STRING

- Sequence of characters
- Small Set
- contiguous integer value 'a' to 'z' and 'A' to 'Z' in both ASCII and UTF-16

C/C++	Java
char :- ASCII	UTF-16
8 Bit	16 Bit

- Also supports unicode

```
char x = 'a'
cout << (int)x;   //97
```

Print the frequencies of character in sorted order

```
string str = "geeksforgeeks";
int count[26] = {0};
for(int i=0; i< str.length(); i++)
    count[str[i] - 'a']++;
for(int i=0; i<26; i++){
    if(count[i] > 0){
        cout <<(char)(i+'a')<<" ";
        cout << count[i];
    }
}
```

O/P

e	4
f	1
g	2
k	2
o	1
r	1
s	2

Strings in C++ (char arrays)

```
char str[] = "gfg"
```

compiles →

g	f	g	\0

if we give definite size

```
str[6] = "gfg"
```

g	f	g	\0	\0	\0

func

- strlen → gives length of string

- **strcmp (a,b)**

  compare strings lexicographically (dictionary types) res = strcmp(a,b)

  if a < b then res < 0
  if a = b then res = 0
  if a > b then res > 0

- **strcpy (a,b)**

  To copy another string we have to use strcpy

  strcpy(str, "gfg");

I can't do this

```
char str[5];
str = "gfg";
cout << str;
```

str is an address and we can't assign anything directly to an address.

C++ String

- Richer library
- supports ~~++~~ operations like +, <, >, ==, <=, >= are possible
- can assign a string later.
- Do not have to worry about size
- C++ strings are ~~classes~~ objects of classes.

* str.length() → length of string
* str.substring(start, length) →

"geeksforgeeks" ← (1,3) → eek

"abcdef" ← (2,3) (cde)

beginning index

length of string you want

str.find("eek") → substring → returns 1 if present

- Returns index of first occurance.
- Otherwise string::npos.

```
string str = "geeksforgeeks"
str.find("eek");  → 1
str.find("ksf");  → 3.
```

strcmp

In C++ we have ==, >, <

Reading String from console

```
string str;
cin >> str
cout << str;
```

case 1

user entered → sandeep
output → sandeep

case 2

user entered - sandeep jain
output - sandeep

Reason: whenever user entered a ' ' (space) at the ~~cor~~ cin operator stops reading the character.

For this purpose we use getline

```
string str;
getline(cin, str)
```

↓

It will stop reading the character after you presses enter

It can also accept an optional character.

getline(cin, str, 'g')

↑ stop when you see g

Anagram of Each other

- checking if two strings are permutation of each other
- Every character in the first string should be present in the second and the frequency must also be same.

S_1 = "listen", S_2 = "silent".

O/P → Yes

S_1 = "aab", S_2 = "bab"

O/P ⇒ NO.

Naive Approach O(nlogn)

Sort both the string and then compare them.

```
sort (s1.begin(), s1.end())
sort (s2.begin(), s2.end())
return (s1 == s2);
```

↓ we can immediately return false if length is not same

Efficient Solution

We create a count array and then used characters of string as indexes.

on each occurance of character in 1st string we increament the count of that characters and on each occurance of ~~coun~~ characters in second string we decreament the count.

```
bool anagram(string s1, string s2) {
    // check for length
    int count[char] = 25 {0};
    for (int i=0; i<length; i++) {
      count[s1[i]]++;
      count[s2[i]]--;
    }.
    if for (int i=0; i<n; i++) if (count[i] != 0) return false
```

Leftmost Repeating character

I/P: str = "geeksforgeeks"

O/P → 0 'g' is the first repeating char.

I/P → str = "abbcc"

O/P = 1 'b' is the first repeating

I/P → str = "abcd"

O/P = -1

I/P → str = "abba"

O/P = 0

Naive Approach

Run two loops and check if particular character appears in the string after it.

$O(n^2)$

Effective approach

use characters as indexes and then run another loop to check count[str[i]] > 1 → return i;

```
const int CHAR = 256;
int getleftmostRep(string str) {
    int count[CHAR] = {0};
    for (int i=0; i<str.length(); i++) {
        count[str[i]]++;
    }.
    for (int i=0; i<str.length(); i++) {
    if (count[str[i]] > 1)
            return i;
    }
    return -1;
}.
```

Effective Approach (To solve in one traversal)

We use an array which is initialised as -1 and whenever we see a character we store its first index into that array.

If we see a character whose corresponding value in f-array is not -1, we get to know that this is repeating character. To get the leftmost we take the minimum of all repeating

```
int leftMost (string str){
    int f-index [char];
    fill ( fIndex , fIndex + char , -1 );
    int res = INT_MAX;
    for (int i=0 ; i< str.length() ; i++){
        int fi = fIndex [str[i]];
        if (fi == -1)
            fIndex [str[i]] = i ;
        else
            res = min (res, fi);
    }.
    return (res == INT_MAX) ? -1 : res;
}
```

Effective Approach-2

We traverse from right side

```
int leftMost (string str){
    bool visited [CHAR];
    fill visited , visited + char, false);
    int res = -1;
    for (int i= str.length()-1 ; i>=0 ; i++){
        if (visited [str[i]])
            res = i;
        else
            visited [str[i]] = true; }
return res; }
```

Index of Leftmost Non-repeating char

I/P → "geeksforgeeks" ;

O/P → 5

I/P → aabb

O/P → -1

1st solution

Make an array count and then store frequencies of every character

Now just traverse ~~on~~ the array and check if its count is 1 (return i).

2nd solution

we will make an array ~~named~~ status

if status [str[i]] = -1 (unvisited)
status [str[i]] = -2 (repeated)
status [str[i]] = >=0 (element is present only once)

```
int nonRep (string str){
    int count [CHAR];
    fill (count, count + CHAR, -1)
    for (int i=0 ; i < str.length() ; i++) {
        if (count [str[i]] == -1)
            count [str[i]] = i ;
        else
            count [str[i]] = -2;
    }
    for (int i=0 ; i < CHAR ; i++) {
        ~~res = min~~ ;
        if (count[i] > 0)
            res = min (res, count[i]);
    }.
```

* Requires one traversal of string
* and one traversal of count array (256).

Reverse Words in a String

I/P: "Welcome to gfg"
O/P: "gfg to Welcome"

Naive Solution

Push the words inside the stack and then pop the words ~~which~~ while appending space

gfg + " " + to + " " + welcome.

Auxilary space → O(n).

Effective Solution

We reverse individual words and then we reverse the whole string.

I/P → "welcome to gfg"
↓
"emoclew ot gfg"
↓
gfg to welcome ← required string

```
void reverseword (char str[], int n){
    int start = 0
    for (int end = 0 ; end < n ; end++){
        if (str[end] == ' '){
            reverse (start, end-1, str);
            start = end+1;
        }
    }
    reverse (start, n-1, str);
    reverse (0, n-1, str);
}
```

fore reversing last word

```
void reverse (char str[], int low, int high){
    while (low < high){
        swap (str[low], str[high]);
        low++;
        high--;
    }
}
```

- If we use character array and get its size using size of operator, we will get one extra as compiler adds '\n' at end, so we can call the function get (str, n-1);

Overview of Pattern Searching

I/P - "Geeksforgeeks" pattern → "eks"
O/P = 2, 10

I/P → "AAAAAA" AAA
O/P → 0, 1, 2

I/P → ABCDEFABD patt → "ABF"
O/P → not present

Pattern searching Algorithm

m → pattern length
n → text length
1 ≤ m ≤ n

Naive :- O((n-m+1)*m). ← no preprocessing

when all character are distinct → O(n).

Robin Karp :- O((n-m+1)*m). ← Preprocesses pattern

KMP :- O(n)

Suffix Tree :- O(m) ← preprocesses text

used when we have to find all occurrence of pattern & per thousands of patt

Naive Pattern Searching

We slide the ~~text~~ ~~text~~ pattern over the text and check if it matches with the substring

```
void patsearching(string txt, string pat){
    int n = text.length();
    int m = pat.length();
    string curr = text.substring(0,m);
    for(int i=0; i<n-m; i++){

    int n = text.length();
    int m = pat.length();
    string curr = text.substring(0,m);
    for(int i = m; i < n; i++){
        curr
```

```
int patsearching(string txt, string pat){
    int n = txt.length();
    int m = pat.length();
    for(int i=0; i<= n-m; i++){
        int j;
        for(int j=0; j<m; j++)
            if(pat[j] != text[i+j]
                break;
            if(j==m)
                cout << i << " ";
    }
}
```

Time comp → $O((n-m+1)*m)$

Improve Naive Algorithm → when pattern is distinct

txt → ABCABCD

pat → ABCD

- similar to naive, we match the pat with the text but if in naive we move the pattern by 1 every time.

Here as the pattern is distinct we can move the pattern by j times if j is the no of char. matched.

```
void printchar(string txt, string pat){
    int n = ~~patsea~~ txt.length();
    int m = pat.length();
    for(int i=0; i<= n-m; ~~i++~~){
        int j=0;
        for(j=0; j<m && j++)
            if(pat[j] != txt[i+j])
                break;
        if(j==m)
            print(i);
        if(j==0)
            i++;          } naive
        else
            i = i+j;      } shift by j
    }
}
```

Rabin Karp Algorithm

We don't directly compare every window with pattern. We compute the hash of pattern and windows of text. If hash value match we compare the character.

hash-function → sum of ascii value of characters

let the ascii values of a → 1, b → 2, c → 3, d → 4, e → 5

txt = "abdabcbabc"

pat = "abc"

P → hash value of pattern (1+2+3) = 6 (

abdabcbabc

abd → 1+2+4 = 7

bda → 7
dab → 7
abc → 6 (Match)
bcb → 7
cba → 6 (Match) → spurious hit. Hash value matches but char. doesn't.
bab → 5
abc → 6 (Match)

hash match & pattern found.

- If hash value of window matches, we match individual characters.

We can compute next hash by previous hash — this is also known as rolling hash

$$t_{i+1} = t_i + txt[i+m] - txt[i];$$

- In simple hash, we have problem of spurious hits.

- To reduce the chances of matching hash when window is not same with pattern we use another more effective hash function

* It uses the concept of weighted sum.

let the string be $a_0 a_1 a_2 a_3 \cdots a_{t} a_{t+1} \cdots a_{n-2} a_{n-1}$.

To the hash value for this string would be

we have to calculate hash for a particular window only. → let it be $a_{t-2} a_{t-1} a_t$ → $a_{t-2} \times d^2 + a_{t-1} \times d^1 + a_t \times d^0$.

supposing that pattern is of size = 3.

d = no of characters in text

* we can also use the concept of rolling hash here

$$t_{i+1} = d(t_i - txt \times d^{m-1}) + txt[i+m].$$

understanding this formulae. → let str → "abc" & txt → "babcde" → To move the window 1 step ahead.

→ we have to subtract $a \times d^{m-1}$.

→ and we'll add $d \times d^0 = d$.

→ also we'll multiply d* (subtracted hash).

txt → $a_{n-1} a_{n-2} \cdots a_t a_{t-1} \cdots a_2 a_1 a_0$

let curve window = $a_t a_{t-1} a_{t-2}$

hash = $a_t \times d^2 + a_{t-1} \times d^1 + a_{t-2} \times d^0$

next hash = $a_{t-1} \times d^2 + a_{t-2} \times d^1 + a_{t-3} \times d^0$

Removed, *d, being added

- we will store the hash value under modulo q
  - q → prime number
    - we try to choose bigger values because
      let $q = 13$
      ↓
      hash range → (0, 12)
  - ↓ because hash value may be large & can produce integer overflow

- we precompute pattern and first window hash
  - we can calculate hash by a simple formula

```
p = 0
for (i=0 → m)
    p = p*d + pat[i]        (Horner's Rule)
```
Step-2

at i=0	p → pat[0]
at i=1	p → pat[0]*d + pat[1]
at i=2	p → pat[0]*d^2 + pat[1]*d + pat[2]
	⋮

- we also precomput $(d^{M-1}) \% q$ → this value is used in calculating next window hash.

Step-1

```
void Robinkarp (pat, txt, m, n){
    int h = 1;
    for (int i = 1; i <= m-1; i++)
        h = (h*d) % q;
    int p=0, t=0;
    for (int i=0; i<m; i++){
        p = (p*d + pat[i]) % q
        t = (t*d + pat[i]) % q.
    }
    for (int i=0; i<= N-M; i++)
```

$(d^{M-1}) \% q$

☆☆

Algorithm

```
void RBSearch(pat, text, m, n){
    int h = 1;
    for (int i = 1; i <= m-1; i++)
        h = (h*d) % q;
```
$(d^{m-1}) \% q$

```
    int p = 0, t = 0;
    for (int i = 0; i < m; i++){
        p = (p*d + pat[i]) % q
        t = (t*d + txt[i]) % q
    }
```
using Horner's rule

```
    for (int i = 0; i <= N-m; i++){
        if (p == t){
            bool flag = true;
            for (int j=0; j<m; j++)
                if (txt[i+j] != pat[j]){
                    flag = false;
                    break;
                }
            if (flag == true){
                print(i);
            }
        }
        if (i < n-m){
            t = ((d*(t - txt[i]*h) + txt[i+m])
            if (t < 0) t = t+q;
        }
    }
```
check for spurious hit

calculating next hash

Constructing longest possible prefix suffix array

I/P → "ababac"

O/P → {0, 0, 1, 2, 0}

I/P → "aaaa"

O/P → {0, 1, 2, 3}

prefix of "abcd" (string is not inc. in prefix)

"", "a", "ab", "abc"

suffix of "abcd"

"", "d", "cd", "bcd", "abcd"

for string ababc

(a) at i=0 prefix → "" suffix → "", "a" → 0

(ab) at i = 1 prefix → "", "a" suffix → "b", "ba", "" → 0

(aba) at i = 2 ~~prefix~~ s → "", "a", "ba", "aba" p → "", "a", "ab", → 1

(abab) at i = 3 p → "a", "ab", "aba", "" → 2 s → "", "b", "ab", "bab", "abab"

ababc at i = 4 p → "a", "ab", "aba", "abab", "" s → "", "c", "bc", "abc", "babc", "ababc". → 0

O/P → {0, 0, 1, 2, 0}.

- for all character same (length − n) → {0, 1, 2 --- n-1}.
- for all character distinct → {0, 0, 0 --- 0, 0}

Str → "abacabad"

O/P → {0, 0, 1, 0, 1, 2, 3, 0}

Str → "abiabib"

O/P → {0, 0, 0, 1, 2, 3}

Naive Approach O(n³)

We check for every index whether their prefix and suffix match.

For example → let i = 6 ababacab n = 8

we only consider = ababac

we start by last index

ie, i = 5 — we know the max. possible length of prefix is i − 1

ababac

we will compare

arr[i] with arr[n − len + i]

ie for i = 5 a[0], a[1] (len = 1)
a[1], a[2] (both are same)
a[2], a[3]
⋮

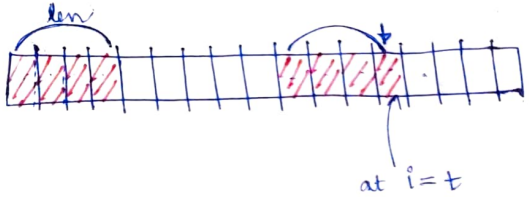
for i = 3 a[0], a[3]
a[1], a[4]
a[2], a[5]
⋮

```
int longProperPrefix (str, n) {
    for (int len = n-1; len > 0 ; len--) {
        bool flag = true;
        for (int i=0; i< len ; i++) {
            if ( str[i] != str[n- len+i]){
                flag = false;
                break;
            }
        }
        if (flag == true,)
            return len;
    }
    return 0; }
```

```
void fillLPS (str, lps[]) {
    for (int i=0; i<str.length(); i++){
        lps[i] = longProperPrefix (str, i+1);
    }
}.
```

$O(n^3)$

O(N) solution



not at $i = t+1$

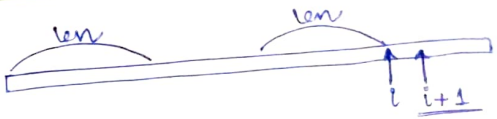
if str[len] = str[i] then o/p → len+1

we compare str[len] and str[i]

if str[len] == str[i] then LPS[i] = ~~lps[i]~~ lps[i-1]+1.

but if they are not same

O(N) solution



Let lps[i-1] = len

then len chars must be matching.

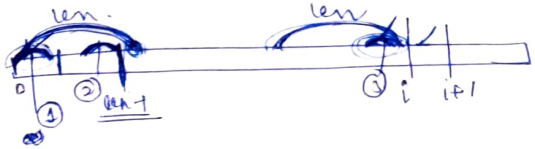
if str[len] = str[i] then lps[i] = len + 1

if they do not match

(i) len = 0

then we don't have any common substring lps[i] = 0.

(ii) else



we know that

characters from

0 to l-1 =

char from [i-len) to i

we will recursive call lps[len-1]

since ① = ②

and ② = ③

so ① = ③ if str[lps[len-1]] = str[i] then lps[i] = lps[len-1].

Basic idea

① → if str[i] and str[len] match lps[i] = len+1 . len++

② → if they do not match

(a) len == 0

lps[i] = 0

(b) else

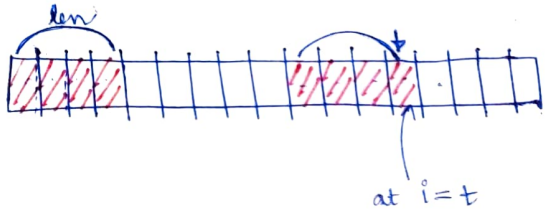
len = lps[len-1]

we now compare str[len] and str[i] if they match → lps[i] = len else recursively call.

```
void fillLPS (str, lps[]) {
    for (int i=0; i<str.length(); i++){
        lps[i] = longProperPrefix (str, i+1);
    }
}.
```

$O(n^3)$

O(N) solution



not at $i = t+1$

if str[len] = str[i] then O/p → len+1

~~but str if str[len] != str[i] then O/p → 0~~

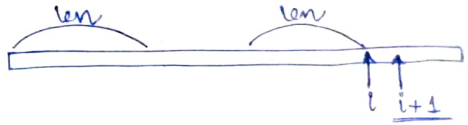
~~The context is that either the length will increase or it will drop to zero~~

we compare str[len] and str[i]

if str[len] == str[i] then LPS[i] = ~~lps[i]~~ lps[i-1]+1.

but if they are not same

O(N) solution



Let lps[i-1] = len

then len chars must be matching.

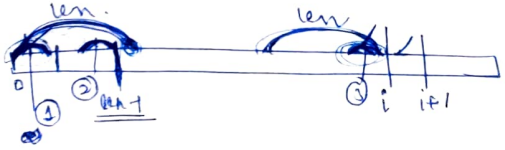
if str[len] = str[i] then lps[i] = len + 1

if they do not match

(i) len = 0

then we don't have any common substring lps[i] = 0

(ii) else



we know that characters from 0 to l-1 = char from [i-len] to i

we will recursive call lps[len-1]

since ① = ②

and ② = ③

so ① = ③ if str[lps[len-1]] = str[i] then lps[i] = lps[len-1]

Basic idea

① → if str[i] and str[len] match lps[i] = len+1 · len++

② → if they do not match

(a) len == 0

lps[i] = 0

(b) else

len = lps[len-1]

we now compare str[len] and str[i] if they match → lps[i] = len else recursively call.

```
void fillLPS(string str, lps[]){
    int n = str.length(), len = 0;
    lps[0] = 0;
    int i = 1;
    while(i < n){
      if(str[i] == str[len]){
            len++;
            lps[i] = len;
            i++;
      }
      else{
        if(len == 0){
              lps[i] = 0;
              i++;
        }
        else{
          len = lps[len-1];
        }
      }
    }
}
```

→ we keep to reducing length recursively until we fall into the above two cases.

* when characters are distinct → O(n)
* when we have all chars same → O(2n) → O(n)

KMP String Matching

① KMP is introduced to ~~introduce~~ reduce the runtime of algorithm when there are some matches to the pattern.

$a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8$ $\quad a_1 = m_1$, $a_2 = m_2$, $a_3 \neq m_3$.

$m_1 m_2 m_3$

$t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12} t_{13}$

$P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8$

We know that $t_4 = P_1$ / $t_5 = P_2$ / $t_6 = P_3$ but $t_7 \neq P_4$ as the lps[2] i.e P_3 is 2 (suppose).

Hence $P_1 P_2 = P_2 P_3$.

also $t_5 t_6 = P_2 P_3$

then $t_5 t_8 = P_1 P_2$

instead of shifting string by 1 (everytime), we can make a shift of $j - lps[j]$.

* we know that the longest proper prefix that could exist is lps[j].

$t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12} t_{13}$ $\quad t_3 \cdots t_8 = P_1 \cdots P_6$

$P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_{10}$ $\quad t_9 \neq P_7$

Instead of shifting by 1 character everytime we just check lps[j] because if there is a chance that next character will also match with P_1 and so on then lps[j] → [j] i.e it gives the longest common substring of the lps of $P_6 \rightarrow 4$ then

chars $P_3 \cdots P_6 = P_1 \cdots P_4$

and $P_3 \cdots P_6 = t_5 \cdots t_8$

then $P_1 \cdots P_4 = t_5 \cdots t_8$

so the next char. with which P_1 will match is t_5

Naive and KMP execute similar when the all the pattern in the characters is distinct

```
void KMP (pat, txt) {
    int n = txt.length();
    int m = pat.length();
    int lps[m];
    fillLPS(pat, lps);
    int i=0, j=0
    while (i < N) {
        if (pat[j] == txt[i]) { i++; j++; }
        else
        if (j == M) {cout<<(i-j); j = lps[j-1]}
        else if (i< N && pat[j] != txt[i]) {
            if (j==0) {i++;}
            else { j = lps[j-1]; }
        }
    }
}
```

txt = ababcabababaad
pat = ababa

lps = {0,0,1,2,3}

i=0 → j++ → 1

i = 0,1,2,3,4
j = 0,1,2,3

at i = 4 and j = 4 they don't match

j = lps[3] = 2 → j=2

we know that the previous will match so we simply move the pattern there.

a b a b c a b a b a a d
a b a b a

Now i = 4 and j = 2 don't match

j = lps[1] = 0.

j = 0

ababcababababaad
ababa

Now at i = 4 and j = 0 they don't match, we come to 2nd cond^n.

i++.

ababcababababaad
ababa

now at i = 6 and j = 0 they match

7,8,9,10 1,2,3,4

and j = 4 == M.

hence cout << (10-4) = 6

and j = lps[3] = 2

i = 10 and j = 3

ababcababababaad
ababa

they mismatch

j = lps[2] = 1

ababcababababaad
ababa

they mismatch

j = lps[0] = 0

ababcababababaad
ababa

i = 10 → 11 (matched)

j = 0 → 1

but mismatch

j = lps[0] = 0

ababcababababaad
ababa

mismatch

i++ → 12

(loop cond^n false)

Out of loop

Time complexity

maximum the i value in the string can be shifted by N times
and the pattern will slide by the text n time
so the complexity will be upper bounded by $O(2n)$

$O(n)$

Check if Strings are Rotations

I/P → ABCD & CDAB

Yes

ABCD → BCDA → CDAB

I/P → ABAAA & BAAAA

Yes

~~BAAAA~~

I/P → ABAB & ABBA

No.

Naive solution

Rotate string one by one and check if it matches with S_2. This is $O(n^2)$ solution

$O(n)$ → To rotate ~~the~~ clockwise by 1.
$O(n)$ → To compare two strings

Efficient solution

① we can pattern search in a circular manner
② we can concatinate S_1 with itself and then search the pattern in itself

```
bool areRotations (string s1, string s2) {
    if (s1.length() != s2.length()) return false;
    return ((s1+s2).find(s2) != string::npos);
}
```

Anagram Search

We have to check if pattern is present or any of its permutation is present in the text.

I/P: txt → "geeksforgeeks"
pat → "frog"
O/P → Yes

I/P: text → "geeksforgeeks"
pat → "rsek" → These characters were present but they are not contigous
O/P → NO

Naive solution

- If Run a naive pattern searching algorithm and instead of searching only pattern we search for anagram.

```
bool isPresent (string str, string pat) {
    int n = str.length();
    int m = pat.length();
    for (int i=0; i<= M-N; i++) {
        if (areAnagram (str, pat, i))
            return true
    }
    return false;
}
bool areAnagram (string txt, string pat, int i) {
    int count[CHAR] = 0;
    for (int j=0 ; j < txt.length(); ; j++) {
        count [pat[j]]++;
        count [txt[i+j]]--;
    }
    for (int i=0; i < CHAR; i++) {
        if (count[i] != 0) return false; } return true;
```

Effective solution

just a modification of naive approach, we make two count arrays ① → pattern
② → curr_window

initially we compute count arrays for 1st window and then pattern:

for ith window we just have to ~~CT[txt[i]]++; CT[txt[i-pat~~

```
CT[txt[i]]++;
CT[txt[i-m]]--;
```

O(n * CHAR)

if CT and CP match then return true → count arr for pattern

```
bool isPresent(string stu, string pat){
    int CP[CHAR] = {0};
    int CT[CHAR] = {0};
    for(int i=0; i< pat.length(); i++){
        CP[txt[i]]++;
        CP[pat[i]]++;
    }
    for(int i= pat.length(); i < txt.length(); i++){
        if(CT == CP){
            return true;
        }
        CT[txt[i]]++;
        CT[txt[i-pat.length()]]--;
    }
    return false;
}
```

Lexicographic Rank of String

I/P : BAC
O/P : 3.

ABC
ACB
→ BAC
BCA
;

I/P → STRING
O/P → 598

Effective method

- count characters which are smaller & which are on right

R	S R	STI	STRIG — STRING
I	S I	STN	
N	S N	STG	
G	S G	STRG	

4! x 4 + 3! x 4 + 2!

5! x 4 + 4! x 4 + 3! x 3 + 1 + 2!

= 120 x 4 + 24 x 4 + 6 x 3 + 1

= 480 + 96 + 18 + 1 + 2

= 597 → 597 + 1

DCBA → 24

A	DA	DCA	DCBA ✓
B	DB		
C			

3! x 3 + 2! x 2 + 1 x 1

= 18 + 4 + 1 ⇒ 23 → 23 + 1