08-11-25

Date

Hashing Theory

Hashing → Hashing is a technique used in computer science to quickly find, store and manage data.

Prestore the data somewhere so that we can use it Later.

Number Hashing →

arr[] = [5, 6, 4, 4, 6, 5, 5]

→ If a person comes and ask you, how many times does 5 exists in an array.

You would go and traverse through the array and count the occurance of 5.

arr[] = [5, 6, 4, 4, 6, 5, 5]

start

count = ~~0~~ ~~1~~ ~~2~~ 3

You would say, 5 occurred 3 times in the array

→ Now another person comes and ask you, how many times does 4 occurs in the array.

You would go with the same process again, check each element of array & say 4 occurred 2 times

Date

→ Now one more person comes and ask you, how many times 5 exists in an array.

Now you're fed up, because you did not store the count of 5 earlier so you need to count again.

So, the size of array is → 7

And number of queries → 3

Comparisons you made → $7 \times 3 = 21$

So every time, someone comes and ask you you would go and check in array.

<u>This is very inefficient</u>

why ?

Let's say size of array $\Rightarrow 10^5$

Number of queries $\Rightarrow 10^5$

(Time complexity) Comparisons we need to make $\Rightarrow 10^5 \times 10^5$
$\Rightarrow 10^{10}$

But on any online platform like leetcode + <u>1 sec</u> given to solve problem which can have <u>10^8 operations</u>, but here we have 10^{10}.
So we will get TLE → Time Limit Exceeded

Date

Here comes the hashing part

We will traverse the array once, and store the count of each number of array somewhere.

Somewhere ?

Let's try to store the count in array.

If we want to store the count of 5, we will store it on the index 5 of the array so we need array of 6 size.

We have 2 options →
- → Create array of size 10^6
- → Create array of size
  ↓
  max of array + 1

Now it's upto you, which one you want to use, in one you will have extra space complexity and other one will require extra traversal of array to find out maximum of array.

Let's precompute now

arr [] = [5, 6, 4, 4, 6, 5, 5]

Steps

1 → Traverse through the array & find out max.

Here max of array is → 6

Date

2 ⇒ Create a hash array with size ⇒ max + 1
Fill it with zeros

hash[] =

0	1	2	3	4	5	6
0	0	0	0	0	0	0

3 ⇒ Traverse through the array and keep on updating the count in hash array.

arr[] = [5, 6, 4, 4, 6, 5, 5]

hash[] =

0	1	2	3	4	5	6
0	0	0	0	~~0~~ ~~1~~ 2	~~0~~ ~~1~~ ~~2~~ 3	~~0~~ ~~1~~ 2

4 ⇒ Final hash array will look like below after traversal & update in step 3 ⇒

hash[] =

0	1	2	3	4	5	6
0	0	0	0	2	3	2

So now whenever someone comes up and ask you the count of any number. You get it from hash array and return. No need to traverse again.

• $TC \Rightarrow \underbrace{O(N)}_{\text{Traversal to find max}} + \underbrace{O(Q)}_{\text{No of queries}} + \underbrace{O(N)}_{\text{Traversal to update hash arr}}$

$SC \Rightarrow O(\text{max of array})$

Date

Challenges with number hashing implemented with array →

What if inside in any array, max value is 10^9.

We won't be able to store in hash array because →

In Java → We can only create array with size

10^6 → Inside Main method
10^7 → Outside " "
10^8 → Array to store boolean

In Javascript → We can create $2^{32}-1$ size → 10^9

* Character Hashing →

As we did number hashing in array, what if we have given the string and need to find out count of any character in the string.

str = 'abaca'

First Query → Find count of 'a' in str

Second Query → Find count of 'c' in str

Third Query → Find count of 'z' in str

Date

Instead, of doing traversal is string and find out the count again and again.
We will do character hashing.

[But how do we do character hashing when we have indexes as number in array?]

Every character has an ASCII Value.

a → 97
b → 98
c → 99
:
:
z → 122

So we store the characters count in the hash array based on their ascii value.

Let's say 'a' charcter, 97 ASCII value so we will store the count of 'a' character on the index 97 in the hash array.

Let' say we have a string with lower case characters.
So we have ASCII range → 97 - 122

Date

Now if we create an hash array with size → 123 it will work fine because we will be able to store all characters as the max value in range 122 of 'z'.

But first 97 values are waste which has indexes from 0 - 96, we will not be storing in them.

To optimise this →

⇒ We take an hash array with size 26 char.

While storing →

'a' - 'a' ⇒ 0 So 'a' gets stored on 0

'b' - 'a' ⇒ 1 So 'b' " " on 1

'c' - 'a' ⇒ 2 So 'c' " " on 2

So formula becomes →

In Java →

str.charAt(i) - 'a' char gets converted to ASCII value automatically

In Javascript →

str.charAt(i).charCodeAt(0) - 'a'.charCodeAt(0)

There is no character, one character also treated as string so we need to convert it to ASCII value using charCodeAt(0)

Date

str = 'abaca'

	0	1	2	3	4	5	6	7	- - - - - - - -	26
hash[] =	~~0~~	~~0~~	~~0~~	0	0	0	0	0		0
	~~1~~	1	1							
	~~2~~									
	3									

Same as number hashing.

★ Challenge in number hashing we faced →

> 10^6

C++	Java	Javascript
Map	HashMap	Map
STL & collections Unordered Map	TreeMap	

Instead of hash[], we will use hashmap.

new Map() → map.get()

Map → Time complexity to get key value pair-

O(1) → Best Average

O(N) → Worst

Very Very Rare ↙ → N is no of elements in Map

Date

* Internal Hashing Methods ⇒

Division Method ⇒ One of the simplest and most commonly used hashing techniques.
The fundamental idea behind this method is to divide the key by a suitable prime number and use the remainder as the hash value. The choice of prime number is crucial as it ensures a more uniform distribution of hash values, thereby minimizing collisions.

In real life, max value which can be stored in array is very big. But to understand let's say we can only store values upto 10 in an array.

hash =

0	1	2	3	4	5	6	7	8	9
~~0~~ 1	~~0~~ 1	0	~~0~~ 1	~~0~~ 1	0	0	0	0	0

arr = [21, 25, 36, 52]

Modulo ⇒ Largest prime number upto 10 → 7

Idx

21 % 7 ⇒ 0

25 % 7 ⇒ 4

36 % 7 ⇒ 1

52 % 7 ⇒ 3

hash arr final ⇒

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

To get also, we need to do modulo.

Date

arr = [21, 25, 36, 52, 14, 35, 7, 38]

hash[] =

0	1	2	3	4	5	6	7	8	9
~~0~~	~~0~~	0	~~0~~	~~0~~	0	0	0	0	0
~~1~~ ~~2~~ ~~3~~ 4	1		1	1					

Idx where value will be stored

21 % 7 = 0
25 % 7 = 4
36 % 7 = 1
52 % 7 = 3
14 % 7 = 0
35 % 7 = 0
7 % 7 = 0

Final hash array →

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

If you see, 21, 14, 35, 7 all got stored on index 0. If we get 21, its count comes as 4.

Wrong ✗

Solution for this ?

Instead of storing index as int in array, we will store linked list as sorted so that when we $\overset{\text{make}}{\text{~~get~~}}$ the get request, its time complexity is optimised.

Don't worry if this does not make sense

Date

now, it will all make sense once you get to know binary search & linked List!

arr = [21, 25, 36, 52, 14, 35, 7, 38]

		hash array vertically
21 % 7	= 0	0 → 7 → 14 → 21 → 35
		1 → 36
25 % 7	= 4	2
		3 → 52
36 % 7	= 1	4 → 25
		5
52 % 7	= 3	6
		7
14 % 7	= 0	8
		9
35 % 7	= 0	
7 % 7	= 0	

So now when we make the get request of 21,

we do the modulo ⇒ 21 % 7 = 0

Now we search linked list on index 0 →

7 → 14 → 21 → 35

Search for 21 in the linked list.

As it is sorted linked list, we can apply Binary Search in it.

That's it

Happy Coding