Today's Content

* Length of longest substring with all distinct chars.
* Anagrams Check.
* Count of Permutation of A in B.
* Class of Anagrams.

Q1: Given string S. Find length of longest substring without repeating characters.

$1 <= |S| <= 10^6$

Eg: S = a b c a b c b b — 3

S = a a b b — 2

S = a b c a b c d — 4

BF approach: Generate all possible substrings & check if it has all distinct chars. ↳ $\frac{N(N+1)}{2} * N \Rightarrow O(N^3)$

Idea2: With every s[i] as start index of substring, get max length window which contains all distinct chars.

	0	1	2	3	4	5	6	7
S =	a	b	a	c	e	a	f	f
	2	4	3	4	3	2	1	1

ans = 0

TC: $O(n^2)$

SC: $O(n)$

```
for(i=0; i<n; i++){
    HashSet<char> hs  // hs = set()
    for(j=i; j<n; j++){
        if(hs.search(s[j]) == false){
            hs.add(s[j])
        }
        else{ break }
    }
    ans = max(ans, hs.size())
}
```

idea 3: Use 2 pointers approach

	0	1	2	3	4	5	6	7	8
s =	a	b	a	c	e	a	f	g	f
						l			r

ans = ~~4~~ 5

l	r	str	s[r] in hs?
0	0	a	x
0	1	ab	x
0	2	aba	
1	2	b	
1	3	bac	
1	4	bace	
1	5	bacea	
2	5	acea	
3	5	cea	
3	6	ceaf	
3	7	ceafg	
3	8	ceafgf	
4	8	eafgf	

hs: { ~~a~~ ~~b~~ ~~a~~ ~~c~~ ~~e~~ a f g }

∴ Variable Sliding Window

Pseudocode :

```
int longestSub(String S){
    HashSet<char> hs
    ans = 0
    l = 0, r = 0
    while(r < S.length){
        if(hs.find(S[r]) == false){
            hs.add(S[r])
            r++
            ans = max(ans, hs.size())
        }
        else{ hs.delete(S[l])   // repeating char found!
            l++
        }
    }
    return ans
}
```

TC: O(n)
SC: O(n)

idea 4: Binary Search

a) Target : Length of longest substring
b) Search space: l (1) , r (N)

	l	r	m
String with 10 chars	1	10	5

* is it possible to have substring of len = 5 with distinct chars?
↳ Sliding window! ⇒ O(n)

Pattern: 1 2 3 4 5 ..... N
T T T T T F

Look for last occurence of True.

TC: O(N log N), SC: O(N)

Qn: Given 2 strings A & B. Check if they are anagrams of each other.

Anagrams: 2 strings are said to be anagrams of each other if they are permutations of each other.

Eg:

tac	cat	✓
satya	ytas	✗
lps	pls	✓
madam	dadam	✗

idea 1: Sort + Compare

TC: $O(n \log n + n)$

idea: Frequency of all chars in HM should be same

Step1: Insert all chars of A in hm1 } O(n

Step2: Insert all chars of B in hm2 }

Step3: Iterate over hashmap & compare if A & B have same chars or not. ⇒ O(n)

JAVA

```
bool compare (Hashmap<char, int> hm1, hm2) {
    if(hm1.size() == hm2.size()) {
        for(char K: hm1.KeySet()) {
            if(hm2.find(K) == false ||
               hm1[K] != hm2[K]) {
                return false
            }
        }
        return true
    }
    return false
}
```

TC: O(n)

PYTHON

```
for k, v in hm1.items():
    if k not in hm2 or hm2[k] != v:
        return False
```

N M (N >= M)

Qn: Given 2 strings A & B. Count no. of permutations of B in A

{lowercase alphabets}

↳ no. of substrings in A which are anagrams of B.

Eg:

	0	1	2	3	4	5	6	7	8	9	10	11
A (12) :	a	b	c	a	c	c	b	a	b	c	a	c

B : a b a c c

(5)

ideal: For all substrings of len 5 in A, check if they are anagrams of B.

	A	B
0-4	a b c a c	a b a c c
1-5	b c a c c	a b a c c
2-6	c a c c b	a b a c c
3-7	a c c b a	a b a c c
4-8	c c b a b	a b a c c
5-9	c b a b c	a b a c c
6-10	b a b c a	a b a c c
7-11	a b c a c	a b a c c

ans = 3

TC : O(N * M)

SC : O(1)

idea2: Use sliding window

```
   0 1 2 3 4 5 6 7 8 9 10 11
A: a b c a c c b a b c a c
(12)

B: a b a c c
(5)
```

HB
{a:2, b:1, c:2}

	Add	Remove	HA:	HA == HB
0-4			{a:2, b:1, c:2}	✓
1-5	5 (c)	0 (a)	{a:1, b:1, c:3}	✗
2-6	6 (b)	1 (b)	{a:1, b:1, c:3}	✗
3-7	7 (a)	2 (c)	{a:2, b:1, c:2}	✓
4-8	8 (b)	3 (a)	{a:1, b:2, c:2}	✗
5-9	9 (c)	4 (c)	{a:1, b:2, c:2}	✗
6-10	10 (a)	5 (c)	{a:2, b:2, c:1}	✗
7-11	11 (c)	6 (b)	{a:2, b:1, c:2}	✓

Steps:
a) Insert all chars of B in hm2 — ans = 3
b) Insert FIRST B chars of A in hm1 & compare
c)

```
for (i = B.length ; i < A.length ; i++) {
    c = A[i]    //add
    r = A[i - B.length] // remove
    hm1[c]++
    hm1[r]--
    if (hm1[r] == 0) { // freq = 0, remove it
        hm1.delete[r]
    }
    if (compare(hm1, hm2)) { ans++ }
}
return ans
```

TC: O(N = length of A)
SC: O(26) ⇒ O(1)

Qn: Given an array of strings A[N]. Return all groups of strings that are anagrams. Represent the group by 1-based indexing.

Eg:- A = { cat, dog, god, tac }
ans = { [1, 4]
[2, 3] }

A = { dog, cat, bat, rat }
ans = { [1]
[2]
[3]
[4] }

A = { rat, tar, art }
ans = { [1, 2, 3] }

idea: Sort the strings within the array & add them to Hashmap<String, List<int>>

Iterate over the strings & compare with hashmap
if key found: add to ans
else: continue

A = { cat, dog, god, tac }
↳ Sort: { act, dgo, dgo, act }

{ act: [1, 4]
dgo: [2, 3] }

ans = { [1 4], [2, 3] }

Pseudocode :

```
anagrams(String A[N]) {
    Hashmap<String, List<int>> hm
    for(i=0; i<N; i++) {
        Sort(A[i])                    N*(MlogM)
        hm[A[i]].add(i+1) // 1 based indexing
    }

    List<List<int>> ans
    for(i=0; i<N; i++) {              — O(N)
        if(hm.find(A[i])) {
            ans.add(hm[A[i]])
            hm.remove(A[i])
        }
    }

    return ans
}
```

TC: $O(NM \log M)$

SC: $O(N*M + N) = O(N*M)$

N^{th} idx as val being stored.