
Data Structures

Arrays

Static Arrays:

They have fixed sizes. Contiguous memory. All elements in the array are of same type.

- accessing elements = $O(1)$
- insertion = $O(1)$ //at the end else $O(n)$
- deletion = $O(1)$ //at the end else $O(n)$
- searching = $O(n)$ or $O(\log n)$ if sorted

Assumption: Space is available when inserting at the end.

Dynamic Arrays:

Same for dynamic arrays, except it has a resizing property.

Inserting/Removing at the end is still $O(1)$, even when space isn't available.

But it is amortized $O(1)$.

When no space, the array doubles/resizes, which is $O(n)$. But this happens infrequently i.e. this $O(n)$ is distributed over multiple insertions, which makes each insertion $O(1)$ on average.

Hashmap

- insertion, deletion, access, search among keys: $O(1)$ //called average case
- unless collision happens, in which case it could be $O(n)$ //worst case

checking if a key exists or not

`hm.count('key') = 1 if exists, else 0`

In C++, two implementations:

1. `map` - using trees - $O(\log n)$
2. `unordered_map` - using hash function - $O(1)$

implementation

a hashmap is implemented via a hash table, which contains:

- hash function, bucket array, collision resolution strategy
- **hash function:** takes **key** as input, returns hash code which is an integer, usually

example hash function:

```
int hashing (int key, int arraysize) {
    return key%arraysize;
}
```

how a key, value pair is stored

key → hash function → hash code → compressed to fit in indexes of bucket array → value is stored

When, for two different keys, the same index is generated, it is called **collision**.

Collision Handling

1. **Chaining** - Head of a linked list is stored at that index.
Worst case search/insert/delete is $O(n)$.
2. **Open Addressing** - All the elements are stored in the bucket array itself, so size of array must be \geq number of elements to store.

Linear Probing

- $new\ index = old\ index + i$, for i^{th} attempt

Quadratic Probing

- $new\ index = old\ index + i^2$, for i^{th} attempt

Worst case search/insert/delete is $O(n)$, but rare.

Load Factor = Number of expected elements at each position in the bucket array.
= (number of elements to insert)/(size of bucket array)

Ideally, $LF < 0.7$

Rehashing

When LF goes $> 0.7 \Rightarrow$ array size is doubled. All elements are rehashed. $O(n)$.

Stacks (LIFO)

STL:

```
stack<int> mystack;  
mystack.push(3);  
mystack.pop();  
mystack.top();
```

Functions:

- push, pop, peek, isempty, isfull, size
- monotonic decreasing stack is used to find next greater element. a stack that decreases as you go up.

Usage:

- used in undo mechanism, recursion, function call stack, browser history (back button)
- when you need to store history of choices, and need most recent on top
- to reverse a string, do decimal to binary
- Reverse Polish Notation:
Notation for mathematical expressions. Here you don't need brackets to signify precedence.
First, you write operands. Then operator.
e.g. "3 4 - 5 +" = 3 - 4 + 5

Implementation:

- via arrays (array size is fixed, can't resize stack)
- via linked lists (linked lists have dynamic size)

Array implementation:

```
#define MAX_SIZE 101  
int a[MAX_SIZE];  
int top = -1;  
  
void push(int ele){  
    if(top <= MAX_SIZE - 1){  
        a[++top] = ele;  
        cout<<"Pushed: "<<ele<<"\n";  
    }  
    else{  
        cout<<"Stack is full. Cannot push: "<<ele<<"\n";  
    }  
}  
  
int pop(){  
    if(top >= 0){
```

```

        int ele = a[top];
        top--;
        cout<<"Popped: "<<ele<<"\n";
        return ele;
    }
    else{
        cout<<"Stack is empty. Cannot pop.\n";
        return '-1';
    }
}

int peek(){
    if(top >= 0){
        int ele = a[top];
        cout<<"Peeked: "<<ele<<"\n";
        return 1;
    }
    else{
        cout<<"Stack is empty. Cannot peek.\n";
        return 0;
    }
}

int isEmpty(){
    return top == -1;
}

int isFull(){
    if(top >= MAX_SIZE){
        return 1;
    }
    else{
        return 0;
    }
}

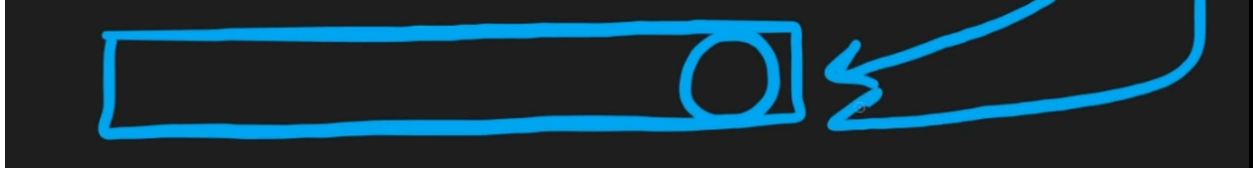
```

Queues

FIFO - First in first out

Enqueue - O(1)

Dequeue - O(1)



Two implementations:

1. Via linked list (Common)
2. Via arrays

Linked Lists

Vectors, when full, are doubled. Memory wastage. Amortized $O(1)$ addition at end.

On adding an element, only one new node is created in linked lists, hence no wastage. Actual $O(1)$ addition.

Insertion/deletion in the middle is also more efficient, as no need to shift elements like a vector. It is $O(1)$, but you have to reach there, which is $O(n)$.

Trees

In a tree with N nodes, there are $N-1$ edges. Proof: Induction. Find for base cases $f(0)$. Now, assume that it's true for $f(n)$, and using it prove for $f(n+1)$. Now if it's true for 0, then it's true for 1 as well, and 2, and so on.

Degree of a node = number of children

Depth & Height: Count vertices.

Note: The depth of root node is \emptyset and height of leaf nodes is \emptyset .

To ways to store:

Adjacency matrix - space: $O(V*V)$

Adjacency list - space: $O(V + E)$

Common mistake: While taking input, run loop till $(n-1)$ only as only $n-1$ inputs will be given. $n-1$ edges.

Depth-First Search (Stack, Recursion)

Time Complexity = $O(N+E) = O(N + N - 1) = O(N)$

Space Complexity = $O(1)$

```
void dfs(int node, int d, int parent, int & maxd) {
    maxd = max(d, maxd);
    for (auto i : tree[node]) {
        if (i!=parent) {
            dfs(i, d+1, node, maxd);
        }
    }
}

int main() {
    int maxd = 0;
    dfs(1, 0, -1, maxd);
}
```

Breadth-First Search (Queue)

We only go below a level after all the nodes at a level are traversed.

```
void bfs(int root, int n) {
    vector<bool> vis(n+1, false);
    queue<int> q;
    q.push(root);

    while (q.size()) {
        int front = q.front();
        q.pop();
        cout << front << " ";
        vis[front] = true;
        for (int i : tree[front]) {
            if (!vis[i]) {
                q.push(i);
            }
        }
    }
}
```

Binary Trees

Every node has at maximum two children.

Full Binary Tree: Each node has 0 or 2 children.

Complete Binary Tree (used in heaps): Each level is completely filled, except the last one, which is filled from the left.

Balanced Binary Tree: For each node, (height of left subtree ~ height of right subtree) ≤ 1

For a balanced binary tree with n nodes, the height is $\log_2(n)$.

Implementation

```
struct node{
    int val;
    node* left;
    node* right;
};

node* node_new(int val){
    node* temp = new node();
    temp->val = val;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}
```

Types of traversals

```
void preOrderTraversal(Node* root) {
    if (root==NULL) return;
    cout << root->val << " ";
    preOrderTraversal(root->left);
    preOrderTraversal(root->right);
}

void postOrderTraversal(Node* root) {
    if (root==NULL) return;

    postOrderTraversal(root->left);
    postOrderTraversal(root->right);
    cout << root-> val << " ";
}

void inOrderTraversal(Node* root) {
    if (root==NULL) return;

    inOrderTraversal(root->left);
```

```

    cout << root->val << " ";
    inOrderTraversal(root->right);
}

```

pre-order traversal: root → left subtree → right subtree //works like normal dfs

post-order traversal: left subtree → right subtree → root

in-order traversal: left subtree → root → right subtree

time complexities: $O(n)$

space complexities: $O(h)$ for a recursive approach, where h is height of tree.

level-order traversal:

- (bfs)
- traversing all the nodes level-wise

basic level order:

```

void levelorder(Node* root) {
    if (root == NULL) return;
    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        Node* top = q.front();
        cout << top->data << " ";
        if (top->left != NULL) q.push(top->left);
        if (top->right != NULL) q.push(top->right);
        q.pop();
    }
}

```

more advanced level order:

```

vector<vector<int>> arr;
void levelorder(Node* root) {
    if (root == NULL) return;

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        int size = q.size();
        vector<int> lev;
        for (int i=0; i<size; i++) {
            Node* top = q.front();
            lev.push_back(top->data);
            if (top->left != NULL) q.push(top->left);
            if (top->right != NULL) q.push(top->right);
            q.pop();
        }
    }
}

```



```
    arr.push_back(lev);  
    }  
}
```

vertical traversal (storing coordinates of each node):

[leetcode](#)

Binary Search Trees

- For each node, every node on the left is smaller and every node on the right is larger.
- Inorder traversal of a BST = Sorted Array
- Search, Insertion, Deletion = $O(h)$ time, or $O(\log 2n)$ for height-balanced trees
- For a skewed BST, $O(n)$

Definition

```
struct TreeNode {  
    int key;  
    TreeNode* left;  
    TreeNode* right;  
    TreeNode(int x) : key(x), left(NULL), right(NULL) {}  
};
```

Insertion

```
TreeNode* insert(TreeNode* node, int key) {  
    // If the tree is empty, return a new node  
    if (node == NULL) {  
        return new TreeNode(key);  
    }  
  
    // Otherwise, recur down the tree  
    if (key < node->key) {  
        node->left = insert(node->left, key);  
    } else if (key > node->key) {  
        node->right = insert(node->right, key);  
    }  
  
    // Return the (unchanged) node pointer  
    return node;  
}
```

Deletion

```
TreeNode* deleteNode(TreeNode* root, int key) {
    // Base case: if the tree is empty
    if (root == NULL) return root;

    // Otherwise, recur down the tree
    if (key < root->key) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->key) {
        root->right = deleteNode(root->right, key);
    } else {
        // Node with only one child or no child
        if (root->left == NULL) {
            TreeNode* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == NULL) {
            TreeNode* temp = root->left;
            delete root;
            return temp;
        }
    }
}
```

Searching

```
bool searchInBST(Node* root, int x) {
    if (root==NULL) return false;
    if(root->val==x) return true;

    if (x>root->val) return searchInBST(root->right, x);
    else return searchInBST(root->left, x);
}
```

Maximum/Minimum Node in BST

Maximum node is the rightmost node.

```
int maxNodeInBST(Node* root) {
    if (root->right==NULL) return root->val;
    return maxNodeInBST(root->right);
}
```

Heap

STL:

In the form of priority queue.

```
priority_queue<int> maxHeap;
```

```
priority_queue<int, vector<int>, greater<int>> minHeap;
```

```
push(), pop(), size(), empty(), top().
```

A special type of **complete binary tree**. Every parent has higher priority than its children.
Complete binary tree: Every level is completely filled except the last level. At the last level, nodes are always added from the left.

In fact, height of a complete binary tree is $\log(n)$.

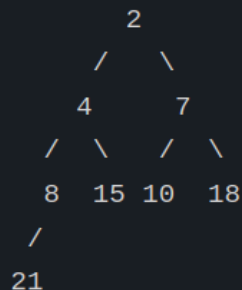
MinHeap - Parent smaller than or equal to its children.

MaxHeap - Opposite.

Some Common Applications of Heaps include:

- **Greedy Algorithms** - Heaps are used in many greedy algorithms where we have to make the most optimal choice at every step. Since the top of the heap always has the highest priority, we can choose the top element at every step.
- **Scheduling Algorithms** - Heaps are very useful in algorithms where we have to schedule tasks based on their priority.
- **Sorting** - Heaps can be very useful in sorting elements based on priority.

For Example - The binary tree



can be represented in array as **[2, 4, 7, 8, 15, 10, 18, 21]**

Here the children of a node at index i are stored at indexes $2i + 1$ and $2i + 2$.

Note that the arrays is 0-indexed.

$children = 2i + 1$ and $2i + 2$

$parent = (child - 1)/2$

MaxHeap Implementation:

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

int peek(vector<int>& heap) {
    return heap[0];
}

void insert(vector<int>& heap, int key) {
    heap.push_back(key);
    siftUp(heap, heap.size() - 1);
}

int pop(vector<int>& heap) {
    int key = heap[0];
    swap(heap[0], heap.back());
    heap.pop_back();
    siftDown(heap, 0);
    return key;
}

int siftUp(vector<int>& heap, int index) {
    while (index > 0 && heap[index] > heap[(index - 1) / 2]) {
        swap(heap[index], heap[(index - 1) / 2]);
        index = (index - 1) / 2;
    }
    return index;
}

int siftDown(vector<int>& heap, int index) {
    int size = heap.size();
    while (2 * index + 1 < size) {
        int left_child = 2 * index + 1;
        int right_child = 2 * index + 2;
        int largest_child = left_child;

        if (right_child < size && heap[right_child] > heap[left_child]) {
            largest_child = right_child;
        }

        if (heap[index] >= heap[largest_child]) {
            break;
        }

        swap(heap[index], heap[largest_child]);
    }
}
```

```
    index = largest_child;
  }
  return index;
}
```

Algorithms

Linear Search

Best: $O(1)$

Worst: $O(n)$

Binary Search (Monotonic Functions)

Can be applied on:

Monotonic functions: Anything that follows an order. Example: 1,2,3,4 or 5,2,1

Predicate function: That returns a boolean value in the following way (pehle T and then sirf F or vice versa) -

T T T T T F F F F or F F F F F T T T T

Best: $O(1)$

Worst: $O(\log_2(n))$

Derivation: $N/2^k = 1$ (After k steps, one element is left)

STL:

binary_search Function

```
bool binary_search(start_ptr, end_ptr, num);
```

This function returns true if the element *num* is present in the container's binary search space defined by *start_ptr* and *end_ptr*, otherwise, it returns false.

- *end_ptr* is not included. search range: [*start_ptr*, *end_ptr*)

lower_bound Function

```
iterator lower_bound(start_ptr, end_ptr, num);
```

The *lower_bound* function returns an iterator pointing to:

- The first occurrence of *num* if the container has multiple occurrences.
- The position of *num* if there's only one occurrence.
- The position of the first element greater than *num* if *num* is not found.

upper_bound Function

```
iterator upper_bound(start_ptr, end_ptr, num);
```

The *upper_bound* function provides an iterator pointing to:

- The position of the next higher number than *num*.

In other words, *lower_bound* returns the position of the first element which is $\geq num$. And *upper_bound* returns the position of the first element which is $> num$.

Another way to look at it is that *lower_bound* is the first index where you can insert *num*, and *upper_bound* is the last index where you can insert *num*, such that after insertion, the array is still sorted.

Multiple Choice Question

If the array is *A* = [2, 5, 8, 9, 13, 18, 20, 28, 35, 40], what would be *binary_search*(*A*, *A*+8, 28)?

- Search space: 2 (A), 5 (A+1), 8, 9, 13, 18, 20, 28 (A+7)
- (True)

Multiple Choice Question

If the array is $A = [2, 5, 8, 11, 11, 11, 20, 28, 35, 40]$, what would be `lower_bound(A, A+10, 100)`?

- *it points at $end + 1$, i.e end_ptr , which is $A[10]$ (DNE)*

- To figure out if a given array is ascending or descending sorted, compare the first and last element.
- Can apply on a range of (sorted) numbers, by assuming an imaginary array.
- If multiple equals exist, and you want the leftmost one:
Send left on finding both equal, and greater element.

End and Start pointers:

When the element is found: at that element.

When the element is not found: start is at the just bigger element, end is at the just smaller element.

The code below finds the number if it exists, or just greater element (ceil). Same thing but for just smaller element, return 'right' at the end (floor).

```
int left = 0;
int right = size - 1;

while (left <= right) {
    int mid = left + (right - left) / 2;
    if (arr[mid] == target) {
        return mid;
    }
    if (arr[mid] < target) {
        left = mid + 1;
    }
    else {
        right = mid - 1;
    }
}
```

```
    return left;
}
```

Q. First & Last position of occurrence of an element in an array, {-1, -1} if DNE

=> See Leetcode

Recursion

- bigger problem → smaller problem of same type
- recursion = base case + recurrence relation
- for all recursions, you can draw: **function call stack**, **recursion tree**

function call stack

```
function fun(){
    fun()
}
function main(){
    fun()
}
```

Infinite Recursion!

A diagram illustrating a function call stack. It consists of a vertical rectangle with a thin white border. Inside the rectangle, the following function names are listed from top to bottom: fun(), fun(), fun(), fun(), and main(). The 'main()' function is at the bottom, and four 'fun()' functions are stacked on top of it, representing a recursive call that never returns.

One-Branch Recursion

$$2^n = 2 * 2^{n-1}$$

recurrence relation: $f(n) = 2 * f(n - 1)$

$$n! = n * (n - 1)!$$

Recurrence Relation: $f(n) = n * f(n - 1)$

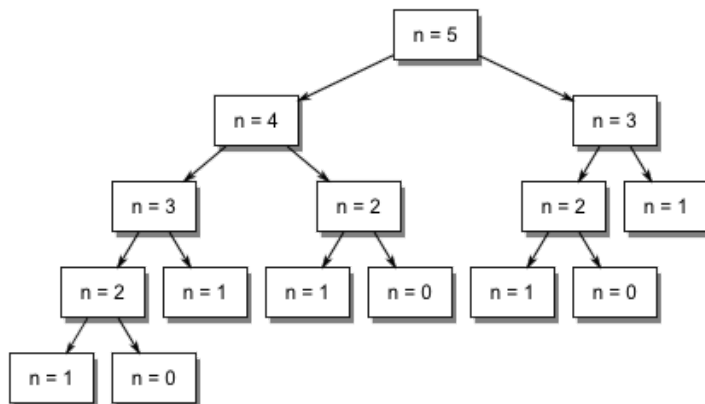
Two-Branch Recursion

Fibonacci series: 0 1 1 2 3 5 8...

1st term, 2nd term, etc..

$$f(n) = f(n - 1) + f(n - 2)$$

```
int fib(int n) {
    if (n<=2) {
        return n-1;
    }
    return fib(n-1) + fib(n-2);
}
```



Head vs. Tail recursion

Note: base case is ALWAYS 1st

head(3) is: 2 3

```
void head(int n)
{
    if(n == 1)
        return;
    else
        head(n-1); // ←
    printf("head - n=%i\n",n);
}
```

tail(3) is: 3 2 1

```
void tail(int n)
{
    if(n == 0)
        return;
    else
        printf("tail - n=%i\n",n);
    tail(n-1); // ←
}
```

Questions:

Print Pattern

You are given an integer N , you have to print an upside down triangle with stars of base N .

For example: for $N = 5$:

```
*****
****
***
**
*
```

```

void print_pattern(int n) {
    if (n == 0) {
        return;
    }
    for (int i = 0; i < n; ++i) {
        cout << "*";
    }
    cout << endl;
    print_pattern(n - 1);
}

```

Linear Search using Recursion

You are given an array Arr and an integer X . You have to search the array Arr and check if X exists in Arr or not, if yes print the first position of X else print -1 .

Input Format

- The first line contains two integers N and X - denoting the number of elements in Arr and the number to search in the array.
- The second line contains N integers.

Output Format

Output the first index of X in Arr if it exists, else output -1 .

Constraints

- $1 \leq N \leq 10^5$.
- $1 \leq A_i \leq 10^5$.
- $1 \leq X \leq 10^5$.

```
int linearSearch(vector<int>& arr, int n, int x) {
    if (n==arr.size()) {
        return -1;
    }

    if (arr[n]==x) {
        return n;
    }

    return linearSearch(arr, n+1, x);
}
```

Q. Check if string s is a palindrome or not

```
string checkPalindrome(string s, int n) {
    if (s[n]!=s[s.size()-1-n]) {
        return "No";
    }

    if (n>=s.size()-1-n) {
        return "Yes";
    }

    return checkPalindrome(s, n+1);
}
```

Q. Decimal to binary

method 1: printing

```
void Binary(int n) {
    if (n==0 || n == 1) {
        cout << n;
        return;
    }

    Binary(n/2);

    cout << n%2;
}
```

- head recursion

- head recursion mein cheeze ulti hoti hain. reverse order mein print hoga yahan.

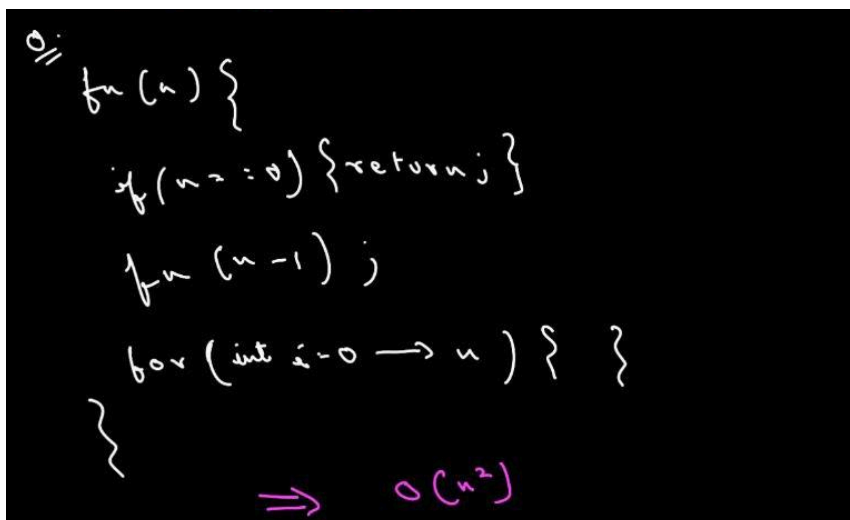
method 2: directly returning

```
string Binary(int n) {
    if (n==0 || n==1) {
        return to_string(n);
    }

    return Binary(n/2) + to_string(n%2);
}
```

Time Complexity in Recursion

TC = (TC of each call)*(number of calls)



Q.//

```
f(n) {
    if (n == 0) { return; }
    f(n-1);
    for (int i = 0 → n) { }
}
```

$\Rightarrow O(n^2)$

Backtracking

Backtracking has a template. You create an array, then you call a function. The base case of the function pushes to the array. Backtracking, you do, when you have to generate subsets or all possible combinations of brackets or something.

Prefix Sums

Prefix - A contiguous array, starting at index 0.

Prefix Sum - Sum of this array.

Let prefArray the prefix sum array of origArray:

$\text{prefArray}[i] = \text{prefArray}[i-1] + \text{origArray}[i]$

Q. Calculate sum of an array from index i to j .

=> Make prefix/postfix array once in $O(n)$, then calculate as many times as you want in $O(1)$.

Two Pointers

Q: Check if an array is a palindrome.

Q: Given a sorted input array, return the two indices of two elements which sum up to the target value. Assume there's exactly one solution.

Dynamic Programming

Pattern

- **Need to try all possible ways** => implies recursion
- Now, find the best way out of all the possible ways or count the number of ways or number of permutations or number of subsets => implies recursion
- After you know it's recursion, make it **DP**

Trick to write recursion

- Represent the problem in terms of index
- Do all possible things with that index, given in question
- Sum of all possible things (if you have to count ways)
- 2^{20} is approximately 10^7 - 10^8 . Above this, we can't compute.
- $1 + 2 + 2^2 + 2^4 + \dots + 2^n = 2^{(n+1)} - 1$
- TC for fibonacci: 2^n . Hence, can't calculate 2^{20} .

Two approaches:

1. Top-Down (Breaking a larger problem into smaller ones). Writing recursion then memoising it.
2. Bottom-Up (Starting from the bottom, calculating values for $n = 0, 1$ (initial) and then finding bigger values). Using a loop.

Complexity transformation

1. $O(2^n) \rightarrow O(n)$
2. $O(n!) \rightarrow O(2^n)$

```
int dp[N];  
// TOP DOWN  
  
int fib(int n){  
    if(n == 0) return 0;  
    if(n == 1) return 1;  
    if(dp[n] != -1) return dp[n];  
    // memoise  
  
    return dp[n] = fib(n-1) + fib(n-2);  
}
```

```
// BOTTOM APPROACH  
dp[0] = 0;  
dp[1] = 1;  
for(int i=2; i <= n; ++i){  
    dp[i] = dp[i-1] + dp[i-2];  
}
```

Graphs

