1. **Find whether an array is subset of another array:  
     
   2. Union and intersection of linked list**

**3.Given an array A[] and a number x, check for pair in A[] with sum as x.**

(sort, and two pointer problem)

**4.Check If a given array contains duplicate elements within k distance from each other.**I can do that.

**5.Pair With Given Product:**I can do that.

**6.Finding the missing element of a given range from a given array:**Hash table might not produce the best solution in this case.

If the array is not sorted, we need to sort it first. Then find the lower bound of the starting point of the given range in the given array. And then normal traversal.

**7.Given An Array In Pairs, Find all The Symmetric Pairs In It.**

1. **Group multiple occurrence of array elements, ordered the first occurrence.**

1) Create an empty **Binary Search Tree (BST**). Every BST node is going to contain an array element and its count.

2) Traverse the input array and do following for every element.

……..a) If element is not present in BST, then insert it with count as 0.

……..b) If element is present, then increment count in corresponding BST node.

3) Traverse the array again and do following for every element.

…….. If element is present in BST, then do following

……….a) Get its count and print the element ‘count’ times.

……….b) Delete the element from BST.

**The same approach can be followed using hashing.**

**9.Convert The Array Into Its Reduced Form:**I can do it.

You need to sort the array. Then do it.

**10.Return Maximum Occurring Character In An Input String:**

**11.Find the first repeating element in an array:**

**12.Print all the distinct elements in an array:**

**13.Find all permuted row of a given row in matrix:**

**14.Count distinct elements in every window of size k:**

**15.Find Itinerary from a given list of tickets:**

(Important step is to create a reverse map)

Given a list of tickets, find itinerary in order using the given list.

**Input:**

"Chennai" -> "Bangalore"

"Bombay" -> "Delhi"

"Goa" -> "Chennai"

"Delhi" -> "Goa"

**Output:**

Bombay->Delhi, Delhi->Goa, Goa->Chennai, Chennai->Banglore,

1) Create a HashMap of given pair of tickets. Let the created hashMap be 'dataset'. Every entry of 'dataset' is of the form "from->to" like "Chennai" -> "Banglore"

2) Find the starting point of itinerary.

a) Create a reverse HashMap. Let the reverse be 'reverseMap' Entries of 'reverseMap' are of the form "to->form". Following is 'reverseMap' for above example.

“Banglore"-> "Chennai"

"Delhi" -> "Bombay"

"Chennai" -> "Goa"

"Goa" -> "Delhi"

b) Traverse 'dataset'. For every key of dataset, check if it is there in 'reverseMap'. If a key is not present, then we found the starting point. In the above example, "Bombay" is starting point.

**Start from above found starting point and traverse the 'dataset' to print itinerary.**

It’s all about finding out the itinerary.

Given a dictionary that contains mapping of employee and his manager as a number of (employee, manager) pairs like below.

{ "A", "C" },

{ "B", "C" },

{ "C", "F" },

{ "D", "E" },

{ "E", "F" },

{ "F", "F" }

1. Create a reverse map with Manager->DirectReportingEmployee combination. Off-course employee will be multiple so Value in Map is List of Strings.

"C" --> "A", "B",

"E" --> "D"

"F" --> "C", "E", "F"

2. Now use the given employee-manager map to iterate and at the same time use newly reverse map to find the count of employees under manager.

Let the map created in step 2 be 'mngrEmpMap' Do following for every employee 'emp'.

a) If 'emp' is not present in 'mngrEmpMap' Count under 'emp' is 0 [Nobody reports to 'emp']

b) If 'emp' is present in 'mngrEmpMap'

Use the list of direct reports from map 'mngrEmpMap' and recursively calculate number of total employees under 'emp'.

A trick in step 2.b is to use memorization(Dynamic programming) while finding number of employees under a manager so that we don’t need to find number of employees again for any of the employees. In the below code populateResultUtil() is the recursive function that uses memoization to avoid re-computation of same results.

**16.Check if an array can be divided into pairs whose sum is divisible by k:**

I can do that.

**17.Find four elements a, b, c and d in an array such that a+b = c+d:**

**18.Find the largest subarray with 0 sum:**

1. **Longest Consecutive Subsequence:**

1) Create an empty hash.

2) Insert all array elements to hash.

3) Do following for every element arr[i]

....a) Check if this element is the starting point of a subsequence. To check this, we simply look for

arr[i] - 1 in hash, if not found, then this is the first element a subsequence.

If this element is a first element, then count number of elements in the consecutive starting with this element.

If count is more than current res, then update res.

1. **Length of the largest subarray with contiguous elements:**

we checked whether maximum value minus minimum value is equal to ending index minus starting index or not. Since duplicate elements are allowed, we also need to check if the subarray contains duplicate elements or not. For example, the array {12, 14, 12} follows the first property, but numbers in it are not contiguous elements.

To check duplicate elements in a subarray, we create a hash set for every subarray and if we find an element already in hash, we don’t consider the current subarray.

**/\* CPP program to find length of the largest**

**subarray which has all contiguous elements \*/**

**#include<bits/stdc++.h>**

**using namespace std;**

**// This function prints all distinct elements**

**int findLength(int arr[], int n)**

**{**

**int max\_len = 1; // Inialize result**

**// One by one fix the starting points**

**for (int i=0; i<n-1; i++)**

**{**

**// Create an empty hash set and**

**// add i'th element to it.**

**set<int> myset;**

**myset.insert(arr[i]);**

**// Initialize max and min in**

**// current subarray**

**int mn = arr[i], mx = arr[i];**

**// One by one fix ending points**

**for (int j=i+1; j<n; j++)**

**{**

**// If current element is already**

**// in hash set, then this subarray**

**// cannot contain contiguous elements**

**if (myset.find(arr[j]) != myset.end())**

**break;**

**// Else add current element to hash**

**// set and update min, max if required.**

**myset.insert(arr[j]);**

**mn = min(mn, arr[j]);**

**mx = max(mx, arr[j]);**

**// We have already checked for**

**// duplicates, now check for other**

**// property and update max\_len**

**// if needed**

**if (mx - mn == j - i)**

**max\_len = max(max\_len, mx - mn + 1);**

**}**

**}**

**return max\_len; // Return result**

**}**

**// Driver method to test above method**

**int main ()**

**{**

**int arr[] = {10, 12, 12, 10, 10, 11, 10};**

**int n = sizeof(arr) / sizeof(arr[0]);**

**cout << "Length of the longest contiguous"**

**<< " subarray is " << findLength(arr, n);**

**}**

Now, why set is better in this case? To keep track of the elements? To find the largest subarray with contiguous element?

Because, we only need to know about the count of unique element or we only need to check the uniqueness of the element.

**21.Design a data structure that supports insert, delete, search and getRandom in constant time:**

We can use hashing to support first 3 operations in Θ(1) time. How to do the 4th operation? The idea is to use a resizable array (ArrayList in Java, vector in C) together with hashing. Resizable arrays support insert in Θ(1) amortized time complexity. To implement getRandom(), we can simply pick a random number from 0 to size-1 (size is number of current elements) and return the element at that index. The hash map stores array values as keys and array indexes as values.

Following are detailed operations.

**insert(x)**

1) Check if x is already present by doing a hash map lookup.

2) If not present, then insert it at the end of the array.

3) Add in hash table also, x is added as key and last array index as index.

**remove(x)**

1) Check if x is present by doing a hash map lookup.

2) If present, then find its index and remove it from hash map.

3) Swap the last element with this element in array and remove the last element.

**Swapping is done because the last element can be removed in O(1) time.**

4) **Update index of last element in hash map. (since, swapping was done)**

**getRandom()**

1) Generate a random number from 0 to last index.

2) Return the array element at the randomly generated index.

**search(x)**

Do a lookup for x in hash map.

**22.Find If There Is A Subarray With 0 Sum:**

**23.Group Shifted String:**

I can do that.

**24.Check for Palindrome after every character replacement Query:**

I can do that.

**25.Largest subarray with equal number of 0s and 1s**Following is a solution that uses O(n) extra space and solves the problem in O(n) time complexity.

Let input array be arr[] of size n and maxsize be the size of output subarray.

1) Consider all 0 values as -1. The problem now reduces to find out the maximum length subarray with sum = 0.

2) Create a temporary array sumleft[] of size n. Store the sum of all elements from arr[0] to arr[i] in sumleft[i]. This can be done in O(n) time.

3) There are two cases, the output subarray may start from 0th index or may start from some other index. We will return the max of the values obtained by two cases.

4) To find the maximum length subarray starting from 0th index, scan the sumleft[] and find the maximum i where sumleft[i] = 0.

5) Now, we need to find the subarray where subarray sum is 0 and start index is not 0. This problem is equivalent to finding two indexes i & j in sumleft[] such that sumleft[i] = sumleft[j] and j-i is maximum. To solve this, we can create a hash table with size = max-min+1 where min is the minimum value in the sumleft[] and max is the maximum value in the sumleft[]. The idea is to hash the leftmost occurrences of all different values in sumleft[]. The size of hash is chosen as max-min+1 because there can be these many different possible values in sumleft[]. Initialize all values in hash as -1

6) To fill and use hash[], traverse sumleft[] from 0 to n-1. If a value is not present in hash[], then store its index in hash. If the value is present, then calculate the difference of current index of sumleft[] and previously stored value in hash[]. If this difference is more than maxsize, then update the maxsize.

7) To handle corner cases (all 1s and all 0s), we initialize maxsize as -1. If the maxsize remains -1, then print there is no such subarray.

**26.Palindrome Substring Queries**

The idea is similar to Rabin Karp string matching. We use string hashing. What we do is that we calculate cumulative hash values of the string in the original string as well as the reversed string in two arrays- prefix[] and suffix[].

How to calculate the cumulative hash values ?

Suppose our string is str[], then the cumulative hash function to fill our prefix[] array used is-

prefix[0] = 0

prefix[i] = str[0] + str[1] \* 101 + str[2] \* 1012 +

...... + str[i-1] \* 101i-1

For example, take the string- “abaaabxyaba”

prefix[0] = 0

prefix[1] = 97 (ASCII Value of ‘a’ is 97)

prefix[2] = 97 + 98 \* 101

prefix[3] = 97 + 98 \* 101 + 97 \* 1012

...........................

...........................

prefix[11] = 97 + 98 \* 101 + 97 \* 1012 +

........+ 97 \* 10110

Now the reason to store in that way is that we can easily find the hash value of any substring in O(1) time using-

hash(L, R) = prefix[R+1] – prefix[L]

For example, hash (1, 5) = hash (“baaab”) = prefix[6] – prefix[1] = 98 \* 101 + 97 \* 1012 + 97 \* 1013 + 97 \* 1014 + 98 \* 1015 = 1040184646587 [We will use this weird value later to explain what’s happening].

Similar to this we will fill our suffix[] array as-

suffix[0] = 0

suffix[i] = str[n-1] + str[n-2] \* 101 + str[n-3] \* 1012 +

...... + str[n-i] \* 101i-1

For example, take the string- “abaaabxyaba”

suffix[0] = 0

suffix[1] = 97 (ASCII Value of ‘a’ is 97)

suffix[2] = 97 + 98 \* 101

suffix[3] = 97 + 98 \* 101 + 97 \* 1012

...........................

...........................

suffix[11] = 97 + 98 \* 101 + 97 \* 1012 + ........+ 97 \* 10110

Now the reason to store in that way is that we can easily find the reverse hash value of any substring in O(1) time using

reverse\_hash(L, R) = hash (R, L) = suffix[n-L] – suffix[n-R-1]

**27.Find Smallest Range containing Elements From K List:**

I can do that. Minheap is required.

**28.Advantages of BST over Hash Table:**

Hash Table supports following operations in Θ(1) time.

1) Search

2) Insert

3) Delete

The time complexity of above operations in a self-balancing Binary Search Tree (BST) (like Red-Black Tree, AVL Tree, Splay Tree, etc) is O(Logn).

So Hash Table seems to beating BST in all common operations. When should we prefer BST over Hash

Tables, what are advantages. Following are some important points in favor of BSTs.

We can get all keys in sorted order by just doing Inorder Traversal of BST. This is not a natural operation in Hash Tables and requires extra efforts.

Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with BSTs. Like sorting, these operations are not a natural operation with Hash Tables.

BSTs are easy to implement compared to hashing, we can easily implement our own customized BST. To implement Hashing, we generally rely on libraries provided by programming languages.

With Self-Balancing BSTs, all operations are guaranteed to work in O(Logn) time. But with Hashing, Θ(1) is average time and some particular operations may be costly, especially when table resizing happens.

**29.Linear Probing hashing Function:**

h(k,i)=(h’(k)+i)mod m  
  
where 0<=i<m

**Linear Probing Problem:**Long runs of occupied slot build up, increasing the average time of searching, clusters arise because an empty slot preceded by i full slots gets filled next with probability (i+1)/m. Long run of occupied slots tend to get longer, and the average search time increases more.

**Quadratic Probing Hashing Function:**

h(k,i)=(h’(k)+c1i+c2i2)mod m

Where, c1 and c2 are positive auxiliary constants and i=0,1,…m-1.

**Quadratic Probing Problem:**

If two keys have the same initial probe position, then the probe sequence are same. This property leads use to milder form of clustering.

**Double Hashing:**

h(k,i)=(h1(k)+h2(k))mod m