Hashing

→ It is mainly used to implement dictionary, where you have key value pair, as well it also used when you set

→ Search
Insert — O(1) on average
delete

→ Not useful for
ⓐ finding closest value
ⓑ sorted data
ⓒ prefix searching.

We use
AVL trees (or) Red black tree.
trie

Application of Hashing.

1. Dictionaries
2. Database Indexing [hashing & B or B+ trees]
3. cryptography
4. Caches, Data Compression
5. Symbol tables in compilers/ Interpreters
6. Routers, Load Balancing.
7. , Blockchain, Image Processing.
8. getting data from databases

and so on.

* Prestoring values into some data structure like list/dict/set and the fetching it.
→ In hashing, we always have unique values, all value are unique. [If will override previous case

☺ Hashing is a Technique
ReBiT Hashtable is a Data stru

2nd most used DS in Computer science.

Direct Address Table

DAT is a DS that has the capability of mapping record to their corresponding keys using arrays. In direct address tables, records are placed using their key values directly as indexes. These facilitate fast searching, insertion and deletion operation.

MAY 05 SUNDAY '24
19th Week 126-240

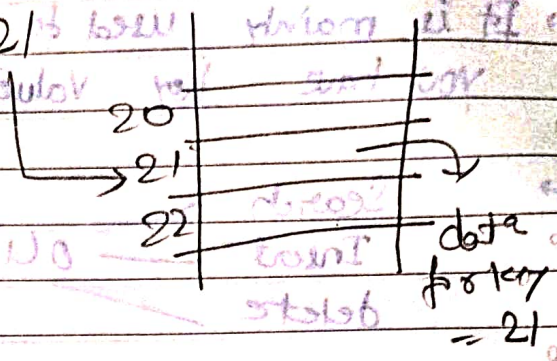
S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

Hashing

Advantages
- Searching
- Insertion
- deletion

$\Rightarrow O(1)$ time



Limitions :-
1. Prior knowledge of maximum key value.
2. Practically useful only if the maximum value is very less.
3. It causes wastage of memory spaces if there is a significant difference b/w total records and maximum value.

Hashing can overcome these limition of direct address table.

Hash Functions

Hashing is a techniques (or) process of mapping keys and values into the hash table by using a hash function. It is done for faster access to elements.

The efficiency depend on the hash function.

The task of hash function to convert those large keys, string, floating point number into small values that can be used as an index in the hash table.

(a) Should always map a large key to some small keys.
(b) Should generate values from 0 to m-1. [m is size of hash table]
(c) Should be fast, $O(1)$ for integers and $O(len)$ for string of length 'len'.
(d) Should uniformly distribute large keys into hash tables slots.

Example Hash function.

① h[large_key) = large_key % m

② for string, weight sum str = 'abcd'

eg: (str[0] * n^0 + str[1] * n^1 + str[2] * n^2 + str[3] * n^4) % m

③ universal hashing.

↳ We choose Prime No. which are not close to power.

→ Set of hash function, we have to pick randomly and you hash function

Collision handling

Birthday Paradox 23 men

(collision is the conditions in which two keys map to the same entry in the hash table.

* If we know keys in advance, then we can go with perfect hashing
* If we don't know keys, then we use one of the following.

Following are the ways to handle collision
- chaining
- open addressing
  - Linear Probing.
  - Quadratic Probing.
  - double hashing.

Chaining (separate chaining):

→ The idea behind separate chaining is to implement the array as a linked list called a chain.

→ The linked list data structure is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these element are inserted into a singly linked list which is known as chain.

→ we have array of linked list headers, whenever a collision happens you insert and item at the end of linked list.

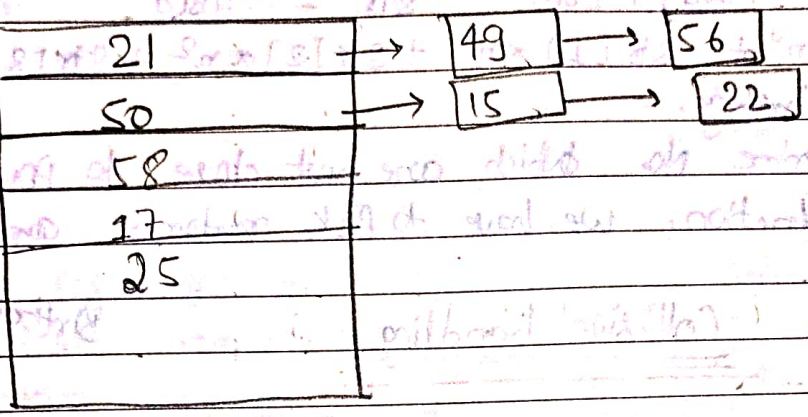
MAY
07
'24
19th Week
128-238
TUESDAY

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

hash(key) = key % 7

keys = {50, 21, 58, 17, 15, 49, 56, 22, 23, 8, 5}



Performance

m = No. of slots in hash table.
n = No. of keys to be inserted.
Load factor $\alpha = n/m$

Expected chain length = α
Expected time to search = $O(1+\alpha)$
" " " insert/Delete = $O(1+\alpha)$

} assumption that all the keys are uniformly distributed.

Data Structure to storing chains

(a) Linked list — search $O(l)$, Delete $O(l)$, insert $O(1)$ (not cache friendly)

(b) Dynamic Size arrays (cache friendly) { vector in c++, Array list in Java, list in Python } $T_C = O(l)$

(c) Self balancing of BST (AVL tree, Red black tree) (not cache friendly)

$O(\log_2 l)$ — search, insert, delete

from Java & they started to use BST for hash maps.

Implementation of chaining in Python

BUCKET = 7 [No. of slots in hash table]

0	[]
1	[]
2	[]
3	[]
4	[]
5	[]
6	[]

Insert → 70, 71, 9, 56, 72

70	56
71	
9	72
[]	
[]	
[]	
[]	

[[], [], [], [], [], [], []] | [[70, 56], [71], [9, 72], [], [], [], []]

[hash = n % BUCKET]

```
h = MyHash(7)
h.insert(70)
h.insert(71)
h.insert(9)
h.insert(56)
h.insert(72)
print(h.search(56))
h.remove(56)
print(h.search(56))
```

o/p – True
False

Insertion method

```
class MyHash:
    def __init__(self, b):
        self.BUCKET = b
        self.table = [[] for _ in range(b)]
    def insert(self, n):
        i = n % self.BUCKET
        self.table[i].append(n)
```

Removable method

```
    def remove(self, n):
        i = n % self.BUCKET
        self.table[i].remove(n)
```

Searchable method

```
    def search(self, n):
        i = n % self.BUCKET
        return n in self.table[i]
```

(a) dict() → It is used to create empty dictionary.
(b) len() → return the No. of items in each dictionary.
(c) clear() → to remove all element from dictionary.
(d) get() → to get the value associated with the key.
(e) pop() → It remove the entry associated with the specified key and return the corresponding value.
(f) popitem() → It remove arbitrary item from the dictionary and returns it.
(g) keys() → It returns all keys associated with dictionary.
(h) values() → It returns all values associated with dictionary.
(i) items() → It returns list of tuples representing key-value pairs.
(j) copy() → to create exactly duplicate dictionary (cloned copy).
(k) set default() : → to set default value in dictionary.

Q1

n = [5,3,2,2,1,5,5,7,5,10]

m = [10, 11, 1, 9, 5, 67, 2]

Q We have to find the elements of m appears How many times in n.

Brute force

```
for i in m:
    count = 0
    for J in n:
        if i == J:
            count += 1
    Print(count)
```

TC = O(mxn)
SC = O(1)

hash-list = | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Initially all are occurring zero times that's why we are keeping 0.
then we are searching that element in hash-list ⓐ if it is there the incrementing its value by 1.

Optimal Solution

```
hash_list = [0] x len(n)      # In this case 11
for i in n:
    hash_list[i] += 1
```

TC = O(n)

1 से कम or 10 से बड़ा कोई element हो तो 0 time occurrences.

```
for i in m:
    if i < 1 or i > 10:
        Print(0)
    else:
        Print(hash_list[i])
```

TC = O(m)

TC = O(m+n)
SC = O(len(n)) // O(1)

Above solution will only works when the elements of list is not varying time to time and prior we knows all those element.
else we have to do this question using dictionary.

dictionary Sol"

```
d = {}
for i in n:
    if i in d:
        d[i] += 1
    else:
        d[i] = 1
for i in m:
    if i in d:
        Print(d[i])
    else:
        Print(0)
```

Q FOR same question we they will ask us about character then also we can implement it using above methods.

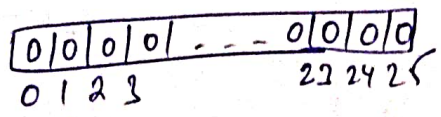
S = "azyxyyzaaaa"

q = ["d", "a", "y", "z"]

M-1 Same as above [Brute force]

M-2 Same as above but we will be using ascii value of each character to find out the same.

M-2

hash_list = 


```
hash_list = [0] * 26
for i in s:                          } n
    ascii_value = ord(i)
    index = ascii - 97
    hash_list[index] += 1

for i in q:
    ascii_value = ord(i)             } m
    index = ascii_value - 97
    Print(hash_list[index])
```

TC = O(n+m)
SC = O(1)

Q If the Question contains all upper case & lower case then apply dictionary method

```
d = {}                               { only for lowercase }
for i in s:
    if i in d:
        d[i] += 1
    else:
        d[i] = 1

for i in q:
    if i in d:
        Print(d[i])
    else:
        Print(0)
```

```
d = {}                               { for handling both upper & lower case. }
for i in s:
    if i in d:
        d[i] += 1
    else:
        d[i] = 1

for i in q:
    if i in d:
        Print(d[i])
    else:
        Print(0)
```

MAY
09
'24
19th Week
130-236
THURSDAY

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

Frequencies of Array Elements

i/p - arr[] = {10, 20, 12, 10, 15, 10, 12, 12}

o/p - 10 3
12 3
15 1
20 1

i/p - arr[] = {10, 10, 10, 10}

o/p - 10 4

i/p - arr[] = {10, 20}

o/p - 10 1
20 1

M-1

```
def countfreq(arr,n):
    mp = dict()
    for i in range(n):
        if arr[i] in mp.keys():
            mp[arr[i]] += 1
        else:
            mp[arr[i]] = 1
    for x in mp:
        print(x, ' ', mp[x])
```

TC = O(n), SC = O(n)

```
d = {}
for i in range(0,len(l)):
    if l[i] in d:
        d[l[i]] += 1
    else:
        d[l[i]] = 1
print(d)
print(d[1]) // Printing occurrence of 1 only.
```

M-2

```
d = {}
n = len(l)
for i in range(0,n):
    d[l[i]] = d.get(l[i],0) + 1
```

eg

d[l[0]] = d.get(10,0) + 1 (0 if not available.)

10 = 0 + 1

10 = 1

Sets ({} or S = set())

08 → A set is an unordered collection data types that is iterable, mutable and has no duplicate elements.

09 (a) Distinct element
(b) unordered.
10 (c) No indexing is applied.
(d) union, intersection, set difference etc are fast.
11 (*)(e) uses hashing Internally.

12 s.add()
s.update()
01 s.discard()
s.remove()
02 s.clear()
len(s)

$S_1 = \{2, 4, 6, 8\}$, $S_2 = \{3, 6, 9\}$
Print($S_1 | S_2$) — {2, 3, 4, 6, 8, 9}
Print($S_1 \& S_2$) — {6}
Print($S_1 - S_2$) — {8, 2, 4}
Print($S_1 \wedge S_2$) — {2, 3, 4, 8, 9}

03 (*) union → S_1.union(S_2)
(*) Intersection → S_1.intersection(S_2)
(*) difference → S_1.difference(S_2)
(*) Symmetric difference → S_1.symmetric_difference(S_2)

Dictionary

05 Key : Value pair

06 → Dictionary is a collection of keys values, used to store data value like a map, which, unlike other data types which holds only single value
07 as an elements.

(a) unordered.
(b) All keys must be distinct.
(c) Value may be repeated.
(*)(d) uses hashing internally.

d = {110 : 'xyz', 101 : 'abc', 105 : 'bcd', 104 : 'abc'}
→ Similar to hashmap in Java, unordered map in C++.

MAY 11 '24
19th Week
132-234

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

SATURDAY

d.get(101) → nyc
d.get(125) → None
len(d) → 4
d.pop(105) → {110:'abc', 101:'wny', 106:'bcd'}
del d[106]
d.popitem(1)

Open addressing

Like separate chaining, open addressing is a method for handling collision. 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 that we can increase table size by copying old data if needed) This approach is also known as closed hashing.

The entire procedure is based upon probing. We will understand the types of probing ahead. (cache friendly)

Ways of Implementing open addressing.

(a) Linear Probing.

Linearly search for next empty slots.

⇒ rehash(key) = (n+1) % table_size.

search in circular manner for next empty slot

50, 51, 49, 16, 56, 15, 19

hash(Key) = Key % 7

insert(50), insert(51), insert(15), search(15), search(64), delete(15) search(15)

0	49
1	50
2	51
3	16
4	56
5	15
6	19

Search — We compute hash function we go to that index and compare If we find, we return true. Otherwise we linearly hash.

→ We stop when one of the three cases arise

① Empty slot — False

② key found — True

③ Traversed through the whole table — False

Delete — Problem with simply making slot empty when we delete. So we don't marked the slot empty we marked it as deleted.

→ Primary clusting is problem with linear probing.

To handle Primary clusting problem with linear probing is

$h(key) = key \% 7$

$hash(key, i) = (h(key) + i) \% 7$

① Quadratic Probing

$hash(key, i) = (h(key) + i^2) \% m$

② Double hashing

$hash(key, i) = (h_1(key) + i \times h_2(key)) \% m$.

ⓐ Secondary clusting suffer by Quadratic probing.

ⓑ It might happend that it doesn't find empty slot even if there are empty slot

① $\alpha < 0.5$ if load factor is less than 0.5 then only you will able to find empty slot.

② m is prime in (n/m).

if two these things happen together then quadratic probing works and they will finds empty slots.

MAY
13
'24
MONDAY
20th Week
134-232

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

(b) **Quadratic Probing:**

Quadratic Probing is a method with the help of which we can solve the problem of clustering that was discussed above.
This method is also known as the mid square method.
In this method, we look for the i^2th slot in the ith iteration.
We always start from the original hash location. If only the location is occupied then we check the other slots.

(c) **Double hashing:**

Double hashing is a technique that reduces clustering in an optimized way. $hash(key, i) = (h_1(key) + i\,h_2(key)) \% m$
In this technique, the increments for the probing sequence are computed by using another hash function.
We use another hash function $hash2(x)$ and look for the $i * hash2(x)$ slot in the ith rotation.

(a) If $h_2(key)$ is relatively prime to m, then it always finds a free slot if there is one.

(b) Distributes keys more uniformly than linear probing and quadratic hashing.

(c) No clustering.

$hash(key, i) = (h_1(key) + i\,h_2(key)) \% m$

49, 63, 56, 52, 54, 48

0	49
1	
2	54
3	63
4	56
5	52
6	48

$m = 7$

$h_1(key) = (key \% 7)$

$h_2(key) = 6 - (key \% 6)$

$56 \rightarrow (0 + (4)) \% 7 = 4$

$52 \rightarrow (3 + (6 - (52 \% 6))$

$(3 - 2)$

1 ✓

$54 \rightarrow 5 + (6 - 54 \% 6)$

$5 + 6 = 11 \% 7 = 4$

Will a wild cat observe the fast of Sivaratri?

Performance Analysis of search

(a) Cache Performance of chaining is not good because when we traverse a linked list, we are basically jumping from one node to another, all across the computer's memory.

Like chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table.

m = No. of slots in hash table.

n = No. of keys to be inserted in hash table.

Load factor $\alpha = \frac{n}{m}$ (< 1)

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

$(1-\alpha)$ fraction of the table is empty.

Unsuccessful search means when you have either traverse through the whole table or you ended up with a empty slot.

Chaining

(a) It is simple to implement

(b) Hash table never fills up, we can always add more elements.

(c) Less sensitive to hash function

(d) Cache performance of chaining is not good as keys are stored using linked list.

(e) extra spaces are required for links $(1+\alpha)$

→ better technique

→ It is used when frequency and No. of keys are not known may dynamically increase.

Open Addressing.

(a) It requires more computation.

(b) It may becomes full.

(c) Extra care required for clusting.

(d) It provide better cache performance as everything is stored in same table

(e) might be needed to achieve same performance $= [1/(1-\alpha)]$

→ old technique

→ It is used when frequency and No of keys are known.

When poverty come in the door, love goes out the window

MAY
15
'24
20th Week
136-230
WEDNESDAY

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

Implementation of open addressing in Python

```
Class myhash:
    def __init__(self, c):
        self.cap = c
        self.table = [-1]*c
        self.size = 0

    def hash(self, n):
        return n%self.cap

    def search(self, n):
        h = self.hash(n)
        t = self.table
        i = h
        while t[i] != -1:
            if t[i] == n:
                return True
            i = (i+1)%self.cap
            if i == h:
                return False
        return False

    def insert(self, n):
        if self.size == self.cap: return False
        if self.search(n) == True: return False
        i = self.hash(n)
        t = self.table
        while t[i] not in (-1, -2):
            i = (i+1)%self.cap
        t[i] = n
        self.size = self.size + 1
        return True

    def remove(self, n):
        h = self.hash(n)
        t = self.table
        i = h
        while t[i] != -1:
            if t[i] == n:
                t[i] = -2
                return -2
            i = (i+1)%self.cap
            if i == h:
                return False
        return False

h = MyHash(7)
h.insert(70)
h.insert(71)
h.insert(9)
h.insert(50)
h.insert(72)
print(h.search(56))
h.remove(56)
print(h.search(56))
h.remove(56)
```

o/p - True
False

Hashing – 2

'24 | 20th Week | 137-229

MAY 16 THURSDAY

Intersection of unsorted Array.

M-1

i/p : a[] = [10, 15, 20, 5, 30]
b[] = [30, 5, 30, 80]
o/p : 2

len(set(a).intersection(set(b)))

Naive Approach: → TC = O(m × (m+n))

(1) Initialize : res = 0
(2) Traverse through every element of a[]
↳ check if it has not appeared already
↳ If a new element and also present in b[], do res++
(3) return res.

x → already checked [appeared]
new element

a[] = [10, 10, 30, 20]
b[] = [20, 10, 40, 10, 40]

check if element has appeared

```
for i in range 1(i-1):
    if arr1[j] == arr1[i]:
        flag = True
        break
if flag == True:
    continue
```

If appeared Ignore

If not appeared look for it in another array

```
for j in range(n):
    if arr1[i] == arr2[j]:
        res += 1
        break
return res
```

M-2

```
def intersection(arr1, m, arr2, n):
    res = 0
    for i in range(m):
        flag = False
```

Efficient Solution:

M-3

```
def intersection(arr1, m, arr2, n)
    us = set()
    for i range(m):
        us.add(arr1[i])
    res = 0
    for i in range(n):
        if arr2[i] in us:
            res += 1
            us.remove(arr2[i])
    return res
```

TC = O(m+n)
SC = O(m)

(1) Insert all element of a[] in set
S-a = [10, 20, 30]
(2) Traverse through b[], search for every element b[i] in S-a. If present:
(a) Increment res.
(b) Remove b[i] from S-a.

a = [10, 20, 10, 30, 20]
b = [20, 10, 10, 40]
S-a = [10, 20, 30]
res = 2

MAY 17 '24 FRIDAY

20th Week 138-228

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

Union of unsorted Array

i/p: a = [15, 20, 5, 15]
b = [15, 15, 15, 20, 10]
o/p : 4

(i) Creating a empty set
(ii) adding element of list1. TC = O(m+n)
(iii) adding element of list2 SC = O(m+n)
(iv) printing len(us).

M-1

```
Print( len(set(a).union(set(b)))
```

M-2

```
def unionset(arr1, m, arr2, n):
    us = set()
    for i in arr1:
        us.add(i)
    for i in arr2:
        us.add(i)
    return len(us)
```

Pair with given sum in unsorted Array:—

i/p : arr[] = [3, 2, 8, 15, -8]
Sum = 17
o/p : True

idea for efficient sol^n.

Whenever we are at the i^{th} Position.
If the difference b/w the sum of element and i is in set us the return be. M-2
If not then add it into set.

arr = [3, 2, 8, 15, -8]
Sum = 17
us = {3, 2, 8, 15,

17 - 3 = 14	Not there	add it
17 - 2 = 15	" "	add it
17 - 8 = 9	Not there	add it
17 - 15 = 2	It is there	return True

Naive Solution

M-1

```
def pair(arr, n):
    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            if (arr[i] + arr[j] == n):
                return True
    return False
```

TC = O(n^2)
SC = O(1)

Efficient solution [Two Sum using hashing]

```
def pair(arr, n):
    us = set()
    for i in arr:
        if n - i in us:
            True
        us.add(i)
    return 0
```

TC = O(n)
SC = O(n)

Two pointer is more suitable for specific cases like searching for pairs (or) working with sorted array but the problem of checking for any subarray sum of zero is best solved by cumulative sum.

M-3 Two pointers technique

for using two pointers, the array must be sorted.

```
def Pair(arr,n):
    arr=sort()  # Sort the array first
    left, right = 0, len(1)-1  # Set the pointer at beginning & at end.
    while left < right:
        curr_sum = arr[left] + arr[right]  # Sum of 2 element at pointers
        if curr_sum == n:  # If the sum is equal to target return True
            return True
        elif curr_sum < n:  # If sum is smallest move left
            left += 1  # Pointer to right
        else:  # if the sum is largest, move the
            right -= 1  # right pointer to left.
    return False  # If no pair found, return false.
```

Check for Palindrome Permutation

```
from Collections import Counter
def isPal(s):
    cnt = Counter(s)
    odd = 0
    for i in cnt.values():
        if i % 2 != 0:
            odd = odd + 1
            if odd > 1:
                return False
    return True
```

O(n)

Answer is going to be true if there is at most one character with odd frequency.

abba, acbcab, aaaabbbb ddd, ...

else false

abcc, abc, aaabb cdd, ...

Let $(e_1,e_1)(e_2,e_2)\ldots(e_n,e_n)$ be pair of even frequency character, we can always form a palindrome, $e_1 e_2 e_3$

If there is one odd frequency character we can still form a palindrome $e_1 e_2 \ldots e_{k+1} 0 0 \ldots$

→ using collections module and counter is sub class of dictionary and it is especially designed for counting purpose.

MAY
19
'24
SUNDAY
21st Week
140-226

Subarray of [10, 20, 30]
→ [10]
[10, 20]
[10, 20, 30]
[20]
[20, 30]
[30]

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

Subarray with 0 sum in Python

i/p: l = [1, 4, 13, -3, -10, 5]
o/p: True

i/p: l = [-1, 4, -3, 5, 1]
o/p: True

i/p: l = [3, -1, 2, 5, 6]
o/p: False

i/p: l = [5, 6, 0, 8]
o/p: True

M1 O(n²)

```
def isZero(l):
    n = len(l)
    for i in range(n):
        for j in range(i+1, n+1):
            if sum(l[i:j]) == 0:
                return True
    return False
```

i=0, j=1, sum(l[0:1]) = 4
j=2, sum(l[0:2]) = 7
j=3, sum(l[0:3]) = 5
j=4, sum(l[0:4]) = 6
j=5, sum(l[0:5]) = 7
i=1, j=2, sum(l[1:2]) = 3
→ print(l[i:j]) # for printing.

M2 O(n)

```
def isZero(l):
    Pre_sum = 0
    h = set()
    for i in range(len(l)):        # for i in l:
        Pre_sum += l[i]            # Pre_sum += i
        if Pre_sum == 0 or Pre_sum in h:
            return True
        h.add(Pre_sum)
    return False
```

Pre_sum = 0, h = {}
i=0 : Pre_sum = -3, h = {-3}
i=1 : Pre_sum = 1, h = {-3, 1}
i=2 : Pre_sum = -2, h = {-3, 1, -2}
i=3 : Pre_sum = {-3} already present in hash, return True.

→ Declare a variable sum, to store the sum of prefix elements.
→ Traverse the array and at each index, add the element into the sum and check if this sum exist earlier. If the sum exists, then return true.
→ Also, insert every prefix sum into map, so that later on it can be found whether the current sum is seen before or not.
→ At the end return false, as no such subarray is found.

Sub array with Given Sum. [NO negative elements in array]

i/p: l = [1, 4, 20, 3, 10, 5], sum = 33 , i/p: [1, 4, 0, 0, 3, 10, 5], sum = 7
o/p: Yes o/p: Yes

Naive

Sum = 6
[3, 2, 0, 4, 7]
i=0; J=0, Curr = 3
J=1, Curr = 5
J=2, Curr = 5
J=3 Curr = 9
J=4 Curr = 16
i=1; J=1, Curr = 2
J=2, Curr = 2
J=3, Curr = 6 → return True

TC = O(n2)
SC = O(1)

```
def subsum(arr, sum):
    for i in range (len(arr)):
        curr = 0
        for J in range (i, len(arr)):
            curr += arr[J]
            if (curr == sum)
                return True
    return false
```

Efficient

```
def issum(arr, sum):
    s, curr = 0, 0
    for e in range (len(arr)):
        curr += arr[e]
        while (curr > sum):
            curr -= arr[s]
            s += 1
        if (curr == sum):
            return True
    return False
```

TC = O(n)
SC = O(1)

{We use sliding window technique with a window of variable size.

[1, 4, 20, 3, 10, 5] sum = 33

s=0, e=0, curr = 1
s=0, e=1, curr = 5
s=0, e=2, curr = 25
s=0, e=3, curr = 28
s=0, e=4, curr = 38
s=1, e=4, curr = 37
s=2, e=4, curr = 33

While curr is smaller than sum, extend the window by increasing e.
While curr is greater shrink the window by increasing s.

Longest Subarray with equal number of zero and one.

i/p: arr = [1, 0, 1, 1, 1, 0, 0]

o/p: 6

Naive

```
def longestZero(arr):
    n = len(arr)
    res = 0
    for i in range(n):
        c0 = 0
        c1 = 0
        for j in range(i, n):
            if arr[j] == 0:
                c0 += 1
            else:
                c1 += 1
            if (c0 == c1):
                res = max(res, j - i + 1)
    return res
```

O(n2)

Efficient

```
def longestZero(arr):
    n = len(arr)
    for i in range(n):
        if arr[i] == 0:
            arr[i] = -1
    my_dict = dict()
    sum = 0
    maxlen = 0
    for i in range(n):
        sum += arr[i]
        if sum == 0:
            maxlen = i + 1
        if sum in my_dict:
            maxlen = max(maxlen,
                         i - mydict[sum])
        else:
            mydict[sum] = i
    return maxlen
```

T(n) = O(n)
Sc = O(n)

longest common subarray with given sum.

We are given two binary subarray of same sizes.

i/p: arr1[] = [0, 1, 1, 0, 0, 0]
arr2[] = [1, 0, 1, 0, 0, 1]
o/p: 4

i/p: arr1[] = [0, 1, 0, 1, 1, 1, 1]
arr2[] = [1, 1, 1, 1, 1, 0, 1]
o/p: 6

Naive

```
def longestCommonSpan(arr1, arr2):
    res = 0
    n = len(arr1)
    for i in range(n):
        sum1 = 0
        sum2 = 0
        for j in range(i, n):
            sum1 += arr1[j]
            sum2 += arr2[j]
            if sum1 == sum2:
                res = max(res, j - i + 1)
    return res
```

{ TC = O(n2), SC = O(1) }

Efficient solution

The problem is going to reduce into the problem of longest subarray with 0 sum in an array.

```
def longestSpan(arr1, arr2):
    n = len(arr1)
    arr = [0]*n
    for i in range(n):
        arr[i] = arr1[i] - arr2[i]
    mydict = dict()
    sum = 0
    max_len = 0
    for i in range(n):
        sum += arr[i]
        if sum == 0:
            max_len = i+1
        if sum in mydict:
            max_len = max(max_len, i-mydict[sum])
        else:
            mydict[sum] = i
    return max_len
```

TC = O(n)
SC = O(n)

① Compute a difference array.

```
int temp[n]:
for (int i=0; i<n; i++)
    temp[i] = arr1[i] - arr2[i];
```

eg

arr1[] = [0,1,0,0,0,0]
arr2[] = [1,0,1,0,0,1]
temp[] = [-1,1,-1,0,0,-1]

② Return length of the longest subarray with 0 sum in temp.

Value in temp[]
- * we get 0 when values are same in both
- * we get 1 when arr1[i]=1 and arr2[i]=0
- * we get -1 when arr1[i]=0 and arr2[i]=1.

longest Consecutive Subsequence

Given an array, we need to find the longest subsequence that has consecutive element. These consecutive element may appear in any order in the subsequence.

i/p: arr[] = [1,9,3,4,2,20]
o/p: 4

i/p: arr[] = [8,20,7,30]
o/p: 2.

Naive

```
def longest(arr):
    n = len(arr)
    arr.sort()
    res = 1
    curr = 1
    for i in range(1,n):
        if (arr[i] == arr[i-1]+1):
            curr += 1
        else:
            res = max(res, curr)
            curr = 1
    res = max(res, curr)
    return res
```

TC = O(nlogn)
SC = O(1)

Efficient

```
def longest(arr):
    S = set()
    res = 0
    for i in arr:
        S.add(i)
    for i in arr:
        if (i-1) not in S:
            curr = 1
            while (curr+i) in S:
                curr += 1
            res = max(res, curr)
    return res
```

TC = O(n)
SC = O(n)

Given an array of size n and an integers K, find all element in the array that appear more than n/K times.

Eg: i/p: l = [3,1,2,2,1,2,3,3] , k=4
o/p: [2,3] Here n/k is 8/4 = 2 ; therefore 2 appear 3 times, 3 appear 3 times } greater than 2

Naive
[10,10,10,20,30,20,10] k=2 , n=7, n/k=3

M-1 ① Sort the given array/list
[10,10,10,10, 20,20,30]
② Traverse through the sorted
10: 4 Print (10)
20: 2
30: 1

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

```
def PrintNK (arr, K):
    arr.sort()
    i = 1
    count = 1
    while (i < len(arr)):
        while (i < len(arr) and arr[i] == arr[i-1])
            count += 1
            i += 1
        if (count > n//K):
            print (arr[i-1], end = " ")
        count = 1
        i = i+1
```

TC = O(nlogn)
SC = O(1)

M-2

```
from collections import Counter        # Class in Python
def PrintNK (arr, K):
    C = Counter(arr)
    for i in C:
        if C[i] > n//K :
            print (i)
```

M-3

```
def PrintNK (arr, K):
    n = n//K
    d = { }
    for i in range(n):
        if arr[i] in d:
            d[arr[i]] += 1
        else:
            d[arr[i]] = 1
    for i in d:
        if (d[i] > n):
            print (i)
```

TC = O(n)
SC = O(n)

Count distinct element in every window

i/p: [10, 20, 20, 10, 30, 40, 10] , k=4
o/p: 2 3 4 3

i/p: [10, 10, 10, 10] k=3
o/p: 1 1

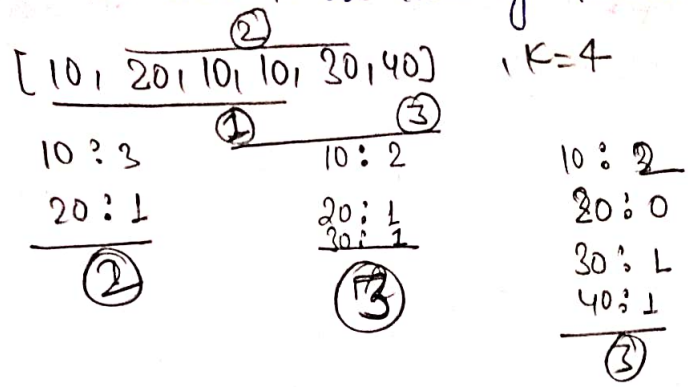
Naive Approach ① There will be (n-k+1) windows [10, 20, 10, 30, 40] , k=3 ; (5-3+1) = 3 windows
② Traverse through every window and count distinct element in it.

```
def PrintDistinct (arr, K):
    n = len (arr)
    for i in range (n-k+1):
        print (len (set (arr[i:i+k])))
```

TC = O((n-k+1)*k)
SC = O(k)

Efficient

The idea is to use count of previous window to get the current count.



from the collections i'm using Counter

```
from collections import Counter
def countdist(arr,k):
    mp = Counter(arr[0:k])
    print(len(mp))
    for i in range(k,len(arr)):
        n = arr[i-k]
        mp[n] -= 1
        mp[arr[i]] += 1
        if(mp[n]==0):
            del mp[n]
        print(len(mp))
```

first window

TC=O(n)

SC=O(k)

remove first item of prev window

Add last item of current window.

longest subarray with given sum.

Given an array arr[] of size n containing integers, The problem is to find the length of the longest sub-array having sum equal to the given value K.

eg Naive

arr[] = {10, 5, 2, 7, 1, 9} , K=15

o/p: 4 → subarray is {5, 2, 7, 1}

```
def longestsum(arr, sum):
    res = 0
    for i in range(len(arr)):
        curr = 0
        for j in range(i, len(arr)):
            curr += arr[j]
            if (curr == sum):
                res = max(res, j-i+1)
    return res
```

TC=O(n2)

Efficient

```
def longestsum(arr, sum):
    n = len(arr)
    mydict = dict()
    Pre-sum = 0
    res = 0
    for i in range(n):
        Pre-sum += arr[i]
        if Pre-sum == sum:
            res = i+1
        if Pre-sum not in mydict:
            mydict[Pre-sum] = i
        if Pre-sum - sum in mydict:
            res = max(res, i-mydict[Pre-sum - sum])
    return res
```

TC= O(n)

SC=O(n)