RECURSION :-

A function calling itself.

Example 1 :- Print numbers in decreasing order

a) **High Level Thinking :-** $n = 5 \Rightarrow 5, 4, 3, 2, 1$

main problem	sub problem
f(5) → 5	f(4) → 4
4	3
3	2
2	1
1	

Solve main problem by recursively calling sub problem.

f(5) → print 5

call f(4)

$\therefore f(5) = 5$ f(4)

$= 5$ (4, 3, 2, 1)

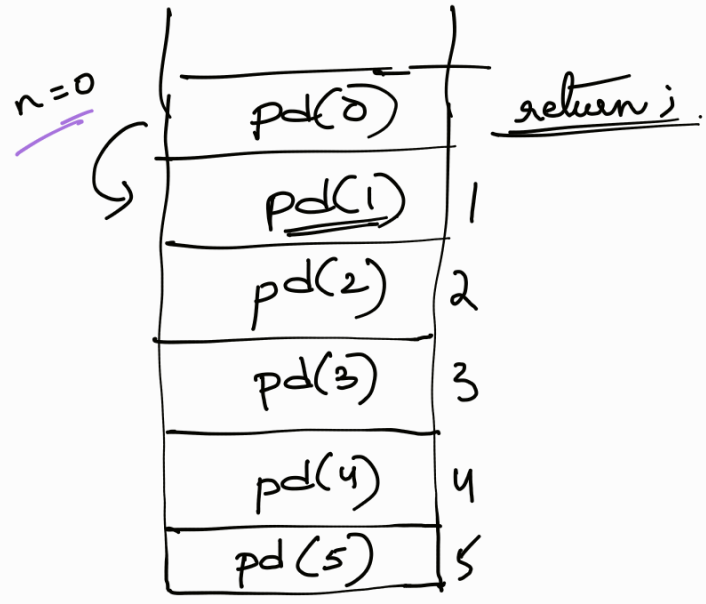
$\therefore$ $\boxed{f(n) = n + f(n-1)}$

Initial code :-

```
void printDecreasing(int n) {

    System.out.println(n);
    printDecreasing(n-1);

}
```

b) **Low level thinking :-**

$n = 5$



o/p:-

5

4

3

2

1

Base case :-

opt 1:-

```
if(n==0)
    return;
```

opt 2:-

```
sout(n)
if(n>1){
    printDecreasing(n-1);
}
```

Final code :-

```
App:- void printDecreasing(int n){
    if(n==0) return;
    System.out.println(n);
    printDecreasing(n-1);
}
```

pd(5)
pd(4)
pd(3)
pd(2)
pd(1)
pd(0)

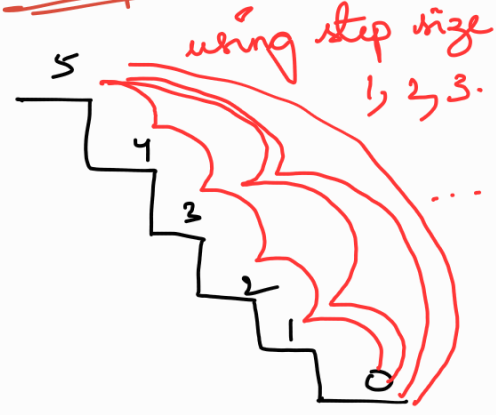
App 2 :-

```
void printDecreasing(int n) {
    System.out.println(n);
    if(n > 1) {
        printDecreasing(n-1);
    }
}
```

pd(5)
pd(4)
pd(3)
pd(2)
pd(1)

App 2 avoids extra recursive call to pd(0).

Example 2 :- Get Stair Paths. (Print all paths from 0 to n using step size 1, 2, 3.

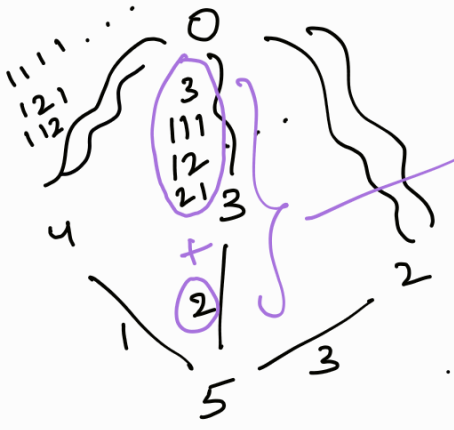


n = 3 step = 1, 2, 3.

[111, 12, 21, 3].

n = 5 step = 1, 2, 3

[11111, 1112, 1121, 113, 1211, 122, 131, 2111, 212, 221, 23, 311, 32]

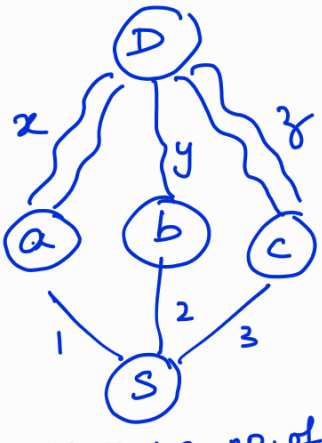


subpaths

step → 2 + {21, 12, 111, 3}

= {221, 212, 2111, 23}

11111, 1121, 1112 ...
221, 212, 2111, 23 ...



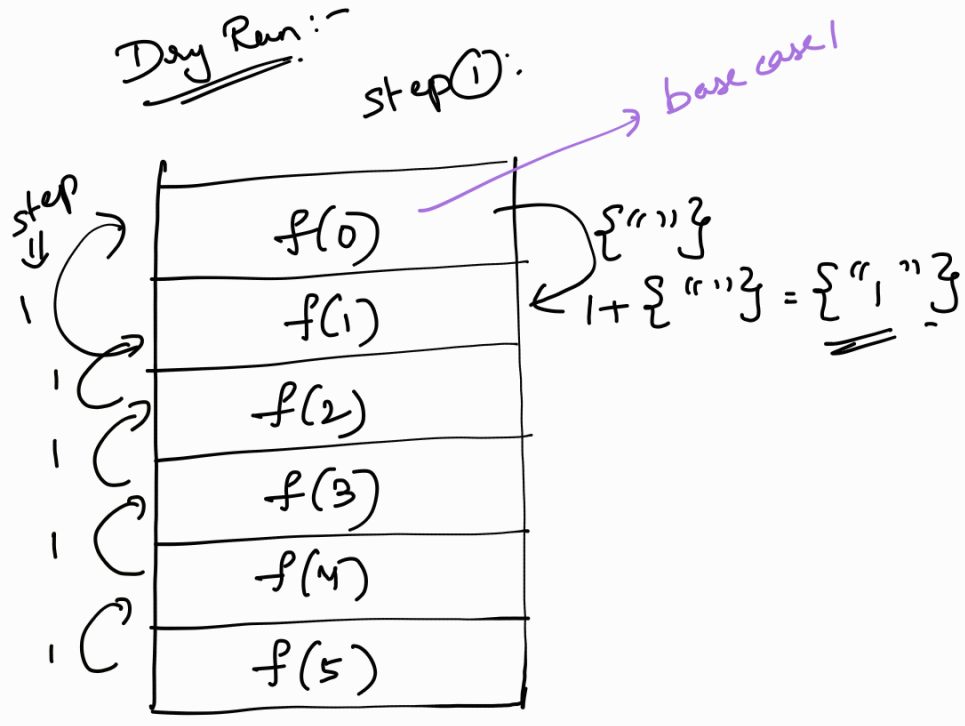
If there are x, y, z no. of ways from (a, b, c) to D, then there are x, y, z no. of of ways from s to D as well.

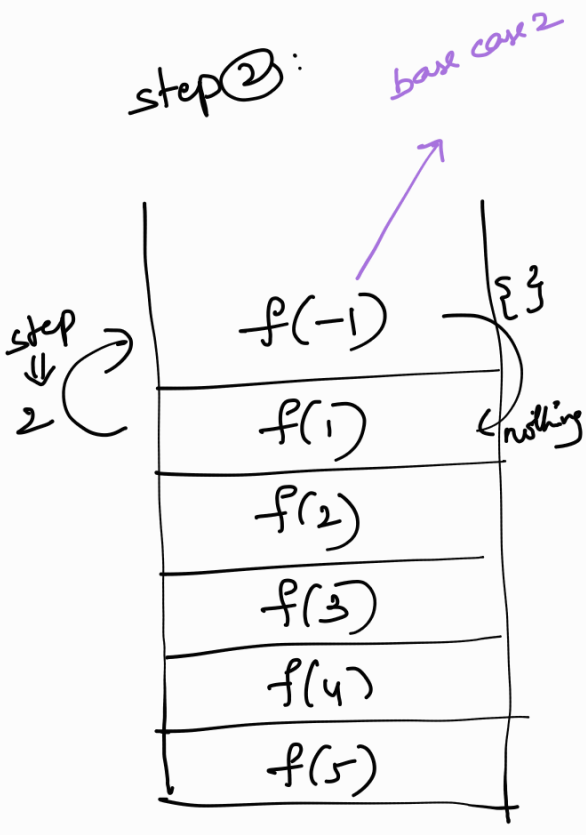
Code :-

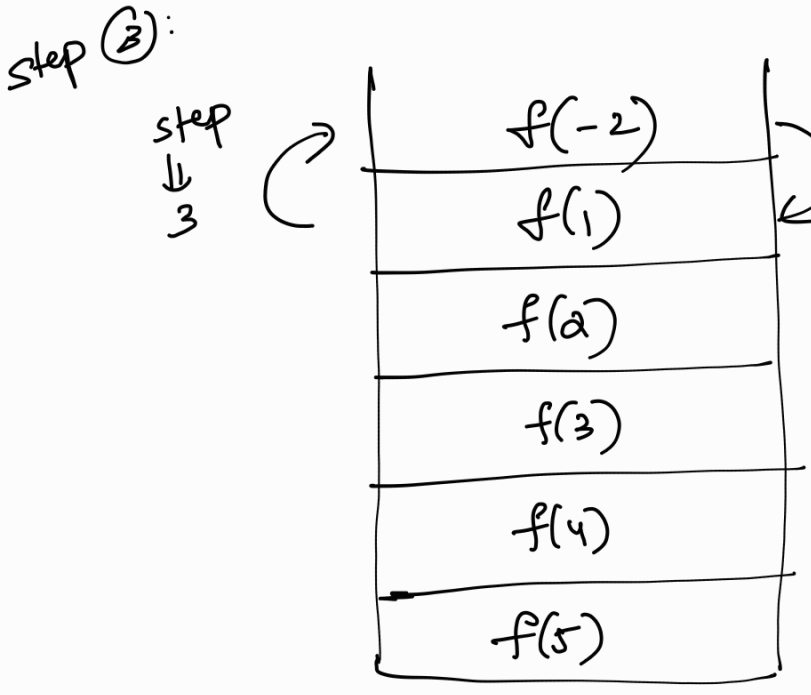
```
ArrayList<String> getStairPaths(int src, int[] arr) {
    ArrayList<String> paths = new ArrayList<>();
    for(int step : arr) {
        ArrayList<String> subStairPaths =
```

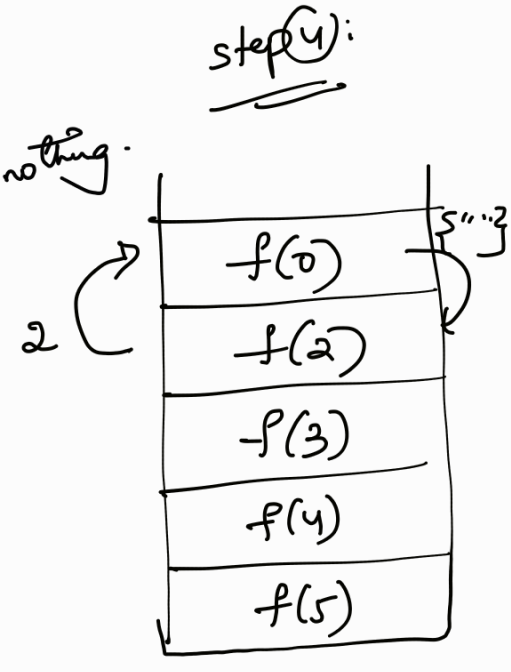
```
        getStairPaths(src-step, arr);
        for(String subPath: subStairPaths){
            paths.add(step + subPath);
        }
    }
    return paths;
}
```

Dry Run:-









so f(1) = {"1"} f(1) is complete
pop from stack

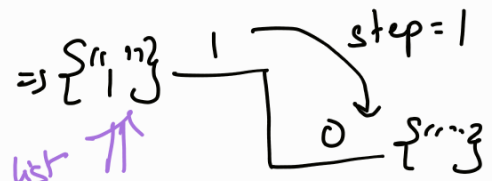
Code: -

```
ArrayList<String> getStairPaths(int src, int[] steps) {
    // base case(1) :-
    if (n == 0) {
        ArrayList<String> pathList =
                new ArrayList<>();
        pathList.add("");
        return pathList;
    }
    // base case(2) :-
    if (n < 0) {
        return new ArrayList<>();
    }
    ArrayList<String> paths = new ArrayList<>();
    for(int step : steps) {
        ArrayList<String> subpaths = getStairPaths
                (src-step, steps);
        for(String subPath : subPaths) {
            paths.add(step + subPath);
        }
    }
    return paths;
}
```

i/p :- n = 5

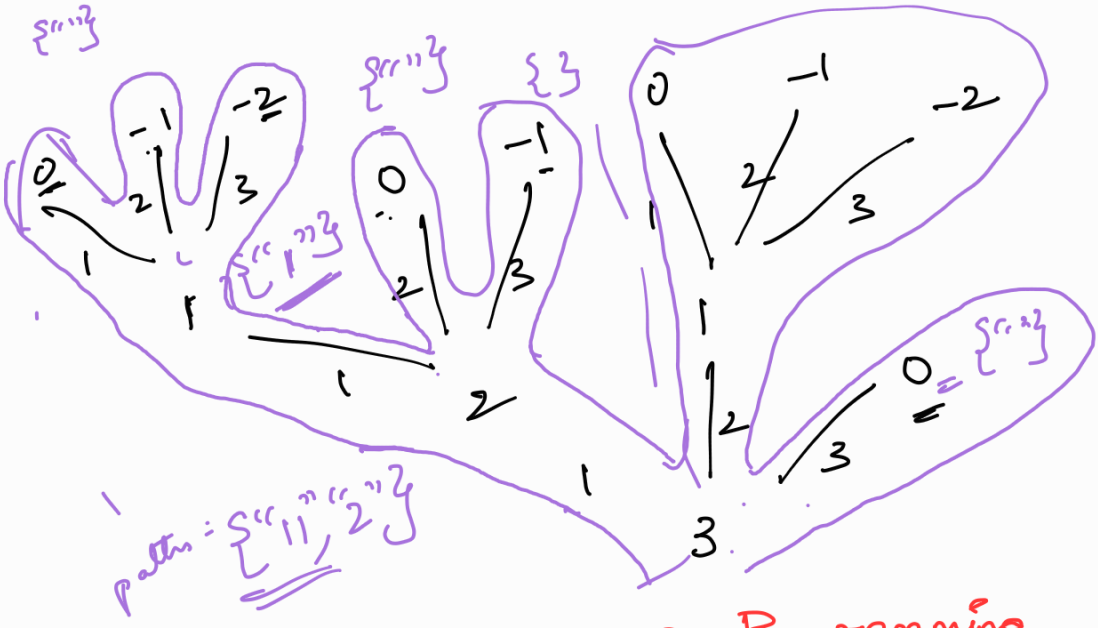
o/p :- [11111, 1112, 1121, 113, 1211, 122, 13,
2111, 212, 221, 23, 311, 32]

Base case 1: If we reached destination 0, we need to return an ArrayList of empty string such that the function calling function with src = 0 could append step size [1, 2, 3] to it.
If $f(1) \xrightarrow{step=1} f(0)$ and $f(0) = \{""\}$, then $f(1) = \{"1"\}$.

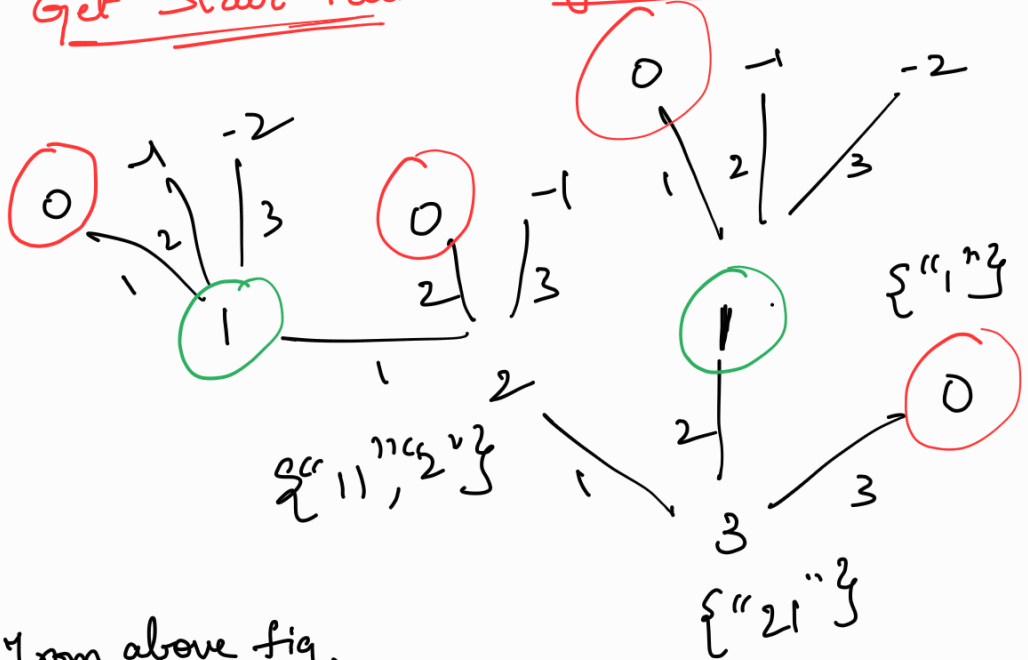


Base case 2:

Return empty list for src < 0 as we do nothing when we reached a point beyond 0.



Get Stair Paths - Dynamic Programming



cache:

f(0)	{""}
f(1)	{"1"}
f(2)	{"11","2"}

{""}

From above fig,

Function call to 1 happens 2 times

Function call to 0 happens 4 times

We can reduce it to 1 time by implementing cache. It takes additional space O(n).

```
ArrayList<String> getStairPaths(int src, int[] steps,
                                HashMap<Integer,
                                ArrayList<String>>
                                cache) {
    if (n == 0) {
        ArrayList<String> pathList =
                new ArrayList<>();
        pathList.add("");
        return pathList;
    }
    if (n < 0) {
        return new ArrayList<>();
    }
    if (cache.containsKey(src)) {
        return cache.get(src);
    }
    ArrayList<String> paths = new ArrayList<>();
    for(int step : steps) {
        ArrayList<String> subPaths = getStairPaths
                (src-step, steps);
        for(String subPath : subPaths) {
            paths.add(step + subPath);
        }
    }
    cache.put(src, paths);
    return paths;
}
```

RECURSION

The process in which a function calls by itself directly or indirectly is called recursion and the corresponding function is recursive function.

A recursive solution runs back onto itself, solving a problem by breaking it down into sub-problems of the same nature.

Structure :-

1) Base Case
2) Recursive Case
3) Function Prologue and Epilogue.

```
public void countdown(int n) {
    if (n == 0) {  // base case
        System.out.println("Happy new year!");
    } else {
        System.out.println(n);  // prologue
        countdown(n-1);  // recursive case.
    }
}
```

1) Base Case :-

* Simplest instance of a problem.
* Condition to stop further recursive calls.

2) Recursive Case :-

* Handles more complex instances of the problem.
* Further breaks down the problem and calls the function with the reduced problem size.

3) Function prologue and epilogue :-

Prologue: Operations before issuing new recursive call.
Epilogue: Operations after the recursive call.

Common mistakes :-

① Infinite recursion :-

```
decrementToZero(int n) {
    Sout(n);
    if (n>0) {
        decrementToZero(n); // n=5, 5, 5, .... ∞
    }
}
```

n-1 ✓ // n = 5, 4, 3, 2, 1

Tip: Ensure your function progresses towards base case.

② Ignoring the returned value :-

```
public int factorial (int n) {
    if (n==0) {
        return 1; ✓✓
    } else {
        n * factorial(n-1);
    }
}
```

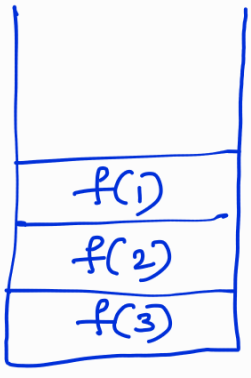
→ missing return

n = 3
O/p : 1

```
    return -1;
}
```

(3) **Excessive recursion:-**

* In the context of recursion, a call stack is a mechanism that keeps track of function calls in a program. It's an important part of recursion because it helps keep track of where the program should return to after a function call is required.



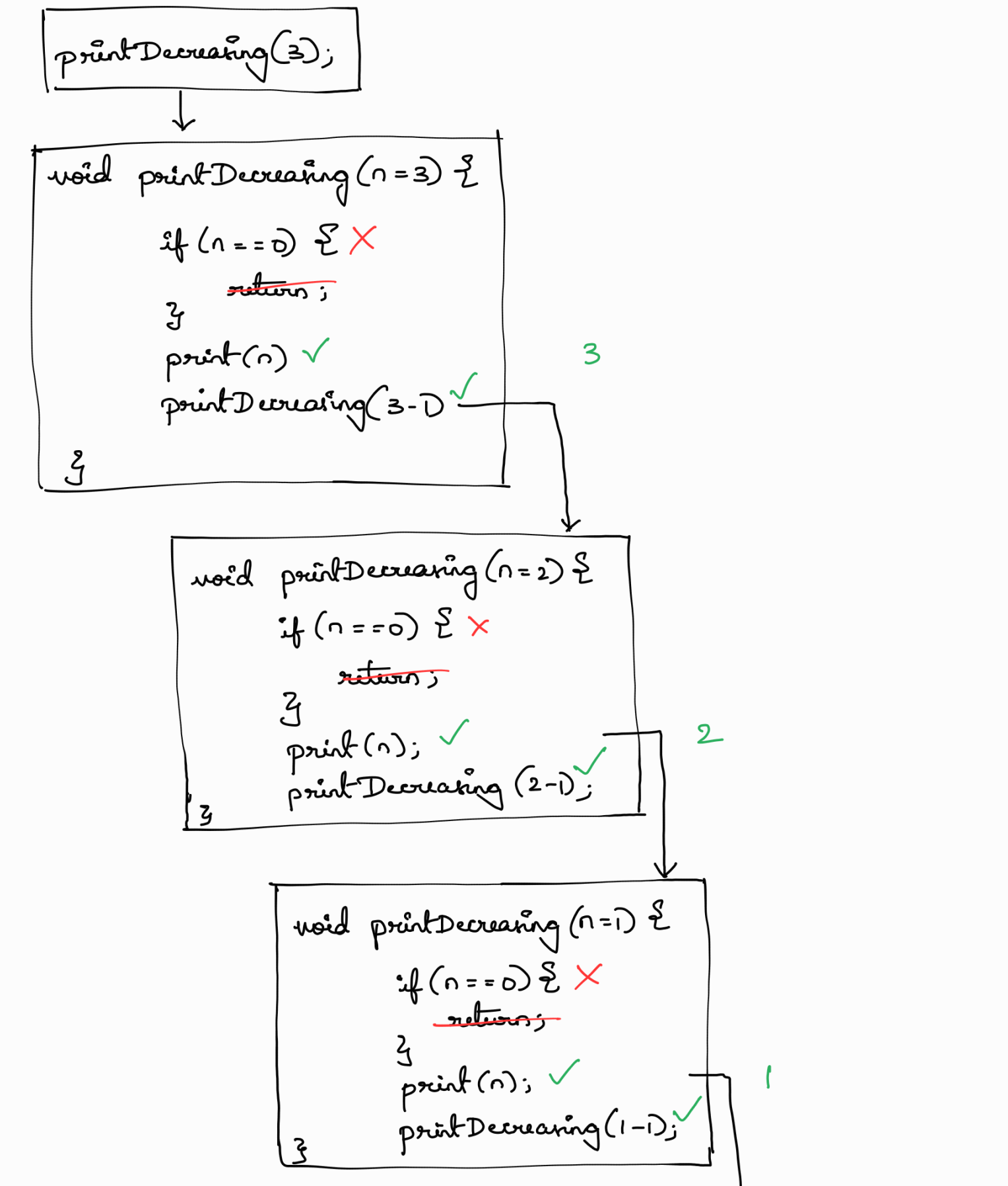
Call stack.

* When a function calls itself, each call to the function is added to the call stack. This includes the values of any parameters and variables. The program uses the call stack to keep track of these function calls.

* Then, when a base condition is reached, the program starts to "unwind" the stack. This means it starts to take function calls off the stack and finish them off one by one, using the variables and parameter values that were stored. This process continues until the stack is empty, which means all the function calls have been completed.

Since each function call adds a stack frame over the call stack, thereby, naively underestimating the recursion depth can lead to excessive load on the call stack. Hence, it can overload it resulting in the stack-overflow exception.

Recursion Tree Representation:-



```
void printDecreasing(n=0){
    if (n==0) {
        return;
    }
    printDecreasing(0-1);
    print(n);
}
```

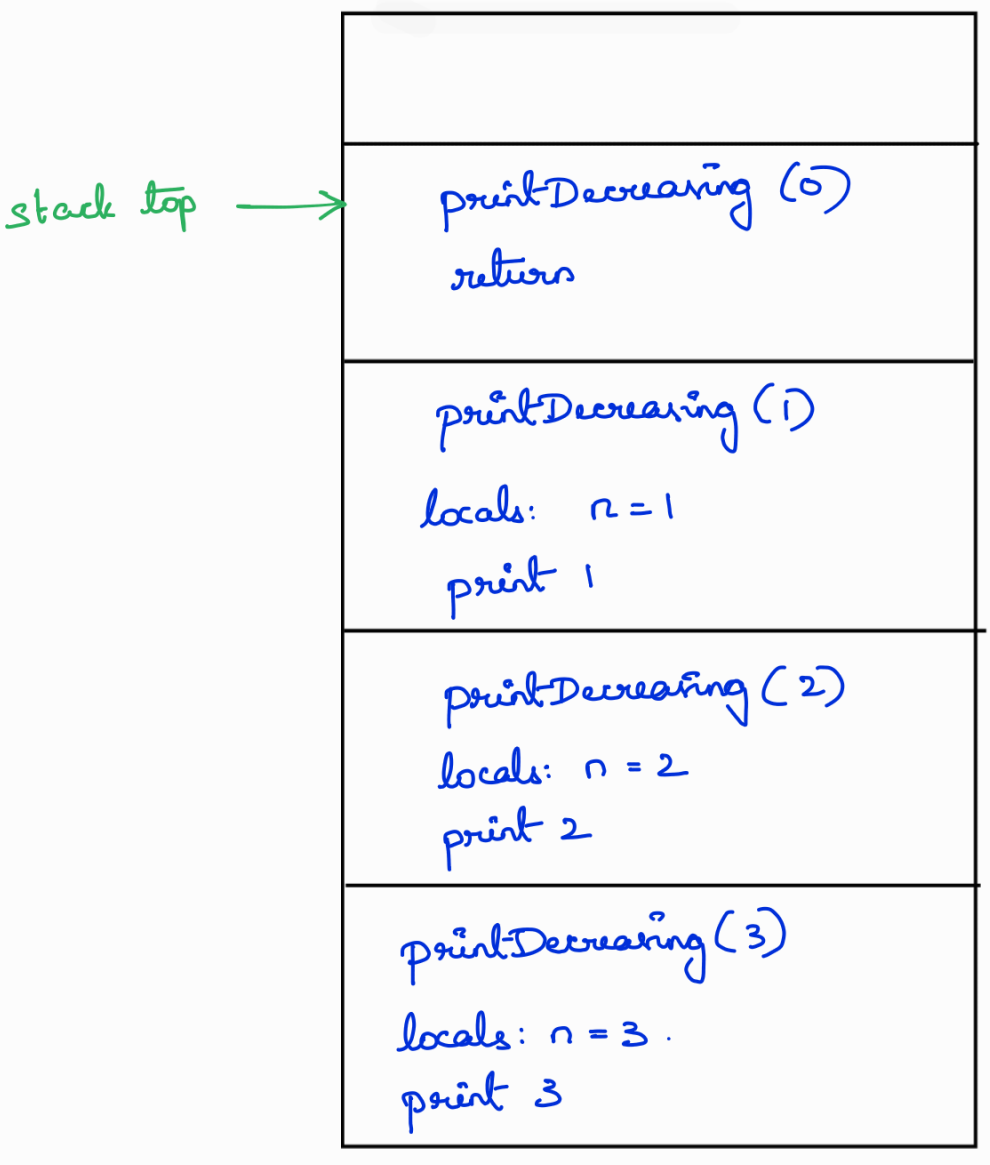
O/p:- 3
2
1

printDecreasing(3)
- print(3)
- printDecreasing(2)
  - print(2)
  - printDecreasing(1)
    - print(1)
    - printDecreasing(0)

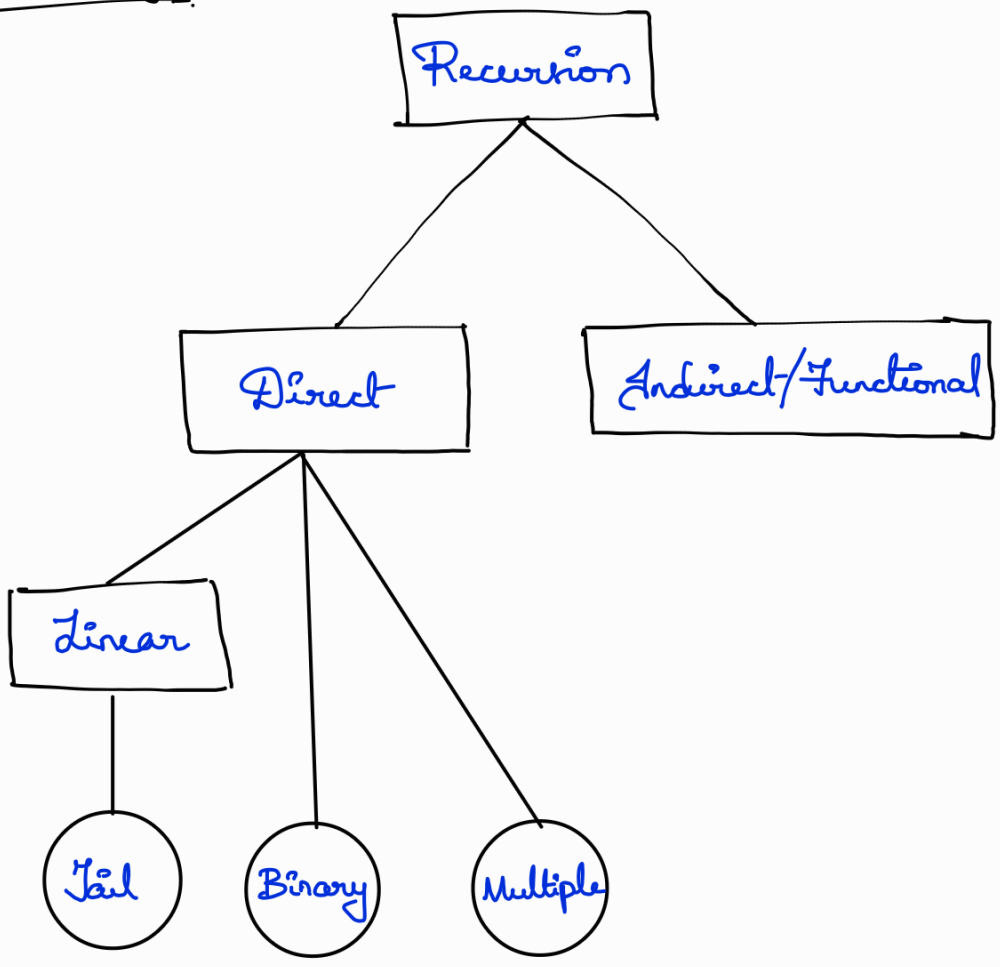
Call stack representation :-

```
void printDecreasing(n) {
    if(n==0) {
        return;
    }
    print n;
    printDecreasing(n-1);
}
```

1. Initially printDecreasing(3) is called. The state of the function, including the value 3, is stored in the first frame, and placed onto the call stack.
2. The function calls itself with n-1, which results in printDecreasing(2). This new state is stored in the another frame, and placed on top of the previous frame in the stack.
3. This process continues until we reach printDecreasing(0). This state is added to the top of the stack, and because n is 0, we simply don't return anything and make any recursive calls.



Recursion Types:-



① Linear recursion:- function makes a single recursive call to itself.
also known as single recursion.

Problems using linear recursion:-

① Factorial

```
int calculateFactorial(int n) {
    if(n==0 || n==1) {
        return 1;
    }
    return n * calculateFactorial(n-1);
}
```

② Array Summation

```
// sum(arr, 0);
```

```
int sum(int[] arr, int i) {
    if (i == arr.length) {
        return 0;
    }
    return arr[i] + sum(arr, i+1);
}
```

(3) String reversal

```
// reverseString(input, input.length-1);
String reverseString(String input, int i) {
    if (i < 0) {
        return new String();
    }
    return input.charAt(i) + reverseString
                             (input, i-1);
}
```

(4) Linked list traversal

```
void traverseLinkedList(Node head) {
    if (head == null) {
        return;
    }
    print head.val;
    traverseLinkedList(head.next);
}
```

(5) Checking palindrome

```
boolean checkPalindrome(String s, int i) {
    int n = s.length();
```

```
        if (i < n/2 &&
            s.charAt(i) == s.charAt(n-i-1)) {
                return checkPalindrome(s, i+1);
        }
        return false;
    }
```

(8)

```
boolean checkPalindrome (String s) {
    int n = s.length();
    if (n <= 1)
        return true;
    }
    if (s.charAt(i) == s.charAt(n-i-1)){
        return checkPalindrome(
            s.substring(1, n-1));
    }
    return false;
}
```

(6) Finding maximum in an array:

```
int findMax (int[] arr, int i) {
    if (i == arr.length) return arr[i];
    int max = findMax(arr, i+1);
    return arr[i] >= max : arr[i] : max;
}
```

② Tail Recursion :- subtype of linear recursion.
recursive call is the last operation in the function. ⇒ empty epilogue.

③ Binary Recursion :- A function makes two recursive calls to itself.

Problems using Binary Recursion :-

① fibonacci

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

② Merge Sort Divide array into two halves, sort each half independently and merge them back.

```
void mergeSort(int[] arr, int start, int end) {
    if (start < end) {
        int mid = (start+end)/2;
        mergeSort(arr, start, mid); //left half
        mergeSort(arr, mid+1, end); // right half
        merge(arr, start, mid, end); // merge halves.
    }
}
```

③ Quick Sort: Partition array around pivot, then sort both partitions independently.

```
void quickSort(List<Integer> array, int low,
                                    int high) {
    if (low < high) {
        int pivotIndex = partition(array, low,
                                    high);
        quickSort(array, low, pivotIndex - 1);
        quickSort(array, pivotIndex + 1, high);
    }
}
```

(4) Binary Tree Traversals (In-Order, Pre-Order, Post-Order) Each node requires visiting the left and right children independently.

```
void inorder(Node node, List<Integer> result) {
    if (node == null)
        return;
    inorder(node.left, result);
    result.add(node.val);
    inorder(node.right, result);
}

void preorder(Node node, List<Integer> result) {
    if(node == null)
        return;
    result.add(node.val);
```

```
        preOrder(node.left, result);
        preOrder(node.right, result);
    }
    void postOrder(Node node, List<Integer> result){
        if(node == null)
            return;
        preOrder(node.left, result);
        preOrder(node.right, result);
        result.add(node.val);
    }
```

(5) Height of Binary Tree: by finding maximum height of left and right sub-trees.

```
int maxDepth(TreeNode root) {
    if(root == null) return 0;
    int leftHeight = 1 + maxDepth(root.left);
    int rightHeight = 1 + maxDepth(root.
                                   right);
    return Math.max(leftHeight, rightHeight);
}
```

(6) Maximal Subarray Sum: Divide the array into two equal halves and find max sum in each half, then merge results considering the case when maximal sum subarray crosses middle of array.

```
public int maxSubArray(int[] nums) {
    return maxSum(nums, 0, nums.length-1);
}
public int maxSum(int[] nums, int left,
                  int right) {
```

```
if (left == right)
    return nums[left];

int mid = (left + right) / 2;
int sum = 0, leftMaxSum = Integer.
                          MIN_VALUE;
for (int k = mid; k >= left; k--) {
    sum += nums[k];
    if (sum > leftMaxSum) {
        leftMaxSum = sum;
    }
}
int rightMaxSum = Integer.MIN_VALUE;
sum = 0;
for (int k = mid+1; k <= right; k++) {
    sum += nums[k];
    if (sum > rightMaxSum) {
        rightMaxSum = sum;
    }
}
int maxLeftRight = Math.max(
    maxSum(nums, left, mid), maxSum(nums,
                             mid+1, right));
return Math.max(maxLeftRight,
        leftMaxSum + rightMaxSum);
}
```

Binary Recursion

(7) Closest Pair of Points: Divide the given set of points into two halves, find the closest pairs in each half, and finally, find the closest pairs across the two halves.

```
public int[][] kClosest(int[][] points, int K) {
    quick(points, 0, points.length-1, K);
    return Arrays.copyOfRange(points, 0, K);
}

private void quick(int[][] arr, int l, int h, int k){
    if (l >= h) return;
    int p = partition(arr, l, h);
    if (p == k-1) {
        return;
    } else if (p < k-1) {
        quick(arr, p+1, h, k);
    } else {
        quick(arr, l, p-1, k);
    }
}
```

Binary recursion

(8) Tower of Hanoi: Solve by recursively moving n-1 disks to an auxiliary peg, moving the n^{th} disk, and finally moving the n-1 disks again.

```
void move (int n, int src, int aux, int dest) {
    if (n == 1) {
        sout("Move disc1 from " + src + " to " +
                                        dest);
        return;
    }
    move(n-1, src, dest, aux);
    sout("Move disc " + n + " from " + src + " to "
                                        + dest);
    move(n-1, aux, src, dest);
}
```

Binary recursion

(9) Karatsuba algorithm: Split the digits of numbers into two halves and perform multiplications on the halves

```
String karatsuba (String X, String Y) {
    int n = Math.max(X.length(), Y.length());
    X = String.format("%" + n + "s", X).replace(' ', '0');
    Y = String.format("%" + n + "s", Y).replace(' ', '0');
    if (n == 1) {
        return Integer.toString(Integer.parseInt(X) *
                                Integer.parseInt(Y));
    }
    int m = n/2;
```

```
String Xl = X.substring(0, m);
String Xr = X.substring(m);
String Yl = Y.substring(0, m);
String Yr = Y.substring(m);

String P1 = karatsuba(Xl, Yl);
String P2 = karatsuba(Xr, Yr);
String P3 = karatsuba(addBinary(Xl, Xr),
                      addBinary(Yl, Yr));
```

Binary Recursion

. . . .
. . . .
. . .

}

(10) Skyline Problem: Divide the buildings into two halves, solve for each half, and then merge the results.

```
public List<int[]> getSkyline(int[][] buildings) {
    return getSkyline(buildings, 0, buildings.length-1);
}
private ArrayList<int[]> getSkyline(int[][] buildings,
                                    int lo, int hi) {
    ArrayList<int[]> result = new ArrayList<>();
    if (lo > hi) return result;
    if (lo == hi) {
        result.add(new int[] { buildings[lo][0],
```

```
                buildings[lo][2]});
        result.add(new int[]{buildings[lo][1], 0});
        return result;
    }
    int mid = (lo + hi)/2;
    ArrayList<int[]> left = getSkyline(buildings,
                                       lo, mid);
    ArrayList<int[]> right = getSkyline(buildings,
                                        mid+1, hi);
```

binary recursion

. . . . .
- - - -
- - - -

}

(11) Strassen's Algorithm for Matrix Multiplication:

Divide the matrices into four equal parts and recursively calculate the product for each pair of parts.

```
int[][] multiply(int[][] A, int[][] B) {
    int n = A.length;
    int[][] R = new int[n][n];
    if (n==1)
        R[0][0] = A[0][0] * B[0][0];
    else {
        int[][] A11 = new int[n/2][n/2];
        int[][] A12 = new int[n/2][n/2];
        int[][] A21 = new int[n/2][n/2];
```

```
int[][] A22 = new int[n/2][n/2];
int[][] B11 = new int[n/2][n/2];
int[][] B12 = new int[n/2][n/2];
int[][] B21 = new int[n/2][n/2];
int[][] B22 = new int[n/2][n/2];

split(A, A11, 0, 0);
split(A, A12, 0, n/2);
split(A, A21, n/2, 0);
split(A, A22, n/2, n/2);

split(B, B11, 0, 0);
split(B, B12, 0, n/2);
split(B, B21, n/2, 0);
split(B, B22, n/2, n/2);

int[][] M1 = multiply(add(A11, A22),
                      add(B11, B22));
int[][] M2 = multiply(add(A21, A22), B11);
int[][] M3 = multiply(A11, sub(B12, B22));
int[][] M4 = multiply(A22, sub(B21, B11));
int[][] M5 = multiply(add(A11, A12), B22);
int[][] M6 = multiply(sub(A21, A11),
                      add(B11, B12));
int[][] M7 = multiply(sub(A12, A22),
                      add(B21, B22));
```

Binary recursion

multiple recursion

. . . . .
. - - .

④ Multiple recursion:-

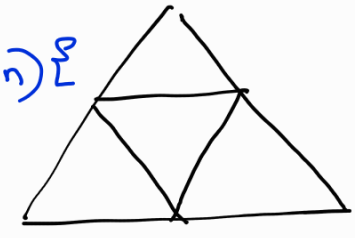
* sometimes known as "multi-way recursion"
* a function makes more than two recursive calls.

Eg: Ternary tree.

```
function traverse (node):
    if node is not null:
        print (node.value)
        traverse (node.left)
        traverse (node.middle)
        traverse (node.right)
```

Problems using Multiple recursion:

① Sierpinski triangle: A fractal design where each large triangle subdivides into 3 smaller ones in the next iteration, naturally leading to 3 recursive calls.



```
sierpinski (double x, double y, double s, int n){
    if (n<=0) {
        return;
    }
    . . . .
    . . . .
    . . . .
    sierpinski (x1, y1, s/2.0, n-1);
    sierpinski ((x1+x2)/2.0, (y1+y2)/2.0, s/2.0, n-1);
    sierpinski ((x1+x3)/2.0, (y1+y3)/2.0, s/2.0, n-1);
}
```

(2) Fractal Tree: Drawing a tree with multiple branches leads to multiple recursive calls - one for each branch

(3) Tremaux's algorithm: Tremaux's algorithm leverages upto four recursive calls to explore all possible directions (North, South, East, West) from each cell in a maze, marking a successful path, and backtracking when a dead-end is encountered.

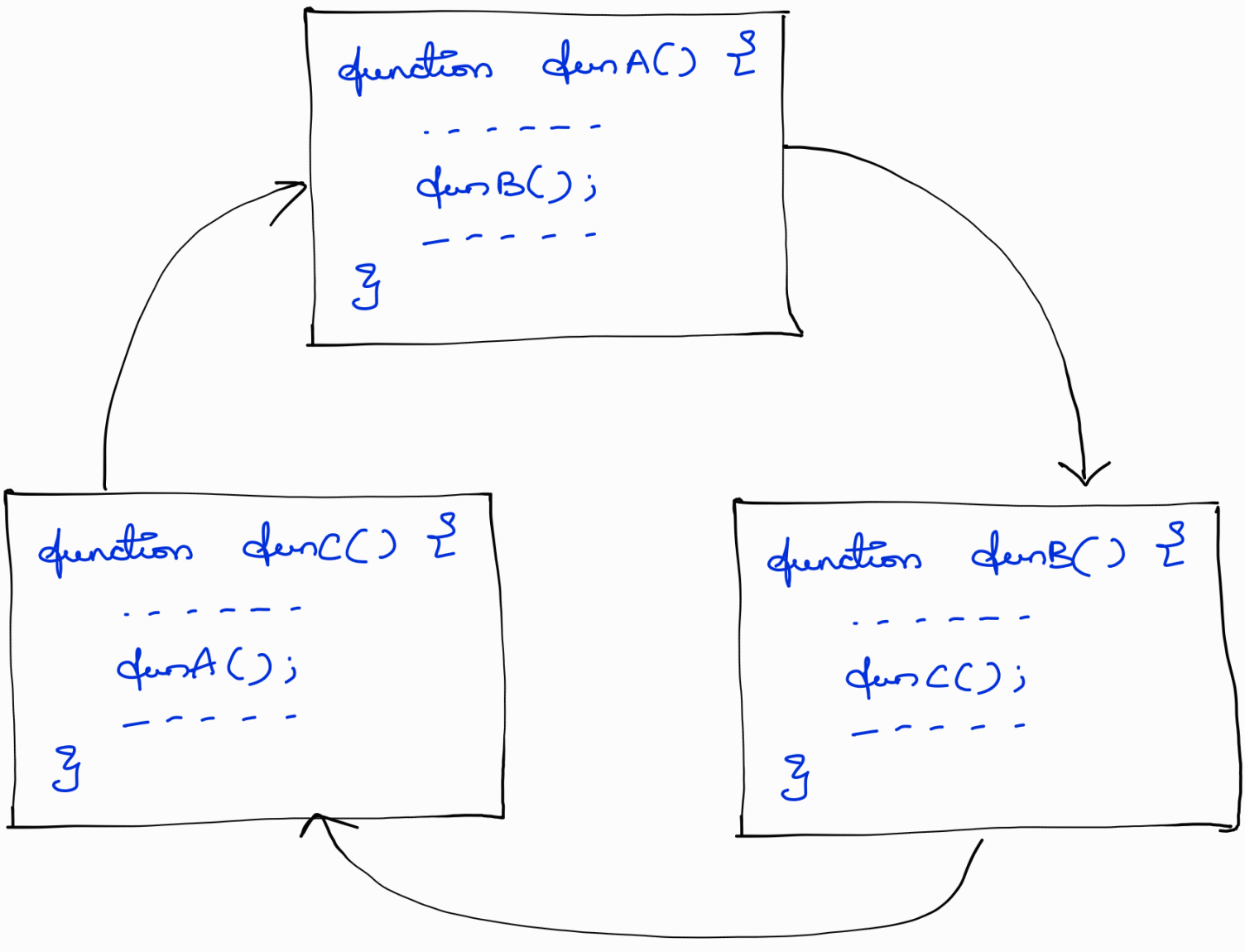
(4) FloodFill algorithm: Given a screen, a point on the screen and a color, the FloodFill algorithm colors the selected pixel and all adjacent (vertically and horizontally) pixels of the original color with the new color. It employs upto four recursive calls to fill a pixel and its adjacent pixels on a screen with a new color, replicating the fill tool in graphics editors.

```
void floodfill(int[][] maze, int sr, int sc, String asf,
                                        boolean[][] visited) {

    if(sr<0 || sc<0 || sr > maze.length-1 ||
sc > maze[0].length-1 || maze[sr][sc] == 1 || visited[sr][sc])
        return;

    if (sr == maze.length-1 && sc == maze[0].length-1){
        sout(asf);
        return;
    }
```

```
visited [sr][sc] = true;
floodfill (maze, sr-1, sc, asf + "t", visited);
floodfill (maze, sr, sc-1, asf + "l", visited);
floodfill (maze, sr+1, sc, asf + "d", visited);
floodfill (maze, sr, sc+1, asf + "r", visited);
visited [sr][sc] = false;
}
```

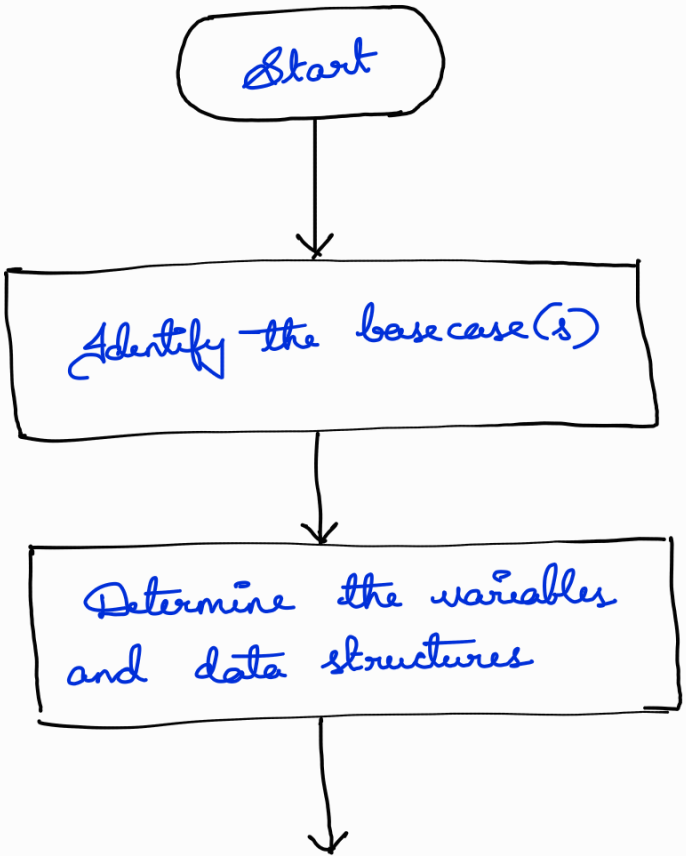
⑤ Indirect / functional / multi-level Recursion :-

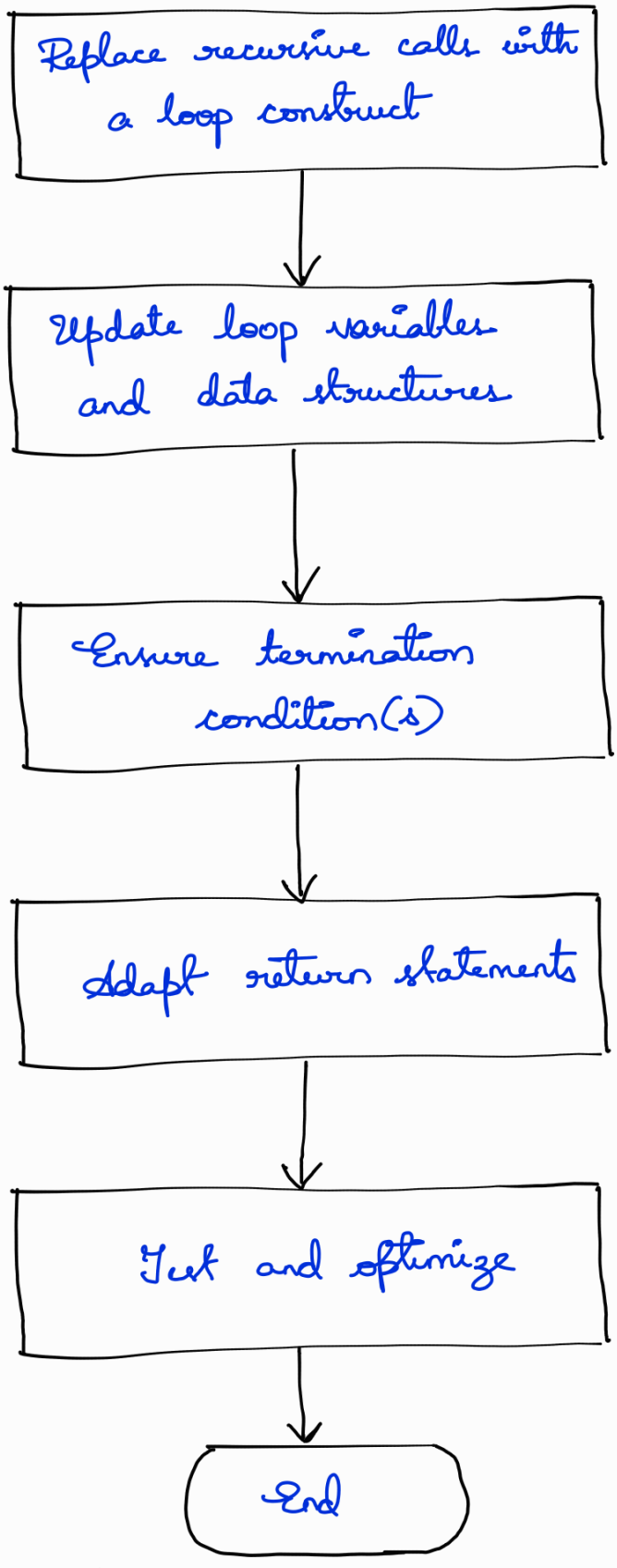
when a function is called not directly by itself but by another function that it calls, resulting in a cycle of function calls.



Linear	Tail	Binary	Multiple	Multi-level / Indirect
Single call	Single call	Two calls	Multiple	Multiple (via different functions)
One branch	Last action	Split in half	Complex splits	Cross-function calling
Least computational/low memory usage	Can easily translated into iterative equivalent	Suits problems that naturally partition to halves	Can solve complex problems -	Can solve complex multi-functional problems.
Limited to linear problems	Less intuitive than other forms of recursion	Potential for excessive recursive calls, can be memory intensive	High time complexity, hard to manage.	Increased complexity, hard to debug.

Converting a Recursive Solution to an Iterative Equivalent :-





Example:- factorial.

```
function factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Step① base case: when n is 0 & 1, value is 1.

Step② single variable "result" is needed to store result.

Step③ for loop n → 1.

Step④ i = decreasing value of n

Step⑤ i reaches until base case 1

Step⑥ return result

Step⑦ Test & optimize.

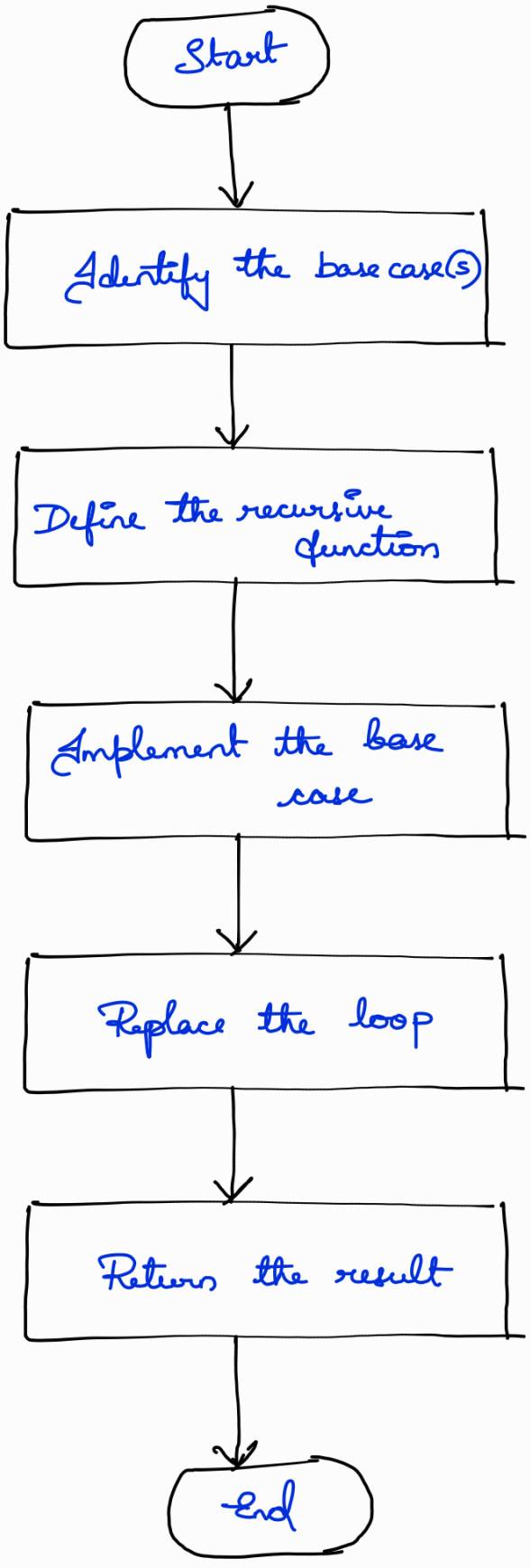
```
function factorialIterative(n):
    result = 1
    for i from n down to 1:
        result = result * i
    return result
```

Note:- Every recursive solution has an iterative equivalent. Because, Recursion and Iteration are both programming constructs that allow you to repeat certain instructions or steps.

Recursion does this by having a function call itself until a base case is reached, while iteration does it with looping constructs like for, while, and do-while loops.

To convert a recursive solution to an iterative one, you generally need to manage the state that would normally be maintained by the recursion stack itself. This often involves using your own stack or queue structure.

Converting an Iterative Solution to a Recursive Equivalent:-



Example: factorial

```
function factorialIterative(n):
    result = 1
    for i from n down to 1:
        result = result * i
    return result
```

Step①: Base case is when i reaches 1 & for input 0, function returns 1.

Step②: factorialRecursive(n)

Step③: if n==0 || n==1 return 1

Step④: replace loop with recursive call factorialRecursive(n-1)

Step⑤: return result of recursive call multiplied by n.

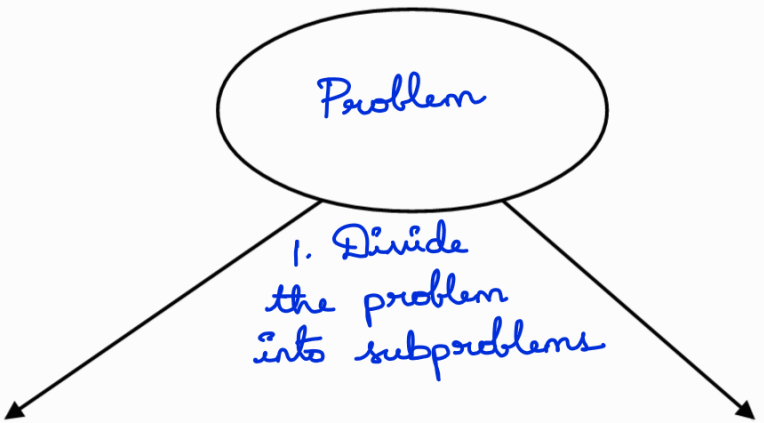
```
function factorialRecursive(n):
    if n==0 || n==1:
        return 1
    else:
        return n * factorialRecursive(n-1)
```

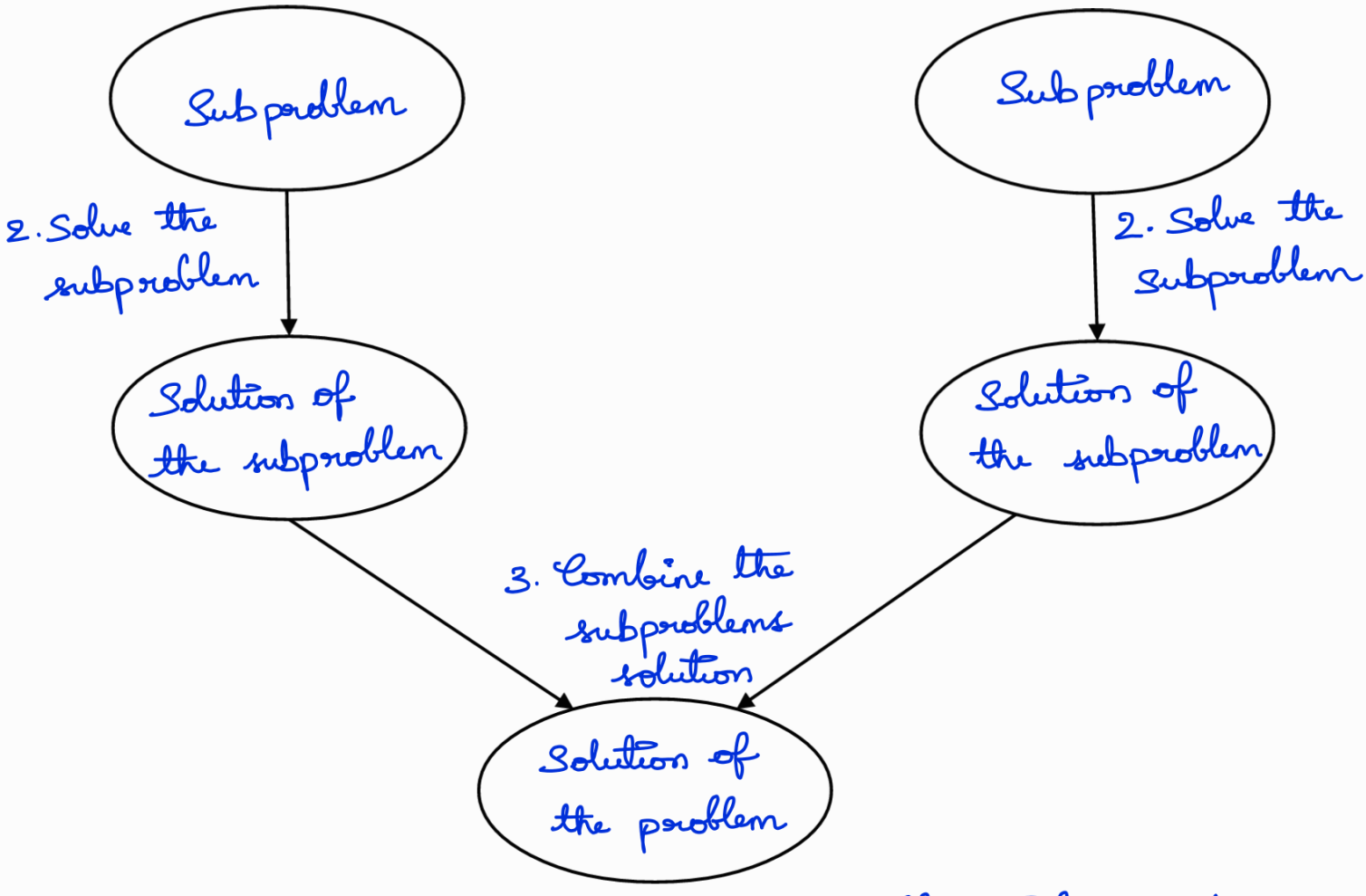
RECURSIVE ALGORITHM STRATEGIES

---

* **Divide & Conquer:-**

recursive algorithm design technique that involves breaking down a problem into smaller subproblems, solving each subproblem independently, and combining the solutions to obtain the final result.





(1) Divide: Break the original problem into smaller, more manageable subproblems. This step can often be represented by recursively calling the same algorithm on the subproblems.

(2) Conquer: Solve the subproblems independently. If the subproblems are small enough, they can be solved directly using a base case or a simple algorithm.

(3) Combine: Combine the subproblems' solution to obtain the final solution to the original problem. This step may involve merging or aggregating the subproblem solutions or applying additional operations to obtain the desired result.

Problems solved by Divide & Conquer strategy:

* Merge Sort
* Quick Sort

* Binary Search
* Strassen's Matrix Multiplication
* Closest Pair

Note:- One should prefer Divide & Conquer strategy when problem size significantly reduces with each division and combining sub-problems can solve larger problems.

* Dynamic Programming:-

A technique used to solve complex problems by breaking them down into overlapping subproblems and solving each subproblem only once. It is often used to optimize recursive algorithms by storing and reusing the results of subproblems.

DP follows these steps:-

1. Identify the problem that can be divided into smaller, overlapping subproblems. This property is known as the optimal substructure property.
2. Define the recurrence relation, which expresses the solution to the problem in terms of solutions to its subproblems. The recurrence relation should be based on the optimal substructure property.
3. Determine the base cases, which are the simplest subproblems that can be solved directly without further decomposition.
4. Decide on the method of solving the subproblems. Dynamic programming can be implemented using either top-down approach (memoization) or a bottom-up approach (tabulation)

* Memoization:

In the top-down approach, the recursive algorithm is enhanced with a memoization technique. The results of subproblems are stored in a cache or a table to avoid redundant computations. Before solving a subproblem, the algorithm checks if its result is already available in cache. If so, the cached result is used instead of re-computing it.

* Tabulation:

In the bottom-up approach, the algorithm solves the subproblems iteratively, starting from the base cases and progressively building up the final solution. The results of subproblems are stored in a table or an array, and each subproblem is solved only once.

(5) Determine the order of solving the subproblems. The order can vary depending on the problem, but it should ensure that all dependencies of a subproblem have been solved before solving that subproblem.

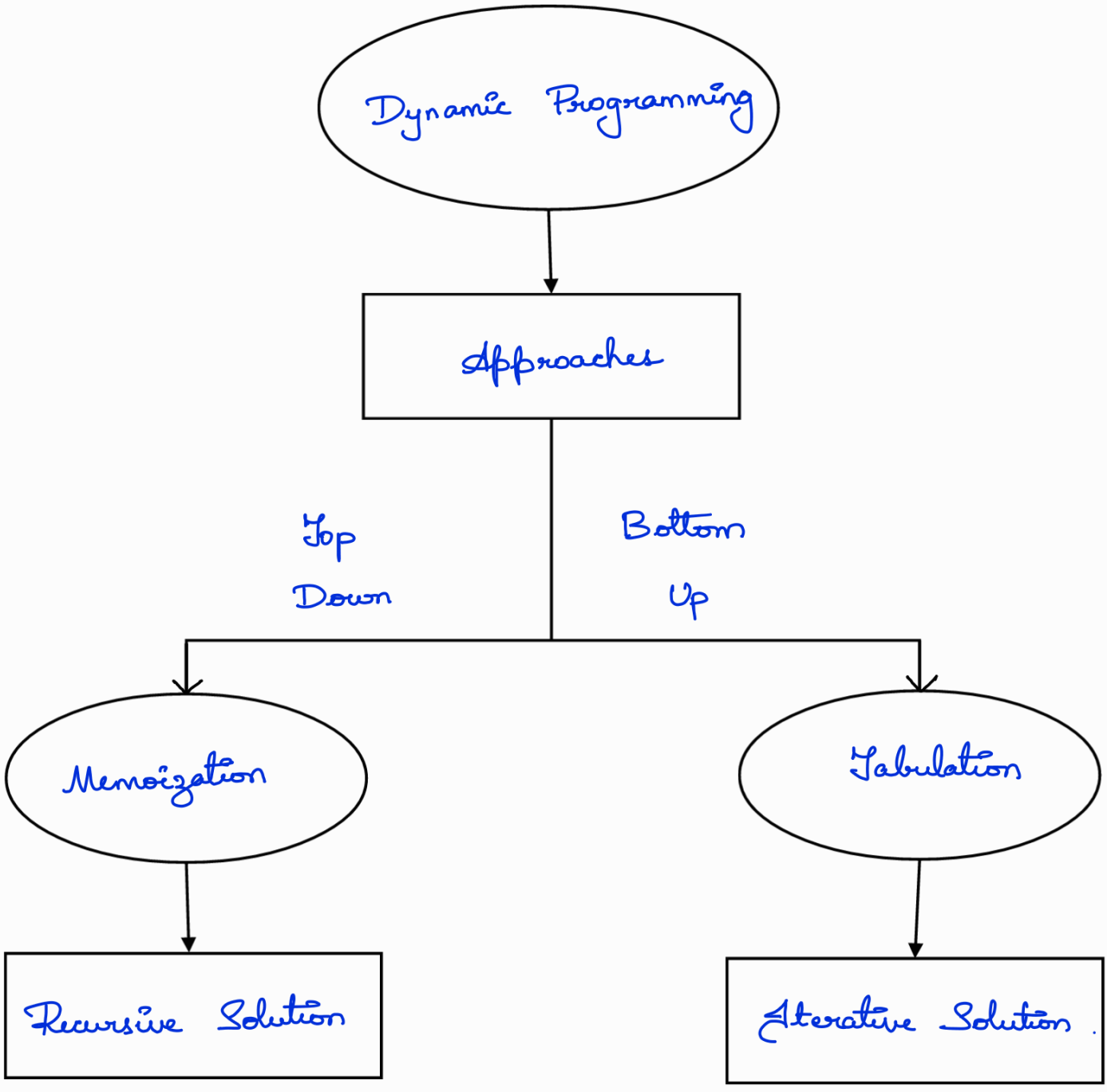
(6) Compute and store the results of subproblems based on the recurrence relation, either using memoization or tabulation.

(7) Finally, return the solution to the original problem, which is typically the result stored in the table or cache for the largest subproblem.

Commonly used in:

optimization, graph algorithms, sequence alignment etc.

By solving each subproblem only once and reusing the results, DP can significantly improve the efficiency of algorithms.



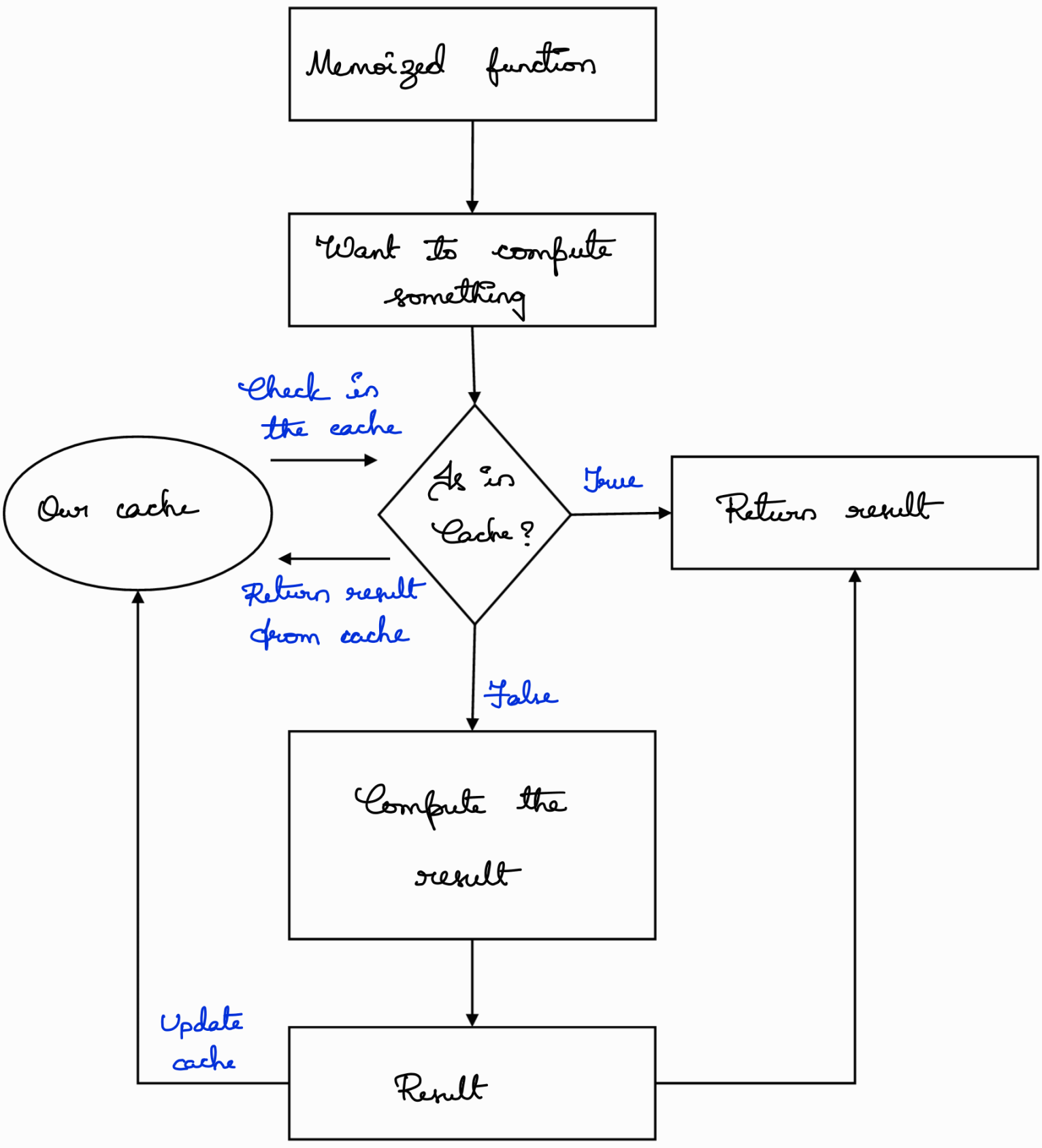
Memoization (top-down approach) :-

① Creating a data structure, typically an array or a hash map, to store the computed results. The keys of the datastructure are the inputs to the function, and the values are the corresponding outputs.

② Before making a recursive function call, check if the result for the given inputs is already present in the cache. If it is, return the cached result instead of performing computation again.

(3) If result is not in the cache, compute the result as usual and store it in the cache before returning it.

(4) With each subsequent function call, check the cache first. If result is found, return it directly, eliminating the need for redundant computations.



The memoization technique ensures that each unique set of inputs is computed only once. This becomes possible because the subsequent calls with the same inputs can directly fetch the cached result. Thereby, it leads to significant performance improvements in scenarios where the same function is called with the same inputs repeatedly.

```
function fibonacci (n, cache):
    if n is in cache:          → fetching result from cache
        return cache[n]          if present.

    if n equals 0:
        result = 0
    else if n equals 1:
        result = 1
    else:
        result = fibonacci(n-1) + fibonacci(n-2)
    cache[n] = result          → storing result in cache.
    return result
```

Note: In some cases, it may be necessary to limit the size of the cache or clear it when needed to avoid excessive memory consumption.

Tabulation (bottom-up approach):-

(1) Identify the subproblems: Analyze the problem and break it down into smaller overlapping subproblems. Each subproblem should have a well-defined input and output.

(2) Define a table: Create a table (usually an array or a matrix) to store the solutions to the subproblems. The table size is determined based on the input size and the number of subproblems.

(3) Initialize base cases: Fill in the table with the solutions to the base cases, which are the smallest subproblems that can be directly solved.

(4) Build solutions iteratively: Use the values from previously solved subproblems to solve larger subproblems. Iterate through the table in a bottom-up manner, solving each subproblem and storing its solution in the table.

(5) Return the final solution: Once all subproblems are solved, the solution to the original problem will be stored in the table at the appropriate position.

Bottom-up is often preferred when implementing dynamic programming algorithms because it eliminates the overhead of function calls and avoids issues with recursion depth. It also guarantees that all necessary subproblems are solved, as the solutions are computed in a systematic and incremental manner.

```
function fibonacci(n):

    fibTable = array of size n+1

    # Initialize base cases
    fibTable[0] = 0
    fibTable[1] = 1

    # Build solutions iteratively
    for i = 2 to n:
        fibTable[i] = fibTable[i-1] + fibTable[i-2]

    return fibTable[n]
```

computes base cases first

bottom - up

Complexity Analysis :-

1. Time Complexity Analysis for recursive algorithms.
2. Space Complexity Analysis for recursive algorithms.

ASYMPTOTIC TIME COMPLEXITY FOR RECURSIVE ALGORITHM

---

* Recursion tree method
* Substitution method
* Master theorem.

Prerequisite:
Recurrence relation.

* Recurrence relation :-

This is a way to mathematically capture the approximate behavior of a recursive algorithm. It comes handy to estimate asymptotic time complexity.

$$T(n) = aT\left(\frac{n}{b}\right) + F(n)$$

where,

$T(n)$: time complexity of algorithm for input size n.

a : number of subproblems that breaking a larger problem result in.

$\frac{n}{b}$: size of each subproblem that resulted in breaking a larger problem of size n.

$F(n)$: asymptotic time bound for dividing and then conquering/merging a problem of size n. Please note if n approaches base case, we say that $F(n)$ is bounded by $O(1)$.

Example:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

means:

(1) The recursive algorithm divides the larger problem into two subproblems.

(2) The size of each subproblem is the exact half the larger one.

(3) For dividing and then conquering problem of size n, the algorithm takes $O(n)$ time.

1) Recursion tree method :-

A technique used to analyze time complexity of recursive algorithms.

It involves representing the recursive calls of an algorithm as a tree structure, known as a recursion tree, where each node represents a recursive call.

Steps :-

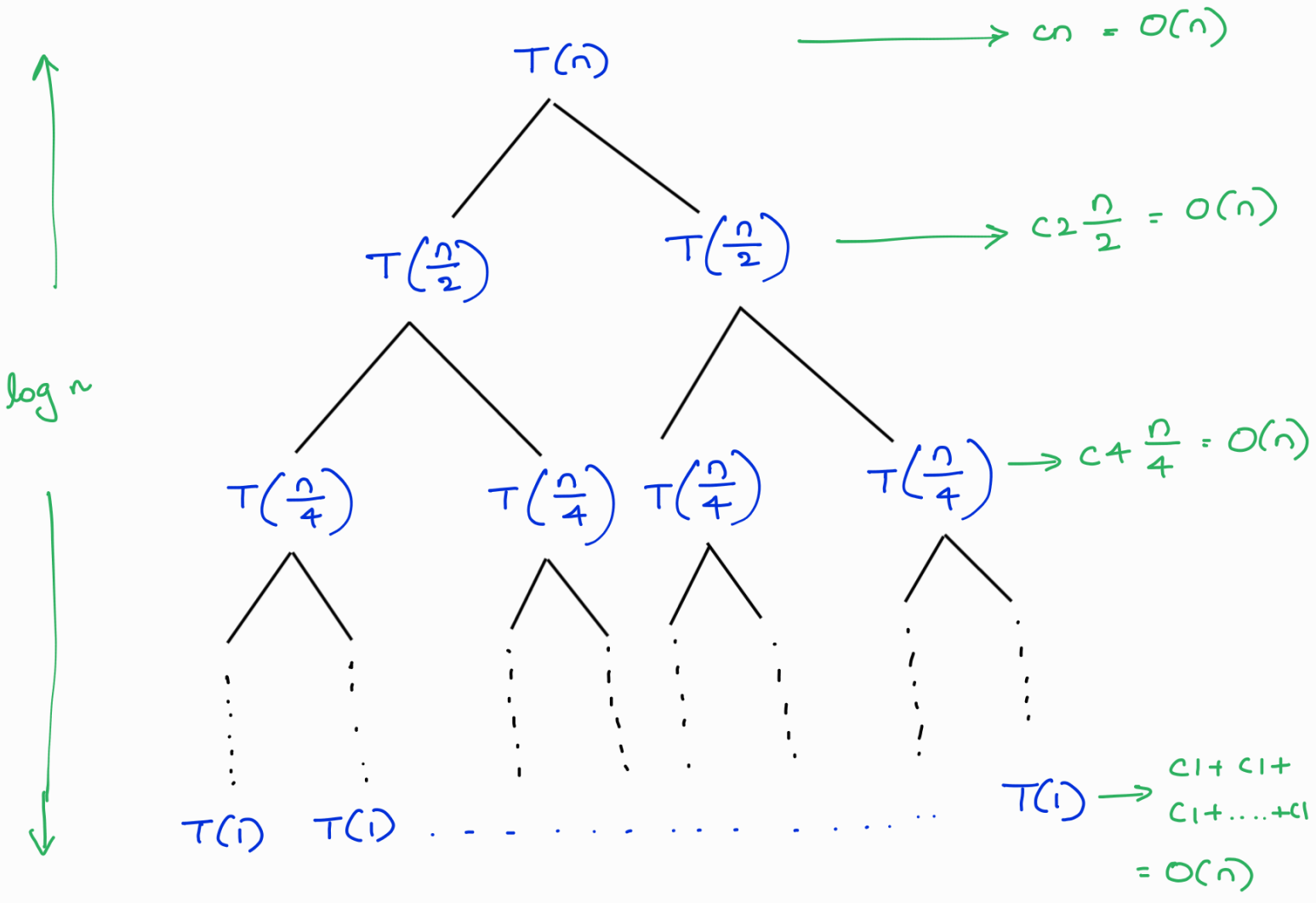
1. Start with root node representing initial call to the recursive procedure
2. For each recursive call, create child nodes that represent subsequent recursive calls made within that call. Repeat until base case of recursion is reached.
3. Assign a work or cost value to each node, representing the amount of work done in that recursive call.
4. Analyze the tree by calculating the total work done at each level of recursion. This can be done by summing up the work values of all the nodes at each level.
5. Determine the depth of the recursion tree, which represents the number of levels in the tree. This is often related to the input size & the number of recursive calls made.
6. Finally, analyze the total work done as a function of the recursion depth. This helps in determining the time complexity of the algorithm.

This method is particularly useful for algorithms with multiple recursive calls & when the time complexity is not immediately apparent.

Example :

A divide & conquer algorithm with recurrence equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$



Total work = $[O(n) + O(n) + O(n) + \ldots + O(n)]$ ($\log n$ times)

$= O(n \log n)$

fig., Recursion Tree.

2) Substitution method :-

A technique used to analyze the time complexity of algorithms by making educated guesses about the time complexity and then proving them using mathematical induction.

1. make educated guess about time complexity of the algorithm
2. assume guess is correct and solve recurrence relation using mathematical induction.

Base case:

Verify that the guess holds for the recurrence relation's base case(s). Typically, this involves checking the time complexity for small input sizes.

Inductive hypothesis:

Assume that the guess holds for all input sizes smaller than n.

Inductive step:

Using assumption from the inductive hypothesis, prove that the guess holds for the input size n.

3. Once the recurrence relation is solved, you obtain a closed-form solution, which represents the algorithm's time complexity.

Example:

```
mergeSort(arr[], low, high)
    if low < high
```

```
mid = (low + high)/2
mergeSort(arr, low, mid)
mergeSort(arr, mid+1, high)
merge(arr, low, mid, high)
```

Recurrence Relation of above pseudocode:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Educated guess: $O(n \log n)$

To prove this, use mathematical induction:

① Base case: For small input sizes, algorithm performs a constant amount of work.

② Inductive hypothesis:

Assume that the guess holds for arrays of size smaller than n.

③ Inductive step:

Assuming the guess holds for arrays of size $n/2$, we can express the time complexity of the algorithm as:

$$T(n) = 2\,T\left(\frac{n}{2}\right) + O(n)$$

By the inductive hypothesis, we substitute $T(n/2)$ with the assumed time complexity:

$$T(n) = 2\left[\frac{n}{2}\log\left(\frac{n}{2}\right)\right] + O(n)$$

Simplifying further, we have:

$$T(n) = n\log\left(\frac{n}{2}\right) + O(n) = n\log(n) - n\log(2) + O(n)$$

$$= n\log n - n + O(n)$$

$$= O(n\log n)$$

Thus we have proved that time complexity of Merge Sort is $O(n\log n)$

3) Master theorem method :-

Mathematical tool to analyze and solve certain types of recurrence relations.

The master theorem is typically used for recurrence relations of the following form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where,

$T(n)$: represents the time complexity (or recurrence) of an algorithm or function for a problem size of n.

$a \geq 1$ and $b > 1$: constants that define the number of recursive subproblems and the size reduction factor.

$f(n)$: represents non-recursive part of the time complexity, which includes any additional work done outside of the recursive calls.

The master theorem has three cases based on the relationship between the recursive and non-recursive parts:

Case 1: If $f(n) = O(n^c)$ for some constant $c < \log_b(a)$, then the time complexity can be expressed as:

$$T(n) = \Theta(n^{\log_b(a)}).$$

Case 2: If $f(n) = \Theta(n^c \log(n))$ for some constants $c \geq 0$, and if a recursive subproblem dominates the non-recursive part, i.e., $aT\left(\frac{n}{b}\right) \geq f(n)$ for sufficiently large n, then the time complexity can be expressed as:

$$T(n) = \Theta\left(n^{\log_b(a)}\right) \log(n)$$

Case 3: If $f(n) = \Omega(n^c)$ for some constant $c > \log_b(a)$, and if the non-recursive part dominates the recursive subproblems, i.e., $aT\left(\frac{n}{b}\right) \leq f(n)$ for sufficiently large n, then the time complexity can be expressed as:

$$T(n) = \Theta(f(n)).$$

To apply master theorem, you must identify which case applies to your recurrence relation by comparing the growth rates of $f(n)$ and $n^{\log_b(a)}$.

Example 1: $T(n) = 9T\left(\frac{n}{3}\right) + n^2$

$a = 9 \qquad b = 3 \qquad f(n) = n^2$

To apply the master theorem, we compare the growth rate of $f(n) = n^2$ with $n^{\log_b(a)} = n^{\log_3 9} = n^2$

Since $f(n) = n^2$ matches the growth rate of $n^{\log_b^{(a)}} = n^2$, we are in Case 1 of master theorem.

$\therefore$ Time complexity $T(n) = \Theta\left(n^{\log_b^{(a)}}\right) = \underline{\Theta(n^2)}$.

Example 2: $T(n) = 2T\left(\frac{n}{2}\right) + n^3$

$a = 2, \quad b = 2 \quad f(n) = n^3. \quad n^{\log_2^2} = n.$

$\because f(n)$ grows faster than $n^{\log_b^{(a)}}$ we are in Case 3.

$\therefore$ Time complexity $T(n) = \Theta(f(n)) = \underline{\Theta(n^3)}$.

Example 3: $T(n) = 8T\left(\frac{n}{2}\right) + n^2 \log n$

$a = 8 \quad b = 2 \quad f(n) = n^2 \log n \quad n^{\log_2^8} = n^3$

$\because n^{\log_b^{(a)}}$ dominates $f(n)$ we are in Case 2.

$\therefore$ Time complexity $T(n) = \underline{\Theta(n^3) \log(n)}$.

SPACE COMPLEXITY ANALYSIS FOR RECURSIVE ALGORITHMS.

---

* When analysing space complexity, consider:

① Recursive stack space.

Recursive algorithms typically rely on function calls that create stack frames to store local variables and the return address. Each function call adds a new

stack frame to the call stack, which consumes space. The depth of the recursive calls determines the maximum number of stack frames needed at any given time.

② Additional data structure.

Recursive algorithms may create additional data structures or arrays during their execution. The space required by these data structures can contribute to the overall space complexity.

* Steps :-

① Identify additional space used by each recursive call.

– space required for each stack frame includes :

* local variables
* parameters
* return addresses.

stored in the stack frame.

② Determine the number of recursive calls made based on input size. Helps in determining max depth of recursive calls and consequently, max number of stack frames needed.

③ Calculate space complexity

= [(additional space of each recursion call) * (max no. of recursive calls)] + additional space by extra datastructures if present.

Space complexity analysis does not include any space taken by input.

Some recursive algorithms may be optimized to reduce space usage. Techniques like – tail recursion, memoization, or iterative versions of algorithm can often be employed to reduce or eliminate the need for additional stack frames or data structures, resulting in improved space efficiency.

Example:- Space complexity of factorial(n)

```
function factorial(n):
    if n<=1:
        return 1
    else:
        return n * factorial(n-1)
```

Step(1): additional space used by each recursive call. →O(1) because only local variables and return address.

Step(2): number of recursive calls n-1.

Step(3): Calculate space complexity O(n)

Space complexity of factorial is linear.