Strings Introduction

⇒ sequence of characters

⇒ small set

⇒ contiguous integer values for 'a' to 'z' and 'A' to 'Z' in both ASCII and UTF-16.

'a' → 97

'A' → 65

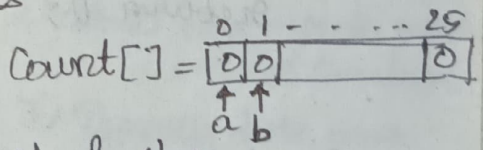
total ⇒ 128 characters

char:
- C / C++ → ASCII → 8 bit → Also supports wchar_t
- Java → UTF-16 → 16 bit → Also supports byte.

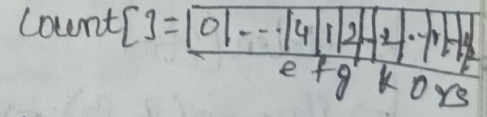
```
⇒ char a = 'a';
  cout << (int) a;
```

o/p: 97

ex: print frequencies of character (in sorted order) in a string of lower case alphabets



After first loop




```
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') << " "
             << count[i] << endl;
```

T.C = O(len(str))

A.S = O(1) (26)

Strings in C++

① C-Style Strings :

```
int main()
{
  char str[] = "gfg";
  cout << str;
  return 0;
}
```

compiler takes care of putting '\0' {'g', 'f', 'g'}

g	f	g	\0

strlen(str) = 3
until '\0' it increments count by 1.

```
int main()
{ char str[] = "gfg";
  cout << sizeof(str); // 4
  return 0;
}
```

```
int main()
{
  char str[] = {'g', 'f', 'g', '\0'};
  cout << str;
  return 0;
}
```

without \0 it gives segmentation fault until finding \0 it prints diff chars.

⇒ Functions
- strlen(str)
- strcmp(s1, s2) ⇒ lexographic comparison (or simply use "gfg")
  - $s_1 > s_2$ ⇒ returns +ve value
  - $s_1 = s_2$ ⇒ returns 0
  - $s_1 < s_2$ ⇒ returns -ve value

```
int main()
{ char s1[] = "abc";
  char s2[] = "bcd";
  int res = strcmp(s1, s2);
  if (res > 0)
    cout << "greater";
  else if (res == 0)
    cout << "same";
  else cout << "smaller";
}
```

s1 is smaller.

- strcpy(s1, s2)

```
int main()
{
  char str[5];
  str = "gfg";
  cout << str;
  return 0;
}
```

compiler error can't assign something to an address.

strcpy(str, "gfg");

② C++ Strings

C++ strings are like vectors

- Richer library
- Supports operators like +, <, >, ==, <=, >= are possible.
- can assign a string later → `string str; str = "gfg";`
- Do not have to worry about size.
- can be converted to C-style if needed.

⇒
```
int main()
{ string str = "geeksforgeeks"; // 0 1 2 ... 12
  cout << str.length() << " "; // 13
  str = str + "xyz";
  cout << str << "\n"; // geeksforgeeksxyz
  cout << str.substr(1,3) << "\n"; // eek
  cout << str.find("eek") << " "; // 1
  return 0;
}
```

substr(beginning index, length)

find(): index of first occurrence-1 if not present returns string::npos

1) string is a class in C++
2) classes have → 1) data members 2) functions
3) Typically data members accessed through functions
4) obj is a variable of a class. (objects)

```
int main()
{ String s1 = "abcd"
  String s2 = "bcd"
  if (s1 == s2)
     cout << "Same";
  else if (s1 < s2)
     Cout << "Greater";
  else
     cout << "Smaller";
  return 0;
}
```

O/P : Greater

Reading Strings from Console :

```
int main()
{
  string str;
  cout << "Enter your name";
  cin >> str;   → It doesn't read spaces (Abdhesha gandla)
  cout << "\n your name is " << str;
  return 0;
}
```

getline(cin, str);

getline(cin, str, 'a');

on input if it sees 'a' it stop printing after a

Iterating through a String :

```
int main()
{ String str = "geeksforgeeks";
  for (int i=0; i< str.length(); i++)
     cout << str[i];
          (OR)
  for (char x : str)
     cout << x;
}
```

palindrome check :

I/P : str = "ABCDCBA"
O/P : yes

I/P : str = "ABBA"
O/P : yes

I/P : str = "geeks"
O/P : No

Naive method :

```
bool ispal (string &str)
{ string rev = str;
  reverse (rev.begin(), rev.end());
  return (rev == str);
}
```

str = "ABCBA"
rev = "ABCBA"

⇓

T.C = Θ(n)
A.S = Θ(n)

efficient method :

T.C = O(n)
A.S = O(1)

A B C B A
↑ ↑ ↑ ↑ ↑

```
bool ispal (String &str)
{ int begin = 0;
  int end = str.length() - 1;
  while (begin < end)
  { if (str[begin] != str[end])
       return false;
    begin++;
    end--;
  }
  return true;
}
```

check if a string is subsequence of other

I/P: s_1 = "ABCD"
s_2 = "AD"
O/P: TRUE

I/P: s_1 = "ABCDE"
s_2 = "AED"
O/P: FALSE

Substring: Contiguous chars
Subsequence

All subsequences of "ABC" are: $[2^n]$
"", "A", "B", "C", "AB", "AC", "BC", "ABC"

Naive: T.C = $O(2^n \times n)$

Idea for solution:

s_1 = GEEKSFORGEEKS
s_2 = GRGES

```
if(s1[i]==s2[j]) {i++; j++;}
else {i++;}
```

Iterative Solution:

```
bool isSubseq(string ss1, string ss2, int n, int m)
{ int j=0;
  if(n<m) return false;
  for(int i=0; i<n && j<m ; i++)
  { if(s1[i]==s2[j])
      j++;
  }
  return (j==m);
}
```

T.C = $O(n+m)$
A.S = $O(1)$

s_1 = "ABCDEF"
s_2 = "ADE"

Initially: j=0
i=0: j=1
i=1:
i=2:
i=3: j=2
i=4: j=3

Recursive Solution:

```
bool isSubseqRec(string ss1, string ss2, int n, int m)
{ if(m==0)
    return true;
  if(n==0)
    return false;
  if(s1[n-1]==s2[m-1])
    return isSubseqRec(s1, s2, n-1, m-1);
  else
    return isSubseqRec(s1, s2, n-1, m);
}
```

T.C = $O(n+m)$
A.S = $O(n+m)$

isSSR("ABC", "AC", 3, 2)
↳ isSSR("ABC", "AC", 2, 1)
↳ isSSR("ABC", "AC", 1, 1)
↳ isSSR("ABC", "AC", 0, 0)

checking for Anagram

I/P: s_1 = "listen", s_2 = "silent"
O/P: yes

I/P: s_1 = "aaacb", s_2 = "cabaa"
O/P: yes

I/P: s_1 = "aab", s_2 = "bab"
O/P: No

Naive solution:

```
bool areAnagram (string &s1, string &s2)
{ if (s1.length() != s2.length())
      return false;
  sort (s1.begin(), s2.end());
  sort (s2.begin(), s2.end());
  return (s1 == s2);
}
```

$s_1 = "abaaac"$
$s_2 = "aacbaa"$

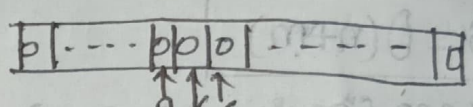
After sorting:
$s_1 = "aaaabc"$
$s_2 = "aaaabc"$

$$T.C = O(n \log n)$$

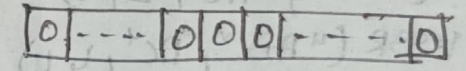
Efficient solution:

$s_1 = "aabcca"$
$s_2 = "acabca"$

Initially:



After the loop:



```
const int CHAR = 256;
bool areAnagram (string &s1, string &s2)
{ if (s1.length() != s2.length())
      return false;
  int count[CHAR] = {0};
  for (int i=0; i<s1.length; i++)
  { count [s1[i]]++;
    count [s2[i]]--;
  }
  for (int i=0; i<CHAR; i++)
      if (count[i] != 0)
          return false;
  return true;
}
```

~~T.C = O(n)~~
~~A.S = O(1)~~

$$T.C = O(n + CHAR)$$
$$A.S = O(CHAR)$$

Strings

Leftmost repeating character:

I/p: str = "geeksforgeeks"

o/p: 0

I/p: str = "aBbcc"

o/p: 1

I/p: str = "abcd"

o/p: -1

Naive Approach:

T.C = $O(n^2)$
A.S = $O(1)$

```
int leftmost (string &str)
{
  for (int i=0; i<str.length(); i++)
  {
    for (int j= i+1; j< str.length(); j++)
    {
      if (str[i] == str[j])
        return i;
    }
  }
  return -1;
}
```

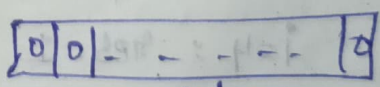
"cabbad"
return 1.

Better Approach:

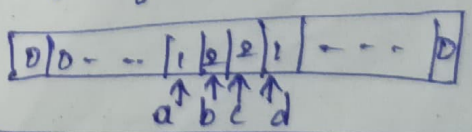
Two ~~times~~ we traversed through string.

```
const int CHAR = 256;
int leftmost (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;
```

abccbd



count []
After first loop.



T.C = O(n) but two loops
A.S = O(CHAR)

efficient Approach-1 :-

T.C = O(n+CHAR)
A.S = O(CHAR)

```
const int CHAR = 256;
int leftmost (string & str)
{
  int finder[CHAR];
  fill (finder, finder+CHAR, -1);
  int res = INT_MAX;
  for(int i=0; i<str.length(); i++)
  {
    int fi = finder[str[i]];
    if (fi == -1)
      finder[str[i]] = i;
    else res = min (res, fi);
  }
  return (res == INT_MAX)?-1 : res;
}
```

str = "abccbd"

Initially : res = ∞

finder[]

-1	-1	- - - -	-1	-1

i=0 : finder['a'] = 0

i=1 : finder['b'] = 1

i=2 : finder['c'] = 2

i=3 : res = 2

i=4 : res = 1

i=5 : finder['d'] = 5

efficient Approach-2 :

T.C = O(n+CHAR)
A.S = O(CHAR)

```
const int CHAR = 256;
int leftmost (string & str)
{
  bool visited[CHAR];
  fill (visited, visited+CHAR, false);
  for (int i=str.length()-1; i>=0; i--)
  {
    if (visited[str[i]])
      res = i;
    else
      visited[str[i]] = true;
  }
  return res;
}
```

str = "abccbd"

Initially res = -1

F	F	- - - -	F

i=5 : visited['d'] = true

i=4 : visited['b'] = true

i=3 : visited['c'] = true

i=2 : visited['c'] = true res=2

i=1 : ~~visited~~ res=1

i=0 : visited['a'] = true

Leftmost Non-Repeating Element

i/p: str = "geeksforgeeks"
o/p: 5 // index of 'f'

i/p: str = "abcabc"
o/p: -1

i/p: str = "apple"
o/p: 0 // index of 'a'.

Naive solution:

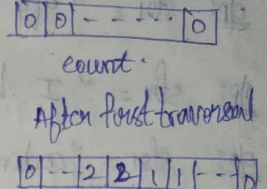
T.C = O(n²)
A.S = O(1)

```
int nonRep(string &str)
{ for(int i=0; i<str.length(); i++)
  { bool flag=false;
    for(int j=0; j<str.length(); j++)
    { if(i!=j && str[i]==str[j])
      { flag=true;
        break;
      }
    }
    if(flag==false)
      return i;
  }
return -1;
}
```

"geeksforgeeks"

Better solution [Two Traversals]

"abbcda"



T.C = O(n+CHAR)
A.S = O(CHAR)

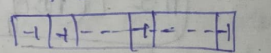
```
const int CHAR=256;
int nonRep(string &str)
{ int count[CHAR]={0};
  for(int i=0; i<str.length(); i++)
      count[str[i]]++;
  for(int i=0; i<str.length(); i++)
    if(count[str[i]]==1)
      return i;
  return -1;
}
```

Efficient Solution (One Traversal)

T.C = O(n+CHAR)
A.S = O(CHAR)

```
const int CHAR=256;
int nonRep(string &str)
{ int fI[CHAR];
  fill(fI, fI+CHAR, -1);
  for(int i=0; i<str.length(); i++)
  { if(fI[str[i]]==-1)
      fI[str[i]]=i;
    else
      fI[str[i]]=-2;
  }
  int res=INT_MAX;
  for(int i=0; i<CHAR; i++)
    if(fI[i]>=0)
      res=min(res, fI[i]);
  return (res==INT_MAX)?-1:res;
}
```

0 1 2 3 4 5 6
a b b c b d a
fI[]



i=0 : fI['a']=0
i=1 : fI['b']=1
i=2 : fI['b']=-2
i=3 : fI['c']=3
i=4 : fI['b']=-2
i=5 : fI['d']=5
i=6 : fI['a']=-2

Reverse Words in a String:

Ip: str = "welcome to gfg"

Op: str = "gfg to welcome"

Ip: str = "I love coding"

Op: str = "coding love I"

Ip: str = "abc"

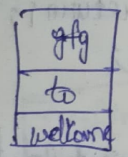
Op: str = "abc"

Naive method:

① create a stack

② push words one by one the stack.

③ pop words from the stack and append to output.



str = " "

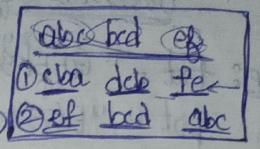
str = "gfg to welcome"

A.S = O(no of words) + O(n)

↓ Individual words

Idea: A.S = O(1)

T.C = O(len(str))




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

⇒ 0 1 2 3 4 5 6 7 8 9 10 11 12 13

welcome to gfg

start = 0, n = 14

end = 0, 1, 2, 3, 4, 5, 6 : No change

end = 7 : emocleW to gfg.

start = 8

end = 8, 9 : No change

end = 10 : emocleW ot gfg

start = 11

end = 11, 12, 13 : No change.

After loop: emocleW ot gfg // last word reversed

gfg to welcome // whole string reversed

In case you initialise a character array like below

char str[] = "welcome to gfg";

do pass array size minus one as parameter.

Overview of pattern searching

I/p: txt = "GEEKSFORGEEKS"
pat = "EKS"

O/p: 2 10

I/p: txt = "AAAAA"
pat = "AAA"

O/p: 0 1 2

I/p: txt = "ABCABCD"
pat = "ABD"

O/p: Not present.

⇒ m → pattern length
n → Text length
$1 \le m \le n$

Naive: $O((n-m+1)*m)$ → check every window of txt
No preprocessing

Naive when all characters of pattern are distinct: O(n)

Robin Karp: $O((n-m+1)*m)$ (hash fn) [preprocess pattern]
But Better than Naive on Average (Redundancy)

kmp: O(n)

Suffix Tree: O(m) [preprocess Text]

Naive pattern Searching

I/p: txt = "ABABABCD"
pat = "ABAB"

O/p: 0 2

I/p: txt = "ABCABCD"
pat = "ABCD"

O/p: 3

I/p: txt = "AAAAA"
pat = "AAA"

O/p: 0 1 2

Slide the pattern over the text one by one
(window comparing)

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

txt → ABCABCD
pat → ABCD

i=0 ⇒ ABCABCD
ABCD X

i=1 ⇒ ABCABCD
ABCD

i=2 ⇒ ABCABCD
ABCD

i=3 ⇒ ABCABCD
ABCD

O/p: 3

A.S = O(1)

T.C = $O((n-m+1)*m)$

worstcase ⇒ AAAAAA
AAA

Improved Naive Algorithm for Distinct pattern

I/p: txt = "ABCABCD ABCD"
pat = "ABCD"
o/p: 3 7

I/p: txt = "GEEKSFORGEEKS"
pat = "EKS"
o/p: 2 10

I/p: txt = "ABCAAAD"
pat = "ABD"
o/p: (not found)

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

T.C = O(n)
A.S = O(1)

txt = ABCABCD
pat = ABCD
ABCD

o/p: 3

txt = ABCEABEFABCD
pat = ABCD
n=12 m=4

i=0: ⇒ i += 3 ⇒ i = 0+3
i=3 ⇒ i=4
i=4 ⇒ i = 4+2 = 6
i=6 ⇒ i=7
i=7 ⇒ i=8
i=8 ⇒ j=4
print(8)

Rabin Karp Algorithm

I/p: txt = "abdabdgabc"
pat = "abc"
o/p: 3 7

I/p: txt = "aaaaaa"
pat = "aaa"
o/p: 0 1 2

I/p: txt = "abcd"
pat = "xyz"
o/p: not found

Application: Used to search multiple patterns in a text

① Like naive algorithm, slide the pattern one by one.
② Compare hash values of pattern and current text window. If hash values match, then only compare individual characters.

txt = "0 1 2 3 4 5 6 7 8 9 abd abc babc"
pat = "abc"

a: 1
b: 2
c: 3
d: 4
e: 5

p: Hash value of pattern
t: Hash value of current window of text

p = (1+2+3) = 6 ⇒ T.C = O(n²)

i=0: t = (1+2+4) = 7
i=1: t = (2+4+1) = 7
i=2: t = (4+1+2) = 7
i=3: t = (1+2+3) = 6 (match)
i=4: t = (2+3+2) = 7
i=5: t = (3+2+1) = 6 (Spurious Hit) ⇒ (bca, abc) Not same
i=6: t = (2+1+2) = 5
i=7: t = (1+2+3) = 6 (match)

Simple Hash: Sum of values
problem: Spurious Hits

Rolling Hash:

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

m : length of pattern

Improved Hash: (To avoid spurious hits)

d = 5

Weighted sum of characters of string (ASCII values)

a : 1
b : 2
c : 3
d : 4
e : 5

$h("abc") = 1 \times d^2 + 2 \times d^1 + 3 \times d^0$

$= 1 \times 5^2 + 2 \times 5^1 + 3 \times 5^0 = 38$

$h("dab") = 4 \times d^2 + 1 \times d^1 + 2 \times d^0$

$= 4 \times 5^2 + 1 \times 5^1 + 2 \times 5^0 = 107$

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

m ⇒ length of pattern

txt = "abdabcbabc"
(0 1 2 3 4 5 6 7 8 9)

pat = "abc"

$t_0 = 1 \times 5^2 + 2 \times 5^1 + 4 \times 5^0 = 39$

$t_1 = 5(t_0 - 1 \times 5^2) + 1 = 71$

$t_2 = 5(t_1 - 2 \times 5^2) + 2 = 107$

$t_3 = 5(t_2 - 4 \times 5^2) + 3 = 38$

$p = 1 \times 5^2 + 2 \times 5 + 3 \times 5^0$

$= 38$

txt = "132456"

d = 10 m = 4

$t_0 = 1324$

$t_1 = ?$ ⇒ 3245

$t_1 = (324 \times 10 + 5)$

$= (1324 - 1 \times 10^3) \times 10 + 5$

$= (324 \times 10) + 5$

$= 3245$

Why %q?

When we do weighted sum, this number can become large even for smaller strings (q ⇒ prime no) we like to choose as big as possible so that there are less spurious hits

Horner's Rule:

$P = pat[0] * d^{m-1} + pat[1] * d^{m-2} + \cdots + pat[m] * d^0$

$P = 0$

$P = p*d + pat[0] \Rightarrow P = pat[0]$

$P = (p*d + pat[1]) \Rightarrow pat[0]*d + pat[1]$

$P = (p*d + pat[2]) \Rightarrow pat[0]*d^2 + pat[1]*d + pat[2]$

```
void RBSearch(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 + txt[i])%q;
    }
    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])%q;
            if(t<0) t=t+q;
        }
    }
}
```

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

Compute p and t_0

check for spurious hit

Compute t_{i+1} using t_i

for comp +ve value is required

-3%13
(3+13)%13
10%13

Time Complexity:

worst case: txt = "AAAAA" pat = "AAA" $\Rightarrow O((N-M+1)\times M)$

⇓

on Avg works better than Naive

KMP Algorithm

Constructing longest proper prefix Suffix Array (LPS)

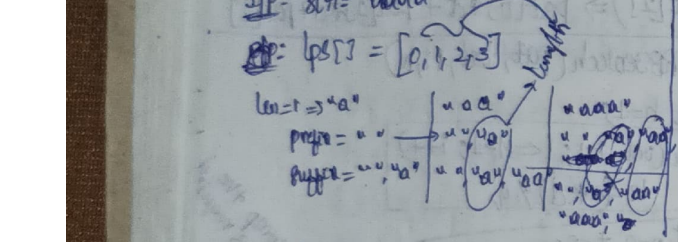
(length of prefix < string length)

Ip: str = "ababc"

Op: lps[] = [0, 0, 1, 2, 0]

Ip: str = "aaaaa"

Op: lps[] = [0, 1, 2, 3, 4]



proper prefixes of "abc"

" ", "a", "ab"

suffixes of "abc"

" ", "c", "bc", "abc"

proper prefixes of "abcd"

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

proper suffixes of "abcd"

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

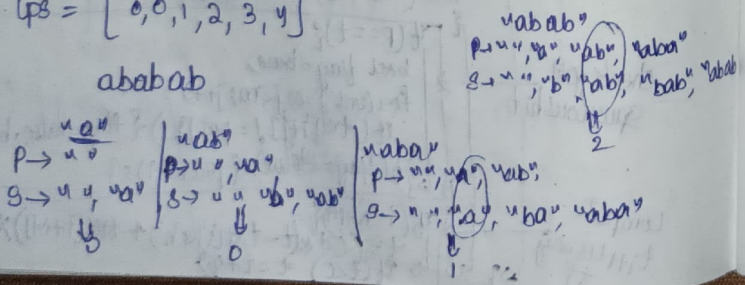
Ip: str = "abcd"

Op: lps[] = [0, 0, 0, 0]

Ip: str = "ababab"

lps = [0, 0, 1, 2, 3, 4]

ababab



Exercise:

str = "abacabad"

lps = [0, 0, 1, 0, 1, 2, 3, 0]

str = "abbabab"

lps = [0, 0, 0, 1, 2, 3]

Naive: $O(n^3)$ A-S = $O(n)$

str = ababacab n=8

len=7

str[0], str[1]

str[1], str[2]

⋮

str[6], str[7]

len=6

str[0], str[2]

str[1], str[3]

⋮

str[5], str[7]

len=5

str[0], str[3]

str[1], str[4]

str[4], str[7]

n=8

len=7 ababacab

len=6 ababacab

len=5 ababacab

len=4 ababacab

len=3 ababacab

len=2 ababacab ✓

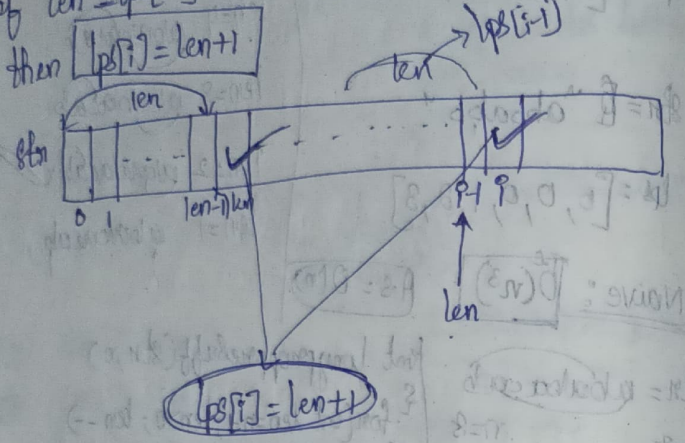
len=1 ababacab

```
int LongProppreSuff(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;
    if (flag == true)
      return len;
  }
  return 0;
}
Void fillLPS (str, lps[])
{ for(int i=0; i<str.length(); i++)
    lps[i] = longpropprefsuff(str, i+1);
}
```

Effecient : O(n)

Basic Ideas:

① If len = lps[i-1] and str[len] and str[i] are same,
then lps[i] = len+1



str[] = aaaa
lps[] = [0 1 2 3]

i=0 : len=0, lps[0]

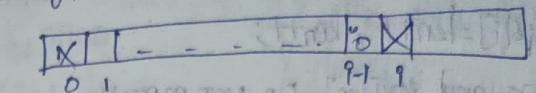
i=1 : str[i] and str[len] are same
lps[1]=1, len=1

i=2 : str[i] and str[len] are same
lps[2]=2, len=2

i=3 : str[i] and str[len] are same
lps[3]=3, len=3

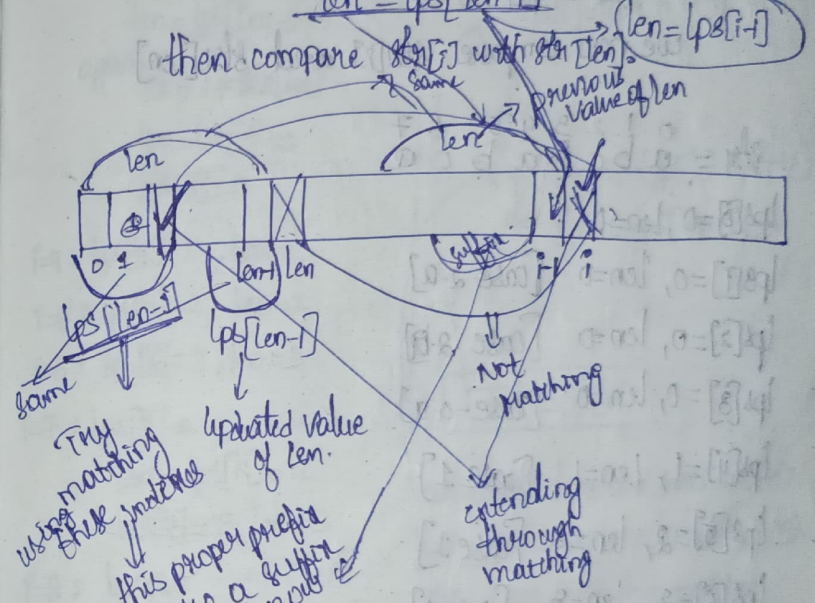
② If str[i] and str[len] are not same.

(a) If, len==0



lps[i]=0

b) Else, we recursively apply lps[]
len = lps[len-1]
then compare str[i] with str[len].



Basic Ideas: len=lps[i-1]

① If str[i] & str[len] match

lps[i] = len+1, len++;

② If str[i] and str[len] DO NOT MATCH

a) If len=0, lps[i]=0

b) Else

len = lps[len-1]

we now compare str[i] and str[len]

	0	1	2	3	4	5	6	7
str =	a	b	c	b	a	b	c	a

lps[0]=0, len=0

lps[1]=0, len=0 [case 2-a]

lps[2]=0, len=0 [case 2-b]

lps[3]=0, len=0 [case-a a]

lps[4]=1, len=1 [case 1]

lps[5]=2, len=2 [case 1]

lps[6]=3, len=3 [case 1]

lps[7] [case 2-b]

len = lps[len-1] = lps[2] = 0

[case-1]

lps[7]=1, len=1

Implementation of O(n) Algorithm:

str = "AAABAAAAC"

len=0, lps[0]=0

i=1: lps[1]=1, len=1

i=2: lps[2]=2, len=2

i=3: lps[3]

str[i] ≠ str[len] and len>0

len = lps[len-1] = lps[1] = 1

again str[i] ≠ str[len]

len = lps[0] = 0

lps[3]=0

i=4: lps[4]=1, len=1

i=5: lps[5]=2, len=2

i=6: lps[6]=3, len=3

i=7: lps[7]

len = lps[2] = 2

lps[7]=3, len=3

i=8: lps[8]

len = lps[2] = 2

len = lps[1] = 1

len = lps[0] = 0

lps[8]=0

```
void fillLps(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]; }
    }
  }
```

T.C = θ(n)

all characters are same ⇒ θ(n)

all character are distinct ⇒ θ(n)

when some characters matched and thereafter mismatches ⇒ almost 2N work

KMP String Matching Algorithm

I/p: txt = "abcdefg" pat = "bcd"
O/p: 1

I/p: txt = "aaaaaab" pat = "aaaaa"
O/p: 0 1

I/p: txt = "abcd" pat = "ba"
O/p: NOT FOUND

Naive:
$O((n-m)\times m)$ Time
$O(1)$ Aux Space

Robin Karp:
$O((n-m)\times m)$ Time
$O(1)$ Aux Space.

Naive for Distinct characters:
$O(n)$ Time
$O(1)$ Aux Space

kmp:
$O(n)$ Time ⇒ atmost 2N
$O(m)$ Aux Space

n ⇒ length of text
m ⇒ length of pat

n=6 txt = a a a a a b (0 1 2 3 4 5)
m=5 pat = a a a a a

a a a a
a a a a
a a a b

Naive: 12 comparisons
$(n-m+1)\times m$

kmp:

lps[m] = a a a a a
[0, 1, 2, 3]

lps[i] → Longest proper prefix of pat[0...i] which is also a suffix.

n=6 txt = a a a a a b
m=5 pat = a a a a a

a a a a

o/p: 0 1

for next step it looks up the lps array for last matched character lps[] = [0, 1, 2, 3]

⇒ whenever both are matched ⇒ In txt we always ↑'s index
In pattern we always consult lps array

a a a a a
a a a a

prefix
suffix

In kmp:
At every iteration we are moving ahead in text or sliding the pattern
at most N times sliding occurs
at most N times text moving ahead.
T.C = $O(N+N) = O(2n)$

```
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++; }
    if (j==M) { print(i-j); j = lps[j-1]; }
    else if (i<N && pat[j]!=txt[i])
    { if (j==0) { i++; }
      else { j = lps[j-1]; } // after some matches there was mismatches
    }
  }
```

txt = ababcababaaad

pat = ababa

lps[] = [0, 0, 1, 2, 3]

i=0, j=0

ababcababaaad
ababa

i=1, j=1

i=2, j=2

i=3, j=3

i=4, j=4 ⟹ j = lps[4-1] = lps[3] = 2

i=4, j=2

ababcababaaad
ababa

j = lps[2-1] = lps[1] = 0

i=4, j=0

ababcababaaad
ababa

i != j ⇒ i = 5

i=5, j=0 ⟹ i=6, j=1

i=7, j=2

i=8, j=3

i=9, j=4

i=10, j=5 ⟹ print(5)

j = lps[5-1] = lps[4] = 3

i=10, j=3

ababcababaaad
ababa

j = lps[3-1] = lps[2] = 1

i=10, j=1

ababcababaaad
ababa

i=10, j = lps[1-1] = lps[0] = 0

i=10, j=0

ababcababaaad
ababa

i=11, j=1

j = lps[1-1] = lps[0] = 0

i=11, j=0

ababcababaaad
ababa

i=12

i>N

kmp stops

check for Rotation

Ip: s_1 = "ABCD", s_2 = "CDAB"

Op: yes ABCD → BCDA → CDAB

Ip: s_1 = "ABAAA", s_2 = "BAAAA"

Op: yes ABAAA

Ip: s_1 = "ABAB", s_2 = "ABBA"

Op: No

Naive Solution: T.C = $O(n^2)$

efficient solution 1: s_1 = "ABCD", s_2 = "CDAB" } pattern search on text circularly

s_1 = "ABAB", s_2 = "ABBA" } Naive (or) KMP

efficient solution 2: T.C = O(n), AS = O(n)

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

s_1 = "ABCD", s_2 = "CDAB", s_1+s_1 = "ABCDABCD"

Anagram Search

Two strings which are permutations of each other which are contiguous

Ip: txt = "geeksforgeeks"
pat = "frog"

Op: yes

Ip: txt = "geeks-forgeeks"
pat = "qseek"

Op: No

Naive solution

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

T.C = O((n-m+1)×m)
A.S = O(CHAR)

Efficient Solution

Sliding window technique:

```
bool isPresent(string &txt, string &pat)
{
  int CT[CHAR]={0}, CP[CHAR]={0};
  for(int i=0; i<pat.length(); i++)
  {
    CT[txt[i]]++;          ⟹ O(m)
    CP[pat[i]]++;
  }
  for(int i=pat.length(); i<txt.length(); i++)
  {
    if(areSame(CT,CP))     ⟹ O((n-m)*CHAR)
      return true;
    CT[txt[i]]++;
    CT[txt[i-pat.length()]]--;
  }
  return false;
}
```

geeksforgeeks

CT: g e k s f o r

CP: g e k s

$$T.C = O(m + (n-m) \times CHAR) = O(n * CHAR)$$

$$A.S = \theta(CHAR)$$

LEXICOGRAPHIC Rank

I/P: str = "BAC"

O/P: 3

pen and paper method:

(1) sort the str ⇒ "ABC"

(2) Generate all permutations in increasing order

String	Rank
"ABC"	1
"ACB"	2
"BAC"	3
"BCA"	4
"CAB"	5
"CBA"	6

I/P: str = "CBA"

O/P: 6

I/P: str = "DCBA"

O/P: 24

I/P: str = "STRING"

O/P: 598

Naive solution: $O(n * n!)$ — $n!$: permutations

Efficient Solution: Count of lexographically smaller strings + 1

"STRING"

Idea:

① Traverse through the string from left to right

② for every characters see how many characters are smaller than the present character on Right side

③ If we replace the character with those smaller characters on right side we get lexographically smaller strings.

S is not as a 1st character so Replace S with every character.

S > A, I, N, G

① R _ _ _ _ _ ⇒ 5! strings
I _ _ _ _ _ ⇒ " "
N _ _ _ _ _ ⇒ " "
G _ _ _ _ _ ⇒ " "

4 × 5! strings

② count "S" as a least character (Begin with S & do not Begin with T) on Right of T

T > R, I, N, G

S R _ _ _ _
S I _ _ _ _
S N _ _ _ _
S G _ _ _ _

4 × 4! strings

③ R > I, N, G

S T I _ _ _
S T N _ _ _
S T G _ _ _

3 × 3! strings

④ I > G

S T R G _ _ } 1 × 2!

⑤ S T R I G _ } 1 × 1!

4 × 5! + 4 × 4! + 3 × 3! + 2 × 2 + 1 × 1!
= 597

Ans = 597 + 1 = 598

"DCBA"

D > A, B, C

A _ _ _
B _ _ _ } 3 × 3!

C > A, B

D A _ _
D B _ _ } 2 × 2!

B > A

D C A _ } 1 × 1!

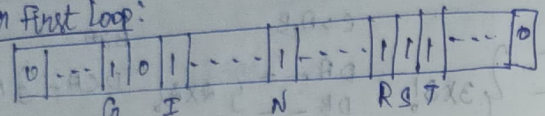
Ans = 3 × 3! + 2 × 2! + 1 × 1! = 3 × 6 + 2 × 2 + 1
= 18 + 4 + 1 = 23 + 1
= 24

```
const int CHAR = 256;
int lexRank(string &str)
{
    int res = 1;
    int n = str.length();
    int mul = fact(n);
    int count[CHAR] = {0};
    for(int i=0; i<n; i++)
        count[str[i]]++;                    } O(n)
    for(int i=1; i<CHAR; i++)
        count[i] += count[i-1];              } O(CHAR)
    for(int i=0; i<n-1; i++)
    {
        mul = mul/(n-i);
        res = res + count[str[i]-1] * mul;  } O(n × CHAR)
        for(int j=str[i]; j<CHAR; j++)
            count[j]--;
    }
    return res;
}
```

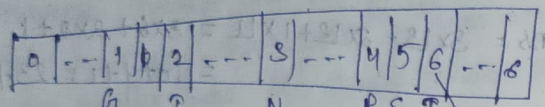
T.C = O(CHAR + n + n × CHAR)
= O(n × CHAR)
A.S = O(CHAR)

"STRING"

After first Loop:



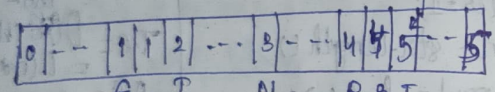
After second Loop:



mul = 6! = 720

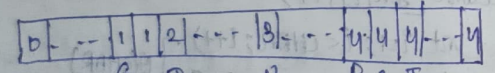
i=0 : mul = 120

res = 1 + 4 * 120 = 481



i=1 : mul = 24

res = 481 + 4 * 24 = 577

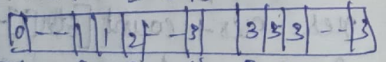


i=2 : mul = 6

res = 577 + 3 * 6 = 595

i=3 : mul = 2

res = 595 + 2 * 1 = 597



= 597

i=4 : res = 598 + 1 * 1 = 598

Longest Substring with Distinct characters

I/p: str = "abcdabc" I/p: str = "aaa" I/p: str = ""

O/p: 4 O/p: 1 O/p: 0

I/p: str = "abaacdbab"

O/p: 4

Naive : $O(n^3)$

```
bool areDistinct(string str, int i, int j)
{
    vector<bool> visited(256);
    for (int k=i; k<=j; k++)
    {
        if (visited[str[k]] == true)
            return false;
        visited[str[k]] = true;
    }
    return true;
}
int longestDistinct(string str)
{
    int n = str.length(), res = 0;
    for (int i=0; i<n; i++)
        for (int j=i; j<n; j++)
            if (areDistinct(str, i, j))
                res = max(res, j-i+1);
    return res;
}
```

O(n) — areDistinct

O(n²) — i => start index, j => end index

"abac" substrings:

a
ab
aba
abac
b
ba
bac
a
ac
c

T.C = $O(n^3)$

A.S

~~Efficient~~

Better solution : $O(n^2)$

```
int longestDistinct (string str)
{ int n = str.size(), res=0;
  for (int i=0; i<n; i++)
  { vector<bool> visited (256);
    for (int j=i; j<n ; j++)
    { if (visited[str[j]]==true)
        break;
      else
      { res = max (res, j-i+1);
        visited[str[j]] = true;
      }
    }
  }
  return res;
}
```

Efficient solution: $O(n)$

~~abba~~ a b c a d b d

maxEnd[] : 1 2 3 3 4 4 2

longest substring with distinct characters count.
If the current char is repeated, then we start counting previous repeating char, next char

maxEnd(j) = length of the longest substring that has distinct characters and ends with j.

$$res = \max_{j=0}^{n-1} (maxEnd(j))$$

$x_0, x_1, x_2, x_3 \ldots x_i \ldots x_{j-1}, x_j \ldots x_{n-1}$

$maxEnd(j-1)$

$maxEnd(j) = maxEnd(j-1)+1$ ⇒ if str[j] is not present in $maxEnd(j-1)$

$j - prev[str[j]] + 1$ ⇒ when prev(str[j]) is previous of the character str[j]

```
int longestDist (string str)
{ int n= str.length(), res=0;
  Vector <int> prev(256, -1);
  int i=0;
  for (int j=0; j<n ; j++)
  { i= max (i, prev[str[j]]+1);
    int maxEnd = j-i+1;
    res = max (res, maxEnd);
    pre[str[j]] = j;
  }
  return res;
}
```

prev[ ] = [-1, -1, -1, - - - -1]

i=0, res=0

0	1	2	3	4	5	6
a	b	c	a	d	b	d

j=0, i=0, maxEnd=1, res=1, prev['a']=0

j=1, i=0, maxEnd=2, res=2, prev['b']=1

j=2, i=0, maxEnd=3, res=3, prev['c']=2

j=3, i=1, maxEnd=3, res=3, prev['a']=3

j=4, i=1, maxEnd=4, res=4, prev['d']=4

j=5, i=2, maxEnd=4, res=4, prev['b']=5

j=6, i=5, maxEnd=2, res=4, prev['d']=6