Hashing

Hashing is a method of storing and retrieving data from a database efficiently.

Suppose that we want to design a system for storing employee records keyed using phone numbers. And we want the following queries to be performed efficiently:

1. Insert a phone number and the corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and the related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. An array of phone numbers and records.
2. A linked list of phone numbers and records.
3. A balanced binary search tree with phone numbers as keys.
4. A direct access table.

For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(Logn) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

With a **balanced binary search tree**, we get a moderate search, insert and delete time. All of these operations can be guaranteed to be in O(Logn) time.

Another solution that one can think of is to use a **direct access table** where we make a big array and use phone numbers as indexes in the array. An entry in the array is NIL if the phone number is not present, else the array entry stores pointer to records corresponding to the phone number. Time complexity wise this solution is the best of all, we can do all operations in O(1) time. For example, to insert a phone number, we create a record with details of the given phone number, use the phone number as an index and store the pointer to the record created in the table.
This solution has many practical limitations. The first problem with this solution is that the extra space required is huge. For example, if the phone number is of n digits, we need $O(m * 10^n)$ space for the table where m is the size of a pointer to the record. Another problem is an integer in a programming language may not store n digits.

Due to the above limitations, the Direct Access Table cannot always be used. **Hashing** is the solution that can be used in almost all such situations and performs extremely well as compared to above data structures like Array, Linked

List, Balanced BST in practice. With hashing, we get O(1) search time on average (under reasonable assumptions) and O(n) in the worst case.

Hashing is an improvement over Direct Access Table. The idea is to use a hash function that converts a given phone number or any other key to a smaller number and uses the small number as an index in a table called a hash table.

Hash Function: A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as an index in the hash table.
A good hash function should have following properties:

1. It should be efficiently computable.
2. It should uniformly distribute the keys (Each table position be equally likely for each key).

For example, for phone numbers, a bad hash function is to take the first three digits. A better function will consider the last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a big key, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

- **Chaining:**The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple, but it requires additional memory outside the table.
- **Open Addressing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine the table slots until the desired element is found or it is clear that the element is not present in the table.

Open Addressing: Like separate chaining, open addressing is a method for handling collisions. 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).

Important Operations:

- Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- Search(k): Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.
- Delete(k): ***Delete operation is interesting***. If we simply delete a key, then the search may fail. So slots of the deleted keys are marked specially as "deleted".

Insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

Open Addressing is done in the following ways:

1. ***Linear Probing:*** In linear probing, we linearly probe for the next slot. For example, the typical gap between the two probes is 1 as taken in the below example also.
   let **hash(x)** be the slot index computed using a hash function and **S** be the table size.

```
If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
..................................................
..................................................
```

1. **Clustering:** The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.
2. ***Quadratic Probing*** We look for i^2'th slot in i'th iteration.

   let hash(x) be the slot index computed using hash function.
   If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S
   If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S
   If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S
   ....
   ....

3. **Double Hashing** We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation.

   let hash(x) be the slot index computed using hash function.
   If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
   If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S
   If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S
   ....
   ....

   See this for step by step diagrams.

Comparison of above three:

- Linear probing has the best cache performance but it suffers from clustering. One more advantage of Linear probing that it is easy to compute.
- Quadratic probing lies between the two in terms of cache performance and clustering.
- Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

S.No.	**Seperate Chaining**	**Open Addressing**
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care for to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

Performance of Open Addressing: 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 (simple uniform hashing).

```
m = Number of slots in the hash table
n = Number of keys to be inserted in the hash table

Load factor α = n/m  ( < 1 )

Expected time to search/insert/delete < 1/(1 - α)

So Search, Insert and Delete take (1/(1 - α)) time
```

Binary Search Iterative

Hashing in C++ STL

Hashing in C++ can be implemented using different containers present in STL as per the requirement. Usually, STL offers the below-mentioned containers for implementing hashing:

- set
- unordered_set
- map
- unordered_map

Let us take a look at each of these containers in details:

set

Sets are a type of associative containers in which each element has to be unique, because the value of the element identifies it. The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.

Sets are used in the situation where it is needed to check if an element is present in a list or not. It can also be done with the help of arrays, but it would take up a lot of space. Sets can also be used to solve many problems related to sorting as the elements in the set are arranged in a sorted order.

Some basic functions associated with Set:

- **begin()** – Returns an iterator to the first element in the set.
- **end()** – Returns an iterator to the theoretical element that follows last element in the set.
- **size()** – Returns the number of elements in the set.
- **insert(val)** – Inserts a new element *val* in the Set.
- **find(val)** - Returns an iterator pointing to the element *val* in the set if it is present.
- **empty()** – Returns whether the set is empty.

unordered_set

The unordered_set container is implemented using a hash table where keys are hashed into indices of this hash table so it is not possible to maintain any order. All operation on unordered_set takes constant time O(1) on an average which can go up to linear time in the worst case which depends on the internally used hash function but practically they perform very well and generally provide constant time search operation.

The unordered-set can contain key of any type – predefined or user-defined data structure but when we define key of a user-defined type, we need to specify our comparison function according to which keys will be compared.

set vs unordered_set

- Set is an ordered sequence of unique keys whereas unordered_set is a set in which key can be stored in any order, so unordered.
- Set is implemented as a balanced tree structure that is why it is possible to maintain order between the elements (by specific tree traversal). The time complexity of set operations is O(Log n) while for unordered_set, it is O(1).

Note: Like set containers, the Unordered_set also allows only unique keys.

Map container

As a set, the Map container is also associative and stores elements in an ordered way but Maps store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have the same key values.

Some basic functions associated with Map:

- **begin()** – Returns an iterator to the first element in the map.
- **end()** – Returns an iterator to the theoretical element that follows last element in the map.
- **size()** – Returns the number of elements in the map.
- **empty()** – Returns whether the map is empty.
- **insert({keyvalue, mapvalue})** – Adds a new element to the map.
- **erase(iterator position)** – Removes the element at the position pointed by the iterator
- **erase(const g)**– Removes the key value ‘g’ from the map.
- **clear()** – Removes all the elements from the map.

unordered_map Container

The unordered_map is an associated container that stores elements formed by a combination of key value and a mapped value. The key value is used to uniquely identify the element and mapped value is the content associated with the key. Both key and value can be of any type predefined or user-defined.
Internally unordered_map is implemented using Hash Table, the key provided to map are hashed into indices of a hash table that is why the performance of data structure depends on hash function a lot but on an average, the cost of search, insert and delete from hash table is O(1).

unordered_map vs unordered_set

In unordered_set, we have only key, no value, these are mainly used to see presence/absence in a set. For example, consider the problem of counting frequencies of individual words. We can't use unordered_set (or set) as we can't store counts.

unordered_map vs map

map (like set) is an ordered sequence of unique keys whereas in the unordered_map key can be stored in any order, so unordered.
A map is implemented as a balanced tree structure that is why it is possible to maintain order between the elements (by specific tree traversal). The time complexity of map operations is O(Log n) while for unordered_set, it is O(1) on average.

Applicaton of Hashing:

1) **dictionary**
2) **database indexing**
3) **cryptography**
4) **caches**
5) **symbol table in compiler and interpreter**
6) **router**
7) **getting data from database and many more**

Direct Address Table

```cpp
#include<iostream>
using namespace std;

struct DAT
{
    int table[1000]={0};

    void insert(int i){
        table[i]=1;

    }

    void del(int i){
        table[i]=0;
    }

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

};


int main(){
    DAT dat;
    dat.insert(10);
    dat.insert(30);
    dat.insert(119);
    cout<<dat.search(10)<<"\n";
    cout<<dat.search(30)<<"\n";
    dat.del(119);
    cout<<dat.search(119);

    return 0;
}
```

OUTPUT:

```
1

1

0
```

Implementation Of Chaining

```cpp
#include<iostream>
#include<list>
using namespace std;
struct MyHash
{
    int BUKCET;
    list<int> *table;
    MyHash(int b){
        BUKCET=b;
        table= new list<int>[BUKCET];
    }

    void insert (int k){
        int i=k%BUKCET;
        table[i].push_back(k);
    }

    bool search(int k){
        int i=k%BUKCET;
        for(auto x: table[i])
            if(x==k)
                return true;
        return false;
    }

    void remove(int k){
        int i=k%BUKCET;
        table[i].remove(k);
    }

};
int main(){
    MyHash mh(7);
    mh.insert(10);
    mh.insert(20);
    mh.insert(22);
    mh.insert(15);
    mh.insert(7);
    cout<<mh.search(10)<<endl;
    mh.remove(15);
    cout<<mh.search(15);
}
```

OUTPUT: 1

0

Implementation Of Open Addressing

```cpp
#include<iostream>
using namespace std;

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;
    }

    int hash(int key){
        return key%cap;
    }

    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 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;
    }

    bool earse(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;
        }
    }


};

int main(){
    MyHash mh(7);
    mh.insert(49);
    mh.insert(56);
    mh.insert(72);
    if(mh.search(56)==true){
        cout<<"Yes"<<endl;
    }
    else{
        cout<<"No"<<endl;
    }
}
```

OUTPUT :

```
Yes
```

Unordered_set in C++ STL

```cpp
// The auto keyword specifies that the type of the variable that is
// being declared will be automatically deducted from its initializer.
// In case of functions, if their return type is auto then that will be
//  evaluated by return type expression at runtime

#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<<" ";

    cout<<endl;
    for(auto it=s.begin(); it!=s.end(); it++)
        cout<<*it<<" ";
    cout<<endl;
    cout<<s.size()<<endl;
    s.clear();
    cout<<s.size()<<endl;

    s.insert(10);
    s.insert(5);
    s.insert(15);
    s.insert(20);
    cout<<s.size()<<endl;

    if(s.find(15)==s.end())
        cout<<"Not Found";
    else
        cout<<"Found "<<(*s.find(15));

    cout<<endl;
    if(s.count(15))
        cout<<"Found";
    else
        cout<<"Not Found";
    cout<<endl;

    cout<<s.size()<<endl;
    s.erase(15);
    cout<<s.size()<<endl;
    auto it=s.find(10);
    s.erase(it);
    cout<<s.size()<<endl;

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

    return 0;

}
```

OUTPUT:

```
20 15 10 5

20 15 10 5

4

0

4

Found 15

Found

4

3

2
```

Unordered_map in C++ STL

```cpp
// The auto keyword specifies that the type of the variable that is
// being declared will be automatically deducted from its initializer.
// In case of functions, if their return type is auto then that will be
//  evaluated by return type expression at runtime
#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});

    if(m.find("ide")!=m.end())
        cout<<"Found";
    else
        cout<<"Not Found";

    cout<<endl;

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

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

    cout<<endl;

    cout<<m.size()<<endl;
    m.erase("ide");
    m.erase(m.begin());
    cout<<m.size()<<endl;
    m.erase(m.begin(),m.end());

    return 0;
}
```

OUTPUT:

```
Found

gfg 20

courses 15

ide 30

Found

3

1
```

Count Distinct Elements

Naïve :

```
//time complexity o(n^2) and auxilary space O(1)

#include<iostream>
using namespace std;

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;
}

int main()
{
    int arr[]={10,20,30,10,20,30,40,60,40};
    int n=sizeof(arr)/sizeof(arr[0]);
    cout<<countDist(arr,n);

    return 0;
}
```

OUTPUT:

```
5
```

Efficient for Count Distinct element :

```
//time compelxity theta(n) and auxiliary space O(n)

#include<iostream>
#include<unordered_set>
using namespace std;

int countDistinct(int arr[], int n)
{
    unordered_set<int>s(arr,arr+n);

    return s.size();
}

int main()
{
    int arr[]={15,16,27,27,28,15};
    int n=sizeof(arr)/sizeof(arr[0]);

    cout<<countDistinct(arr,n);

    return 0;
}
```

OUTPUT:

```
4
```

Frequencies of array Element :

Naïve :

```
//time complexity O(n^2) and auxiliary space O(1)
#include<iostream>
using namespace std;

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;
    }
}

int main()
{
    int arr[]={10,20,30,10,20,30,40,60,40};
    int n=sizeof(arr)/sizeof(arr[0]);
    printFreq(arr,n);

    return 0;
}
```

OUTPUT: 10 2   \n 20 2 \n30 2 \n40 2

60 1

Efficient for Frequencies of array :

```cpp
#include<iostream>
#include<unordered_map>
using namespace std;

void 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;
}

int main()
{
    int arr[]={10,20,30,10,20,30,40,60,40};
    int n=sizeof(arr)/sizeof(arr[0]);
    countFreq(arr,n);

    return 0;
}
```

OUTPUT:

```
60 1

40 2

30 2

10 2

20 2
```

Intersection of two Unsorted Arrays :

Naïve :

```
#include<iostream>
using namespace std;

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(a[i]==a[j])
            {
                flag=true;
                break;
            }
        }

        if(flag==true)
            continue;
        for(int j=0;j<n;j++)
        {
            if(a[i]==b[j])
            {
                res++;
                break;
            }
        }
    }
    return res;
}

int main()
{
    int arr1[] = {15, 17, 27, 27, 28, 15};
    int arr2[] = {16, 17, 28, 17, 31, 17};
    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);

    cout << intersection(arr1,arr2, m, n);

    return 0;
}
OUTPUT:  2
```

Efficient 1 for intersection of two unsorted array :

```
//1. insert all elements of a[] in a set theta(m)
//2. insert all elements of b[] in a set anohter set theta(n)
//3. now traverse s.a  and increment count for elements
//   that are present in s.b  o(m)

//overall time complexity is theta(m+n) and auxiliary space theta(m+n)

#include<iostream>
#include<unordered_set>
using namespace std;

int intersection(int arr1[], int m, int arr2[], int n)
{
    unordered_set<int> us;
    for(int i=0;i<m;i++)
        us.insert(arr1[i]);

    int res=0;
    for(int i=0;i<n;i++)
    {
        if(us.find(arr2[i])!=us.end())
        {
            res++;
            us.erase(arr2[i]);
        }
    }

    return res;
}

int main()
{
    int arr1[] = {15, 17, 27, 27, 28, 15};
    int arr2[] = {16, 17, 28, 17, 31, 17};
    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);

    cout << intersection(arr1, m, arr2, n);

    return 0;
}
```

OUTPUT:

```
2
```

Efficient 2 for intersection of two unsorted array :

```cpp
//1. insert all elements of a[] in a set theta(m)
//2. traverse through b[]. search for every elements
//   b[i] in s.a  if present
        // a) increment res
        // b) remove b[i]  form  s.a
#include<iostream>
#include<unordered_set>
using namespace std;

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

    return res;
}

int main()
{
    int arr1[] = {15, 17, 27, 27, 28, 15};
    int arr2[] = {16, 17, 28, 17, 31, 17};
    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);

    cout << intersection(arr1,arr2, m, n);

    return 0;
}
```

OUTPUT:

```
2
```

Union of two Unsorted Arrays :

Efficient :

```
//1. create an  empty hash table , h
//2. insert all elements of a[] in h.  theta(m)
//3. insert all elements of b[] in h.  theta(n)
//4. return h.size()
//overall time complexity theta(m+n) and ausiliary space O(m+n)


#include<iostream>
#include<unordered_set>
using namespace std;

int unionSize(int arr1[], int m, int arr2[], int n)
{
    unordered_set<int> us;
    for(int i=0;i<m;i++)
        us.insert(arr1[i]);
    for(int i=0;i<n;i++)
        us.insert(arr2[i]);

    return us.size();
}

int main()
{
    int arr1[] = {2, 8, 3, 5, 6};
    int arr2[] = {4, 8, 3, 6, 1};
    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);

    cout << unionSize(arr1, m, arr2, n);
}
```

OUTPUT:

```
7
```

Improved Efficient :

```
#include<iostream>
#include<unordered_set>
using namespace std;

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();
}

int main()
{
    int arr1[] = {2, 8, 3, 5, 6};
    int arr2[] = {4, 8, 3, 6, 1};
    int m = sizeof(arr1)/sizeof(arr1[0]);
    int n = sizeof(arr2)/sizeof(arr2[0]);

    cout << unionCount(arr1, arr2, m, n);
}
```

OUTPUT:

```
7
```

Pair with given sum in Unsorted Array :

Naïve:

```cpp
//there are n*(n-1)/2 pair
//time completixy o(n^2) and auxiliary space o(1)

#include<iostream>
using namespace std;
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;
}

int main()
{
    int arr[] = {3, 8, 4, 7, 6, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    int X = 14;

    cout<<isPair(arr, n, X);
    return 0;
}
```

OUTPUT:

```
1
```

Efficient for pair with sum in unsorted array :

```cpp
// time complexity o(n) and auxiliary space o(n)
#include<iostream>
#include<unordered_set>
using namespace std;

int pairWithSumX(int arr[], int n, int X)
{
    unordered_set<int> us;
    for(int i=0;i<n;i++)
    {
        if(us.find(X-arr[i])!=us.end())
            return 1;

        us.insert(arr[i]);
    }

    return 0;
}

int main()
{
    int arr[] = {3, 8, 4, 7, 6, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    int X = 14;

    cout << pairWithSumX(arr, n, X);
    return 0;
}
```

OUTPUT:

```
1
```

Subarray With Zero Sum :

Naïve :

```
//time complexity o(n^2)
#include<iostream>
using namespace std;

bool isZeroSubarray(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;
}

int main()
{
    int arr[] = {5, 3, 9, -4, -6, 7, -1};
    int n = sizeof(arr)/sizeof(arr[0]);

    cout << isZeroSubarray(arr, n);

    return 0;
}
```

OUTPUT:

```
1
```

Efficient for Subarray with zero sum :

```cpp
// time complexity is o(n)
#include<iostream>
#include<unordered_set>
using namespace std;

int ZeroSumSubarray(int arr[], int n)
{
    unordered_set<int> us;
    int prefix_sum=0;
    us.insert(0);
    for(int i=0;i<n;i++)
    {
        prefix_sum+=arr[i];
        if(us.find(prefix_sum)!=us.end())
            return 1;
        us.insert(prefix_sum);
    }

    return 0;
}

int main()
{
    int arr[] = {5, 3, 9, -4, -6, 7, -1};
    int n = sizeof(arr)/sizeof(arr[0]);

    cout << ZeroSumSubarray(arr, n);
}
```

OUTPUT :

```
1
```

Subarray With given Sum :

Naïve :

```cpp
//time complexity o(n^2) and auxiliary space o(1)

#include<iostream>
using namespace std;

bool isSum(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;
}

int main()
{
    int arr[] = {5, 8, 6, 13, 3, -1};
    int sum=22;
    int n = sizeof(arr)/sizeof(arr[0]);

    cout << isSum(arr, n, sum);
}
```

OUTPUT :

```
1
```

Efficient for Subarray with Given sum :

```cpp
#include<iostream>
#include<unordered_set>
using namespace std;

bool isSum(int arr[], int n, int sum)
{
    unordered_set<int> s;
    int pre_sum=0;
    for(int i=0;i<n;i++)
    {
        if(pre_sum==sum)
            return true;
        pre_sum+=arr[i];
        if(s.find(pre_sum-sum)!=s.end())
            return true;   //use x+y=z i.e x=z-y property

        s.insert(pre_sum);
    }

    return false;
}

int main()
{
    int arr[] = {5, 8, 6, 13, 3, -1};
    int sum=22;
    int n = sizeof(arr)/sizeof(arr[0]);

    cout << isSum(arr, n, sum);
}
```

OUTPUT :

```
1
```

Longest Subarray With given Sum :

Naïve :

```cpp
// time complexity theta(n^2) and auxiliary space O(1)

#include<iostream>
using namespace std;

int largestSubarrayWithSumX(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;
}

int main()
{
    int arr[] = {5,2,3,4};
    int n = sizeof(arr)/sizeof(arr[0]);
    int sum = 5;

    cout << largestSubarrayWithSumX(arr, n, sum);

}
```

OUTPUT :

```
2
```

Efficient for Longest Subarray with Given sum :

```
//time complexity theta(n) auxilary space o(n)
#include<iostream>
#include<unordered_map>
using namespace std;

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

    return res;
}

int main()
{
    int arr[] = {8, 3, -7, -4, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    int sum = -8;

    cout << largestSubarrayWithSumX(arr, n, sum);

}
```

OUTPUT :

```
3
```

Longest Subarray With Equal number of 0s and 1s :

Naïve :

```
//time complexity theta (n^2) and auxilary space o(1)
#include<iostream>
using namespace std;

int largestZeroSubarray(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;
}

int main()
{
    int arr[] = {1, 1, 1, 0, 1, 0, 1, 1, 1};
    int n = sizeof(arr)/sizeof(arr[0]);

    cout << largestZeroSubarray(arr, n);
}
```

OUTPUT:

```
4
```

Efficient for Longest Subarray with Equal number of 0s and 1s :

```
#include<iostream>
#include<unordered_map>
using namespace std;

int largestZeroSubarray(int arr[], int n)
{
    for(int i=0;i<n;i++)
        arr[i]=(arr[i]==0) ? -1: 1;

    unordered_map<int, int>ump;
    int sum=0,maxLen=0;
    for(int i=0;i<n;i++)
    {
        sum+=arr[i];
        if(sum==0)
            maxLen=i+1;
        if(ump.find(sum+n)!=ump.end())
        {
            if(maxLen<i-ump[sum+n])  //saglay tikali sum+n na use karata
                maxLen=i-ump[sum+n];  //fkt sum use kel tar better aahe
        }
        else
            ump[sum+n]=i;
    }
    return maxLen;
}

int main()
{
    int arr[] = {1, 1, 1, 0, 1, 0, 1, 1, 1};
    int n = sizeof(arr)/sizeof(arr[0]);

    cout << largestZeroSubarray(arr, n);
}
```

OUTPUT :

```
4
```

Efficient 2 for Longest Subarray with Equal number of 0s and 1s

BY ME:

```
#include<iostream>
#include<unordered_map>
using namespace std;

int largestZeroSubarray(int arr[], int n)
{
    for(int i=0;i<n;i++)
        arr[i]=(arr[i]==0) ? -1: 1;

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

    return res;
}

int main()
{
    int arr[] = {1, 1, 1, 0, 1, 0, 1, 1, 1};
    int n = sizeof(arr)/sizeof(arr[0]);

    cout << largestZeroSubarray(arr, n);
}
```

OUTPUT :

```
4
```

Longest Common span with same sum in binary array :

Naïve :

```
//time complexity theta(n^2) and auxiliary space o(1)

#include<iostream>
using namespace std;

int longestCommonSum(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;
}

int main()
{
    int arr1[] = {0, 1, 0, 1, 1, 1, 1};
    int arr2[] = {1, 1, 1, 1, 1, 0, 1};
    int n = sizeof(arr1)/sizeof(arr1[0]);
    cout << longestCommonSum(arr1, arr2, n);
    return 0;
}
```

OUTPUT :

```
6
```

Efficient for Longest Common span with same sum in binary array :

```
//time complexity theta(n) and auxiliary space theta(n)
#include<iostream>
#include<unordered_map>
using namespace std;

int longestCommonSum(bool arr1[], bool arr2[], int n)
{
    int arr[n];
    for(int i=0;i<n;i++)
        arr[i]=arr1[i]-arr2[i];

    unordered_map<int,int>uom;

    int sum=0, maxLen=0;

    for(int i=0;i<n;i++)
    {
        sum+=arr[i];

        if(sum==0)
            maxLen=i+1;
        if(uom.find(sum)!=uom.end())
            maxLen=max(maxLen,i-uom[sum]);
        else
            uom[sum]=i;
    }

    return maxLen;
}

int main()
{
    bool arr1[] = {0, 1, 0, 1, 1, 1, 1};
    bool arr2[] = {1, 1, 1, 1, 1, 0, 1};
    int n = sizeof(arr1)/sizeof(arr1[0]);
    cout << longestCommonSum(arr1, arr2, n);
    return 0;
}
```

```
/*
// C++ program to find largest subarray
// with equal number of 0's and 1's.
#include <bits/stdc++.h>
using namespace std;

// Returns largest common subarray with equal
// number of 0s and 1s in both of t
int longestCommonSum(bool arr1[], bool arr2[], int n)
{
    // Find difference between the two
    int arr[n];
    for (int i=0; i<n; i++)
    arr[i] = arr1[i] - arr2[i];

    // Creates an empty hashMap hM
    unordered_map<int, int> hM;

    int sum = 0;     // Initialize sum of elements
    int max_len = 0; // Initialize result

    // Traverse through the given array
    for (int i = 0; i < n; i++)
    {
        // Add current element to sum
        sum += arr[i];

        // To handle sum=0 at last index
        if (sum == 0)
            max_len = i + 1;

        // If this sum is seen before,
        // then update max_len if required
        if (hM.find(sum) != hM.end())
        max_len = max(max_len, i - hM[sum]);

        else // Else put this sum in hash table
            hM[sum] = i;
    }

    return max_len;
}

// Driver progra+m to test above function
int main()
{
    bool arr1[] = {0, 1, 0, 1, 1, 1, 1};
    bool arr2[] = {1, 1, 1, 1, 1, 0, 1};
    int n = sizeof(arr1)/sizeof(arr1[0]);
    cout << longestCommonSum(arr1, arr2, n);
    return 0;
}

*/
```

OUTPUT :

```
6
```

Longest Consecutive Subsequence :

Sorting naïve :

```cpp
// we need to find the longest subsequence that has consecutive elements.
//  These consecutive elements may appear in any order in the subsequence.

//time complexity o(nlogn)
#include<bits/stdc++.h>
using namespace std;
int findLongest(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
        {
            res=max(res,curr);
            curr=1;
        }
    }

    res=max(res,curr);
    return res;
}

int main()
{
    int arr[] = {1, 9, 3, 4, 2, 10, 13};

    int n = sizeof(arr)/sizeof(arr[0]);

    cout << findLongest(arr, n);
}
```

OUTPUT :

```
4
```

Efficient for Longest Consecutive Subsequence :

```cpp
// we need to find the longest subsequence that has consecutive elements.
//  These consecutive elements may appear in any order in the subsequence.

//time complexity theta(n) and auxiliary space o(n)
//there are always 2n lookups at most
//for first elements : 2+(len-1)
//for other elements : 1
//length is the length of the subsequence with the given as first
#include<iostream>
#include<unordered_set>
using namespace std;

int findLongest(int arr[], int n)
{
    unordered_set<int> s;
    int res=0;

    for(int i=0;i<n;i++)
        s.insert(arr[i]);

    for(int i=0;i<n;i++)
    {
        if(s.find(arr[i]-1)==s.end())
        {
            int curr=1;
            while(s.find(curr+arr[i])!=s.end())
                curr++;

            res=max(res,curr);
        }
    }

    return res;
}

int main()
{
    int arr[] = {1, 9, 3, 4, 2, 10, 13};

    int n = sizeof(arr)/sizeof(arr[0]);

    cout << findLongest(arr, n);
}
```

OUTPUT :   4

Count Distinct Element in every window :

Naïve :

```cpp
//Given an array, one needs to Count Distinct Elements In Every Window of Size
 K.
// time complexity o((n-k)*k*k)
#include<iostream>
using namespace std;

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<<" ";

    }
}

int main()
{
    int arr[] = {10, 10, 5, 3, 20, 5};

    int n = sizeof(arr)/sizeof(arr[0]);
    int k=4;
    printDistinct(arr, n, k);
}
```

OUTPUT : 3 4 3

Efficient for Count Distinct Element in every window :

```
//time complexity o(n) and auxiliary space o(k)
#include<iostream>
#include<map>
using namespace std;

void printDistinct(int arr[], int n, int k)
{
    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()<<" ";
    }
}

int main()
{
    int arr[] = {10, 10, 5, 3, 20, 5};

    int n = sizeof(arr)/sizeof(arr[0]);
    int k=4;
    printDistinct(arr, n, k);
}
```

OUTPUT :

```
3 4 3
```

More than n/k occurances

Naïve:

```
#include<bits/stdc++.h>
using namespace std;

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)
            cout<<arr[i-1]<<" ";

        count=1;
        i++;
    }
}

int main()
{
    int arr[]={10,10,20,30,20,10,10};
    int n=sizeof(arr)/sizeof(arr[0]);
    int k=2;
    printNByK(arr,n,k);
}
```

OUTPUT:

```
10
```

Efficient for More than n/k occurances :

```
#include<bits/stdc++.h>
using namespace std;

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<<" ";

}

int main()
{
    int arr[]={10,10,20,30,20,10,10};

    int n=sizeof(arr)/sizeof(arr[0]);

    int k=2;

    printNByK(arr,n,k);
}
```

OUTPUT :

```
10
```

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

```cpp
#include <bits/stdc++.h>
using namespace std;

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<<" ";
    }
}

int main()
{
    int arr[] = {30, 10, 20, 20, 20, 10, 40, 30, 30};

    int n = sizeof(arr)/sizeof(arr[0]);
    int k=4;
    printNByK(arr, n, k);
}
```

OUTPUT :

```
30 20
```