Basics for these types of questions is that you should know about HashSet/set/map in the preferred language of your choice

Set in C++
HashSet in Java
Map in C++
HashMap in Java

Some problems you can practice to understand these data structures:
Jewels And Stones
Unique Number of Occurrences
Intersection of Two Arrays

Longest Consecutive Sequence

Intuition is for any number we will find that its previous number is present or not if yes would skip this number because we want to find the minimum number so that we would get the longest number.

Watch this video for a full understanding

C++

```
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
            unordered_set<int> hs;
            int i;
            for(i = 0; i < nums.size(); i++)
            {
                  hs.insert(nums[i]);
            }
            int seq = 0;
            for(i = 0; i < nums.size(); i++)
            {
                  if(hs.count(nums[i] - 1))continue;
                  int curr_seq = 1;
                  int curr_num = nums[i];
                  while(hs.count(curr_num + 1))
                  {
                        curr_seq++;
                        curr_num++;
                  }
                  seq = max(seq, curr_seq);
            }
            return seq;
    }
};
```

Java

```
class Solution {
    public int longestConsecutive(int[] nums) {
        HashSet<Integer> hs = new HashSet<Integer>();
            int i;
            for(i = 0; i < nums.length; i++)
            {
                  hs.add(nums[i]);
            }
            int seq = 0;
            for(i = 0; i < nums.length; i++)
            {
                  if(hs.contains(nums[i] - 1))continue;
                  int curr_seq = 1;
                  int curr_num = nums[i];
                  while(hs.contains(curr_num + 1))
                  {
                        curr_seq++;
                        curr_num++;
                  }
                  seq = Math.max(seq, curr_seq);
            }
            return seq;
    }
}
```

Python

```python
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        st = set(nums)
        mx = 0
        for num in nums:
            if num-1 not in st:
                tmp = 1
                while num+1 in st:
                    tmp += 1
                    num += 1
                mx = max(mx, tmp)

        return mx
```

Subarray Sum Equals K

C++

```
class Solution {
 public:
     int subarraySum(vector<int>&nums, int k) {
        int n = nums.size();
        vector<int> prefixSum(n,0);
        int i , j;
        prefixSum[0] = nums[0];
        for(i = 1; i < n; i++)
            prefixSum[i] = prefixSum[i-1] + nums[i];
        map<int,int>m1;
        m1[0] = 1;
        int cnt = 0;
        for(i = 0; i < n; i++)
        {
            if(m1[prefixSum[i] - k])
            {
                cnt += m1[prefixSum[i] - k];
            }
            m1[prefixSum[i]]++;
        }
        return cnt;
    }
};
```

Java

```
class Solution {
    public int subarraySum(int[] nums, int k) {
        int n = nums.length;
        int prefixSum[] = new int[n];
        int i , j;
        prefixSum[0] = nums[0];
        for(i = 1; i < n; i++)
            prefixSum[i] = prefixSum[i-1] + nums[i];
        HashMap<Integer,Integer> m1 = new HashMap<Integer,Integer>();
        m1.put(0 ,1);
        int cnt = 0;
        for(i = 0; i < n; i++)
        {
            if(m1.containsKey(prefixSum[i] - k))
            {
                cnt += m1.get(prefixSum[i] - k);
            }
            m1.put(prefixSum[i],m1.getOrDefault(prefixSum[i],0) + 1);
        }
        return cnt;
    }
}
```

Python

```
class Solution(object):
   def subarraySum(self, A, K):
    count = collections.Counter()
    count[0] = 1
    ans = su = 0
    for x in A:
        su += x
        ans += count[su-K]
        count[su] += 1
    return ans
```

Valid Anagram

An anagram means all the words of one string is present in other strings.
We would create an array of 26 sizes and we count the occurrences of each char in string s and
Then we would subtract the occurrence of each char from the same array. In the end, if all 26 char are 0 we got our anagram.

C++

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        vector<int>alpha(26 , 0);
        int i;
        for(i = 0; i < s.size() ; i++)
            alpha[s[i] - 'a']++;
        for(i = 0; i < t.size(); i++)
            alpha[t[i] - 'a']--;
        for(auto it : alpha)
        {
            if(it != 0)return false;
        }
        return true;
    }
};
```

Java

```
class Solution {
    public boolean isAnagram(String s, String t) {
        int alpha[] = new int[26];
        int i;
        for(i = 0; i < s.length() ; i++)
            alpha[s.charAt(i) - 'a']++;
        for(i = 0; i < t.length(); i++)
            alpha[t.charAt(i) - 'a']--;
        for(int j : alpha)
        {
            if(j != 0)return false;
        }
        return true;
    }
}
```

Python

```
class Solution(object):
    def isAnagram(self, s, t):
        dic1, dic2 = [0]*26, [0]*26
        for item in s:
            dic1[ord(item)-ord('a')] += 1
        for item in t:
            dic2[ord(item)-ord('a')] += 1
        return dic1 == dic2
```

Valid Sudoku

Recall how we filled the sudoku in backtracking class. But the difference is here we are given the same sudoku except the answer is given and we have to evaluate if it is valid or not.

C++

```
class Solution {
public:
    bool isValidSudoku(vector<vector<char>>& board) {
       vector<vector<bool>>
row(9,vector<bool>(9)),column(9,vector<bool>(9)),block(9,vector<bool>(9));

       for(int i = 0;i<9;i++){
           for(int j=0;j<9;j++){
                int c = board[i][j] - '1';
                if(board[i][j]=='.'){
                    continue;
                }
                if(row[i][c]||column[j][c]||block[i - i % 3 + j / 3][c]){
                    return false;
                }
                row[i][c] = column[j][c] = block[i - i % 3 + j / 3][c] = true;
           }
       }
       return true;
    }
}
```

Java

```
public class Solution {
    public boolean isValidSudoku(char[][] board) {

       boolean[][] row = new boolean[9][9];
       boolean[][] column = new boolean[9][9];
       boolean[][] block = new boolean[9][9];

       for(int i = 0;i<9;i++){
           for(int j=0;j<9;j++){
                int c = board[i][j] - '1';
                if(board[i][j]=='.'){
                    continue;
                }
                if(row[i][c]||column[j][c]||block[i - i % 3 + j / 3][c]){
                    return false;
                }
                row[i][c] = column[j][c] = block[i - i % 3 + j / 3][c] = true;
           }
       }
       return true;
    }
}
```

Python

```
class Solution(object):
    def isValidSudoku(self, board):
        big = set()
        for i in range(0,9):
            for j in range(0,9):
                if board[i][j]!='.':
                    cur = board[i][j]
                    if (i,cur) in big or (cur,j) in big or (i/3,j/3,cur) in big:
                        return False
                    big.add((i,cur))
                    big.add((cur,j))
                    big.add((i/3,j/3,cur))
        return True
```

Solve this from Gfg

Longest Substring Without Repeating Characters

This type of Of problem in fact whenever you see a substring there is a chance that it can be solved using hashmap or HashSet.
Start by inserting each character in a set and whenever you see a duplicate character start moving the left pointer to right.
Calculate the size of the set every time for max size.

C++

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
         int i = 0, j =0;
         int size = 0;
         set<char>s1;
         while(j < s.length())
         {
             if(!s1.count(s[j]))
             {
                 s1.insert(s[j++]);
                 if(size <s1.size())
                     size = s1.size();
             }
             else{
                 s1.erase(s[i++]);
             }
         }
        return size;
    }
};
```

Java

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
    int i = 0, j = 0, max = 0;
    Set<Character> set = new HashSet<>();

    while (j < s.length()) {
        if (!set.contains(s.charAt(j))) {
            set.add(s.charAt(j++));
            max = Math.max(max, set.size());
        } else {
            set.remove(s.charAt(i++));
        }
    }

    return max;
}
}
```

Python

```
class Solution:
    def lengthOfLongestSubstring(self, s):
        m = set()
        res = 0
        left = 0
        i = 0
        while i < len(s):
            if s[i] not in m:
                m.add(s[i])
                res = max(res, len(m))
                i+=1
            else:
                m.remove(s[left])
                left+=1
        return res
```

Minimum Window Substring

This problem is same as previous one but the difference is that you have to find the subtring which is already present and it should minimum . First you will find the substring and then start chopping it up to make it shorter.
Note: You can use map to store the characters but I have used ASCII map of chars here.

C++

```
class Solution {
public:
    string minWindow(string s, string t) {
        vector<int>v(128, 0);
        for(auto c:t) v[c]++;
        int start = 0, end = 0, counter = t.size();
        int minStart = 0, minLen = INT_MAX;
        int len = s.size();
        while(end < len){
            if(v[s[end]] > 0)  counter--;
            v[s[end]]--;
            end++;
            while(counter == 0){
                if(end-start < minLen){
                    minStart = start;
                    minLen = end-start;
                }
                v[s[start]]++;
                if(v[s[start]]>0)  counter++;
                start++;
            }
        }
        if(minLen != INT_MAX)
            return s.substr(minStart, minLen);
        return "";
    }
};
```

Java

```
class Solution {
   public String minWindow(String s, String t) {
    String result = "";
    if(s==""||t.length()>s.length())
        return result;
    int[] map = new int[128];
    int start = 0;
    int minStart = 0;
    int end = 0;
    int count = t.length();
    int minLength = Integer.MAX_VALUE;
    for(char temp:t.toCharArray()){
        map[temp]++;
    }
    while(end < s.length()){
        if(map[s.charAt(end)]>0)
            count--;
        map[s.charAt(end)]--;
        end++;
        while(count == 0){
                  if (end - start < minLength) {
                        minStart = start;
                        minLength = end - start;
                  }
                  map[s.charAt(start)]++;
                  if (map[s.charAt(start)] > 0)
                        count++;
                  start++;
        }
    }
    return (minLength==Integer.MAX_VALUE)?"":s.substring(minStart, minStart+minLength);
}
}
```

Python

```
class Solution(object):
    def minWindow(self, s, t):
        n, counter = len(s), len(t)
        begin, end, head = 0, 0, 0
        min_len = n+1
        dic = dict()
        for ch in t:
            dic[ch] = dic.get(ch, 0)+1
        while end < n:
            if s[end] in dic:
                if dic[s[end]] > 0:
                    counter -= 1
                dic[s[end]] -= 1

            end += 1
            while counter == 0:
                if end - begin < min_len:
                    min_len = end-begin
                    head = begin
                if s[begin] in dic:
                    dic[s[begin]] += 1
                    if dic[s[begin]] > 0:
                        counter += 1
                begin += 1
        if min_len == n+1:
            return ""
        return s[head: head+min_len]
```

Palindrome Pairs

In this problem we will derive the inspiration from pallindrome division problem from backtracking . Since we know that we will break the string in two parts and check if prefix is pallindrome then we perform the operation on second string but here same concept is applied instead of dividing the string further we will lookup the reverse Of splitted part and check if there is any string .

For a very awesome explanation read this post

C++

```
class Solution {
public:
     bool isPallindrome(string s)
        {
            int i , j;
            i = 0, j = s.length() - 1;
            while(i <= j)
            {
                if(s[i] != s[j])return false;
                i++; j --;
            }
            return true;
        }
    vector<vector<int>> palindromePairs(vector<string>& words) {
        vector<vector<int>>res;
        map<string,int>m1;
        int i;
        for(i = 0; i < words.size(); i++)
            m1[words[i]] = i;

        int j;

        for(i = 0; i < words.size(); i++)
        {
            for(j = 0; j <= words[i].size(); j++)
            {
                string str1 = words[i].substr(0 , j);
                string str2 = words[i].substr(j);
                if(isPallindrome(str1))
                {
                    string rev(str2.rbegin(),str2.rend());
                    if(m1.find(rev)!=m1.end() and m1[rev]!=i)
                    {
                        res.push_back({m1[rev],i});
                    }
                }
                if(isPallindrome(str2))
                {
                     string rev(str1.rbegin(),str1.rend());
                    if(m1.find(rev)!=m1.end() and m1[rev]!=i and !str2.empty())
                    {
                        res.push_back({i,m1[rev]});
                    }

                }
            }
        }
        return res;

    }
};
```

Java

```
class Solution {
   public List<List<Integer>> palindromePairs(String[] words) {
    List<List<Integer>> ret = new ArrayList<>();
    if (words == null || words.length < 2) return ret;
    Map<String, Integer> map = new HashMap<String, Integer>();
    for (int i=0; i < words.length; i++) map.put(words[i], i);
    for (int i=0; i < words.length; i++) {
        for (int j = 0; j< = words[i].length(); j++)
        {
            String str1 = words[i].substring(0, j);
            String str2 = words[i].substring(j);
            if (isPalindrome(str1)) {
                String str2rvs = new StringBuilder(str2).reverse().toString();
                if (map.containsKey(str2rvs) && map.get(str2rvs) != i) {
                    List<Integer> list = new ArrayList<Integer>();
                    list.add(map.get(str2rvs));
                    list.add(i);
                    ret.add(list);
                }
            }
            if (isPalindrome(str2)) {
                String str1rvs = new StringBuilder(str1).reverse().toString();
                if (map.containsKey(str1rvs) && map.get(str1rvs) != i &&
str2.length()!=0) {
                    List<Integer> list = new ArrayList<Integer>();
                    list.add(i);
                    list.add(map.get(str1rvs));
                    ret.add(list);
                  }
            }
        }
    }
    return ret;
}

private boolean isPalindrome(String str) {
    int left = 0;
    int right = str.length() - 1;
    while (left <= right) {
        if (str.charAt(left++) !=  str.charAt(right--)) return false;
    }
    return true;
}
}
```

Python

```
class Solution:
    def palindromePairs(self, words: List[str]) -> List[List[int]]:
        res = []
        length = len(words)
        if length == 0:
            return res
        d = {}
        for i in range(length):
            d[words[i]] = i

        for i in range(length):
            for j in range(len(words[i])+1):
                s1 = words[i][0:j]
                s2 = words[i][j:]
                if s1 == s1[::-1]:

                    reversedS2 = s2[::-1]
                    if reversedS2 in d and d[reversedS2] != i:
                        res.append([d[reversedS2], i])
                if s2 == s2[::-1] and len(s2) !=0:
                    reversedS1 = s1[::-1]
                    if reversedS1 in d and d[reversedS1] != i:
                        res.append([i,d[reversedS1]])
        return res
```