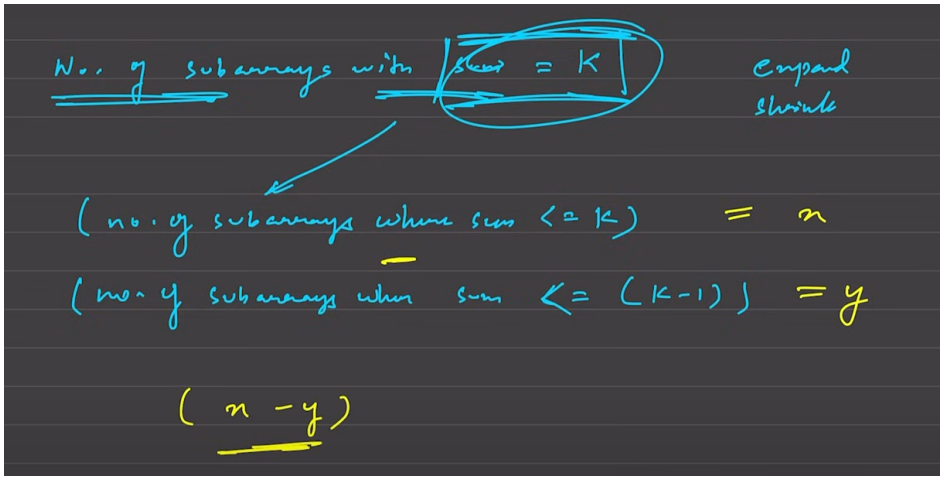
Two Pointer / Sliding Window

- Constant window : as we move we update left and right pointers also remove the outgoing element and add the incoming element
- Longest subarray / substring where {condition}

1. Brute force {find all subarrays}
2. Better {sliding window} : We increase the window when required and when the condition fails we try to shrink it back
3. Optimize the sliding window {only valid in case they are asking for length and not the substring itself} : we don't let the size of window go less than current max window length obtained rather we keep shifting to right and once the condition is re-satisfied we increase it now

- no of subarrays with {condition}



- Shortest / Minimum Window with {condition}

Given n cards, you can pick them from either start or end. Each cars has a point. Tell me max points achieved by picking k cards

```
int maxScore(vector<int>& cardPoints, int k) {
    int n = cardPoints.size();
    int total_sum = 0;

    // Calculate the sum of the first k elements
    for(int i = 0; i < k; i++) {
        total_sum += cardPoints[i];
    }

    int max_points = total_sum;

    // Start taking elements from the end
```

```
    // and removing from the start
    for(int i = 0; i < k; i++) {
        total_sum = total_sum - cardPoints[k - 1 - i]
                                    + cardPoints[n - 1 - i];
        max_points = max(max_points, total_sum);
    }

    return max_points;
}
```

Longest Substring Without Repeating Characters

```
int lengthOfLongestSubstring(string s) {
    int n = s.size();
    // Store the index of the characters
    unordered_map<char, int> mp;
    // Left pointer of the sliding window
    int l = 0;
    int ans = 0;

    for (int i = 0; i < n; i++) {
        // If the character is found in the map
        // and its index is within the current window
        if (mp.find(s[i]) != mp.end() && mp[s[i]] >= l) {
            // Update the left pointer to the
            // right of the last occurrence {KMP Algp}
            l = mp[s[i]] + 1;
        }
            else ans = max(ans, i - l + 1);
        // Update the character's latest index in the map
        mp[s[i]] = i;
        // This is important as previous location will
        // be overwritten by new one
    }

    return ans;
}
```

Given an array of 0’s and 1’s find the length of longest subarray with consecutive 1’s and you can change k 0’s to 1’s. {It is equivalent saying the current window can at max hold k zeroes}

```
int longestOnes(vector<int>& nums, int k) {
    int n = nums.size();
    int l = 0;  // Left pointer of the sliding window
    int ans = 0;
    int zeroCount = 0;
```

```
    for (int r = 0; r < n; r++) {
        if (nums[r] == 0) {
            zeroCount++;
        }

        // If zeroCount exceeds k, move the left pointer
        // to reduce zeroCount {SHRINKING}
        while (zeroCount > k) {
            if (nums[l] == 0) {
                zeroCount--;
            }
            l++;
        }

        ans = max(ans, r - l + 1);
    }
    return ans;
}
```

```
int longestOnes(vector<int>& nums, int k) {
    int n = nums.size();
    int l = 0;  // Left pointer of the sliding window
    int ans = 0;
    int zeroCount = 0;

    for (int r = 0; r < n; r++) {
        if (nums[r] == 0) {
            zeroCount++;
        }

        /*RATHER THAN SHRINKING THAT WE DID PREVIOUSLY
        WE SHRINK ONLY BY ONE,
        {TO MATCH THE CURRENT MAX LENGHT}
            AND THEN WE UPDATE ONLY WHEN THE CONDITION
            RE-SATISFIES AND THIS TIME WE MAY HAVE LENGTH
            TO BE SAME OR BIGGER*/
        if (zeroCount > k) {
            if (nums[l] == 0) {
                zeroCount--;
            }
            l++;
        }

        else ans = max(ans, r - l + 1);
    }
```

```
    return ans;
}
```

- basically when condition fails we are moving ahead with a constant window

---

Fruits into baskets : {array given with different type of fruit types of trees indexed from 0} but can harvest only two type of fruits {max harvests ?} ⇒ max length of subarray which has at max two unique numbers.

```
int totalFruit(vector<int>& fruits) {
  int n  = fruits.size();
  // TO KEEP TRACK OF FREQUENCY OF TYPE OF FRUIT
  unordered_map<int, int> mp;
  int l = 0;
  int ans = 0;

  for(int i = 0;i < n;i++) {
      mp[fruits[i]]++;
      if(mp.size() > 2) { DIRECT OPTIMIZATION
          mp[fruits[l]]--;
          if(mp[fruits[l]] == 0) {
              mp.erase(fruits[l]); IMP AS WE ARE CHECKING SIZE
          }
          l++;
      }
      else ans = max(ans, i - l + 1);
  }
  return ans;
}
```

Given an string made up of 3 characters, how many substrings are possible having all three characters

```
int numberOfSubstrings(string s) {
  int n = s.size();
  int ans = 0;
  int last_seen[3] = {-1, -1, -1};
  // {-1 REPRESENTS NOT YET SEEN SUCH CHARACTER}
  for(int i = 0;i < n;i++) {
      last_seen[s[i] - 'a'] = i;
      if(last_seen[0] != -1 && last_seen[1] != -1 && last_seen[2] != -1) {
          ans += 1 + min({last_seen[0], last_seen[1],last_seen[2]});
          // ONCE YOU HAVE FOUND THE LEAST POSSIBLE STRING,
          // YOU CAN JUST CONSIDER ADDING THE PREVIOUS ELEMENTS
          // WHICH WILL NOT MAKE A DIFFERENCE
      }
  }
```

```
  return ans;
}
```

- here we can omit the if condition, as min will return -1 and 1+(-1) the ans doesn't change.

Given a string, we can replace a character by any other atmost k times. Give me the length of longest substring of same char.

```
int characterReplacement(string s, int k) {

  int n = s.length();
  unordered_map<char, int> mp;
  int max_char = 0;
  int ans = 0;
  int l = 0;

  for(int r = 0;r < n;r++)
  {
      mp[s[r]]++; //count updation
      // Find the most frequent char in current window
      max_char = max(max_char, mp[s[r]]);
      // as by adding a char its frequency increases
      while((r-l+1) - max_char > k)
      {
          //its time we update the l as k cant be bared
          mp[s[l]]--;
          l++;
          //it is possible ki max_char has decreased
          max_char = 0;
          for(auto p : mp){
              max_char = max(max_char, p.second);
          }
      }
      ans = max(ans, (r-l+1));
  }
  return ans;
}
```

OPTIMISATION 1 :

```
int characterReplacement(string s, int k) {

  int n = s.length();
  unordered_map<char, int> mp;
  int max_char = 0;
  int ans = 0;
  int l = 0;
```

```
  for(int r = 0;r < n;r++)
  {
      mp[s[r]]++; //count updation
      // Find the most frequent char in current window
      max_char = max(max_char, mp[s[r]]);
      // as by adding a char its frequency increases
      if((r-l+1) - max_char > k)
      {
          //its time we update the l as k cant be bared
          mp[s[l]]--;
          l++;
          //it is possible ki max_char has decreased
          max_char = 0;
          for(auto p : mp){
              max_char = max(max_char, p.second);
          }
      }
      else ans = max(ans, (r-l+1));
  }
  return ans;
}
```

OPTIMISATION 2:

```
int characterReplacement(string s, int k) {

  int n = s.length();
  unordered_map<char, int> mp;
  int max_char = 0;
  int ans = 0;
  int l = 0;

  for(int r = 0;r < n;r++)
  {
      mp[s[r]]++; //count updation
      // Find the most frequent char in current window
      max_char = max(max_char, mp[s[r]]);
      // as by adding a char its frequency increases
      if((r-l+1) - max_char > k)
      {
          //its time we update the l as k cant be bared
          mp[s[l]]--;
          l++;
      }
      else ans = max(ans, (r-l+1));
  }
```

```
  return ans;
}
```

REASON WHY THIS WORKS : By doing optimization 1 we have maintained the window size to be constant. So ans will not be be updated until condition is met which will be met only is max_freq is greater than before : {Meaning there is no need decrease only to increase it again in the end}

Given an array consisting of 0's and 1's return number of subarrays which sum upto goal.

- when we asked number of rather than length we can not optimize it.

CATEGORY 3 :

```
int f(vector<int> &nums, int goal) {
  if(goal < 0) return 0; // as goal can't be -ve
  int l = 0;
  int count = 0;
  int sum = 0; // curr window sum
  int n = nums.size();

  for(int r = 0; r < n; r++) {
      sum += nums[r];
      while(sum > goal) {
      // THE REASON WHY WE ARE DOING THIS IS BEACUSE
      // WE ONLY SHRINK THE WINDOW WHEN THE CONDITION FAILS
          sum -= nums[l];
          l++;
      }
      count += (r - l + 1);
      // THE NUMBER OF SUBARRY WHICH SATISY THE CONDITION
      // WITH THE RIGHT MOST ELEMENT BEING nums[r]
  }
  return count;
}

int numSubarraysWithSum(vector<int>& nums, int goal) {
  return f(nums, goal) - f(nums, goal - 1);
}
```

- In all the previous problems, we were given a condition of at most k, but not = k.
- So whenever = k pops, it is category 3 problem.

Given an array with +ve integers, return the total count of subarray which have exactly k odd numbers.

- so even can be or not, but number of odd is fixed
- make even = 0 and odd = 1, apply BINARY SUM {done}
- How ? sum += / -= num[l/r] % 2;

Given an array of integers return the number of subarrays with **exactly three** different integers in it.

```
int atMostKDistinct(vector<int>& nums, int k) {
  int n = nums.size();
  int l = 0;
  int count = 0;
  unordered_map<int, int> mp;

  for(int r = 0; r < n; r++) {
      mp[nums[r]]++;

      while(mp.size() > k) {
          mp[nums[l]]--;
          if(mp[nums[l]] == 0) {
              mp.erase(nums[l]);
          }
          l++;
      }
      count += (r - l + 1);
      // ALLOWING FROM 1 - K {DIFFERENT INTEGERS}
  }
  return count;
}

int subarraysWithKDistinct(vector<int>& nums, int k) {
  return atMostKDistinct(nums, k) -
                              atMostKDistinct(nums, k - 1);
}
```

```
int subarraysWithKDistinct(vector<int>& nums, int k) {
  int n = nums.size();
  int l = 0;
  int count = 0;
  unordered_map<int, int> ind; // {a, b, c} problem -> min_index
  unordered_map<int, int> mp;  // frequency counter

  for(int r = 0;r < n;r++) {
      ind[nums[r]] = r;
      mp[nums[r]]++;

      while(mp.size() > k) {
          mp[nums[l]]--;
          if(mp[nums[l]] == 0) {
              mp.erase(nums[l]);
              ind.erase(nums[l]);
          }
          l++;
```

```
        }

        if(mp.size() == k) {
            int min_index = INT_MAX;
            for(auto p : ind) {
                min_index = min(min_index, p.second);
            }
            count += (min_index-l+1);
            // ALLOWING EXACTLY K DIFFERENT INTEGERS
        }
    }
    return count++;
}
```

Minimum Window : Given two strings s & t. we have to return minimum substring which has all elements of t {including duplicates}

- make a frequency counter of string t
- iterate over s will decreasing the frequency count if character matches, {also increasing the count} and when count reaches the size of t : condition met note down the left and right index
- Now try to decrease the window and increasing the frequency count and decreasing the count
- We are done!