\0" where '\0' is string termination character. Following are all suffixes of "banana\0"

```
banana\0
anana\0
nana\0
ana\0
na\0
a\0
\0
```

If we consider all of the above suffixes as individual words and build a Trie, we get following.



How to search a pattern in the built Trie?
Following are steps to search a pattern in the built Trie.
1) Starting from the first character of the pattern and root of the Trie, do following for every character.
…..**a)** For the current character of pattern, if there is an edge from the current node, follow the edge.

…..**b)** If there is no edge, print “pattern doesn’t exist in text” and return.
2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print print all indexes where pattern is present. To store indexes, we use a list with every node that stores indexes of suffixes starting at the node.

Following is C++ implementation of the above idea.

```
// A simple C++ implementation of substring search using trie of suffi
#include<iostream>
#include<list>
#define MAX_CHAR 256
using namespace std;

// A Suffix Trie (A Trie of all suffixes) Node
class SuffixTrieNode
{
private:
    SuffixTrieNode *children[MAX_CHAR];
    list<int> *indexes;
public:
    SuffixTrieNode() // Constructor
    {
        // Create an empty linked list for indexes of
        // suffixes starting from this node
        indexes = new list<int>;

        // Initialize all child pointers as NULL
        for (int i = 0; i < MAX_CHAR; i++)
          children[i] = NULL;
    }

    // A recursive function to insert a suffix of the txt
    // in subtree rooted with this node
    void insertSuffix(string suffix, int index);

    // A function to search a pattern in subtree rooted
    // with this node.The function returns pointer to a linked
    // list containing all indexes where pattern is present.
    // The returned indexes are indexes of last characters
    // of matched text.
    list<int>* search(string pat);
};

// A Trie of all suffixes
class SuffixTrie
{
private:
    SuffixTrieNode root;
public:
    // Constructor (Builds a trie of suffies of the given text)
    SuffixTrie(string txt)
    {
        // Consider all suffixes of given string and insert
        // them into the Suffix Trie using recursive function
        // insertSuffix() in SuffixTrieNode class
        for (int i = 0; i < txt.length(); i++)
            root.insertSuffix(txt.substr(i), i);
    }
```

```
    // Function to searches a pattern in this suffix trie.
    void search(string pat);
};

// A recursive function to insert a suffix of the txt in
// subtree rooted with this node
void SuffixTrieNode::insertSuffix(string s, int index)
{
    // Store index in linked list
    indexes->push_front(index);

    // If string has more characters
    if (s.length() > 0)
    {
        // Find the first character
        char cIndex = s.at(0);

        // If there is no edge for this character, add a new edge
        if (children[cIndex] == NULL)
            children[cIndex] = new SuffixTrieNode();

        // Recur for next suffix
        children[cIndex]->insertSuffix(s.substr(1), index+1);
    }
}

// A recursive function to search a pattern in subtree rooted with
// this node
list<int>* SuffixTrieNode::search(string s)
{
    // If all characters of pattern have been processed,
    if (s.length() == 0)
        return indexes;

    // if there is an edge from the current node of suffix trie,
    // follow the edge.
    if (children[s.at(0)] != NULL)
        return (children[s.at(0)])->search(s.substr(1));

    // If there is no edge, pattern doesn't exist in text
    else return NULL;
}

/* Prints all occurrences of pat in the Suffix Trie S (built for text)
void SuffixTrie::search(string pat)
{
    // Let us call recursive search function for root of Trie.
    // We get a list of all indexes (where pat is present in text) in
    // variable 'result'
    list<int> *result = root.search(pat);

    // Check if the list of indexes is empty or not
    if (result == NULL)
        cout << "Pattern not found" << endl;
    else
    {
       list<int>::iterator i;
       int patLen = pat.length();
       for (i = result->begin(); i != result->end(); ++i)
         cout << "Pattern found at position " << *i - patLen<< endl;
    }
}
```

```
// driver program to test above functions
int main()
{
    // Let us build a suffix trie for text "geeksforgeeks.org"
    string txt = "geeksforgeeks.org";
    SuffixTrie S(txt);

    cout << "Search for 'ee'" << endl;
    S.search("ee");

    cout << "\nSearch for 'geek'" << endl;
    S.search("geek");

    cout << "\nSearch for 'quiz'" << endl;
    S.search("quiz");

    cout << "\nSearch for 'forgeeks'" << endl;
    S.search("forgeeks");

    return 0;
}
```

Output:

```
Search for 'ee'
Pattern found at position 9
Pattern found at position 1

Search for 'geek'
Pattern found at position 8
Pattern found at position 0

Search for 'quiz'
Pattern not found

Search for 'forgeeks'
Pattern found at position 5
```

Time Complexity of the above search function is O(m+k) where m is length of the pattern and k is the number of occurrences of pattern in text.

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

53. Given two strings, find if first string is a subsequence of second

Given two strings str1 and str2, find if str1 is a subsequence of str2. A subsequence is a

sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements (source: wiki). Expected time complexity is linear.

Examples:

```
Input: str1 = "AXY", str2 = "ADXCPY"
Output: True (str1 is a subsequence of str2)

Input: str1 = "AXY", str2 = "YADXCP"
Output: False (str1 is not a subsequence of str2)

Input: str1 = "gksrek", str2 = "geeksforgeeks"
Output: True (str1 is a subsequence of str2)
```

We strongly recommend to minimize the browser and try this yourself first.

The idea is simple, we traverse both strings from one side to other side (say from rightmost character to leftmost). If we find a matching character, we move ahead in both strings. Otherwise we move ahead only in str2.

Following is **Recursive Implementation** in C++ of the above idea.

```
// Recursive C++ program to check if a string is subsequence of anothe
#include<iostream>
#include<cstring>
using namespace std;

// Returns true if str1[] is a subsequence of str2[]. m is
// length of str1 and n is length of str2
bool isSubSequence(char str1[], char str2[], int m, int n)
{
    // Base Cases
    if (m == 0) return true;
    if (n == 0) return false;

    // If last characters of two strings are matching
    if (str1[m-1] == str2[n-1])
        return isSubSequence(str1, str2, m-1, n-1);

    // If last characters are not matching
    return isSubSequence(str1, str2, m, n-1);
}

// Driver program to test methods of graph class
int main()
{
    char str1[] = "gksrek";
    char str2[] = "geeksforgeeks";
    int m = strlen(str1);
    int n = strlen(str2);
    isSubSequence(str1, str2, m, n)? cout << "Yes ":
                                     cout << "No";
    return 0;
}
```

Output:

```
Yes
```

Following is **Iterative Implementation** in C++ for the same.

```
// Iterative C++ program to check if a string is subsequence of anothe
#include<iostream>
#include<cstring>
using namespace std;

// Returns true if str1[] is a subsequence of str2[]. m is
// length of str1 and n is length of str2
bool isSubSequence(char str1[], char str2[], int m, int n)
{
   int j = 0; // For index of str1 (or subsequence

   // Traverse str2 and str1, and compare current character
   // of str2 with first unmatched char of str1, if matched
   // then move ahead in str1
   for (int i=0; i<n&&j<m; i++)
       if (str1[j] == str2[i])
         j++;

   // If all characters of str1 were found in str2
   return (j==m);
}

// Driver program to test methods of graph class
int main()
{
    char str1[] = "gksrek";
    char str2[] = "geeksforgeeks";
    int m = strlen(str1);
    int n = strlen(str2);
    isSubSequence(str1, str2, m, n)? cout << "Yes ":
                                     cout << "No";
    return 0;
}
```

Output:

```
Yes
```

Time Complexity of both implementations above is O(n) where n is the length of str2.

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

54. Ukkonen's Suffix Tree Construction – Part 1

Suffix Tree is very useful in numerous string processing and computational biology

problems. Many books and e-resources talk about it theoretically and in few places, code implementation is discussed. But still, I felt something is missing and it's not easy to implement code to construct suffix tree and it's usage in many applications. This is an attempt to bridge the gap between theory and complete working code implementation. Here we will discuss Ukkonen's Suffix Tree Construction Algorithm. We will discuss it in step by step detailed way and in multiple parts from theory to implementation. We will start with brute force way and try to understand different concepts, tricks involved in Ukkonen's algorithm and in the last part, code implementation will be discussed.

Note: You may find some portion of the algorithm difficult to understand while 1st or 2nd reading and it's perfectly fine. With few more attempts and thought, you should be able to understand such portions.

Book Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology by **Dan Gusfield** explains the concepts very well.

A suffix tree **T** for a m-character string S is a rooted directed tree with exactly m leaves numbered 1 to **m.** (Given that last string character is unique in string)

- Root can have zero, one or more children.
- Each internal node, other than the root, has at least two children.
- Each edge is labelled with a nonempty substring of S.
- No two edges coming out of same node can have edge-labels beginning with the same character.

Concatenation of the edge-labels on the path from the root to leaf i gives the suffix of S that starts at position i, i.e. S[i…m].

Note: Position starts with 1 (it's not zero indexed, but later, while code implementation, we will used zero indexed position)

For string S = xabxac with m = 6, suffix tree will look like following:

It has one root node and two internal nodes and 6 leaf nodes.

String Depth of red path is 1 and it represents suffix c starting at position 6
String Depth of blue path is 4 and it represents suffix bxca starting at position 3
String Depth of green path is 2 and it represents suffix ac starting at position 5
String Depth of orange path is 6 and it represents suffix xabxac starting at position 1

Edges with labels a (green) and xa (orange) are non-leaf edge (which ends at an internal node). All other edges are leaf edge (ends at a leaf)

If one suffix of S matches a prefix of another suffix of S (when last character in not unique in string), then path for the first suffix would not end at a leaf.

For String S = xabxa, with m = 5, following is the suffix tree:

Here we will have 5 suffixes: xabxa, abxa, bxa, xa and a.
Path for suffixes ‘xa’ and ‘a’ do not end at a leaf. A tree like above (Figure 2) is called implicit suffix tree as some suffixes (‘xa’ and ‘a’) are not seen explicitly in tree.

To avoid this problem, we add a character which is not present in string already. We normally use $, # etc as termination characters.
Following is the suffix tree for string S = xabxa$ with m = 6 and now all 6 suffixes end at leaf.

A naive algorithm to build a suffix tree
Given a string S of length m, enter a single edge for suffix S[l ..m]$ (the entire string) into the tree, then successively enter suffix S[i..m]$ into the growing tree, for i increasing from 2 to m. Let N_i denote the intermediate tree that encodes all the suffixes from 1 to i. So N_i+1 is constructed from N_i as follows:

- Start at the root of N_i
- Find the longest path from the root which matches a prefix of S[i+1..m]$
- Match ends either at the node (say w) or in the middle of an edge [say (u, v)].
- If it is in the middle of an edge (u, v), break the edge (u, v) into two edges by inserting a new node w just after the last character on the edge that matched a character in S[i+l..m] and just before the first character on the edge that mismatched. The new edge (u, w) is labelled with the part of the (u, v) label that matched with S[i+1..m], and the new edge (w, v) is labelled with the remaining part of the (u, v) label.
- Create a new edge (w, i+1) from w to a new leaf labelled i+1 and it labels the new edge with the unmatched part of suffix S[i+1..m]

This takes $O(m^2)$ to build the suffix tree for the string S of length m.
Following are few steps to build suffix tree based for string “xabxa$” based on above algorithm:

Implicit suffix tree
While generating suffix tree using Ukkonen’s algorithm, we will see implicit suffix tree in intermediate steps few times depending on characters in string S. In implicit suffix trees, there will be no edge with $ (or # or any other termination character) label and no internal node with only one edge going out of it.

To get implicit suffix tree from a suffix tree S$,

- Remove all terminal symbol $ from the edge labels of the tree,
- Remove any edge that has no label
- Remove any node that has only one edge going out of it and merge the edges.

High Level Description of Ukkonen's algorithm

Ukkonen's algorithm constructs an implicit suffix tree T_i for each prefix S[l ..i] of S (of length m).

It first builds T_1 using 1st character, then T_2 using 2nd character, then T_3 using 3rd character, …, T_m using mth character.

Implicit suffix tree T_i+1 is built on top of implicit suffix tree T_i.

The true suffix tree for S is built from T_m by adding $.

At any time, Ukkonen's algorithm builds the suffix tree for the characters seen so far and so it has **on-line** property that may be useful in some situations.

Time taken is O(m).

Ukkonen's algorithm is divided into m phases (one phase for each character in the string with length m)

In phase i+1, tree T_i+1 is built from tree T_i.

Each phase i+1 is further divided into i+1 extensions, one for each of the i+1 suffixes of S[1..i+1]

In extension j of phase i+1, the algorithm first finds the end of the path from the root labelled with substring S[j..i].

It then extends the substring by adding the character S(i+1) to its end (if it is not there already).

In extension 1 of phase i+1, we put string S[1..i+1] in the tree. Here S[1..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already).

In extension 2 of phase i+1, we put string S[2..i+1] in the tree. Here S[2..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already)

In extension 3 of phase i+1, we put string S[3..i+1] in the tree. Here S[3..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already)

.

.

In extension i+1 of phase i+1, we put string S[i+1..i+1] in the tree. This is just one character which may not be in tree (if character is seen first time so far). If so, we just add a new leaf edge with label S[i+1].

High Level Ukkonen's algorithm

```
Construct tree T1
For i from 1 to m-1 do
begin {phase i+1}
    For j from 1 to i+1
        begin {extension j}
        Find the end of the path from the root labelled S[j..i] in the current tree.
        Extend that path by adding character S[i+l] if it is not there already
    end;
end;
```

Suffix extension is all about adding the next character into the suffix tree built so far.
In extension j of phase i+1, algorithm finds the end of S[j..i] (which is already in the tree due to previous phase i) and then it extends S[j..i] to be sure the suffix S[j..i+1] is in the tree.

There are 3 extension rules:
Rule 1: If the path from the root labelled S[j..i] ends at leaf edge (i.e. S[i] is last character on leaf edge) then character S[i+1] is just added to the end of the label on that leaf edge.

Rule 2: If the path from the root labelled S[j..i] ends at non-leaf edge (i.e. there are more characters after S[i] on path) and next character is not s[i+1], then a new leaf edge with label s{i+1] and number j is created starting from character S[i+1].
A new internal node will also be created if s[1..i] ends inside (in-between) a non-leaf edge.

Rule 3: If the path from the root labelled S[j..i] ends at non-leaf edge (i.e. there are more characters after S[i] on path) and next character is s[i+1] (already in tree), do nothing.

One important point to note here is that from a given node (root or internal), there will be one and only one edge starting from one character. There will not be more than one edges going out of any node, starting with same character.

Following is a step by step suffix tree construction of string xabxac using Ukkonen's algorithm:

In next parts (Part 2, Part 3, Part 4 and Part 5), we will discuss suffix links, active points, few tricks and finally code implementations (Part 6).

References:
http://web.stanford.edu/~mjkay/gusfield.pdf

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

55. Ukkonen's Suffix Tree Construction – Part 2

In Ukkonen's Suffix Tree Construction – Part 1, we have seen high level Ukkonen's Algorithm. This 2nd part is continuation of Part 1.
Please go through Part 1, before looking at current article.

In Suffix Tree Construction of string S of length m, there are m phases and for a phase j (1 <= j <= m), we add j^{th} character in tree built so far and this is done through j extensions. All extensions follow one of the three extension rules (discussed in Part 1).

To do j^{th} extension of phase i+1 (adding character S[i+1]), we first need to find end of the path from the root labelled S[j..i] in the current tree. One way is start from root and traverse the edges matching S[j..i] string. This will take $O(m^3)$ time to build the suffix tree. Using few observations and implementation tricks, it can be done in O(m) which we will see now.

Suffix links
For an internal node v with path-label xA, where x denotes a single character and A denotes a (possibly empty) substring, if there is another node s(v) with path-label A, then a pointer from v to s(v) is called a suffix link.
If A is empty string, suffix link from internal node will go to root node.
There will not be any suffix link from root node (As it's not considered as internal node).

In extension j of some phase i, if a new internal node v with path-label xA is added, then in extension j+1 in the same phase i:

- Either the path labelled A already ends at an internal node (or root node if A is empty)
- OR a new internal node at the end of string A will be created

In extension j+1 of same phase i, we will create a suffix link from the internal node created in j^{th} extension to the node with path labelled A.

So in a given phase, any newly created internal node (with path-label xA) will have a

suffix link from it (pointing to another node with path-label A) by the end of the next extension.

In any implicit suffix tree T_i after phase i, if internal node v has path-label xA, then there is a node s(v) in T_i with path-label A and node v will point to node s(v) using suffix link.

At any time, all internal nodes in the changing tree will have suffix links from them to another internal node (or root) except for the most recently added internal node, which will receive its suffix link by the end of the next extension.

How suffix links are used to speed up the implementation?
In extension j of phase i+1, we need to find the end of the path from the root labelled S[j..i] in the current tree. One way is start from root and traverse the edges matching S[j..i] string. Suffix links provide a short cut to find end of the path.

So we can see that, to find end of path S[j..i], we need not traverse from root. We can start from the end of path S[j-1..i], walk up one edge to node v (i.e. go to parent node), follow the suffix link to s(v), then walk down the path y (which is abcd here in Figure 17). This shows the use of suffix link is an improvement over the process.
Note: In the next part 3, we will introduce activePoint which will help to avoid “walk up”. We can directly go to node s(v) from node v.

When there is a suffix link from node v to node s(v), then if there is a path labelled with string y from node v to a leaf, then there must be a path labelled with string y from node s(v) to a leaf. In Figure 17, there is a path label “abcd” from node v to a leaf, then there is a path will same label “abcd” from node s(v) to a leaf.
This fact can be used to improve the walk from s(v) to leaf along the path y. This is called “skip/count” trick.

Skip/Count Trick
When walking down from node s(v) to leaf, instead of matching path character by character as we travel, we can directly skip to the next node if number of characters on the edge is less than the number of characters we need to travel. If number of characters on the edge is more than the number of characters we need to travel, we directly skip to the last character on that edge.
If implementation is such a way that number of characters on any edge, character at a given position in string S should be obtained in constant time, then skip/count trick will do the walk down in proportional to the number of nodes on it rather than the number of characters on it.

Using suffix link along with skip/count trick, suffix tree can be built in $O(m^2)$ as there are

m phases and each phase takes O(m).

Edge-label compression
So far, path labels are represented as characters in string. Such a suffix tree will take $O(m^2)$ space to store the path labels. To avoid this, we can use two pair of indices (start, end) on each edge for path labels, instead of substring itself. The indices start and end tells the path label start and end position in string S. With this, suffix tree needs O(m) space.

There are two observations about the way extension rules interact in successive extensions and phases. These two observations lead to two more implementation tricks (first trick "skip/count" is seen already while walk down).

Observation 1: Rule 3 is show stopper
In a phase i, there are i extensions (1 to i) to be done.
When rule 3 applies in any extension j of phase i+1 (i.e. path labelled S[j..i] continues with character S[i+1]), then it will also apply in all further extensions of same phase (i.e. extensions j+1 to i+1 in phase i+1). That's because if path labelled S[j..i] continues with character S[i+1], then path labelled S[j+1..i], S[j+2..i], S[j+3..i],…, S[i..i] will also continue with character S[i+1].
Consider Figure 11, Figure12 and Figure 13 in Part 1 where Rule 3 is applied.
In Figure 11, "xab" is added in tree and in Figure 12 (Phase 4), we add next character "x". In this, 3 extensions are done (which adds 3 suffixes). Last suffix "x" is already present in tree.
In Figure 13, we add character "a" in tree (Phase 5). First 3 suffixes are added in tree and last two suffixes "xa" and "a" are already present in tree. This shows that if suffix S[j..i] present in tree, then ALL the remaining suffixes S[j+1..i], S[j+2..i], S[j+3..i],…, S[i..i] will also be there in tree and no work needed to add those remaining suffixes.
So no more work needed to be done in any phase as soon as rule 3 applies in any extension in that phase. If a new internal node v gets created in extension j and rule 3 applies in next extension j+1, then we need to add suffix link from node v to current node (if we are on internal node) or root node. ActiveNode, which will be discussed in part 3, will help while setting suffix links.

Trick 2
Stop the processing of any phase as soon as rule 3 applies. All further extensions are already present in tree implicitly.

Observation 2: Once a leaf, always a leaf
Once a leaf is created and labelled j (for suffix starting at position j in string S), then this leaf will always be a leaf in successive phases and extensions. Once a leaf is labelled as j, extension rule 1 will always apply to extension j in all successive phases.
Consider Figure 9 to Figure 14 in Part 1.
In Figure 10 (Phase 2), Rule 1 is applied on leaf labelled 1. After this, in all successive

phases, rule 1 is always applied on this leaf.
In Figure 11 (Phase 3), Rule 1 is applied on leaf labelled 2. After this, in all successive phases, rule 1 is always applied on this leaf.
In Figure 12 (Phase 4), Rule 1 is applied on leaf labelled 3. After this, in all successive phases, rule 1 is always applied on this leaf.

In any phase i, there is an initial sequence of consecutive extensions where rule 1 or rule 2 are applied and then as soon as rule 3 is applied, phase i ends.
Also rule 2 creates a new leaf always (and internal node sometimes).
If J_i represents the last extension in phase i when rule 1 or 2 was applied (i.e after i^{th} phase, there will be J_i leaves labelled 1, 2, 3, …, J_i) , then $J_i <= J_{i+1}$
J_i will be equal to J_{i+1} when there are no new leaf created in phase i+1 (i.e rule 3 is applied in J_{i+1} extension)
In Figure 11 (Phase 3), Rule 1 is applied in 1st two extensions and Rule 2 is applied in 3rd extension, so here $J_3 = 3$
In Figure 12 (Phase 4), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_4 = 3 = J_3$
In Figure 13 (Phase 5), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_5 = 3 = J_4$
J_i will be less than J_{i+1} when few new leaves are created in phase i+1.
In Figure 14 (Phase 6), new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 2 is applied in last 3 extension which ends the phase). Here $J_6 = 6 > J_5$

So we can see that in phase i+1, only rule 1 will apply in extensions 1 to J_i (which really doesn't need much work, can be done in constant time and that's the trick 3), extension J_{i+1} onwards, rule 2 may apply to zero or more extensions and then finally rule 3, which ends the phase.
Now edge labels are represented using two indices (start, end), for any leaf edge, end will always be equal to phase number i.e. for phase i, end = i for leaf edges, for phase i+1, end = i+1 for leaf edges.

Trick 3
In any phase i, leaf edges may look like (p, i), (q, i), (r, i), …. where p, q, r are starting position of different edges and i is end position of all. Then in phase i+1, these leaf edges will look like (p, i+1), (q, i+1), (r, i+1),…. This way, in each phase, end position has to be incremented in all leaf edges. For this, we need to traverse through all leaf edges and increment end position for them. To do same thing in constant time, maintain a global index e and e will be equal to phase number. So now leaf edges will look like (p, e), (q, e), (r, e).. In any phase, just increment e and extension on all leaf edges will be done. Figure 19 shows this.

So using suffix links and tricks 1, 2 and 3, a suffix tree can be built in linear time.

Tree Tm could be implicit tree if a suffix is prefix of another. So we can add a $ terminal

symbol first and then run algorithm to get a true suffix tree (A true suffix tree contains all suffixes explicitly). To label each leaf with corresponding suffix starting position (all leaves are labelled as global index e), a linear time traversal can be done on tree.

At this point, we have gone through most of the things we needed to know to create suffix tree using Ukkonen's algorithm. In next Part 3, we will take string S = "abcabxabcd" as an example and go through all the things step by step and create the tree. While building the tree, we will discuss few more implementation issues which will be addressed by ActivePoints.
We will continue to discuss the algorithm in Part 4 and Part 5. Code implementation will be discussed in Part 6.

References:
http://web.stanford.edu/~mjkay/gusfield.pdf

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

56. Ukkonen's Suffix Tree Construction – Part 3

This article is continuation of following two articles:
Ukkonen's Suffix Tree Construction – Part 1
Ukkonen's Suffix Tree Construction – Part 2

Please go through Part 1 and Part 2, before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks.

Here we will take string S = "abcabxabcd" as an example and go through all the things step by step and create the tree.
We will add $ (discussed in Part 1 why we do this) so string S would be "abcabxabcd$".

While building suffix tree for string S of length m:

- There will be m phases 1 to m (one phase for each character)
  In our current example, m is 11, so there will be 11 phases.
- First phase will add first character 'a' in the tree, second phase will add second character 'b' in tree, third phase will add third character 'c' in tree, ……, m^{th} phase will add m^{th} character in tree (This makes Ukkonen's algorithm an online algorithm)
- Each phase i will go through at-most i extensions (from 1 to i). If current character being added in tree is not seen so far, all i extensions will be completed (Extension Rule 3 will not apply in this phase). If current character being added in tree is seen before, then phase i will complete early (as soon as Extension Rule 3 applies)

without going through all i extensions

- There are three extension rules (1, 2 and 3) and each extension j (from 1 to i) of any phase i will adhere to one of these three rules.
- Rule 1 adds a new character on existing leaf edge
- Rule 2 creates a new leaf edge (And may also create new internal node, if the path label ends in between an edge)
- Rule 3 ends the current phase (when current character is found in current edge being traversed)
- Phase 1 will read first character from the string, will go through 1 extension.
  (In figures, we are showing characters on edge labels just for explanation, while writing code, we will only use start and end indices – The Edge-label compression discussed in Part 2)
  Extension 1 will add suffix “a” in tree. We start from root and traverse path with label ‘a’. There is no path from root, going out with label ‘a’, so create a leaf edge (Rule 2).

  Phase 1 completes with the completion of extension 1 (As a phase i has at most i extensions)
  For any string, Phase 1 will have only one extension and it will always follow Rule 2.

- Phase 2 will read second character, will go through at least 1 and at most 2 extensions.
  In our example, phase 2 will read second character ‘b’. Suffixes to be added are “ab” and “b”.
  Extension 1 adds suffix “ab” in tree.
  Path for label ‘a’ ends at leaf edge, so add ‘b’ at the end of this edge.
  Extension 1 just increments the end index by 1 (from 1 to 2) on first edge (Rule 1).

  Extension 2 adds suffix “b” in tree. There is no path from root, going out with label ‘b’, so creates a leaf edge (Rule 2).

  Phase 2 completes with the completion of extension 2.
  Phase 2 went through two extensions here. Rule 1 applied in 1st Extension and Rule 2 applied in 2nd Extension.

- Phase 3 will read third character, will go through at least 1 and at most 3 extensions.
  In our example, phase 3 will read third character ‘c’. Suffixes to be added are “abc”, “bc” and “c”.
  Extension 1 adds suffix “abc” in tree.
  Path for label ‘ab’ ends at leaf edge, so add ‘c’ at the end of this edge.
  Extension 1 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

Extension 2 adds suffix “bc” in tree.
Path for label ‘b’ ends at leaf edge, so add ‘c’ at the end of this edge.
Extension 2 just increments the end index by 1 (from 2 to 3) on this edge (Rule 1).

Extension 3 adds suffix “c” in tree. There is no path from root, going out with label ‘c’, so creates a leaf edge (Rule 2).

Phase 3 completes with the completion of extension 3.
Phase 3 went through three extensions here. Rule 1 applied in first two Extensions and Rule 2 applied in 3rd Extension.

- Phase 4 will read fourth character, will go to at least 1 and at most 4 extensions.
  In our example, phase 4 will read fourth character ‘a’. Suffixes to be added are “abca”, “bca”, “ca” and “a”.
  Extension 1 adds suffix “abca” in tree.
  Path for label ‘abc’ ends at leaf edge, so add ‘a’ at the end of this edge.
  Extension 1 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

  Extension 2 adds suffix “bca” in tree.
  Path for label ‘bc’ ends at leaf edge, so add ‘a’ at the end of this edge.
  Extension 2 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

  Extension 3 adds suffix “ca” in tree.
  Path for label ‘c’ ends at leaf edge, so add ‘a’ at the end of this edge.
  Extension 3 just increments the end index by 1 (from 3 to 4) on this edge (Rule 1).

  Extension 4 adds suffix “a” in tree.
  Path for label ‘a’ exists in the tree. No more work needed and Phase 4 ends here (Rule 3 and Trick 2). This is an example of implicit suffix tree. Here suffix “a” is not seen explicitly (because it doesn’t end at a leaf edge) but it is in the tree implicitly. So there is no change in tree structure after extension 4. It will remain as above in Figure 28.
  Phase 4 completes as soon as Rule 3 is applied while Extension 4.
  Phase 4 went through four extensions here. Rule 1 applied in first three Extensions and Rule 3 applied in 4th Extension.

Now we will see few observations and how to implement those.

1. At the end of any phase i, there are at most i leaf edges (if i^{th} character is not seen so far, there will be i leaf edges, else there will be less than i leaf edges).
   e.g. After phases 1, 2 and 3 in our example, there are 1, 2 and 3 leaf edges respectively, but after phase 4, there are 3 leaf edges only (not 4).
2. After completing phase i, "end" indices of all leaf edges are i. How do we implement this in code? Do we need to iterate through all those extensions, find leaf edges by traversing from root to leaf and increment the "end" index? Answer is "NO".
   For this, we will maintain a global variable (say "END") and we will just increment this global variable "END" and all leaf edge end indices will point to this global variable. So this way, if we have j leaf edges after phase i, then in phase i+1, first j extensions (1 to j) will be done by just incrementing variable "END" by 1 (END will be i+1 at the point).
   Here we just implemented the trick 3 – **Once a leaf, always a leaf**. This trick processes all the j leaf edges (i.e. extension 1 to j) using rule 1 in a constant time in any phase. Rule 1 will not apply to subsequent extensions in the same phase. This can be verified in the four phases we discussed above. If at all Rule 1 applies in any phase, it only applies in initial few phases continuously (say 1 to j). Rule 1 never applies later in a given phase once Rule 2 or Rule 3 is applied in that phase.
3. In the example explained so far, in each extension (where trick 3 is not applied) of any phase to add a suffix in tree, we are traversing from root by matching path labels against the suffix being added. If there are j leaf edges after phase i, then in phase i+1, first j extensions will follow Rule 1 and will be done in constant time using trick 3. There are i+1-j extensions yet to be performed. For these extensions, which node (root or some other internal node) to start from and which path to go? Answer to this depends on how previous phase i is completed.

   If previous phase i went through all the i extensions (when i^{th} character is unique so far), then in next phase i+1, trick 3 will take care of first i suffixes (the i leaf edges) and then extension i+1 will start from root node and it will insert just one character [$(i+1)^{th}$] suffix in tree by creating a leaf edge using Rule 2.
   If previous phase i completes early (and this will happen if and only if rule 3 applies – when i^{th} character is already seen before), say at j^{th} extension (i.e. rule 3 is applied at j^{th} extension), then there are j-1 leaf edges so far.
   We will state few more facts (which may be a repeat, but we want to make sure it's clear to you at this point) here based on discussion so far:
   - *Phase 1 starts with Rule 2, all other phases start with Rule 1*
   - *Any phase ends with either Rule 2 or Rule 3*
   - *Any phase i may go through a series of j extensions (1 <= j <= i). In these j extensions, first p (0 <= p < i) extensions will follow Rule 1, next q (0 <= q <= i-p) extensions will follow Rule 2 and next r (0<= r <= 1) extensions will follow Rule 3. The order in which Rule 1, Rule 2 and Rule 3 apply, is never intermixed in a phase. They apply in order of their number (if at all applied), i.e. in a phase, Rule 1 applies 1st, then Rule 2 and then Rule 3*
   - *In a phase i, p + q + r <= i*

- *At the end of any phase i, there will be p+q leaf edges and next phase i+1 will go through Rule 1 for first p+q extensions*

In the next phase i+1, trick 3 (Rule 1) will take care of first j-1 suffixes (the j-1 leaf edges), then extension j will start where we will add j^{th} suffix in tree. For this, we need to find the best possible matching edge and then add new character at the end of that edge. How to find the end of best matching edge? Do we need to traverse from root node and match tree edges against the j^{th} suffix being added character by character? This will take time and overall algorithm will not be linear. activePoint comes to the rescue here.
In previous phase i, while j^{th} extension, path traversal ended at a point (which could be an internal node or some point in the middle of an edge) where i^{th} character being added was found in tree already and Rule 3 applied, j^{th} extension of phase i+1 will start exactly from the same point and we start matching path against $(i+1)^{th}$ character. activePoint helps to avoid unnecessary path traversal from root in any extension based on the knowledge gained in traversals done in previous extension. There is no traversal needed in 1^{st} p extensions where Rule 1 is applied. Traversal is done where Rule 2 or Rule 3 gets applied and that's where activePoint tells the starting point for traversal where we match the path against the current character being added in tree. Implementation is done in such a way that, in any extension where we need a traversal, activePoint is set to right location already (with one exception case **APCFALZ** discussed below) and at the end of current extension, we reset activePoint as apprppriate so that next extension (of same phase or next phase) where a traversal is required, activePoint points to the right place already.

activePoint: This could be root node, any internal node or any point in the middle of an edge. This is the point where traversal starts in any extension. For the 1st extension of phase 1, activePoint is set to root. Other extension will get activePoint set correctly by previous extension (with one exception case **APCFALZ** discussed below) and it is the responsibility of current extension to reset activePoint appropriately at the end, to be used in next extension where Rule 2 or Rule 3 is applied (of same or next phase).
To accomplish this, we need a way to store activePoint. We will store this using three variables: **activeNode**, **activeEdge**, **activeLength**.
activeNode: This could be root node or an internal node.
activeEdge: When we are on root node or internal node and we need to walk down, we need to know which edge to choose. activeEdge will store that information. In case, activeNode itself is the point from where traversal starts, then activeEdge will be set to next character being processed in next phase.
activeLength: This tells how many characters we need to walk down (on the path represented by activeEdge) from activeNode to reach the activePoint where traversal starts. In case, activeNode itself is the point from where traversal starts, then activeLength will be ZERO.

(click on below image to see it clearly)

After phase i, if there are j leaf edges then in phase i+1, first j extensions will be done by trick 3. activePoint will be needed for the extensions from j+1 to i+1 and activePoint may or may not change between two extensions depending on the point where previous extension ends.

activePoint change for extension rule 3 (APCFER3): When rule 3 applies in any phase i, then before we move on to next phase i+1, we increment activeLength by 1. There is no change in activeNode and activeEdge. Why? Because in case of rule 3, the current character from string S is matched on the same path represented by current activePoint, so for next activePoint, activeNode and activeEdge remain the same, only activeLenth is increased by 1 (because of matched character in current phase). This new activePoint (same node, same edge and incremented length) will be used in phase i+1.

activePoint change for walk down (APCFWD): activePoint may change at the end of an extension based on extension rule applied. activePoint may also change during the extension when we do walk down. Let's consider an activePoint is (A, s, 11) in the above activePoint example figure. If this is the activePoint at the start of some extension, then while walk down from activeNode A, other internal nodes will be seen. Anytime if we encounter an internal node while walk down, that node will become activeNode (it will change activeEdge and activeLenght as appropriate so that new activePoint represents the same point as earlier). In this walk down, below is the sequence of changes in activePoint:
(A, s, 11) — >>> (B, w, 7) —- >>> (C, a, 3)
All above three activePoints refer to same point 'c'
Let's take another example.
If activePoint is (D, a, 11) at the start of an extension, then while walk down, below is the sequence of changes in activePoint:
(D, a, 10) — >>> (E, d, 7) — >>> (F, f, 5) — >> (F, j, 1)
All above activePoints refer to same point 'k'.
If activePoints are (A, s, 3), (A, t, 5), (B, w, 1), (D, a, 2) etc when no internal node comes in the way while walk down, then there will be no change in activePoint for APCFWD.
The idea is that, at any time, the closest internal node from the point, where we want to reach, should be the activePoint. Why? This will minimize the length of traversal in the next extension.

activePoint change for Active Length ZERO (APCFALZ): Let's consider an activePoint (A, s, 0) in the above activePoint example figure. And let's say current character being processed from string S is 'x' (or any other character). At the start of extension, when activeLength is ZERO, activeEdge is set to the current character being processed, i.e. 'x', because there is no walk down needed here (as

activeLength is ZERO) and so next character we look for is current character being processed.

4. While code implementation, we will loop through all the characters of string S one by one. Each loop for i^{th} character will do processing for phase i. Loop will run one or more time depending on how many extensions are left to be performed (Please note that in a phase i+1, we don't really have to perform all i+1 extensions explicitly, as trick 3 will take care of j extensions for all j leaf edges coming from previous phase i). We will use a variable **remainingSuffixCount**, to track how many extensions are yet to be performed explicitly in any phase (after trick 3 is performed). Also, at the end of any phase, if remainingSuffixCount is ZERO, this tells that all suffixes supposed to be added in tree, are added explicitly and present in tree. If remainingSuffixCount is non-zero at the end of any phase, that tells that suffixes of that many count are not added in tree explicitly (because of rule 3, we stopped early), but they are in tree implicitly though (Such trees are called implicit suffix tree). These implicit suffixes will be added explicitly in subsequent phases when a unique character comes in the way.

We will continue our discussion in Part 4 and Part 5. Code implementation will be discussed in Part 6.

References:
http://web.stanford.edu/~mjkay/gusfield.pdf
Ukkonen's suffix tree algorithm in plain English

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

57. Ukkonen's Suffix Tree Construction – Part 4

This article is continuation of following three articles:
Ukkonen's Suffix Tree Construction – Part 1
Ukkonen's Suffix Tree Construction – Part 2
Ukkonen's Suffix Tree Construction – Part 3

Please go through Part 1, Part 2 and Part 3, before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and some details on activePoint along with an example string "abcabxabcd" where we went through four phases of building suffix tree.

Let's revisit those four phases we have seen already in Part 3, in terms of trick 2, trick 3 and activePoint.

- activePoint is initialized to (root, NULL, 0), i.e. activeNode is root, activeEdge is NULL (for easy understanding, we are giving character value to activeEdge, but in code implemntation, it will be index of the character) and activeLength is ZERO.
- The global variable END and remainingSuffixCount are initialized to ZERO

**********Phase 1**********

In Phase 1, we read 1st character (a) from string S

- Set END to 1
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
  - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**.
  - Check if there is an edge going out from activeNode (which is root in this phase 1) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created (Rule 2).
  - Once extension is performed, decrement the remainingSuffixCount by 1
  - At this point, activePoint is (root, a, 0)

At the end of phase 1, remainingSuffixCount is ZERO (All suffixes are added explicitly). Figure 20 in Part 3 is the resulting tree after phase 1.

**********Phase 2**********

In Phase 2, we read 2nd character (b) from string S

- Set END to 2 (This will do extension 1)
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
  - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'b'). This is **APCFALZ**.
  - Check if there is an edge going out from activeNode (which is root in this phase 2) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created.
  - Once extension is performed, decrement the remainingSuffixCount by 1
  - At this point, activePoint is (root, b, 0)

At the end of phase 2, remainingSuffixCount is ZERO (All suffixes are added explicitly). Figure 22 in Part 3 is the resulting tree after phase 2.

**********Phase 3**********

In Phase 3, we read 3rd character (c) from string S

- Set END to 3 (This will do extensions 1 and 2)
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)

- Run a loop remainingSuffixCount times (i.e. one time) as below:
  - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'c'). This is **APCFALZ**
  - Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, leaf edge gets created.
  - Once extension is performed, decrement the remainingSuffixCount by 1
  - At this point, activePoint is (root, c, 0)

At the end of phase 3, remainingSuffixCount is ZERO (All suffixes are added explicitly). Figure 25 in Part 3 is the resulting tree after phase 3.

**********Phase 4**********

In Phase 4, we read 4th character (a) from string S

- Set END to 4 (This will do extensions 1, 2 and 3)
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is 1 extension left to be performed)
- Run a loop remainingSuffixCount times (i.e. one time) as below:
  - If activeLength is ZERO, set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**
  - Check if there is an edge going out from activeNode (which is root in this phase 3) for the activeEdge. If not, create a leaf edge. If present, walk down (The trick 1 – skip/count). In our example, edge 'a' is present going out of activeNode (i.e. root). No walk down needed as activeLength < edgeLength. We increment activeLength from zero to 1 (**APCFER3**) and stop any further processing (Rule 3).
  - At this point, activePoint is (root, a, 1) and remainingSuffixCount remains set to 1 (no change there)

At the end of phase 4, remainingSuffixCount is 1 (One suffix 'a', the last one, is not added explicitly in tree, but it is there in tree implicitly).
Figure 28 in Part 3 is the resulting tree after phase 4.

Revisiting completed for 1st four phases, we will continue building the tree and see how it goes.

**********Phase 5**********

In phase 5, we read 5th character (b) from string S

- Set END to 5 (This will do extensions 1, 2 and 3). See Figure 29 shown below.
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 2 here, i.e. there are 2 extension left to be performed, which are extensions 4 and 5. Extension 4 is supposed to add suffix "ab" and extension 5 is supposed to add suffix "b" in tree)
- Run a loop remainingSuffixCount times (i.e. two times) as below:
  - Check if there is an edge going out from activeNode (which is root in this phase

3) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
    - Do a walk down (The trick 1 – skip/count) if necessary. In current phase 5, no walk down needed as activeLength < edgeLength. Here activePoint is (root, a, 1) for extension 4 (remainingSuffixCount = 2)
    - Check if current character of string S (which is 'b') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 1 to 2 (**APCFER3**) and we stop here (Rule 3).
    - At this point, activePoint is (root, a, 2) and remainingSuffixCount remains set to 2 (no change in remainingSuffixCount)

At the end of phase 5, remainingSuffixCount is 2 (Two suffixes, 'ab' and 'b', the last two, are not added explicitly in tree, but they are in tree implicitly).

**********Phase 6**********

In phase 6, we read 6th character (x) from string S

- Set END to 6 (This will do extensions 1, 2 and 3)
- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 3 here, i.e. there are 3 extension left to be performed, which are extensions 4, 5 and 6 for suffixes "abx", "bx" and "x" respectively)
- Run a loop remainingSuffixCount times (i.e. three times) as below:
    - While extension 4, the activePoint is (root, a, 2) which points to 'b' on edge starting with 'a'.
    - In extension 4, current character 'x' from string S doesn't match with the next character on the edge after activePoint, so this is the case of extension rule 2. So a leaf edge is created here with edge label x. Also here traversal ends in middle of an edge, so a new internal node also gets created at the end of activePoint.
    - Decrement the remainingSuffixCount by 1 (from 3 to 2) as suffix "abx" added in tree.

Now activePoint will change after applying rule 2. Three other cases, (**APCFER3**, **APCFWD** and **APCFALZ**) where activePoint changes, are already discussed in Part 3.

activePoint change for extension rule 2 (APCFER2):
Case 1 (APCFER2C1): If activeNode is root and activeLength is greater than ZERO, then decrement the activeLength by 1 and activeEdge will be set "S[i – remainingSuffixCount + 1]" where i is current phase number. Can you see why this

change in activePoint? Look at current extension we just discussed above for phase 6 (i=6) again where we added suffix “abx”. There activeLength is 2 and activeEdge is ‘a’. Now in next extension, we need to add suffix “bx” in the tree, i.e. path label in next extension should start with ‘b’. So ‘b’ (the 5th character in string S) should be active edge for next extension and index of b will be “i – remainingSuffixCount + 1” (6 – 2 + 1 = 5). activeLength is decremented by 1 because activePoint gets closer to root by length 1 after every extension.
What will happen If activeNode is root and activeLength is ZERO? This case is already taken care by **APCFALZ**.

Case 2 (APCFER2C2): If activeNode is not root, then follow the suffix link from current activeNode. The new node (which can be root node or another internal node) pointed by suffix link will be the activeNode for next extension. No change in activeLength and activeEdge. Can you see why this change in activePoint? This is because: If two nodes are connected by a suffix link, then labels on all paths going down from those two nodes, starting with same character, will be exactly same and so for two corresponding similar point on those paths, activeEdge and activeLength will be same and the two nodes will be the activeNode. Look at Figure 18 in Part 2. Let’s say in phase i and extension j, suffix ‘xAabcdedg’ was added in tree. At that point, let’s say activePoint was (Node-V, a, 7), i.e. point ‘g’. So for next extension j+1, we would add suffix ‘Aabcdefg’ and for that we need to traverse 2nd path shown in Figure 18. This can be done by following suffix link from current activeNode v. Suffix link takes us to the path to be traversed somewhere in between [Node s(v)] below which the path is exactly same as how it was below the previous activeNode v. As said earlier, “activePoint gets closer to root by length 1 after every extension”, this reduction in length will happen above the node s(v) but below s(v), no change at all. So when activeNode is not root in current extension, then for next extension, only activeNode changes (No change in activeEdge and activeLength).

-
  - At this point in extension 4, current activePoint is (root, a, 2) and based on **APCFER2C1**, new activePoint for next extension 5 will be (root, b, 1)
  - Next suffix to be added is ‘bx’ (with remainingSuffixCount 2).
  - Current character ‘x’ from string S doesn’t match with the next character on the edge after activePoint, so this is the case of extension rule 2. So a leaf edge is created here with edge label x. Also here traversal ends in middle of an edge, so a new internal node also gets created at the end of activePoint.
    Suffix link is also created from previous internal node (of extension 4) to the new internal node created in current extension 5.
  - Decrement the remainingSuffixCount by 1 (from 2 to 1) as suffix “bx” added in tree.

-

- At this point in extension 5, current activePoint is (root, b, 1) and based on **APCFER2C1** new activePoint for next extension 6 will be (root, x, 0)
- Next suffix to be added is 'x' (with remainingSuffixCount 1).
- In the next extension 6, character x will not match to any existing edge from root, so a new edge with label x will be created from root node. Also suffix link from previous extension's internal node goes to root (as no new internal node created in current extension 6).
- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix "x" added in tree

This completes the phase 6.
Note that phase 6 has completed all its 6 extensions (Why? Because the current character c was not seen in string so far, so rule 3, which stops further extensions never got chance to get applied in phase 6) and so the tree generated after phase 6 is a true suffix tree (i.e. not an implicit tree) for the characters 'abcabx' read so far and it has all suffixes explicitly in the tree.

While building the tree above, following facts were noticed:

- A newly created internal node in extension i, points to another internal node or root (if activeNode is root in extension i+1) by the end of extension i+1 via suffix link (Every internal node MUST have a suffix link pointing to another internal node or root)
- Suffix link provides short cut while searching path label end of next suffix
- With proper tracking of activePoints between extensions/phases, unnecessary walkdown from root can be avoided.

We will go through rest of the phases (7 to 11) in Part 5 and build the tree completely and after that, we will see the code for the algorithm in Part 6.

References:
http://web.stanford.edu/~mjkay/gusfield.pdf
Ukkonen's suffix tree algorithm in plain English

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

58. Ukkonen's Suffix Tree Construction – Part 5

This article is continuation of following four articles:
Ukkonen's Suffix Tree Construction – Part 1

Ukkonen's Suffix Tree Construction – Part 2
Ukkonen's Suffix Tree Construction – Part 3
Ukkonen's Suffix Tree Construction – Part 4

Please go through Part 1, Part 2, Part 3 and Part 4, before looking at current article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and some details on activePoint along with an example string "abcabxabcd" where we went through six phases of building suffix tree.
Here, we will go through rest of the phases (7 to 11) and build the tree completely.

**********Phase 7**********

In phase 7, we read 7th character (a) from string S

- Set END to 7 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 6.

- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 1 here, i.e. there is only 1 extension left to be performed, which is extensions 7 for suffix 'a')
- Run a loop remainingSuffixCount times (i.e. one time) as below:
  - If activeLength is ZERO [activePoint in previous phase was (root, x, 0)], set activeEdge to the current character (here activeEdge will be 'a'). This is **APCFALZ**. Now activePoint becomes (root, 'a', 0).
  - Check if there is an edge going out from activeNode (which is root in this phase 7) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root), here we increment activeLength from zero to 1 (**APCFER3**) and stop any further processing.
  - At this point, activePoint is (root, a, 1) and remainingSuffixCount remains set to 1 (no change there)

At the end of phase 7, remainingSuffixCount is 1 (One suffix 'a', the last one, is not added explicitly in tree, but it is there in tree implicitly).
Above Figure 33 is the resulting tree after phase 7.

**********Phase 8**********

In phase 8, we read 8th character (b) from string S

- Set END to 8 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 7 (Figure 34).

- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 2 here, i.e. there are two extensions left to be performed, which are extensions 7 and 8 for suffixes 'ab' and 'b' respectively)

- Run a loop remainingSuffixCount times (i.e. two times) as below:
  - Check if there is an edge going out from activeNode (which is root in this phase 8) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
  - Do a walk down (The trick 1 – skip/count) if necessary. In current phase 8, no walk down needed as activeLength < edgeLength. Here activePoint is (root, a, 1) for extension 7 (remainingSuffixCount = 2)
  - Check if current character of string S (which is 'b') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 1 to 2 (**APCFER3**) and we stop here (Rule 3).
  - At this point, activePoint is (root, a, 2) and remainingSuffixCount remains set to 2 (no change in remainingSuffixCount)

At the end of phase 8, remainingSuffixCount is 2 (Two suffixes, 'ab' and 'b', the last two, are not added explicitly in tree explicitly, but they are in tree implicitly).

**********Phase 9**********

In phase 9, we read 9th character (c) from string S

- Set END to 9 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 8.

- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 3 here, i.e. there are three extensions left to be performed, which are extensions 7, 8 and 9 for suffixes 'abc', 'bc' and 'c' respectively)
- Run a loop remainingSuffixCount times (i.e. three times) as below:
  - Check if there is an edge going out from activeNode (which is root in this phase 9) for the activeEdge. If not, create a leaf edge. If present, walk down. In our example, edge 'a' is present going out of activeNode (i.e. root).
  - Do a walk down (The trick 1 – skip/count) if necessary. In current phase 9,walk down needed as activeLength(2) >= edgeLength(2). While walk down, activePoint changes to (Node A, c, 0) based on **APCFWD** (This is first time **APCFWD** is being applied in our example).
  - Check if current character of string S (which is 'c') is already present after the activePoint. If yes, no more processing (rule 3). Same is the case in our example, so we increment activeLength from 0 to 1 (**APCFER3**) and we stop here (Rule 3).
  - At this point, activePoint is (Node A, c, 1) and remainingSuffixCount remains set to 3 (no change in remainingSuffixCount)

At the end of phase 9, remainingSuffixCount is 3 (Three suffixes, 'abc', 'bc' and 'c', the last three, are not added explicitly in tree explicitly, but they are in tree implicitly).

**********Phase 10**********

In phase 10, we read 10th character (d) from string S

- Set END to 10 (This will do extensions 1, 2, 3, 4, 5 and 6) – because we have 6 leaf edges so far by the end of previous phase 9.

- Increment remainingSuffixCount by 1 (remainingSuffixCount will be 4 here, i.e. there are four extensions left to be performed, which are extensions 7, 8, 9 and 10 for suffixes 'abcd', 'bcd', 'cd' and 'd' respectively)
- Run a loop remainingSuffixCount times (i.e. four times) as below:

**********Extension 7********** Check if there is an edge going out from activeNode (Node A) for the activeEdge(c). If not, create a leaf edge. If present, walk down. In our example, edge 'c' is present going out of activeNode (Node A). Do a walk down (The trick 1 – skip/count) if necessary. In current Extension 7, no walk down needed as activeLength < edgeLength. Check if current character of string S (which is 'd') is already present after the activePoint. If not, rule 2 will apply. In our example, there is no path starting with 'd' going out of activePoint, so we create a leaf edge with label 'd'. Since activePoint ends in the middle of an edge, we will create a new internal node just after the activePoint (Rule 2)

The newly created internal node c (in above Figure) in current extension 7, will get it's suffix link set in next extension 8 (see Figure 38 below). Decrement the remainingSuffixCount by 1 (from 4 to 3) as suffix "abcd" added in tree. Now activePoint will change for next extension 8. Current activeNode is an internal node (Node A), so there must be a suffix link from there and we will follow that to get new activeNode and that's going to be 'Node B'. There is no change in activeEdge and activeLength (This is **APCFER2C2**). So new activePoint is (Node B, c, 1).

**********Extension 8**********

Now in extension 8 (here we will add suffix 'bcd'), while adding character 'd' after the current activePoint, exactly same logic will apply as previous extension 7. In previous extension 7, we added character 'd' at activePoint (Node A, c, 1) and in current extension 8, we are going to add same character 'd' at activePoint (Node B c, 1). So logic will be same and here we a new leaf edge with label 'd' and a new internal node will be created. And the new internal node (C) of previous extension will point to the new node (D) of current extension via suffix link.

Please note the node C from previous extension (see Figure 37 above) got it's suffix link set here and node D created in current extension will get it's suffix link set in next extension. What happens if no new node created in next phase? We have seen this

before in Phase 6 (Part 4) and will see again in last extension of this Phase 10. Stay Tuned. Decrement the remainingSuffixCount by 1 (from 3 to 2) as suffix “bcd” added in tree. Now activePoint will change for next extension 9. Current activeNode is an internal node (Node B), so there must be a suffix link from there and we will follow that to get new activeNode and that is ‘Root Node’. There is no change in activeEdge and activeLength (This is **APCFER2C2**). So new activePoint is (root, c, 1).

**********Extension 9**********

Now in extension 9 (here we will add suffix ‘cd’), while adding character ‘d’ after the current activePoint, exactly same logic will apply as previous extensions 7 and 8. Note that internal node D created in previous extension 8, now points to internal node E (created in current extension) via suffix link.

-
    - Decrement the remainingSuffixCount by 1 (from 2 to 1) as suffix “cd” added in tree.
    - Now activePoint will change for next extension 10. Current activeNode is root and activeLength is 1, based on **APCFER2C1**, activeNode will remain ‘root’, activeLength will be decremented by 1 (from 1 to ZERO) and activeEdge will be ‘d’. So new activePoint is (root, d, 0).

**********Extension 10**********

Now in extension 10 (here we will add suffix ‘d’), while adding character ‘d’ after the current activePoint, there is no edge starting with d going out of activeNode root, so a new leaf edge with label d is created (Rule 2). Note that internal node E created in previous extension 9, now points to root node via suffix link (as no new internal node created in this extension).

Internal Node created in previous extension, waiting for suffix link to be set in next extension, points to root if no internal node created in next extension. In code implementation, as soon as a new internal node (Say A) gets created in an extension j, we will set it’s suffix link to root node and in next extension j+1, if Rule 2 applies on an existing or newly created node (Say B), then suffix link of node A will change to the new node B , else node A will keep pointing to root Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix “d” added in tree. That means no more suffix is there to add and so the phase 10 ends here. Note that this tree is an explicit tree as all suffixes are added in tree explicitly (Why ?? because character d was not seen before in string S so far) activePoint for next phase 11 is (root, d, 0).

We see following facts in Phase 10:

- Internal Nodes connected via suffix links have exactly same tree below them, e.g. In

above Figure 40, A and B have same tree below them, similarly C, D and E have same tree below them.

- Due to above fact, in any extension, when current activeNode is derived via suffix link from previous extension's activeNode, then exactly same extension logic apply in current extension as previous extension. (In Phase 10, same extension logic is applied in extensions 7, 8 and 9)
- If a new internal node gets created in extension j of any phase i, then this newly created internal node will get it's suffix link set by the end of next extension j+1 of same phase i. e.g. node C got created in extension 7 of phase 10 (Figure 37) and it got it's suffix link set to node D in extension 8 of same phase 10 (Figure 38). Similarly node D got created in extension 8 of phase 10 (Figure 38) and it got its suffix link set to node E in extension 9 of same phase 10 (Figure 39). Similarly node E got created in extension 9 of phase 10 (Figure 39) and it got its suffix link set to root in extension 10 of same phase 10 (Figure 40).
- Based on above fact, every internal node will have a suffix link to some other internal node or root. Root is not an internal node and it will not have suffix link.

**********Phase 11**********

In phase 11, we read 11th character ($) from string S

- Set END to 11 (This will do extensions 1 to 10) – because we have 10 leaf edges so far by the end of previous phase 10.

- Increment remainingSuffixCount by 1 (from 0 to 1), i.e. there is only one suffix '$' to be added in tree.
- Since activeLength is ZERO, activeEdge will change to current character '$' of string S being processed (**APCFALZ**).
- There is no edge going out from activeNode root, so a leaf edge with label '$' will be created (Rule 2).

- Decrement the remainingSuffixCount by 1 (from 1 to 0) as suffix "$" added in tree. That means no more suffix is there to add and so the phase 11 ends here. Note that this tree is an explicit tree as all suffixes are added in tree explicitly (Why ?? because character $ was not seen before in string S so far)

Now we have added all suffixes of string 'abcabxabcd$' in suffix tree. There are 11 leaf ends in this tree and labels on the path from root to leaf end represents one suffix. Now the only one thing left is to assign a number (suffix index) to each leaf end and that number would be the suffix starting position in the string S. This can be done by a DFS traversal on tree. While DFS traversal, keep track of label length and when a leaf end is found, set the suffix index as "stringSize – labelSize + 1". Indexed suffix tree will look like below:

In above Figure, suffix indices are shown as character position starting with 1 (It's not zero indexed). In code implementation, suffix index will be set as zero indexed, i.e. where we see suffix index j (1 to m for string of length m) in above figure, in code implementation, it will be j-1 (0 to m-1)
And we are done !!!!

Data Structure to represent suffix tree
How to represent the suffix tree?? There are nodes, edges, labels and suffix links and indices.
Below are some of the operations/query we will be doing while building suffix tree and later on while using the suffix tree in different applications/usages:

- What length of path label on some edge?
- What is the path label on some edge?
- Check if there is an outgoing edge for a given character from a node.
- What is the character value on an edge at some given distance from a node?
- Where an internal node is pointing via suffix link?
- What is the suffix index on a path from root to leaf?
- Check if a given string present in suffix tree (as substring, suffix or prefix)?

We may think of different data structures which can fulfil these requirements.
In the next Part 6, we will discuss the data structure we will use in our code implementation and the code as well.

References:
http://web.stanford.edu/~mjkay/gusfield.pdf
Ukkonen's suffix tree algorithm in plain English

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

59. Ukkonen's Suffix Tree Construction – Part 6

This article is continuation of following five articles:
Ukkonen's Suffix Tree Construction – Part 1
Ukkonen's Suffix Tree Construction – Part 2
Ukkonen's Suffix Tree Construction – Part 3
Ukkonen's Suffix Tree Construction – Part 4
Ukkonen's Suffix Tree Construction – Part 5

Please go through Part 1, Part 2, Part 3, Part 4 and Part 5, before looking at current

article, where we have seen few basics on suffix tree, high level ukkonen's algorithm, suffix link and three implementation tricks and activePoints along with an example string "abcabxabcd" where we went through all phases of building suffix tree.
Here, we will see the data structure used to represent suffix tree and the code implementation.

At that end of Part 5 article, we have discussed some of the operations we will be doing while building suffix tree and later when we use suffix tree in different applications.
There could be different possible data structures we may think of to fulfill the requirements where some data structure may be slow on some operations and some fast. Here we will use following in our implementation:

We will have SuffixTreeNode structure to represent each node in tree. SuffixTreeNode structure will have following members:

- **children** – This will be an array of alphabet size. This will store all the children nodes of current node on different edges starting with different characters.
- **suffixLink** – This will point to other node where current node should point via suffix link.
- **start, end** – These two will store the edge label details from parent node to current node. (start, end) interval specifies the edge, by which the node is connected to its parent node. Each edge will connect two nodes, one parent and one child, and (start, end) interval of a given edge will be stored in the child node. Lets say there are two nods A (parent) and B (Child) connected by an edge with indices (5, 8) then this indices (5, 8) will be stored in node B.
- **suffixIndex** – This will be non-negative for leaves and will give index of suffix for the path from root to this leaf. For non-leaf node, it will be -1 .

This data structure will answer to the required queries quickly as below:

- How to check if a node is root ? — Root is a special node, with no parent and so it's start and end will be -1, for all other nodes, start and end indices will be non-negative.
- How to check if a node is internal or leaf node ? — suffixIndex will help here. It will be -1 for internal node and non-negative for leaf nodes.
- What is the length of path label on some edge? — Each edge will have start and end indices and length of path label will be end-start+1
- What is the path label on some edge ? — If string is S, then path label will be substring of S from start index to end index inclusive, [start, end].
- How to check if there is an outgoing edge for a given character c from a node A ? — If A->children[c] is not NULL, there is a path, if NULL, no path.
- What is the character value on an edge at some given distance d from a node A ? — Character at distance d from node A will be S[A->start + d], where S is the string.
- Where an internal node is pointing via suffix link ? — Node A will point to A->suffixLink
- What is the suffix index on a path from root to leaf ? — If leaf node is A on the path,

then suffix index on that path will be A->suffixIndex

Following is C implementation of Ukkonen's Suffix Tree Construction. The code may look a bit lengthy, probably because of a good amount of comments.

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

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

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

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

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

typedef struct SuffixTreeNode Node;

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

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

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

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

Node *newNode(int start, int *end)
{
```

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

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

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

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

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

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

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

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

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

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

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

            /*A new leaf edge is created in above line starting
             from  an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
             reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
            {
                //Start from next node (the new activeNode)
                continue;
            }
            /*Extension Rule 3 (current character being processed
              is already on the edge)*/
            if (text[next->start + activeLength] == text[pos])
            {
                //If a newly created node waiting for it's
                //suffix link to be set, then set suffix link
                //of that waiting node to curent active node
                if(lastNewNode != NULL && activeNode != root)
                {
                    lastNewNode->suffixLink = activeNode;
                    lastNewNode = NULL;
                }

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

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

```
            *splitEnd = next->start + activeLength - 1;

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

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

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

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

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

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

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

    if (n->start != -1) //A non-root node
```

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

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

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

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

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

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

```
    setSuffixIndexByDFS(root, labelHeight);

    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
//  strcpy(text, "abc"); buildSuffixTree();
//  strcpy(text, "xabxac#");    buildSuffixTree();
//  strcpy(text, "xabxa");  buildSuffixTree();
//  strcpy(text, "xabxa$"); buildSuffixTree();
    strcpy(text, "abcabxabcd$"); buildSuffixTree();
//  strcpy(text, "geeksforgeeks$"); buildSuffixTree();
//  strcpy(text, "THIS IS A TEST TEXT$"); buildSuffixTree();
//  strcpy(text, "AABAACAADAABAAABAA$"); buildSuffixTree();
    return 0;
}
```

Output (Each edge of Tree, along with suffix index of child node on edge, is printed in DFS order. To understand the output better, match it with the last figure no 43 in previous Part 5 article):

```
$ [10]
ab [-1]
c [-1]
abxabcd$ [0]
d$ [6]
xabcd$ [3]
b [-1]
c [-1]
abxabcd$ [1]
d$ [7]
xabcd$ [4]
c [-1]
abxabcd$ [2]
d$ [8]
d$ [9]
xabcd$ [5]
```

Now we are able to build suffix tree in linear time, we can solve many string problem in efficient way:

- Check if a given pattern P is substring of text T (Useful when text is fixed and pattern changes, KMP otherwise
- Find all occurrences of a given pattern P present in text T
- Find longest repeated substring
- Linear Time Suffix Array Creation

The above basic problems can be solved by DFS traversal on suffix tree.

We will soon post articles on above problems and others like below:

- Build Generalized suffix tree
- Linear Time Longest common substring problem
- Linear Time Longest palindromic substring

And More.

Test you understanding?

1. Draw suffix tree (with proper suffix link, suffix indices) for string "AABAACAADAABAAABAA$" on paper and see if that matches with code output.
2. Every extension must follow one of the three rules: Rule 1, Rule 2 and Rule 3. Following are the rules applied on five consecutive extensions in some Phase i (i > 5), which ones are valid:
   A) Rule 1, Rule 2, Rule 2, Rule 3, Rule 3
   B) Rule 1, Rule 2, Rule 2, Rule 3, Rule 2
   C) Rule 2, Rule 1, Rule 1, Rule 3, Rule 3
   D) Rule 1, Rule 1, Rule 1, Rule 1, Rule 1
   E) Rule 2, Rule 2, Rule 2, Rule 2, Rule 2
   F) Rule 3, Rule 3, Rule 3, Rule 3, Rule 3
3. What are the valid sequences in above for Phase 5
4. Every internal node MUST have it's suffix link set to another node (internal or root). Can a newly created node point to already existing internal node or not ? Can it happen that a new node created in extension j, may not get it's right suffix link in next extension j+1 and get the right one in later extensions like j+2, j+3 etc ?
5. Try solving the basic problems discussed above.

We have published following articles on suffix tree applications:

- Suffix Tree Application 1 – Substring Check
- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 3 – Longest Repeated Substring
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Generalized Suffix Tree 1
- Suffix Tree Application 5 – Longest Common Substring
- Suffix Tree Application 6 – Longest Palindromic Substring

References:
http://web.stanford.edu/~mjkay/gusfield.pdf
Ukkonen's suffix tree algorithm in plain English

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

60. Suffix Tree Application 1 – Substring Check

Given a text string and a pattern string, check if pattern exists in text or not.

Few pattern searching algorithms (KMP, Rabin-Karp, Naive Algorithm, Finite Automata) are already discussed, which can be used for this check.
Here we will discuss suffix tree based algorithm.

As a prerequisite, we must know how to build a suffix tree in one or the other way.
Once we have a suffix tree built for given text, we need to traverse the tree from root to leaf against the characters in pattern. If we do not fall off the tree (i.e. there is a path from root to leaf or somewhere in middle) while traversal, then pattern exists in text as a substring.

Here we will build suffix tree using Ukkonen's Algorithm, discussed already as below:
Ukkonen's Suffix Tree Construction – Part 1
Ukkonen's Suffix Tree Construction – Part 2
Ukkonen's Suffix Tree Construction – Part 3
Ukkonen's Suffix Tree Construction – Part 4
Ukkonen's Suffix Tree Construction – Part 5
Ukkonen's Suffix Tree Construction – Part 6

The core traversal implementation for substring check, can be modified accordingly for suffix trees built by other algorithms.

```
// A C program for substring check using Ukkonen's Suffix Tree Construc
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

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

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

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

```
typedef struct SuffixTreeNode Node;

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

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

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

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

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

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

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

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

int walkDown(Node *currNode)
{
    /*activePoint change for walk down (APCFWD) using
     Skip/Count Trick  (Trick 1). If activeLength is greater
     than current edge length, set next  internal node as
     activeNode and adjust activeEdge and activeLength
```

```
     accordingly to represent same activePoint*/
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge += edgeLength(currNode);
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}

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

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

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

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

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

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

            /*A new leaf edge is created in above line starting
             from  an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
             reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
            // Get the next node at the end of edge starting
            // with activeEdge
            Node *next = activeNode->children[text[activeEdge]];
            if (walkDown(next))//Do walkdown
            {
```

```
}
    //Start from next node (the new activeNode)
    continue;
}
/*Extension Rule 3 (current character being processed
  is already on the edge)*/
if (text[next->start + activeLength] == text[pos])
{
    //If a newly created node waiting for it's
    //suffix link to be set, then set suffix link
    //of that waiting node to curent active node
    if(lastNewNode != NULL && activeNode != root)
    {
        lastNewNode->suffixLink = activeNode;
        lastNewNode = NULL;
    }

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

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

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

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

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

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

```
        }

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

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

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

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

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

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

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

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

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

```
int traverseEdge(char *str, int idx, int start, int end)
{
    int k = 0;
    //Traverse the edge with character by character matching
    for(k=start; k<=end && str[idx] != '\0'; k++, idx++)
    {
        if(text[k] != str[idx])
            return -1;  // mo match
    }
    if(str[idx] == '\0')
        return 1;  // match
    return 0;  // more characters yet to match
}

int doTraversal(Node *n, char* str, int idx)
{
    if(n == NULL)
    {
        return -1; // no match
    }
    int res = -1;
    //If node n is not root node, then traverse edge
    //from node n's parent to node n.
    if(n->start != -1)
    {
```

```
    {
        res = traverseEdge(str, idx, n->start, *(n->end));
        if(res != 0)
            return res;  // match (res = 1) or no match (res = -1)
    }
    //Get the character index to search
    idx = idx + edgeLength(n);
    //If there is an edge from node n going out
    //with current character str[idx], travrse that edge
    if(n->children[str[idx]] != NULL)
        return doTraversal(n->children[str[idx]], str, idx);
    else
        return -1;  // no match
}

void checkForSubString(char* str)
{
    int res = doTraversal(root, str, 0);
    if(res == 1)
        printf("Pattern <%s> is a Substring\n", str);
    else
        printf("Pattern <%s> is NOT a Substring\n", str);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "THIS IS A TEST TEXT$");
    buildSuffixTree();

    checkForSubString("TEST");
    checkForSubString("A");
    checkForSubString(" ");
    checkForSubString("IS A");
    checkForSubString(" IS A ");
    checkForSubString("TEST1");
    checkForSubString("THIS IS GOOD");
    checkForSubString("TES");
    checkForSubString("TESA");
    checkForSubString("ISB");

    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Output:

```
Pattern <TEST> is a Substring
Pattern <A> is a Substring
Pattern < > is a Substring
Pattern <IS A> is a Substring
Pattern < IS A > is a Substring
Pattern <TEST1> is NOT a Substring
Pattern <THIS IS GOOD> is NOT a Substring
Pattern <TES> is a Substring
Pattern <TESA> is NOT a Substring
```

```
Pattern <ISB> is NOT a Substring
```

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that, traversal for substring check takes O(M) for a pattern of length M.

With slight modification in traversal algorithm discussed here, we can answer following:

1. Find all occurrences of a given pattern P present in text T.
2. How to check if a pattern is prefix of a text?
3. How to check if a pattern is suffix of a text?

We have published following more articles on suffix tree applications:

- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 3 – Longest Repeated Substring
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Generalized Suffix Tree 1
- Suffix Tree Application 5 – Longest Common Substring
- Suffix Tree Application 6 – Longest Palindromic Substring

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

61. Suffix Tree Application 2 – Searching All Patterns

Given a text string and a pattern string, find all occurrences of the pattern in string.

Few pattern searching algorithms (KMP, Rabin-Karp, Naive Algorithm, Finite Automata) are already discussed, which can be used for this check.
Here we will discuss suffix tree based algorithm.

In the 1st Suffix Tree Application (Substring Check), we saw how to check whether a given pattern is substring of a text or not. It is advised to go through Substring Check 1st.
In this article, we will go a bit further on same problem. If a pattern is substring of a text, then we will find all the positions on pattern in the text.

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

Ukkonen's Suffix Tree Construction – Part 4
Ukkonen's Suffix Tree Construction – Part 5
Ukkonen's Suffix Tree Construction – Part 6

Lets look at following figure:

This is suffix tree for String "abcabxabcd$", showing suffix indices and edge label indices (start, end). The (sub)string value on edges are shown only for explanatory purpose. We never store path label string in the tree.
Suffix Index of a path tells the index of a substring (starting from root) on that path. Consider a path "bcd$" in above tree with suffix index 7. It tells that substrings b, bc, bcd, bcd$ are at index 7 in string.
Similarly path "bxabcd$" with suffix index 4 tells that substrings b, bx, bxa, bxab, bxabc, bxabcd, bxabcd$ are at index 4.
Similarly path "bcabxabcd$" with suffix index 1 tells that substrings b, bc, bca, bcab, bcabx, bcabxa, bcabxab, bcabxabc, bcabxabcd, bcabxabcd$ are at index 1.

If we see all the above three paths together, we can see that:

- Substring "b" is at indices 1, 4 and 7
- Substring "bc" is at indices 1 and 7

With above explanation, we should be able to see following:

- Substring "ab" is at indices 0, 3 and 6
- Substring "abc" is at indices 0 and 6
- Substring "c" is at indices 2 and 8
- Substring "xab" is at index 5
- Substring "d" is at index 9
- Substring "cd" is at index 8

…..
…..

Can you see how to find all the occurrences of a pattern in a string ?

1. 1st of all, check if the given pattern really exists in string or not (As we did in Substring Check). For this, traverse the suffix tree against the pattern.
2. If you find pattern in suffix tree (don't fall off the tree), then traverse the subtree below that point and find all suffix indices on leaf nodes. All those suffix indices will be pattern indices in string

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And find all locations of a pattern in string
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

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

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

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

typedef struct SuffixTreeNode Node;

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

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

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

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

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

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

```
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

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

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

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

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

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

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

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

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

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

```
                                    newNode(pos, &leafEnd);

        /*A new leaf edge is created in above line starting
         from  an existng node (the current activeNode), and
         if there is any internal node waiting for it's suffix
         link get reset, point the suffix link from that last
         internal node to current activeNode. Then set lastNewNode
         to NULL indicating no more node waiting for suffix link
         reset.*/
        if (lastNewNode != NULL)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }
    }
    // There is an outgoing edge starting with activeEdge
    // from activeNode
    else
    {
        // Get the next node at the end of edge starting
        // with activeEdge
        Node *next = activeNode->children[text[activeEdge]];
        if (walkDown(next))//Do walkdown
        {
            //Start from next node (the new activeNode)
            continue;
        }
        /*Extension Rule 3 (current character being processed
          is already on the edge)*/
        if (text[next->start + activeLength] == text[pos])
        {
            //If a newly created node waiting for it's
            //suffix link to be set, then set suffix link
            //of that waiting node to curent active node
            if(lastNewNode != NULL && activeNode != root)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }

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

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

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

        //New leaf coming out of new internal node
```

```
            split->children[text[pos]] = newNode(pos, &leafEnd);
            next->start += activeLength;
            split->children[text[next->start]] = next;

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

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

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

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

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

    if (n->start != -1) //A non-root node
    {
        //Print the label on edge from parent to current node
        //Uncomment below line to print suffix tree
       // print(n->start, *(n->end));
    }
    int leaf = 1;
```

```
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            //Uncomment below two lines to print suffix index
           // if (leaf == 1 && n->start != -1)
             //   printf(" [%d]\n", n->suffixIndex);

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

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

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

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

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

int traverseEdge(char *str, int idx, int start, int end)
```

```
int traverseEdge(char* str, int idx, int start, int end)
{
    int k = 0;
    //Traverse the edge with character by character matching
    for(k=start; k<=end && str[idx] != '\0'; k++, idx++)
    {
        if(text[k] != str[idx])
            return -1;  // mo match
    }
    if(str[idx] == '\0')
        return 1;  // match
    return 0;  // more characters yet to match
}

int doTraversalToCountLeaf(Node *n)
{
    if(n == NULL)
        return 0;
    if(n->suffixIndex > -1)
    {
        printf("\nFound at position: %d", n->suffixIndex);
        return 1;
    }
    int count = 0;
    int i = 0;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if(n->children[i] != NULL)
        {
            count += doTraversalToCountLeaf(n->children[i]);
        }
    }
    return count;
}

int countLeaf(Node *n)
{
    if(n == NULL)
        return 0;
    return doTraversalToCountLeaf(n);
}

int doTraversal(Node *n, char* str, int idx)
{
    if(n == NULL)
    {
        return -1; // no match
    }
    int res = -1;
    //If node n is not root node, then traverse edge
    //from node n's parent to node n.
    if(n->start != -1)
    {
        res = traverseEdge(str, idx, n->start, *(n->end));
        if(res == -1)  //no match
            return -1;
        if(res == 1) //match
        {
            if(n->suffixIndex > -1)
                printf("\nsubstring count: 1 and position: %d",
                                n->suffixIndex);
            else
                printf("\nsubstring count: %d", countLeaf(n));
```

```
            return 1;
        }
    }
    //Get the character index to search
    idx = idx + edgeLength(n);
    //If there is an edge from node n going out
    //with current character str[idx], travrse that edge
    if(n->children[str[idx]] != NULL)
        return doTraversal(n->children[str[idx]], str, idx);
    else
        return -1;  // no match
}

void checkForSubString(char* str)
{
    int res = doTraversal(root, str, 0);
    if(res == 1)
        printf("\nPattern <%s> is a Substring\n", str);
    else
        printf("\nPattern <%s> is NOT a Substring\n", str);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    printf("Text: GEEKSFORGEEKS, Pattern to search: GEEKS");
    checkForSubString("GEEKS");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: GEEK1");
    checkForSubString("GEEK1");
    printf("\n\nText: GEEKSFORGEEKS, Pattern to search: FOR");
    checkForSubString("FOR");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AABAACAADAABAAABAA$");
    buildSuffixTree();
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AABA");
    checkForSubString("AABA");
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AABAACAADAABAAABAA, Pattern to search: AAE");
    checkForSubString("AAE");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    strcpy(text, "AAAAAAAAA$");
    buildSuffixTree();
    printf("\n\nText: AAAAAAAAA, Pattern to search: AAAA");
    checkForSubString("AAAA");
    printf("\n\nText: AAAAAAAAA, Pattern to search: AA");
    checkForSubString("AA");
    printf("\n\nText: AAAAAAAAA, Pattern to search: A");
    checkForSubString("A");
    printf("\n\nText: AAAAAAAAA, Pattern to search: AB");
    checkForSubString("AB");
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Output:

```
Text: GEEKSFORGEEKS, Pattern to search: GEEKS
Found at position: 8
Found at position: 0
substring count: 2
Pattern <GEEKS> is a Substring


Text: GEEKSFORGEEKS, Pattern to search: GEEK1
Pattern <GEEK1> is NOT a Substring


Text: GEEKSFORGEEKS, Pattern to search: FOR
substring count: 1 and position: 5
Pattern <FOR> is a Substring


Text: AABAACAADAABAAABAA, Pattern to search: AABA
Found at position: 13
Found at position: 9
Found at position: 0
substring count: 3
Pattern <AABA> is a Substring


Text: AABAACAADAABAAABAA, Pattern to search: AA
Found at position: 16
Found at position: 12
Found at position: 13
Found at position: 9
Found at position: 0
Found at position: 3
Found at position: 6
substring count: 7
Pattern <AA> is a Substring


Text: AABAACAADAABAAABAA, Pattern to search: AAE
Pattern <AAE> is NOT a Substring


Text: AAAAAAAAA, Pattern to search: AAAA
Found at position: 5
```

```
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 6
Pattern <AAAA> is a Substring


Text: AAAAAAAAA, Pattern to search: AA
Found at position: 7
Found at position: 6
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 8
Pattern <AA> is a Substring


Text: AAAAAAAAA, Pattern to search: A
Found at position: 8
Found at position: 7
Found at position: 6
Found at position: 5
Found at position: 4
Found at position: 3
Found at position: 2
Found at position: 1
Found at position: 0
substring count: 9
Pattern <A> is a Substring


Text: AAAAAAAAA, Pattern to search: AB
Pattern <AB> is NOT a Substring
```

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that, traversal for substring check takes O(M) for a pattern of length M and then if there are Z occurrences of the pattern, it will take O(Z) to find indices of all those Z occurrences.
Overall pattern complexity is linear: O(M + Z).

A bit more detailed analysis
How many internal nodes will there in a suffix tree of string of length N ??
Answer: N-1 (Why ??)
There will be N suffixes in a string of length N.
Each suffix will have one leaf.
So a suffix tree of string of length N will have N leaves.
As each internal node has at least 2 children, an N-leaf suffix tree has at most N-1 internal nodes.
If a pattern occurs Z times in string, means it will be part of Z suffixes, so there will be Z leaves below in point (internal node and in between edge) where pattern match ends in tree and so subtree with Z leaves below that point will have Z-1 internal nodes. A tree with Z leaves can be traversed in O(Z) time.
Overall pattern complexity is linear: O(M + Z).
For a given pattern, Z (the number of occurrences) can be atmost N.
So worst case complexity can be: O(M + N) if Z is close/equal to N (A tree traversal with N nodes take O(N) time).

Followup questions:

1. Check if a pattern is prefix of a text?
2. Check if a pattern is suffix of a text?

We have published following more articles on suffix tree applications:

- Suffix Tree Application 1 – Substring Check
- Suffix Tree Application 3 – Longest Repeated Substring
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Generalized Suffix Tree 1
- Suffix Tree Application 5 – Longest Common Substring
- Suffix Tree Application 6 – Longest Palindromic Substring

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

62. Suffix Tree Application 3 – Longest Repeated Substring

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

```
Longest Repeated Substring in GEEKSFORGEEKS is: GEEKS
Longest Repeated Substring in AAAAAAAAAA is: AAAAAAAAA
Longest Repeated Substring in ABCDEFG is: No repeated substring
Longest Repeated Substring in ABABABA is: ABABA
```

```
Longest Repeated Substring in ATCGATCGA is: ATCGA
Longest Repeated Substring in banana is: ana
Longest Repeated Substring in abcpqrabpqpq is: ab (pq is another LRS here)
```

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

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

Lets look at following figure:

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

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

All above substrings are repeated.

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

Can you see how to find longest repeated substring ??

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

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

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

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

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

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

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

typedef struct SuffixTreeNode Node;

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

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

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

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
```

```
int *splitEnd = NULL;
int size = -1; //Length of input string

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

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

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

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

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

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

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

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

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

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

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

        /*A new leaf edge is created in above line starting
         from  an existng node (the current activeNode), and
         if there is any internal node waiting for it's suffix
         link get reset, point the suffix link from that last
         internal node to current activeNode. Then set lastNewNode
         to NULL indicating no more node waiting for suffix link
         reset.*/
        if (lastNewNode != NULL)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }
    }
    // There is an outgoing edge starting with activeEdge
    // from activeNode
    else
    {
        // Get the next node at the end of edge starting
        // with activeEdge
        Node *next = activeNode->children[text[activeEdge]];
        if (walkDown(next))//Do walkdown
        {
            //Start from next node (the new activeNode)
            continue;
        }
        /*Extension Rule 3 (current character being processed
          is already on the edge)*/
        if (text[next->start + activeLength] == text[pos])
        {
            //If a newly created node waiting for it's
            //suffix link to be set, then set suffix link
            //of that waiting node to curent active node
            if(lastNewNode != NULL && activeNode != root)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }

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

        /*We will be here when activePoint is in middle of
          the edge being traversed and current character
          being processed is not  on the edge (we fall off
```

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

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

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

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

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

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

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

    //Print the suffix tree as well along with setting suffix index
    //So tree will be printed in DFS manner
```

```
//Each edge along with it's suffix index will be printed
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL)  return;

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

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

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

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

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

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

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

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

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

```
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

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

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

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

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

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

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

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

    return 0;
}
```

Output:

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

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

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

Followup questions:

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

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

We have published following more articles on suffix tree applications:

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

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

63. Suffix Tree Application 4 – Build Linear Time Suffix Array

Given a string, build it's Suffix Array
We have already discussed following two ways of building suffix array:

- Naive O(n^2Logn) algorithm
- Enhanced O(nLogn) algorithm

Please go through these to have the basic understanding.
Here we will see how to build suffix array in linear time using suffix tree.

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

Ukkonen's Suffix Tree Construction – Part 2
Ukkonen's Suffix Tree Construction – Part 3
Ukkonen's Suffix Tree Construction – Part 4
Ukkonen's Suffix Tree Construction – Part 5
Ukkonen's Suffix Tree Construction – Part 6

Lets consider string abcabxabcd.
It's suffix array would be:

```
0 6 3 1 7 4 2 8 9 5
```

Lets look at following figure:

This is suffix tree for String "abcabxabcd$"
If we do a DFS traversal, visiting edges in lexicographic order (we have been doing the same traversal in other Suffix Tree Application articles as well) and print suffix indices on leaves, we will get following:

```
10 0 6 3 1 7 4 2 8 9 5
```

"$" is lexicographically lesser than [a-zA-Z].
The suffix index 10 corresponds to edge with "$" label.
Except this 1st suffix index, the sequence of all other numbers gives the suffix array of the string.

So if we have a suffix tree of the string, then to get it's suffix array, we just need to do a lexicographic order DFS traversal and store all the suffix indices in resultant suffix array, except the very 1st suffix index.

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And and then create suffix array in linear time
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

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

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

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

typedef struct SuffixTreeNode Node;

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

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

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

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

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

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

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

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

int walkDown(Node *currNode)
```

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

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

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

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

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

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

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

            /*A new leaf edge is created in above line starting
             from  an existng node (the current activeNode), and
             if there is any internal node waiting for it's suffix
             link get reset, point the suffix link from that last
             internal node to current activeNode. Then set lastNewNode
             to NULL indicating no more node waiting for suffix link
             reset.*/
            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }
        // There is an outgoing edge starting with activeEdge
        // from activeNode
        else
        {
```

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

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

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

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

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

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

    /*Make the current newly created internal node waiting
      for it's suffix link reset (which is pointing to root
      at present). If we come across any other internal node
      (existing or newly created) in next extension of same
```

```
                (existing of newly created) in next extension of same
                phase, when a new leaf edge gets added (i.e. when
                Extension Rule 2 applies is any of the next extension
                of same phase) at that point, suffixLink of this node
                will point to that internal node.*/
            lastNewNode = split;
        }

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

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

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

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

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
```

```
        //Uncomment below line to print suffix index
        //printf(" [%d]\n", n->suffixIndex);
    }
}

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

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

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

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

void doTraversal(Node *n, int suffixArray[], int *idx)
{
    if(n == NULL)
    {
        return;
    }
    int i=0;
    if(n->suffixIndex == -1) //If it is internal node
    {
        for (i = 0; i < MAX_CHAR; i++)
        {
            if(n->children[i] != NULL)
            {
                doTraversal(n->children[i], suffixArray, idx);
            }
        }
    }
    //If it is Leaf node other than "$" label
    else if(n->suffixIndex > -1 && n->suffixIndex < size)
    {
```

```
    {
        suffixArray[(*idx)++] = n->suffixIndex;
    }
}

void buildSuffixArray(int suffixArray[])
{
    int i = 0;
    for(i=0; i< size; i++)
        suffixArray[i] = -1;
    int idx = 0;
    doTraversal(root, suffixArray, &idx);
    printf("Suffix Array for String ");
    for(i=0; i<size; i++)
        printf("%c", text[i]);
    printf(" is: ");
    for(i=0; i<size; i++)
        printf("%d ", suffixArray[i]);
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    strcpy(text, "banana$");
    buildSuffixTree();
    size--;
    int *suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "GEEKSFORGEEKS$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "AAAAAAAAAA$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "ABCDEFG$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "ABABABA$");
    buildSuffixTree();
    size--;
```

```
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "abcabxabcd$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    strcpy(text, "CCAAACCCGATTA$");
    buildSuffixTree();
    size--;
    suffixArray =(int*) malloc(sizeof(int) * size);
    buildSuffixArray(suffixArray);
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
    free(suffixArray);

    return 0;
}
```

Output:

```
Suffix Array for String banana is: 5 3 1 0 4 2
Suffix Array for String GEEKSFORGEEKS is: 9 1 10 2 5 8 0 11 3 6 7 12 4
Suffix Array for String AAAAAAAAAA is: 9 8 7 6 5 4 3 2 1 0
Suffix Array for String ABCDEFG is: 0 1 2 3 4 5 6
Suffix Array for String ABABABA is: 6 4 2 0 5 3 1
Suffix Array for String abcabxabcd is: 0 6 3 1 7 4 2 8 9 5
Suffix Array for String CCAAACCCGATTA is: 12 2 3 4 9 1 0 5 6 7 8 11 10
```

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that, traversal of tree take O(N) to build suffix array.
So overall, it's linear in time and space.
Can you see why traversal is O(N) ?? Because a suffix tree of string of length N will have at most N-1 internal nodes and N leaves. Traversal of these nodes can be done in O(N).

We have published following more articles on suffix tree applications:

- Suffix Tree Application 1 – Substring Check
- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 3 – Longest Repeated Substring
- Generalized Suffix Tree 1
- Suffix Tree Application 5 – Longest Common Substring
- Suffix Tree Application 6 – Longest Palindromic Substring

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

64. Generalized Suffix Tree 1

In earlier suffix tree articles, we created suffix tree for one string and then we queried that tree for substring check, searching all patterns, longest repeated substring and built suffix array (All linear time operations).

There are lots of other problems where multiple strings are involved.
e.g. pattern searching in a text file or dictionary, spell checker, phone book, Autocomplete, Longest common substring problem, Longest palindromic substring and More.

For such operations, all the involved strings need to be indexed for faster search and retrieval. One way to do this is using suffix trie or suffix tree. We will discuss suffix tree here.
A suffix tree made of a set of strings is known as Generalized Suffix Tree.
We will discuss a simple way to build Generalized Suffix Tree here for **two strings only**.
Later, we will discuss another approach to build Generalized Suffix Tree for **two or more strings**.

Here we will use the suffix tree implementation for one string discussed already and modify that a bit to build generalized suffix tree.

Lets consider two strings X and Y for which we want to build generalized suffix tree. For this we will make a new string X#Y$ where # and $ both are terminal symbols (must be unique). Then we will build suffix tree for X#Y$ which will be the generalized suffix tree for X and Y. Same logic will apply for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string).

Lets say X = xabxa, and Y = babxba, then
X#Y$ = xabxa#babxba$
If we run the code implemented at Ukkonen's Suffix Tree Construction – Part 6 for string xabxa#babxba$, we get following output:

(Click to see it clearly)

We can use this tree to solve some of the problems, but we can refine it a bit by removing unwanted substrings on a path label. A path label should have substring from only one input string, so if there are path labels having substrings from multiple input strings, we can keep only the initial portion corresponding to one string and remove all

the later portion. For example, for path labels #babxba$, a#babxba$ and bxa#babxba$, we can remove babxba$ (belongs to 2nd input string) and then new path labels will be #, a# and bxa# respectively. With this change, above diagram will look like below:

(Click to see it clearly)

Below implementation is built on top of original implementation. Here we are removing unwanted characters on path labels. If a path label has "#" character in it, then we are trimming all characters after the "#" in that path label.

Note: This implementation builds generalized suffix tree for only two strings X and Y which are concatenated as X#Y$

```
// A C program to implement Ukkonen's Suffix Tree Construction
// And then build generalized suffix tree
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

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

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

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

typedef struct SuffixTreeNode Node;

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

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

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

```
int activeEdge = -1;
int activeLength = 0;

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

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

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

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

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

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

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

    /*Increment remainingSuffixCount indicating that a
```

```
new suffix added to the list of suffixes yet to be
added in tree*/
remainingSuffixCount++;

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

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

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

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

        /*A new leaf edge is created in above line starting
         from  an existng node (the current activeNode), and
         if there is any internal node waiting for it's suffix
         link get reset, point the suffix link from that last
         internal node to current activeNode. Then set lastNewNode
         to NULL indicating no more node waiting for suffix link
         reset.*/
        if (lastNewNode != NULL)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }
    }
    // There is an outgoing edge starting with activeEdge
    // from activeNode
    else
    {
        // Get the next node at the end of edge starting
        // with activeEdge
        Node *next = activeNode->children[text[activeEdge]];
        if (walkDown(next))//Do walkdown
        {
            //Start from next node (the new activeNode)
            continue;
        }
        /*Extension Rule 3 (current character being processed
          is already on the edge)*/
        if (text[next->start + activeLength] == text[pos])
        {
            //If a newly created node waiting for it's
            //suffix link to be set, then set suffix link
            //of that waiting node to curent active node
            if(lastNewNode != NULL && activeNode != root)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }

            //APCFER3
```

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

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

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

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

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

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

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

void pprint(int i, int j)
```

```
void print(int i, int j)
{
    int k;
    for (k=i; k<=j && text[k] != '#'; k++)
        printf("%c", text[k]);
    if(k<=j)
        printf("#");
}

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

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

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        for(i= n->start; i<= *(n->end); i++)
        {
            if(text[i] == '#') //Trim unwanted characters
            {
                n->end = (int*) malloc(sizeof(int));
                *(n->end) = i;
            }
        }
        n->suffixIndex = size - labelHeight;
        printf(" [%d]\n", n->suffixIndex);
    }
}

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

```
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}

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

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

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

    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);
}

// driver program to test above functions
int main(int argc, char *argv[])
{
//  strcpy(text, "xabxac#abcabxabcd$"); buildSuffixTree();
    strcpy(text, "xabxa#babxba$"); buildSuffixTree();
    return 0;
}
```

Output: (You can see that below output corresponds to the 2nd Figure shown above)

```
# [5]
$ [12]
a [-1]
# [4]
$ [11]
bx [-1]
a# [1]
ba$ [7]
b [-1]
a [-1]
$ [10]
bxba$ [6]
x [-1]
a# [2]
ba$ [8]
```

```
x [-1]
a [-1]
# [3]
bxa# [0]
ba$ [9]
```

If two strings are of size M and N, this implementation will take O(M+N) time and space. If input strings are not concatenated already, then it will take 2(M+N) space in total, M+N space to store the generalized suffix tree and another M+N space to store concatenated string.

Followup:
Extend above implementation for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string)

One problem with this approach is the need of unique terminal symbol for each input string. This will work for few strings but if there is too many input strings, we may not be able to find that many unique terminal symbols.
We will discuss another approach to build generalized suffix tree soon where we will need only one unique terminal symbol and that will resolve the above problem and can be used to build generalized suffix tree for any number of input strings.

We have published following more articles on suffix tree applications:

- Suffix Tree Application 1 – Substring Check
- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 3 – Longest Repeated Substring
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Suffix Tree Application 5 – Longest Common Substring
- Suffix Tree Application 6 – Longest Palindromic Substring

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

65. Suffix Tree Application 5 – Longest Common Substring

Given two strings X and Y, find the Longest Common Substring of X and Y.

Naive [O(N*M^2)] and Dynamic Programming [O(N*M)] approaches are already discussed here.

In this article, we will discuss a linear time approach to find LCS using suffix tree (The 5^{th} Suffix Tree Application).
Here we will build generalized suffix tree for two strings X and Y as discussed already at:

Generalized Suffix Tree 1

Lets take same example (X = xabxa, and Y = babxba) we saw in Generalized Suffix Tree 1.
We built following suffix tree for X and Y there:

(Click to see it clearly)

This is generalized suffix tree for xabxa#babxba$
In above, leaves with suffix indices in [0,4] are suffixes of string xabxa and leaves with suffix indices in [6,11] are suffixes of string babxa. Why ??
Because in concatenated string xabxa#babxba$, index of string xabxa is 0 and it's length is 5, so indices of it's suffixes would be 0, 1, 2, 3 and 4. Similarly index of string babxba is 6 and it's length is 6, so indices of it's suffixes would be 6, 7, 8, 9, 10 and 11.

With this, we can see that in the generalized suffix tree figure above, there are some internal nodes having leaves below it from

- both strings X and Y (i.e. there is at least one leaf with suffix index in [0,4] and one leaf with suffix index in [6, 11]
- string X only (i.e. all leaf nodes have suffix indices in [0,4])
- string Y only (i.e. all leaf nodes have suffix indices in [6,11])

Following figure shows the internal nodes marked as "XY", "X" or "Y" depending on which string the leaves belong to, that they have below themselves.

(Click to see it clearly)

What these "XY", "X" or "Y" marking mean ?
Path label from root to an internal node gives a substring of X or Y or both.
For node marked as XY, substring from root to that node belongs to both strings X and Y.
For node marked as X, substring from root to that node belongs to string X only.
For node marked as Y, substring from root to that node belongs to string Y only.

By looking at above figure, can you see how to get LCS of X and Y ?
By now, it should be clear that how to get common substring of X and Y at least.
If we traverse the path from root to nodes marked as XY, we will get common substring of X and Y.

Now we need to find the longest one among all those common substrings.
Can you think how to get LCS now ? Recall how did we get Longest Repeated Substring in a given string using suffix tree already.
The path label from root to the deepest node marked as XY will give the LCS of X and Y. The deepest node is highlighted in above figure and path label "abx" from root to that node is the LCS of X and Y.

```c
// A C program to implement Ukkonen's Suffix Tree Construction
// Here we build generalized suffix tree for two strings
// And then we find longest common substring of the two input strings
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

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

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

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

typedef struct SuffixTreeNode Node;

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

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

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

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string
int size1 = 0; //Size of 1st string

Node *newNode(int start, int *end)
{
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
```

```
    for (i = 0; i < MAX_CHAR; i++)
          node->children[i] = NULL;

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

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

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

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

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

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

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

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

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

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

    /*A new leaf edge is created in above line starting
     from  an existng node (the current activeNode), and
     if there is any internal node waiting for it's suffix
     link get reset, point the suffix link from that last
     internal node to current activeNode. Then set lastNewNode
     to NULL indicating no more node waiting for suffix link
     reset.*/
    if (lastNewNode != NULL)
    {
        lastNewNode->suffixLink = activeNode;
        lastNewNode = NULL;
    }
}
// There is an outgoing edge starting with activeEdge
// from activeNode
else
{
    // Get the next node at the end of edge starting
    // with activeEdge
    Node *next = activeNode->children[text[activeEdge]];
    if (walkDown(next))//Do walkdown
    {
        //Start from next node (the new activeNode)
        continue;
    }
    /*Extension Rule 3 (current character being processed
      is already on the edge)*/
    if (text[next->start + activeLength] == text[pos])
    {
        //If a newly created node waiting for it's
        //suffix link to be set, then set suffix link
        //of that waiting node to curent active node
        if(lastNewNode != NULL && activeNode != root)
        {
            lastNewNode->suffixLink = activeNode;
            lastNewNode = NULL;
        }

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;

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

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

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

                if(n->suffixIndex == -1)
                    n->suffixIndex = ret;
                else if((n->suffixIndex == -2 && ret == -3) ||
                    (n->suffixIndex == -3 && ret == -2) ||
                    n->suffixIndex == -4)
                {
                    n->suffixIndex = -4;//Mark node as XY
                    //Keep track of deepest node
                    if(*maxHeight < labelHeight)
                    {
                        *maxHeight = labelHeight;
                        *substringStartIndex = *(n->end) -
                            labelHeight + 1;
                    }
                }
            }
        }
    }
    else if(n->suffixIndex > -1 && n->suffixIndex < size1)//suffix of X
        return -2;//Mark node as X
    else if(n->suffixIndex >= size1)//suffix of Y
        return -3;//Mark node as Y
    return n->suffixIndex;
}

void getLongestCommonSubstring()
{
    int maxHeight = 0;
```

```
    int substringStartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringStartIndex);

    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringStartIndex]);
    if(k == 0)
        printf("No common substring");
    else
        printf(", of length: %d",maxHeight);
    printf("\n");
}

// driver program to test above functions
int main(int argc, char *argv[])
{
    size1 = 7;
    printf("Longest Common Substring in xabxac and abcabxabcd is: ");
    strcpy(text, "xabxac#abcabxabcd$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 10;
    printf("Longest Common Substring in xabxaabxa and babxba is: ");
    strcpy(text, "xabxaabxa#babxba$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 14;
    printf("Longest Common Substring in GeeksforGeeks and GeeksQuiz is
    strcpy(text, "GeeksforGeeks#GeeksQuiz$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 26;
    printf("Longest Common Substring in OldSite:GeeksforGeeks.org");
    printf(" and NewSite:GeeksQuiz.com is: ");
    strcpy(text, "OldSite:GeeksforGeeks.org#NewSite:GeeksQuiz.com$");
    buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Common Substring in abcde and fghie is: ");
    strcpy(text, "abcde#fghie$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Common Substring in pqrst and uvwxyz is: ");
    strcpy(text, "pqrst#uvwxyz$"); buildSuffixTree();
    getLongestCommonSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Output:

```
Longest Common Substring in xabxac and abcabxabcd is: abxa, of length: 4
Longest Common Substring in xabxaabxa and babxba is: abx, of length: 3
Longest Common Substring in GeeksforGeeks and GeeksQuiz is: Geeks, of length: 5
Longest Common Substring in OldSite:GeeksforGeeks.org and
NewSite:GeeksQuiz.com is: Site:Geeks, of length: 10
Longest Common Substring in abcde and fghie is: e, of length: 1
Longest Common Substring in pqrst and uvwxyz is: No common substring
```

If two strings are of size M and N, then Generalized Suffix Tree construction takes O(M+N) and LCS finding is a DFS on tree which is again O(M+N).
So overall complexity is linear in time and space.

Followup:

1. Given a pattern, check if it is substring of X or Y or both. If it is a substring, find all it's occurrences along with which string (X or Y or both) it belongs to.
2. Extend the implementation to find LCS of more than two strings
3. Solve problem 1 for more than two strings
4. Given a string, find it's Longest Palindromic Substring

We have published following more articles on suffix tree applications:

- Suffix Tree Application 1 – Substring Check
- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 3 – Longest Repeated Substring
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Suffix Tree Application 6 – Longest Palindromic Substring
- Generalized Suffix Tree 1

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

66. Check a given sentence for a given set of simple grammer rules

A simple sentence if syntactically correct if it fulfills given rules. The following are given rules.

1. Sentence must start with a Uppercase character (e.g. Noun/ I/ We/ He etc.)
2. Then lowercase character follows.
3. There must be spaces between words.

4. Then the sentence must end with a full stop(.) after a word.
5. Two continuous spaces are not allowed.
6. Two continuous upper case characters are not allowed.
7. However the sentence can end after an upper case character.

Examples:

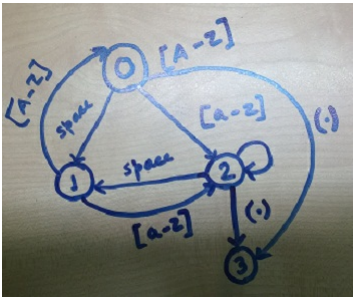
```
Correct sentences -
   "My name is Ram."
   "The vertex is S."
   "I am single."
   "I love Geeksquiz and Geeksforgeeks."

Incorrect sentence -
   "My name is KG."
   "I lovE cinema."
   "GeeksQuiz. is a quiz site."
   "  You are my friend."
   "I love cinema"
```

Question: Given a sentence, validate the given sentence for above given rules.

We strongly recommend to minimize the browser and try this yourself first.
The idea is to use an automata for the given set of rules.

Algorithm :
1. Check for the corner cases
…..1.a) Check if the first character is uppercase or not in the sentence.
…..1.b) Check if the last character is a full stop or not.

2. For rest of the string, this problem could be solved by following a state diagram.
Please refer to the below state diagram for that.



3. We need to maintain previous and current state of different characters in the string. Based on that we can always validate the sentence of every character traversed.

A C based implementation is below. (By the way this sentence is also correct according to the rule and code)

```c
// C program to validate a given sentence for a set of rules
#include<stdio.h>
#include<string.h>
#include<stdbool.h>

// Method to check a given sentence for given rules
bool checkSentence(char str[])
{
    // Calculate the length of the string.
    int len = strlen(str);

    // Check that the first character lies in [A-Z].
    // Otherwise return false.
    if (str[0] < 'A' || str[0] > 'Z')
        return false;

    //If the last character is not a full stop(.) no
    //need to check further.
    if (str[len - 1] != '.')
        return false;

    // Maintain 2 states. Previous and current state based
    // on which vertex state you are. Initialise both with
    // 0 = start state.
    int prev_state = 0, curr_state = 0;

    //Keep the index to the next character in the string.
    int index = 1;

    //Loop to go over the string.
    while (str[index])
    {
        // Set states according to the input characters in the
        // string and the rule defined in the description.
        // If current character is [A-Z]. Set current state as 0.
        if (str[index] >= 'A' && str[index] <= 'Z')
            curr_state = 0;

        // If current character is a space. Set current state as 1.
        else if (str[index] == ' ')
            curr_state = 1;

        // If current character is [a-z]. Set current state as 2.
        else if (str[index] >= 'a' && str[index] <= 'z')
            curr_state = 2;

        // If current state is a dot(.). Set current state as 3.
        else if (str[index] == '.')
            curr_state = 3;

        // Validates all current state with previous state for the
        // rules in the description of the problem.
        if (prev_state == curr_state && curr_state != 2)
            return false;

        if (prev_state == 2 && curr_state == 0)
            return false;

        // If we have reached last state and previous state is not 1,
        // then check next character. If next character is '\0', then
        // return true, else false
        if (curr_state == 3 && prev_state != 1)
```

```
            return (str[index + 1] == '\0');

        index++;

        // Set previous state as current state before going over
        // to the next character.
        prev_state = curr_state;
    }
    return false;
}

// Driver program
int main()
{
    char *str[] = { "I love cinema.", "The vertex is S.",
                    "I am single.", "My name is KG.",
                    "I lovE cinema.", "GeeksQuiz. is a quiz site.",
                    "I love Geeksquiz and Geeksforgeeks.",
                    "  You are my friend.", "I love cinema" };
    int str_size = sizeof(str) / sizeof(str[0]);
    int i = 0;
    for (i = 0; i < str_size; i++)
     checkSentence(str[i])? printf("\"%s\" is correct \n", str[i]):
                            printf("\"%s\" is incorrect \n", str[i]);

    return 0;
}
```

Output:

```
"I love cinema." is correct
"The vertex is S." is correct
"I am single." is correct
"My name is KG." is incorrect
"I lovE cinema." is incorrect
"GeeksQuiz. is a quiz site." is incorrect
"I love Geeksquiz and Geeksforgeeks." is correct
"  You are my friend." is incorrect
"I love cinema" is incorrect
```

Time complexity – O(n), worst case as we have to traverse the full sentence where n is the length of the sentence.
Auxiliary space – O(1)

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

67. Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 1

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

- if the given string is “forgeeksskeegfor”, the output should be “geeksskeeg”
- if the given string is “abaaba”, the output should be “abaaba”
- if the given string is “abababa”, the output should be “abababa”
- if the given string is “abcbabcbabcba”, the output should be “abcbabcba”

We have already discussed Naïve [$O(n^3)$] and quadratic [$O(n^2)$] approaches at Set 1 and Set 2.
In this article, we will talk about Manacher’s algorithm which finds Longest Palindromic Substring in linear time.
One way (Set 2) to find a palindrome is to start from the center of the string and compare characters in both directions one by one. If corresponding characters on both sides (left and right of the center) match, then they will make a palindrome.
Let’s consider string “abababa”.
Here center of the string is 4th character (with index 3) b. If we match characters in left and right of the center, all characters match and so string “abababa” is a palindrome.

Here center position is not only the actual string character position but it could be the position between two characters also.
Consider string “abaaba” of even length. This string is palindrome around the position between 3rd and 4th characters a and a respectively.

To find Longest Palindromic Substring of a string of length N, one way is take each possible 2*N + 1 centers (the N character positions, N-1 between two character positions and 2 positions at left and right ends), do the character match in both left and right directions at each 2*N+ 1 centers and keep track of LPS. This approach takes O(N^2) time and that’s what we are doing in Set 2.

Let’s consider two strings “abababa” and “abaaba” as shown below:

In these two strings, left and right side of the center positions (position 7 in 1st string and position 6 in 2nd string) are symmetric. Why? Because the whole string is palindrome around the center position.
If we need to calculate Longest Palindromic Substring at each 2*N+1 positions from left to right, then palindrome’s symmetric property could help to avoid some of the unnecessary computations (i.e. character comparison). If there is a palindrome of some length L cantered at any position P, then we may not need to compare all characters in left and right side at position P+1. We already calculated LPS at positions before P and

they can help to avoid some of the comparisons after position P.
This use of information from previous positions at a later point of time makes the Manacher's algorithm linear. In Set 2, there is no reuse of previous information and so that is quadratic.

Manacher's algorithm is probably considered complex to understand, so here we will discuss it in as detailed way as we can. Some of it's portions may require multiple reading to understand it properly.

Let's look at string "abababa". In 3rd figure above, 15 center positions are shown. We need to calculate length of longest palindromic string at each of these positions.

- At position 0, there is no LPS at all (no character on left side to compare), so length of LPS will be 0.
- At position 1, LPS is a, so length of LPS will be 1.
- At position 2, there is no LPS at all (left and right characters a and b don't match), so length of LPS will be 0.
- At position 3, LPS is aba, so length of LPS will be 3.
- At position 4, there is no LPS at all (left and right characters b and a don't match), so length of LPS will be 0.
- At position 5, LPS is ababa, so length of LPS will be 5.

…… and so on

We store all these palindromic lengths in an array, say L. Then string S and LPS Length L look like below:

Similarly, LPS Length L of string "abaaba" will look like:

In LPS Array L:

- LPS length value at odd positions (the actual character positions) will be odd and greater than or equal to 1 (1 will come from the center character itself if nothing else matches in left and right side of it)
- LPS length value at even positions (the positions between two characters, extreme left and right positions) will be even and greater than or equal to 0 (0 will come when there is no match in left and right side)

Position and index for the string are two different things here. For a given string S of length N, indexes will be from 0 to N-1 (total N indexes) and positions will be from 0 to 2*N (total 2*N+1 positions).

LPS length value can be interpreted in two ways, one in terms of index and second in terms of position. LPS value d at position I (L[i] = d) tells that:

- Substring from position i-d to i+d is a palindrome of length d (in terms of position)
- Substring from index (i-d)/2 to [(i+d)/2 – 1] is a palindrome of length d (in terms of index)

e.g. in string “abaaba”, L[3] = 3 means substring from position 0 (3-3) to 6 (3+3) is a palindrome which is “aba” of length 3, it also means that substring from index 0 [(3-3)/2] to 2 [(3+3)/2 – 1] is a palindrome which is “aba” of length 3.

Now the main task is to compute LPS array efficiently. Once this array is computed, LPS of string S will be centered at position with maximum LPS length value.
We will see it in Part 2.

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

68. Manacher’s Algorithm – Linear Time Longest Palindromic Substring – Part 2

In Manacher’s Algorithm – Part 1, we gone through some of the basics and LPS length array.
Here we will see how to calculate LPS length array efficiently.

To calculate LPS array efficiently, we need to understand how LPS length for any position may relate to LPS length value of any previous already calculated position.
For string “abaaba”, we see following:

If we look around position 3:

- LPS length value at position 2 and position 4 are same
- LPS length value at position 1 and position 5 are same

We calculate LPS length values from left to right starting from position 0, so we can see if we already know LPS length values at positions 1, 2 and 3 already then we may not need to calculate LPS length at positions 4 and 5 because they are equal to LPS length values at corresponding positions on left side of position 3.

If we look around position 6:

- LPS length value at position 5 and position 7 are same
- LPS length value at position 4 and position 8 are same

……. and so on.
If we already know LPS length values at positions 1, 2, 3, 4, 5 and 6 already then we may not need to calculate LPS length at positions 7, 8, 9, 10 and 11 because they are equal to LPS length values at corresponding positions on left side of position 6.
For string “abababa”, we see following:

If we already know LPS length values at positions 1, 2, 3, 4, 5, 6 and 7 already then we may not need to calculate LPS length at positions 8, 9, 10, 11, 12 and 13 because they are equal to LPS length values at corresponding positions on left side of position 7.

Can you see why LPS length values are symmetric around positions 3, 6, 9 in string “abaaba”? That’s because there is a palindromic substring around these positions.
Same is the case in string “abababa” around position 7.
Is it always true that LPS length values around at palindromic center position are always symmetric (same)?
Answer is NO.
Look at positions 3 and 11 in string “abababa”. Both positions have LPS length 3. Immediate left and right positions are symmetric (with value 0), but not the next one. Positions 1 and 5 (around position 3) are not symmetric. Similarly, positions 9 and 13 (around position 11) are not symmetric.

At this point, we can see that if there is a palindrome in a string centered at some position, then LPS length values around the center position may or may not be symmetric depending on some situation. If we can identify the situation when left and right positions WILL BE SYMMETRIC around the center position, we NEED NOT calculate LPS length of the right position because it will be exactly same as LPS value of corresponding position on the left side which is already known. And this fact where we are avoiding LPS length computation at few positions makes Manacher’s Algorithm linear.

In situations when left and right positions WILL NOT BE SYMMETRIC around the center position, we compare characters in left and right side to find palindrome, but here also algorithm tries to avoid certain no of comparisons. We will see all these scenarios soon.

Let’s introduce few terms to proceed further:

(click to see it clearly)

- **centerPosition** – This is the position for which LPS length is calculated and let’s say LPS length at centerPosition is d (i.e. L[centerPosition] = d)
- **centerRightPosition** – This is the position which is right to the centerPosition and d position away from centerPosition (i.e. **centerRightPosition = centerPosition + d**)
- **centerLeftPosition** – This is the position which is left to the centerPosition and d

position away from centerPosition (i.e. **centerLeftPosition = centerPosition – d**)

- **currentRightPosition** – This is the position which is right of the centerPosition for which LPS length is not yet known and has to be calculated
- **currentLeftPosition** – This is the position on the left side of centerPosition which corresponds to the currentRightPosition
  centerPosition – currentLeftPosition = currentRightPosition – centerPosition
  currentLeftPosition = 2* centerPosition – currentRightPosition
- **i-left palindrome** – The palindrome i positions left of centerPosition, i.e. at currentLeftPosition
- **i-right palindrome** – The palindrome i positions right of centerPosition, i.e. at currentRightPosition
- **center palindrome** – The palindrome at centerPosition

When we are at centerPosition for which LPS length is known, then we also know LPS length of all positions smaller than centerPosition. Let's say LPS length at centerPosition is d, i.e.
L[centerPosition] = d

It means that substring between positions "centerPosition-d" to "centerPosition+d" is a palindrom.
Now we proceed further to calculate LPS length of positions greater than centerPosition. Let's say we are at currentRightPosition (> centerPosition) where we need to find LPS length.
For this we look at LPS length of currentLeftPosition which is already calculated.

If LPS length of currentLeftPosition is less than "centerRightPosition – currentRightPosition", then LPS length of currentRightPosition will be equal to LPS length of currentLeftPosition. So
L[currentRightPosition] = L[currentLeftPosition] if L[currentLeftPosition] < centerRightPosition – currentRightPosition. This is **Case 1**.

Let's consider below scenario for string "abababa":

(click to see it clearly)

We have calculated LPS length up-to position 7 where L[7] = 7, if we consider position 7 as centerPosition, then centerLeftPosition will be 0 and centerRightPosition will be 14. Now we need to calculate LPS length of other positions on the right of centerPosition.

For currentRightPosition = 8, currentLeftPosition is 6 and L[currentLeftPosition] = 0
Also centerRightPosition – currentRightPosition = 14 – 8 = 6
Case 1 applies here and so L[currentRightPosition] = L[8] = 0
Case 1 applies to positions 10 and 12, so,
L[10] = L[4] = 0
L[12] = L[2] = 0

If we look at position 9, then:
currentRightPosition = 9
currentLeftPosition = 2* centerPosition – currentRightPosition = 2*7 – 9 = 5
centerRightPosition – currentRightPosition = 14 – 9 = 5

Here L[currentLeftPosition] = centerRightPosition – currentRightPosition, so Case 1 doesn't apply here. Also note that centerRightPosition is the extreme end position of the string. That means center palindrome is suffix of input string. In that case, L[currentRightPosition] = L[currentLeftPosition]. This is **Case 2**.

Case 2 applies to positions 9, 11, 13 and 14, so:
L[9] = L[5] = 5
L[11] = L[3] = 3
L[13] = L[1] = 1
L[14] = L[0] = 0

What is really happening in Case 1 and Case 2? This is just utilizing the palindromic symmetric property and without any character match, it is finding LPS length of new positions.

When a bigger length palindrome contains a smaller length palindrome centered at left side of it's own center, then based on symmetric property, there will be another same smaller palindrome centered on the right of bigger palindrome center. If left side smaller palindrome is not prefix of bigger palindrome, then **Case 1** applies and if it is a prefix AND bigger palindrome is suffix of the input string itself, then **Case 2** applies.

*The longest palindrome i places to the right of the current center (the i-right palindrome) is as long as the longest palindrome i places to the left of the current center (the i-left palindrome) if the i-left palindrome is completely contained in the longest palindrome around the current center (the center palindrome) and the i-left palindrome is not a prefix of the center palindrome (**Case 1**) or (i.e. when i-left palindrome is a prefix of center palindrome) if the center palindrome is a suffix of the entire string (**Case 2**).*

In Case 1 and Case 2, i-right palindrome can't expand more than corresponding i-left palindrome (can you visualize why it can't expand more?), and so LPS length of i-right palindrome is exactly same as LPS length of i-left palindrome.

Here both i-left and i-right palindromes are completely contained in center palindrome (i.e. L[currentLeftPosition] <= centerRightPosition – currentRightPosition)
Now if i-left palindrome is not a prefix of center palindrome (L[currentLeftPosition] < centerRightPosition – currentRightPosition), that means that i-left palindrome was not able to expand up-to position centerLeftPosition.

If we look at following with centerPosition = 11, then

(click to see it clearly)

centerLeftPosition would be 11 – 9 = 2, and centerRightPosition would be 11 + 9 = 20
If we take currentRightPosition = 15, it's currentLeftPosition is 7. Case 1 applies here and so L[15] = 3. i-left palindrome at position 7 is "bab" which is completely contained in center palindrome at position 11 (which is "dbabcbabd"). We can see that i-right palindrome (at position 15) can't expand more than i-left palindrome (at position 7).

If there was a possibility of expansion, i-left palindrome could have expanded itself more already. But there is no such possibility as i-left palindrome is prefix of center palindrome. So due to symmetry property, i-right palindrome will be exactly same as i-left palindrome and it can't expand more. This makes L[currentRightPosition] = L[currentLeftPosition] in Case 1.

Now if we consider centerPosition = 19, then centerLeftPosition = 12 and centerRightPosition = 26
If we take currentRightPosition = 23, it's currentLeftPosition is 15. Case 2 applies here and so L[23] = 3. i-left palindrome at position 15 is "bab" which is completely contained in center palindrome at position 19 (which is "babdbab"). In Case 2, where i-left palindrome is prefix of center palindrome, i-right palindrome can't expand more than length of i-left palindrome because center palindrome is suffix of input string so there are no more character left to compare and expand. This makes L[currentRightPosition] = L[currentLeftPosition] in Case 2.

Case 1: L[currentRightPosition] = L[currentLeftPosition] applies when:

- i-left palindrome is completely contained in center palindrome
- i-left palindrome is NOT a prefix of center palindrome

Both above conditions are satisfied when
L[currentLeftPosition] < centerRightPosition – currentRightPosition

Case 2: L[currentRightPosition] = L[currentLeftPosition] applies when:

- i-left palindrome is prefix of center palindrome (means completely contained also)
- center palindrome is suffix of input string

Above conditions are satisfied when

L[currentLeftPosition] = centerRightPosition – currentRightPosition (For 1st condition)
AND

centerRightPosition = 2*N where N is input string length N (For 2nd condition).

Case 3: L[currentRightPosition] > = L[currentLeftPosition] applies when:

- i-left palindrome is prefix of center palindrome (and so i-left palindrome is completely contained in center palindrome)
- center palindrome is NOT suffix of input string

Above conditions are satisfied when

L[currentLeftPosition] = centerRightPosition – currentRightPosition (For 1st condition) AND

centerRightPosition < 2*N where N is input string length N (For 2nd condition).
In this case, there is a possibility of i-right palindrome expansion and so length of i-right palindrome is at least as long as length of i-left palindrome.

Case 4: L[currentRightPosition] > = centerRightPosition – currentRightPosition applies when:

- i-left palindrome is NOT completely contained in center palindrome

Above condition is satisfied when
L[currentLeftPosition] > centerRightPosition – currentRightPosition
In this case, length of i-right palindrome is at least as long (centerRightPosition – currentRightPosition) and there is a possibility of i-right palindrome expansion.

In following figure,

(click to see it clearly)

If we take center position 7, then Case 3 applies at currentRightPosition 11 because i-left palindrome at currentLeftPosition 3 is a prefix of center palindrome and i-right palindrome is not suffix of input string, so here L[11] = 9, which is greater than i-left palindrome length L[3] = 3. In the case, it is guaranteed that L[11] will be at least 3, and so in implementation, we 1st set L[11] = 3 and then we try to expand it by comparing characters in left and right side starting from distance 4 (As up-to distance 3, it is already known that characters will match).

If we take center position 11, then Case 4 applies at currentRightPosition 15 because L[currentLeftPosition] = L[7] = 7 > centerRightPosition – currentRightPosition = 20 – 15 = 5. In the case, it is guaranteed that L[15] will be at least 5, and so in implementation, we 1st set L[15] = 5 and then we try to expand it by comparing characters in left and right side starting from distance 5 (As up-to distance 5, it is already known that characters will match).

Now one point left to discuss is, when we work at one center position and compute LPS lengths for different rightPositions, how to know that what would be next center position. We change centerPosition to currentRightPosition if palindrome centered at currentRightPosition expands beyond centerRightPosition.

Here we have seen four different cases on how LPS length of a position will depend on a previous position's LPS length.
In Part 3, we have discussed code implementation of it and also we have looked at these four cases in a different way and implement that too.

This article is contributed by **Anurag Singh**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above

69. Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 3

In Manacher's Algorithm Part 1 and Part 2, we gone through some of the basics, understood LPS length array and how to calculate it efficiently based on four cases. Here we will implement the same.

We have seen that there are no new character comparison needed in case 1 and case 2. In case 3 and case 4, necessary new comparison are needed.
In following figure,

(click to see it clearly)

If at all we need a comparison, we will only compare actual characters, which are at "odd" positions like 1, 3, 5, 7, etc.
Even positions do not represent a character in string, so no comparison will be preformed for even positions.
If two characters at different odd positions match, then they will increase LPS length by 2.

There are many ways to implement this depending on how even and odd positions are handled. One way would be to create a new string 1^{st} where we insert some unique character (say #, $ etc) in all even positions and then run algorithm on that (to avoid different way of even and odd position handling). Other way could be to work on given string itself but here even and odd positions should be handled appropriately.

Here we will start with given string itself. When there is a need of expansion and character comparison required, we will expand in left and right positions one by one. When odd position is found, comparison will be done and LPS Length will be incremented by ONE. When even position is found, no comparison done and LPS Length will be incremented by ONE (So overall, one odd and one even positions on both left and right side will increase LPS Length by TWO).

```
// A C program to implement Manacher's Algorithm
#include <stdio.h>
#include <string.h>

char text[100];
void findLongestPalindromicString()
{
    int N = strlen(text);
    if(N == 0)
        return;
```

```
    return;
N = 2*N + 1; //Position count
int L[N]; //LPS Length Array
L[0] = 0;
L[1] = 1;
int C = 1; //centerPosition
int R = 2; //centerRightPosition
int i = 0; //currentRightPosition
int iMirror; //currentLeftPosition
int expand = -1;
int diff = -1;
int maxLPSLength = 0;
int maxLPSCenterPosition = 0;
int start = -1;
int end = -1;

//Uncomment it to print LPS Length array
//printf("%d %d ", L[0], L[1]);
for (i = 2; i < N; i++)
{
    //get currentLeftPosition iMirror for currentRightPosition i
    iMirror  = 2*C-i;
    //Reset expand - means no expansion required
    expand = 0;
    diff = R - i;
    //If currentRightPosition i is within centerRightPosition R
    if(diff > 0)
    {
        if(L[iMirror] < diff) // Case 1
            L[i] = L[iMirror];
        else if(L[iMirror] == diff && i == N-1) // Case 2
            L[i] = L[iMirror];
        else if(L[iMirror] == diff && i < N-1)  // Case 3
        {
                L[i] = L[iMirror];
                expand = 1;  // expansion required
        }
        else if(L[iMirror] > diff)  // Case 4
        {
            L[i] = diff;
            expand = 1;  // expansion required
        }
    }
    else
    {
        L[i] = 0;
        expand = 1;  // expansion required
    }

    if(expand == 1)
    {
        //Attempt to expand palindrome centered at currentRightPos
        //Here for odd positions, we compare characters and
        //if match then increment LPS Length by ONE
        //If even position, we just increment LPS by ONE without
        //any character comparison
        while ( ((i + L[i]) < N && (i - L[i]) > 0) &&
            ( ((i + L[i] + 1) % 2 == 0) ||
            (text[(i + L[i] + 1)/2] == text[(i - L[i] - 1)/2] )))
        {
            L[i]++;
        }
    }
```

```
        if(L[i] > maxLPSLength)  // Track maxLPSLength
        {
            maxLPSLength = L[i];
            maxLPSCenterPosition = i;
        }

        // If palindrome centered at currentRightPosition i
        // expand beyond centerRightPosition R,
        // adjust centerPosition C based on expanded palindrome.
        if (i + L[i] > R)
        {
            C = i;
            R = i + L[i];
        }
        //Uncomment it to print LPS Length array
        //printf("%d ", L[i]);
    }
    //printf("\n");
    start = (maxLPSCenterPosition - maxLPSLength)/2;
    end = start + maxLPSLength - 1;
    //printf("start: %d end: %d\n", start, end);
    printf("LPS of string is %s : ", text);
    for(i=start; i<=end; i++)
        printf("%c", text[i]);
    printf("\n");
}


int main(int argc, char *argv[])
{

    strcpy(text, "babcbabcbaccba");
    findLongestPalindromicString();

    strcpy(text, "abaaba");
    findLongestPalindromicString();

    strcpy(text, "abababa");
    findLongestPalindromicString();

    strcpy(text, "abcbabcbabcba");
    findLongestPalindromicString();

    strcpy(text, "forgeeksskeegfor");
    findLongestPalindromicString();

    strcpy(text, "caba");
    findLongestPalindromicString();

    strcpy(text, "abacdfgdcaba");
    findLongestPalindromicString();

    strcpy(text, "abacdfgdcabba");
    findLongestPalindromicString();

    strcpy(text, "abacdedcaba");
    findLongestPalindromicString();

    return 0;
}
```

Output:

```
LPS of string is babcbabcbaccba : abcbabcba
LPS of string is abaaba : abaaba
LPS of string is abababa : abababa
LPS of string is abcbabcbabcba : abcbabcbabcba
LPS of string is forgeeksskeegfor : geeksskeeg
LPS of string is caba : aba
LPS of string is abacdfgdcaba : aba
LPS of string is abacdfgdcabba : abba
LPS of string is abacdedcaba : abacdedcaba
```

This is the implementation based on the four cases discussed in Part 2. In Part 4, we have discussed a different way to look at these four cases and few other approaches.

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

70. Longest Even Length Substring such that Sum of First and Second Half is same

Given a string 'str' of digits, find length of the longest substring of 'str', such that the length of the substring is 2k digits and sum of left k digits is equal to the sum of right k digits.

Examples:

```
Input: str = "123123"
Output: 6
The complete string is of even length and sum of first and second
half digits is same

Input: str = "1538023"
Output: 4
The longest substring with same first and second half sum is "5380"
```

A **Simple Solution** is to check every substring of even length. The following is C based implementation of simple approach.

```
// A simple C based program to find length of longest  even length
// substring with same sum of digits in left and right
#include<stdio.h>
#include<string.h>

int findLength(char *str)
{
    int n = strlen(str);
    int maxlen =0;  // Initialize result

    // Choose starting point of every substring
    for (int i=0; i<n; i++)
    {
        // Choose ending point of even length substring
        for (int j =i+1; j<n; j += 2)
        {
            int length = j-i+1;//Find length of current substr

            // Calculate left & right sums for current substr
            int leftsum = 0, rightsum =0;
            for (int k =0; k<length/2; k++)
            {
                leftsum  += (str[i+k]-'0');
                rightsum += (str[i+k+length/2]-'0');
            }

            // Update result if needed
            if (leftsum == rightsum && maxlen < length)
                    maxlen = length;
        }
    }
    return maxlen;
}

// Driver program to test above function
int main(void)
{
    char str[] = "1538023";
    printf("Length of the substring is %d", findLength(str));
    return 0;
}
```

Output:

```
Length of the substring is 4
```

The time complexity of above solution is $O(n^3)$. The above solution can be optimized to work in $O(n^2)$ using **Dynamic Programming**. The idea is to build a 2D table that stores sums of substrings. The following is C based implementation of Dynamic Programming approach.

```
// A C based program that uses Dynamic Programming to find length of tl
// longest even substring with same sum of digits in left and right ha
#include <stdio.h>
#include <string.h>

int findLength(char *str)
{
    int n = strlen(str);
    int maxlen = 0; // Initialize result

    // A 2D table where sum[i][j] stores sum of digits
    // from str[i] to str[j].  Only filled entries are
    // the entries where j >= i
    int sum[n][n];

    // Fill the diagonal values for sunstrings of length 1
    for (int i =0; i<n; i++)
        sum[i][i] = str[i]-'0';

    // Fill entries for substrings of length 2 to n
    for (int len=2; len<=n; len++)
    {
        // Pick i and j for current substring
        for (int i=0; i<n-len+1; i++)
        {
            int j = i+len-1;
            int k = len/2;

            // Calculate value of sum[i][j]
            sum[i][j] = sum[i][j-k] + sum[j-k+1][j];

            // Update result if 'len' is even, left and right
            // sums are same and len is more than maxlen
            if (len%2 == 0 && sum[i][j-k] == sum[(j-k+1)][j]
                           && len > maxlen)
                maxlen = len;
        }
    }
    return maxlen;
}

// Driver program to test above function
int main(void)
{
    char str[] = "153803";
    printf("Length of the substring is %d", findLength(str));
    return 0;
}
```

Output:

```
Length of the substring is 4
```

Time complexity of the above solution is $O(n^2)$, but it requires $O(n^2)$ extra space.

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

71. Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 4

In Manacher's Algorithm Part 1 and Part 2, we gone through some of the basics, understood LPS length array and how to calculate it efficiently based on four cases. In Part 3, we implemented the same.
Here we will review the four cases again and try to see it differently and implement the same.
All four cases depends on LPS length value at currentLeftPosition (L[iMirror]) and value of (centerRightPosition – currentRightPosition), i.e. (R – i). These two information are know before which helps us to reuse previous available information and avoid unnecessary character comparison.

(click to see it clearly)

*If we look at all four cases, we will see that we 1st set minimum of **L[iMirror]** and **R-i** to **L[i]** and then we try to expand the palindrome in whichever case it can expand.*

Above observation may look more intuitive, easier to understand and implement, given that one understands LPS length array, position, index, symmetry property etc.

```
// A C program to implement Manacher's Algorithm
#include <stdio.h>
#include <string.h>

char text[100];
int min(int a, int b)
{
    int res = a;
    if(b < a)
        res = b;
    return res;
}

void findLongestPalindromicString()
{
    int N = strlen(text);
    if(N == 0)
        return;
    N = 2*N + 1; //Position count
    int L[N]; //LPS Length Array
    L[0] = 0;
    L[1] = 1;
    int C = 1; //centerPosition
    int R = 2; //centerRightPosition
    int i = 0; //currentRightPosition
    int iMirror; //currentLeftPosition
    int maxLPSLength = 0;
    int maxLPSCenterPosition = 0;
    int start = -1;
```

```
    int end = -1;
    int diff = -1;

    //Uncomment it to print LPS Length array
    //printf("%d %d ", L[0], L[1]);
    for (i = 2; i < N; i++)
    {
        //get currentLeftPosition iMirror for currentRightPosition i
        iMirror  = 2*C-i;
        L[i] = 0;
        diff = R - i;
        //If currentRightPosition i is within centerRightPosition R
        if(diff > 0)
            L[i] = min(L[iMirror], diff);

        //Attempt to expand palindrome centered at currentRightPositio
        //Here for odd positions, we compare characters and
        //if match then increment LPS Length by ONE
        //If even position, we just increment LPS by ONE without
        //any character comparison
        while ( ((i + L[i]) < N && (i - L[i]) > 0) &&
            ( ((i + L[i] + 1) % 2 == 0) ||
            (text[(i + L[i] + 1)/2] == text[(i - L[i] - 1)/2] )))
        {
            L[i]++;
        }

        if(L[i] > maxLPSLength)  // Track maxLPSLength
        {
            maxLPSLength = L[i];
            maxLPSCenterPosition = i;
        }

        //If palindrome centered at currentRightPosition i
        //expand beyond centerRightPosition R,
        //adjust centerPosition C based on expanded palindrome.
        if (i + L[i] > R)
        {
            C = i;
            R = i + L[i];
        }
        //Uncomment it to print LPS Length array
        //printf("%d ", L[i]);
    }
    //printf("\n");
    start = (maxLPSCenterPosition - maxLPSLength)/2;
    end = start + maxLPSLength - 1;
    printf("LPS of string is %s : ", text);
    for(i=start; i<=end; i++)
        printf("%c", text[i]);
    printf("\n");
}

int main(int argc, char *argv[])
{

    strcpy(text, "babcbabcbaccba");
    findLongestPalindromicString();

    strcpy(text, "abaaba");
    findLongestPalindromicString();
```

```
    strcpy(text, "abababa");
    findLongestPalindromicString();

    strcpy(text, "abcbabcbabcba");
    findLongestPalindromicString();

    strcpy(text, "forgeeksskeegfor");
    findLongestPalindromicString();

    strcpy(text, "caba");
    findLongestPalindromicString();

    strcpy(text, "abacdfgdcaba");
    findLongestPalindromicString();

    strcpy(text, "abacdfgdcabba");
    findLongestPalindromicString();

    strcpy(text, "abacdedcaba");
    findLongestPalindromicString();

    return 0;
}
```

Output:

```
LPS of string is babcbabcbaccba : abcbabcba
LPS of string is abaaba : abaaba
LPS of string is abababa : abababa
LPS of string is abcbabcbabcba : abcbabcbabcba
LPS of string is forgeeksskeegfor : geeksskeeg
LPS of string is caba : aba
LPS of string is abacdfgdcaba : aba
LPS of string is abacdfgdcabba : abba
LPS of string is abacdedcaba : abacdedcaba
```

Other Approaches

We have discussed two approaches here. One in Part 3 and other in current article. In both approaches, we worked on given string. Here we had to handle even and odd positions differently while comparing characters for expansion (because even positions do not represent any character in string).

To avoid this different handling of even and odd positions, we need to make even positions also to represent some character (actually all even positions should represent SAME character because they MUST match while character comparison). One way to do this is to set some character at all even positions by modifying given string or create a new copy of given string. For example, if input string is “abcb”, new string should be “#a#b#c#b#” if we add # as unique character at even positions.

The two approaches discussed already can be modified a bit to work on modified string where different handling of even and odd positions will not be needed.

We may also add two DIFFERENT characters (not yet used anywhere in string at even

and odd positions) at start and end of string as sentinels to avoid bound check. With these changes string “abcb” will look like “^#a#b#c#b#$” where ^ and $ are sentinels. This implementation may look cleaner with the cost of more memory.

We are not implementing these here as it’s a simple change in given implementations.

Implementation of approach discussed in current article on a modified string can be found at Longest Palindromic Substring Part II and a Java Translation of the same by Princeton.

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

72. Print all possible strings that can be made by placing spaces

Given a string you need to print all possible strings that can be made by placing spaces (zero or one) in between them.

```
Input:  str[] = "ABC"
Output: ABC
        AB C
        A BC
        A B C
```

Source: Amazon Interview Experience | Set 158, Round 1 ,Q 1.

We strongly recommend to minimize your browser and try this yourself first.

The idea is to use recursion and create a buffer that one by one contains all output strings having spaces. We keep updating buffer in every recursive call. If the length of given string is ‘n’ our updated string can have maximum length of n + (n-1) i.e. 2n-1. So we create buffer size of 2n (one extra character for string termination).
We leave 1st character as it is, starting from the 2nd character, we can either fill a space or a character. Thus one can write a recursive function like below.

```
// C++ program to print permutations of a given string with spaces.
#include <iostream>
#include <cstring>
using namespace std;

/* Function recursively prints the strings having space pattern.
   i and j are indices in 'str[]' and 'buff[]' respectively */
void printPatternUtil(char str[], char buff[], int i, int j, int n)
{
    if (i==n)
    {
        buff[j] = '\0';
        cout << buff << endl;
        return;
    }

    // Either put the character
    buff[j] = str[i];
    printPatternUtil(str, buff, i+1, j+1, n);

    // Or put a space followed by next character
    buff[j] = ' ';
    buff[j+1] = str[i];

    printPatternUtil(str, buff, i+1, j+2, n);
}

// This function creates buf[] to store individual output string and u
// printPatternUtil() to print all permutations.
void printPattern(char *str)
{
    int n = strlen(str);

    // Buffer to hold the string containing spaces
    char buf[2*n]; // 2n-1 characters and 1 string terminator

    // Copy the first character as it is, since it will be always
    // at first position
    buf[0] = str[0];

    printPatternUtil(str, buf, 1, 1, n);
}

// Driver program to test above functions
int main()
{
    char *str = "ABCDE";
    printPattern(str);
    return 0;
}
```

Output:

```
ABCD
ABC D
AB CD
AB C D
A BCD
A BC D
```

```
A B CD
A B C D
```

Time Complexity: Since number of Gaps are n-1, there are total 2^(n-1) patters each having length ranging from n to 2n-1. Thus overall complexity would be O(n*(2^n)).

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

73. Suffix Tree Application 6 – Longest Palindromic Substring

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

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

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

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

Let's look at following examples:

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

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

To handle this scenario we say that LPS in S is same as LCS in S and R **given that**

LCS in R and S must be from same position in S.

If we look at 2nd example again, substring *aba* in R comes from exactly same position in S as substring *aba* in S which is ZERO (0th index) and so this is LPS.

The Position Constraint:

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

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

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

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

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

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

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

- If suffix index of a leaf is less than N, then that suffix belongs to S and same suffix

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

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

(Click to see it clearly)

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

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

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

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

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

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

- We need to know forward and reverse indices on each node.
- For a given forward index S_i on an internal node, we need know if reverse index $\mathbf{R_i = (N – 2) – (S_i + L – 1)}$ also present on same node.
- Keep track of deepest internal node satisfying above condition.

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

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

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

```
// A C++ program to implement Ukkonen's Suffix Tree Construction
// Here we build generalized suffix tree for given string S
// and it's reverse R, then we find
// longest palindromic substring of given string S
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream>
#include <unordered_set>
#define MAX_CHAR 256
using namespace std;

struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];

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

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

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

    //To store indices of children suffixes in given string
    unordered_set<int> *forwardIndices;

    //To store indices of children suffixes in reversed string
```

```
    unordered_set<int> *reverseIndices;
};

typedef struct SuffixTreeNode Node;

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

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

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

// remainingSuffixCount tells how many suffixes yet to
// be added in tree
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1; //Length of input string
int size1 = 0; //Size of 1st string
int reverseIndex; //Index of a suffix in reversed string
unordered_set<int>::iterator forwardIndex;

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

    /*For root node, suffixLink will be set to NULL
    For internal nodes, suffixLink will be set to root
    by default in  current extension and may change in
    next extension*/
    node->suffixLink = root;
    node->start = start;
    node->end = end;

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

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

```
}

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

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

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

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

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

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

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

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

```
// there is an outgoing edge starting with activeEdge
// from activeNode
else
{
    // Get the next node at the end of edge starting
    // with activeEdge
    Node *next = activeNode->children[text[activeEdge]] ;
    if (walkDown(next))//Do walkdown
    {
        //Start from next node (the new activeNode)
        continue;
    }
    /*Extension Rule 3 (current character being processed
      is already on the edge)*/
    if (text[next->start + activeLength] == text[pos])
    {
        //APCFER3
        activeLength++;
        /*STOP all further processing in this phase
        and move on to next phase*/
        break;
    }

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

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

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

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

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

```
        -

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

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

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

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

            //Current node is not a leaf as it has outgoing
            //edges from it.
            leaf = 0;
            setSuffixIndexByDFS(n->children[i], labelHeight +
                                edgeLength(n->children[i]));
            if(n != root)
            {
                //Add chldren's suffix indices in parent
                n->forwardIndices->insert(
                    n->children[i]->forwardIndices->begin(),
                    n->children[i]->forwardIndices->end());
                n->reverseIndices->insert(
                    n->children[i]->reverseIndices->begin(),
                    n->children[i]->reverseIndices->end());
            }
```

```
            }
        }
    }
    if (leaf == 1)
    {
        for(i= n->start; i<= *(n->end); i++)
        {
            if(text[i] == '#')
            {
                n->end = (int*) malloc(sizeof(int));
                *(n->end) = i;
            }
        }
        n->suffixIndex = size - labelHeight;

        if(n->suffixIndex < size1) //Suffix of Given String
            n->forwardIndices->insert(n->suffixIndex);
        else //Suffix of Reversed String
            n->reverseIndices->insert(n->suffixIndex - size1);

        //Uncomment below line to print suffix index
       // printf(" [%d]\n", n->suffixIndex);
    }
}

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

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

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

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

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

                if(*maxHeight < labelHeight
                    && n->forwardIndices->size() > 0 &&
                    n->reverseIndices->size() > 0)
                {
                    for (forwardIndex=n->forwardIndices->begin();
                            forwardIndex!=n->forwardIndices->end();
                            ++forwardIndex)
                    {
                        reverseIndex = (size1 - 2) -
                            (*forwardIndex + labelHeight - 1);
                        //If reverse suffix comes from
                        //SAME position in given string
                        //Keep track of deepest node
                        if(n->reverseIndices->find(reverseIndex) !=
                            n->reverseIndices->end())
                        {
                            *maxHeight = labelHeight;
                            *substringStartIndex = *(n->end) -
                                labelHeight + 1;
                            break;
                        }
                    }
                }
            }
        }
    }
}

void getLongestPalindromicSubstring()
{
    int maxHeight = 0;
    int substringStartIndex = 0;
    doTraversal(root, 0, &maxHeight, &substringStartIndex);

    int k;
    for (k=0; k<maxHeight; k++)
        printf("%c", text[k + substringStartIndex]);
    if(k == 0)
        printf("No palindromic substring");
    else
        printf(", of length: %d",maxHeight);
    printf("\n");
}
```

```
// driver program to test above functions
int main(int argc, char *argv[])
{
    size1 = 9;
    printf("Longest Palindromic Substring in cabbaabb is: ");
    strcpy(text, "cabbaabb#bbaabbac$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 17;
    printf("Longest Palindromic Substring in forgeeksskeegfor is: ");
    strcpy(text, "forgeeksskeegfor#rofgeeksskeegrof$"); buildSuffixTre
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Palindromic Substring in abcde is: ");
    strcpy(text, "abcde#edcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 7;
    printf("Longest Palindromic Substring in abcdae is: ");
    strcpy(text, "abcdae#eadcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Palindromic Substring in abacd is: ");
    strcpy(text, "abacd#dcaba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Palindromic Substring in abcdc is: ");
    strcpy(text, "abcdc#cdcba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 13;
    printf("Longest Palindromic Substring in abacdfgdcaba is: ");
    strcpy(text, "abacdfgdcaba#abacdgfdcaba$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 15;
    printf("Longest Palindromic Substring in xyabacdfgdcaba is: ");
    strcpy(text, "xyabacdfgdcaba#abacdgfdcabayx$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 9;
    printf("Longest Palindromic Substring in xababayz is: ");
```

```
    strcpy(text, "xababayz#zyababax$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    size1 = 6;
    printf("Longest Palindromic Substring in xabax is: ");
    strcpy(text, "xabax#xabax$"); buildSuffixTree();
    getLongestPalindromicSubstring();
    //Free the dynamically allocated memory
    freeSuffixTreeByPostOrder(root);

    return 0;
}
```

Output:

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

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

We have published following more articles on suffix tree applications:

- Suffix Tree Application 1 – Substring Check
- Suffix Tree Application 2 – Searching All Patterns
- Suffix Tree Application 3 – Longest Repeated Substring
- Suffix Tree Application 4 – Build Linear Time Suffix Array
- Suffix Tree Application 5 – Longest Common Substring
- Generalized Suffix Tree 1

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

74. Find if a given string can be represented from a substring by iterating the substring “n” times

Given a string ‘str’, check if it can be constructed by taking a substring of it and appending multiple copies of the substring together.

Examples:

```
Input: str = "abcabcabc"
Output: true
The given string is 3 times repetition of "abc"

Input: str = "abadabad"
Output: true
The given string is 2 times repetition of "abad"

Input: str = "aabaabaabaab"
Output: true
The given string is 4 times repetition of "aab"

Input: str = "abcdabc"
Output: false
```

Source: Google Interview Question

There can be many solutions to this problem. The challenging part is to solve the problem in O(n) time. Below is a O(n) algorithm.

Let the given string be ‘str’ and length of given string be ‘n’.

1) Find length of the longest proper prefix of ‘str’ which is also a suffix. Let the length of the longest proper prefix suffix be ‘len’. This can be computed in O(n) time using pre-processing step of KMP string matching algorithm.

2) If value of ‘n – len’ divides n (or ‘n % (n-len)’ is 0), then return true, else return false.

In case of ‘true’ , the substring ‘str[0..n-len-1]’ is the substring that repeats n%(n-len) times.

Let us take few examples.

Input: str = “ABCDABCD”, n = 8 (Number of characters in ‘str’)
The value of len is 4 (“ABCD” is the longest substring which is both prefix and suffix)
Since (n-len) divides n, the answer is true.

Input: str = “ABCDABC”, n = 8 (Number of characters in ‘str’)
The value of len is 3 (“ABC” is the longest substring which is both prefix and suffix)
Since (n-len) doesn’t divides n, the answer is false.

Input: str = “ABCABCABCABCABC”, n = 15 (Number of characters in ‘str’)
The value of len is 12 (“ABCABCABCABC” is the longest substring which is both prefix

and suffix)
Since (n-len) divides n, the answer is true.

How does this work?
length of longest proper prefix-suffix (or len) is always between 0 to n-1. If len is n-1, then all characters in string are same. For example len is 3 for “AAAA”. If len is n-2 and n is even, then two characters in string repeat n/2 times. For example “ABABABAB”, length of lps is 6. The reason is if the first n-2 characters are same as last n-2 character, the starting from the first pair, every pair of characters is identical to the next pair. The following diagram demonstrates same for substring of length 4.

There are total 20 characers in given string, i.e., n = 20

The value of longest prefix which is also suffix is 16, i.e., len = 16

So the first 16 characters are identical to last 16 characters in order. It indicates that the first 4 characters are identical to next 4 characters (because first 4 characters are beginning characters of prefix and next 4 characters are begining 4 characters of suffix). Similary next 4 characters are identical to next-next four characters because they are 5th to 8th characters of prefix and suffix respectively

Following is C++ implementation of above algorithm.

```
// A C++ program to check if a string is 'n' times
// repetition of one of its substrings
#include<iostream>
#include<cstring>
using namespace std;

// A utility function to fill lps[] or compute prefix funcrion
// used in KMP string matching algorithm. Refer
// http://www.geeksforgeeks.org/archives/11902 for details
void computeLPSArray(char str[], int M, int lps[])
{
    int len = 0; //lenght of the previous longest prefix suffix
    int i;

    lps[0] = 0; //lps[0] is always 0
    i = 1;

    // the loop calculates lps[i] for i = 1 to M-1
    while (i < M)
    {
       if (str[i] == str[len])
       {
           len++;
           lps[i] = len;
           i++;
       }
       else // (pat[i] != pat[len])
       {
```

```
        i
            if (len != 0)
            {
                // This is tricky. Consider the example AAACAAAA and i = 
                len = lps[len-1];

                // Also, note that we do not increment i here
            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

```
// Returns true if str is repetition of one of its substrings
// else return false.
bool isRepeat(char str[])
{
    // Find length of string and create an array to
    // store lps values used in KMP
    int n = strlen(str);
    int lps[n];

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(str, n, lps);

    // Find length of longest suffix which is also
    // prefix of str.
    int len = lps[n-1];

    // If there exist a suffix which is also prefix AND
    // Length of the remaining substring divides total
    // length, then str[0..n-len-1] is the substring that
    // repeats n/(n-len) times (Readers can print substring
    // and value of n/(n-len) for more clarity.
    return (len > 0 && n%(n-len) == 0)? true: false;
}

// Driver program to test above function
int main()
{
   char txt[][100] = {"ABCABC", "ABABAB", "ABCDABCD", "GEEKSFORGEEKS",
                      "GEEKGEEK", "AAAACAAAAC", "ABCDABC"};
   int n = sizeof(txt)/sizeof(txt[0]);
   for (int i=0; i<n; i++)
      isRepeat(txt[i])? cout << "True\n" : cout << "False\n";
   return 0;
}
```

Output:

```
True
True
True
False
True
```

```
True
False
```

Time Complexity: Time complexity of the above solution is O(n) as it uses KMP preprocessing algorithm which is linear time algorithm.

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

75. Find all distinct palindromic sub-strings of a given string

Given a string of lowercase ASCII characters, find all distinct continuous palindromic sub-strings of it.

Examples:

```
Input: str = "abaaa"
Output:  Below are 5 palindrome sub-strings
a
aa
aaa
aba
b


Input: str = "geek"
Output:  Below are 4 palindrome sub-strings
e
ee
g
k
```

Step 1: Finding all palindromes using modified Manacher's algorithm:
Considering each character as a pivot, expand on both sides to find the length of both even and odd length palindromes centered at the pivot character under consideration and store the length in the 2 arrays (odd & even).
Time complexity for this step is O(n^2)

Step 2: Inserting all the found palindromes in a HashMap:
Insert all the palindromes found from the previous step into a HashMap. Also insert all the individual characters from the string into the HashMap (to generate distinct single letter palindromic sub-strings).

Time complexity of this step is O(n^3) assuming that the hash insert search takes O(1) time. Note that there can be at most O(n^2) palindrome sub-strings of a string. In below C++ code ordered hashmap is used where the time complexity of insert and search is O(Logn). In C++, ordered hashmap is implemented using Red Black Tree.

Step 3: Printing the distinct palindromes and number of such distinct palindromes: The last step is to print all values stored in the HashMap (only distinct elements will be hashed due to the property of HashMap). The size of the map gives the number of distinct palindromic continuous sub-strings.

Below is C++ implementation of the above idea.

```
// C++ program to find all distinct palindrome sub-strings
// of a given string
#include <iostream>
#include <map>
using namespace std;

// Function to print all distinct palindrome sub-strings of s
void palindromeSubStrs(string s)
{
    map<string, int> m;
    int n = s.size();

    // table for storing results (2 rows for odd-
    // and even-length palindromes
    int R[2][n+1];

    // Find all sub-string palindromes from the given input
    // string insert 'guards' to iterate easily over s
    s = "@" + s + "#";

    for (int j = 0; j <= 1; j++)
    {
        int rp = 0;   // length of 'palindrome radius'
        R[j][0] = 0;

        int i = 1;
        while (i <= n)
        {
            //  Attempt to expand palindrome centered at i
            while (s[i - rp - 1] == s[i + j + rp])
                rp++;  // Incrementing the length of palindromic
                       // radius as and when we find vaid palindrome

            // Assigning the found palindromic length to odd/even
            // length array
            R[j][i] = rp;
            int k = 1;
            while ((R[j][i - k] != rp - k) && (k < rp))
            {
                R[j][i + k] = min(R[j][i - k],rp - k);
                k++;
            }
            rp = max(rp - k,0);
            i += k;
        }
    }
```

```
    // remove 'guards'
    s = s.substr(1, n);

    // Put all obtained palindromes in a hash map to
    // find only distinct palindromess
    m[string(1, s[0])]=1;
    for (int i = 1; i <= n; i++)
    {
        for (int j = 0; j <= 1; j++)
            for (int rp = R[j][i]; rp > 0; rp--)
               m[s.substr(i - rp - 1, 2 * rp + j)]=1;
        m[string(1, s[i])]=1;
    }

    //printing all distinct palindromes from hash map
   cout << "Below are " << m.size()-1
        << " palindrome sub-strings";
   map<string, int>::iterator ii;
   for (ii = m.begin(); ii!=m.end(); ++ii)
      cout << (*ii).first << endl;
}

// Driver program
int main()
{
    palindromeSubStrs("abaaa");
    return 0;
}
```

Output:

```
 Below are 5 palindrome sub-strings
a
aa
aaa
aba
b
```

This article is contributed by Vignesh Narayanan and Sowmya Sampath. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

76. Find maximum depth of nested parenthesis in a string

We are given a string having parenthesis like below
“(((X)) (((Y))))”
We need to find the maximum depth of balanced parenthesis, like 4 in above example. Since ‘Y’ is surrounded by 4 balanced parenthesis.

If parenthesis are unbalanced then return -1.

More examples:

```
S = "( a(b) (c) (d(e(f)g)h) I (j(k)l)m)";
Output : 4

S = "( p((q)) ((s)t) )";
Output : 3

S = "";
Output : 0

S = "b) (c) ()";
Output : -1

S = "(b) ((c) ()"
Output : -1
```

Source : Walmart Labs Interview Question

Method 1 (Uses Stack)
A simple solution is to use a stack that keeps track of current open brackets.

```
1) Create a stack.
2) Traverse the string, do following for every character
     a) If current character is '(' push it to the stack .
     b) If character is ')', pop an element.
     c) Maintain maximum count during the traversal.
```

Time Complexity : O(n)
Auxiliary Space : O(n)

Method 2 (O(1) auxiliary space)
This can also be done without using stack.

```
1) Take two variables max and current_max, initialize both of them as 0.
2) Traverse the string, do following for every character
    a) If current character is '(', increment current_max and
       update max value if required.
    b) If character is ')'. Check if current_max is positive or
       not (this condition ensure that parenthesis are balanced).
       If positive that means we previously had a '(' character
       so decrement current_max without worry.
       If not positive then the parenthesis are not balanced.
       Thus return -1.
3) If current_max is not 0, then return -1 to ensure that the parenthesis
   are balanced. Else return max
```

Below is the C++ implementation of above algorithm.

```
// A C++ program to find the maximum depth of nested
// parenthesis in a given expression
#include <iostream>
using namespace std;

// function takes a string and returns the
// maximum depth nested parenthesis
int maxDepth(string S)
{
    int current_max = 0; // current count
    int max = 0;    // overall maximum count
    int n = S.length();

    // Traverse the input string
    for (int i = 0; i< n; i++)
    {
        if (S[i] == '(')
        {
            current_max++;

            // update max if required
            if (current_max> max)
                max = current_max;
        }
        else if (S[i] == ')')
        {
            if (current_max>0)
                current_max--;
            else
                return -1;
        }
    }

    // finally check for unbalanced string
    if (current_max != 0)
        return -1;

    return max;
}

// Driver program
int main()
{
    string s = "( ((X)) (((Y))) )";
    cout << maxDepth(s);
    return 0;
}
```

Output:

```
4
```

Time Complexity : O(n)
Auxiliary Space : O(1)

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

above.

77. Function to find Number of customers who could not get a computer

Write a function “runCustomerSimulation” that takes following two inputs
a) An integer ‘n': total number of computers in a cafe and a string:
b) A sequence of uppercase letters ‘seq': Letters in the sequence occur in pairs. The first occurrence indicates the arrival of a customer; the second indicates the departure of that same customer.

A customer will be serviced if there is an unoccupied computer. No letter will occur more than two times.
Customers who leave without using a computer always depart before customers who are currently using the computers. There are at most 20 computers per cafe.

For each set of input the function should output a number telling how many customers, if any walked away without using a computer. Return 0 if all the customers were able to use a computer.

runCustomerSimulation (2, “ABBAJJKZKZ”) should return 0

runCustomerSimulation (3, “GACCBDDBAGEE”) should return 1 as ‘D’ was not able to get any computer

runCustomerSimulation (3, “GACCBGDDBAEE”) should return 0

runCustomerSimulation (1, “ABCBCA”) should return 2 as ‘B’ and ‘C’ were not able to get any computer.

runCustomerSimulation(1, “ABCBCADEED”) should return 3 as ‘B’, ‘C’ and ‘E’ were not able to get any computer.

Source: Fiberlink (maas360) Interview

We strongly recommend to minimize your browser and try this yourself first.

Below are simple steps to find number of customers who could not get any computer.

1) Initialize result as 0.

2) Traverse the given sequence. While traversing, keep track of occupied computers (this can be done by keeping track of characters which have appeared only once and a computer was available when they appeared). At any point, if count of occupied computers is equal to ‘n’, and there is a new customer, increment result by 1.

The important thing is to keep track of existing customers in cafe in a way that can indicate whether the customer has got a computer or not. Note that in sequence “ABCBCADEED”, customer ‘B’ did not get a seat, but still in cafe as a new customer ‘C’ is next in sequence.

Below is C++ implementation of above idea.

```
// C++ program to find number of customers who couldn't get a resource
#include<iostream>
#include<cstring>
using namespace std;

#define MAX_CHAR 26

// n is number of computers in cafe.
// 'seq' is given sequence of customer entry, exit events
int runCustomerSimulation(int n, const char *seq)
{
    // seen[i] = 0, indicates that customer 'i' is not in cafe
    // seen[1] = 1, indicates that customer 'i' is in cafe but
    //              computer is not assigned yet.
    // seen[2] = 2, indicates that customer 'i' is in cafe and
    //              has occupied a computer.
    char seen[MAX_CHAR] = {0};

    // Initialize result which is number of customers who could
    // not get any computer.
    int res = 0;

    int occupied = 0;  // To keep track of occupied computers

    // Travers the input sequence
    for (int i=0; seq[i]; i++)
    {
        // Find index of current character in seen[0...25]
        int ind = seq[i] - 'A';

        // If First occurrence of 'seq[i]'
        if (seen[ind] == 0)
        {
            // set the current character as seen
            seen[ind] = 1;

            // If number of occupied computers is less than
            // n, then assign a computer to new customer
            if (occupied < n)
            {
                occupied++;

                // Set the current character as occupying a computer
                seen[ind] = 2;
            }

            // Else this customer cannot get a computer,
            // increment result
            else
                res++;
        }

        // If this is second occurrence of 'seq[i]'
        else
```

```
        else
        {
            // Decrement occupied only if this customer
            // was using a computer
            if (seen[ind] == 2)
                occupied--;
            seen[ind] = 0;
        }
    }
    return res;
}

// Driver program
int main()
{
    cout << runCustomerSimulation(2, "ABBAJJKZKZ") << endl;
    cout << runCustomerSimulation(3, "GACCBDDBAGEE") << endl;
    cout << runCustomerSimulation(3, "GACCBGDDBAEE") << endl;
    cout << runCustomerSimulation(1, "ABCBCA") << endl;
    cout << runCustomerSimulation(1, "ABCBCADEED") << endl;
    return 0;
}
```

Output:

```
0
1
0
2
3
```

Time complexity of above solution is O(n) and extra space required is O(CHAR_MAX) where CHAR_MAX is total number of possible characters in given sequence.

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

78. Find the longest substring with k unique characters in a given string

Given a string you need to print longest possible substring that has exactly M unique characters. If there are more than one substring of longest possible length, then print any one of them.

Examples:

```
"aabbcc", k = 1
Max substring can be any one from {"aa" , "bb" , "cc"}.
```

```
"aabbcc", k = 2
Max substring can be any one from {"aabb" , "bbcc"}.

"aabbcc", k = 3
There are substrings with exactly 3 unique characters
{"aabbcc" , "abbcc" , "aabbc" , "abbc" }
Max is "aabbcc" with length 6.

"aaabbb", k = 3
There are only two unique characters, thus show error message.
```

Source: Google Interview Question

Method 1 (Brute Force)
If the length of string is n, then there can be n*(n+1)/2 possible substrings. A simple way is to generate all the substring and check each one whether it has exactly k unique characters or not. If we apply this brute force, it would take $O(n^2)$ to generate all substrings and O(n) to do a check on each one. Thus overall it would go $O(n^3)$.

We can further improve this solution by creating a hash table and while generating the substrings, check the number of unique characters using that hash table. Thus it would improve up to $O(n^2)$.

Method 2 (Linear Time)
The problem can be solved in O(n). Idea is to maintain a window and add elements to the window till it contains less or equal k, update our result if required while doing so. If unique elements exceeds than required in window, start removing the elements from left side.

C++ implementation of above. The implementation assumes that the input string alphabet contains only 26 characters (from 'a' to 'z'). The code can be easily extended to 256 characters.

```
// C++ program to find the longest substring with k unique
// characters in a given string
#include <iostream>
#include <cstring>
#define MAX_CHARS 26
using namespace std;

// This function calculates number of unique characters
// using a associative array count[]. Returns true if
// no. of characters are less than required else returns
// false.
bool isValid(int count[], int k)
{
    int val = 0;
    for (int i=0; i<MAX_CHARS; i++)
        if (count[i] > 0)
            val++;
```

```
    // Return true if k is greater than or equal to val
    return (k >= val);
}

// Finds the maximum substring with exactly k unique chars
void kUniques(string s, int k)
{
    int u = 0; // number of unique characters
    int n = s.length();

    // Associative array to store the count of characters
    int count[MAX_CHARS];
    memset(count, 0, sizeof(count));

    // Traverse the string, Fills the associative array
    // count[] and count number of unique characters
    for (int i=0; i<n; i++)
    {
        if (count[s[i]-'a']==0)
            u++;
        count[s[i]-'a']++;
    }

    // If there are not enough unique characters, show
    // an error message.
    if (u < k)
    {
        cout << "Not enough unique characters";
        return;
    }

    // Otherwise take a window with first element in it.
    // start and end variables.
    int curr_start = 0, curr_end = 0;

    // Also initialize values for result longest window
    int max_window_size = 1, max_window_start = 0;

    // Initialize associative array count[] with zero
    memset(count, 0, sizeof(count));

    count[s[0]-'a']++;  // put the first character

    // Start from the second character and add
    // characters in window according to above
    // explanation
    for (int i=1; i<n; i++)
    {
        // Add the character 's[i]' to current window
        count[s[i]-'a']++;
        curr_end++;

        // If there are more than k unique characters in
        // current window, remove from left side
        while (!isValid(count, k))
        {
            count[s[curr_start]-'a']--;
            curr_start++;
        }

        // Update the max window size if required
        if (curr_end-curr_start+1 > max_window_size)
        {
```

```
            max_window_size = curr_end-curr_start+1;
            max_window_start = curr_start;
        }
    }

    cout << "Max sustring is : "
         << s.substr(max_window_start, max_window_size)
         << " with length " << max_window_size << endl;
}

// Driver function
int main()
{
    string s = "aabacbebebe";
    int k = 3;
    kUniques(s, k);
    return 0;
}
```

Output:

```
Max sustring is : cbebebe with length 7
```

Time Complexity: Considering function “isValid()” takes constant time, time complexity of above solution is O(n).

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

79. Check if a given sequence of moves for a robot is circular or not

Given a sequence of moves for a robot, check if the sequence is circular or not. A sequence of moves is circular if first and last positions of robot are same. A move can be on of the following.

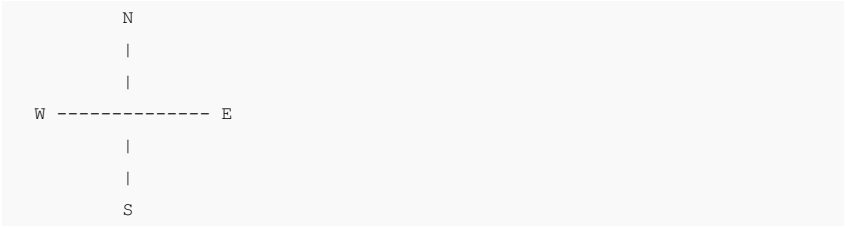
```
G - Go one unit
L - Turn left
R - Turn right
```

Examples:

```
Input: path[] = "GLGLGLG"
Output: Given sequence of moves is circular

Input: path[] = "GLLG"
Output: Given sequence of moves is circular
```

We strongly recommend to minimize your browser and try this yourself first.

The idea is to consider the starting position as (0, 0) and direction as East (We can pick any values for these). If after the given sequence of moves, we come back to (0, 0), then given sequence is circular, otherwise not.



The move ‘G’ changes either x or y according to following rules.
a) If current direction is North, then ‘G’ increments y and doesn’t change x.
b) If current direction is East, then ‘G’ increments x and doesn’t change y.
c) If current direction is South, then ‘G’ decrements y and doesn’t change x.
d) If current direction is West, then ‘G’ decrements x and doesn’t change y.

The moves ‘L’ and ‘R’, do not change x and y coordinates, they only change direction according to following rule.
a) If current direction is North, then ‘L’ changes direction to West and ‘R’ changes to East
b) If current direction is East, then ‘L’ changes direction to North and ‘R’ changes to South
c) If current direction is South, then ‘L’ changes direction to East and ‘R’ changes to West
d) If current direction is West, then ‘L’ changes direction to South and ‘R’ changes to North.

The following is C++ implementation of above idea.

```
// A c++ program to check if the given path for a robot is circular or
#include<iostream>
using namespace std;

// Macros for East, North, South and West
#define N 0
#define E 1
#define S 2
#define W 3

// This function returns true if the given path is circular, else fals
bool isCircular(char path[])
{
  // Initialize starting point for robot as (0, 0) and starting
  // direction as N North
  int x = 0, y = 0;
  int dir = N;

  // Travers the path given for robot
  for (int i=0; path[i]; i++)
  {
      // Find current move
      char move = path[i];

      // If move is left or right, then change direction
      if (move == 'R')
        dir = (dir + 1)%4;
      else if (move == 'L')
        dir = (4 + dir - 1)%4;

      // If move is Go, then change  x or y according to
      // current direction
      else // if (move == 'G')
      {
         if (dir == N)
            y++;
         else if (dir == E)
            x++;
         else if (dir == S)
            y--;
         else // dir == W
            x--;
      }
  }

   // If robot comes back to (0, 0), then path is cyclic
  return (x == 0 && y == 0);
}

// Driver program
int main()
{
    char path[] = "GLGLGLG";
    if (isCircular(path))
      cout << "Given sequence of moves is circular";
    else
      cout << "Given sequence of moves is NOT circular";
}
```

Output:

```
Given sequence of moves is circular
```

Time Complexity: O(n) where n is number of moves in given sequence.

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

80. Recursively print all sentences that can be formed from list of word lists

Given a list of word lists How to print all sentences possible taking one word from a list at a time via recursion?

Example:

```
Input: {{"you", "we"},
        {"have", "are"},
        {"sleep", "eat", "drink"}}

Output:
  you have sleep
  you have eat
  you have drink
  you are sleep
  you are eat
  you are drink
  we have sleep
  we have eat
  we have drink
  we are sleep
  we are eat
  we are drink
```

We strongly recommend to minimize your browser and try this yourself first.
The idea is based on simple depth first traversal. We start from every word of first list as first word of an output sentence, then recur for the remaining lists.

Below is C++ implementation of above idea. In the below implementation, input list of list is considered as a 2D array. If we take a closer look, we can observe that the code is very close to DFS of graph.

```
// C++ program to print all possible sentences from a list of word lis
#include <iostream>
#include <string>

#define R 3
#define C 3
using namespace std;

// A recursive function to print all possible sentences that can be fo
// from a list of word list
void printUtil(string arr[R][C], int m, int n, string output[R])
{
    // Add current word to output array
    output[m] = arr[m][n];

    // If this is last word of current output sentence, then print
    // the output sentence
    if (m==R-1)
    {
        for (int i=0; i<R; i++)
           cout << output[i] << " ";
        cout << endl;
        return;
    }

    // Recur for next row
    for (int i=0; i<C; i++)
       if (arr[m+1][i] != "")
          printUtil(arr, m+1, i, output);
}

// A wrapper over printUtil()
void print(string arr[R][C])
{
   // Create an array to store sentence
   string output[R];

   // Consider all words for first row as starting points and
   // print all sentences
   for (int i=0; i<C; i++)
     if (arr[0][i] != "")
        printUtil(arr, 0, i, output);
}

// Driver program to test above functions
int main()
{
   string arr[R][C]  = {{"you", "we"},
                        {"have", "are"},
                        {"sleep", "eat", "drink"}};

   print(arr);

   return 0;
}
```

Output:

```
you have sleep
you have eat
```

```
you have drink
you are sleep
you are eat
you are drink
we have sleep
we have eat
we have drink
we are sleep
we are eat
we are drink
```

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