HASHING

- provides insert, delete, search operations in O(1) time
- No duplicates are allowed.
- Hashing is not used
  - when we have to find closer values.
  - data is arranged in sorted fashion
  - prefix searching → "gee" in "geeksforgeeks"

  } AVL or Red black tree

Application of Hashing

Hash table is the second most used data structure

- Implementing dictionaries
- Database Indexing
- Cryptography
- caches
- Symbol tables in compilers / interpreters
- Routers.
- Geting data from databases

Direct Address table

We use keys as indexes of an array. on insert operation arr[key] = 1 on delete operation arr[key] = 0 and on search we just check

```
if (arr[key] = 1)
    return true;
else
    return false;
```

- If we have a range of [0 - n] we can make an array of arr[n]
- If we have range from [m-n] we can use. arr[m-n] & arr[key-m] ← 1

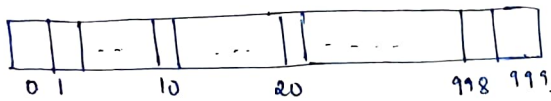


table [].

* since array has random access

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

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

```
insert(i) {
    table(i)=1;
}
```

Problems

① large range of keys

② Floating point numbers (as keys are used as index & index can only be positive integral)

③ keys can be strings.

Hashing

Take large universe of keys and by using hash function convert them to small values.

* Every time hash function must produce same value for same key
* should generate value between 0 to m-1.
  m → size of hash table
* Hash function should be fast
* Should uniformly distribute large keys into Hash table slots.

Examples of Hash Function

② h(large-key) = large-key % m.
  m is chosen as prime no
  • To have less common factors

④ For strings → weighted sum.

① ↳ sum of ascii values of all the characters of string.
Problem is we have same index for abcd, acbd, adcb ....

② → str → "abcd"

str[0] × x^0 + str[1] × x^1 + str[2] × x^2 + str[3] × x^3.

choose a number x

↳ it producing unique keys most of the time

③ Universal Hashing

• group of hash function, you pick one hash function randomly.

Collision handling

It is possible that two large-values maps to the same keys. This situation is called as collision.

① If data is known to us in advance, we can use perfect Hashing to guarantee zero collision.

② If we do not know keys in advance.
To handle collision, we have two techniques

- Chaining
- Open Addressing
  - linear probing
  - quadratic probing
  - Double Hashing

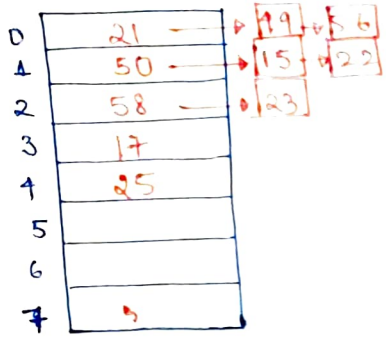
Chaining

hash function

hash (value) → value % 7

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

Hash Table → Array of linked list headers.



- Basic funda is to link the repeated value with the head.

50 % 7 → 1
21 % 7 → 0
58 % 7 → 2
17 % 7 → 3
15 % 7 → 1
49 % 7 → 0
56 % 7 → 0

Performance of Chaining

m → no of slots
n → no of keys to be inserted

load factor $\alpha = n/m$.

α → Trade-off between space & time

- If hash table is small / load factor is big → more no of collision

Average chain length → α

Expected time for search or insert → $O(1+\alpha)$

$O(1)$ → hash function computation
$O(\alpha)$ → searching in chain of length α

Always assumption that keys are uniformly distributed

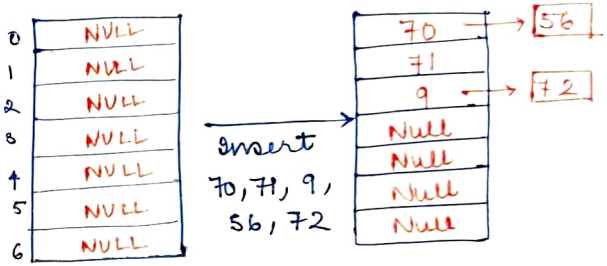
under worst case all the keys will map to same key index - hence chain length → n. & complexity will be O(n)

Data structures to store chains

- linked list → search O(n), Delete O(n), Insert O(1) } But it is cache unfriendly
- Dynamic size arrays → search/insert/delete O(n) } But is provides cache friendliness
- Self Balancing BST → search/ins/del → O(log n) But does not provide cache friendliness
  (AVL trees, Red Black trees)

Implementation of linked list

Bucket → 7



Basic structure of class

```
Struct MyHash{
    int BUCKET;
    list<int> *table;        <-- use to implement linked list
    MyHash(int b){
        Bucket = b;
        table = new list<list>[b];      <-- user will provide size
    }                                       Array of linked list using new.
    void insert(int key){
        ....
    }
    ....
};
```

we can use push_back to insert values in linked list.

```
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

No of slots in the hash table ≥ no of keys to be inserted

Advantage → cache Friendly.

Ex→ key → {50, 51, 49, 16, 56, 15, 19}

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

hash(key) = key % 7

- linearly search for next empty slot when there is collision
- If the collision occurs at last index we search for the next slot in linear circular fashion

0	
1	50
2	51
3	15
4	
5	
6	

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

- In search operation, we first search in i^{th} index $i = key \% 7$

→ and if arr[i] is not empty & contains some other value we linearly move in search of x

we stop our search when

(i) arr[i] = x or Return True

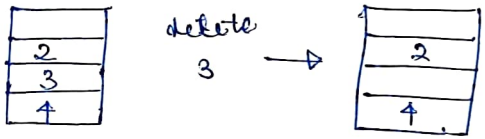
(ii) arr[i] = null → Return false

(iii) or we traverse back at the same slot

In delete operation

we cannot simply make a slot empty as during searching we check for empty slots and this may modify our results

Ex



Let i = 1 & x = 4
we start at i = 1 arr[i] = 2
move linearly
arr[i] = null
↳ stop
return false

wrong result

Hence we mark the slot as deleted slot (not the empty slot)

Problems with linear probing

- Clusters are formed
- Suppose we have a collision → cluster of size 2
- & if another key maps to the index of any one of them, cluster will grow → 3
- Because of this if k is large all insert/search/del operation become costly

We can write linear probing as

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

↳ i is incremented till 6 and if we don't find any empty slot it shows hash table is full.

Quadratic Probing

The problem of clusters are removed to much extent in this, instead of linearly searching for next slot, we search for 1^2, (next 1st slot), 2^2 $(i+4)$, $3^2(i+9)$ ....

* secondary clusters were formed.

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

Disadvantages

There may be a case when quadratic probing will not find an empty slot even if there exist an empty slot

And it is proven that if $\alpha = 0.5$ &
m → prime.
then only quadratic probing guarantees that it'll find free slot

Double Hashing

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

- two hash function
- one hash function to find the slot
- second hash is to find the slot if calculate by the first is not free.
- If m is relatively prime, then it always find a free slot if there is one
- Distribute keys more uniformly than linear probing & quadratic

- No clustering

Also h_2 can never return 0, because if it return zeroes it will repeatedly perform collision.

Ex → $6 - (key \% 6)$ → give value b/w [0-5]

$6 - \{[0-5]\}$ ↳ [1-6].

Keys → {49, 63, 56, 52, 54, 48}

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

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

- initially we put i=0, & and calculate using h_1 only.
- h_2 is used only when collision happens initially put i = 1 then i = 2, 3 ....

→ key = 49
hash(49) = 0

→ key = 63.
hash(63) = 0
collision

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

For the first time put i = 1

hash (63) = (0 + 1(3)) % 7 = 3

→ key = 54
$h_1(key) = 54 \% 7 = 5$

$h_2(key) = 6 - (54 \% 6) = 6$

hash(key) = (5+6)%7
≤
(5 + i(6)) % 7
first time i = 1
(5+6) % 7 → 4 (filled)
second i = 2
(5+12) % 7
17 % 7 = 3 → filled
check → 3
(5+18) % 7 = 23 % 7 → 2 (found)

- we increament i each time we see collision and initially we start with i = 1 for first collision.

Requirement
- $h_2(key)$ & m should be relatively prime

Algorithm for open Addressing

```
void doubleHashing(key) {
    if (table is full)
        return error;
    probe = h1(key), offset = h2(key);
    while (table[probe] is occupied)
        probe = (probe + offset) % m;
    table[probe] = key;
}
```

For linear probing offset → 1.

Performance Analysis of search

$\alpha = n/m$ (should be ≤ 1) i.e. no of slots should be greater than keys

Let $\alpha = 0.8$

it means 80% of table is occupied &
20% is still empty. (1/5)

To find an empty slot you need 5 iterations
ie 4 filled slots & then 1 empty slots.

$\alpha = 0.9$

Average no of probes req → 9

$= \boxed{\frac{1}{1-\alpha}}$

Implementation of Open Addressing

Assumption :- -1, -2 are not present as keys.
-1 is for ~~deleted~~ empty
$-\alpha$ is for deleted.

Structure in C++

```
struct myHash {
    int *arr;
    int cap, size;
    myHash(int c) {
        cap = c;
        size = 0; arr = new int[cap];
        for (int i=0; i<cap; i++)
            arr[i] = -1;          } empty array

        int hash(int key) {
            return key % cap;     } hash function
        }
        bool search(int key) { --- }
        bool insert(int key) { --- }
        bool erase(int key) { --- }
    }
};

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

we return false when
① we see an empty slot ie arr[i] = -1
② we traverse the whole arr (i == h)

we return true when we find the key
simple :p

```
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;
    }
}
```

* we returned false.
  (1) hash table is full.
  (2) when key is already present
* Returned true + Found spot for the key & size++

```
bool erase (int key){
    if(size==0)
        return false;
    int i = hash(key);
    while(arr[i] != -1){ → search for the key.
        if(arr[i] == key){
            arr[i] = -2; size--;
            return true;
        }
    }
    i = (i+1)%cap;
    if(i==h){
        return false;
    }
    return false;
}
```

} Found the key

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

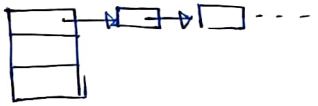
As what libraries do, they store references or pointers. we don't use actual keys we use their pointers so for.

① empty - we can simply use NULL

② deleted - we have a dummy node, whenever we delete something we store pointer or reference to this dummy node.

* this dummy node is ~~not~~ one part of class & is shared by all the members

Comparison between two techniques

Chaining (Better)	Open Addressing (can be used when count of keys is known)
⊙ Hash table would never fill.  May be the size of chain becomes huge but we can always add another node on the cost of performance	⊙ Table may become full and requires resizing after a point. There may be a case, when ~~key~~ not new keys are added, for that we need to resize the table if it is full
⊙ less sensitive to hash function	⊙ Extra care is required for clustering.
⊙ cache friendliness is very poor.	⊙ Cache Friendly
⊙ Extra space is required for links.	⊙ No extra space required
⊙ Performance → $O(1+\alpha)$ Better Performance $1+0.9 \to 1.9$	⊙ Performance $1/1-\alpha$ $\frac{1}{1-0.9} = \frac{1}{0.1} = 10$

Unordered set in C++

- It internally uses hashing for storing elements
- No particular order is maintained
- Output can be any perp permutation of elements
-

```
unordered_set<int> s;
```

① `s.insert(10)` ← to insert an element

② `s.find(15)` ← check if the element is present
- If element is present iterator to that element is returned
- If element is not present iterator to element just after last element is returned (`s.end()`)

```
if(s.find(15) == s.end())
```

③ `s.begin()` → iterator to first element

④ `s.size()` → gives the size of unordered set

⑤ `s.clear()` → it clears the unordered set completely

⑥ `s.count()` → alternative of find function and return either 1 or 0
- 1 → when element is present
- 0 → when element is not present

⑦ `s.erase()` → `s.erase(15)` → find & remove 15
- we can pass iterator also

```
it = s.begin() + 2;
s.erase(it)
```

- we can also use erase to remove set of elements

```
s.erase(s.begin(), s.end())
```

Unordered Map in C++ STL

- Based on Hashing
- unordered containers with fast insert, delete & search operations
- It stores key, value pair

```
unordered_map<string, int> m;
```
(string → data type 1, int → data type 2)

- `m["gfg"] = 20`

If key is present i.e if key value pair having "gfg" as key is present, it returns the reference of the value corresponding to the key and if key is not present, it simply inserts the value and returns the reference to the key.

`m["gfg"] = 20`
- `m["gfg"]` → inserting key & returning reference to the value (initialised default as 0)
- `= 20` → write to that value as 20

⑧ `m.insert({"courses", 15});`

standard function for inserting a key-value pair

⑨ Traversing

```
for(auto x : m)
    cout << x.first << " " << x.second << endl;
```
- `x.first` → accessing key
- `x.second` → accessing value

⊙ m.find("ide") → if m.find("ide") = m.end()
↑ key — it not key is not present

we can also get the corresponding value

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

⊙ m.begin() → iterator pointing to 1st element
⊙ m.end() → iterator pointing beyond last element
⊙ m.size() → number of key, value pairs
⊙ m.erase("ide") → remove key value pair

Time comp → O(1).

Count Distinct Elements

I/P → {10, 20, 30, 20, 10}
O/P → 3

I/P → {10, 10, 10}
O/P → 1

I/P → {10, 20, 30}
O/P → 3.

Naive Approach

```
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;
}
```

Time complexity = O(n²)
Space comp = O(1)

Efficient solution

we ~~are~~ use ~~to~~ unordered set in STL which uses hashing and insert all the elements in it.
As we know in unordered set duplicates are not allowed. Hence we can simply check the size of set for distinct elements.

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

we can directly pass integer array as
unordered_set<int> s(arr, arr+n);

Time complexity → O(n)
~~Extra~~ Aux comp → O(n)

Frequencies of Array Element

I/P → {10, 12, 10, 15, 10, 20, 12, 12}.

O/P →
10 – 3
12 – 3
15 – 1
20 – 1

Naive approach → for every element traverse to the right of it and calculate frequency. You must also check, if that element is appeared at left; if it appeared at left, it must have processed, just skip

```
// at any point of time
// i, we check from
// 0 to i-1 if element
// is present or
// not.
```

Efficient solution

using an unordered map.

```
int countFreq (int arr[], int n){
    unordered_map<int> h;
    for (int x: arr).
        h[x]++;
    for (auto e: h).
        cout << e.first << " " << e.second;
}.
```

Time comp → O(n)
Aux space → O(n)

Intersection of Two Arrays

I/P a[] → {10, 15, 20, 15, 30, 30, 5}
b[] → {30, 5, 30, 80}.

O/P → {30, 5}.

Naive solution

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

check if item is processed before

find common elements

Time comp → O(m x (m+n)).

Efficient solution O(m+n).

idea is to use unordered_set

I/P → a[] = {10, 15, 20, 15, 30, 35, 5}
b[] = {30, 5, 30, 80}.

```
int intersection (int a[], int b[], int m, int n) {
    unordered_set<int> s;
    for (int i=0 ; i<m; i++)
        s.insert(a[i]);
    for (int j=0; j<n ; j++) {
        if (s.find(b[j]) != s.end()) {
            res++;
            s.erase(b[j]);
        }
    }.
    return res;
}
```

To ensure that duplicates are not allowed.

Union of two arrays

I/P → {15, 20, 5, 15}, {15, 15, 15, 20, 10}
O/P → 4 [15, 20, 5, 10]

Naive solution

* Make an auxiliary array c[m+n]
* Copy contents of A and B into auxiliary array
* And then count distinct elements in that array c[m+n]

Time comp → O((m+n)*(m+n))
Aux space → O(m+n).

Efficient Solution

We can insert all the elements in an unordered set and can simply return its size

```
int findUnion(int a[], int b[], int m, int n){
    unordered_set<int> s;
    for(int i=0; i<m; i++){
        s.insert(a[i]); }
    for(int i=0; i<n; i++){
        s.insert(b[j]); }
    return s.size();
}
```

Pair with a given sum in unsorted Array

I/P → {3, 2, 8, 5, -8}
sum → 17
O/P → Yes

Naive Approach

```
for(i → n){
  for(j = i+1 → n){
    if(arr[i]+arr[j] == sum)
        return true; }
}
return false;
```

Time comp → O(n×n)
Aux space → O(1)

Effective Solution

By using set

At any point before inserting an element ‘i’ into the set look for (sum − arr[i]) inside the set

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

Time-complexity → O(n)
Aux-space → O(n)

Subarray with zero sum

- contigous elements

I/P → {1, 4, 13, -3, -10, 5}
O/P → Yes
I/P → {4, -3, 2, 1}
O/P → {4 es}

Naive solution

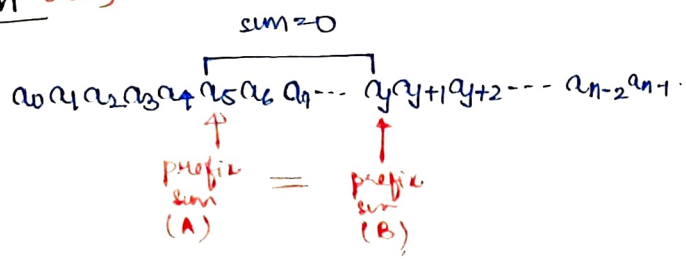
We start with current element as first element and then run loop for remaining elements to check if sum gets 0.

```
bool isOSubarray(int arr[], int n){
    for(int i=0; i<n; i++){
    int curr = arr[i];  int j = i+1;
        while(curr != 0 && j<n){
            curr = curr + arr[j];
            j++;
        }
        if(curr == 0)
            return true;
    }
    return false; }
```

Time comp - O(n²)

Effective solution O(n)

Prefix sum →



We will calculate prefix sum and if there is a ~~sum~~ subarray having 0 sum then the prefix sum of two elements will be equal (To check this we use hashing)

We will return true in two cases.

1. when the prefix sum we have calculated is already present in ~~the~~ set
2. when the calculated prefix sum comes to be 0 i.e first i elements sum up to 0.

```
bool isSubarray (int arr[]){
    unordered_set<int> s;
    int pre_sum = 0;
    for (int i = 0; i<n ; i++){
        pre_sum += arr[i];
        if (s.find(pre_sum)
            != s.end())        // check this cond^n
            return true;
        if (pre_sum == 0)
            return true;
        s.insert(pre_sum);
    }
    return false;
}
```

Subarray with given sum

- This is the extension of previous problem
- Instead ~~fo~~ of comparing prefix sum == 0 we compare

  prefix_sum == sum

- And we search for (prefix_sum − sum) in the unordered set

Longest subarray with given sum

I/P : arr[] = {5, 8, −4, −4, 9, −2, 2}. sum = 0

O/P : 3. (8, −4, −4)

I/P : arr[] = {3, 1, 0, 1, 8, 2, 3, 6}. sum = 5.

O/P : 5

I/P : arr[] = {8, 3, 7} sum = 15

O/P : 0 (No subarray is present).

Naive Solution O(n²)

- We will find the length of subarray. using (j − i + 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;
}
```

length = j − i + 1

0 1 2 3 4

2 − 0 + 1 = 3

Time comp → O(n²)

Effective solution

We will use unordered_map for this problem.
We will store prefix sum as well as index of first occurence of that prefix sum

arr[] → {8, 3, 1, 5, −6, 6, 2, 2}

pref-sum → {8, 11, 12, 17, 11, 17, 19, 21}

We will store occurence because we want to find the longest subarray.

```
int maxSub (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;
}
```

Longest subarray with equal 0's and 1's.

I/P a[] → {1, 0, 1, 1, 1, 0, 0}

O/P →

I/P → a[] → {1, 1, 1, 1}

O/P → 0

I/P → {0, 0, 1, 1, 1, 1, 1, 0}

O/P →

Naive solution

we maintain two pointers and calculate the number of 0's and 1's in the range.

arr[] → {1, 0, 1, 1, 1, 0, 0}

at certain point: i, j

co → 2

c1 → 3

O(n²) quadratic approach

```
int longestSum( bool arr[], int n) {
    int res = 0;
    for (int i = 0; i < n ; i++) {
        int c0 = c1 = 0;
        for (int j = i ; j < n ; j++) {
            if (arr[j] == 0)
                c0++;
            else
                c1++;
            if (c0 == c1)
                res = max (res, j - i + 1);
        }
    }
    return res;
}
```

Effective solution

Trick is to replace every 0 with -1 and calculate prefix sum and then find longest subarray with 0 sum.

a[] → {1, 0, 1, 1, 1, 0, 0}.

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

Longest common span with same sum in Binary Array

starting and ending indexes should be same in subarray & sum of elements of subarray must be equal.

arr1[] – { 0, 1, 0, 0, 0, 0 }
arr2[] – { 1, 0, 1, 0, 0, 1 }
O/P → 4

a1[] → {0, 1, 0, 1, 1, 1, 1}
a2[] → {1, 1, 1, 1, 1, 0, 1}
O/P → 5

a1[] → {0, 0, 0}
a2[] → {1, 1, 1}
O/P → 0.

Naive Approach

Run two pointers i and j on both arrays. and then check sum of both, if sum is equal then we get common subarray, to get maximum, we will store the result of each time and then return max of it

```
int maxCommon (bool arr1[], bool 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;
}
```

O(n²)

Effective solution

- We subtract one array (index to index) from the another and then calculate the subarray with zero sum

There will be 4 cases

① first array contains 1 & second also 1 → 0
② Ist array contains 0 & second contains 0 → 0
③ Ist array contains 1 & 2nd contain 0 → 1
④ Ist array contains 0 & 2nd contain 1 → -1

same sum
add of the sum 2

a1[] → {0, 1, 0, 0, 0, 0}
a2[] → {1, 0, 1, 0, 0, 1}
output → {-1, 1, -1, 0, 0, -1}.
Req. ans

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 → {20, 30, 40}
O/P → 1

Methode – 1

Sort the array and ~~upad~~ ~~upad~~ update curr-res when you see a consecutive value.

1, 9, 3, 4, 2, 20 → 1, 2, 3, 4, 9, 20
max → 4

Time comp → O(nlogn).

Method – 2

- create a hash table of size n and insert all elements into the hash table. and then traverse the array, for every element check if it is the starting point of sequence — if yes → look for elements after it
  — if no → ignore

Algorithm

- Insert all elements of array into h
- for (int i = 0 ; i < n ; i++)
  if (arr[i] - 1 is not present in h).
  curr = 1.
  while (h contains arr[i] + curr)
  curr++ ;
  res = max (res , curr) ;

return res .

Count Distict Elements in every window of size k

I/P → arr[] = {10, 20, 20, 10, 30, 40, 10}.

Effective approach → use map

- create a frequency map of first k elements
- and then get the size of it
- on moving k further
- decreament the frequency of freq[arr[i]] i is before start of window
- if freq become 0 remove element.

```
vector<int> countDistinct (int arr[], int n, int k){
    unordered_map<int, int> freq;
    for(int i=0; i<k; i++) {
        freq.insert ({arr[i], i});
}
```

```
vector <int> countDistinct (int arr[], int n, int k) {
    unordered_map<int, int > freq;   vector<int> res;
    for (int i=0; i<k ; i++)
        freq[arr[i]]++;
    for (int i = k ; i<n ; i++){
        freq[arr[i-k]]--;
        if (freq[arr[i-k]] == 0)
            freq.erase (arr[i-k]) ;
        freq[arr[k]]++;
        res.push_back ( freq.size());
    }.
    return res;
}
```

More than (n/k) occurances

I/P → arr[] = {30, 10, 20, 20, 10, 20, 30, 30}
k = 4 n = 8
O/P = {20, 30} 8/4 = 2

Method-1

Sort the array and then check the count of every element if n/k < count we print that element

Time complexity → O(nlogn)

Method-2

Build and frequency map and then return those element whose frequency > n/k.

Time complexity → O(n)

Effective solution.

- we will use the concept of moore's voting algorithm.
- fact is res_count $\leq k-1$

let's suppose every element occurs n/k times

$$k * \left(\frac{n}{k}+1\right) \leq n$$

$$n+k \leq n \quad \text{(contradicting)}$$

* Algorithm

- create an empty map m
- for (i=0 ; i<n ; i++)
  - (a) if (m contains arr[i])
    m[arr[i]]++;
  - (b) if (m.size() is less than (k-1))
    m.put(arr[i],1).
  - (c) else decrease all values in m by one and if value becomes 0 remove it.
- For all elements in m, print the elements that actually appear more than n/k times.

$O(nk)$

STRING

- Sequence of characters
- small set
- contiguous integer value. 'a' to 'z' and 'A' to 'Z' in both ASCII and UTF-16

C/C++	Java
char:- ASCII	UTF-16
8 Bit	16 Bit

Also supports unicode

```
char x = 'a'
cout << (int)x;   //97
```

Print the frequencies of character in sorted order

```
string str = "geeksforgeeks";
int count[26] = {0};
for(int i=0; i< str.length(); i++)
    count[str[i] - 'a']++;
for(int i=0; i<26; i++){
    if(count[i] > 0){
        cout <<(char)(i+'a')<<" ";
        cout << count[i];
    }
}
```

O/P

e	4
f	1
g	2
k	2
o	1
r	1
s	2

Strings in C++ (char arrays)

```
char str[] = "gfg"
```

compiles → | g | f | g | \0 |

if we give definite size

```
str[6] = "gfg"
```

g	f	g	\0	\0	\0

func

- strlen → gives length of string