Data Structure

8. Hashing

Handwritten by pankaj kumar

Hashing

① Hashing Introduction:-

- Hashing is a method of storing and retrieving data from a database efficiently.

Eg:- For example we want to store employee record keyed using phone numbers. then we can use following data structure.

(i) An array of phone numbers & records
(ii) A linked list of phone number & records.
(iii) A balanced binary search tree with phone no. as keys.
(iv) A direct access table.

(i) Using an Array & linked list

searching of record:- $O(n)$ or $O(\log n)$]
Insert & delete:- $O(n)$] (worst) Costly operation

(iii) balanced binary search tree:-

moderate search, insert & delete:- within $O(\log n)$] (moderate)

(iv) direct Access table:-

- making a big array and use phone numbers as index in the array
- we can do all operation in:- $O(1)$
- but this solution has many practical limitation. extra space required is huged, if phone. no. digit is 'n' then we need $O(m * 10^n)$ space where m is size of pointer to the record.

Another problem a programming language may not store 'n' digits.

⊛ Hashing:—

Hashing is an improvement over Direct Access Table. The idea is to use a hash function that converts a given phone number or any other key to a smaller number and uses the small number as an index in a table called a hash table.

⊛ **Hash Function**:→ A function that converts a given big number to a small practical integer value. The mapped integer value is used as an index in the hash table.

- A good hash function should have following properties:-
  (i) It should be efficiently computable
  (ii) It should uniformly distribute the keys
  (Each table position be equally likely for each key).

Hash table:- An array that stores pointers to record corresponding to a given (phone number or key) An entry in hash table is NIL if no existing (phone number or key) has hash function value equal to the index for the entry.

② Collision Handling:-

- since a hash function gets us small number for a big key. there is a possibility that two keys result in the same value.
- The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

Collision handling technique:-

(i) Chaining:- The idea is to make each cell of the hash table point to a linked list of records that have the same function value. chaining is simple but it required additional memory outsides the table.

(ii) Open Addressing:- in open addressing, all element are stored in the hash table itself. Each table entry contains either a record or a NIL. when searching for an element, we one by one examine the table slots until the desired element is found or it is clear that the element is not present in the table.

③ Chaining:-

go to linked list section Page no. (160-184)

Q. Design a hashset without using builtin hashtable

Q. Design a Hashmap without using any builtin hashtable

④ Open Addressing :-

- like seperate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note:- we can increase table size by copying old data if needed)

⊛ Important Operation :-

(i) insert(k): keep probing until an empty slot is found. once an empty slot is found, insert k.

(ii) Search(k): keep probing until the slots key doesn't become equal to k or an empty slot is reached.

(iii) Delete (k): Delete operation is interesting. if we simply delete a key, then the search may fail. So slots of the deleted key are marked specially as "deleted".

Note:- insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

⊛ Open Addressing is done in the following ways:-

① Linear Probing:-

if slot hash(x) % S is full, then we try (hash(x) + 1) % S
if (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
if (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
- -
- -
- - - - - - - - - - - - - - - - and so on

Ex:- (50, 700, 76, 85, 92, 73, 101)

let $\boxed{S=7}$ 50 % 7 = 1 (index)

700 % 7 = 0 (index) | 76 % 7 = 6

85 % 7 = 1 (collision) ∴ 1+1=2 (idx)

92 % 7 = 1 (collision), 1+1=2 (collision) ∴ 3

73 % 7 = 3 (collision) ∴ 3+1=4

101 % 7 = 3, ∴ 3+1=4, ∴ 4+1=5

0	700
1	50
2	85
3	92
4	73
5	101
6	76

Clustering: The main problem with linear probing is clustering, many consecutive elements from groups and it starts taking time to find a free slot or to search an element

② Quadratic Probing :- We look for (i^2 th) slot in (ith) iteration.

if (hash(x) % S) is Full, then we try (hash(x) + 1*1) % S.
if ((hash (x) + 1*1) % S) is Full, then we try (hash (x) + 2*2) % S.
if ((hash(x) + 2*2) % S) is Full ; then we try (hash(x) + 3*3) % S.
- -
- - - - - - - - - - - - - - - - - - - and so on.

③ Double Hashing :- We use another (hash2(x)) function and look for i* hash2(x) slot in ith iteration.

- if slot [hash (x) % S] is Full, then we try (hash(x) + 1*hash2(x)) % S
- " " [(hash(x) + 1* hash2(x)) % S] " " " " [(hash(x) + 2*hash2(x)) % S]
- " " [(hash (x) + 2*hash2(x)) % S] " " " " [(hash(x) + 3* hash2(x)) % S].

- and so on.

④ Comparison of above three
- linear probing = best cache performance, but suffer from clustering and easy to compute, ~~best cache performance~~
- Qudratic probing = lies b/w the two in terms of cache & clustering.
- Double Hashing = poor cache performance but no clustering. Double hashing require more computation time.

④ Performance of Open Addressing :-

m = Number of slots in hash table | n = number of key to be inserted

Load Factor $\alpha = n/m$ (<1)

Expected time to search/insert/delete $< 1/(1-\alpha)$

So, search, insert/delete take $1/(1-\alpha)$ time.

(5) Hashing in C++ using STL

- Hashing can be implemented using different containers present in STL as per the requirement
  - (i) **Set**
  - (ii) unordered-set
  - (iii) map
  - (iv) unordered_map.

Note:- [all the topics covered in C++ STL page no. 98-100]

(6) Two Sum

- Given an array of integers nums and integer target, return the indices of the two numbers such that they add up to the target.

Ex nums = [2,7,11,15], target = 9, o/p: [0,1]

Approach 1 :- using two loop : O(n²)

Approach 2 :- Optimized:(Hash-table)

- go through all the element take current element and check its other pair are present or not in hash-table, if not present then insert it into hash-table and go through next elements

Code:-

```
vector<int> twoSum (vector <int>& nums, int target){
        vector<int> ans;
        unordered_map <int, int> mpp;
        for (int i=0; i<nums.size(); i++){
            if (mpp.find (target-nums[i]) != mpp.end()){
                ans.push_back (mpp[target-nums[i]]);
                ans.push_back (i);
                return ans;
            }
            mpp [nums[i]] = i;
        }
        return ans;
}
```

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

⑦ 4 sum

Given an array nums of n integers and an integer target, are there elements a, b, c and d in nums such that a+b+c+d = target? Find all unique quadruplets in the array which gives the sum of target.

Ex:- nums = [1, 0, -1, 0, -2, 2], and target = 0.

O/P [
[-1, 0, 0, 1],
[-2, -1, 1, 2],
[-2, 0, 0, 2]
]

Approach 1:- Sort → 3 ptr + Binary Search → Set (for unique)

```
Ex:-  4 3 3 4 4 2 1 2 1 1
      ↳ 1 1 1 2 2 3 3 4 4 4        target = 9
        i j k └──────────────┘
        →
                 9-3 = ⑥ → BS to find 6
 if not found increment k,

        1 1 1 2 2 3 3 4 4 4
        i j k k └────────┘
                 9-4 = 5 → B.S to find 5
```

Loop
i = 0
j = i+1
k = j+1

and so on Find all quadruplets & push into hashset.

TC: $O(N^3 \log N) + O(N \log M)$ | **SC:** $O(1)$

Approach 2 **optimized**

① sort

```
② [1 1 1 2 2 3 3 4 4 4]          target = 9
   i j left            right

   [9-(i+j)] = (7 in (left to right))

   [1 1 [1 2 2 3 3 4 4 4]
         left          right
```

∵ 1+4 < 7 (so we have to increase means include large value i.e increase left)

i j left left left left right
1 1 1 2 2 3 3 4 4 4

2+4 < 7 (increase left)

Now 3+4=7 ∴ one quadruple (1,1,3,4)
put it into hash set.

(1,1,3,4)

hashset

repeat this if(! left >= right) or if(left<right)

Now increase j

i j left right
[1 1 1 2 2 3 3 4 4 4]

i=0
j = i+1
left & right

Now Repeat step ② [Find (target-(i+j)) in[left, right]]

TC: $O(n^3) + O(n\log n)$
SC: O(1)

Note:- don't need to use Hashset if we skip duplicate/repeated value

code:-

```
vector<vector<int>> FourSum (vector<int>& nums, int target){
    vector <vector<int>> total;
    int n=nums.size();
    if (n<4) return total;
    sort (nums.begin(), nums.end());
    for(int i=0; i<n-3; i++){
          if(i>0 && nums[i]==nums[i-1]) continue;
          if (4*nums[i]>target) break;
          For(int j=i+1; j<n-2; j++){
                if (j>i+1 && nums[j]==nums[j-1]) continue;
                if (nums[i] + 3* nums[j] >target) break;
                int left=j+1, right=n-1;
                while(left<right){
                      int sum=nums[left]+nums[right]+nums[i]+nums
                      if (sum<target) left++;
                      elseif (sum>target) right--;
                      else{
                            total.push_back(vector<int>{nums[i], nums[j],
                                              nums[left], nums[right]});
```

```
            do{left++}while(nums[left]==nums[left-1]&& left<right);
            do{right--}while(nums[right]==nums[right+1]&&left<right);
        }
    }
}
return total;
}
```

⑧ Longest Consecutive Sequence in an array

you are given an array of 'N' integers. you need to find the length of the longest sequence which contains the consecutive elements.

Ex:- ① [100, 200, 1, 3, 2, 4] , O/P: 4 (1, 2, 3, 4)
② [3, 8, 5, 7, 6] , O/P: 4 (5, 6, 7, 8)

Approach 1: (Brute Force)

sort → count longest consecutive sequence | TC: O(N) + O(NlogN) = O(NlogN)

Approach 2: (Optimal Search) (using hashset)

First push all elements to the hashset. Then run a for loop on given array and check for every element 'x' that x+1, x+2, x+3, ... are present or not in hashset & find count them. Using this we can calculate the length of the consecutive subsequence.

Code :-

```
int longestConsecutive(vector<int> & nums){
        set<int> hashSet;
        for(int num: nums) hashSet.insert(num);
        int longestStreak=0;
        for(int num: nums){
          if(!hashSet.count(num-1)){
               int currNum = num;
               int currStreak=1;
               while(hashset.count(currNum+1)){
                     currNum++; currStreak++;
```

checking that num is starting number of sequence.

```
            longestStreak = max (longestStream, currStreak);
        }
    }
    return longestStreak;            / TC : O(N)
}
```

⑨ length of longest subarray with zero sum:

(Ex:1) N = 6, arr[] = {9, -3, 3, -1, 6, -5}, O/P = 5

subarray sum with zero are,

{-3, 3}, {-1, 6, -5}, {-3, 3, -1, 6, -5}.

longest is = {-3, 3, -1, 6, -5}, Length = 5

(Ex:2) N = 5, arr[] = {1, 3, -5, 6, -2}, O/P = 0

there is no subarray that sums to zero.

Approach 1: Find longest subarray with sum=0. by iterating two for loop:-

```
for (i=0 to i<n)
    for (j=i to j<n)
        sum+ = arr[j];
        if (sum==0) calculate length (j-i+1)
```

TC : O(N²)

Approach 2:

We'll store the prefix sum of every element, and if we observe that 2 elements have same prefix sum, we can conclude that the 2nd part of this subarray sums to zero

i.e subarray[i, k] = subarray (i, j) & i<k<j then, (k+1, j) sum is zero

Ex:- arr = [9, -3, 3, -1, 6, -5] (indices 0 1 2 3 4 5)

prefixsum = [9, 6, 9, 8, 14, 9]

i.e [1, 2] sum is zero   i.e (3, 5) & (1, 5) have sum is zero

and here length of (1, 5) is maxm so. this will be answer,

- for this we can use Hashmap & store index with prefix sum & then iterate the array with adding element and check if it is of any prefix sum from hashmap. if we find zero sum then it will also be the satisfied condi

Code:-

```
int maxLen(int A[], int n){
    unordered_map<int, int> mpp;
    int maxi = 0;
    int sum = 0;
    for(int i=0; i<n; i++){
        sum += A[i];
        if(sum==0){
            maxi = i+1;
        }
        else{
            if (mpp.find(sum) != mpp.end()){
                maxi = max(maxi, i-mpp[sum]);
            }
            else{
                mpp[sum] = i;
            }                          // O(n) : TC
        }
    }
    return maxi;
}
```

⑩ Count the number of subarrays with given xor k.

Given an array of intergear A, and an integer B. Find the total number of subarrays having bitwise XOR of all element equal to B.

Ex:- A = [4, 2, 2, 6, 4], B=6

O/P: 4

explantion subarray having xor of all element '6' are 4.

[4, 2], [4, 2, 2, 6, 4], [2, 2, 6], [6]

Approach 1 :- [Brute Force] // $O(N^2)$

generate all possible subarray for each subarray get respective XOR and check if it is equal to B then increment answer count to 1. in the end we will get count of subarray with XOR equal to B.

Approach 2: [prefix xor and map] :-

- prefix xor means xor from index 0 to that element.
- create a array of prefix xor for all element.

$$prefix\_xor[i] = XOR(a[0], a[1], a[2], ... a[i])$$

Note:- observation :-

$$P = xor(a[0], a[1], a[2], ... a[q], a[q+1], ... a[p])$$

$$Q = xor[a[0], a[1], a[2], ... a[q])$$

$$\therefore P \wedge Q = xor[a[q+1], ... a[p]) = M.$$

so, we understand that from XOR array when we XOR two element at different indices we get the xor of the element (in the original array) b/w those two indices.

Note:-

let's say we did XOR (P, B) and we get Q (B is given integer) this means that the subarray b/w q and p having xor = B. ex:-

$$P \wedge B = Q$$

$$\Rightarrow P \wedge B \wedge P = Q \wedge P$$

$$\Rightarrow B = Q \wedge P$$

Step1:- this means we will iterate over the array and XOR every element and check if it is B then count will be incremented by 1.

Step2:- we will also store every prefix_xor with frequency in map.

Step3:- xor the current prefix with B (i.e $P \wedge B$, where, $P = a[0] \wedge a[1] ... \wedge a[p]$) if $P \wedge B = Q$ is present in map. it means from Q+1 to P there is sequence with XOR B. now increment the count by frequency of Q.

TC:- O(N)
SC:- O(N)

Code:-

```
#include<bits/stdc++.h>
using namespace std;
class Solution{
    public:
    int solve(vector<int> &A, int B){
        unordered_map<int,int> visited;
        int cpx=0;
        long long c=0;
        For(int i=0;i<A.size();i++){
            cpx^=A[i];
            if(cpx==B) c++;
            int h=cpx^B;
            if(visited.find(h)!=visited.end()){
                c=c+visited[h];
            }
            visited[cpx]++;
        }
        return c;
    }
};
int main(){
    vector<int> A{4,2,2,6,4};
    Solution obj;
    int totalCount=obj.solve(A,6);
    cout<<totalCount<<endl;
}
```

(Code)

⑪ Length of Longest Substring without any Repeating character.

Ex:- I/P: s="abcabcbb" | O/P: 3

I/P: s="bbbbb" | O/P: 1

Approach 1: Brute Force:-

- generate all the substring by taking two for loop in nested forloop one by one check for each & every element if the element is already visited in the current substring then find the length and return/break from the nested loop
- to find if element is present or not in current substring one can use Hash set.

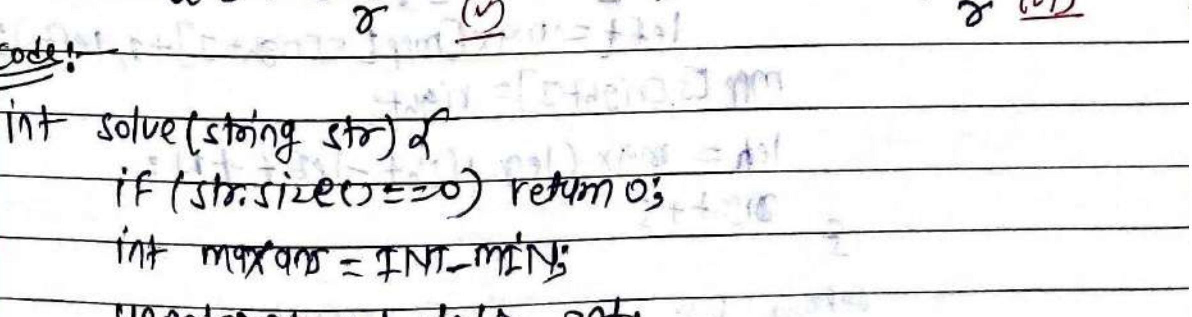
```
int solve (string str) {
        if (str.size() ==0)  return 0;
        int  maxans = INT_MIN;
        for (int i=0; i< str.length(); i++) {
                unordered_set<int> set;
                for (int j=i; j< str.length(); j++) {
                        if (set.find(str[j]) != set.end()) {
                                maxans = max (maxans, j-i);
                                break;
                        }
                        set.insert(str [j]);
                }
        }
        return  maxans;
}
```

TC: O(N²)
SC: O(N)

Approach 2: optimised solution 1:-

we will have two pointers 'left and right. pointer 'left' is used for maintaining the starting point of substring while 'right' will maintain the end point of the substring. right pointer will move forward and check for the duplicate occurrence of the current element if found then 'left' pointer will be shifted ahead so as to delete the duplicate elements.

Ex:-



Code:-

```
int solve(string str) {
        if (str.size()==0) return 0;
        int maxans = INT_MIN;
        unordered_set <int> set;
        int l=0;
```

```
for (int r=0; r< str.length(); r++){
    if (set.find(str[r]) != set.end()) {
        while (l<r && set.find(str[r]) != set.end())
            set.erase(str[l]);
      }     l++;
    }
    set.insert(str[r]);
  } maxans = max (maxans, r-l+1);
  return maxans;
}
```

TC: O(2*N) (sometimes left & right both have to travel complete array)
SC: O(N) (Hashset of size N)

Approach 3: Optimized Solution 2:-

- we will make a map that will take care of counting the elements and maintaining the frequency of each and every element as a unity by taking the latest index of every element.

Code:-

```
int lengthofLongestSubstring (string s){
    vector <int> mpp (256, -1);
    int left =0, right=0;
    int n= s.size();
    int len=0;
    while (right <n) {
        if (mpp [s[right]] !=-1)
            left = max(mpp[s[right]]+1, left);
        mpp [s[right]]= right;
        len = max (len right-left+1);
        right++;
    }
    return len;
}
```

TC: O(N)
SC: O(N)