HASHING

- Introduction:-

- We discover the unparalleled effectiveness of Hashing as the optimal method for data storage in the majority of scenarios, boasting an impressive O(1) complexity for operation such as Search, Insert & Delete.

Search
Insert
Delete
} O(1) on average.

- Not useful for:-

1) finding closest value
2) Sorted Data.
3) Prefix Searching (Trie)

} for this operation we use AVL Tree or Red. Black tree.

- Application of Hashing:- (After array most used data structure)

1. Dictionaries
2. Database Indexing
3. Cryptography
4. Caches
5. Symbol Tables in Compilers/Interpreters
6. Routers
7. Getting data from databases
8. Many more ....

Direct Address Table

- Imagine a situation where you have 1000 keys with values from (0 to 999), how would you implement following in O(1) time.

1) Search
2) Insert
3) Delete.

0 if we get a insert function we go to that index & make it 1 & if we get a delete function we get to that index & make it to 0. for search function we go to that index & if it is 1 the it is present if it is 0 then it is not present

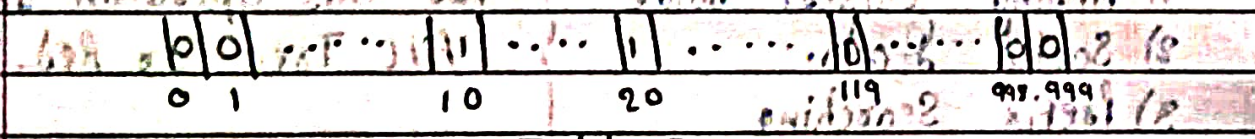


Table [ ]

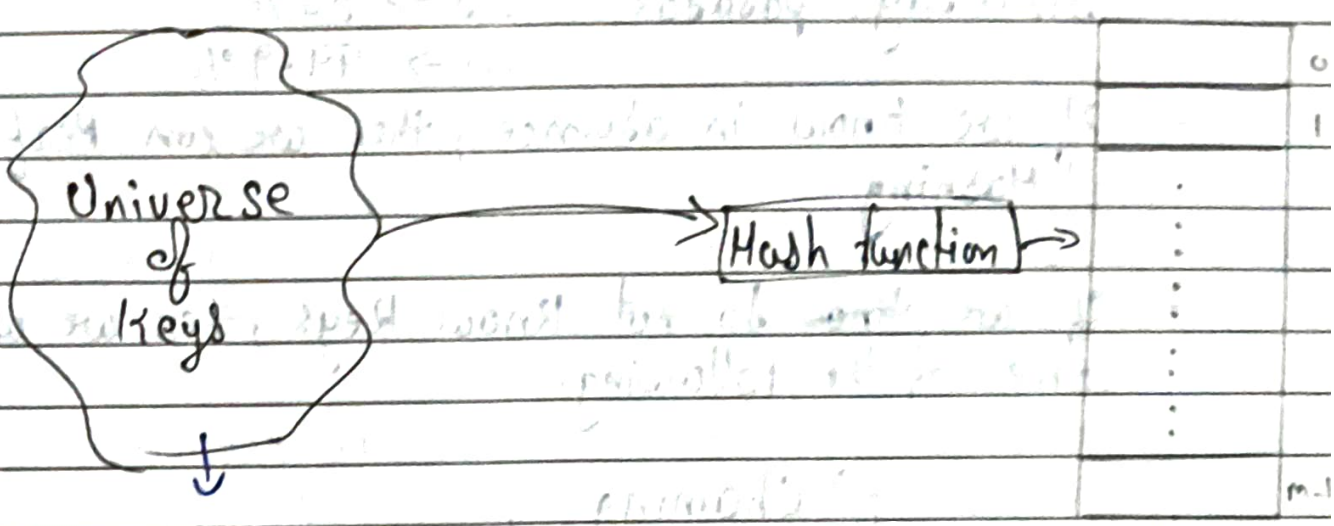
example operation:

insert (10)
insert (20)
insert (119)
Search (10)
Search (20)
delete (119)
Search (119)

- insert (int i) { Table [i] = 1; }
- delete (int i) { Table [i] = 0; }
- Search (int i) { return Table [i]; }

There are problem with direct address if input. We have used array. what if the number we are going to store is a phone number then this kind of array does not support that large number.

- **Hashing function:-**



Hash Table

It can be :-
- Phone Numbers
- String
- Employee ID [E1021D]

⇒ How Hash function work?
- Should always map a large key to some small key.
- Should generate values from 0 to m-1.
- Should be fast, O(1) for integer & O(len) for string of length 'len'.
- Should uniformly distribute large key into Hash table slots.

⇒ **Example Hash function:-**

Best value will be power of 2 or 10 of m. It is depended of the input in the Hash table

① h(large_key) = large_key % m ← How many element you want to insert.

② for string, weighted sum (usually we take prime number)
x = 33 str[] = "abcd"
(str[0] * x^0 + str[1] * x^1 + str[2] * x^2 + str[3] * x^3) % m

③ Universal Hashing.

- **Collision Handling:-**

- Birthday paradox 23 → 50%
  70 → 99.9%

- If we know in advance, then we can Perfect Hashing.

- If we ~~know~~ do not know keys, then we use one of the following.
  - Chaining
  - open Addressing
    - linear probing
    - Quadratic probing
    - Double Hashing

- **Chaining:-** (collision Handling technique)

Idea:- We will maintain a array of linked list

hash(key) = key % 7

Key = {50, 21, 58, 17, 15, 49, 56, 22, 23, 25}

0	21	→ 49 → 56
1	50	→ 15 → 22
2	58	→ 23
3	17	
4	25	
5		
6		

Hash Table (Array of linked list Headers)

⇒ **Performance:-**

m = No. of slots in Hash Table
n = No. of Keys to be inserted.
load factor (α) = n/m
Expected chain length = α
Expected Time for search = $O(1+\alpha)$
Expected Time to Insert or Delete = $O(1+\alpha)$

Note:- This is all under the assumption when you have uniform distribution of Keys in the Hash table.

⇒ **Data structures for storing chains**

① Linked list
② Dynamic Sized Array (Vector in C++)
③ Self Balancing BST (AVL Tree, Red Black Tree

→ {Search O(1), Delete O(1), Insert O(1)}

→ {Search O(log l), Delete O(log l), Insert O(log l)}

- **Implementation of chaining:-**

Bucket = 7

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

Insert → 70, 71, 9, 56, 72

0	70 → 56
1	71
2	9 → 72
3	NULL
4	NULL
5	NULL
6	NULL

Search(56) ⇒ True
Search(57) ⇒ false

⇒
```
struct MyHash
{
 int Bucket;
 list <int> *table;
 myhash (int b)
 { Bucket = b;
   table = new list <int> [b];
 }
 void insert (int key) {
 int i = key % Bucket;
 table [i].push_back (key); }

 Void remove (int key) {
  int i = key % Bucket;
  table [i].remove (key); }
```

```
bool Search (int key) {
 int i = key % Bucket;
 for (auto x : table[i])
  { if (x == key)
      return true; }
    return false; }
```

- **Open Addressing :-** (Another way to handel collision)

⇒ No. of slots in Hash Table ≥ No of keys to be inserted
⇒ Cache friendly.

i) **linear probing :-** linearly search for next empty slot when there is collision.

key = { 50, 51, 49, 16, 56, 15, 19 }

hash(key) = key % 7

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

hash(key, i) = (key % hashsize + i) % hash size.

↓

formula for linear probing & handels collisions well.

linear Probing :- hash(key, i) = (key % hashSize + i) % (hash size)

- How to handle deletion in open addressing?

Search:- We compute hashfunction. we go to that index & compare. If we find, we return true. Otherwise we linearly Hash. we stop when one of the three cases arises
① Empty slot ② Key found ③ & Reached where we have started.

- How to handle deletion in one addressing?

0	
1	50
2	51
3	15
4	
5	
6	

Insert(50), Insert(51), insert(15), Search(15), search(51);

hash(key) = key %7

- So lets delete(51). Let's make that slot empty what are the problems

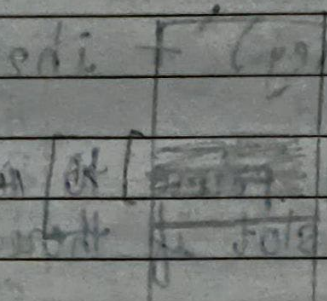
0	
1	50
2	
3	15
4	
5	
6	

Since we have made slot empty so if some how user wish to find (15) it will stop at index 2 since it sees an empty slot.

So, let's make it Deleted slot. so the search function does not stop when it see a Deleted slot & Insert function can insert in the Deleted slot.

0	
1	50
2	Deleted
3	15
4	
5	
6	

- Clustering (A problem with linear probing)



linear probing :- $hash(Key, i) = (h(Key) + i) \% 7$

$h(Key) = Key \% 7$

How to handle clustring problem with linear probing?

① Quadratic probing (secondary cluster)

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

② Double Hashing

$hash(Key, i) = (h_1(Key) + i * h_2(Key)) \% m.$

Qudratic Probing :- hash(key, i) = (key % hashsize + i + i^2) % hash size.

- Problem with quadratic probing :
  - It cannot find a empty slot even if there is an empty slot.
  - It only work if
    - ① $\alpha < 0.5$
      - ↳ n/m
    - ② m is a prime.

- Dable Hashing :-

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

① If $h_2(key)$ is relatively prime to m, then it always find a free slot if there is one

② Distribute Keys more uniformly than linear probing & quadratic hashing).

③ No clustering.

double hashing Probing :- hash(key, i) = $(h_1(key) + i \cdot h_2(key)) \%$ hashsize.

- $h_1(key)$ = key % hashsize ⇒ The primary hash function.
- $h_2(key)$ → The secondary hash function (must never return 0!).
- i → The probe number (0, 1, 2, ...)

$h_2(Key) = R - (Key \% R)$

R is prime number less than hash size

Date
Page 10

example:-

$Key = [49, 63, 56, 52, 54, 48]$

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

$m = 7$

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

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

↳ h_2 function should not return zero.

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

for first Key just find $h_1(Key)$

If collision happen then find $h_2(Key)$ for second position.

63:- collision happen so we calculate h_2

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

$= 6 - (63 \% 6)$

$= 6 - 3$

$= 3$

i will we 1 since we are handling collision once.

$h_1(Key) = 63 \% 7 = 0$

$\therefore (0 + 1 \times 3) \% 7 = 3$

- for 56, so again collision we compute h_2

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

$= 6 - (key \% 6)$

$= 6 - (56 \% 6)$

$= 6 - 2$

$= 4$

$h_1(key) = 0$

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

$= (0 + 1 \times 4) \% 7$

$= 4$

So insert 56 at 4

- for 52

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

$m = 7$

$h_1(key) = (key \% 7) = 3$

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

$= 6 - (52 \% 6)$

$= 6 - 4$

$= 2$

$hash(key, i) = (3 + 2) \% 7 = 5$

- for 54

hash(key, i) = (h1(key) + i h2(key)) % m

h1(key) = (key % 7) = 5
h2(key) = 6 - (key % 6)
= 6 - (54 % 6)
h2(key) = 6

has (key, i) = (5 + 6) % 7
= 4 {another collicion}.

now i = 2

hash(key, i) = (5 + (6 * 2)) % 7
= 3 {another collision}

now i = 3

hash (key, i) = (5 + (6 * 3)) % 7
= 2 {insert the value}

- for 48

hash(key, i) = (h1(key) + i h2 (key)) % m

h1(key) = (key % 7)
= 48 % 7 = 6 {insert}

- Requirement of Double Hashing :-

Why $h_2(key)$ & m should be relatively prime?

$(1 \times 6) \% 7 = 6$
$(2 \times 6) \% 7 = 5$
$(3 \times 6) \% 7 = 4$
$(4 \times 6) \% 7 = 3$
$(5 \times 6) \% 7 = 2$
$(6 \times 6) \% 7 = 1$

} so it checks all the possible slots.

Algorithm :-

```
void doubleHashingInsert(Key) {
    if (table is full)
        { return error; }
    probe = h1(key), offset = h2(key);
    // offset = 1 in linear probing.
    while (table[probe] is occupied)
        probe = (probe + offset) % m;

    table[probe] = key;
}
```

- Performance Analysis of search (unsuccessful)

$\alpha = n/m$ (should be ≤ 1)

Assumption:- Every probe sequence looks at a random location.

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

Expected No. of probes required $= \frac{1}{1-\alpha}$

$\alpha = 0.8, 5$

$\alpha = 0.9, 10$

- Implementation of open Addressing:-

```
myHash mh(7);
mh.insert(45);
mh.insert(56);
mh.insert(72);
if(mh.search(56)==true)
    print("yes")
else
    Print("No")

mh.erase(56);
if(mh.search(56)==true)
    print("yes")
else
    Print("No")
```

0	1	2	3	4	5
49	56	72	-1	-1	-1

erase(56)

0	1	2	3	4	5
49	-2	72	-1	-1	-1

```
struct MyHash {
int *arr;
int cap, size;
MyHash (int c) {
    cap = c
    size = 0;
for (int i=0; i<cap; i++)
        arr[i] = -1;
  }
  int hash (int key)
   { return key % cap; }

bool search (int key) {
  int h = hash(key);
  int i = h;
 while (arr[i] == key)
{ if (arr[i] == key)
      {return true;}
  i = (i+1) % cap;
  if (i == h)
     {return false;}
}
   return false;
  }
```

```
bool insert (int key) {
 if (size == cap)
      { return false; }
int i = hash(key);
while( arr [i] != -1 && arr [i] != 2 && arr [i] != key
    i = (i+1) % cap;
if (arr [i] == key)
   { return false; }
else
   { arr [i] = key;
     size++;
     return true; }
};

bool erase (int key) {
 int h = hash(key);
 int i = h;
 ~~while (arr [i] == key)~~
 while (arr [i] != -1)
     { if (arr [i] == key)
         { arr [i] = -2;
           return true; }
       i = (i+1) % cap;
      if (i == h)
         return false;
   }
   return false;
   }
};
```

- How to handle the cases when -1 & -2 are input keys?

⇒ In library implementation you don't use actual key to represent (-1 or -2) for empty or deleted they use null pointer So to know its empty or deleted.
B [Dummy node]

- Chaining v/s open Addressing

-

Chaining	open Addressing
1) Hash Table never fills	1) Table may become full & ~~re~~resizing becomes mandatory.
2) less sensitive to Hash function	2) Extra care required for clustring
3) Poor cache performance.	3) Cache Friendly
4) Extra space for links	4) Extra space might be needed to achieve some performance as chaining
5) Performance of chaining $(1+\alpha)$ 1.9	5) Performance of ~~Hash~~ open chaining $\left(\frac{1}{1-\alpha}\right) = \frac{1}{1-0.9} = \frac{1}{0.1} = 10$

Unordered Set in C++ STL.

It uses Hash internally.

operation:-
insert(), begin(), end(), size(), clear(), find()

example of insert():-

```
#include <iostream>
#include <unordered_~~map~~set>
using namespace std;
    int main() {
unordered_set<int> S;
 S.insert(10);
 S.insert(5);
 S.insert(15);
 S.insert(20);

for(int x : S)
    { cout << x << " "; }
return 0;
 }
```

More example in diary.

O/P:- Permutation of (10 5 15 20)

* See diary & see all the operation of unordered_set() *

(See diary)

- **Unordered_map in C++ STL:**
  - → Used to store key, value pairs
  - → Uses Hashing
  - → No order of key.
  - → Uses Hash internally.
- example:- [] ⇒ It is used for accessing if present & inserting if not present

```
#include <iostream>
#include <unordered_map>
using namespace std;
int main() {
unordered_map <string, int> m;
m["gfg"] = 20;
m["ide"] = 30;
m.insert ({"courses", 15});

for(auto x : m)
  {cout<<x.first<<" "<<x.second<<endl;}

return 0;}
```

O/P:- permutation of
```
gfg     20
courses 15
ide     30
```

Count Distinct Elements :-

I/P :- arr[] = {15, 12, 13, 12, 13, 13, 18}
O/P :- 4

Naive Soln :-

```
int countDist(int arr[], int n) {
 int res = 0;
for(int i=0; i<n; i++)
 { bool flag = false;
  for(int j=0; j<i; j++)
    { if (arr[i] == arr[j])
      { flag = true;
        break;
      }
    }
    if (flag == false)
        res++;
  }
   return res;
 }
```

Efficient Sol^n :-

Use unordered_set or unordered_map:
we will use unordered_set.

```
int countDistinct (int arr[], int n)
  {
     unordered_set <int> S;
     for (int i=0; i<n; i++)
       { S.insert (arr[i]); }
// It does not increase the count if key is already present.
     return S.size();
  }
```

Dry run :- { 10, 20, 10, 20, 30 }

i=0 = S { 10 }
i=1 = S { 10, 20 }
i=2 = S { 10, 20 }
i=3 = S { 10, 20 }
i=4 = S { 10, 20, 30 }

Time :- O(n)

Aux Space :- O(n)

We can just use this

```
int countDist (int arr[], int n)
{ unordered_set <int> S (arr, arr+n);
     return S.size(); }
```

Time - O(n)
Space - O(n)

- Frequencies of Array element:-

I/P:- arr[] = {10, 12, 10, 15, 10, 20, 12, 12}
O/P:- 10 3
12 3
15 1
20 1

I/P:- arr[] = {10, 10, 10, 10}
O/P:- 10 4

- Naive Soln:-

```
void printfreq (int arr[], int n)
{ for(int i=0; i<n; i++)
  { bool flag = false;
    for (int j=0; j<i; j++)
      { if(arr[i]==arr[j])
        {flag = true;  break; }
      if (flag == true) }
        continue;
    int freq = 1;
    for (int j=i+1; j<n; j++)
      { if (arr[i]==arr[j])
          freq++;
      cout << arr[i] << " " << freq << endl;
  }
}
```

Check if seen before

If seen then ignore.

If not seen before count freq.

- Efficient :-

We will use unordered_map {key, value}

{10, 20, 20, 10, 30, 10}

h = { } // Empty hash map.

i=0 : h = { (10,1) }
i=1 : h = { (10,1), (20,1) }
i=2 : h = { (10,1), (20,2) }
i=3 : h = { (10,2), (20,2) }
i=4 : h = { (10,2), (20,2), (30,1) }
i=5 : h = { (10,3), (20,2), (30,1) }

```
int countfreq(int arr[], int n)
{
    unordered_map<int, int> h;
    for(int i=0; i<n; i++)
        {h[arr[i]]++;}
    for (auto e : h)
        {cout << e.first << " " << e.second << endl;}
}
```

Intersection of Two Unsorted Array:-

I/P:- a[] = {10, 15, 20, 25, 30, 50}
b[] = {30, 5, 15, 80}
O/P:- 15 30

I/P:- a[] = {10, 20}
b[] = {20, 30}
O/P:- 20

⊛ Naive Soln

```
void intersection (int a[], int m, int b[], int n)
{
 for(int i=0; i<m; i++)
   { bool flag = false;
     for(int j=0; j<n; j++)
     { if(a[i] == b[j])
       { flag = true;
         break; }
     }
     if (flag == true)
       {cout<<a[i]<<" "; }
   }
}
```

- Efficient Sol^n :-

```
void intersect(int a[], int b[], int m, int n)
{ unordered_set<int> S(b, b+n);
    for(int i=0; i<m; i++)
      { if(S.find(a[i] != S.end())
          { cout<<a[i]<<" "; }
      }
}
```

- Union of Two Unsorted Array :-

I/P :- a[] = {15, 20, 5, 15}
b[] = {15, 15, 15, 20, 10}
O/P :- 4

I/P :- a[] = {10, 12, 15}
b[] = {18, 12}
O/P :- 4

- Naive Sol^n :- O(m x n + m x (m+n))

1) Initialize : res = 0
2) Create an auxiliary array dist[]
3) Traverse through a[]. for every element in a[].
check if it is present in dist[].
If no, a) Increment res
b) Append the element to dist[].
4) Repeat step 3 for every element for b[].

⊛ Efficient Sol^{n}:- O(m+n)

1) Create a empty hash table, h
2) Insert all elements of a[] in h. O(m)
3) Insert all elements of b[] in h. O(n)
4) Return h.size().

```
int ~~u~~ union(ount (int a[], int b[], int m, int n)
 [ unordered_set <int> h(a, a+m);
    for(int i=0; i<n; i++)
      [ h.insert(b[i]); }
    return h.size();
  }
```

- Pair with given Sum:-

I/P:- arr[] = {3, 2, 8, -15, -8}
Sum = 17
O/P:- True.

I/P:- arr[] = {2, 1, 6, 3}
Sum = 6
O/P:- false.

Date
Page

(*) Naive :- Time :- $O(n^2)$, space :- $O(1)$

1) Consider all possible pairs one by one & check for sum.

arr[] = {8, 3, 9, 4}

Sum = 13.

pairs :- (8, 3), (8, 9), (8, 4), (3, 9), (3, 4), (9, 4).

2) If we find a pair, return true. else return false.

Total pairs = $\left(\frac{n(n-1)}{2}\right)$ pairs.

(*) Efficient Sol" :- Time :- $O(n)$, Aux Space :- $O(n)$

Note :- Inserting everting into a hash table then traversing through the array : does not work.

```
bool isPair(int arr[], int n, int sum)
{ unordered_set<int> h;
    for (int i=0; i<n; i++)
    { if (h.find(sum - arr[i]) != h.end())
        return true;
      else
        h.insert(arr[i]);
    }
    return false;
}
```

- Subarray with 0 sum:-

I/P:- arr[] = {1, 4, 13, -3, -10, 5}
O/P:- yes.

I/P:- arr[] = {-1, 4, -3, 5, 1}
O/P:- yes.

⊛ Naive Soln:-

```
bool isOSubarray (int arr[], int n)
{ for (int i=0; i<n; i++)
   {int curr_sum=0;
   for (int j = i; j<n; j++)
     { curr_sum += arr[j];
       if (curr_sum == 0)
         return true;
     }
   } return false;
  }
```

Time :- O(n²)

⊛ Efficient:- We can use Prefix sum & Hashing

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}
Already present in hash
return trees.

```
bool isOSubarray (int arr[], int n) {
  unordered_set <int> h;
  int pre_sum = 0;
  for(int i=0; i<n; i++)
  { pre_sum += arr[i];
    if (h.find(pre_Sum) != h.end())
        { return true; }
    if (pre_sum == 0)
        { return true; }
    h.insert (pre_sum);
  }
  return false;
}
```

[-3, 4, -3, -1, 1] 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, -3}
i=3 : pre_sum = -3,
Already present in h, return true.

Subarray with given Sum:-

I/P:- arr[] = {5, 8, 6, 13, 3, -1} , sum = 22
sum = 22

O/P:- True:

⊛ Naive Soln:

```
bool isSubArrSum (int arr[], int n, int sum)
{ for(int i=0; i<n; i++)
  { int curr_sum=0;
    for(int j=i; j<n; j++)
    { curr_sum += arr[j];
      if(curr_sum == sum)
        { return true; }
    }
  }
  return false;
}
```

⊛ Efficient Soln:

Pre_Sum2 = Pre_Sum1 + Sum

a_0 a_1 a_2 ··· a_i a_{i+1} ··· a_j ··· a_{n-1}

We need to check "Prefix_Sum - Sum" exists among prefix sum.

i=0: pre_sum = 5, h = {5}
i=1: pre_sum = 13, h = {5, 13}
i=2: Pre_Sum = 19, h = {5, 13, 19}
i=3: pre_sum = 32, h = {5, 13, 19, 32}
i=4: pre_sum = 35, the value "35-22" is present in h.
So we return true.

- bool isSum (int arr[], int n, int sum)
{ unordered_ {int} h;
int pre_sum = 0;
for (int i=0; i<n; i++) {
pre_sum += arr[i];
if (pre_sum == sum) return true;
if (h.find (pre_Sum - Sum) != h.end())
{return true;}
h.insert (pre-sum);
}
return ~~true~~, false; }

Time:- O(n) Aux space O(n)

- Longest Subarray with given sum:-

1) I/P:- arr[] = {5, 8, -4, -4, 9, -2, 2}
Sum = 0
O/P:- 3

2) I/P:- arr[] = {3, 1, 0, 1, 8, 2, 3, 6}
Sum = 5
O/P:- 4

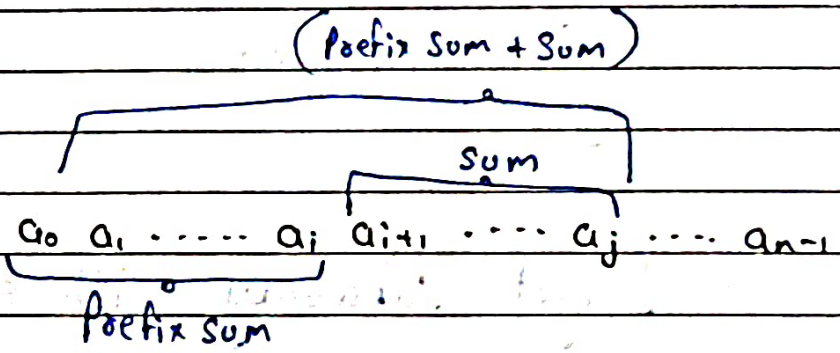
3) I/P:- arr[] = {8, 3, 7} => No subarray
Sum = 15
O/P:- 0

Naive Sol^n:-

```
int maxlen(int arr[], int n, int sum)
{ int res=0;
  for(int i=0; i<n; i++)
  { int curr_sum=0;
    for(int j=i; j<n; j++)
    { curr_sum += arr[j];
      if(curr_sum == sum)
        { res = max(res, ~~curr_sum~~ j-i+1);
    }
  }
  return res;
}
```

Time :- $\theta(n^2)$
Space :- $O(1)$

Efficient Sol^n:-



we will use map:-

arr[] = {8, 3, 1, 5, -6, 6, 2, 2}, Sum = 4

m = { } (Initially)

m = {(8,0)} res = 0

m = {(8,0), (11,1)} res = 0

m = {(8,0), (11,1), (12,2)} res = 2

m = {(8,0), (11,1), (12,2), (17,3)} res = 2

m = " " "

m = " " "

m = {(8,0), (11,1), (12,2), (17,3), (19,6)} res = 2

m = {(8,0), (11,1), (12,2), (17,3), (19,6), (21,7)} res = 4.

```
int maxLength(int arr[], int n, int sum)
{ unordered_map<int, int> m;
  int pre_sum = 0, res = 0;
  for(int i=0; i<n; i++)
  { pre_sum += arr[i];
    if(pre_sum == sum)
      { res = i+1; }
    if(m.find(Pre_Sum) == m.end())
      { m.insert({pre_sum, i}); }
    if(m.find(pre_sum - sum) != m.end())
      { res = max(res, i-m[pre_sum - sum]); }
  }
  return res;
}
```

Time: O(n)
Space: O(n)

- Longest Subarray with equal 0s & 1s

We are given a binary array.

1) I/P:- arr[] = {1, 0, 1, 1, 1, 0, 0}
O/P:- 6

2) I/P:- arr[] = {1, 1, 1, 1, 1}
O/P:- 0

3) I/P:- arr[] = {0, 0, 1, 1, 1, 1, 1}
O/P:- 4

4) arr[] = {0, 0, 1, 0, 1, 1}
O/P:- 6

(*) Naive Soln:-
We consider all subarray we count and check it contains equal no of 0's & 1's & then we compare the current length with maximum length.

```
int longestSub(int arr[], int n) {
int res=0;
for(int i=0; i<n; i++)
 { int c0=0, c1=0;
   for(int j=i; j<n; j++)
    { if(arr[j]==0)
        c0++;
      else
        c1++;
     if(c0==c1)
      {res = max(res, c0+c1);}
    }
 } return res;
 }
```

Time:- $O(n^2)$
Space:- $O(1)$

(*) Efficient Soln:- Base of length of the longest subarray with given sum.

1) Replace every 0 with -1.

```
for(int i=0; i<n; i++)
 { if(arr[i]==0)
    {arr[i]=-1;} }
```

$O(n)$ time

2) Now call the function to find length of the largest subarray with 0 sum

$O(n)$ time
$O(n)$ space.

- **Longest common Subarray with Given Sum:-**

We are Given two binary array of some size. We have to find longest subarray with common values.

1) I/P:- arr1[] = {0, 1, 0, 0, 0, 0}
arr2[] = {1, 0, 1, 0, 0, 1}
O/P:- 4

2) I/P:- arr1[] = {0, 1, 0, 1, 1, 1, 1}
arr2[] = {1, 1, 1, 1, 0, 1}
O/P:- 6

④ Naive Sol^n:-

```
int maxCommon(int arr1[], int arr2[], int n)
{ int res=0;
  for(int i=0; i<n; i++)
  {int sum1=0, sum2=0;
   for(int j=i; j<n; j++)
   { sum1 += arr1[j];
     sum2 += arr2[j];
     if(sum1==sum2)
        {res = max(res, j-i+1); }
   }
}
  return res;
}
```

Time :- $O(n^2)$
Space :- $O(1)$

Efficient Sol:-

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

1) Compute a difference array.

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

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

Values in temp[]
- → We get 0 when values are same in both
- → We get 1 when arr1[i]=1 & arr2[i]=0
- → We get -1 when arr1[i]=0 & arr2[i]=1.

Date
Page

- Longest Consecutive Subsequence:-

1) I/p:- arr[] = {1, 9, 3, 4, 2, 20}
O/p:- 4

2) I/p:- arr[] = {8, 20, 7, 30}
O/p:- 2

3) I/p:- arr[] = {20, 30, 40}
O/p:- 1

We need to find the longest subsequence in the form of $x, x+1, x+2, \ldots x+i$ ~~toh~~ with these elements appearing in any order.

(*) Naive Soln:- We use Sorting.

```
int longestSub (int arr[], int n)
{ sort (arr, arr+n);
  int res=1, curr=1;
  for (int i=1; i<n; i++)
  { if (arr[i] == arr[i]+1) { curr++; }
    else if (arr[i] != arr[i-1])
      { res = max(res, curr);
        curr = 1; }
  } return max (res, curr);
```

Time:- O(n log n)

⊛ Efficient Method :-

Hint :- ⊛ We first insert all elements in a hash table
⊛ Then with 2n lookups, we find the result.

arr[] = {1, 3, 4, 3, 3, 2, 9, 10}
h = {1, 3, 4, 2, 9, 10}

```
int longestSub (int arr[], int n)
{ unordered_set <int> h (arr, arr+n);
  int res = 1;
  for(auto x : h)
  { if(h.find(x-1) == h.end())
    { int curr = 1;
      while (h.find(x+curr) != h.end())
      {curr++;}
      res = max(res, curr); }
    }
  }
  return res;
}
```

Time :- O(n)
Space :- O(n)
There are always 2n lookup.

for first element :- 2 + (len - 1)
for other elements :- 1

Date
Page

Count Distinct Element in every window

1) I/P:- arr[] = {10, 20, 20, 10, 30, 40, 10}
K = 4
O/P:- 2 3 4 3

2) I/P:- arr[] = {10, 10, 10, 10}
K = 3
O/P:- 1

K <= n

No. of windows of size K will be ⇒ (n-K+1)

* Naive Soln:- Time (O((n-K)*K*K))

```
void printDistinct (int arr[], int n, int k)
  { for(int i=0; i<=n-K; i++)
    { int count = 0;
      for(int j=0; j<k; j++)
      { bool flag = false;
        for(int p=0; j<j; p++)
        { if (arr[i+j] == arr[i+p])
          { flag = true; break; }
        if (flag == false)
          { count++; }
      }
      cout << count << " ";
    }
  }
```

(*) Efficient Soln:- O(n)

1) Creati a frequency map of first K item
freq[10] = 3, freq[20] = 1.
2) Print size of the frequency map.
3) for(int i=K; i<n; i++)
a) Decrease frequency of arr[i-K].
b) If the frequency of arr[i-K] become 0, remov
it from map.
c) If arr[i] does not exist in the map, insert i
Else increase its frequencies in the map
d) print size of the map

=> Code:-

• More than n/k Occurences :-

1) I/P:- arr[] = {30, 10, 20, 20, 10, 20, 30, 30}
(1 2 3 4 5 6 7 8)
K = 4
O/P:- 20 30

Note:- n = 8 & n/k = 2

2) I/P:- arr[] = {30, 10, 20, 30, 30, 40, 30, 40, 30}
(1 2 3 4 5 6 7 8 9)
K = 2
O/P:- 30

Note:- n = 9 , n/k = 4.5 (5 or 5↑)

(*) Naive Soln:- Use sort

```
void printNByK(int arr[], int n, int k)
{ sort(arr, arr+n);
  int i=1, count=1;
  while(i<n)
  { while(i<n && arr[i]==arr[i-1])
    { count++;
      i++; }
    if(count > n/k)
      { print(arr[i-1] + " "); }}

    count=1
    i++;
  }
}
```

Time: O(nlogn)

* Efficient Sol":- Use unordered_map.

```
void printNByK(int arr[], int n, int k)
{
    unordered_map<int, int> m;
    for(int i=0; i<n; i++)
        { m[arr[i]]++; }          (Time :- O(n))
    for(auto e : m)
        if (e.second > n/k)
            cout<<e.first<<" ";
}
```

arr[] = {10, 20, 30, 10, 10, 20}
m = {(10, 3), (20, 2), (30, 1)}

⇒ for k : too small compare to n then this is inefficient. Solution for that.

{30, 10, 20, 20, 20, 10, 40, 30, 30}
k=4, n=9.

Let 'res_count' be the number of elements in the result.

res_count ≤ k-1

Extension of Moore voting algorithm.

Steps:-

Phase 1

① Create an empty map m.

② for(i=0; i<n; i++)

a) if(m contains arr[i])

m[arr[i]]++;

b) Else if m.size() is less than (k-1)

m.put(arr[i], 1)

c) Else

Decrease all values in m by one.
If value become 0, remove.

Phase 2

③ for all elements in m, print the elements that actually appears more that n/k time.

{30, 10, 20, 20, 20, 10, 40, 30, 30}

i=0: m = {(30, 1)} 2.b

i=1: m = {(30, 1), (10, 1)} 2.b

i=2: m = {(30, 1), (10, 1), (20, 1)} 2.b

i=3: m = {(30, 1), (10, 1), (20, 2)} 2.a

i=4: m = {(30, 1), (10, 1), (20, 3)} 2.a

i=5: m = {(30, 1), (10, 2), (20, 3)} 2.a

i=6: m = {(10, 1), (20, 2)} 2.c

i=7: m = {(10, 1), (20, 2), (30, 1)} 2.b

i=8: m = {(10, 1), (20, 2), (30, 2)} 2.a.

- How does the approach work?

{30, 10, 20, 20, 20, 10, 40, 30, 30}

final map : {(10, 1), (20, 2), (30, 2)}

10, 30,
20, 40

Rejected

40 Rejected it self
& three other.

10, 20,
20, 30,
30

In rejected set an element rejects. (k-1) distinct others.