Linked List in C++ (Class-based)

# Introduction

A Linked List is a linear data structure where elements are not stored at contiguous memory locations. Instead, the elements are linked using pointers.   
In a Linked List, each element (called a "node") contains two parts:   
1. Data part  
2. Pointer to the next node.  
  
Linked lists can be singly linked, doubly linked, or circular depending on how the nodes are connected. This document demonstrates the implementation of Linked Lists in C++ using classes.

## Types of Linked Lists

### 1. Singly Linked List

In a singly linked list, each node points to the next node in the sequence. The last node in the list points to NULL.

Singly Linked List Structure:

class Node {  
public:  
 int data;  
 Node\* next;  
 Node(int val) {   
 data = val;   
 next = NULL;   
 }  
};

### 2. Doubly Linked List

In a doubly linked list, each node points to both the next node and the previous node, making it possible to traverse in both directions.

Doubly Linked List Structure:

class Node {  
public:  
 int data;  
 Node\* next;  
 Node\* prev;  
 Node(int val) {   
 data = val;   
 next = NULL;   
 prev = NULL;   
 }  
};

### 3. Circular Linked List

In a circular linked list, the last node points back to the first node. It can either be singly or doubly circular.

Circular Linked List Structure (Singly):

class Node {  
public:  
 int data;  
 Node\* next;  
 Node(int val) {   
 data = val;   
 next = NULL;   
 }  
};

## Basic Operations on Linked Lists

### 1. Insertion

Nodes can be inserted into a linked list in three possible locations:   
1. At the beginning  
2. In the middle (after a given node)  
3. At the end

Example code for inserting at the beginning:

class LinkedList {  
public:  
 Node\* head;  
 LinkedList() { head = NULL; }  
  
 void insertAtBeginning(int newData) {  
 Node\* newNode = new Node(newData);  
 newNode->next = head;  
 head = newNode;  
 }  
};

### 2. Deletion

A node can be deleted from:  
1. The beginning  
2. The middle (a specific node)  
3. The end

Example code for deleting a node:

void deleteNode(int key) {  
 Node\* temp = head;  
 Node\* prev = NULL;  
  
 if (temp != NULL && temp->data == key) {  
 head = temp->next;  
 delete temp;  
 return;  
 }  
  
 while (temp != NULL && temp->data != key) {  
 prev = temp;  
 temp = temp->next;  
 }  
  
 if (temp == NULL) return;  
  
 prev->next = temp->next;  
 delete temp;  
}

### 3. Traversal

To print the data in a linked list, we can traverse through each node, starting from the head node, and move towards the last node.

Example code for traversal:

void printList() {  
 Node\* node = head;  
 while (node != NULL) {  
 cout << node->data << " ";  
 node = node->next;  
 }  
}

### 4. Reversing a Linked List

Reversing a linked list involves changing the next pointers of the nodes so that the direction of traversal is reversed.

Example code for reversing a linked list:

Node\* reverse() {  
 Node\* prev = NULL;  
 Node\* current = head;  
 Node\* next = NULL;  
  
 while (current != NULL) {  
 next = current->next;  
 current->next = prev;  
 prev = current;  
 current = next;  
 }  
 head = prev;  
 return head;  
}

# Conclusion

Linked lists implemented using classes offer a more object-oriented approach to managing dynamic memory. With this method, data encapsulation and reuse are improved.   
Although linked lists provide dynamic memory allocation, they have limitations such as slower access time compared to arrays due to the lack of direct indexing.

Recursion

**Defination :** When a function calls itself until a special condition is met. Recursion works like stack.

Question : Find the factorial of a number.

int fact(int n){

if(n==0) return 1;

int partialAns = fact(n-1);

return n\*partialAns;

}

**How does recursion work**

* **Function Call Stack**: Recursion relies on the function call stack to manage the sequence of recursive calls. Each recursive call adds a new frame to the stack, and when a base case is met, the functions begin returning their results, and the stack frames are popped off one by one.

**Example: Factorial Function**

The factorial of a number n (denoted as n!) is defined as the product of all positive integers less than or equal to n. For example, 5! = 5 \* 4 \* 3 \* 2 \* 1 = 120.

**int factorial(int n) {**

**// Base case: if n is 0 or 1, return 1**

**if (n == 0 || n == 1) {**

**return 1;**

**}**

**// Recursive case: n \* factorial of (n - 1)**

**return n \* factorial(n - 1);**

**}**

**Breakdown of Execution:**

If we call factorial(5), the following steps occur:

1. factorial(5) is called:
   * 5 \* factorial(4) is computed.
2. factorial(4) is called:
   * 4 \* factorial(3) is computed.
3. factorial(3) is called:
   * 3 \* factorial(2) is computed.
4. factorial(2) is called:
   * 2 \* factorial(1) is computed.
5. factorial(1) is called:
   * The base case is met, so 1 is returned.

Now, the recursion starts **unwinding**, and the results are returned:

1. factorial(2) returns 2 \* 1 = 2.
2. factorial(3) returns 3 \* 2 = 6.
3. factorial(4) returns 4 \* 6 = 24.
4. factorial(5) returns 5 \* 24 = 120.

Thus, factorial(5) returns 120.

**A function call does take up space in memory**. When a function is called, the system allocates memory to keep track of the execution of that function. This memory is used to store important information such as:

1. **Function parameters**: The values passed to the function.
2. **Local variables**: Any variables defined within the function.
3. **Return address**: The point in the program where control should return once the function finishes execution.
4. **Saved state**: The state of the calling function, such as the values of registers or the instruction pointer.

This information is stored in a data structure known as the **call stack**.

**Function Call Stack**

The function call stack (or simply **call stack**) is a special data structure used by the operating system and programming environment to manage function calls. It works in a **Last-In, First-Out (LIFO)** manner, meaning the most recent function call is the first to complete. Here's how it works:

1. **Pushing to the stack**: When a function is called, a new stack frame (or activation record) is created for that function, which stores the function’s local variables, parameters, and return address. This frame is pushed onto the top of the call stack.
2. **Popping from the stack**: When the function finishes executing, the function’s stack frame is popped off the stack, and the control returns to the calling function. The memory allocated for the function is then released.

**Stack Frame**

A **stack frame** is created for each function call and contains:

* **Local variables and arguments**: Space for variables used in the function.
* **Return address**: The address of the next instruction to execute after the function completes.
* **Saved registers**: CPU registers that need to be restored once the function returns.

**Example**

void A() {

int x = 10;

B(); // Function call to B

}

void B() {

int y = 20;

// Do something

}

int main() {

A(); // Function call to A

}

* When main() calls A(), a stack frame for A is pushed onto the stack.
* Inside A(), a call to B() pushes another stack frame for B onto the stack.
* Once B() finishes, its stack frame is popped, and control returns to A().
* When A() completes, its stack frame is popped, returning control to main().

If a program uses recursive function calls, the stack can grow quite large, potentially leading to **stack overflow** if the recursion is too deep.

In summary, function calls do take memory, and the **function call stack** is the mechanism that keeps track of function calls, their parameters, local variables, and return addresses during program execution.

**Recursion Tree**

**Time complexity : number of node in the recursion tree; Space complexity : height of the recursion tree.**

A **recursion tree** is a visual representation of the recursive calls made during the execution of a recursive algorithm. It is useful for understanding and analyzing the behavior of recursive algorithms, especially when determining the time complexity.

Each node in the recursion tree represents a recursive call, and the children of a node represent the recursive calls made by that function. The root of the tree represents the initial call, and the depth of the tree represents the number of recursive levels.

**How to Build a Recursion Tree:**

1. **Root Node**: This represents the original problem and the first recursive call.
2. **Child Nodes**: These represent the subproblems generated by the recursive calls. Each recursive function call typically divides the problem into smaller subproblems, represented by the children.
3. **Continue Recursion**: The tree expands as the recursive calls break down the problem further.
4. **Base Case**: At the leaves of the tree, you have the base cases, where no further recursive calls are made, and the recursion terminates.

**Example: Recursion Tree for a Simple Algorithm**

Consider the following recursive function for calculating the Fibonacci number:

int fibonacci(int n) {

if (n <= 1) {

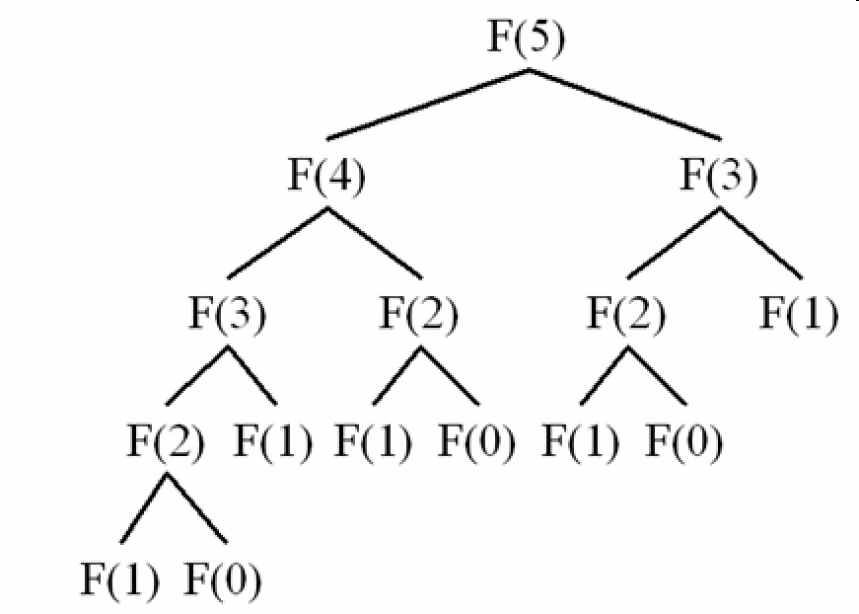
return n;

}

return fibonacci(n - 1) + fibonacci(n - 2);

}

**For fibonacci(4), the recursion tree looks like this:**

****

In this tree:

* The root node represents the call to fibonacci(4).
* The child nodes represent further recursive calls to fibonacci(3) and fibonacci(2).
* The tree continues to grow as the function makes more recursive calls.
* The leaves of the tree are the base cases (fibonacci(0) and fibonacci(1)).

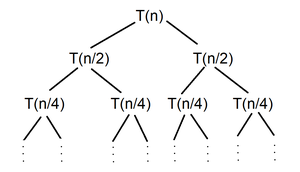
**Use of Recursion Trees:**

1. **Time Complexity Analysis**: By visualizing how many recursive calls are made and their relationships, a recursion tree helps analyze the time complexity. For example, in the Fibonacci example above, we can see that many calls are duplicated, leading to exponential time complexity (O(2^n)).
2. **Understanding Recursive Flow**: Recursion trees help break down a complex recursive function into smaller, manageable parts to understand the recursive flow and behavior better.

**Recursion Tree for Divide-and-Conquer:**

In divide-and-conquer algorithms like merge sort or binary search, a recursion tree is very useful for analyzing how the problem is broken down.

For example, in **merge sort**, the problem of sorting an array of size n is divided into two smaller arrays of size n/2:



This kind of recursion tree helps analyze the algorithm’s **logarithmic depth** and linear work at each level, leading to an overall time complexity of **O(n log n)**.

**Conclusion:**

A recursion tree is a powerful tool for visualizing and understanding the flow of recursive algorithms and for analyzing their time complexity and behavior.

**Find Power of a Number**

**Method 1.**

long long Pow(int x, int n){

    if(n==0) return 1;

    long long partialAns = (Pow(x, n-1));

    return x\*partialAns;

}

**Method 2.**

long long Pow(int x, int n){

    if(n == 1) return x;

    if(n==0) return 1;

    return Pow(x, n/2) \* Pow(x, n-(n/2));

}

**Method 3**.

long long Pow(int x, int n){

    if(n==0) return 1;

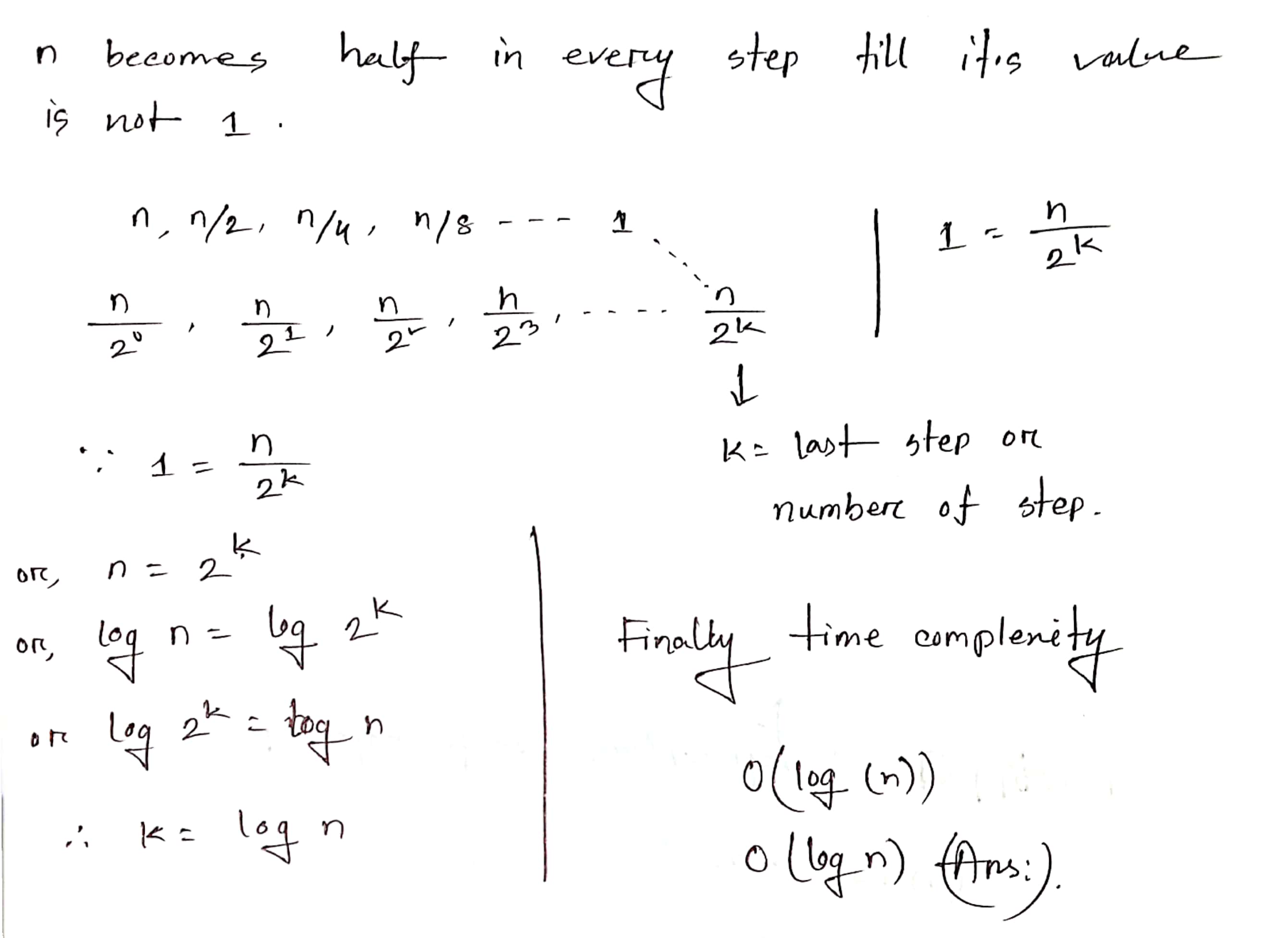
    long long partialAns = Pow(x, n/2);

    if(n&1) return partialAns\*partialAns\*x;

    return partialAns\*partialAns;

}

**Time Complexity of Method3**



**Question:** You're given an alphabetical string ‘S’.Determine whether it is palindrome or not. A palindrome is a string that is equal to itself upon reversing it.

For example:

‘S’ = racecar

The reverse of ‘S’ is: racecar. Since ‘S’ is equal to its reverse. So ‘S’ is a palindrome.Hence output will be 1.

bool palindromeHelper(string &s, int l, int r){

    if(l >= r) return true;

    if(s[l] != s[r]) return false;

    return palindromeHelper(s, ++l, --r);

}

bool isPalindrome(string &s){

    return palindromeHelper(s, 0, s.size()-1);

}

**4 Steps to Solve any recursion problem**

**1. Identify the Problem and the Recursive Case**

* **What to do:** Determine the task or problem that can be broken down into smaller subproblems.
* **Key Questions:** Can the problem be simplified into similar subproblems? What is the relationship between the larger problem and smaller instances of the problem? Like factorial(n) is the large problem and factorial(n-1) is the sub problem.

**2. Do Small Work (Work for the Current Call)**

* **What to do:** Perform the smallest unit of work that can be done without recursion. This typically involves processing part of the input or calculating something based on the current state.
* **Key Questions:** What is the smallest meaningful unit of work? What information do you need to pass to the next recursive call?

**3. Ask Recursion to Do the Remaining Work**

* **What to do:** Call the function recursively to handle the smaller version of the problem.
* **Key Questions:** What part of the work should recursion handle? How do you simplify the problem in each step? What parameters are you passing to the next recursive call?

**4. Base Condition (Stopping Condition)**

* **What to do:** Specify the condition under which recursion will stop. Without this, the function will continue indefinitely.
* **Key Questions:** When is the problem small enough to not require further recursion? What is the base case that will return a result without making a further recursive call?

This framework applies well to many recursive problems, such as factorial calculation, tree traversal, or solving linked list problems. Would you like to explore a specific example of recursion to see how these steps apply?

**Generating subsets:**

vector<vector<int>> subsets;

vector<int> a;

int n = 3;

void search(int k){

if(k==n){

subsets.push\_back(a);

return;

}

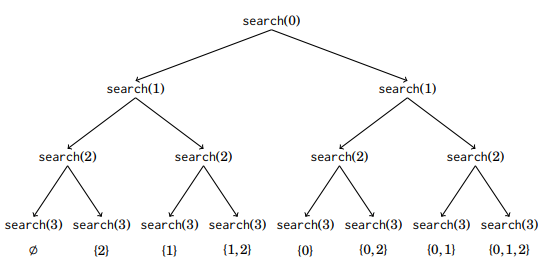
search(k+1);

a.push\_back(k);

search(k+1);

a.pop\_back();

}



**Note :** It’s also a backtracking way.

**Generating Permutation:**

int n = 3;

vector<bool> chosen;

vector<int> permutation;

vector<vector<int>> permutations;

void process\_permutation(){

permutations.push\_back(permutation);

}

void search(){

if(permutation.size() == n){

process\_permutation();

return;

}

for(int i=0; i<n; i++){

if(chosen[i]) continue;

chosen[i] = true;

permutation.push\_back(i);

search();

chosen[i] = false;

permutation.pop\_back();

}

}

void display\_permutations(){

for(auto p: permutations)

printa(p);

}

**Key Elements:**

* permutation: This is a list that stores the current permutation.
* n: The size of the set from which permutations are drawn.
* chosen[]: A boolean array used to mark which elements have already been included in the current permutation.

**How it works:**

1. **Base case**: When the size of the permutation equals n, it means a complete permutation has been formed. At this point, the function process\_permutation() is called to handle this complete permutation (e.g., printing or storing it).
2. **Recursive exploration**: For each element i from 0 to n-1, the algorithm checks whether i is already part of the current permutation (chosen[i]). If not, it adds i to the permutation, marks it as chosen, and calls search() recursively to continue building the permutation.
3. **Backtracking**: After exploring permutations with i in the current position, the algorithm "backtracks" by removing i from the permutation, unmarking it (chosen[i] = false), and trying the next element.

**Example Walkthrough:**

For n = 3, the function will explore all 6 permutations of the set {0, 1, 2}:

* 0 1 2
* 0 2 1
* 1 0 2
* 1 2 0
* 2 0 1
* 2 1 0

**[ ]**

**/ | \**

**/ | \**

**[0] [1] [2]**

**/ \ / \ / \**

**[0,1] [0,2] [1,0] [1,2] [2,0] [2,1]**

**| | | | | |**

**[0,1,2] [0,2,1] [1,0,2] [1,2,0] [2,0,1] [2,1,0]**

**N\_Queen Problem :**

void n\_queen(int row){

if(row==n){

rslt++;

return;

}

for(int col = 0; col<n; col++){

if(column[col] || diag1[col+row] || diag2[row-col+n-1]) continue;

column[col] = diag1[col+row] = diag2[row-col+n-1] = true;

n\_queen(row+1);

column[col] = diag1[col+row] = diag2[row-col+n-1] = false;

}

}

For More details : see the Competitive Programmer’s Handbook.

Permutation (Math)

**Introduction**

A permutation is a specific arrangement or ordering of objects. In other words, a permutation of a set

of objects is an arrangement of those objects in a specific sequence. In permutations the

arrangement (or order) is critical.

Key Concepts

**1. Factorial:**

The factorial of a number n is the product of all positive integers from 1 to n. It is denoted by n!.

For example:

4! = 4 × 3 × 2 × 1 = 24

Special case: 0! = 1.

**2. Definition of Permutation:**

A permutation of n distinct objects is an ordered arrangement of these objects. The number of

permutations of n objects taken r at a time is:

P(n, r) = n! / (n - r)!

Where n is the total number of distinct objects, and r is the number of objects you want to arrange.

Types of Permutations:

**1. Permutations of All Objects (Simple Permutations):**

If you are arranging all n objects, the number of permutations is simply n!.

For example, arranging 4 objects A, B, C, D:

P(4, 4) = 4! = 24.

**2. Permutations of Subsets (Partial Permutations):**

When you want to arrange r objects from a set of n distinct objects, use the formula:

P(n, r) = n! / (n - r)!

Example: Arranging 3 objects from 5 distinct objects:

P(5, 3) = 5! / 2! = 60.

**Permutation with Repetition:**

If repetition is allowed, the number of permutations is n^r.

Example: How many 4-letter words can be formed using A, B, C, with repetition allowed?

Answer: 3^4 = 81.

**Permutations of Non-Distinct Objects:**

When some objects in the set are identical, the formula becomes:

P = n! / (k1! \* k2! \* ... \* km!)

Where k1, k2,...,km are the frequencies of repeated objects.

Example: Distinct arrangements of "BANANA":

P = 6! / (3! \* 2!) = 60.

Applications:

Permutations have applications in cryptography, computer science, statistics, and games.

Summary:

Permutations focus on ordered arrangements. The formula for permutations is:

P(n, r) = n! / (n - r)!.

For non-distinct objects, use P = n! / (k1! \* k2! \* ... \* km!).

Start New Start