Hashing

used to implement dictionaries ⇒ have ⇒ key value pairs
" " " sets ⇒ keys

Search
Insert → O(1) Average
delete

Not useful for :-

1) finding closest value
2) keeping sorted data
] AVL Tree (DS) Red-Black Tree
3) prefix searching] Trie DS.

Applications of Hashing :

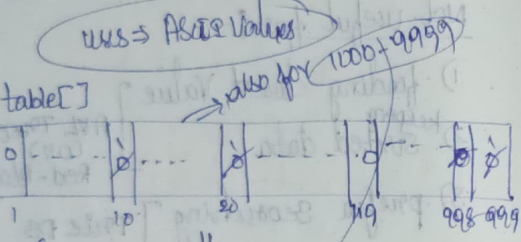
1. Dictionaries.
2. Database Indexing.
3. Cryptography.
4. caches.
5. Symbol Tables in compilers / Interpreters
6. Routers
7. Getting data from databases.
8. many more.

Direct Address Table ⇒ only for small values

It doesn't handle
1) large values
2) floating point numbers
3) strings (or) addresses
4) combination of digits & chars.

⇒ 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



example operations:

insert (10)
insert (20)
insert (119)
search (10)
search (20)
delete (119)
search (119)

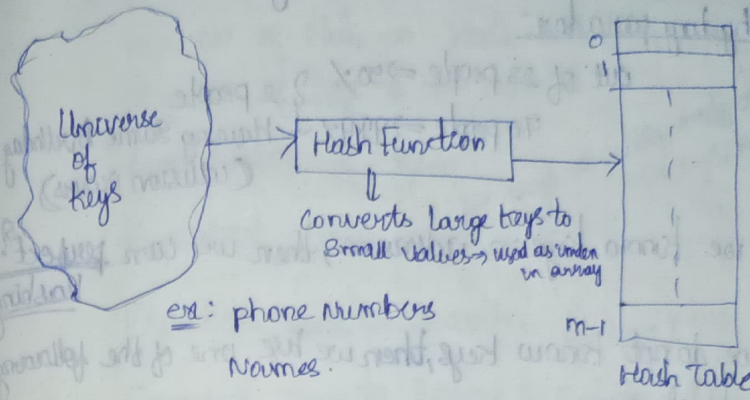
```
delete(i)
{
  table[i]=0;
}
// table[i-1000]=0;

insert(i)
{
  table[i]=1;
}
// table[i-1000]=1;

search(i)
{
  return table[i];
}
```

O(1)

Hashing Introduction



ex: phone numbers
Names.
Employee ID [E1021]

How Hash Functions Work?

↳ should always map a large key to same small key
↳ should generate values from 0 to m-1
↳ should be fast O(1) for integers and O(len) for string of length 'len'
↳ should uniformly distribute large keys into Hash table slots.

examples of Hash functions

① $h(\text{large-key}) = \text{large-key} \% m$

m ⇒ prime no.
m ≠ 2 powers (2^m ⇒ we choose only m bits)
10 powers (10^m ⇒ we choose only m digits)

② for strings; weighted sum ⇒ to avoid permutations

str[] = "abcd"

$(str[0] * x^0 + str[1] * x^1 + str[2] * x^2 + str[3] * x^3) \% m$

x = 33

3) Universal Hashing

Collision Handling

Birthday paradox:

out of 23 people ⇒ 50% } 2 people having same birthday (collision occurs)
70 people ⇒ 99.9% }

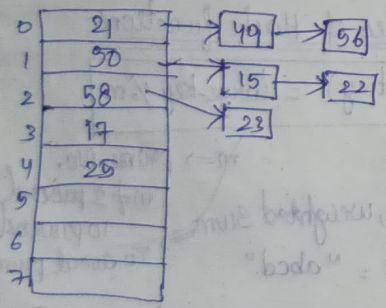
⇒ If we know keys in advance, then we can ~~perfect~~ Hashing

⇒ If we do not know keys, then we use one of the following
- chaining
- open Addressing
  - Linear probing
  - Quadratic probing
  - Double Hashing

Chaining

hash(key) = key % 7

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



Hash Table (Array of Linked List Headers)

performance:

m = No of slots in Hash Table
n = No. of keys to be inserted
Load factor $\alpha = n/m$ (keys are uniformly distributed (n keys, m slots))
Expected chain length = α
Expected Time to search = $O(1+\alpha)$
Expected Time to Insert & Delete = $O(1+\alpha)$

Data structure for storing chains:

↳ Linked List ⇒ search O(1)
insert O(1)
delete O(1)

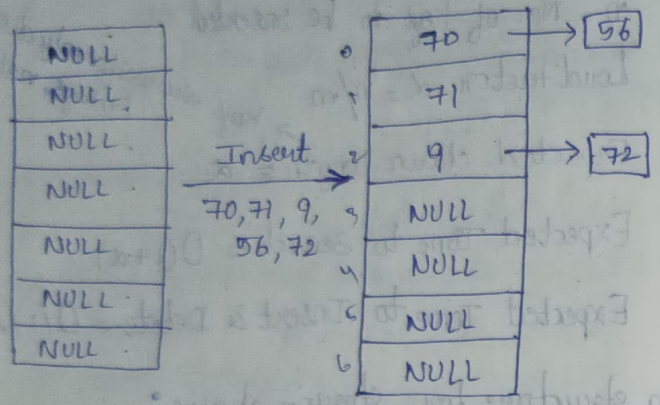
↳ Dynamic sized Arrays (vector in c++, ArrayList in java, List in python)
↳ Cache friendly (advantage)

↳ Self Balancing BST (AVL Trees, Red Black Tree)

Search, insert, Delete → O(Log n)

Implementation of chaining

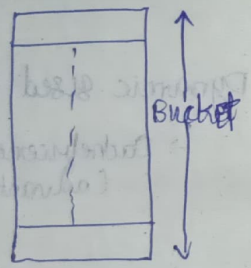
BUCKET = 7



Search(56) ⇒ True

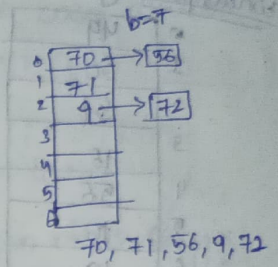
Search(97) ⇒ False

```
struct MyHash {
  int BUCKET;
  list<int> *table;
  MyHash (int b)
  { BUCKET = b;
    table = new list<int>[b];
  }
  void insert(int key) {---}
  bool search(int key) {---}
  void remove (int key) {---}
};
```



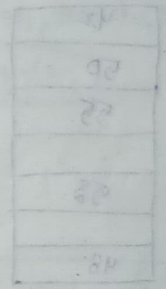
Insert & Delete

```
struct MyHash {
  int BUCKET;
  list<int> *table;
  . . . . . .
  . . . . . .
  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);
  }
  . . . . . .
  . . . . . .
};
```



Search :

```
struct MyHash
{ int BUCKET;
  list<int> *table;
  . . . . . .
  bool search(int key)
  { int i = key % BUCKET;
    for (auto x : table[i])
      if (x == key)
        return true;
    return false;
  }
  . . . . . .
};
```



Open Addressing

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

⇒ Cache-friendly

collision

Linear probing

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

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

hash(key) = key%7

If there is a collision then linearly search for next empty slot

⇒ If last slot is occupied then start search in circular manner

Linear probing: Linearly search for next empty slot when there is a collision

0	42
1	50
2	55
3	
4	53
5	
6	48

48, 42, 50, 55, 53

How to handle deletions in open addressing?

0	
1	50
2	51
3	15
4	
5	
6	

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

hash(key) = key % 7

Search: we compute Hash function, we goto that index and 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) Traverse through the whole table

How to handle deletions in open addressing?

↳ problem with simply making slot empty when we delete.

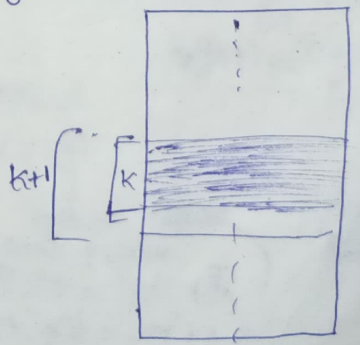
So Mark the slots as ⇒ Deleted

0	
1	50
2	Deleted (51)
3	15
4	
5	
6	

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

hash(key) = key%7

Clustering (A problem with linear probing)



(due to clusters all operations are costly (S, I, D)

How to handle clustering problem with Linear probing?

Linear probing: $hash(key,i) = (h(key) + i)\%7 \Rightarrow h(key) = key\%7$

problem: primary cluster

① Quadratic probing: ⟹ problem: secondary cluster

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

load factor

1. $(n/m) < 0.5$
2. m is a prime no.

⟹ then only it guarantees to find a free slot.

② Double Hashing:

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

Double Hashing:

$h_2(key) = (m-1) - (key\%(m-1))$

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

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

→ Distributes keys more uniformly than linear probing and quadratic Hashing

→ No clustering

49, 63, 56, 52, 54, 48

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

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

$m = 7$

$h_1(key) = key\%7$ → returns values from 0 to 5.

$h_2(key) = 6 - (key\%6)$ → it never returns "0".

49

$h_1(49) = 49\%7 = 0$

i=0

$hash(key,i) = 0$

63

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

i=1

$h_2(63) = 6 - 63\%6 = 6-3 = 3$

$hash(63,1) = (0 + 1*3)\%7 = 3$

56

$h_1 = 0$

$6 - 56\%6 = 6-2 = 4$

$(1*4)\%7 = 4$

54

$h_1 = 5$

$h_2 = 6 - 54\%6 = 6$

$(5+6)\%7 = 4$ → collision happens

So increment i value

$(5+6*2)\%7 = 17\%7 = 3$

$(5+6*3)\%7 = 23\%7 = 2$

52

$h_1 = 3$

$6 - 52\%6 = 6-4 = 2$

$(3+2)\%7 = 5$

48

$h_1 = 6$

$h_2 = 6 - 48\%6 = 6$

why $h_2(key)$ and m should be relatively prime?

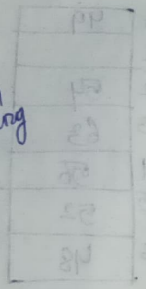
↳ then always find free slot if there is

$(1\times6)\%7 = 6$
$(2\times6)\%7 = 5$
$(3\times6)\%7 = 4$
$(4\times6)\%7 = 3$
$(5\times6)\%7 = 2$
$(6\times6)\%7 = 1$

$h_2(key)=6$ ⇒ shifts when there are collisions

(6,7) are relatively prime ✓

```
Void doubleHashingInsert (key)
{
    if (table is full)
        return error;
    probe = h1(key), offset = h2(key);   // I ⇒ 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 = ? ⇒ $\frac{1}{1-\alpha}$

$\alpha = 0.8 \Rightarrow$ (5 Searches)
80% Table full
20% " free ⇒ $\frac{20}{100} = \frac{1}{5}$ slots are empty

$\alpha = 0.9 \Rightarrow$ (10 searches)
90% Table full
10% " free ⇒ $\frac{10}{100} = \frac{1}{10}$

Implementation of open Addressing:

empty => -1
deleted => -2

```
MyHash * mh(7);
mh.insert(49);
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");
```

49	56	72	-1	-1	-1	-1
0	1	2	3	4	5	6

49	-2	72	-1	-1	-1	-1
0	1	2	3	4	5	6

-2

```
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) {----}
    bool insert(int key) {----}
    bool delete(int key) {----}
};
```

linear probing

search:

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

49	50	51	63	-1	-1	64
0	1	2	3	4	5	6

=> 63: h=0, i=0 (True) ; 64: h=0, i=1 (false)

Insert:

```
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;
  }
}
```

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

size=0

Insert : 49, 50, 63, 64, 69, 68

49	50	63	64	-1	68	69
0	1	2	3	4	5	6

size=0, 1, 2, 3, 4, 5, 6

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

0	1	2	3	4	5	6
49	50	51	58	7	68	75
	-2		-2			

erase (50)
erase (63)
erase (64) => false

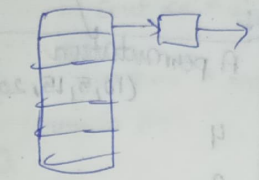
How to Handle the cases when -1 and -2 are I/p keys?

=> Library impl => uses pointers to store at slots
empty can be checked by NULL instead of -1.

=> for delete => create Dummy node.

chaining

1) Hash table never fills
2) less sensitive to Hash functions
3) poor cache performance
4) extra space for links

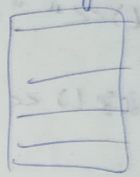


$(1+\alpha)$ $\alpha = 0.9$

1.9 Comparisons

open Addressing

1) Table may become full and resizing becomes mandatory
2) extra care required for clustering
3) cache friendly
4) extra space might needed to achieve same performance as chaining.



$\frac{1}{1-\alpha} = \frac{1}{1-0.9} = \frac{1}{0.1} = 10$ Comparisons

Unordered set in C++ STL:

Set is based on Red-Black tree (Self Balancing BST)
unordered set is based on Hashing.

```
#include<iostream>
#include<unordered_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;
}
```

O/P: All elements on any order

10 5 15 20
(OR)
5 10 20 15
- - - -
- - - -
— —

Permutations

```
int main()
{ unordered_set<int> s;
  s.insert(10);
  s.insert(5);
  s.insert(15);
  s.insert(20);
  for (auto it = s.begin(); it != s.end(); it++)
   cout << (*it) << " ";
  cout << endl;
  cout << s.size() << " ";
  s.clear();
  cout << s.size() << " ";
  return 0;
}
```

o/p: A permutation (10,5,15,20)
4
0

insert() ✓
begin() ✓
end() ✓
size() ✓
clear() ⇒ clear the unordered set (or) container
find() ⇒ find element is present or not (when it present then it returns location (address of this element) otherwise it returns s.end()
count()

```
⇒ int main()
{ unordered_set<int> s;
  s.insert(10);
  s.insert(5);
  s.insert(15);
  s.insert(20);
  if (s.find(15) == s.end())
      cout << "Not found";
  else
     cout << "Found" << *(s.find(15)); // Found 15
  return 0;
}
```

count(): used as substitute for find fn (returns either '1' or '0')

```
int main()
{
  ..  ..
  .  ..
  if (s.count(15))
     cout << "Found";
  else
     cout << "Not found";
  return 0;
}
```

erase():

```
int main()
{
  .  .
  .  .
  cout << s.size() << " ";   ⇒ o/p: 4 3 2
  s.erase(15);
  cout << s.size() << " ";
  auto it = s.find(10);
  s.erase(it);
  cout << s.size() << " ";
  return 0;
}
```

```
int main()
{
  .  .
  .  .
  s.erase(s.begin(), s.end());
  cout << s.size();
  return 0;
}
```

o/p: 0

Internal working & Time complexities:

begin(), end() > O(1)
cbegin(), cend()

insert(), erase(val) > O(1) on average
erase(it), find()

size(), empty() —— O(1)

Applications:

We can use anywhere when we need following operations (or a subset of following operations quickly)

- Search
- Insert
- delete

Unordered set ⇒ only keys
Unordered maps ⇒ keys & key value pairs

unordered map in C++ STL:

map ⇒ Red Black Tree
Unordered map ⇒ Hashing

- Used to store key, value pairs
- Uses Hashing
- No order of keys.

```
#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;
}
```

Member access operator. When we put key inside in that [] op. to key is present then it gives the reference of a value corresponding to that key otherwise inserts key.

m({"gfg",20}, {"ide", 30}, {"courses", 15})

o/p: gfg 20
courses 15
ide 30
} any order.

```
int main()
{ unordered_map<string, int> m;
  m["gfg"] = 20;
  m["ide"] = 30;
  m["courses"] = 15;
  if (m.find("ide") != m.end())
      cout << "Found \n";
  else
      cout << "Not Found \n";

  for( auto it = m.begin(); it != m.end(); it++)
      cout<<(it->first) << " " <<(it->second) << endl;
  return 0;
}
```

```
auto it = m.find("ide");
if (it != m.end())
{ cout << (it->second);
}
```

o/p: found

gfg 20
courses 15
ide 30
} Any order

```
if(m.count("ide")>0)
    cout << "Found";
else cout << "Not found";
```

```
cout << m.size() << " "; //3
m.erase("ide");
m.erase( m.begin());
cout << m.size() << " "; //1
```

Time complexities

- begin()
- end()
- size()
- empty()

O(1) in worst case.

- count()
- find()
- erase(key)
- insert()
- [ ]
- at

(based on hashing)

O(1) on average.

Count distinct elements:

I/p: arr[] = {15, 12, 13, 12, 13, 13, 18}

o/p: 4

I/p: arr[] = [10, 10, 10]

o/p: 1

I/p: arr[] = [10, 11, 12]

o/p: 3

Naive solution:

TC = O(n²)
A.S = O(1)

```
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;
}
```

arr[] = {10, 20, 10, 20, 30}

i=0, res=1
i=1, res=2
i=2,
i=3,
i=4, res=3

Efficient Solution:

T.C = O(n)
A.S = O(n)

```
int countDistinct (int arr[], int n)
{
  unordered_set<int> s;
  for (int i=0; i<n; i++)
    s.insert(arr[i]);
  return s.size();
}
```

arr[] = {10, 20, 10, 20, 30}
s = { }
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}

Improved Efficient Implementation

T.C = O(n)
A.S = O(n)

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

Frequencies of Array elements:

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

I/p: arr[] = [10, 20]

O/p:
10 1
20 1

Naive solution:

T.C = O(n²)
A.S = O(1)

```
void printFreq (int arr[], int n)
{
  for (int i=0; i<n; i++)
  {
    // check if seen before
    bool flag = false;
    for (int j=0; j<i; j++)
      if (arr[i] == arr[j]) { flag=true; break; }
    // If seen Ignore
    if (flag == true) continue;
    // If not seen before count frequency
    int freq = 1;
    for (int j=i+1; j<n; j++)
      if (arr[i] == arr[j])
        freq++;
    cout << arr[i] << " " << freq << endl;
  }
}
```

[10, 20, 20, 30, 10]
i=0 : print(10, 2)
i=1 : print(20, 2)
i=2 :
i=3 : print(30, 1)
i=4 :

efficient solution: T.C = O(n) A.S = O(n)

[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 Arrays

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

I/p: a[] = {10, 20, 30}
b[] = {30, 20, 10}

o/p: 10 20 30

Naive solution: T.C = O(m x n) A.S = O(1)

```
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)
      print(a[i] + " ");
  }
}
```

a[] = [10, 20, 30]
b[] = [10, 30]
m = 3
n = 2
i=0 : flag=true, print(10)
i=1 : flag=false
i=2 : flag=true, print(30)

Efficient solution:

a[] = {10, 20, 30} m
b[] = {30, 10} n

h = { } ⇒ θ(n)
h = {30, 10} ⇒ θ(m) ⇒ compare ⇒ [10, 30]

(we want on a order array)

T.C = θ(m+n) | A.S = θ(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] << " ";
}
```

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

After S-1: s = [30, 10]

S-2: i=0 : print(10)
i=1 :
i=2 : print(30)

Union of Two Unsorted Arrays

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

I/P: a[] = [3, 3, 3]
b[] = [3, 3]

O/P: 1

Naive Solution: T.C = θ(m*m + n*n)
A.S =

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 of b[].

T.C = O(m*m + n*(m+n))

a[] = [10, 30, 10]
b[] = [5, 10, 5]

Initially:
res = 0
dist[] = { }

After traversing a[]:
res = 2
dist[] = [10, 30].

After traversing b[]:
res = 3
dist[] = [10, 30, 5]

~~int unionCount (int a[], int b[],~~

Efficient solution:

T.C = $\theta(m+n)$
A.S = $O(m+n)$

1) create an empty hash table, h
2) Insert all elements of a[] in h. $\Rightarrow \theta(m)$
3) Insert all elements of b[] in h. $\Rightarrow \theta(n)$
4) Return h.size().

```
int unionCount (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();
}
```

a[] = [10, 30, 10, 40]
b[] = [40, 5, 10, 15]

Initially:
h = { }

After processing a[]
h = {10, 30, 40}

After processing b[]
h = {10, 30, 40, 5, 15}

pairs with given sum:

I/p: arr[] = {8, 2, 8, 15, -8}
sum = 17
O/p: true

I/p: arr[] = [5, 8, -3, 6]
sum = 3
O/p: true

I/p: arr[] = [2, 1, 6, 5]
sum = 6
O/p: false

Naive solution:

① consider all possible pairs one by one, checks 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.

there are $\frac{n*(n-1)}{2}$ pairs

T.C = $O(n^2)$
A.S = $O(1)$

```
bool isPair (int arr[], int n, int sum)
{
    for (int i=0; i<n; i++)
        for (int j=i+1; j<n; j++)
            if (arr[i] + arr[j] == sum)
                return true;
    return false;
}
```

arr[] = [8, 4, 3, 9]
sum = 13
i=0 : j=1
: j=2
: j=3
i=1 : j=2
j=3 return true

Note: First inserting everything into a hashtable then traversing through the array doesn't work.

arr[] = [3, 3, 4, 5]
sum = 6
h = [3, 3, 4, 5]
Now if we traverse and look for (sum - arr[i]) in h, we would return incorrect result.

```
bool isSum (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;
}
```

T.C = O(n)
A.S = O(n)

arr[] = [3, 3, 4, 2, 5]
sum = 6
Initially: h = []
i=0 : h = [3]
i=1 : h = [3, 3]
i=2 : h = [3, 3, 4]
i=3 : return true.

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

I/p: arr[] = [3, 1, -2, 5, 6]
o/p: NO

I/p: arr[] = [5, 6, 0, 8]
o/p: yes

A.S = O(1)
T.C = O(n²)

Subarrays of [10, 20, 30]:
[10]
[10, 20]
[10, 20, 30]
[20]
[20, 30]
[30]

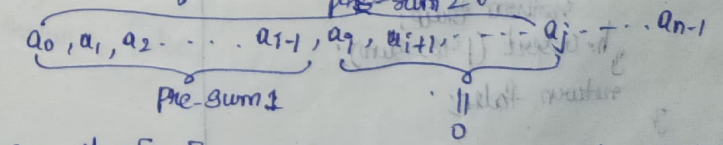
Naive solution:-

```
bool isSubarray (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;
}
```

arr[] = [4, -3, 2, 1]
i=0, j=0) curr_sum = 4
j=1, curr_sum = 1
j=2, curr_sum = 3
j=3, curr_sum = 4
i=1 : j=1; curr_sum = -3
j=2, curr_sum = -1
j=3, curr_sum = 0
return true.

Efficient solution:

Idea: Use prefix sum and Hashing.



If sum of a[i:j] is 0, then pre_sum1 must be same as pre_sum2.

[-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, -2}

i = 3 : pre_sum = -3
Already present in hash
Return True

T.C = O(n)
A.S = O(n)

```
bool isSubarray (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~~

Subarray with given sum:

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

o/p: True

I/p: arr[] = [15, 2, 8, 10, -5, -8]
sum = 3

o/p: True

I/p: arr[] = [3, 2, 5, 6]
sum = 10

o/p: True

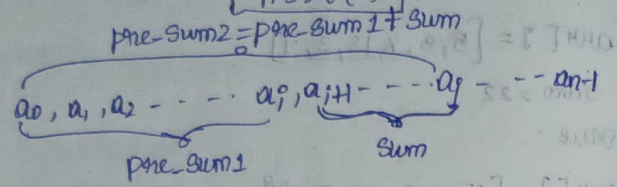
Naive solution: T.C = O(n²) A.S = O(1)

```
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 solution:

T.C = O(n)
A.S = O(n)

pre_sum2 = pre_sum1 + sum



arr[] = [5, 8, 6, 13, 3, -1]
sum = 22

h = { }

we need to check "prefix sum - sum" exits among prefix sums.

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_set<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 false;
}
```

arr[] = [2, 3, 4, 6]
sum = 9
h = { 2, 5, 9, 15 }
-7 -4 0 6

Longest subarray with Given Sum:

I/p: arr[] = [5, 8, -4, -4, 9, -2, 2]
sum = 0
O/p : 3

I/p: arr[] = [3, 1, 0, 1, 8, 2, 3, 6]
sum = 5
O/p: 4

I/p: arr[] = [8, 3, 7]
sum = 15
O/p: 0

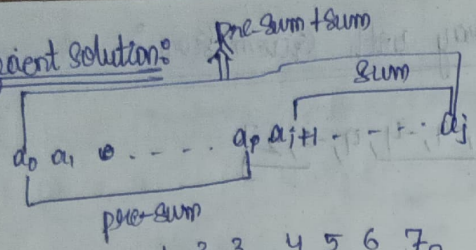
Naive solution:

T.C = O(n²)
A.S = O(1)

```
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, j-i+1); }
    }
  }
  return res;
}
```

arr[] = [5, 2, 3, 4]
sum = 5
initially res = 0
i=0 : res = 1
i=1 : res = 2
i=2 :
i=3 :

Efficient solution:



arr[] = [8, 3, 1, 5, -6, 6, 2, 2]
0 1 2 3 4 5 6 7

sum=4

m = { } (Initially)

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

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

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

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

m = " " "

m = " " "

} To keep indexes min we don't updating

m = {(8,0), (11,1), (12,2), (17,3) (19,5)} res=2

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

T.C = O(n)
A.S = O(n)

```
int maxLen(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;
}
```

arr[] = [5,2,3] sum = 5

m = { }, res = 0.

i=0: pre_sum = 5, res = 1, m = {(5,0)}

i=1: pre_sum = 7, res = 1, m = {(5,0), (7,1)}

i=2: pre_sum = 10, res = 2, m = {(5,0), (7,1), (10,2)}

Longest Subarray with equal 0's and 1's

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

O/p: 6

I/p: arr[] = [1,1,1,1]

O/p: 0

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

O/p: 4

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

O/p: 6

Naive solution:

T.C = O(n²)
A.S = O(1)

```
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(a[j]==0)
                c0++;
            else
                c1++;
            if(c0==c1)
                res = max(res, c0+c1);
        }
    }
    return res;
}
```

arr[] = [1,1,0,1,0]

i=0: res=0

i=1: res=0

i=2: res=4

i=3: 2

i=4: 1

i=5: 4

Efficient solution:

① Replace every 0 with -1.

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

Time $\theta(n)$

for example [1,0,1,1,1,0,0] becomes

[1,-1,1,1,1,-1,-1]

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

$\Rightarrow$ $\theta(n)$ Time, $O(n)$ A.S

Longest Common Subarray with given Sum (Binary arrays)

We are given two binary subarrays of same sizes

I/p: arr1[] = [0,1,0,0,0,0]
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

I/p: arr1[] = [0,0,0]
arr2[] = [1,1,1]

O/p: 0

I/p: arr1[] = [0,0,1,0]
arr2[] = [1,1,1,1]

O/p: 1

Naive solution:

T.C = $\theta(n^2)$
A.S = $O(1)$

```
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;
}
```

arr1[] = [0,1,0,0,0,0]
arr2[] = [1,0,1,0,0,1]
Initially: res=0
i=0 : j=1 : res=2
i=1 : j=4, res=4
i=2:
i=3:
i=4:
i=5:

Efficient solution:

T.C = $\theta(n)$
A.S = $O(n)$

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];
```

$\theta(n)$ Time
$\theta(n)$ A.S

for example: arr1[] = [0,1,0,0,0,0]
arr2[] = [1,0,1,0,0,1]
temp[] = [-1,1,-1,0,0,-1]

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

Values in temp[i]:
- 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

$\Rightarrow$ T.C = $O(n)$
A.S = $O(n)$

Longest Consecutive Subsequence:

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

o/p: 4

I/p: arr[] = [8, 20, 7, 30]

o/p: 2

I/p: arr[] = [20, 30, 40]

o/p: 1

We need to find longest subsequence in the form of
x, x+1, x+2, x+3 - - - - - x+i with these elements appearing in any order.

Naive solution:

Hint: use sorting

TC = O(nlogn)
AS = O(1)

```
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]+1)
            curr++;
        else if(arr[i] != arr[i-1])
        {res= max(res,curr);
         curr=1;
        }
    }
    return max(res,curr);
}
```

[3,8,2]
→ [2,3,8]

arr[] = [2, 9, 4, 3, 10]

After sorting:

arr[] = [2, 3, 4, 9, 10]

Initially: res=1, curr=1

i=1 : curr = 2

i=2 : curr = 3

i=3 : res=3, curr=1

i=4 : curr = 2

efficient solution:

Hints: ⊛ 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}
res=4 curr=2

T.C = O(n)
A.S = O(n)
- there are always 2n lookups at most

```
int longSub(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;
}
```

for first elements: 2+(len-1)
for other elements: 1
"len" is the length of the subsequence with the given as first.

arr[] = [1, 3, 9, 2, 8, 2]
h = {1, 3, 9, 2, 8}
x=1 : curr=1, curr=2, curr=3, res=3 } 4
x=3 :]1
x=9 :]1
x=2 :]1
x=8 : curr=1, curr=2]3

h={1,3,4,2}
x=1 : curr=1, curr=2, curr=3, curr=4 } 5
x=3 :]1
x=4 :]1
x=2 :]1
Total 8 lookups

h={8,20,4}
x=8 : curr=1]2
x=20 : curr=1]2
x=4 : curr=1]2
Total 6 lookups

h={2,4,8}
x=2 : curr=1, curr=2, curr=3 } 4
x=4 :]1
x=8 :]curr=1]2
x=3 :]1
Total 8 lookups

Count Distinct elements in every window:

I/p: arr[] = [10, 20, 20, 10, 30, 40, 10]
k=4

o/p: 2 3 4 3

I/p: arr[] = [10, 10, 10, 10]
k=3

o/p: 1 1

I/p: arr[] = [10, 20, 30, 40]
k=3

o/p: 3 3

Naive Solution: T.C = 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; p<j; p++)
        if (arr[i+j] == arr[i+p])
        { flag=true; break; }
      if (flag==false)
        count++;
    }
    cout<<count<<" ";
  }
}
```

arr[] = [10, 10, 5, 3, 20, 5]
k=4
i=0 : count = 3
i=1 : count = 4
i=2 : count = 3

Efficient solution: T.C = O(n) → O((n-k)+O(k))
A.S = O(k)

⇒ [10, 20, 10, 10, 30, 40] (indices 0 1 2 3 4 5)

k=4

① :

10	3
20	1

② : count = 2

③ : i=4 arr[i] = 30
arr[i-k] = 10

10	2
20	1
30	1

count = 3

i=5 arr[i] = 40
arr[i-k] = 20

10	2
30	1
40	1

count = 3

① Create a frequency map of first k items
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] becomes 0, remove it from the map.
c) If arr[i] does not exist in the map, insert it.
d) else increase its frequency in the map.
d) print size of the map.

```
int findLongest(int arr[], int n)
{ unordered_map<int,int> m;
  for(int i=0; i<k; i++)
    m[arr[i]]+=1;
  cout<<m.size()<<" ";
  for(int i=k; i<n; i++){
    m[arr[i-k]]-=1;
    if (m[arr[i-k]]==0){
      m.erase(arr[i-k]);
    }
    m[arr[i]]+=1;
    cout<<m.size()<<" ";
  }
}
```

More than n/k Occurrences:

I/p: arr[] = [30, 10, 20, 20, 10, 20, 30, 30]
k=4

O/p: 20 30
Note that n=8 and n/k=2

I/p: arr[] = [30, 10, 20, 30, 30, 40, 30, 40, 30]
k=2

O/p: 30
Note that n=9 & n/k=4.5

Naive Solution:

```
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++;
  }
}
```

n=7, k=2, n/k=3.5
[10, 10, 20, 30, 20, 10, 10]
after sorting:
[10, 10, 10, 10, 20, 30, 30]
i=1 : count=2
i=2 : count=3
i=3 : count=4
i=4 : print(arr[3]), count=1
i=5 : count=2
i=6 : count=1

T.C = O(nlogn) + O(n) => O(nlogn)
A.S = O(1)

Efficient solution

```
void printNBYK(int arr[], int n, int k)
{ unordered_map<int, int> m;
  for(int i=0; i<n; i++)
    m[arr[i]]++;
  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)}
n=6, k=3, n/k=2
=> 10

T.C = O(n)
A.S = O(n)

More than n/k occurrences (O(nk) solution)

[30, 10, 20, 20, 20, 10, 40, 30, 30]
k=4, n=9
let 'res_count' be the no of elements in the result

res_count ≤ k-1

Moore voting algorithm

~~k * (n/k)~~ ~~≤ n~~

k * (n/k + 1) ≤ n

n + k ≤ n

k elements in o/p. every element can have more than n/k occurrences

Not true

So if it is not true for k then it is not true for any other larger No. as well
So No of elements in o/p not > k-1

① Create an empty map m.

② for (int 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 becomes 0, remove

phase-I
finding candidates

③ for all elements in m, print the elements that actually appear more than n/k times

phase-II
actually majority or Not

[30, 10, 20, 30, 20, 10, 40, 30, 30]
0 1 2 3 4 5 6 7 8

k=4, n=9

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={(30,1),(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

check actual occurrences of these elements & print

20 30

```
void printNBYK(int arr[], int n, int k)
{ unordered_map<int, int> m;
  for(int i=0;i<n;i++)
  { if(m.find(arr[i])!= m.end())
      m[arr[i]]++;
    else if (m.size()<k-1)
      m[arr[i]]=1;
    else
      for(auto x : m)
      { x.second--;
        if(x.second==0)
          m.erase(x.first);
      }
  }
  for(auto x :m)
  { int count =0;
    for(int i=0;i<n;i++)
    { if(x.first==arr[i])
        count++;
    }
    if(count>n/k)
    { cout<< x.first<<" ";
    }
  }
}
```

k → O(n*k)

O(k)

T.C = O(nk)

How does the approach works?

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

10, 30
20, 40
Rejected

10, 20
20, 30,
30
Selected

40 Rejected itself and three others

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