Arrays

Array elements are always stored in a consecutive manner

The 2D array will only print the value of defined, for other it will print the garbage value

String

Stores every character in terms of index

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

int main() {
    string s = "Striver";
    int len = s.size();
    s[len-1] = 'z';
    cout << s[len - 1];
    return 0;
}
```

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

int main() {
    string s = "Striver";
    int len = s.size();
    cout << s[len - 1];
    return 0;
}
```

Functions:

```cpp
void printName(string name) {
    cout << "hey " << name << endl;
}
int main() {
    string name;
    cin >> name;
    printName(name);

    string name2;
    cin >> name2;
    printName(name2);
    return 0;
}
```

output.txt

```
hey AMan
hey Raj
```

Basics in One Shot - Strivers A2Z DSA Course - L1

```cpp
#include<bits/stdc++.h>
using namespace std;
// Functions are set of code which performs something for you
// Functions are used to modularise code
// Functions are used to increase readability
// Functions are used to use same code multiple times
// void -> which does not returns anything
// return
// parameterised
// non parameterised

void printName(string name) {
    cout << "hey " << name;
}
int main() {
    string name;
    cin >> name;
    printName(name);
    return 0;
}
```

input.txt

```
1   AMan
```

output.txt

```
1   hey AMan
```

```cpp
// Take two numbers and print its sum
int sum(int num1, int num2) {
    int num3 = num1 + num2; // 5 + 6 = 11
    return num3;
}

int main() {
    int num1, num2;
    cin >> num1 >> num2;
    int res = sum(num1, num2);
    cout << res;
    return 0;
}
```

output.txt

```
1   11
```

Pass by value:

Edits on duplicate value

```cpp
// pass by value
void doSomething(int num) {
    cout << num << endl;
    num += 5;
    cout << num << endl;
    num += 5;
    cout << num << endl;
}
int main() {
    int num = 10;
    doSomething(num);
    cout << num << endl;
    return 0;
}
```

output.txt

```
1   10
2   15
3   20
4   10
5
```

Pass by Reference:

Edits original value

```cpp
// pass by reference
void doSomething(string &s) {
    s[0] = 't';
    cout << s << endl;
}
int main() {
    string s = "raj";
    doSomething(s);
    cout << s << endl;
    return 0;
}
```

output.txt

```
taj
taj
```

```cpp
// pass by reference
void doSomething(int &num) {
    cout << num << endl;
    num += 5;
    cout << num << endl;
    num += 5;
    cout << num << endl;
}
int main() {
    int num = 10;
    doSomething(num);
    cout << num << endl;
    return 0;
}
```

output.txt

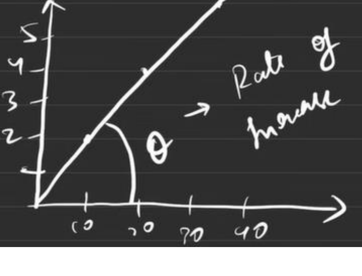
```
10
15
20
20
```

Time Complexity

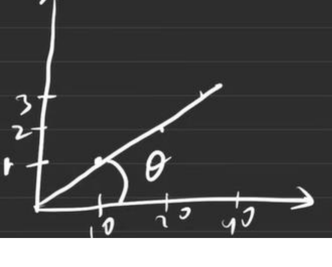
What is Time Complexity? ⇒ TC != time taken

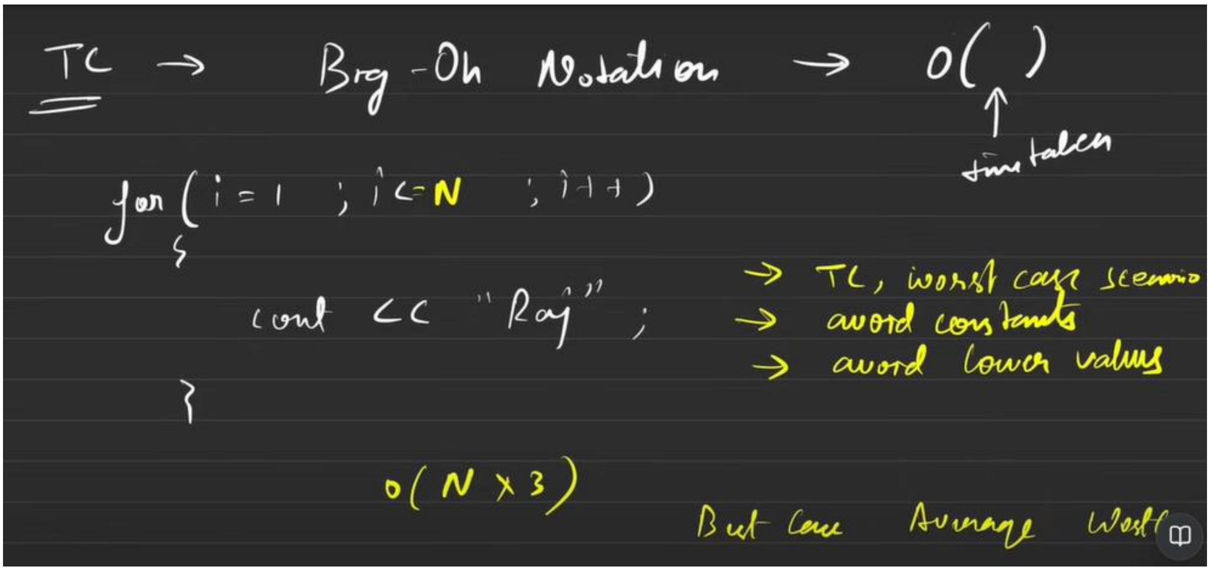
↳ Rate at which the time taken increases with respect to the input size.

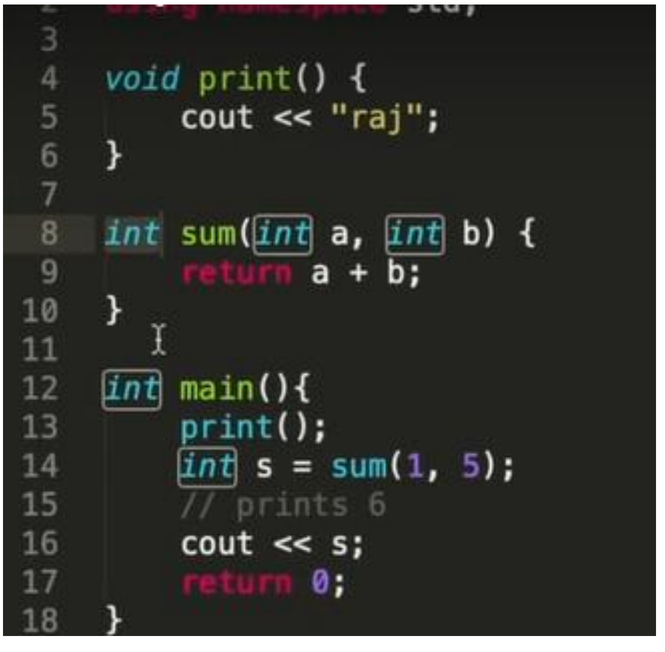
Old Windows



New Macbook



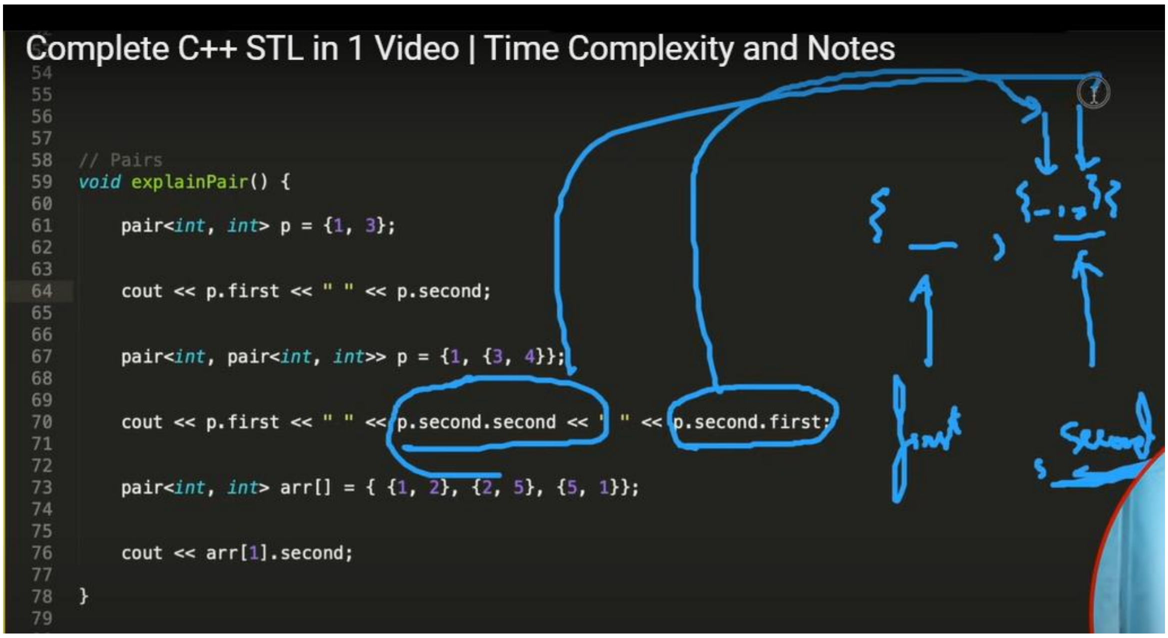




```cpp

void print() {
    cout << "raj";
}

int sum(int a, int b) {
    return a + b;
}

int main(){
    print();
    int s = sum(1, 5);
    // prints 6
    cout << s;
    return 0;
}
```

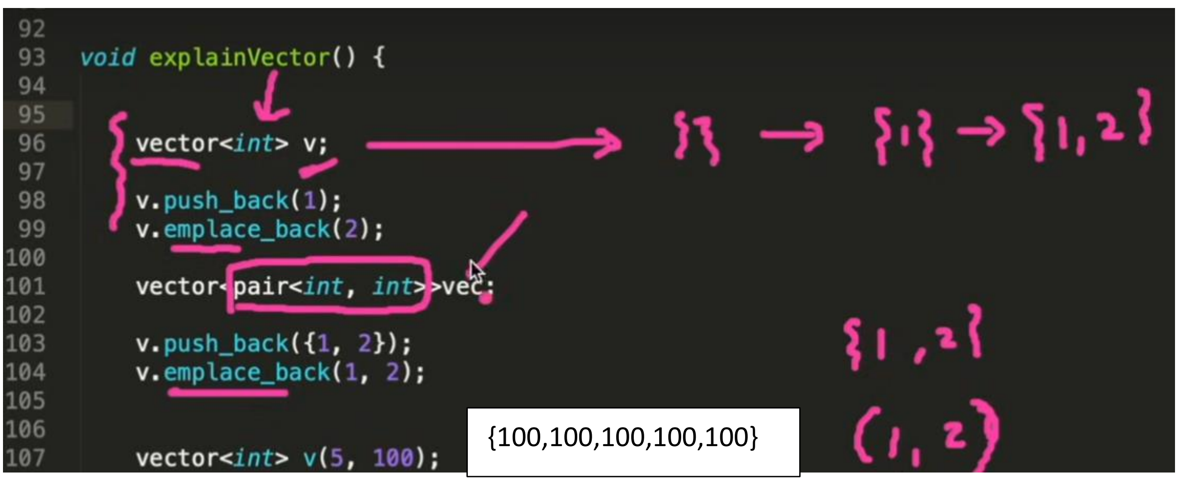
Pairs

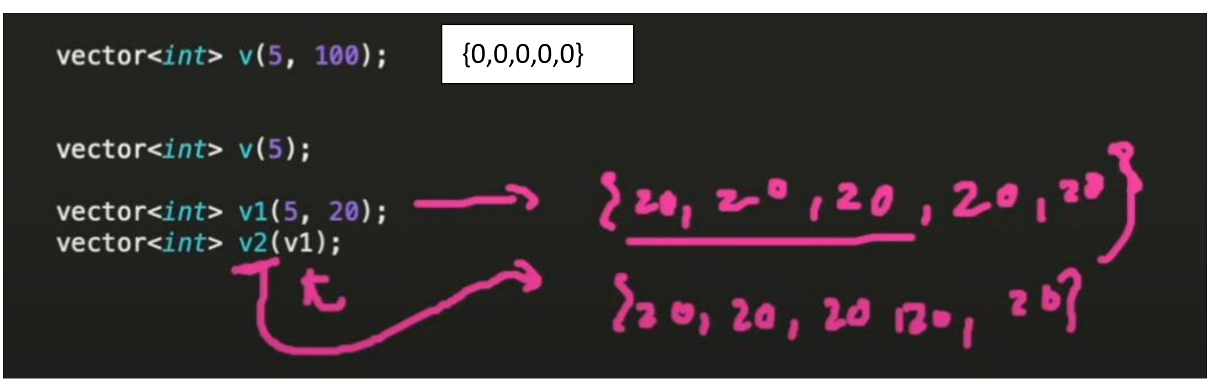
Lies inside the utility library



Vectors:

- It is a container that is dynamic in nature.
- You can always increase the size of the vector





S. No.	Function	Description	Time Complexity
1.	begin()	Returns an iterator to the first element.	O(1)
2.	end()	Returns an iterator to the theoretical element after the last element.	O(1)
3.	size()	Returns the number of elements present.	O(1)
4.	empty()	Returns true if the vector is empty, false otherwise.	O(1)
5.	at()	Return the element at a particular position.	O(1)
6.	assign()	Assign a new value to the vector elements.	O(n)
7.	push_back()	Adds an element to the back of the vector.	O(1)
8.	pop_back()	Removes an element from the end.	O(1)
9.	insert()	Insert an element at the specified position.	O(n)
10.	erase()	Delete the elements at a specified position or range.	O(n)
11.	clear()	Removes all elements.	O(n)

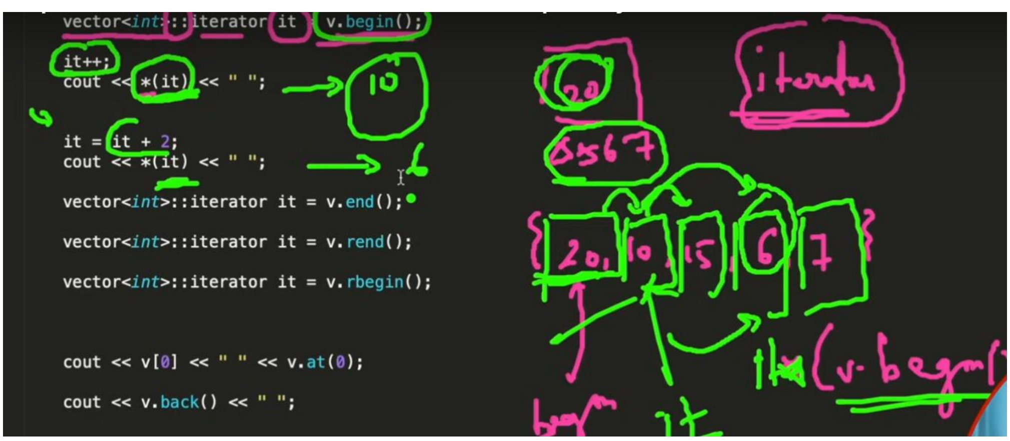
Iterator

- iterators are used to point at the memory addresses of STL containers. They are primarily used in sequences of numbers, characters, etc. They reduce the complexity and execution time of the program.
- Iterators can be used to traverse the elements of a vector.

Operations of iterators :-

1. begin() :- This function is used to return the **beginning position** of the container.

2. end() :- This function is used to return the ***after-end*** **position** of the container.



Insert:

```cpp
// Insert function

vector<int>v(2, 100); // {100, 100}
v.insert(v.begin(), 300); // {300, 100, 100};
v.insert(v.begin() + 1, 2, 10); // {300, 10, 10, 100, 100}
```

```cpp
// {10, 20}
cout << v.size(); // 2

//{10, 20}
v.pop_back(); // {10}

// v1 -> {10, 20}
// v2 -> {30, 40}
v1.swap(v2); // v1 -> {30, 40} , v2 -> {10, 20}

v.clear(); // erases the entire vector

cout << v.empty();
```

List

Compared to the vector, the list has slow traversal, but once a position has been found, insertion and deletion are quick (constant time). Normally, when we say a List, we discuss a doubly linked list. For implementing a singly linked list, we use a forward_list.

```cpp
void explainList() {
    list<int> ls;

    ls.push_back(2); // {2}
    ls.emplace_back(4); // {2, 4}

    ls.push_front(5); // {5, 2, 4};

    ls.emplace_front(); {2, 4};

    // rest functions same as vector
    // begin, end, rbegin, rend, clear, insert, size, swap
```

Dequeue

The deque implements the double-ended queue which follows the FIFO mode of operation but unlike the queue, the deque can grow and shrink from both ends. It is defined as **std::deque** inside the **<deque>** header file.

```
void explainDeque() {

    deque<int>dq;
    dq.push_back(1); // {1}
    dq.emplace_back(2); // {1, 2}
    dq.push_front(4); // {4, 1, 2}
    dq.emplace_front(3); // {3, 4, 1, 2}

    dq.pop_back(); // {3, 4, 1}
    dq.pop_front(); // {4, 1}

    dq.back();

    dq.front();
```

Stack : LIFO

S. No.	Function	Description	Time Complexity
1.	empty()	Returns true if the stack is empty, false otherwise.	O(1)
2.	size()	Returns the number of elements in the stack.	O(1)
3.	top()	Returns the top element.	O(1)
4.	push(g)	Push one element in the stack.	O(1)
5.	pop()	Removes one element from the stack.	O(1)

```cpp
void explainStack() {
    stack<int> st;
    st.push(1); // {1}
    st.push(2); // {2, 1}
    st.push(3); // {3, 2, 1}
    st.push(3); // {3, 3, 2, 1}
    st.emplace(5); // {5, 3, 3, 2, 1}

    cout << st.top(); // prints 5   "** st[2

    st.pop(); // st looks like {3, 3, 2, 1}

    cout << st.top(); // 3

    cout << st.size(); // 4

    cout << st.empty();        False

    stack<int>st1, st2;                4
    st1.swap(st2);
```

Queue:

```cpp
void explainQueue() {
    queue<int> q;
    q.push(1); // {1}
    q.push(2); // {1, 2}
    q.emplace(4); // {1, 2, 4}

    q.back() += 5

    cout << q.back(); // prints 9

    // Q is {1, 2, 9}
    cout << q.front(); // prints 1

    q.pop(); // {2, 9}

    cout << q.front(); // prints 2

    // size swap empty same as stack
```

Set : used to store unique values

The set is an associative container that stores unique values in sorted order, either ascending or descending. It generally implements a red-black tree as an underlying data structure

```cpp
void explainSet() {
    set<int>st;
    st.insert(1); // {1}
    st.emplace(2); // {1, 2}
    st.insert(2); // {1, 2}
    st.insert(4); // {1, 2, 4}
    st.insert(3); // {1, 2, 3, 4}
```

```cpp
    // {1, 2, 3, 4, 5}
    auto it = st.find(3);

    // {1, 2, 3, 4, 5}
    auto it = st.find(6);

    // {1, 4, 5}
    st.erase(5); // erases 5 // takes logarithmic ti


    int cnt = st.count(1);


    auto it = st.find(3);
    st.erase(it); // it takes constant time

    // {1, 2, 3, 4, 5}
    auto it1 = st.find(2);
    auto it2 = st.find(4);
    st.erase(it1, it2); // after erase {1, 4, 5} [fi

    // lower_bound() and upper_bound() function work
    // as in vector it does.

    // This is the syntax
    auto it = st.lower_bound(2);

    auto it = st.upper_bound(3);
```

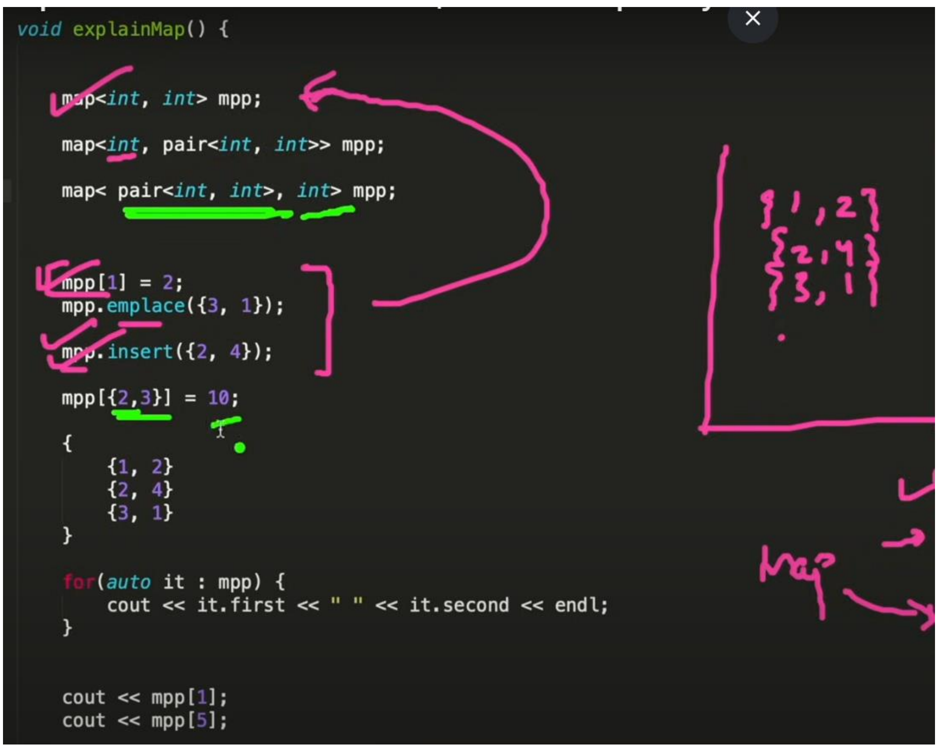
```cpp
void explainMultiSet() {
    // Everything is same as set
    // only stores duplicate elements also


    multiset<int>ms;
    ms.insert(1); // {1}
    ms.insert(1); // {1, 1}
    ms.insert(1); // {1, 1, 1}

    ms.erase(1); // all 1's erased

    int cnt = ms.count(1);

    // only a single one erased
    ms.erase(ms.find(1));


    ms.erase(ms.find(1), ms.find(1)+2);

    // rest all function same as set
}
```

Unordered List

The unordered set is the version of the set container where the data is not sorted but we can still perform a quick search. This is because these unordered sets are implemented using hash tables.

Map:

[Maps](#) are associative containers used to store the key-value pairs where each key should be unique. It generally implements a red-black tree to store data in sorted order.



multimaps:

1. **Key-Value Pairing**: Each key can be associated with multiple values.
2. **Duplicates**: Keys can have duplicate values.

3. **Operations**: Common operations include insertion, deletion, and searching of key-value pairs.
4. **Use Cases**: They are useful in situations like storing multiple phone numbers for a single contact, organizing students' grades where a single student can have multiple grades, or managing tasks that can belong to multiple categories.

Recursion

It is a phenomenon when a function calls itself indefinitely until a specified condition is fulfilled.

Stack Overflow:

when there is no base condition given for a particular recursive function, it gets called indefinitely which results in a Stack Overflow

Example:

```
#include<bits/stdc++.h>

using namespace std;

int cnt = 0;

void print(){

  // Base Condition.

  if(cnt == 3)  return;

  cout<<cnt<<endl;

  // Count Incremented

  cnt++;

  print();}

int main(){  print();
return 0; }
```

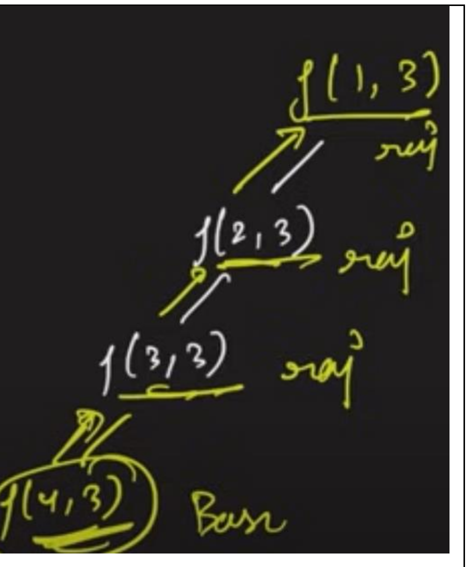
Output

```
0
1
2
```

Print name N Times

```
void func( i, n )
{
  if(i>n) return;
 print("Vraj");
  f( i+1,N );
}
main()
{
 int n;
 input(n);
 f(1,n);
}
TC: O(n)
```

```
Input:3
Output:
Vraj
Vraj
Vraj
```



Print form N to 1

```
void func(int i, int n){

  // Base Condition.
  if(i<1) return;
  cout<<i<<endl;

  // Function call to print i till i decrements to 1.
  func(i-1,n);

}
int main(){

 // Here, let's take the value of n to be 4.
 int n = 4;
 func(n,n);
 return 0;
}
```

Backword recursion

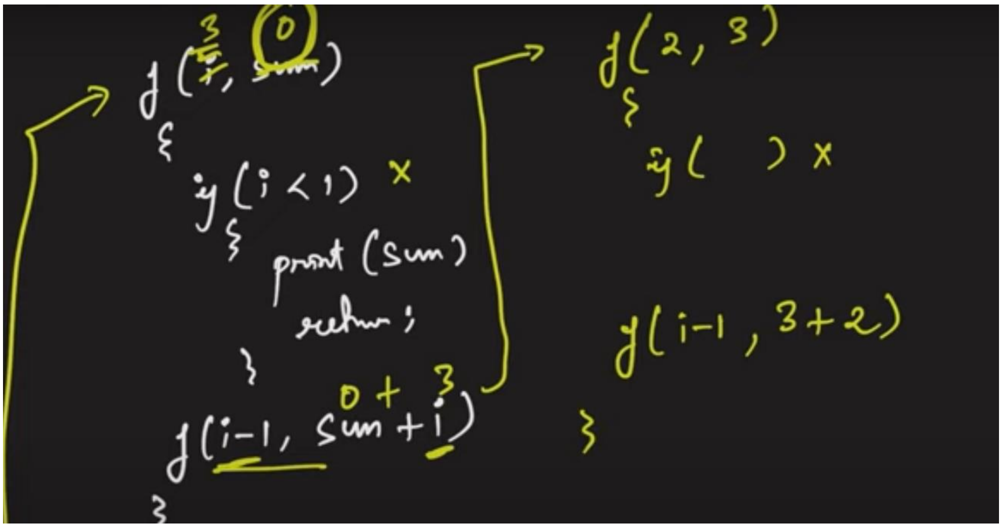
```
void func( i, n ){
  if(i>n) return;
 f( i+1,N );
print(i); }
main(){
 int n;
 input(n);
 f(1,n); }
```

Output

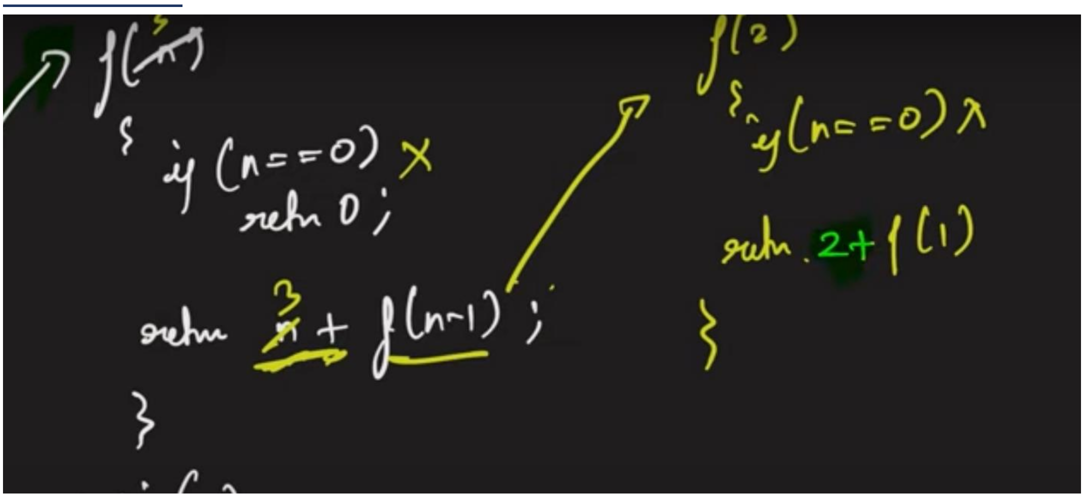
```
4
3
2
1
```

Sum of first N Natural Numbers

Parameterized:



Functional



func(3)

3 + func(2)

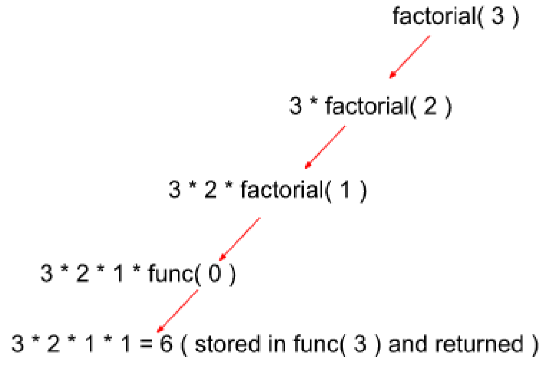
3 + 2 + func(1)

3 + 2 + 1 + func(0)

3 + 2 + 1 + 0 = 6 (stored in func(3)

Factorial of N

```
int factorial(n)
{
  if(n == 0)
  {
    return 1;
  }
return n * factorial(n-1);
}
main()
{
  input(n);
  factorial(n);

}
TC: O(n) ; SC: O(n)
```



Remaining:

https://www.youtube.com/watch?v=twuC1F6gLI8&list=PLgUwDviBIf0rGlzIn_7rsaR2FQ5e6ZOL9&index=5

Palindrome:

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

bool palindrome(int i, string& s){

   // Base Condition
   // If i exceeds half of the string, it means all the elements
   // are compared, we return true.
   if(i>=s.length()/2) return true;

   // If the start is not equal to the end, not the palindrome.
   if(s[i]!=s[s.length()-i-1]) return false;

   // If both characters are the same, increment i and check start+1 and end-
1.
```

```
    return palindrome(i+1,s);}
int main() {

        // Example string.
        string s = "madam";
        cout<<palindrome(0,s);
        cout<<endl;
        return 0;
}
```

Hashing:

Hashing is a fundamental data structure that efficiently stores and retrieves data in a way that allows for quick access. It involves mapping data to a specific index in a hash table using a **hash function** that enables fast retrieval of information based on its key

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

int main() {
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    //precompute:
    int hash[13] = {0};
    for (int i = 0; i < n; i++) {
        hash[arr[i]] += 1;  }
    int q;
    cin >> q;
    while (q--) {
        int number;
        cin >> number;
        // fetching:
        cout << hash[number] << endl;
    }
    return 0;
```

Input:

```
5
1 3 2 1 3
5
1 4 2 3 12
```

Output:

```
2
0
1
2
0
```

```
}
```

Array Declaration	Maximum size(Integer type)	Maximum size(Boolean type)
Inside main function	10^6	10^7
Globally	10^7	10^8

Character Hashing:

- Character hashing is a technique used to map characters or strings to integer values.
- To map the characters we need to use the ASCII values of the respective characters

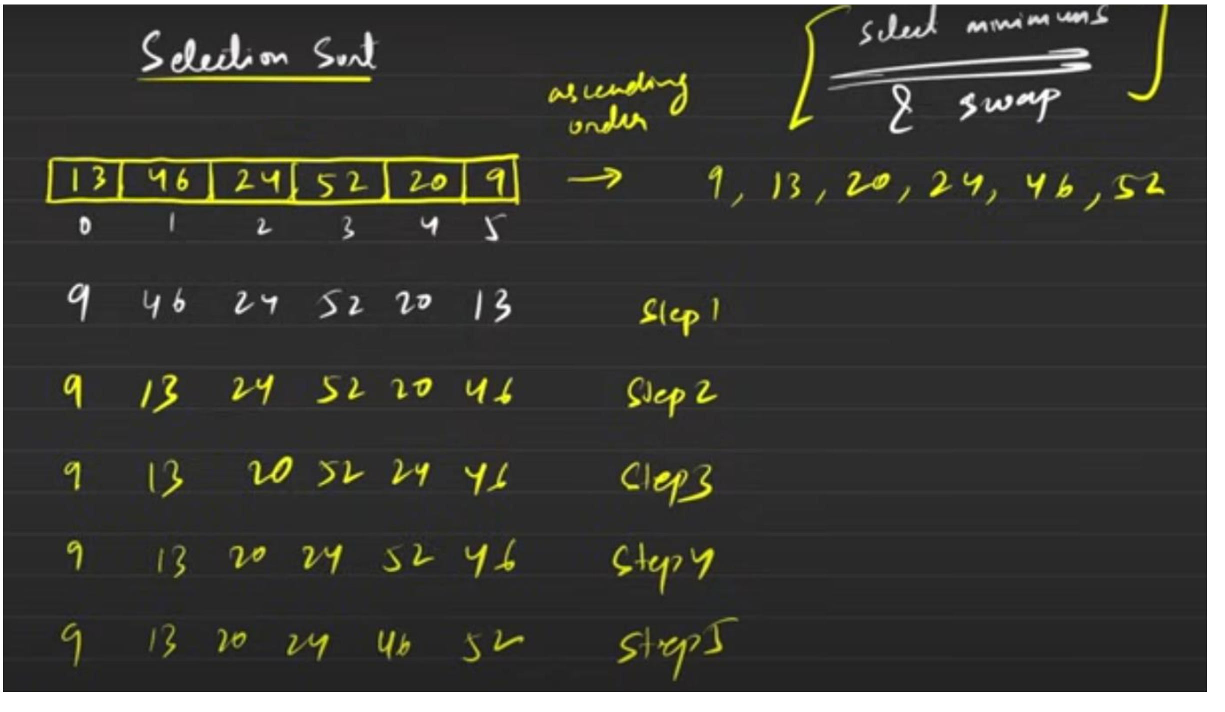
```
#include <bits/stdc++.h>
using namespace std;

int main() {

    string s;
    cin >> s;

    //precompute:
    int hash[256] = {0};
    for (int i = 0; i < s.size(); i++) {
        hash[s[i]]++;
    }

    int q;
    cin >> q;
    while (q--) {
        char c;
        cin >> c;
        // fetch:
        cout << hash[c] << endl;
    }
    return 0;
}
```

Sorting

Selection Sort: Select minimum & swap   TC: o(n^2)

- First, we will select the range of the unsorted array using a loop (say i) that indicates the starting index of the range.
  The loop will run forward from 0 to n-1. The value i = 0 means the range is from 0 to n-1(*Initially, the range will be the whole array starting from the first index.*)

- Now, in each iteration, we will select the minimum element from the range of the unsorted array using an inner loop.
- After that, we will swap the minimum element with the first element of the selected range(*in step 1*).
- Finally, after each iteration, we will find that the array is sorted up to the first index of the range.



Bubble Sort: Push max to the last by adjacent swaps TC: O(n)

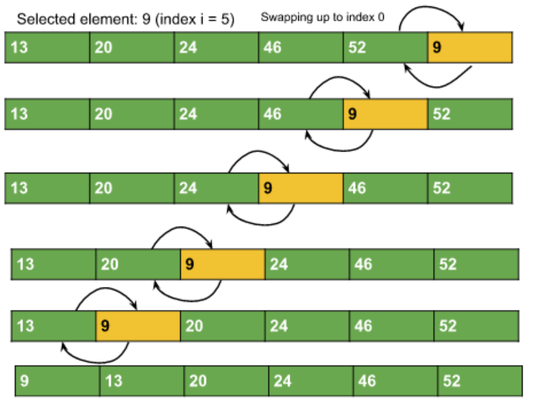
```
void bubble_sort(int arr[], int n){
  //loop iterates n-1 times, starting from the last element of the array.
  for (int i = n - 1; i >= 0; i--) {
     //inner loop compares adjacent elements of the array, starting from the first element to the
current element i
```

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

13	46	24	52	20	9	
13	46	24	52	20	9	46 > 23 ==> swap
13	24	46	52	20	9	
13	24	46	52	20	9	52 > 20 ==> swap
13	24	46	20	52	9	52 > 9 ==> swap
13	24	46	20	9	52	52 gets sorted

Insertion Sort

```
void insertion_sort(int arr[], int n) {
    // Loop over each element in the array starting from the second element
    for (int i = 0; i <= n - 1; i++) {
        // Set j to the current index i
        int j = i;
        // Continue swapping until the element is in the correct position
        while (j > 0 && arr[j - 1] > arr[j]) {
            // Swap arr[j] and arr[j-1]
            int temp = arr[j - 1];
            arr[j - 1] = arr[j];
            arr[j] = temp;
            // Move one position to the left
            j--;
        }
    }
}
```



Quick Sort:

- Select 1 element as a pivot.
- Now, shift smaller elements to the left of the pivot and shift larger elements to the right of the pivot
- Here, pivot is at the correct position, and left and right subarrays are unsorted
- Apply the same to both left and right arrays

```
int partition(vector<int> &arr, int low, int high) {
   int pivot = arr[low];
   int i = low;
   int j = high;

   while (i < j) {
      while (arr[i] <= pivot && i <= high - 1) {
         i++;
      }

      while (arr[j] > pivot && j >= low + 1) {
         j--;
      }
      if (i < j) swap(arr[i], arr[j]);
   }
   swap(arr[low], arr[j]);
   return j;
}

void qs(vector<int> &arr, int low, int high) {
   if (low < high) {
      int pIndex = partition(arr, low, high);
```

```
        qs(arr, low, pIndex - 1);
        qs(arr, pIndex + 1, high);
    }
}

vector<int> quickSort(vector<int> arr) {
    qs(arr, 0, arr.size() - 1);
    return arr;
}
```

Merge Sort: Divide & Merge

```
void merge(vector<int> &arr, int low, int mid, int high) {
    vector<int> temp; // temporary array
    int left = low;      // starting index of left half of arr
    int right = mid + 1;   // starting index of right half of arr

    //storing elements in the temporary array in a sorted manner//
    while (left <= mid && right <= high) {
        if (arr[left] <= arr[right]) {
            temp.push_back(arr[left]);
            left++;
        }
        else {
            temp.push_back(arr[right]);
            right++;
        }
    }
    // if elements on the left half are still left //

    while (left <= mid) {
        temp.push_back(arr[left]);
        left++;
    }
    //  if elements on the right half are still left //
    while (right <= high) {
        temp.push_back(arr[right]);
        right++;
    }
    // transfering all elements from temporary to arr //
    for (int i = low; i <= high; i++) {
```

```
        arr[i] = temp[i - low];
    }}
void mergeSort(vector<int> &arr, int low, int high) {
    if (low >= high) return;
    int mid = (low + high) / 2 ;
    mergeSort(arr, low, mid);  // left half
    mergeSort(arr, mid + 1, high); // right half
    merge(arr, low, mid, high);  // merging sorted halves
}
```

Arrays

Find Largest element

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

Find 2^{nd} Largest & 2^{nd} smallest

Remove Duplicates

```
int removeDuplicates(int arr[], int n) {
  set < int > set;
  for (int i = 0; i < n; i++) {
    set.insert(arr[i]);
  }
  int k = set.size();
  int j = 0;
  for (int x: set) {
    arr[j++] = x;
  }
  return k;
}

// Optimal Approach

int removeDuplicates(int arr[], int n)
{
  int i = 0;
  for (int j = 1; j < n; j++) {
    if (arr[i] != arr[j]) {
```

```
    i++;
    arr[i] = arr[j];
   }
  }
  return i + 1;
}
```

Find the missing number in the array

```
#include <iostream>
#include <vector>

int missingNumber(const std::vector<int>& nums) {
    int n = nums.size();
    int expected_sum = n * (n + 1) / 2;
    int actual_sum = 0;
    for(int num : nums) {
        actual_sum += num;
    }
    return expected_sum - actual_sum;
}

int main() {
    std::vector<int> nums = {3, 0, 1};
    std::cout << "The missing number is: " << missingNumber(nums) << std::endl;
    return 0;
}
```

count Maximum Consecutive One's in the array

```
int findMaxConsecutiveOnes(vector < int > & nums) {
      int cnt = 0;
      int maxi = 0;
      for (int i = 0; i < nums.size(); i++) {
        if (nums[i] == 1) {
          cnt++;
        } else {
          cnt = 0;
        }

        maxi = max(maxi, cnt);
      }
      return maxi;
    }
```

Linear Seach:

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

Binary Search

- Binary search is only applicable in a sorted search space.
- In binary search, we generally divide the search space into two halves and then try to locate which half contains the target. According to that, we shrink the search space size.

Use Case:

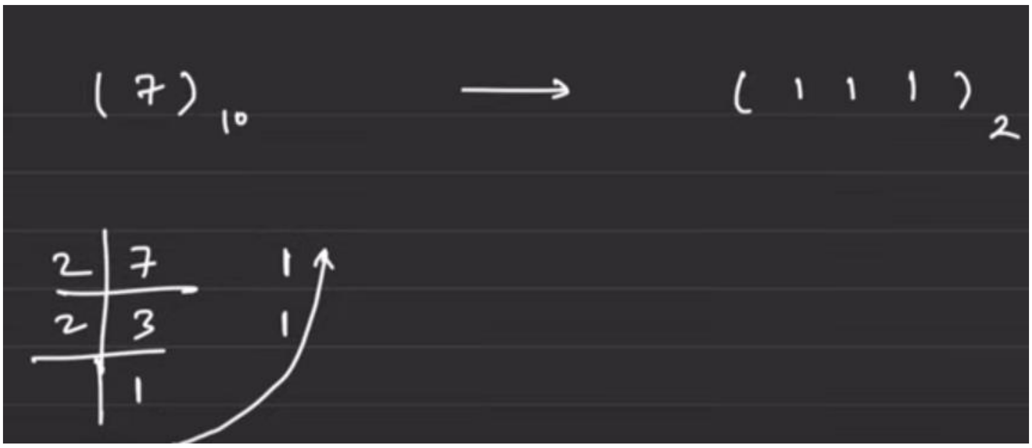
Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one. We used binary search in the guessing game

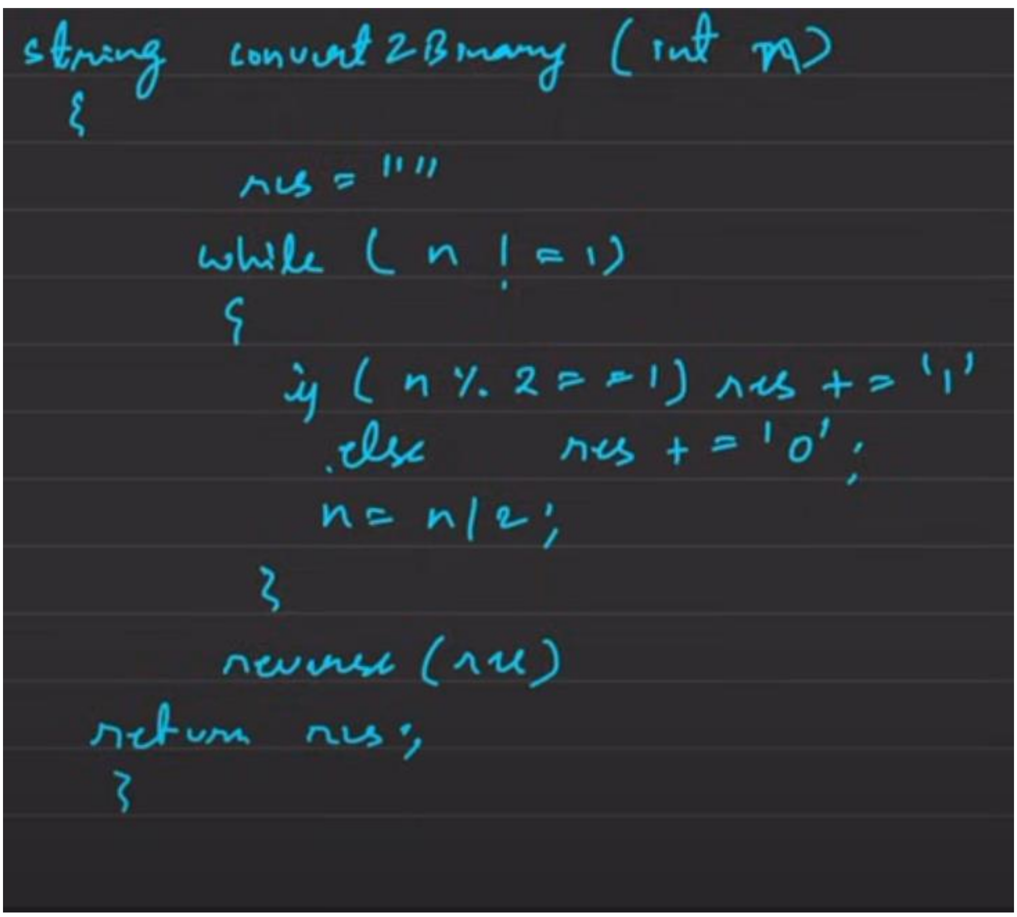
```
int binarySearch(vector<int>& nums, int target) {
   int n = nums.size(); //size of the array
   int low = 0, high = n - 1;

   // Perform the steps:
   while (low <= high) {
      int mid = (low + high) / 2;
      if (nums[mid] == target) return mid;
      else if (target > nums[mid]) low = mid + 1;
      else high = mid - 1;
   }
   return -1;
}
```

The lower bound algorithm finds the first or the smallest index in a sorted array where the value at that index is greater than or equal to a given key i.e. x.

The upper bound algorithm finds the first or the smallest index in a sorted array where the value at that index is greater than the given key i.e. x.

Bit manipulation





```
int convert2Decimal (string x)
{
    int len = x.length    p2 = 1    num = 0

    for( i = len-1 → 0 )
    {
        if ( x[i] == '1' )
            num = num + p2;

        p2 = p2 x 2;
    }

    return num;
}
```

TC → O(len)
SC → O(1)

1s compliment

$(13) \longrightarrow (1101)_2$

↓ flip

$(0010)_2 \xrightarrow{+1}$

2s compliment

1. 1s compliment
2. add 1 to it.

AND → all true → true

1 false → false

OR → 1 true → true

all false → false

XOR → no. of 1s → odd → 1

no. of 1s → even → 0

Right Shift

$n = 13 >> 1$

$= 6$

$\frac{13}{2^1} = 6$

1 1 0 1

1 1 0 → 6

$n = 13 >> 2$

$= 3$

0 0 1 1 0 1

0 0 0 0 1 1 → 3

$n >> k = \left(\frac{n}{2^k}\right)$

2^1

2

$13 >> 1$

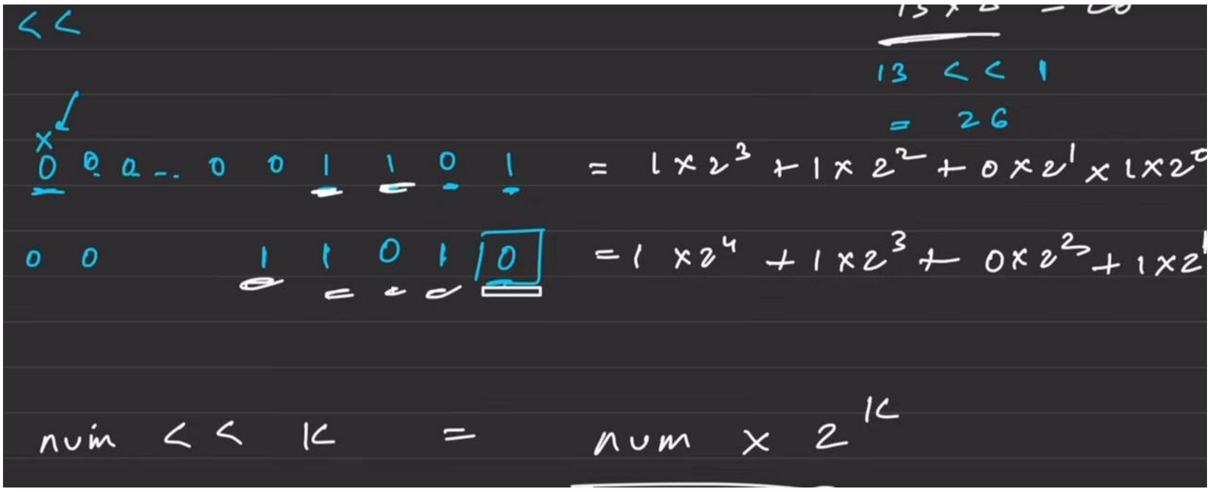
1 1 0 1 → $(1\times2^3 + 1\times2^2 + 0\times2^1) + \cancel{1\times2^0}$

1 1 0 → $1\times2^2 + 1\times2^1 + 0\times2^0$

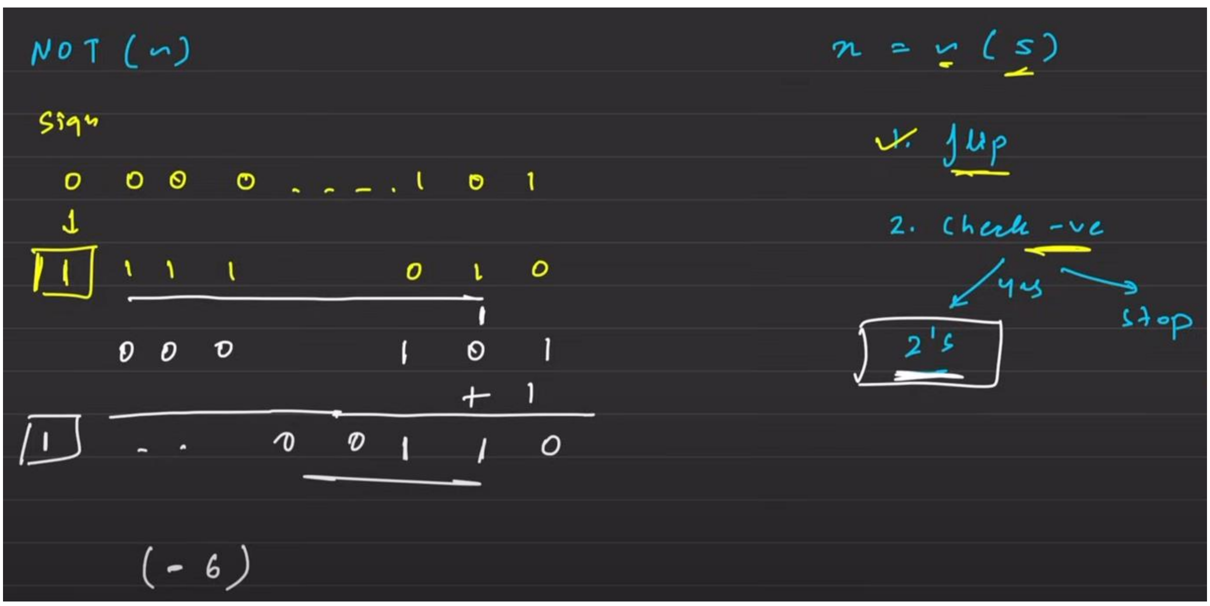
INT_MAX =(2^31 -1)

INT_MIN = -2^31

Shift left



NOT



Swap Two numbers using XOR:

```
a=a ^ b;
b=a ^ b; // (a ^b) ^b =a
a=a ^ b; // (a ^b) ^b= (a ^b) ^a =b
```

Check if the ith bit is set or not

```
//using left shift
If (N &(1<<i)!=0)
```

```
  Set;
Else
 Not set;

//using the right shift
If(N>>I &1==0)
 Not set;
Else
 Set;
```

Clear the ith bit

N & ~(1<<i)

Toggle the ith bit

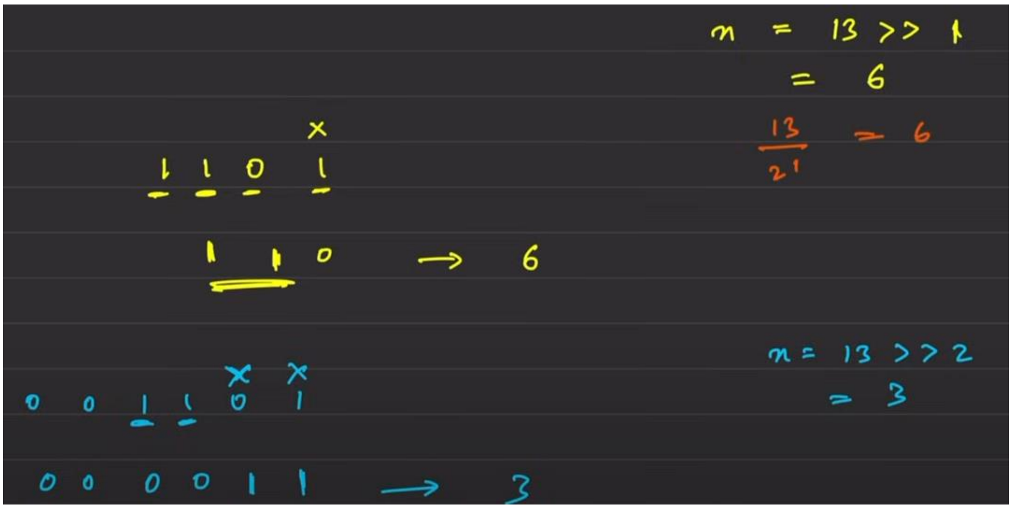
N ^(1<<i)

Coun the no of set bits

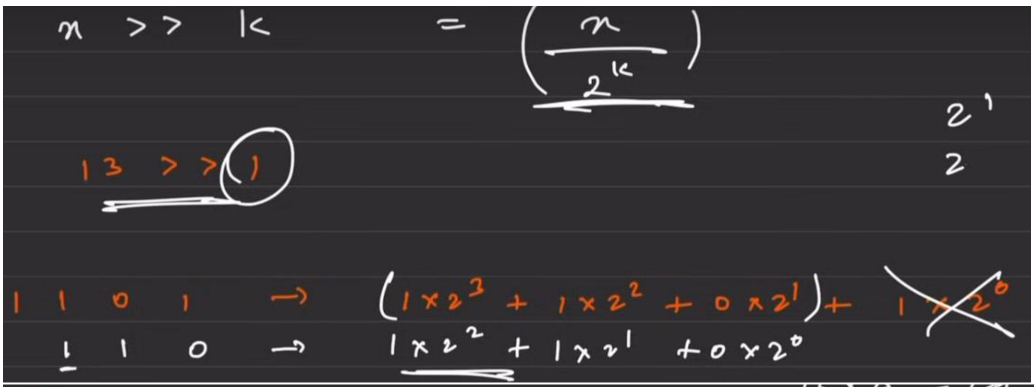
```
int countsetBits(int n){
    int count=0;

    while(n>1){
       if(n%2==1)
           count++;
       n=n/2;
    }
    if(n==1)
       count++;
    return count;

//other
Cnt=0;

While (N! = 0){
  N=N&(N-1)
  Cnt++;
}
```

Miscellaneous

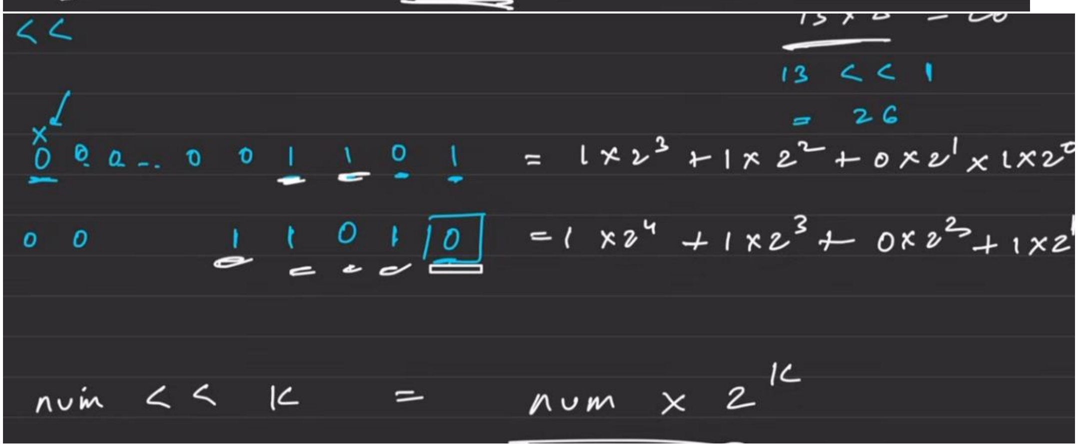
Power of x



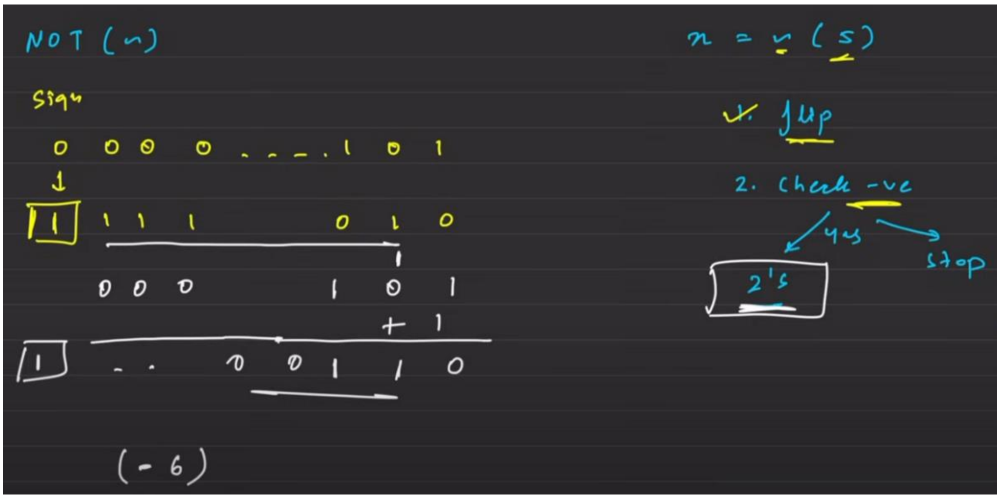
n = 13 >> 1
= 6
13 / 2^1 = 6
1 1 0 1
1 1 0 → 6
n = 13 >> 2
= 3
0 0 1 1 0 1
0 0 0 0 1 1 → 3

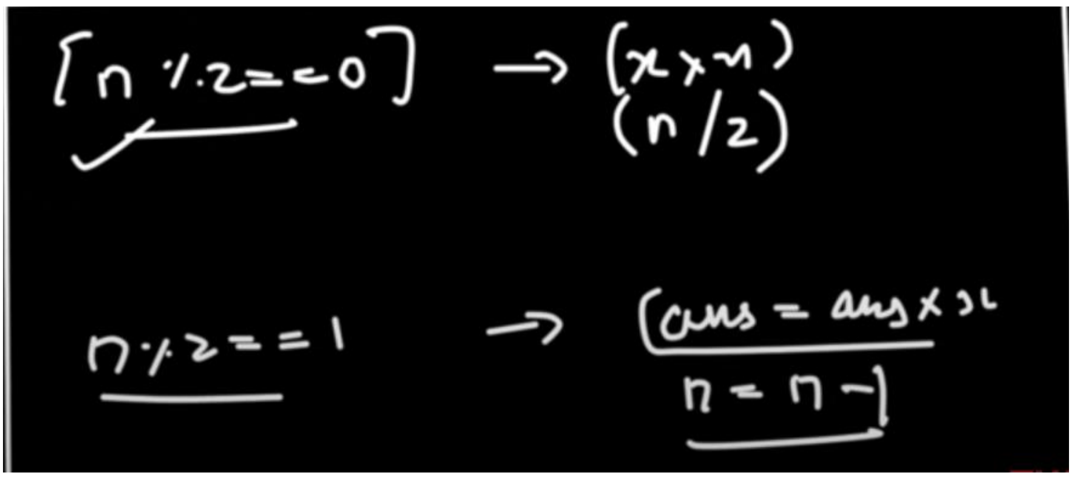


n >> k = (n / 2^k)
13 >> 1
2^1
2
1 1 0 1 → (1×2^3 + 1×2^2 + 0×2^1) + 1×2^0
1 1 0 → 1×2^2 + 1×2^1 + 0×2^0



<<
13 << 1
= 26
0 0 0 .. 0 0 1 1 0 1 = 1×2^3 + 1×2^2 + 0×2^1 × 1×2^0
0 0 1 1 0 1 0 = 1×2^4 + 1×2^3 + 0×2^2 + 1×2^1
num << k = num × 2^k





Graph

graph does not necessarily mean to be an enclosed structure, it can be an open structure as well. A graph is said to have a cycle if it starts from a node and ends at the same node. There can be multiple cycles in a graph.

Types of Graph

1) Directed
2) Undirected

	0	1	2	3	4	5
0						
1			1	1		
2		1			1	
3		1			1	1
4			1	1		1
5				1	1	

in the previous storing method, we saw it was taking n^2 space to store the graph, this is where the adjacency list comes into the picture, it takes a very small amount of space.

```
int main()
{
```

```
    int n, m;
    cin >> n >> m;
    // adjacency matrix for undirected graph
    // time complexity: O(n)
    int adj[n+1][n+1];
    for(int i = 0; i < m; i++)
    {
        int u, v;
        cin >> u >> v;
        adj[u][v] = 1;
        adj[v][u] = 1  // this statement will be removed in case of directed graph
    }
    return 0;
}
```

Using vector

```
int main()
{
    int n, m;
    cin >> n >> m;
    // adjacency list for undirected graph
    // time complexity: O(2E)
    vector<int> adj[n+1];
    for(int i = 0; i < m; i++)
    {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    return 0;
}
```

Patterns:

1. For the outer loop, count the number of lines
2. For the inner loop, focus on the columns and connect them somehow to the rows.
3. Print them '*' inside for loop
4. Observe symmetry [optional]

Dynamic Programming:

KnapSack

```
public class Knapsack {
  // Function to solve the 0/1 Knapsack problem using dynamic programming
  public static int knapsack(int[] weights, int[] values, int capacity) {
    int n = weights.length; // Number of items
    int[][] dp = new int[n + 1][capacity + 1]; // DP table

    // Build the DP table
    for (int i = 1; i <= n; i++) {
      for (int w = 0; w <= capacity; w++) {
        if (weights[i - 1] <= w) {
          // Max of including the current item or excluding it
          dp[i][w] = Math.max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);
        } else {
          // Exclude the current item
          dp[i][w] = dp[i - 1][w];
        }
      }
    }

    // The maximum value that can be achieved with the given capacity
    return dp[n][capacity];
  }

  public static void main(String[] args) {
    int[] values = {60, 100, 120};  // Values of the items
    int[] weights = {10, 20, 30};  // Weights of the items
    int capacity = 50;  // Capacity of the knapsack

    int maxValue = knapsack(weights, values, capacity);
    System.out.println("Maximum value that can be obtained: " + maxValue);
  }
}
```

N-Queens:

```
public class NQueens {
  // Function to solve the N-Queens problem
  public static boolean solveNQueens(int board[][], int col, int N) {
    // If all queens are placed, return true
    if (col >= N) {
      return true;
    }

    // Try placing the queen in each row of the current column
```

```
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col, N)) {
            // Place the queen
            board[i][col] = 1;

            // Recur to place the rest of the queens
            if (solveNQueens(board, col + 1, N)) {
                return true;
            }

            // If placing the queen doesn't lead to a solution, backtrack
            board[i][col] = 0; // Remove the queen
        }
    }

    // If no placement is possible, return false
    return false;
}

// Function to check if it's safe to place a queen at board[row][col]
public static boolean isSafe(int board[][], int row, int col, int N) {
    // Check the left side of the current row
    for (int i = 0; i < col; i++) {
        if (board[row][i] == 1) {
            return false;
        }
    }

    // Check the upper diagonal on the left side
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    // Check the lower diagonal on the left side
    for (int i = row, j = col; i < N && j >= 0; i++, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    return true;
}

// Function to print the solution
public static void printSolution(int board[][], int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (board[i][j] == 1) {
```

```
                    System.out.print("Q ");
                } else {
                    System.out.print(". ");
                }
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        int N = 8; // Change this value for different sizes of the board
        int[][] board = new int[N][N];

        if (solveNQueens(board, 0, N)) {
            printSolution(board, N);
        } else {
            System.out.println("No solution exists.");
        }
    }
}
```

Interview Questions:

Data structures:

Data structures are the building blocks of any computer program as they help in organizing and manipulating data in an efficient manner. Without data structures, the computer would be unable to understand how to follow a program's instructions properly. It also defines their relationship with one another.