## *. Hashing

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 1 | 3 | 2 |

| Key | Frequency |
|---|---|
| 1 | 2 |
| 2 | 2 |
| 3 | 1 |
| 10 | 0 |
| 4 | 0 |

- Brutforce approach:-

→ Pseudocode:-

```
int fun(number, a[])
  {
     int Count = 0;
     for( i=0; i<n; i++)
      {
        if (arr[i] == number)
            Count += 1;
      }
     return Count;
   }
```

→ Time Complexity:- O(N)

If we have 'Q' inputs then Q × O(N)

i.e. Time Complexity :- O(Q×N)

suppose if $Q = 10^5$ and $N = 10^5$

then $O(10^5 \times 10^5)$

$O(10^{10})$

$O(10^8) \approx 1$ second.

i.e. Program takes 1second.

$10^8 \rightarrow 1$sec

$10^{10} \rightarrow \cancel{\frac{1}{10^8}} \frac{1}{10^8} \times 10^{10} \approx 10^2$ i.e. 100 seconds.

So, its not good.

So, we use **Hashing** for these type of problems

→ Hashing :- Prestoring and Fetching.

i.e. We store some values and fetch those values whenever needed.

→ Number Hashing

| 1 | 2 | 1 | 3 | 2 |
|---|---|---|---|---|

↳ suppose we can take atmax (12)

|  |  |
|---|---|
| 1 | 2 |
| 3 | 1 |
| 4 | 0 |
| 2 | 2 |
| 10 | 0 |
| ¦ | ¦ |
| 12 | 0 |

Hash Array

|  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

→ Code :-

```
#include <-->
using ---
int main(){
  int n;
   cin>> n;
  int arr[n];
  for(int i=0; i<n; i++){
      cin>> arr[i];
     }
```

```
//Precompute
int hash[13] = {0};
for(int i=0; i<n; i++){
  hash[arr[i]] += 1;
 }

int q;
cin >> q;
while(q--){
     int number;
     cin>> number;
  // Fetch
cout << hash [number] << endl;
  }
  return 0;
}
```

[NOTE :-

→ size of Hash array allowed inside main funcⁿ :- arr [$10^6$] ↳ in case of 'int'

if size > $10^6$
then it will give segmentation fault.

→ size of Hash array allowed outside main funcⁿ or Globally :- arr [$10^7$] ↳ in case of 'int'

i.e. Array Hashing cannot be done if size of array is > $10^7$.

→ character Hashing

S = "abcdabefc"
↑
string

- Brutforce approach :-

→ Pseudocode :-

```
int fun(char ch, string s){
  int count =0;
  for (i=0; i<n ; i++)
    { if (s[i] == ch)
        count ++;
    }
  return count;
}
```

→ Time complexity :- O(Q×N)

Q-inputs
a → 2
b → 2
c → 2
|
z → 0

→ In above question, it is given for lower case characters.

so,

Hash Array →

| 2 | 2 | 2 | 1 | - - - - | 0 | 0 |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | - - - - | 24 | 25 |
| a | b | c | d | - - - - | y | z |

abcdabefc

★ → Observe that here we are restricted for lowercase characters, if there is no condition of upper / lower case then we can make a Hash Array[ ] of size '256'.

↓ since, there are 256 characters.

Hash Array →

|  |  |  | - - |  | - - - |  | - - - |  |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 |  | 65 |  | 97 |  | 255 |
|  |  |  |  | Ⓐ |  | ⓐ |  |  |

→ Code :-

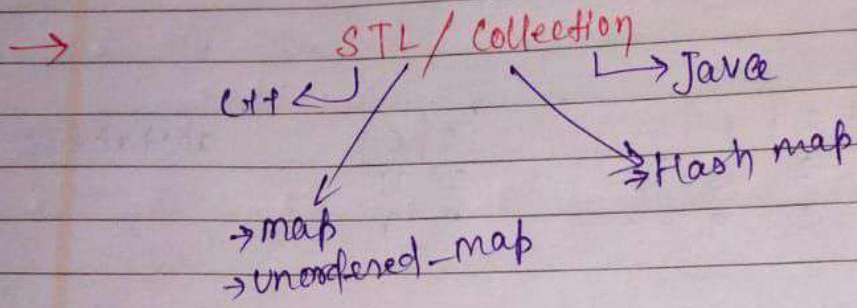
```
int main(){
string s;
cin>>s;
//Pre-Compute
int hash[26] = {0};
for(int i=0; i< s.size(); i++){
    hash[s[i] - 'a']++;
  }

int q;
cin>>q;
while(q--){
  char ch;
  cin>>ch;
  //fetch
  cout<< hash[ch-'a']<<endl;
  } return 0; }
```

→ Code for all characters :-

```
int main(){
  string s;
  cin>>s;
// pre-compute
int hash[256] = {0};
for (int i=0; i< s.size; i++){
   hash[s[i]]++;
  }
int q;
cin>>q;
while(q--){
  char ch;
  cin>>ch;
  // fetch
cout << hash[ch] <<endl;
  }
  return 0;
  }
```

→



→ Map

| arr = | 1 | 2 | 3 | 1 | 3 | 2 | 12 |
|---|---|---|---|---|---|---|---|

(12→1)
(3→2)
(2→2)
(1→2)
map

mpp [arr[i]]++

mpp[1] → 0
mpp[1]++ → 1

- map < int , int >

Key ie number

value Give frequency or count

[NOTE!-

|   |   |   |   |   |   |    |
|---|---|---|---|---|---|----|
| 1 | 2 | 3 | 1 | 3 | 2 | 12 |

In Array Hashing, here we have to make a hash array[ ] of size 13.
But, in map, we only need to store (1, 2, 3, 12)
↳ count of these

so, map takes some less memory. ]

→ Code :-

```
#include <-->
using ---
int main(){
  int n;
  cin>>n;
 int arr[n];
 for (int i=0; i<n ; i++){
     cin>>arr[i];
   }
 // Pre-compute
  map< int, int> mpp;
 for( int i=0; i<n; i++){
     mpp [arr[i]]++;
   }
```

```
int q;
cin>> q;
 while (q--){
  int number;
   cin>> number;
    // fetch
  cout << mpp[number]
             <<endl;
   }
  return 0;
 }
```

[NOTE!-

• Values in map are stored in sorted order.

|   |   |   |   |   |   |    |
|---|---|---|---|---|---|----|
| 1 | 2 | 3 | 1 | 3 | 2 | 12 |

```
for( auto it : mpp){ // iterate in the map.
    cout << it.first << " → " << it.second << endl;
   }
```

O/P screen.

```
1→2
2→2
3→2
12→1
```

print key

print count/ frequency

- Time complexity of Map :-

storing }
Fetching } → (log n) ; n → no. of elements in map.
↳ in all cases (Best, Avg., Worst)

→ Unordered-Map

- Values in Unordered_map are not stored in sorted order.

- Time complexity of unordered_map :-

storing }
Fetching } → O(1) (Best Avg.)
→ O(n) ; n is no. of elements in unordered-map.
(Worst)

☆☆☆ [NOTE:-

We generally use unordered_map, but if its give time limit exceeded (TLE), then we switch to map.

→ TLE occurs due to worst case. i.e. O(n) of unordered map.
And, worst case occurs due to internal collisions.

→ How Hashing is done?

- Division method
- Folding method
- mid-square method

→ Division method :-

arr → {2, 5, 16, 28, 139}

arr[i] % 10

↓ 2, ↓ 5, ↓ 6, ↓ 8, ↓ 9

Hash Array

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
|   |   | 1 |   |   | 1 | 1 |   | 1 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

{suppose, we have given that array size cannot be greater than 10}.

arr → {2, 5, 16, 28, 139, 38, 48, 28, 18}

arr[i] % 10

↓ 2, ↓ 5, ↓ 6, ↓ 8, ↓ 9, ↓ 8, ↓ 8, ↓ 8, ↓ 8

Hash Array

0
1
2 → 2
3
4
5 → 5
6 → 16
7
8 → 18 → 28 → 28 → 38 → 48
9 → 139

↓ Will be in sorted order

↳ can be implemented using Linked list.

→ If we have an array like

arr → {18, 28, 38, 48, ---- 1008}

0
1
2
3
4
5
6
7
8 → 18 → 28 → 38 → 48 → ---- → 1008
9

All keys ends at hash in same index.

↳ Due to this collision happens

[NOTE:-

→ map <___, ___>

↳ It can be any data structure.
e.g. Pair<int, int>

→ unordered_map<___, ___>

↳ Here, it cannot be in pairs.

e.g. Pair<int, int> ✗✗

vector ✗✗