Hashing - Part 1

29th April, 2022

Agenda:

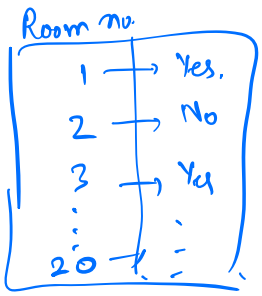
- What/Why Hash Map / Dictionary ?
- Dictionary in Python
  - Operations defined on them
- Sneak peek into Implementation of Dictionary
- Some interesting problems

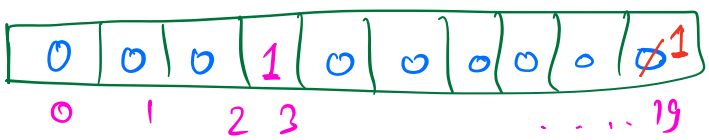
* Why HashMap / Dictionary ?

Hotel.

1 - 20 rooms.

Checking each room's availability.





→ Iterate, whether I have an entry in this array with '0'.

(**) Room nos 1, 2, 3, ... 20 ←

X

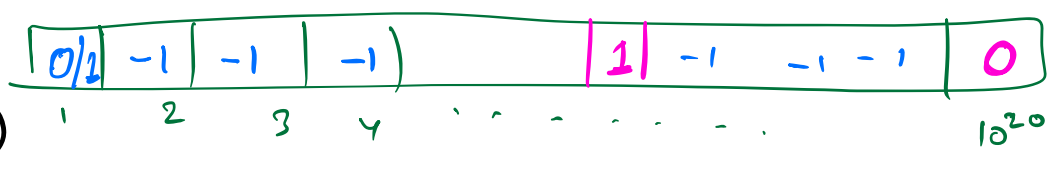
Room nos. 1 10 100 10^3 10^4 ... 10^{20}

101, 102, 104

201

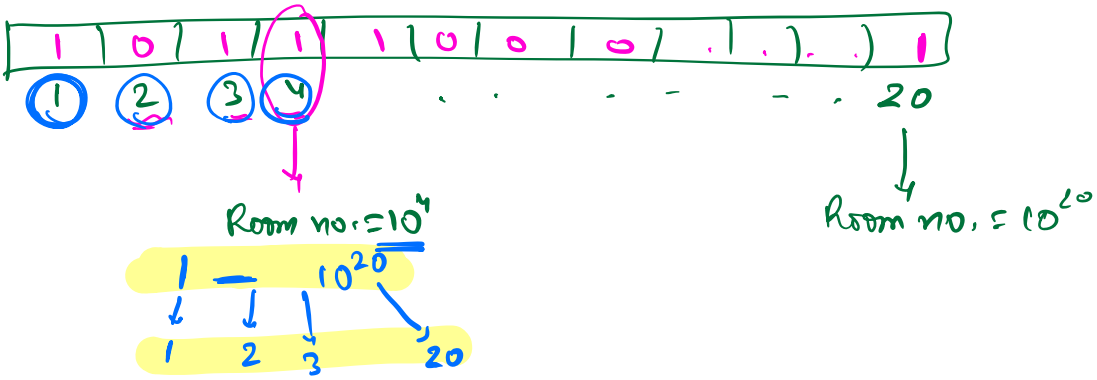
301

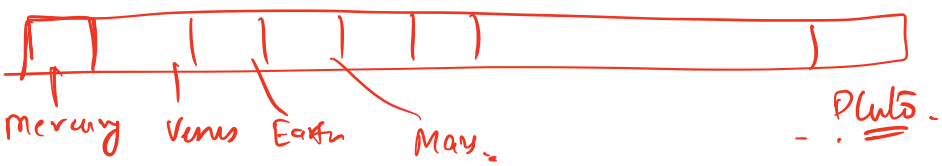
(XX)

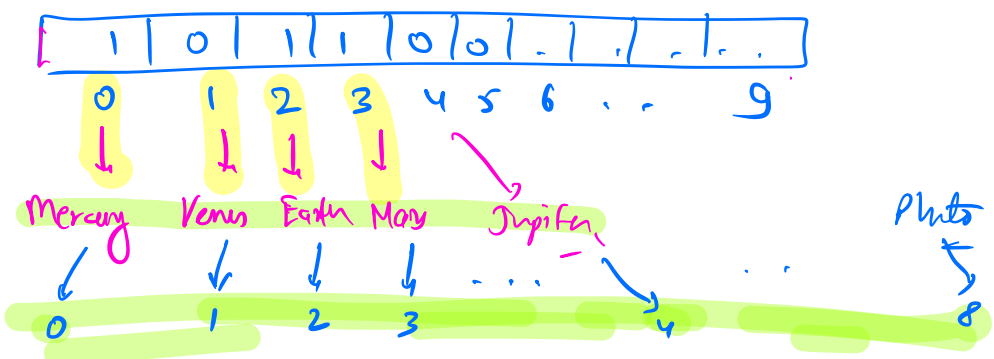


Waste of space. ✗

Use:
** Index as the power of 10

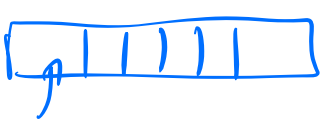






Hash Maps.

(key - value pair)



Element of list integer

String.

Entry in a Hash Map.

(key-value pair.)

Room name (key)	Occupancy (value)
Mercury	0
Venus	1
Earth	1
⋮	⋮

← key

← 4 entries.

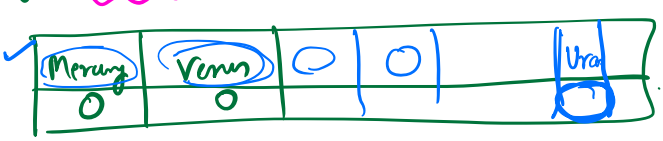
Uranus	0
Neptune	1

↖ Direct access.

O(1) time

* 2D array / 2D list

2 × N.



↙ 9 entries

9 × 2 entries.

O(N)

↖ Iterate.

* Dictionary.

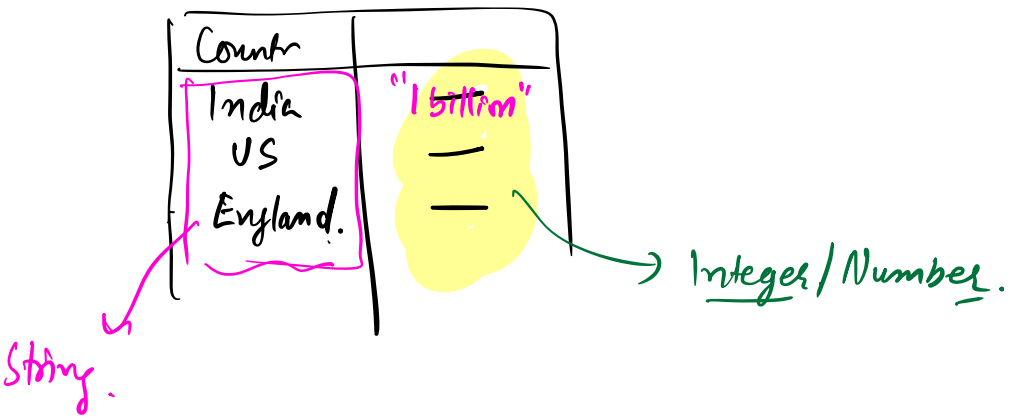
Lookup.

(Given a key, returns value) → O(1) time.

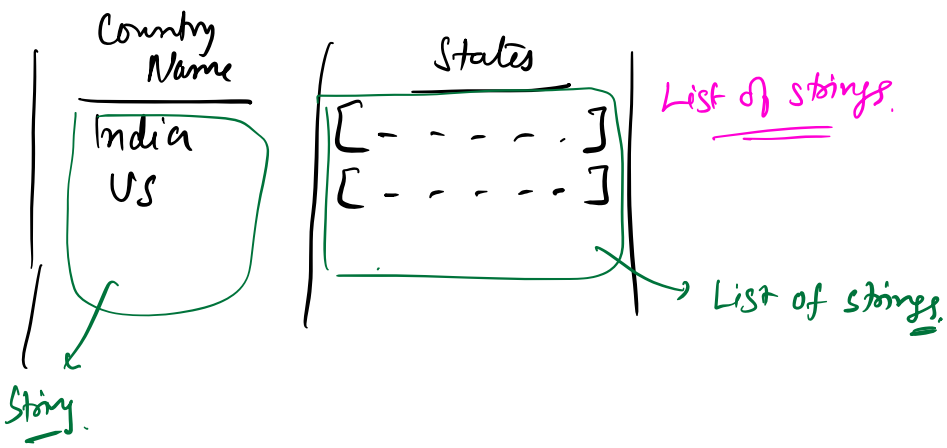
Insertion.

(Insert a K.V pair) → O(1) time

* Population of every country.

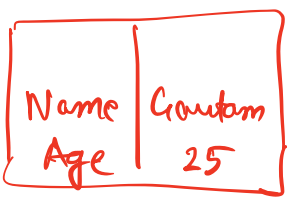


*



Dictionary in Python.

```
li = []
my_dict = {}    // Dictionary.
my_dict = { "Name" : "Gautam", "Age" : 25 }
```



2 important properties:

→ Keys should be immutable.

↳ String, Tuple, Number

Mutable.

↳ List, Dictionary

* Can values be mutable ?

Yes.

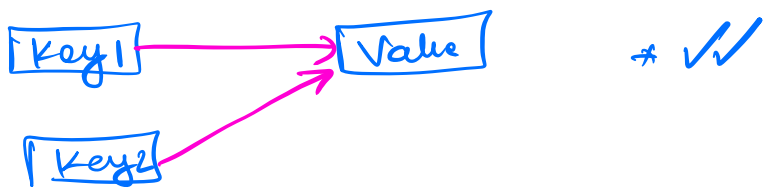
↳ Number, string, List, Tuple, Dictionary.

→ Keys are unique.

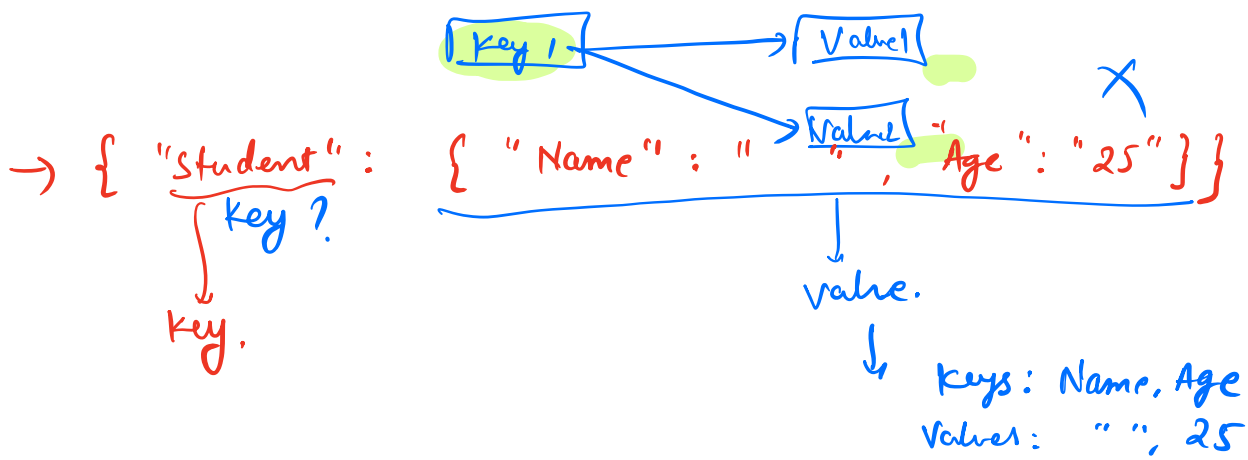
(Each key is mapped to only one value).

India → 1 billion
India → 2 billion ✗

India → 1 billion
China → 1 billion } ✓



* ✓✓



Functions on Dictionary.

* Length

len(my_dict)

↓

No. of entries present.

* `my_dict = {}`

Add a new k.v pair:

→ `my_dict["Govind"] = 27`

= =

{"Govind": 27}

→ `my_dict.update({"Govind": 27, "Deepak": 29})`

↳ Add multiple k.v pairs at once.

* `del my_dict["Govind"]`

↳ {"Deepak": 29}

* Iterate over a dictionary.

```
for ele in arr:
```

↓ ↓

```
for key, value in my_dict.items():
    print(key)
    print(value)
```

my_dict: {"Govind": 27, "Deepak": 29}

↑ ↑

Govind

↳

Deepak

29

* Iterate over the keys.

```
for k in my_dict.keys():
```

↳ List of keys, iterator

range(0,10) → Iterator

* Iterate over the values.

```
for v in my_dict.values():
```

* Get a value given a key.
  * `my_dict[key]` → value. None if key is not present.
  * `my_dict.get(key, default)` → value.

* Check if a key is present in dictionary.

①
```
if my_dict.get(key) == None:
    // key is not present.
```
O(1) time

②
```
if key in my_dict.keys():
    // key is present.
```
O(N) times

** ③
```
if key in my_dict:
    // key is present.
```
O(1) time

* ["Ban", "Mumbai"] : "cities"

X Invalid dictionary

**

Date	→	25
Date	→	31

X

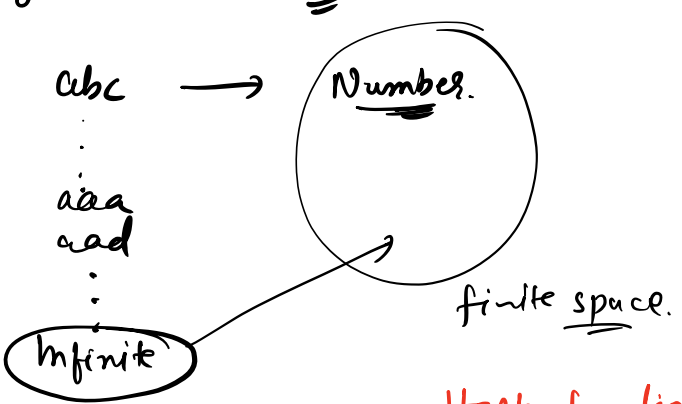
Date → ~~25~~ 31

Date → 31 Reassignment.

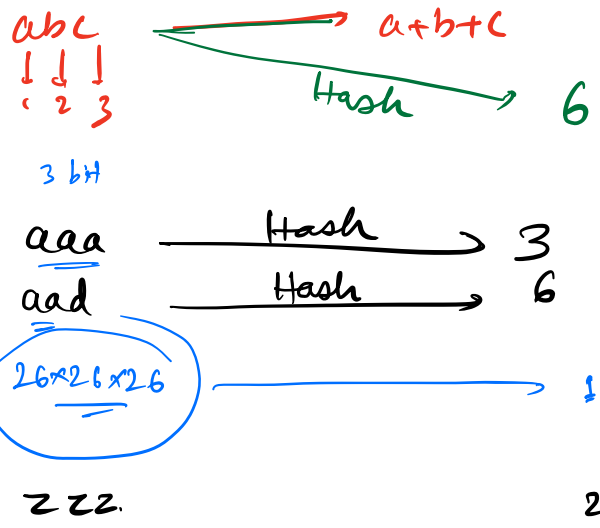
** How hashmaps are implemented ?

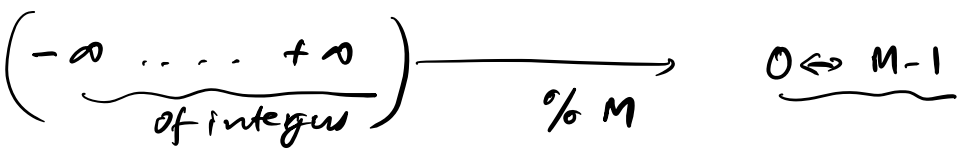
Key		Value
abc	→	"Mumbai"
aaa	→	"Gurgaon"
aad	→	"Kolkata"

Keys are hashed.

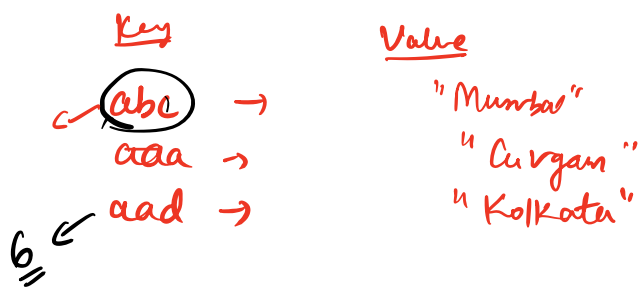


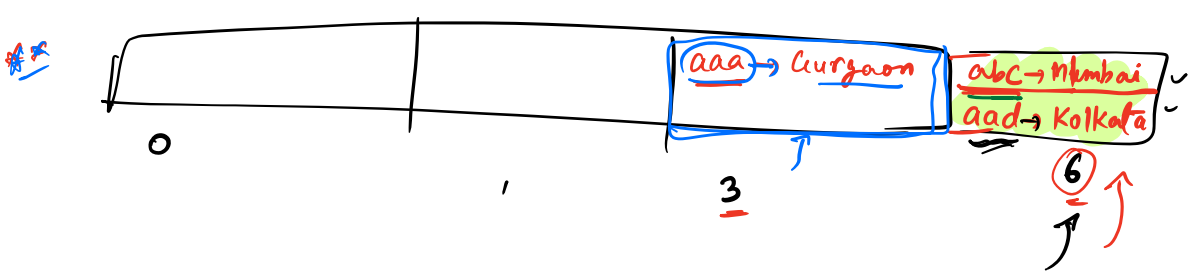
Hash function

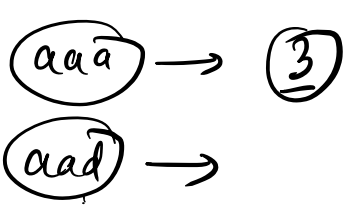




map







aad
abc
ada
cba
} 6

Worst case TC of lookup: O(N).

(**) In real life, Hash functions are very well designed.

"collision."

A good hash function → collisions should be very less.

* Avg time complexity → O(1)

SHA-256

{ aaa → $a \times 26^2 + a \times 26^1 + a \times 26^0$
zzz → $z \times 26^2 +$.

{ "a" : None } ✓

** Break till 10:18 **

** * [10, 20, 40] → 2

Problems.

Q. You are given an array of N elements.
Return Frequency of value 'x'.

Q queries

arr = [2, 6, 3, 8, 2, 8, 2, 3, 8]

Q queries { Freq of 2 → 3
Freq of 3 → 2
Freq of 8 → 3

{ 2 : ~~x~~ 3
6 : 1
3 : x 2
8 : ~~x~~ 3 }

Brute force.

```
for i in Q :
    // ith Query.
    for ele in arr :
        cnt // Count
```

TC: O(N*Q)

Optimise ?

** Create a dictionary // O(N)
Key → elements in array
value → Count of each element

** For Q queries, lookup from // O(Q)
dictionary & return.

TC: O(N+Q)

```
my_dict = { }
for ele in arr:
    if ele not in my_dict:
        my_dict[ele] = 1
    else
        my_dict[ele] += 1
```

→ O(N)

```
for q in Q:
    // return value of val.
    print my_dict[val]
```

→ O(Q).

O: (N+Q)

Q. Given an array, count no. of distinct elements.

Create a dictionary.

return len(my_dict)

** Sets.

Keys

values X.

len(set(array)) **

Imp

Order in a dictionary

→ "Deepak" : 25,
→ "Alok" : 20

< Python 3.7	≥ Python 3.7
Unordered	* Insertion order is maintained.

Q. Given an array, return the first non-repeating element in the array.

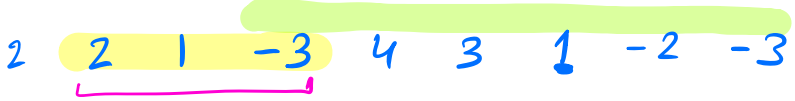
e.g. [8, 2, 8, 3, 1, 2, 6, 5]

ans = 3

Create a dictionary

{
→ 8 : 2
→ 2 : 2
→ 3 : 1
1 : 1
6 : 1
5 : 1 }

Q. Given an array, check if there exists a subarray with sum = 0.



Brute force

→ Consider all subarrays.
Get their sum.

```
for i in range(len(A)):                    # O(N^2)
    for j in range(i, len(A)):
        // Get sum of A[i:j]
        sum = 0
        for k in range(i, j+1):            # O(N)
            sum += A[k]                    # Prefix sum
        if sum == 0
            return True
return False
```

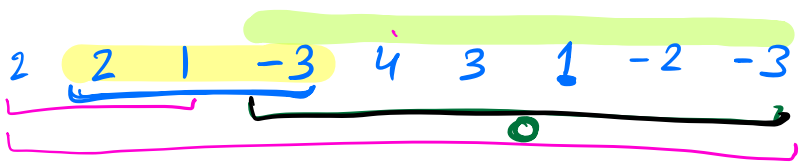
O(N^3)

II optimisation

```
// Build prefix sum.
// pre[i] → Gives prefix sum till i
for i in range(len(A)):
    for j in range(i, len(A)):
        //sum of A[i:j]
        if pre[j] - pre[i-1] == 0
            return True
return False.
```

$O(N^2)$

Optimisation



Prefix-sum → 2 4 5 2 6 9 10 8 5

Pre[i] → sum from 0 to i
Pre[j] → sum from 0 to j

* If for some i,j → Pre[i] == Pre[j],

** Build the prefix sum array. ← O(N)
Check if there is a repeating element.
→ Dictionary.

O(N) ↙

[5, 3, 0, 4]

5 8 8 12 ✓

[0 0 0 0 0]

0 0 0 0 0 ✓

[5 5 5 5]

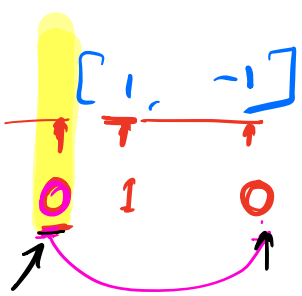
[5 10 15 20] ✓

** [-1, 1]

[-1, 0]

No repeating element

Edge case. → × Subarray with sum 0 is starting from index 0.



[-1, 1, 3, 4]

[-1, 0, 3, 7]

pre[i] = arr[i] + pre[i-1]

0

Pseudo code.

```
dict[0] = 1        // For edge case.
sum = 0
for ele in A:
    sum += ele
    if sum in dict:
        return True
    else
        dict[sum] = 1
return False
```

Can also do using sets.

Doubt session

s = "cbobcdbobc"

prev = 0 ← s.find("bob", 0)

-1

Prev = -1

cbobcdbobc

x

obcdbobc

x

b obc