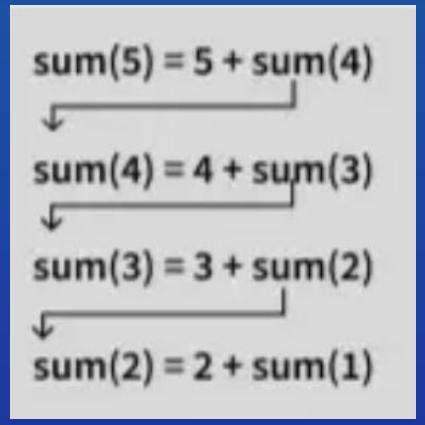
01

Recursion and Stack

Fatima Khan
Omer Shamsi
Areeba Zehra Jaffri
Faiq Ahmed

CONTENTS

Part 1
Recursion



Definition

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

Recursion breaks down problems into simpler sub-problems and hence, finds optimal solution for a problem.

Simplicity

Recursive solutions are often shorter and easier to understand (e.g., factorial, Fibonacci, tree traversals).

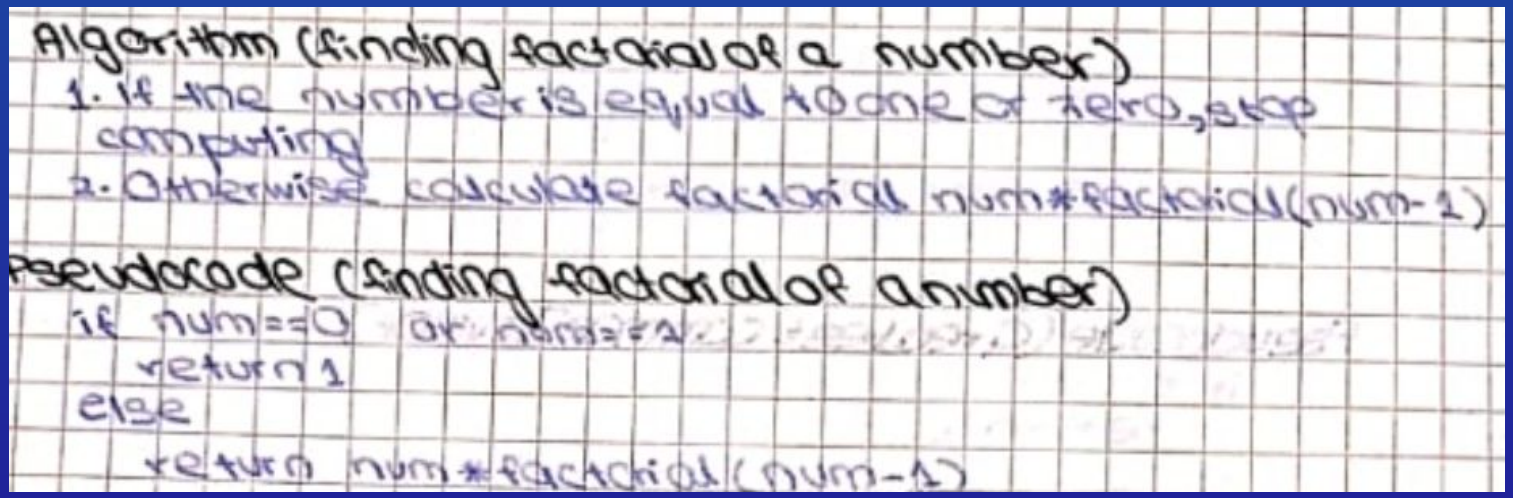
Reduced Code Size

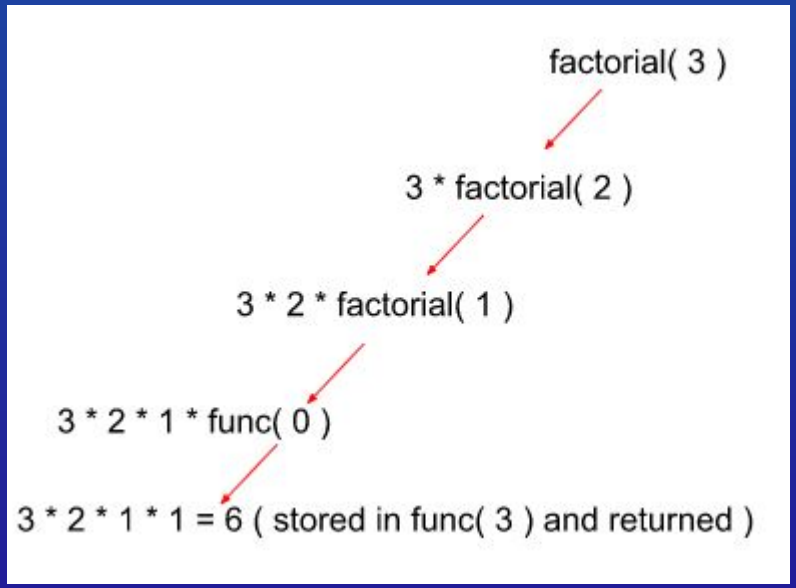
Recursion can replace loops and complex logic, making code more concise.

Higher memory usage

Takes up more memory, hence, its not space efficient.

You need to identify two cases to solve a recursive problem
Base Case: the case where your recursive call is stopped.
Recursive case: invokes the recursive call.





In this example, base case will be applied to (number==0 or number==1)
Recursive case will be applied to all values except 0 and 1.

Non tail recursion

Tail recursion is defined by having the recursive call as the last operation in the function before returning

```
// Tail recursive GCD function
int gcd(int a, int b) {
    if (b == 0)
        return a;             // base case
    return gcd(b, a % b);     // recursive call is the last operation
}
```

Tail recursion

Tail recursion is when other operations are invoked after the recursive call.

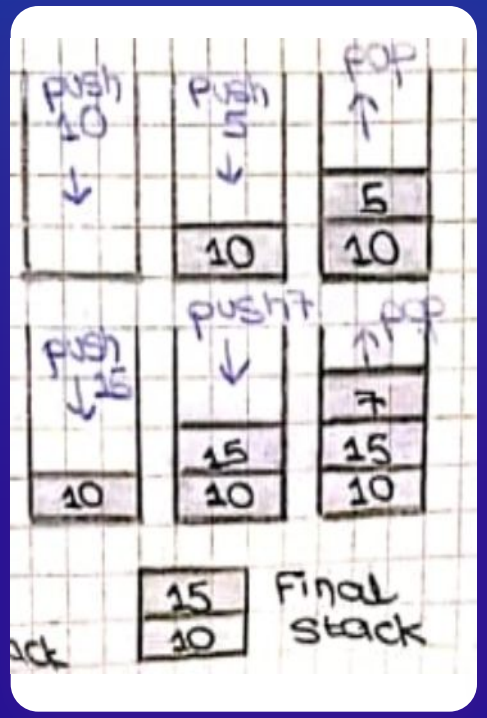
```
// Non-tail recursive factorial
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);    // recursive call is not the last
operation
}
```

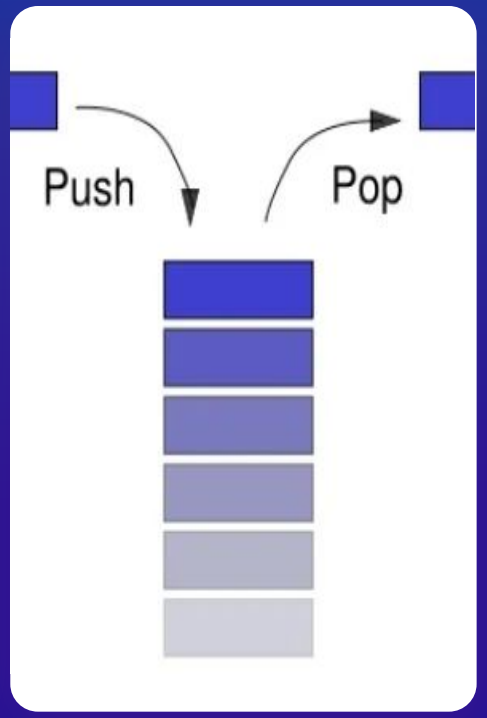
Stack

Stack is a linear data structure that can be accessed only at one of its ends for storing and retrieving data. It follows LIFO. Example a stack of plates.

Operations of stack

Clear() – To clear the stack
IsEmpty() – Check If Stack Is Empty
Push(El) – Pushing element on top of stack
Pop()- Remove from stack
TopEl() – Peek the top element






```
 void Clear() {
     top = -1;
     cout << "Stack cleared!" << endl;
 }
// IsEmpty() - Check If Stack Is Empty
bool IsEmpty() {
    return (top == -1);
}

// Push(El) - Pushing element on top of stack
void Push(int El) {
    if (top == capacity - 1) {
        cout << "Stack Overflow!" << endl;
    } else {
        arr[++top] = El;
        cout << El << " pushed into stack." << endl;
    }
}

// Pop() - Remove from stack
void Pop() {
    if (IsEmpty()) {
        cout << "Stack Underflow!" << endl;
    } else {
        cout << arr[top--] << " popped from stack." << endl;
    }
}
// TopEl() - Peek the top element
int TopEl() {
    if (IsEmpty()) {
        cout << "Stack is empty!" << endl; return -1;
    } else { return arr[top]; } }
```

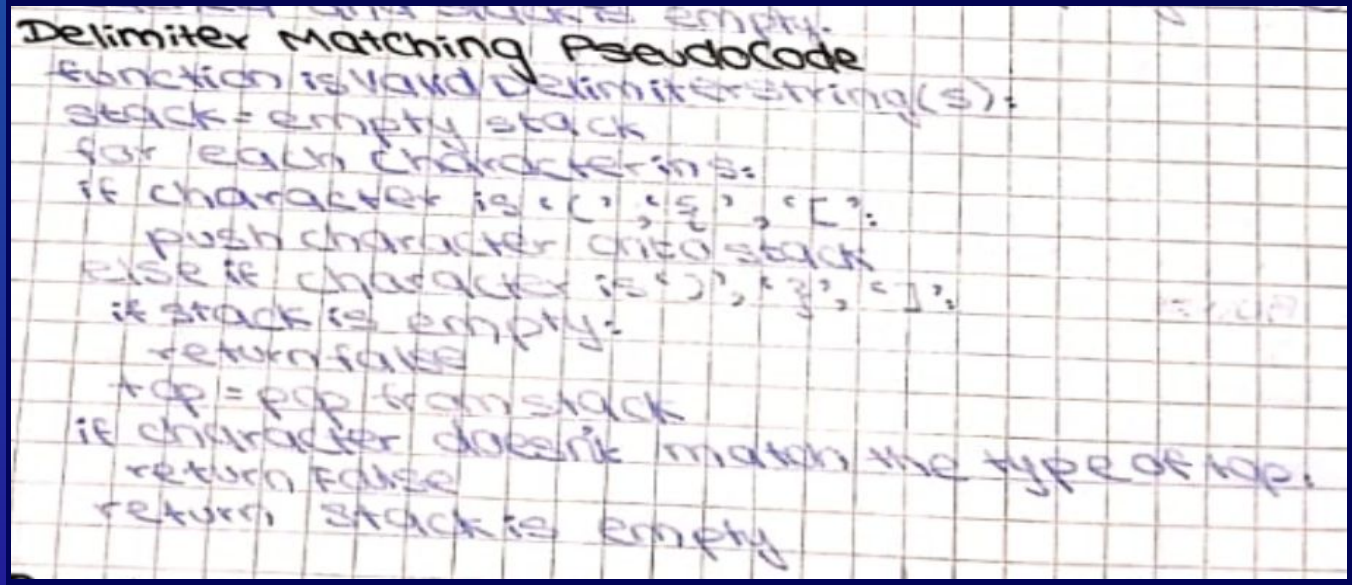
Applications Of stack

1. Delimiter Matching
2. Postfix Expression
3. Tower Of Hanoi

Delimiter Matching

- Stack helps solve complex mathematical equations

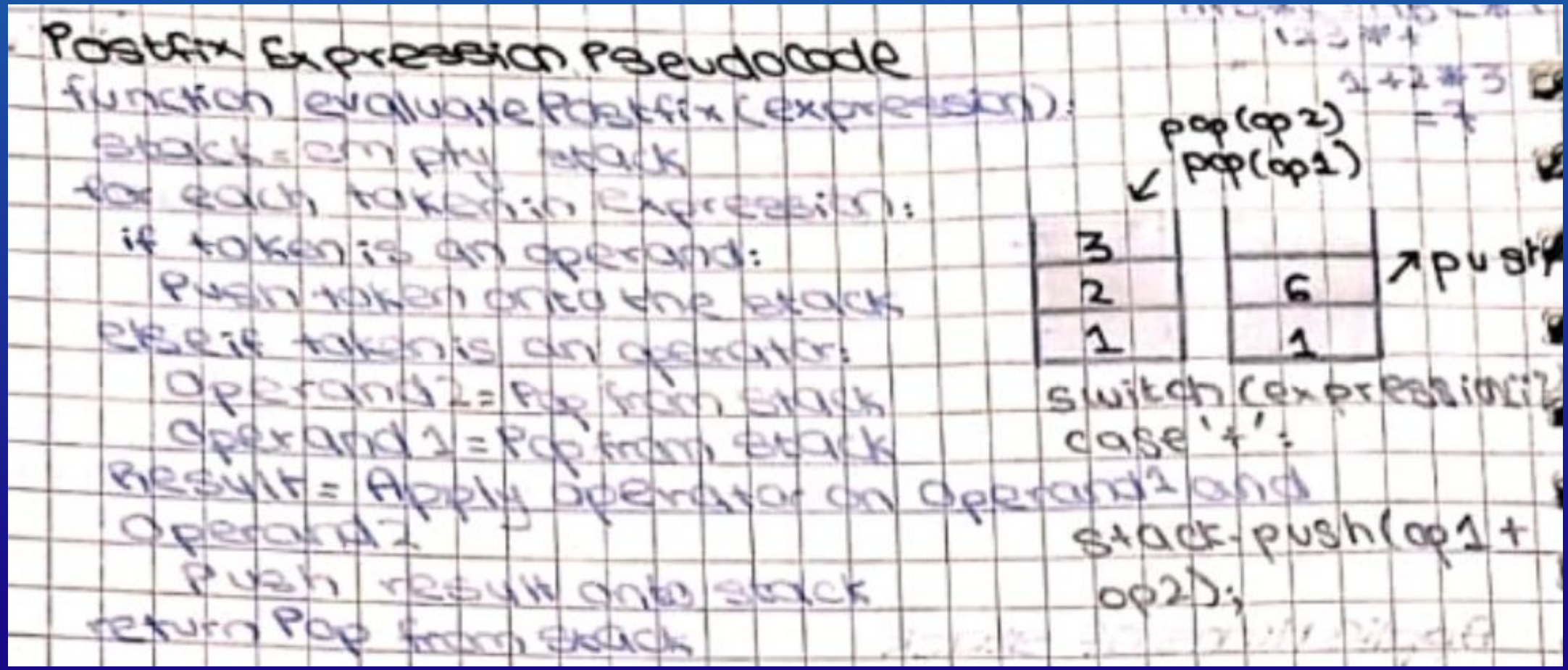
```
(a+b)*{c+[d-e]}
[{(x+y)*z}]
a*(b+c)
(a+b]*{c+d)      // mismatched
[(a+b)           // missing closing ]
a+b)*(c+d        // extra closing )
```



Postfix Expression

The **use of postfix expression** is mainly in **computer systems and compilers** because it makes evaluation easier and faster.

(5 + 3) * 2 will turn to 5 3 + 2 *

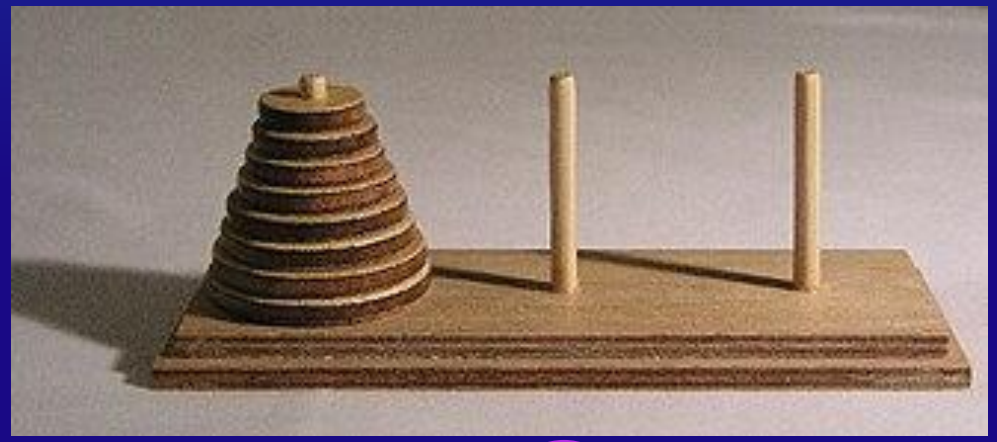


Operand: The values or variables on which an operation is performed. Example values: numbers (2, 5, 10) or variables (A, B, X).

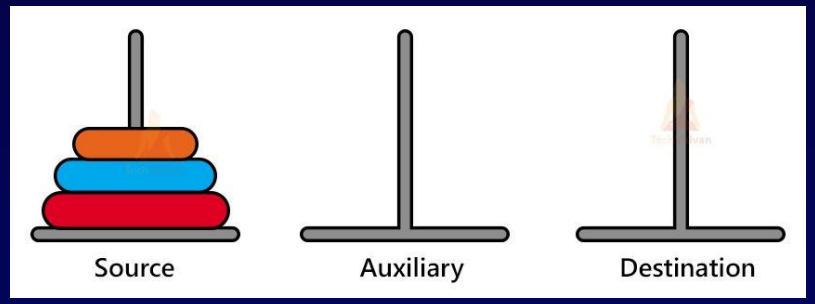
Operator: The symbol that tells what operation to perform on operands. Example operators:
Arithmetic: + , - , * , /
Relational: > , < , == , !=
Logical: && , || , !

Tower Of Hanoi

The tower of Hanoi is a classic problem in computer science that involves moving a set of sticks from one rod to another, following specific rules.

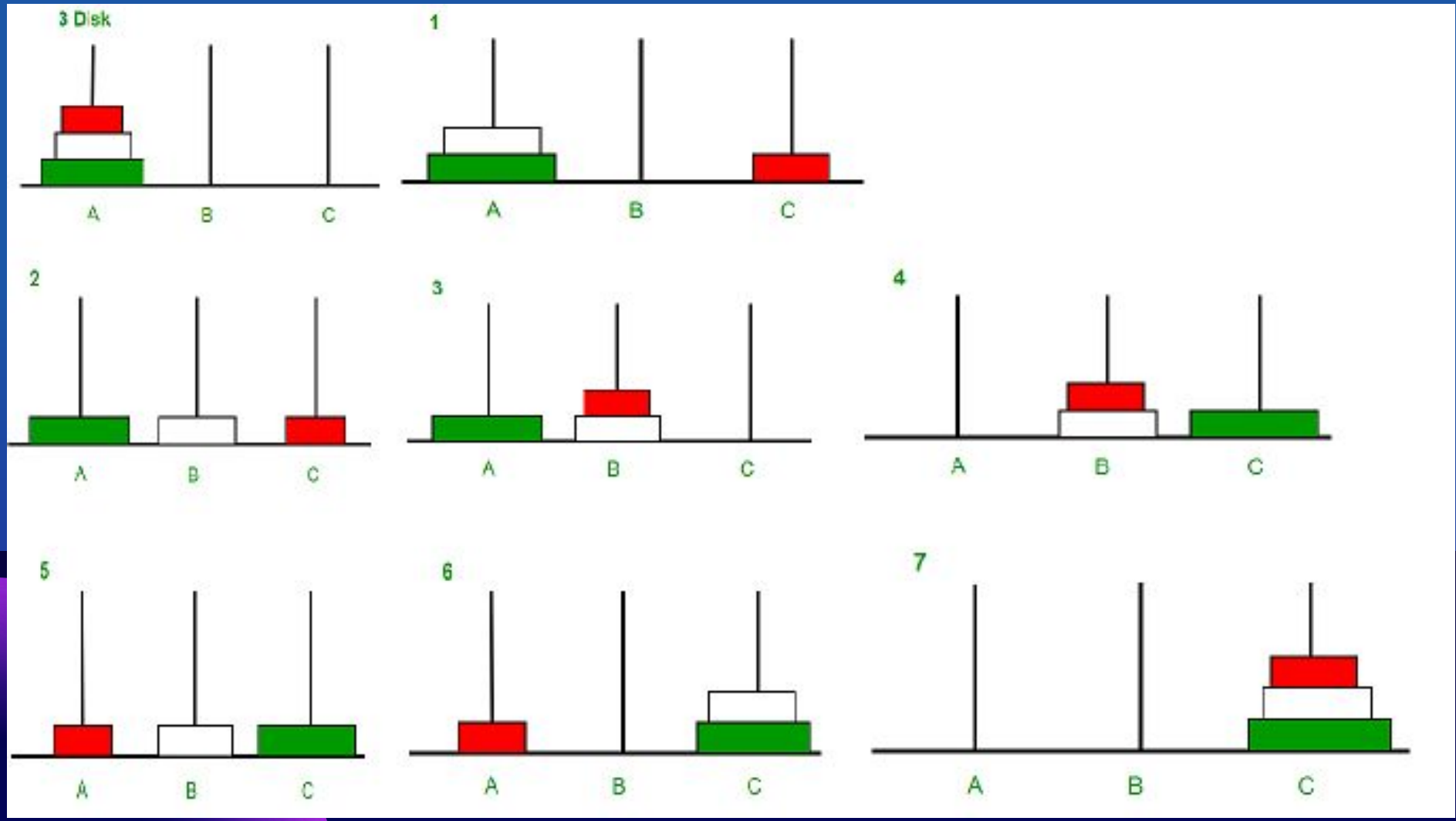


Rules Of Tower Of Hanoi



Constraints

1. Only one disk can be moved at a time
2. A disk can only be placed on top of a larger disk
3. All disks must be moved from source rod to destination rod


3 Disk
A
B
C
1
A
B
C
2
A
B
C
3
A
B
C
4
A
B
C
5
A
B
C
6
A
B
C
7
A
B
C

Leetcode Questions

921. Minimum Add to Make Parentheses Valid

A parentheses string is valid if and only if:

- It is the empty string,
- It can be written as `AB` (`A` concatenated with `B`), where `A` and `B` are valid strings, or
- It can be written as `(A)`, where `A` is a valid string.

You are given a parentheses string `s`. In one move, you can insert a parenthesis at any position of the string.

- For example, if `s = "()))"`, you can insert an opening parenthesis to be `"(()))"` or a closing parenthesis to be `"())))"`.

Return *the minimum number of moves required to make* `s` *valid.*

Example 1:

> **Input:** s = "())"
> **Output:** 1

Example 2:

> **Input:** s = "((("
> **Output:** 3

Constraints:

- `1 <= s.length <= 1000`
- `s[i]` is either `'('` or `')'`.

1021. Remove Outermost Parentheses

A valid parentheses string is either empty `""`, `"(" + A + ")"`, or `A + B`, where `A` and `B` are valid parentheses strings, and `+` represents string concatenation.

- For example, `""`, `"()"`, `"(())()"`, and `"(()(()))"` are all valid parentheses strings.

A valid parentheses string `s` is primitive if it is nonempty, and there does not exist a way to split it into `s = A + B`, with `A` and `B` nonempty valid parentheses strings.

Given a valid parentheses string `s`, consider its primitive decomposition: `s = P1 + P2 + ... + Pk`, where `Pi` are primitive valid parentheses strings.

Return `s` *after removing the outermost parentheses of every primitive string in the primitive decomposition of* `s`.

Example 1:

> **Input:** s = "(()())(())"
> **Output:** "()()()"
> **Explanation:**
> The input string is "(()())(())", with primitive decomposition "(()())" + "(())".
> After removing outer parentheses of each part, this is "()()" + "()" = "()()()".

Example 2:

> **Input:** s = "(()())(())(()(()))"
> **Output:** "()()()()(())"
> **Explanation:**
> The input string is "(()())(())(()(()))", with primitive decomposition "(()())" + "(())" + "(()(()))".
> After removing outer parentheses of each part, this is "()()" + "()" + "()(())" = "()()()()(())".

Constraints:

- $1 \leq s.length \leq 10^5$
- `s[i]` is either `'('` or `')'`.
- `s` is a valid parentheses string.

02

Thank you