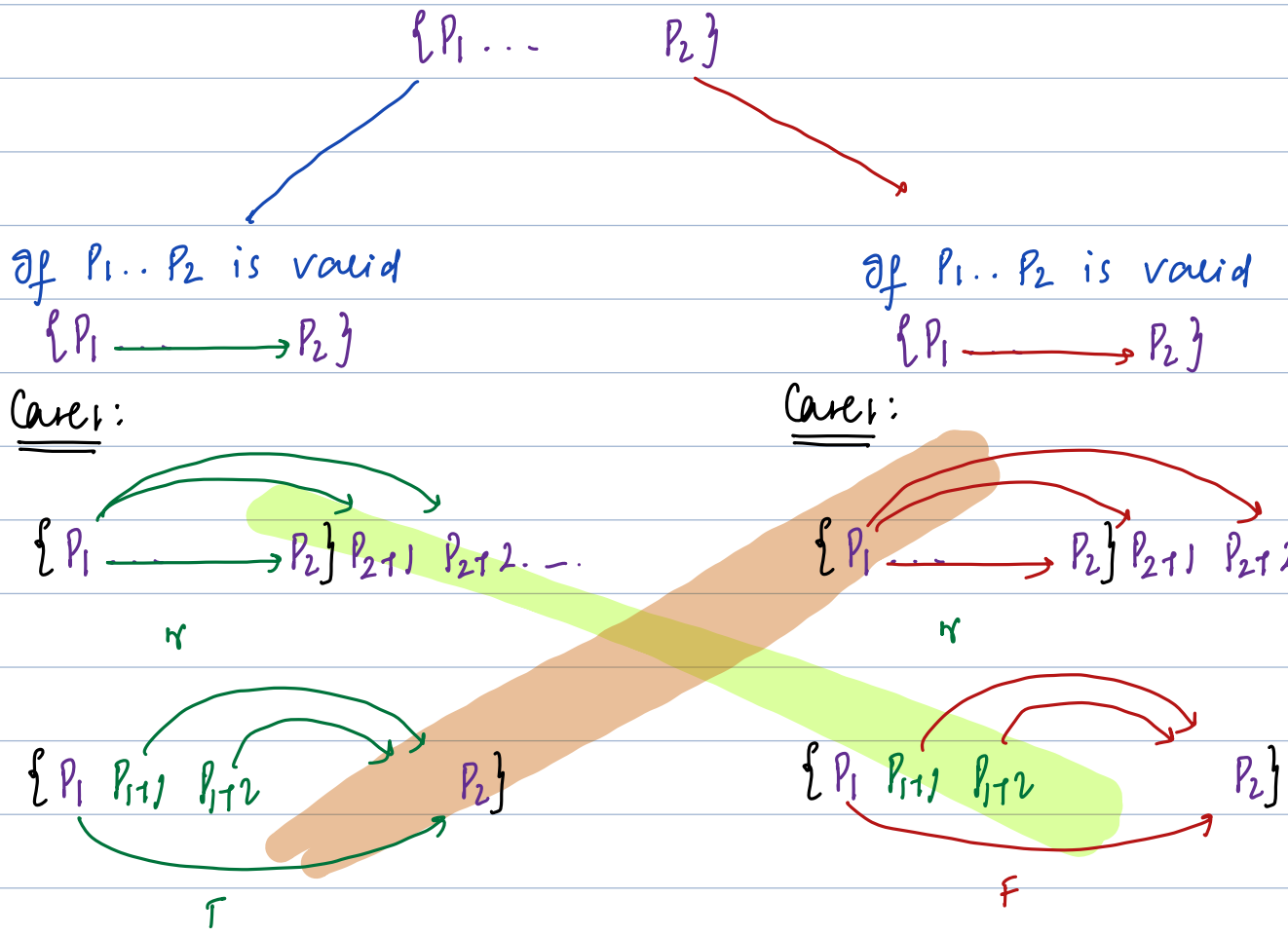
Todays Content:

& length of smallest string in A, which contain all occurence B

#If we want to apply 2 pointer on subarray, show me of the below



38: Minimum Window String:

Given two strings s & t of lengths N and k respectively.

Return length of smallest substring of s which contains all character of T including duplicates

#Note: S & T contain only lower case.

Ex1: T_k = b a b c

	0	1	2	3	4	5	6	7	8	9	10	11	12
S_N =	b	d	a	c	e	b	a	n	c	b	b	a	c

Possible: len

s[0 6) 7

s[10 12] *

s[9 12] 4

Ideal:

1. Insert T[] in tmT : {b:2 c:1 a:1}

Generate all substrings of s:

for a substring of S, check if it contain all character of T. ?

#substring s[x:y) = a a c b c e b a

T[] = b a b c

#Approach:

Insert s[x:y] in tmS : {a:3 c:2 b:2 e:1}

tmT : {b:2 c:1 a:1}

for every char ch in tmT:

tmT[ch] <= tmS[ch]

TC: O(k + N^2 * (N + 26)) SC: O(26) = O(1)

To generate tmS ↳ Two iterate & compare 2 hashmaps.

At max, distinct keys are 26, because, Given S & T contain only smaller alphabets.

Idea2: Optimize using carry forward

Say we have

s[x y] = a a c b c e b a

T[ ] = b a b c

To compare both:

tmpS : {a:3 c:2 b:2 e:1}

tmpT : {b:2 c:1 a:1}

s[1 4)

s[1 5)

Say we have

s[x..y y+1] = a a c b c e b a c

Add, new char to hashmap tmpS

T[ ] = b a b c

To compare both:

tmpS : {a:3 c:3 b:2 e:1}

tmpT : {b:2 c:1 a:1}

TC: O(k + N² * (1 + 26)) SC: O(26) ≈ O(1)

To compare both tmpT & tmpS

To generate tmpS for each substring.

N² substring

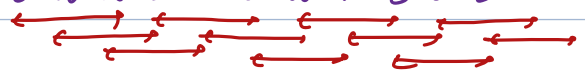
To generate tmpT

Because, we only need to add new char in tmpS

Ideas: Using BS

	0	1	2	3	4
T_N =	b	a	b	c	a

	0	1	2	3	4	5	6	7	8	9	10	11	12
S_N =	b	d	a	c	e	b	a	n	c	b	b	a	c



Target = Min length of substring in S

Search Space = lo: 1 hi: S.length();

Discard ?

lo	hi	m	
1	13	7	: Check if there exists substring of len = 7 which contains all character of T.
			Q: We check all substrings of len 7
			7 8 9 10 ..
			ans = m;
			h = m - 1;
1	6	3	: Check if there exists substring of len = 3 which contains all character of T.
			Q: We check all substrings of len = 3
			1 2 3
			lo = m + 1;

TC: $O(\log_2^N * N) \approx O(N \log_2^N)$ SC: $O(26) \approx O(1)$

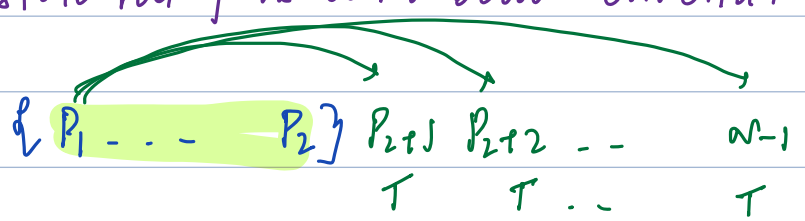
```
int pal(string s, string t){
    int l = 1, h = s.length(), ans = -1; # of no such substring exist.
    while(l <= h){
        int m = (l+h)/2; → # Check if there exists a substring of m
        if( check(s, m, t)){ length in s, which contains all chars in T.
            ans = m;                Try with sliding window TC: O(N)
            h = m-1;
        }
        else{
            l = m+1;
        }
    }
    return ans;
}
```

Ideas: 2 pointers?

Say in $S[P_1 .. P_2]$ we take substring $P_1 .. P_2$.

If it contains all character of T. It's a guarantee that $P_1 .. P_{2+1}] P_{2+2}]$. — will also contain all chars of T.

Above statement holds under below criteria:



DyRun: T_K = a a b b c

	-1	0	1	2	3	4	5	6	7	8	9	10	11	12
S_N =		c	a	b	a	a	c	c	e	b	a	b	a	c
				P_1						P_2				

↳ Does HMS contain all char of HMT

HMT: { a: 2 b: 2 c: 1 }

HMS: { c: 2 a: 2 b: 2 e: 1 }

P_1	P_2	Valid	ans	Update
0	-1	✗	✗	P_2++ HMS[S[P_2]]++;
0	0	✗	✗	P_2++ HMS[S[P_2]]++;
0	1	✗	✗	P_2++ HMS[S[P_2]]++;
0	2	✗	✗	P_2++ HMS[S[P_2]]++;
0	3	✗	✗	P_2++ HMS[S[P_2]]++;
0	4	✗	✗	P_2++ HMS[S[P_2]]++;
0	5	✗	✗	P_2++ HMS[S[P_2]]++;
0	6	✗	✗	P_2++ HMS[S[P_2]]++;
0	7	✗	✗	P_2++ HMS[S[P_2]]++;
0	8	✓	9	HMS[S[P_1]]-- P_1++;
1	8	✓	8	HMS[S[P_1]]-- P_1++;
2	8	TODO		

↳ Update length

↳ Updating hashmap & pointer.

TC: O(K + 2N * {26 + 1 + 1 + 1}) SC: O(26) ≈ O(1)

↳ # Comparing 2 hashmaps

↳ # Total 2 pointer iterating

TC: O(K + 58N) SC: O(1)

≈ O(K+N)

↳ TLE, if can exceed $>10^8$ iterations

DyRvn: TC: O(k + 2N * {1 + 1 + 1 + 1?}) → match value TC: O(k + 8N) = O(N + k)

T_k = a a b b c ↳ valid → ans → freshmap q ptr

0	1	2	3	4	5	6	7	8	9	10	11	12
~~x~~	~~x~~	~~b~~	a	a	c	b	c	e	a	b	a	c
		P_1				P_2						

keep a track of how many of char of hms are matching hmT

hmT: { a: 2 b: 2 c: 1 }

hms: { c: 1 a: 2 b: 1 }

if match == length of T

P_1	P_2	match	Valid	ans	Update	Relevant	
0	-1	0	*	*	P_2++ hms[s[P_2]]++	✓	match++;
0	0	1	*	*	P_2++ hms[s[P_2]]++	✓	match++;
0	1	2	*	*	P_2++ hms[s[P_2]]++	✓	match++;
0	2	3	*	*	P_2++ hms[s[P_2]]++	✓	match++;
0	3	4	*	*	P_2++ hms[s[P_2]]++	*	
0	4	4	*	*	P_2++ hms[s[P_2]]++	*	
0	5	4	*	*	P_2++ hms[s[P_2]]++	✓	match++;
0	6	5	✓	7	hms[s[P_1]]--; P_1++	*	
1	6	5	✓	6	hms[s[P_1]]--; P_1++	*	
2	6	5	✓	5	hms[s[P_1]]--; P_1++	✓	match--;

#Obs:

Add relevant char: ch

```
if( hms[ch] <= hmT[ch]){
    # ch is need q it's correct char, Inc count of match by 1
    match++;
}
```

Remove relevant char: ch

```
if( hmT[ch] > hms[ch]){
    # ch is removed q it's correct char, Dec count of match by 1
    match--;
}
```

```
int smallestString(string s, string T){

    unordered_map<char, int> hmt;
    for(int i=0; i<T.size(); i++){
        hmt[T[i]]++;  #Insert T[i] in hmt;
    }

    int match=0;
    int i=0, j=-1, ans=INT_MAX;
    unordered_map<char, int> hms;
    while(j<s.size()){
        if( match==T.size()){ #[i..j]
            ans= min(ans, j-i+1);
            hms[s[i]]--;
            if( hms[s[i]]<hmt[s[i]]){
                match--;
            }
            i++;
        }

        else{
            j++;
            if(j==s.size()){ break;}
            hms[s[j]]++;
            if( hms[s[j]]==hmt[s[j]]){
                match++;
            }
        }
    }

    return ans;
}
```