# DAA

# Unit 1

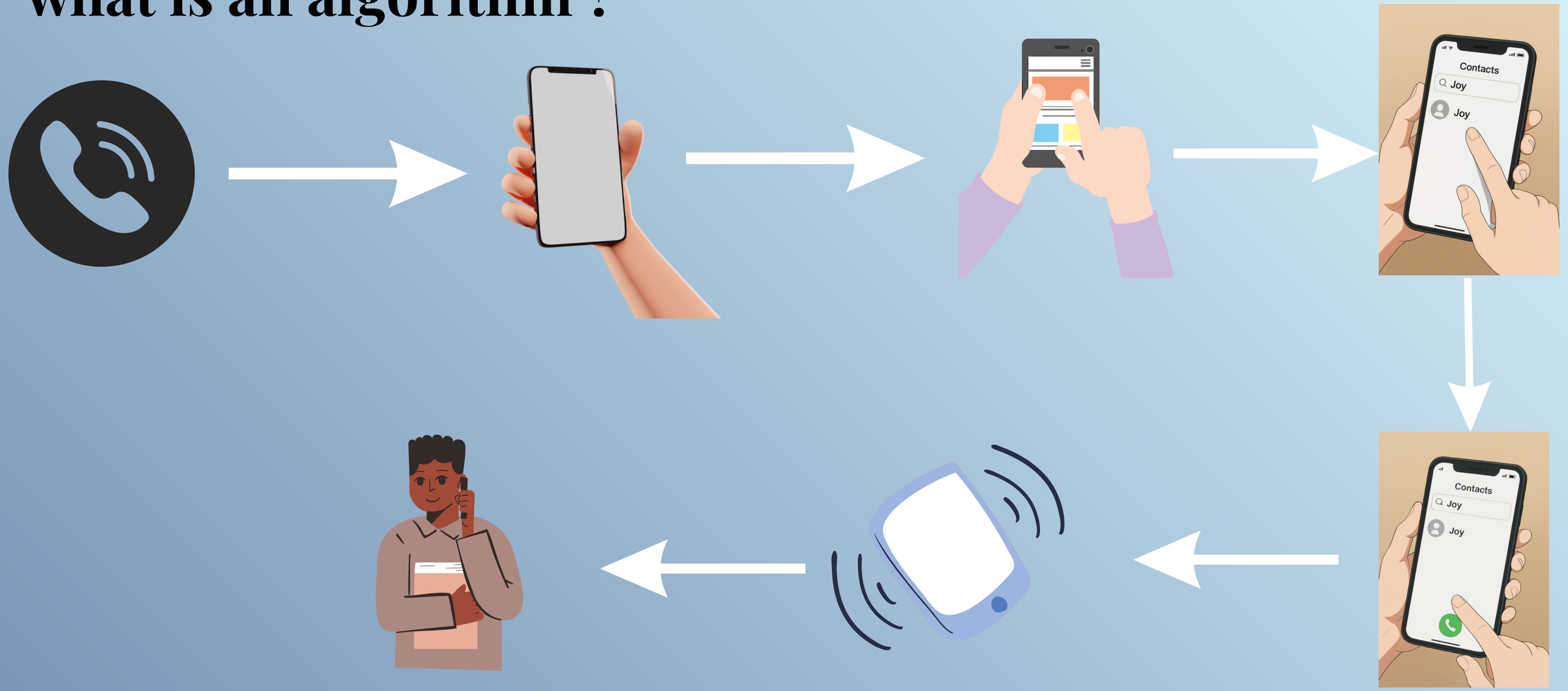
# Algorithms and Problem Solving

# Contents

- Algorithm
- Design and analysis of algorithm
- Need of an algorithm
- Correctness of an algorithm
- Invariant Loop
- Iterative Algorithm and its design issues
- Problem Solving Strategies

# Algorithm

## what is an algorithm ?



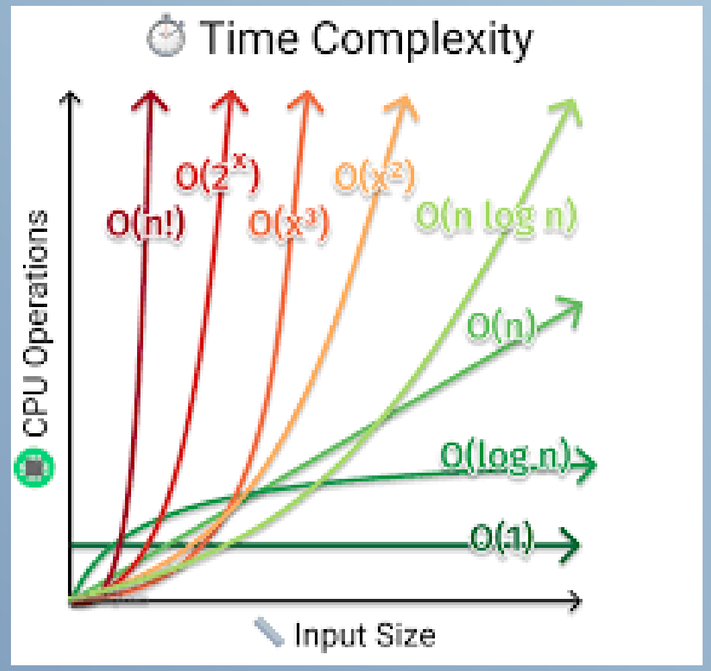
# Find Maximum element in List

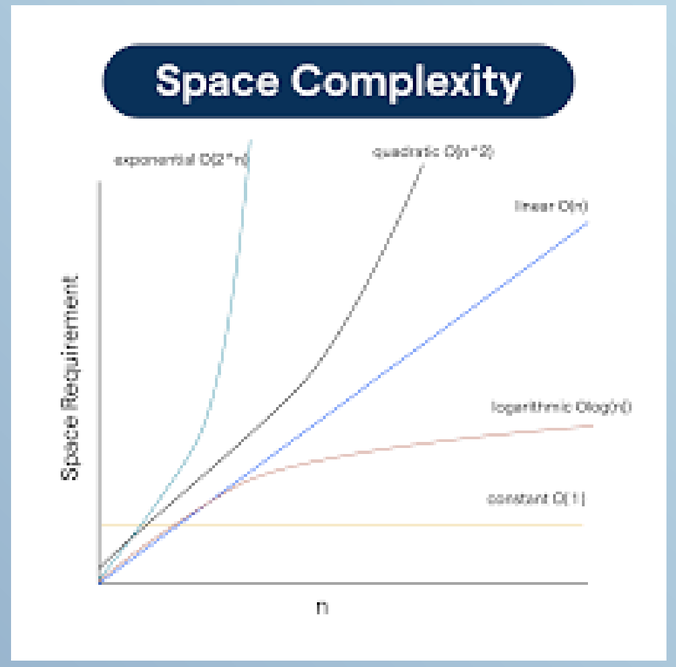
- Start
- Take the first element as maximum.
- Compare each next element with the maximum.
- If bigger, update maximum.
- After all comparisons, maximum is the largest element.
- End

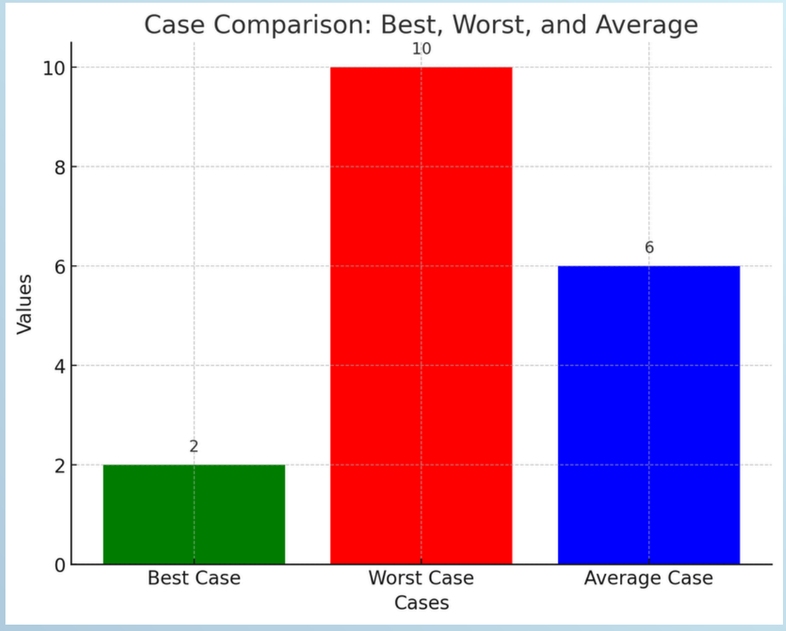
# Design of an Algorithm

- Clearly define inputs, outputs, and constraints 📝
- Decide technique: Divide & Conquer, Greedy, Dynamic Programming, Brute Force
- Write step-by-step instructions or flowchart 🔹
- Ensure algorithm produces correct output for all valid inputs ✅
- Evaluate time and space complexity, optimize if necessary ⏱️💾
- Test the algorithm with different inputs 🧪
- Refine or simplify steps for easier understanding ✨

# Analysis of an algorithm




**Time Complexity**

**Space Complexity**

**Best ,Worst . Average Case**

# Need of an algorithm

- Provides step-by-step clarity in solving problems 📝
- Ensures efficiency in terms of time ⏱️ and space 💾
- Improves reusability across different problems ♻️
- Helps in optimization by finding the best solution 📈
- Reduces chances of errors ❌
- Acts as the foundation of programming 💻

# Correctness of an algorithm

- **Definition:** An algorithm is correct if it produces the expected output for all valid inputs. ✅
- **Importance:** Ensures reliability and trustworthiness of the solution.
- **Method of Proof:** Can be proven using:
- Mathematical induction
- Loop invariants

## How To Prove Correctness Of an Algorithm ?

*Q2)* a) Prove the correctness of the following algorithm, which finds the factorial of a given number.

```
int fact (int n)
{ if (n==0) then return 1;
  else
       return (n * fact(n-1));
```



- **Proving Correctness (Steps):**

1. **Precondition**: n ≥ 0 (factorial is defined for non-negative integers)

2. **Postcondition:** Output = n! (factorial of n)

3. **Base Case:** n=0 ↠ fact(0) returns 1 ✅ (0! = 1)

## 4. Inductive Step:

**Assume fact(k) returns k! correctly**

**Then fact(k+1) = (k+1) * fact(k) = (k+1) * k! = (k+1)!** ✅

## 5. Conclusion:

**Algorithm correctly returns factorial for all n ≥ 0**

# Invariant Loop

1. **Definition:** A loop invariant is a condition or property that remains true before and after each iteration of a loop. ✅

2. **Purpose:** Helps to prove the correctness of loops in an algorithm.

3. **Usage:**

- Identify invariant before loop starts
- Show it holds during iterations

**How To Prove Correctness Of an Algorithm ?**

# Summation of n numbers

```
sum = 0;
cout << "Enter a positive integer: ";
cin >> num;
for (int count = 1; count <= num; ++count) {
    sum += count;
```

- **Steps:**

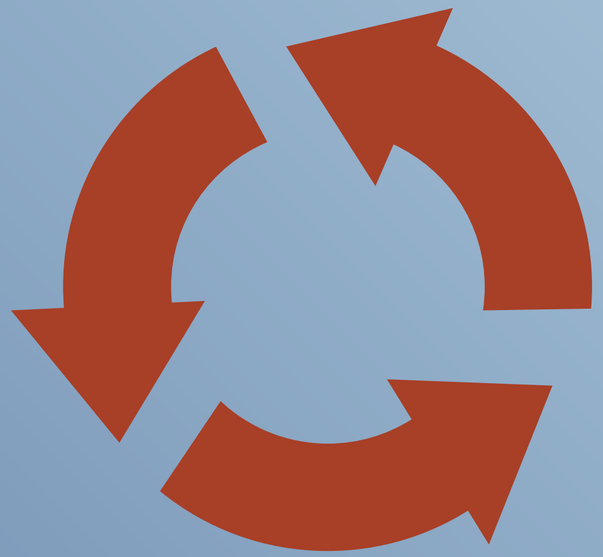
**1. Initialization: Before first iteration (i=1), sum = 0 = 1 + 2 + ... + (1-1) ✅**

**2. Maintenance: Assume invariant holds at start of iteration i ↠ sum of first (i-1) numbers**

**After adding i ↠ sum = 1 + 2 + ... + i ✅**

3. **Termination:** After last iteration i=n, invariant ↠ sum = 1 + 2 + ... + n ✅

4. **Conclusion:** Loop correctly calculates summation of n numbers



# Iterative algorithm

1. Definition: Repeats a set of instructions until a condition is met using loops (for, while, etc.)

2. Key Feature: Solves problems by iteration instead of recursion, usually with loops and counters 🔢

## DESIGN ISSUES

1. Loop Control Complexity: Ensure loop starts, runs correct number of times, and terminates properly ⚠️

2. Initialization Errors: Variables must be initialized correctly, otherwise wrong results ?

3. **Boundary Conditions:** Handle first, last, or edge values correctly ✅

4. **Maintenance of Invariants:** Ensure variables/conditions that must remain true are updated 🔄

5. **Performance / Efficiency**

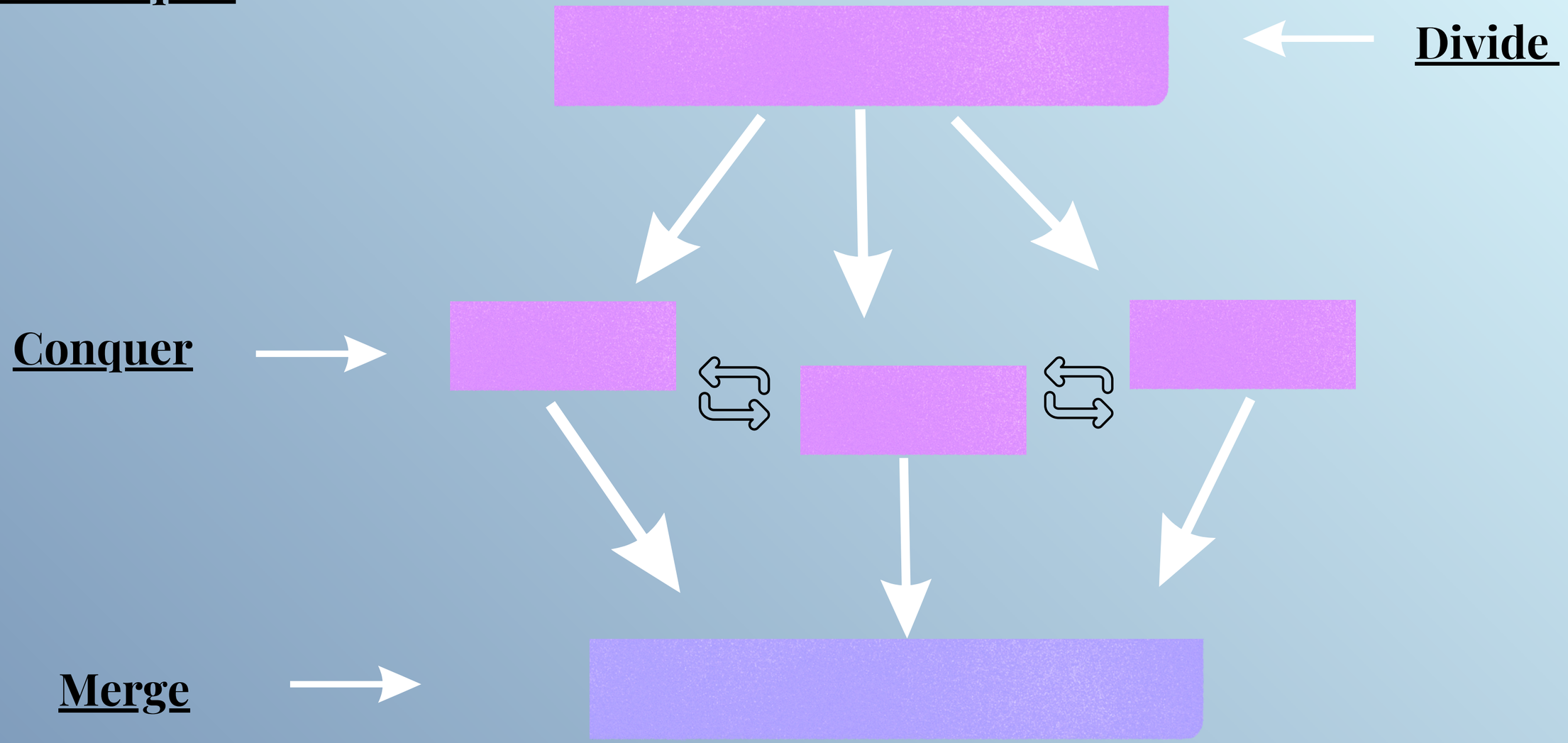
Iterative loops can sometimes run many unnecessary iterations if not designed carefully

6. **Readability / Maintainability**

Complex loops with many counters and conditions make code hard to read and debug 📝

# Problem solving Strategies

## Divide and Conquer



# Greedy Strategy

## Make rs 37 by using coins of 25,10,5,1

Start 37


choose 25 ,remaining -12

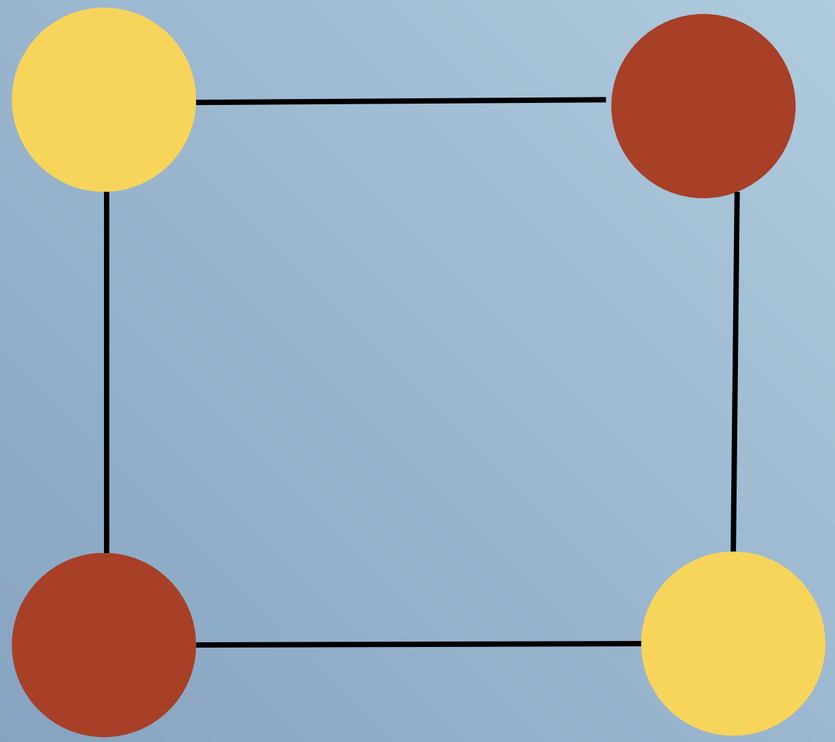
choose 10 ,remaining -2


choose 1 ,remaining 1


choose 1 ,remaining 0

# Backtracking

## Graph coloring Problem



- **Available colors:**

1. **Red**
2. **yellow**

**Constraint:-**

**No two ajacent sides should have same color**

Thank you