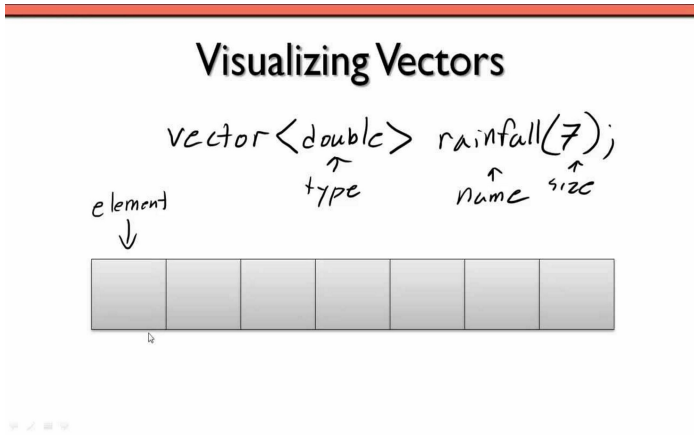
OVERVIEW OF STL

18 January 2022 08:43

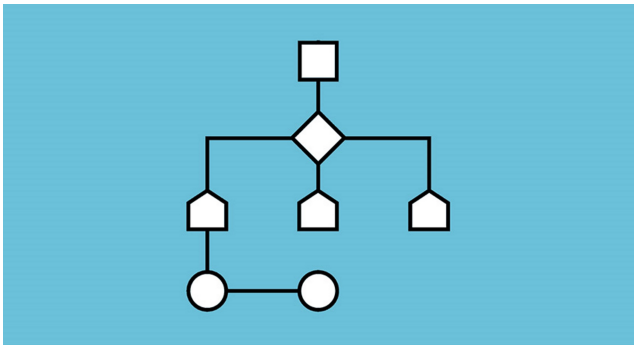
Header file used should be #include<bits/stdc++.h>
using namespace std;

- **Container**
  - Sequential
    - Vectors
    - Stack
    - Queue
    - Pair (not a container, a class in CPP)
  - Ordered (the values stored are in some order)
    - Maps
    - Multimap
    - Set
    - Multiset
  - Unordered (the hash values are stored)
    - Unordered map
    - Unordered set

Nested containers are hard to learn but very useful in implementing complex algorithm.



- **Iterators**
  - Are used to point to memory address of the containers
- **Algorithms**
  - Upper bound
  - Lower bound
  - Sort (comparator)
  - Max-element
  - Min-element
  - Reverse
  - Etc.



- **Functors** (classes which can act as functions)(not so important in CP)

PAIRS AND VECTORS

18 January 2022 10:30

Pair:

- Initialisation: **pair<datatype1,datatype2> variable_name;**
  - Example: pair<int,string> p;
- Assigning values: **variable_name=make_pair(value1, value2);** or **variable_name={value1,value2};**
  - Example: p=make_pair(2,"abc"); or p={2,"abc"};
- **A pair value can be directly assigned using another pair. And it can also assigned as reference value.**
- Accessing values: **p.first, p.second** refers to first and second values respectively.
- Pairs are mostly used when there is a requirement of array with two values to store with same/different datatypes. For example, if an array of elements are given and we have to store 0 and 1 if they are even or odd respectively along with the values given. **pair<int,int> p[10]**
- Then these values can be easily manipulated like **swap(p[0],p[9]);**
- **Pairs can be used as similar as variables in cin and cout.**

Vectors (array with dynamic size):

- Initialisation: **vector<datatype> variable_name;**
  - Example: vector<int> v;
- Assigning values: **variable_name.push_back(value);**
  - Example: for(int i; i<n;i++) {cin>>x; v.push_back(x);}

input

- **A vector value can be directly assigned using another vector. And it can also assigned as reference value.**
- Accessing values: **variable_name[i]** refers to the ith element in the vector.
  - Example: for (int i;i<v.size():i++) cout<<v[i]<<" ";

returns the vector size

- For removing the last element we have a function, **variable_name.pop_back()**
  - Example: v.pop_back();
- **vector<datatype> variable_name(size,value);**

fills with the following value

- **Unlike arrays vectors are by default assigned without reference.** So try to pass it with reference while the values doesn't need to change, because it takes O(n) complexity to copy the entire vector.

Nested Vector:

- Vector of Pair: When declaring vector the datatype used is pair which was discussed above, hence we get a vector of pair.
- Declaring vector of pair: **vector<pair<datatype1,datatype2>> variable_name;**
  - Example: **vector<pair<int,int>> v={{1,2},{3,4}};**
- We can also put the value using **v.push_back({5,6})** or by passing a similar pair datatype.
- Array of vectors: As the name suggests it is a multiple number of vectors under one roof, **vector<int> variable_name[size];**
- Example: **vector<int> v[10];** (makes ten vectors), here the number of rows are fixed and the column size is dynamic. It works like a two dimensional array.
- We can add elements using **v[n].push_back(value)**, and get access of the element using **v[n][m]**.
- Vector of vectors:

```
vector<vector<int> > v;
for(int i = 0; i < N; ++i){
    int n;
    cin >> n;
    vector<int> temp;
    for(int j = 0; j < n; ++j){
        int x;
        cin >> x;
        temp.push_back(x);
    }
    v.push_back(temp);
}
```

```
vector<vector<int> > v;
for(int i = 0; i < N; ++i){
    int n;
    cin >> n;
    v.push_back(vector<int> ());
    for(int j = 0; j < n; ++j){
        int x;
        cin >> x;
        v[i].push_back(x);
    }
}
```

- The above two examples show how to initialise vector of vectors, It should be evident from the code that we can't assign value without making the first vector.
- Here the number of rows and columns are dynamic in nature as both them are vectors.

ITERATORS

19 January 2022 10:56

begin tells the address of first element and end can be used for the next to last elements address.

- Initialisation: **vector<datatype> :: iterator variable_name;**
  - Example: **vector<int> :: iterator it=v.begin();**
- Accessing the value: the value stored in the iterator can be accessed using ***variable_name**.
- For accessing the next element we can use **variable_name++** or **variable_name+1**.
- In the above two examples of accessing the values, the latter one can't be used for maps.
- In case of vector of pairs we can access the elements using two ways, **(*it).first** or **(it->first)**

- Using for-loop for going through all the element using usual way can make the code bigger Instead we can use another shortcut to reduce the code as follows:
  for(datatype variable_name1: variable_name) {code}
- There is a keyword which can detect the datatype of a variable, called **auto**. This can be used to shorten the iterator code just by writing the auto keyword.

```cpp
int main(){
    vector<int> v = {2,3,5,6,7};
    for(int i = 0; i < v.size(); ++i){
        cout << v[i] << " ";
    }
    cout << endl;
    vector<int> ::iterator it = v.begin();
    for(it = v.begin(); it != v.end(); ++it){
        cout << (*it) << endl;
    }

}
```

```cpp
cout << endl;
for(auto it = v.begin(); it != v.end(); ++it){
    cout << (*it) << " ";
}
cout << endl;
vector<pair<int,int >  > v_p = {{1,2}, {2,3}};
for(auto &value : v_p){
    cout << value.first << " " << value.second
}
```

- In the above code we can see how long it is to declare an iterator. It also shows different way of going through the elements of the vector.
- In the first example they have used indices to access the elements. In the second case they have declared an iterator and ran it through the size using for-loop (the code becomes lengthy).
- To make this shorter we use the code in left image which is called range loop, **for(datatype variable_name1: variable_name) {code}**. This process makes a copy in each step and doesn't change the value of the vector itself.

```cpp
vector<int> v={1,2,3,4,5,6};

for(int i;i<v.size();i++)
cout<<v[i]<<" ";
cout<<endl;

vector<int>::iterator it;
for(it=v.begin();it!=v.end();it++)
cout<<*it<<" ";
cout<<endl;

for(int value: v)
cout<<value<<" ";
cout<<endl;
```

MAPS AND SETS

19 January 2022 12:29

Maps:

- Initialisation: **map<datatype1,datatype2> variable_name;**
  - Example: map<int,string> m; here the first element is the "key" and second "value". Each of the key and value are paired and the datatype is similar to pair. There can't be duplicate keys in the map. If we try to add map element with same key then the "value" value will change.

- Assigning values: **variable_name[key]=value, variable_name.insert({key,value})**.
  - Example: m[1]="hello", m[3]="world" and m.insert({4,"day"}).
- The map key values get automatically sorted internally using the "**red black trees**" algorithm. Thus inserting or accessing the elements takes **O(log(n))** time complexity.
- **find(key)** function finds the element and returns its iterator value and if not found then returns the end() iterator value.
- **erase(key or iterator)** function removes the pair if key or iterator is passed. It
- Maps can be used to find the frequency of the elements as there can't be no duplicate keys.

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

int main(){
    map<string,int> m;
    int n;
    cin >> n;
    for(int i = 0; i < n; ++i){
        string s;
        cin >> s;
        // m[s] = m[s] + 1;
        m[s]++;
    }
    for(auto pr : m){
        cout << pr.first << " " << pr.second << el
    }
}
```

Unordered maps:

Initialisation: **unordered_map<datatpype1,datatype2> variable_name;**

Unordered maps are similar to but doesn't sort the keys. It uses hash table to find the duplicates. And it doesn't support complex datatype as its key. Its complexity is **O(1)**.

So it is preferable to use unordered maps instead of maps when possible.

Sets:

- Initialisation: **set<datatype> variable_name;**
  - Example: set<int> s; set is same as maps except that they don't have value. There can't be duplicate keys in the set as maps. Sets just store the keys in sorted order. We can't use indices to access, instead we use the functions to find or insert the elements.
- Assigning values: **variable_name.insert(value);**

Unordered Sets: same as unordered maps but only keys are present.

Multisets:

Multisets are similar to sets except the fact that they can store duplicate elements multiple times but complex datatypes can't be used. So we can erase them in two different ways-using iterator (which erases only that element) and using the key value (which erases all the elements with that value).

STACK AND QUEUE

12 May 2022 18:58

Stack:

- Initialisation: **stack<datatype1> variable_name;**
  - Example: stack<int> s; here we can only access the top most element which is stored, this datatype works similar to the real life stack (like the books on top of each other). LIFO-last in first out.

- Assigning values: **variable_name.push(value)**.
  - Example: s.push(4).
  - We can get the value of top element using the function **variable_name.top().**
  - Or we can delete the top element using the function **variable_name.pop().**
- Some question like **Matching brackets** and **Next greatest element** problems can be solved very easily using stack.

```
unordered_map<char,int> symbols = {{'[',-1},{
string isBalanced(string s) {
    stack<char> st;
    for(char bracket:s){
        if(symbols[bracket] < 0){
            st.push(bracket);
        }else{
            if(st.empty()) return "NO";
            char top = st.top();
            st.pop();
            if(symbols[top] + symbols[bracket
                return "NO";
            }
        }
    }
    if(st.empty()) return "YES";
    return "NO";
}
```

This is how the it is found whether a string is balanced parenthesis or not.
In this code a small trick is used which makes the code smaller and more efficient, the brackets have been assigned values equal magnitude value with opposite signs with same type. This way we can check if the brackets match or not faster.

Queue:

- Initialisation: **queue<datatype1> variable_name;**
  - Example: queue<int> s; here we can only access the front most element which is stored, this datatype works similar to the real life queue (like the queues in theatres). FIFO-first in first out.

- Assigning values: **variable_name.push(value)**.
  - Example: s.push(4).
  - We can get the value of front element using the function **variable_name.front().**
  - Or we can delete the front element using the function **variable_name.pop().**
- We can also get the value of last element using **variable_name.back()**

```
vector<int> NGE(vector<int> v){
    vector<int> nge(v.size());
    stack<int> st;
    for(int i = 0; i < v.size(); ++i){
        while(!st.empty() && v[i] > v[st.top(
            nge[st.top()] = i;
        }
        st.push(i);
    }
    while(!st.empty()){
        nge[st.top()] = -1;
    }
}
```

The code in the left is the code to find the value of the next greatest element for each element.

GRAPH ALGORITHMS

18 March 2022 15:33

Bellman-Ford Algorithm:

```
int n,m;
vector<vector<pair<int,int>>> g;

int BellmanFord(int src,int dest)
{
    vector<int> dis(n+1,1e9);
    dis[src]=0;
    for(int i=0;i<n;i++)
    {
        for(int k=0;k<n;k++)
        {
            for(auto con:g[k])
            {
                if(dis[con.first]>dis[k]+con.second)
                dis[con.first]=dis[k]+con.second;
            }
        }
    }
    return dis[dest];
}
```

For a given graph with V vertices and E edges, bellman-ford algorithm iterators for V-1 times. All the distances from the source vertex is taken as infinite except for the source verted itself which is taken as zero initially.

Then the algorithm checks if the shortest distance from the starting node plus the weight is less than the shortest distance of the ending node, for each edge present in the graph. So, for each iteration there will be new connection. Suppose there is a node which has shortest distance through "i" edges then the distance will be found in at least in ith iteration. This can be confirmed as that route will be found as each of that node would be updated in the iteration.

As there can't be any route which can be shortest with more than V-1 edges (unless the graph has negative cycle), thus the iteration is performed V-1 times.

You can see the code implemented in c++, there is a for loop which runs V times and another loop which goes through all the edges. The update which is made is "**if(dis[u]> dis[v]+weight(v,u)) then dis[u]=dis[v]+weight(v,u)**"

The time complexity of this algorithm is O(V*E) as the loop runs for V times all over the edges. If the total number of nodes is n then the time complexity can go up to n^3.

The input takes weights which is in the form of **vector<vector<pair<int,int>>> g.**

Breadth First Search (BFS):

```
int n,m;
vector<vector<int>> g,levels;

int BFS(int src,int dest)
{
    vector<int> visited(n,1),length(n,n);
    int level=0;
    levels[level].push_back(src);
    visited[src]=0;
    length[src]=0;
    while(levels[level].size())
    {
        for(auto child:levels[level])
        for(auto child2:g[child])
        if(visited[child2])
        {
            levels[level+1].push_back(child2);
            visited[child2]=0;
            length[child2]=level+1;
        }
        level++;
    }
    return length[dest];
}
```

This algorithm goes to every vertex in a level-wise manner, here each level means minimum edges each node is separated from source vertex. Source vertex is considered as the starting point and next reachable vertices as level one and so on.

If the graph has equal weights then it can be used to find the shortest path from a given node to any other node. As the weights are equal we can consider the nodes to belong to each level and hence that becomes the path length of that node.

This algorithm is called Breadth First Search because it first goes to each level and then goes to its next level and so on. Here each level's elements are stored in vector called levels along with there level values in another vector.

It doesn't go to the nodes which has already been visited. As it has already got its shortest distance.

The time complexity of this algorithm is O(V+E) as each node is stored in the levels vector and its connection are taken into consideration for finding the shortest path.

0-1 BFS:

```
int zoBFS(int src,int dest)
{
    vector<int> visited(n,1),length(n,n);
    int level=0;
    levels[level].push_back(src);
    visited[src]=0;
    length[src]=0;
    while(levels[level].size())
    {
        for(int i=0;i<levels[level].size();i++)
        {
            int child=levels[level][i];
            for(auto child2:g[child])
            if((visited[child2.first])&&(child2.
            {
                visited[child2.first]=0;
                levels[level].push_back(child2.f
                length[child2.first]=level;
            }
        }
        for(int i=0;i<levels[level].size();i++)
        {
            int child=levels[level][i];
            for(auto child2:g[child])
            if((visited[child2.first])&&(child2.
            {
                visited[child2.first]=0;
                levels[level+1].push_back(child2
                length[child2.first]=level+1;
            }
        }
        level++;
    }
    return length[dest];
}
```

This algorithm is similar to the above algorithm, BFS, but the difference is if the weight is zero then the node is added to the same level rather than the next level. And the rest of the nodes which can be visited from these nodes are added to the next level.

The input doesn't take weights hence it is in the form of **vector<vector<int>> g.**

Dijkstra Algorithm:

```
int n,m;
vector<vector<pair<int,int>>> g;

int dijkstra(int src,int dest)
{
    vector<int> arrival(n+1,1e9);
    vector<int> departure(n+1,1e9);
    vector<int> visited(n+1,0);
    arrival[src]=0;
    set<pair<int,int>> s;
    s.insert({0,src});
    while(!s.empty())
    {
        auto x=*(s.begin());
        s.erase(x);
        visited[x.second]=1;
        departure[x.second]=arrival[x.second];
        for(auto it:g[x.second])
        {
            if(arrival[it.first]>departure[x.second]+it.second)
            {s.erase({arrival[it.first],it.first});
            arrival[it.first]=departure[x.second]+it.second;
            s.insert({arrival[it.first],it.first});}
        }
    }
    return arrival[dest];
}
```

If the graph is having only positive weights, then we can use Dijkstra algorithm to find the shortest path for each of the nodes from source node.

Suppose we are generating paths from the source node and we have n nodes at the current time then we can say the node with the minimum shortest distance has the actual shortest distance because if we take a different path to reach the node it will take more distance because all the weights are positive.

Dijkstra algorithm uses this property to find the shortest path by arranging the nodes in sorted order and removing the node with the minimum shortest path at that time.

The algorithm simultaneously finds the distance of the connected nodes from the minimum shortest distance node.

The time complexity of this algorithm is O((V+E)logV) as this goes through each node and edges simultaneously sorting the shortest distances of the each vertex.

The input takes weights which is in the form of **vector<vector<pair<int,int>>> g.**

This algorithm can be considered as the optimized way of doing Bellman-Ford algorithm.

Some random points:

1. **Trees** are graph which doesn't have cycles in it and every node can be reached from one node to another, because of this reason the number of edges in a tree is always equal to number of vertices minus one.
2. The set of nodes which can be reached from one another is called connected component. As mentioned above trees are always **connected component**.
3. **How to find a cycle in a graph**: Consider all the nodes in a different set and now start considering an edge at a time and if the nodes belong to two different set then take union else the graph contains a cycle. This can be imagined as if we are building the graph and initially all the nodes are different and edges connect the nodes, so if there comes a situation where the nodes are already in the same set then it is obvious that the graph contains cycle. We can perform the same operation using array by indicating the parent node of that node in the array. Then we can check if the last parent are same or not and tell if they form a cycle or not.
4. **Divide and conquer**: When the problem is big then it can be divided into smaller parts and then we can find the solutions to that smaller parts and combine them all to get the final answer. This is mostly done using recursion where it uses induction.
   Pseudo code for divide and conquer:

```
DAC(P)
{
  if(small(P))
    S(P);
  else
  {
    divide into p1,p2,..pk;
    apply DAC(p1),DAC(p2)...DAC(pk);
    combine(DAC(p1),DAC(p2)...DAC(pk));
  }
}
```

```
int factorial(int n)
{
    if(n==1)
    return 1;
    else
    return n*factorial(n-1);
}
```

```
int gcd(int a, int b)
{//b should be greater than or equal to a
    if(a==0)
    return b;
    else
    return gcd(b%a,a);
}
```

5. If the recursion relation is of following form, T(n)=T(n-1)+f(n), then the time complexity will of order O(n*f(n)). If the recursion is of the form T(n)=k*T(n-1)+1, then the time complexity will be O(k^n).
6. So in general we can write the formula as following: if the relation is T(n)=a*T(n-b)+f(n) then time complexity={if a=0 then O(f(n)),if a=1 then O(n*f(n)),if a>1 then O((a^(n/b))*f(n))}. (This result is called as **Master's theorem**.)
7. If the recursion relation is of the form T(n)=T(n/2)+1, then we will get the time complexity as O(log(n)) and we can generalise these type of division relation as same as above.

Binary Search algorithm:

```
int binarySearch(int *a,int l,int h,int num)
{
    int mid=(l+h)/2;
    if(l==h)
    if(a[mid]!=num)
    return -1;

    if(a[mid]==num)
    return mid;
    else if(a[mid]<num)
```

As we had discussed divide and conquer method, and binary search also uses the same concept to find the number in O(log(n)) time complexity. The algorithm divides the array in two groups every time and checks if the mid value is the

```
int binarySearch(int *a,int l,int h,int num)
{
    while(1)
    {
        int mid=(l+h)/2;
        if(l==h)
        if(a[mid]!=num)
        return -1;
        if(a[mid]==num)
        return mid;
        else if(a[mid]<num)
```

```
    if(a[mid]==num)
    return mid;
    else if(a[mid]<num)
    return binarySearch(a,mid+1,h,num);
    else
    return binarySearch(a,l,mid,num);
}
```

the array in two groups every time and checks if the mid value is the number if not then applies the same process to the sub array which may contain the number.

```
        if(a[mid]==num)
        return mid;
        else if(a[mid]<num)
        l=mid+1;
        else
        h=mid;
    }
}
```

Heaps:

Array representation of binary trees: If the array is numbered from one to n (where n is number of nodes) then **if a node has index "i" then its first child will be at index 2*i and second child at 2*i+1. And it will have the parent at node [i/2]**. If the nodes are represented using the tree diagram and if we fill the array from left to right then the array will automatically follow the above rule. If the child is not present then denote that.

Complete binary tree: When represented in array form, if there are gaps in between or if find a missing child when going from left to right before going through all the nodes then the tree is not complete.

When the tree has maximum number of nodes possible for a given height then the tree is called full tree. And the number of nodes had the form (2^h)-1.

Complete tree

Non-complete tree

Full tree

Heap: It is complete binary tree. If the parent node has its value more than the children node then it is called max heap. And if the parent node has its value less than the children node then the heap is called min heap.

Insert and delete:

During insertion the element is added in the end and then compared with its parent to check if it is lesser than it, if yes then it is swapped and this process happens till the condition is broken. So the time complexity is O(log(n)).

During deletion only the root node can be deleted. Thus the root node is removed and is replaced by the last element and it is then adjusted accordingly. First it checks with its children nodes and decides which will become the parent node and this process goes till it reaches the last level.

Heap sort: Heap sort is nothing but deleting the elements from the array for n-1 times. As the most highest element is swapped with the last element which will be removed later hence the array is sorted in increasing order.

Heapify: In heapify process we check from bottom if each element forms a heap beneath, if yes nothing is done else it is adjusted to form a heap as same as in delete operation.

Priority queue: Heaps are nothing but priority queue because the element accessible has the most priority.

```
void heapdelete(vector<int> &a)
{
    int pos=0,s=a.size();
    swap(a[0],a[s-1]);
    a.pop_back();
    s--;
    while(1)
    {
        int ch,chi;
        if(2*(pos+1)-1<s)
        {
            chi=2*(pos+1)-1;
            ch=a[chi];
            if(2*(pos+1)<s)
            if(ch<a[chi+1])
            {chi++;ch=a[chi];}

            if(a[pos]<ch)
            {swap(a[pos],a[chi]);pos=chi;}
            else
            break;
        }
        else
        break;
    }
}
```

```
void heapsort(vector<int> &a)
{
    int s=a.size();

    while(s!=1)
    {int pos=0;
    swap(a[0],a[s-1]);
    s--;
    while(1)
    {
        int ch,chi;
        if(2*(pos+1)-1<s)
        {
            chi=2*(pos+1)-1;
            ch=a[chi];
            if(2*(pos+1)<s)
            if(ch<a[chi+1])
            {chi++;ch=a[chi];}

            if(a[pos]<ch)
            {swap(a[pos],a[chi]);pos=chi;}
            else
            break;
        }
        else
        break;
    }}
}
```

```
void heapinsert(vector<int> &a,int n)
{
    a.push_back(n);
    int pos=a.size()-1;
    while(pos!=0)
    {
        if(a[pos]>a[((pos+1)/2)-1])
        {swap(a[pos],a[((pos+1)/2)-1]);pos=((pos+1)/2)-1;}
        else
        break;
    }
}
```

LEVELS:

21 June 2022 17:08

- **LEVEL1:** Basics maths and logic, maximum, minimum, summation etc.
- **LEVEL2:** Sorting, bitwise operations, permutation and inversion, two pointers, binary search.
- **LEVEL3:** Dynamic Programming, Trees and graphs.

O(LOGN)

- **BINARY SEARCH:** If the array is sorted then an element can be found in **O(logN)** complexity.
  - Sort the array if it doesn't change the property of the array.
  - Create something that can be sorted out of that array.
- **BINARY SEARCH ON ANSWER:** If answer to a problem is an range of values which satisfy above a value and doesn't satisfy below that value then we use binary search over that range and find the optimised answer.
  - If it takes O(X) time to check then the answer can be found in O(XlogN) complexity.
- **STORE IN SETS AND MAPS:** Deletion and Insertion can be done in O(logN) complexity.
  - Faster accessing of elements.
  - Use of upper and lower bound for solving problems.
  - Very useful to find indices and subarrays (**like distinct elements subarray, equal 0's and 1's subarray**).

```
int bins(vectori val,int l,int h)
{
    while(l<h)
    {
        int mid=l+(h-l)/2;
        if(fun(mid))
        h=mid;
        else
        l=mid+1;
    }

    return l;
}
```

```
map<int,int> freq;
trav(arr)
{
    cin>>child;
    freq[child]++;
}
```

Maps can be used to find the Frequency of elements faster.

Some functions like **gcd**, **fastpow** and **modinv** uses logN time complexity.

PRECOMPUTATION

Precomputation helps us reduce the time complexity by storing the values and accessing them in O(1). Like **Prefix sum** can be calculated in O(n) time complexity and then can be accessed in O(1). Prefix sum can be calculated to any dimensional matrix.
If we try to calculate all the primes individually then it will take long time but if we precompute it using **Sieve of Eratosthenes** it can be done much faster, same goes to prime factorisation.

DYNAMIC PROGRAMMING

Some recursive problems can have exponential time complexity which sometimes can be reduced to polynomial by avoiding the redundant computation just by storing the previously calculated values. For example: **Knapsack problems, coin change** etc.

- **Tabulation (bottom-up):** Calculating the values from top to bottom. The upper n may require the lower n values which can be used in O(1) thus avoiding time wastage. This way all the values are calculated in the process. This is same as precomputation.
- **Memoization (top-bottom):** This method computes the values from top to bottom and then storing the values calculated and uses them if needed again.

First write the recursive solution in recursive function form and then store the values calculated then use them if needed again by checking if they were calculated earlier.

BITWISE OPERATIONS

Using the **| (OR)** we can set a particular bit of a number.
X=(1<<n) will shift one to the nth position, then we can set the number using **num|=X;**

Using the **& (AND)** we can unset a particular bit of a number.
X=~(1<<n) will shift zero to the nth position, then we can unset the number using **num&=X;**

A+B=(A&B)+(A|B)
A+B=(A^B)+2*(A&B)

Using the **^ (XOR)** we can invert a particular bit of a number.
X=(1<<n) will shift one to the nth position, then we can invert the number using **num^=X;**

- To check if a bit is set or unset we can use, **bit=num&(1<<n);**
- **num&=(num-1);** will remove the last set bit.
- **num&=(-num**); will give the last bit which is set.