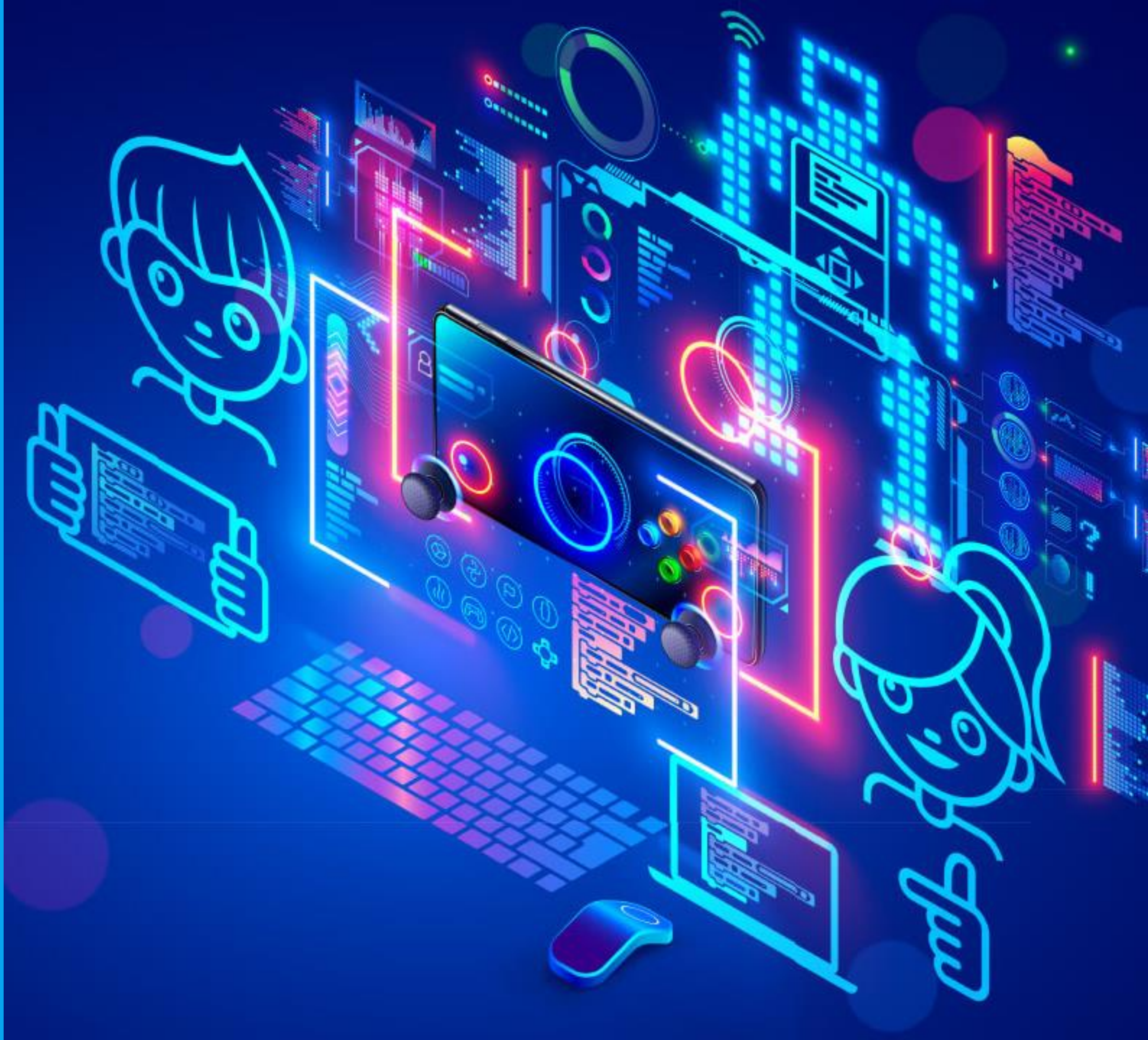
# CS 24 AP Computer Science A Review

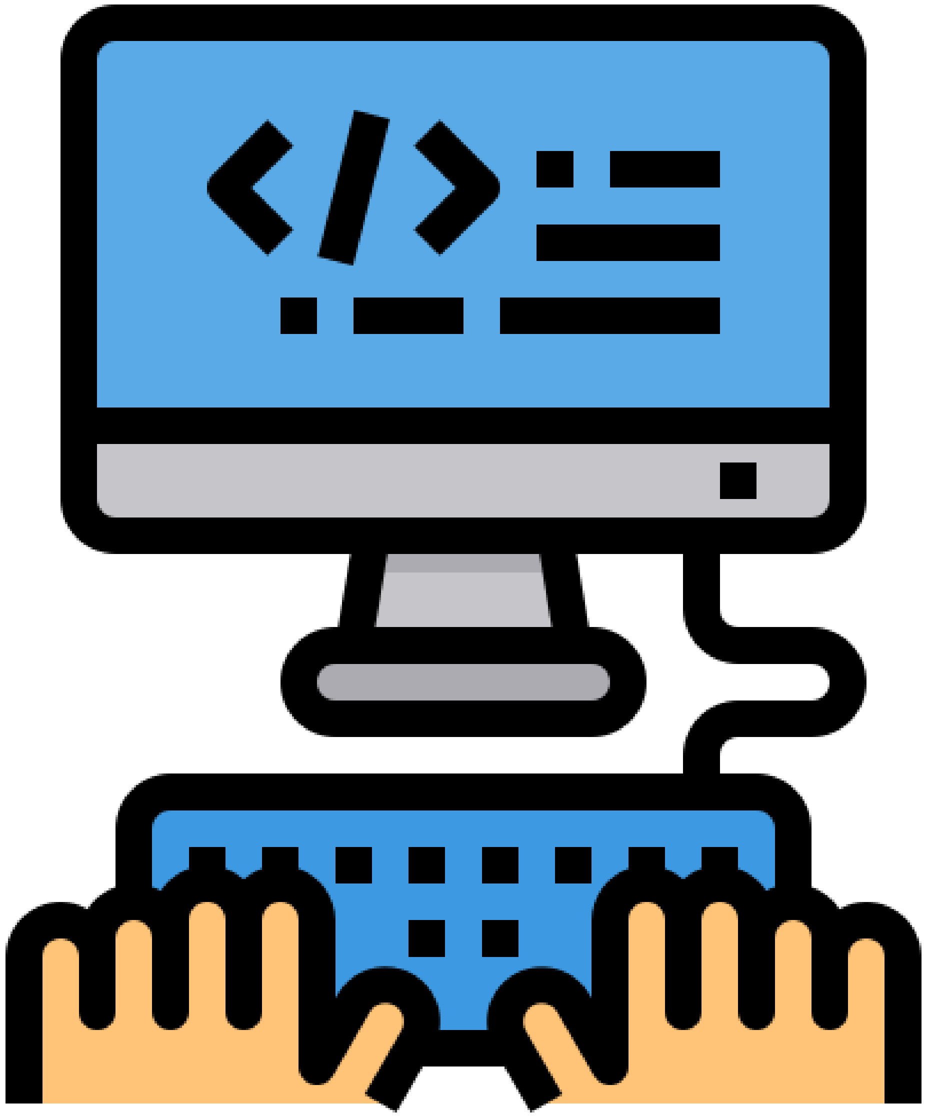
## Week 7: Recursion and Algorithm Analysis

DR. ERIC CHOU
IEEE SENIOR MEMBER



## Topics

- Number of Function Calls, Number of Comparators, Number of Operators, Memory Requirement.
- Complexity Analysis
- Recursion (Sum, Factorial, Successive Division)
- Fibonacci, Euclid Algorithm, Divide and Conquer, Dynamic Programming
- Recursive Function Over Array
- Recursion on 2D Grid
- Recursion versus Iteration



## Number of Function Calls

Section 1

# Developing Algorithms – Problem Solving

## Problem Solving

- Sorting
- Approximate Methods (ad hoc solutions)
- Randomized Solutions
- Dynamic Programming
- Linear Programming
- Cryptography
- Fourier Transform
- Computational Geometry
- Graph Theory

**Algorithm Efficiency Analysis (Time Complexity)**

Big-O Notation

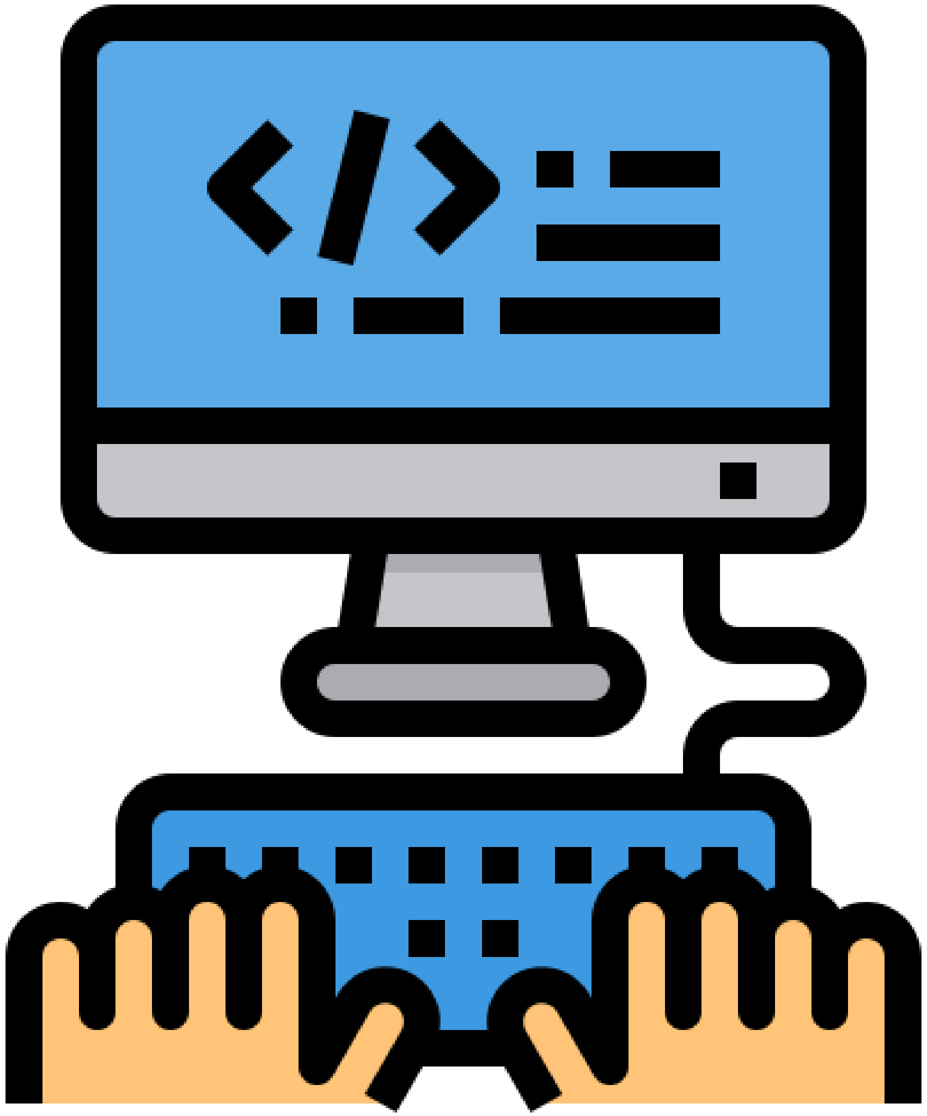
NP-Completeness

**Data Structure (Memory Complexity/Time Complexity Trade-offs)**

Binary

B-Tree and other Tree

Stack, Queue, List, Map, Set, Heap, Sparse Matrix



## Complexity Analysis

Section 2

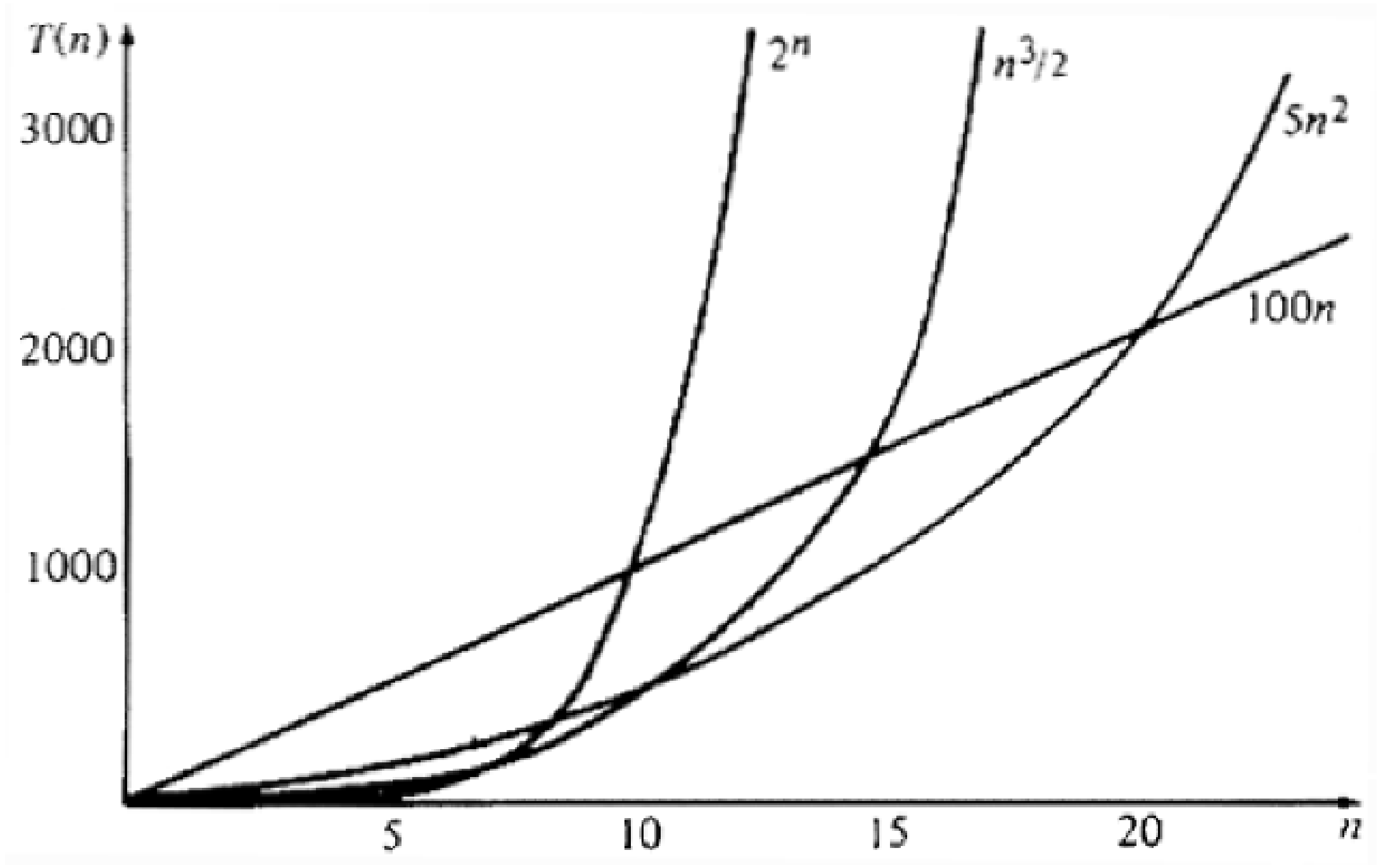
## Commonly asked Questions

- Number of Recursive Calls
- The result of Recursion

## Big O Notation

- Consider linear search. The linear search algorithm compares the key with the elements in the array sequentially until the key is found or the array is exhausted. If the key is not in the array, it requires *n* comparisons for an array of size *n*. If the key is in the array, **it requires *n/2* comparisons on average**.
- The algorithm's execution time is proportional to the size of the array. If you double the size of the array, you will expect the number of comparisons to double. The algorithm grows at a linear rate. The growth rate has an order of magnitude of *n*. Computer scientists use the Big *O* notation to abbreviate for "order of magnitude." Using this notation, the complexity of the linear search algorithm is ***O(n)***, pronounced as "***order of n***."

## Approximate Growth Rate *T(n)*



## Best, Worst, and Average Cases

- For the same input size, an algorithm’s execution time may vary, depending on the input. An input that results in the shortest execution time is called the best-case input and an input that results in the longest execution time is called the worst-case input. Best-case and worst-case are not representative, but worst-case analysis is very useful. You can show that the algorithm will never be slower than the worst-case.
- **An average-case analysis attempts to determine the average amount of time among all possible input of the same size.** Average-case analysis is ideal, but difficult to perform, because it is hard to determine the relative probabilities and distributions of various input instances for many problems. Worst-case analysis is easier to obtain and is thus common. So, the analysis is generally conducted for the worst-case.

## Ignoring Multiplicative Constants

- The linear search algorithm requires n comparisons in the worst-case and **n/2** comparisons in the average-case. Using the Big O notation, both cases require **O(n)** time. The multiplicative constant **(1/2)** can be omitted.
- Algorithm analysis is focused on growth rate. The multiplicative constants have no impact on growth rates.
- The growth rate for n/2 or 100n is the same as n, i.e., **O(n) = O(n/2) = O(100n).**

## Ignoring Non-Dominating Terms

- Consider the algorithm for finding the maximum number in an array of n elements. If  n is 2, it takes one comparison to find the maximum number. If n is 3, it takes two comparisons to find the maximum number. In general, it takes n-1 times of comparisons to find maximum number in a list of  n elements.
- Algorithm analysis is for large input size. If the input size is small, there is no significance to estimate an algorithm's efficiency. As n grows larger, the n part in the expression n-1 dominates the complexity.
- The Big  O notation allows you to ignore the non-dominating part (e.g., -1 in the expression n-1) and highlight the important part (e.g., n in the expression n-1). So, the complexity of this algorithm is O(n).

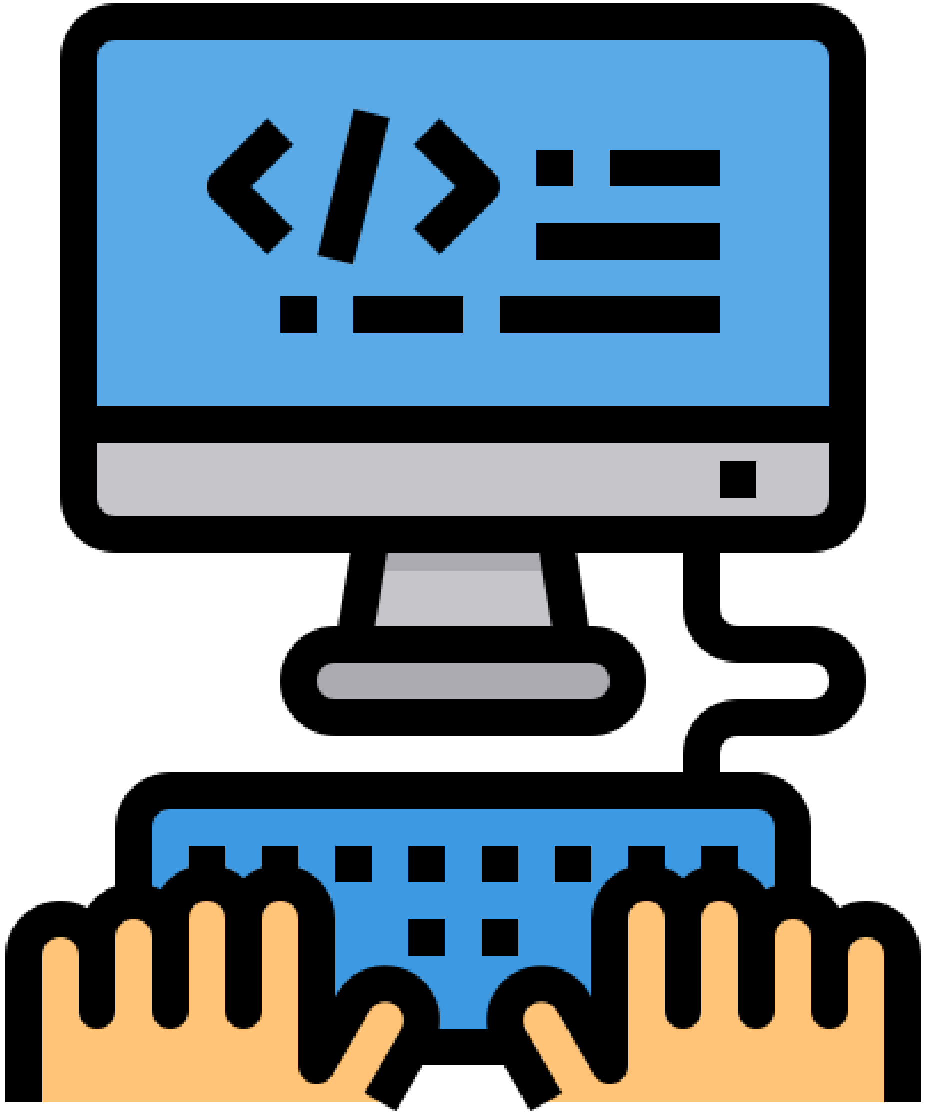
## Constant Time

- The Big **O(n)** notation estimates the execution time of an algorithm in relation to the input size. If the time is not related to the input size, the algorithm is said to take constant time with the notation **O(1)**.
- For example, a method that retrieves an element at a given index in an array takes constant time, because it does not grow as the size of the array increases.

## Comparing Common Growth Functions

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

| | |
|---|---|
| $O(1)$ | Constant time |
| $O(\log n)$ | Logarithmic time |
| $O(n)$ | Linear time |
| $O(n \log n)$ | Log-linear time |
| $O(n^2)$ | Quadratic time |
| $O(n^3)$ | Cubic time |
| $O(2^n)$ | Exponential time |



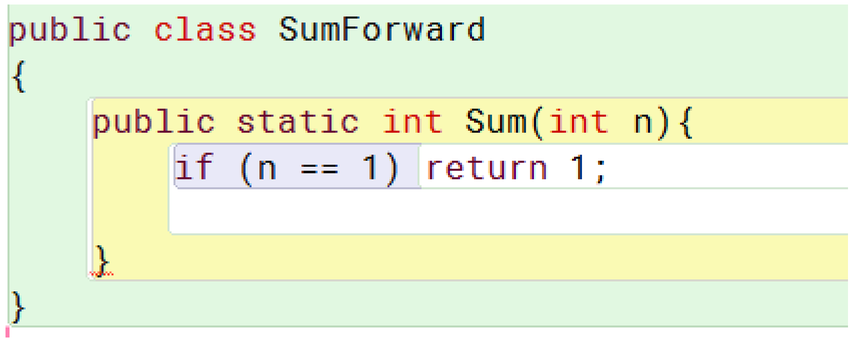
## Recursion

Section 3

## Steps to write a Recursive Program

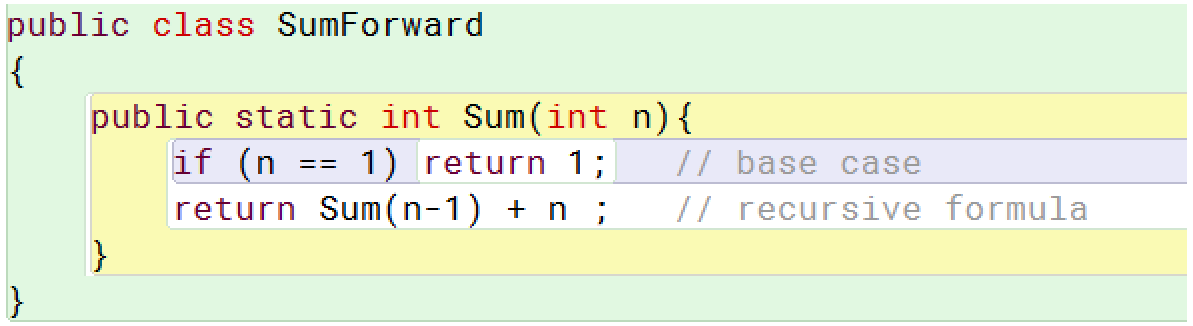
1. The Termination Condition (Base Case)
2. Recursive Formula (Recursive Call)
3. Step by Step Improvement

# Sum of 1 to n



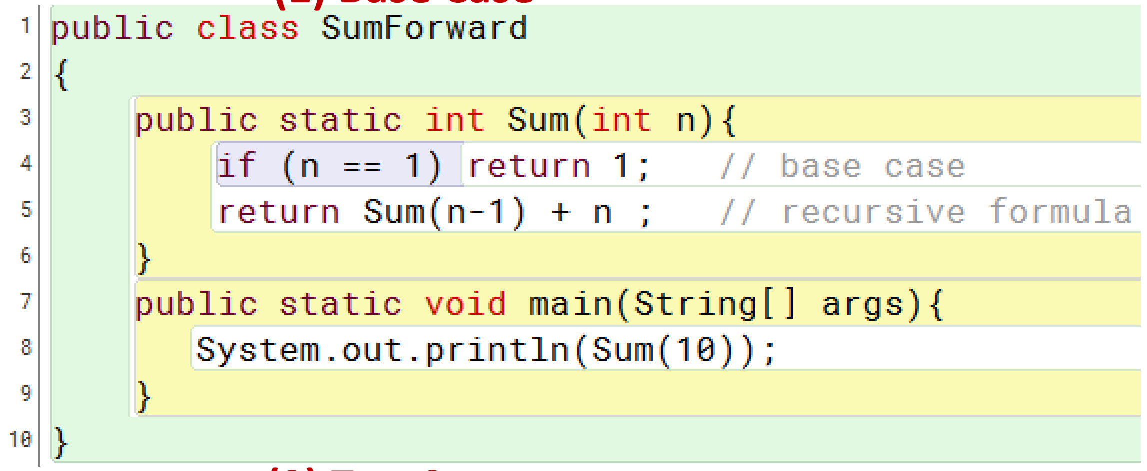
```
public class SumForward
{
    public static int Sum(int n){
        if (n == 1) return 1;

    }
}
```

(1) Base Case



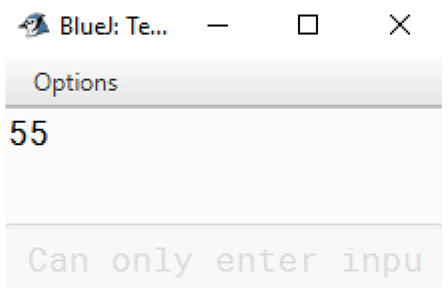
```
public class SumForward
{
    public static int Sum(int n){
        if (n == 1) return 1;    // base case
        return Sum(n-1) + n ;    // recursive formula
    }
}
```

(2) Recursive Formula



```
public class SumForward
{
    public static int Sum(int n){
        if (n == 1) return 1;    // base case
        return Sum(n-1) + n ;    // recursive formula
    }
    public static void main(String[] args){
       System.out.println(Sum(10));
    }
}
```

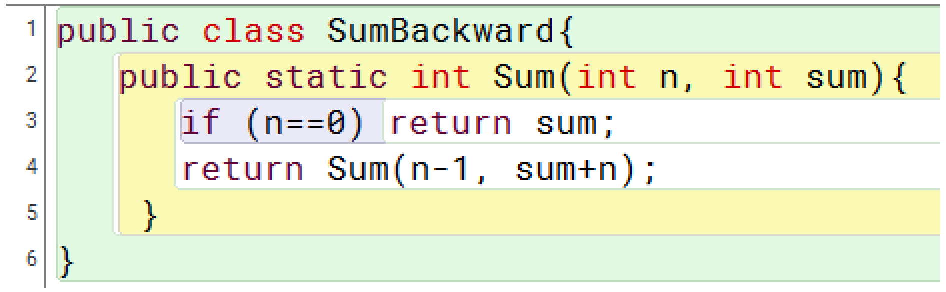
(3) Test Setup



```
55
```

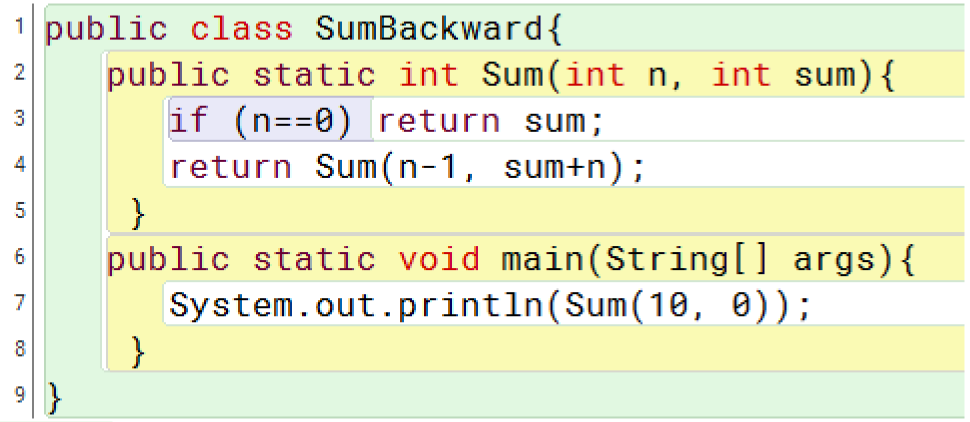
# Sum of 1 to n

## Helper Method with Accumulator (For tail-recursive methods)



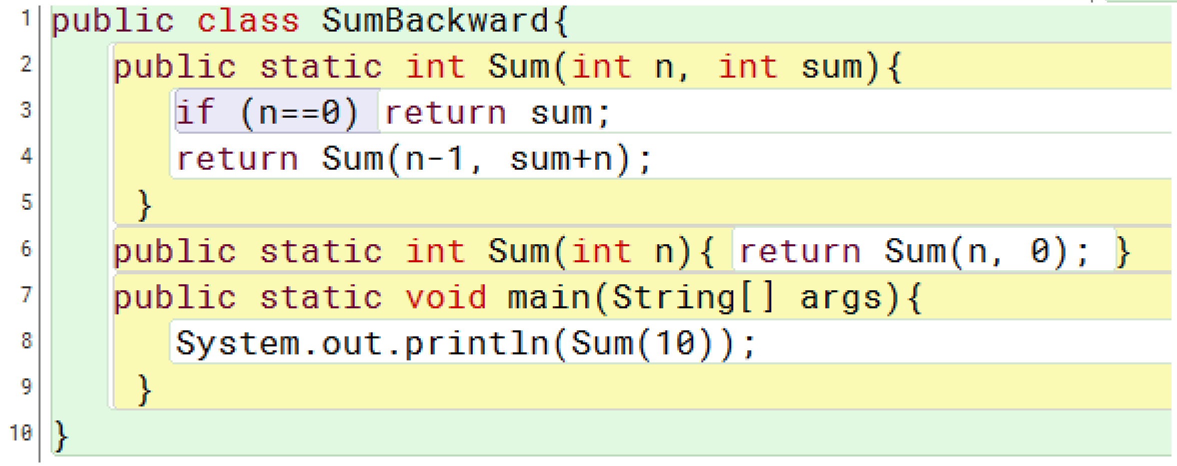
```java
public class SumBackward{
    public static int Sum(int n, int sum){
        if (n==0) return sum;
        return Sum(n-1, sum+n);
    }
}
```

**(1) Base Case**

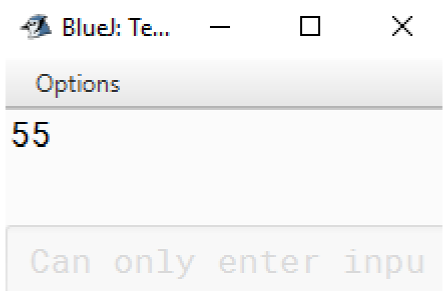


```java
public class SumBackward{
    public static int Sum(int n, int sum){
        if (n==0) return sum;
        return Sum(n-1, sum+n);
    }
    public static void main(String[] args){
        System.out.println(Sum(10, 0));
    }
}
```

**(2) Recursive Formula**



```java
public class SumBackward{
    public static int Sum(int n, int sum){
        if (n==0) return sum;
        return Sum(n-1, sum+n);
    }
    public static int Sum(int n){ return Sum(n, 0); }
    public static void main(String[] args){
        System.out.println(Sum(10));
    }
}
```



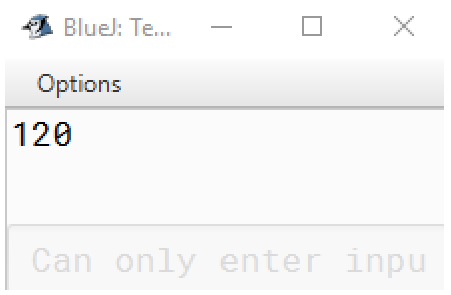
```
55
```

**(3) Test Case with Normal Method**

## Factorial

```java
public class Factorial{
	public static int f(int n){
		if (n==0) return 1;
		return f(n-1)*n;
	}
	public static void main(String[] args){
		System.out.println(f(5));
	}
}
```

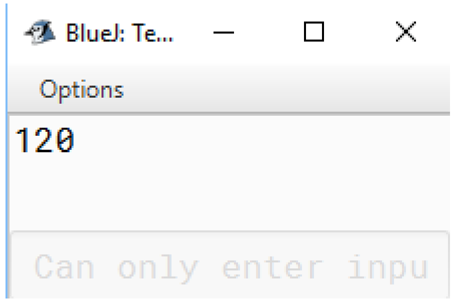
**(1) Normal Version**



```
120
```

```java
public class FactorialHelper{
	public static int f_helper(int n, int product){
		if (n==0) return product;
		return f_helper(n-1, product * n);
	}
	public static int f(int n){ return f_helper(5, 1);}
	public static void main(String[] args){
		System.out.println(f(5));
	}
}
```

**(2) Tail-recursive Version**



```
120
```

## Conversion of Sum to Iterative

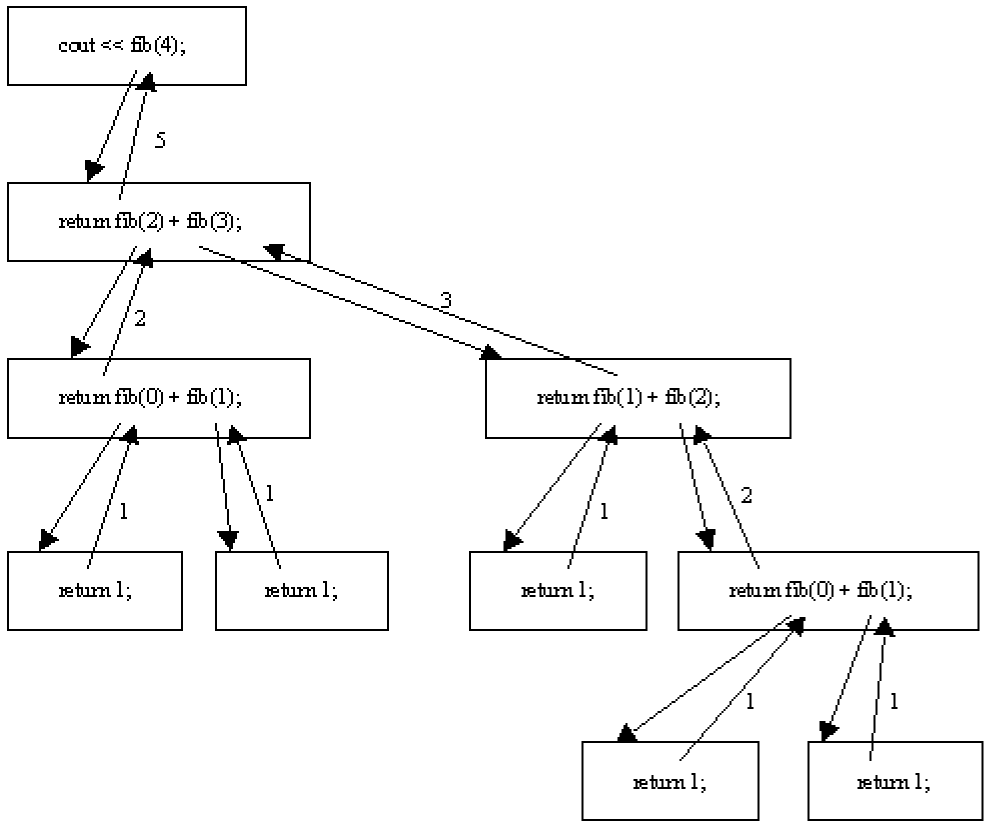
```
public static int Sum(int n){

      if (n == 1) return 1;   // base case

      return Sum(n-1) + n ;   // recursive formula

   }
```

**Base condition**

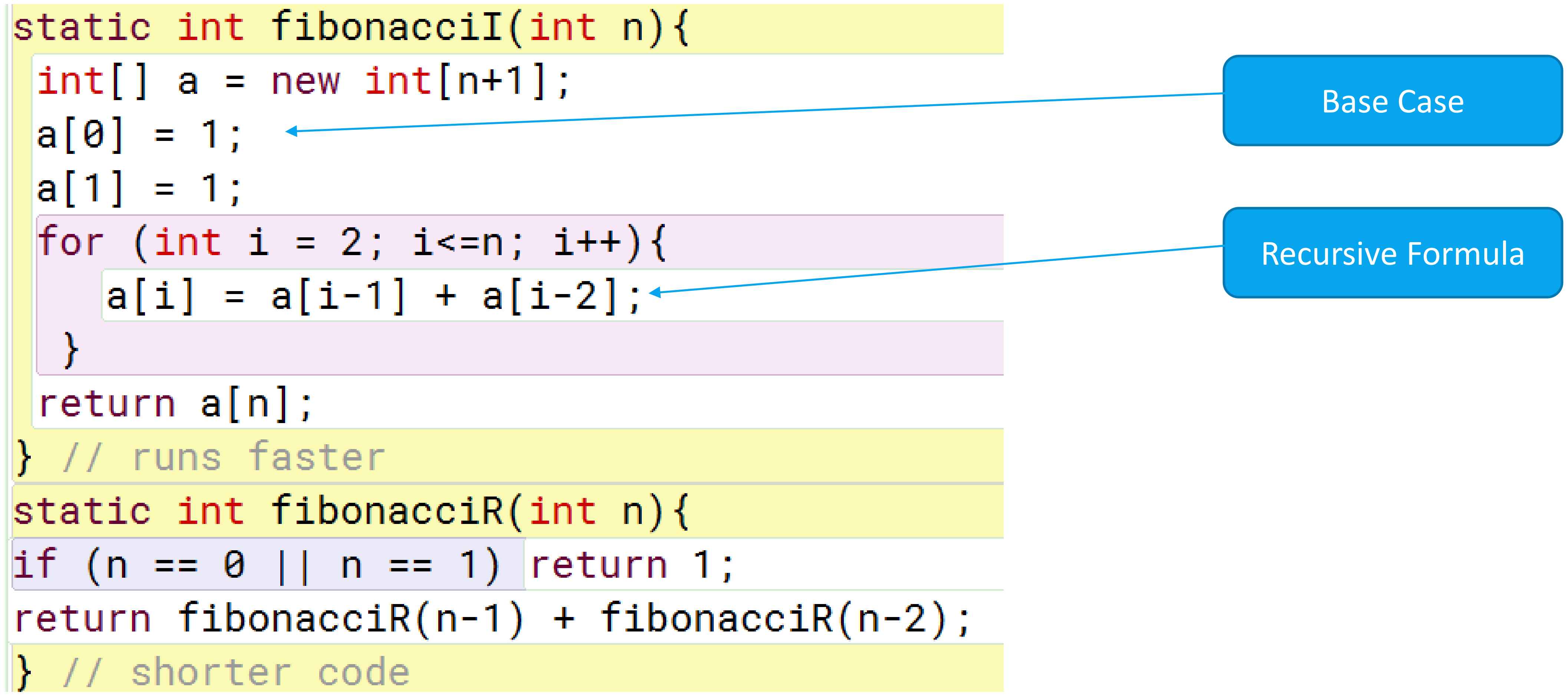
**Recursive Formula**

```
public static int Sum(int n){
   int s = 0;
   for (int i=0; i<=n; i++){
         s = s + i;
   }
   return s;
}
```

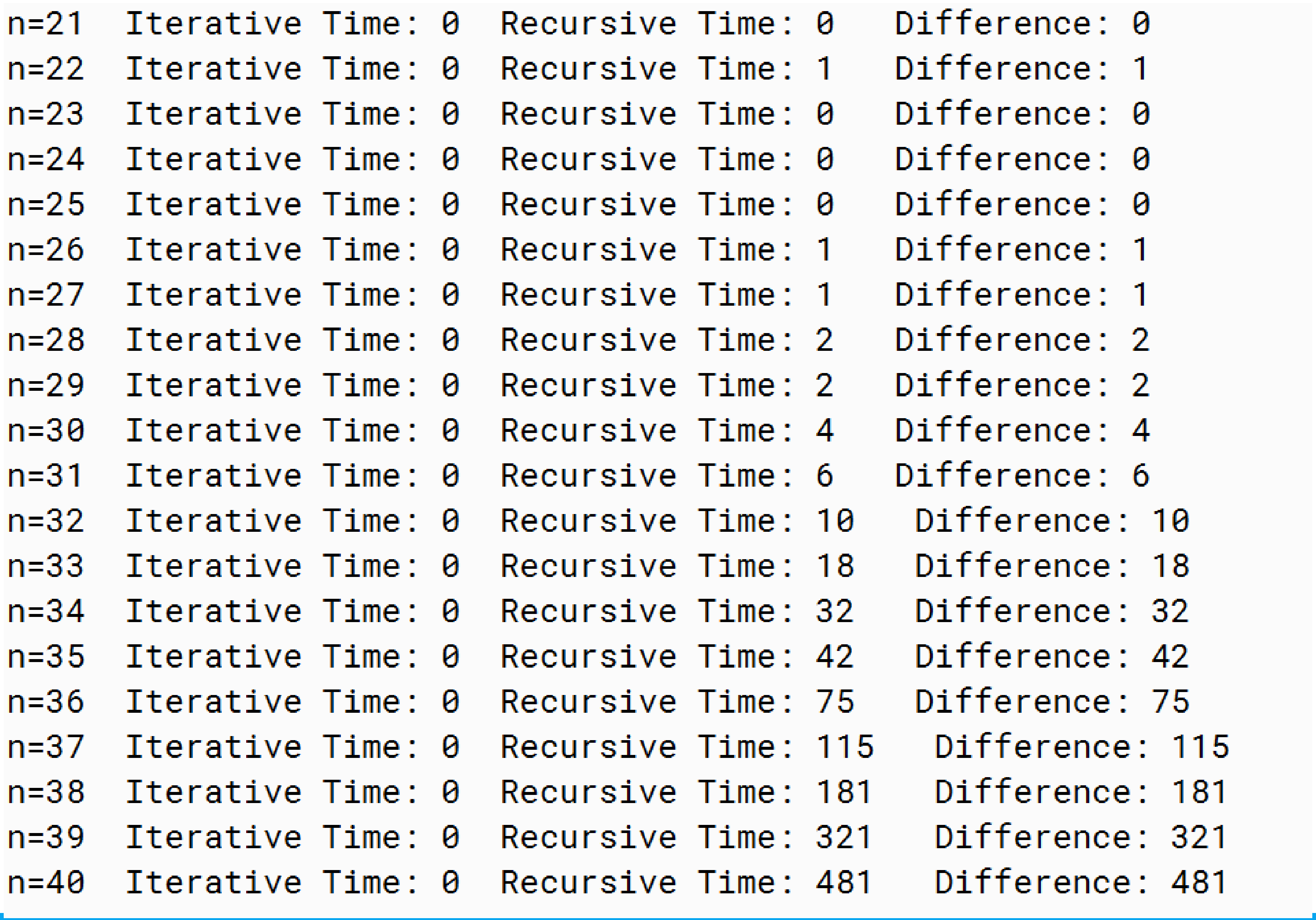
# Tracing Recursive Functions



# Recursive Versus Iterative



```
static int fibonacciI(int n){
 int[] a = new int[n+1];
 a[0] = 1;
 a[1] = 1;
 for (int i = 2; i<=n; i++){
    a[i] = a[i-1] + a[i-2];
  }
 return a[n];
} // runs faster
static int fibonacciR(int n){
if (n == 0 || n == 1) return 1;
return fibonacciR(n-1) + fibonacciR(n-2);
} // shorter code
```



```
n=21  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=22  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=23  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=24  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=25  Iterative Time: 0  Recursive Time: 0   Difference: 0
n=26  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=27  Iterative Time: 0  Recursive Time: 1   Difference: 1
n=28  Iterative Time: 0  Recursive Time: 2   Difference: 2
n=29  Iterative Time: 0  Recursive Time: 2   Difference: 2
n=30  Iterative Time: 0  Recursive Time: 4   Difference: 4
n=31  Iterative Time: 0  Recursive Time: 6   Difference: 6
n=32  Iterative Time: 0  Recursive Time: 10   Difference: 10
n=33  Iterative Time: 0  Recursive Time: 18   Difference: 18
n=34  Iterative Time: 0  Recursive Time: 32   Difference: 32
n=35  Iterative Time: 0  Recursive Time: 42   Difference: 42
n=36  Iterative Time: 0  Recursive Time: 75   Difference: 75
n=37  Iterative Time: 0  Recursive Time: 115   Difference: 115
n=38  Iterative Time: 0  Recursive Time: 181   Difference: 181
n=39  Iterative Time: 0  Recursive Time: 321   Difference: 321
n=40  Iterative Time: 0  Recursive Time: 481   Difference: 481
```

## Tracing Recursive Functions

- Write function calls one by one.
- Write function name in a letter if possible.
- Be aware of what need to be added and what need to be returned.
- Trace them step by step.

39. Consider the following recursive method.

```
public int recur(int n)
{
  if (n <= 10)
    return n * 2;
  else
    return recur(recur(n / 3));
}
```

What value is returned as a result of the call `recur(27)` ?

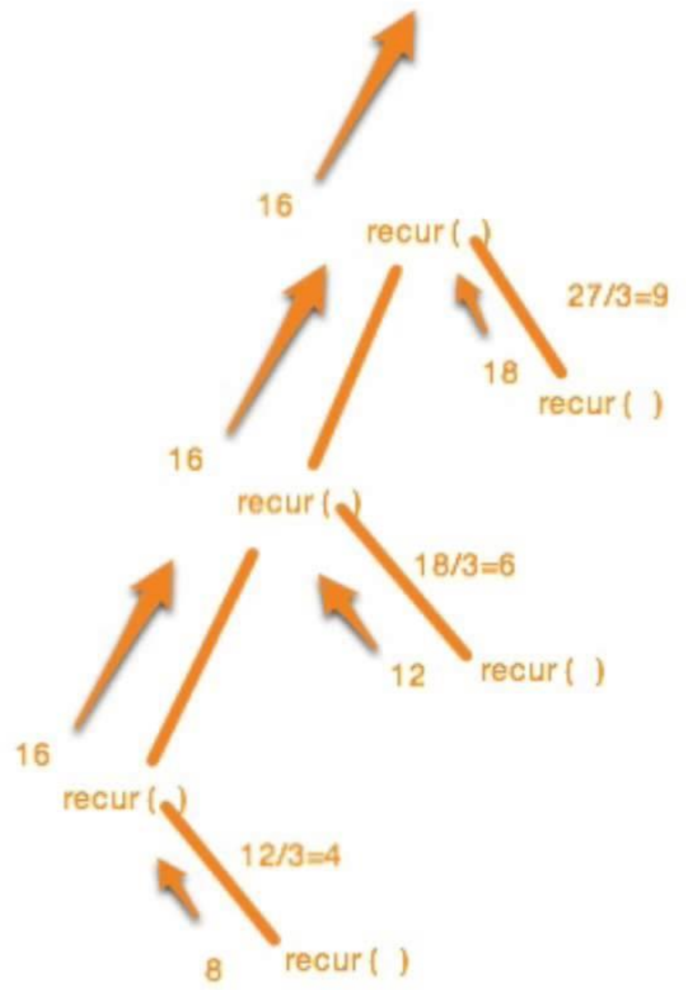
(A) 8

(B) 9

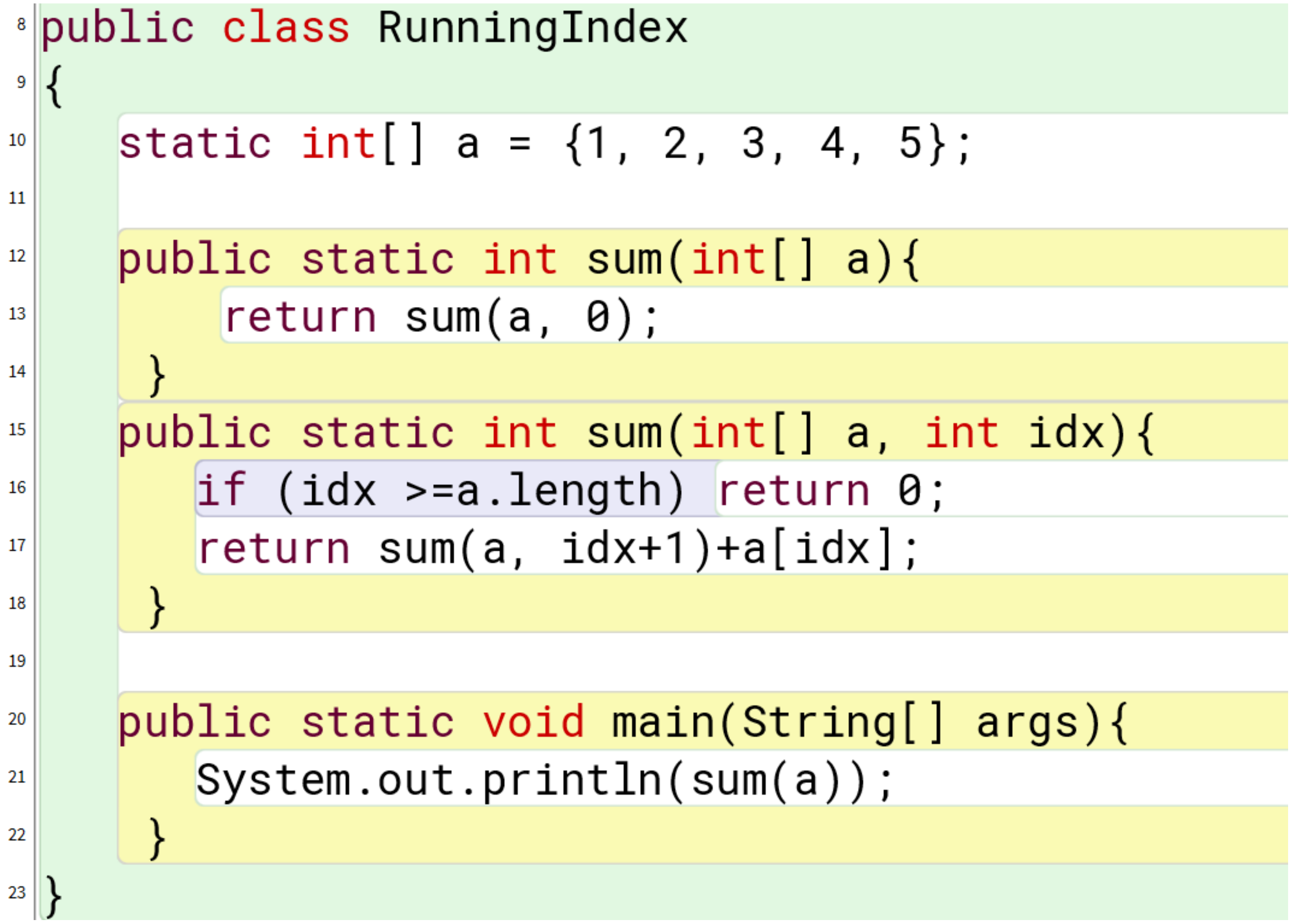
(C) 12

(D) 16

(E) 18



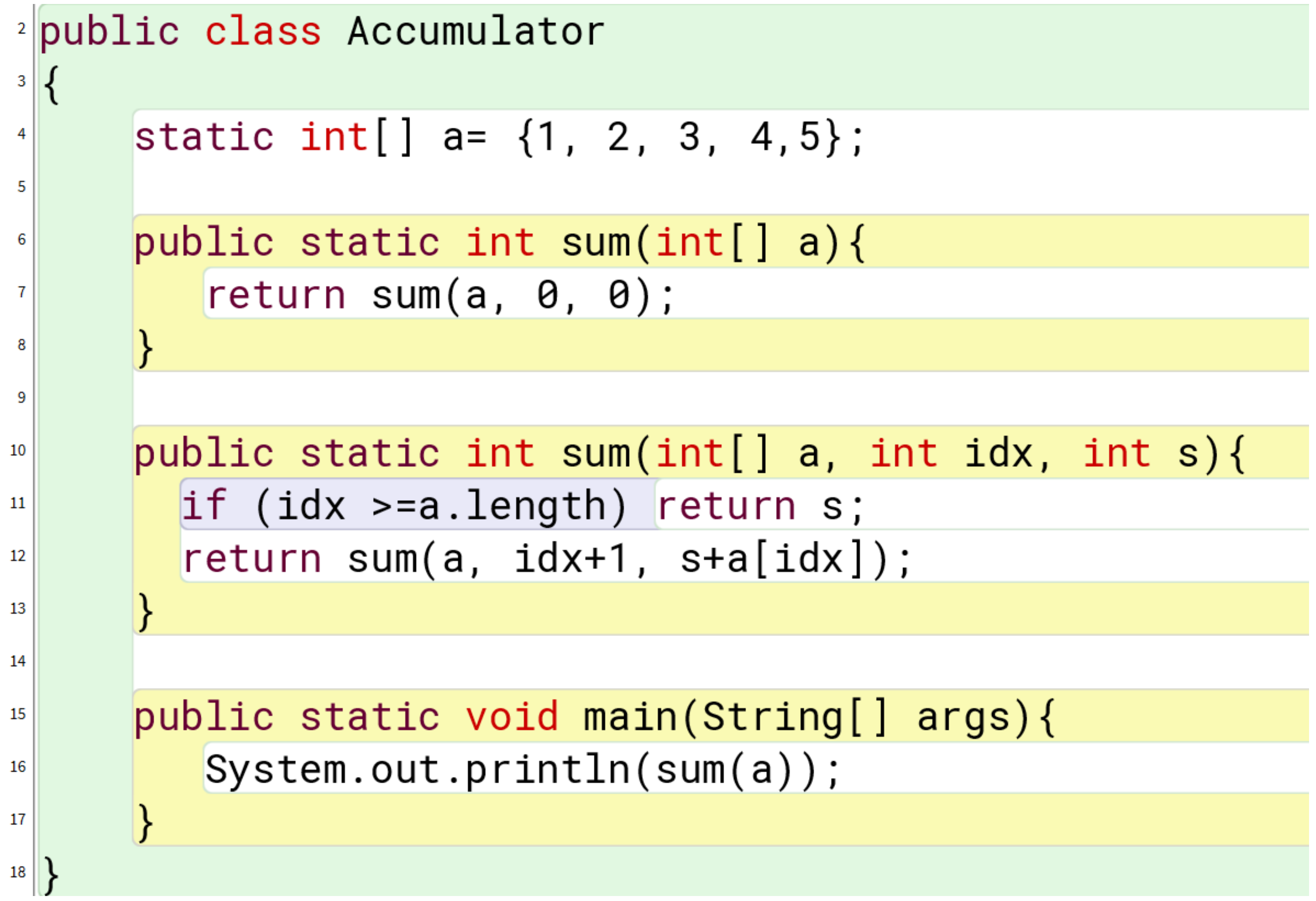
Write down the recursive steps one by one like a robot!!!
recur(27)-> recur(recur(9))->
recur(18)->recur(recur(6))->
recur(12)-> recur(recur(4))->
recur(8) -> return 16



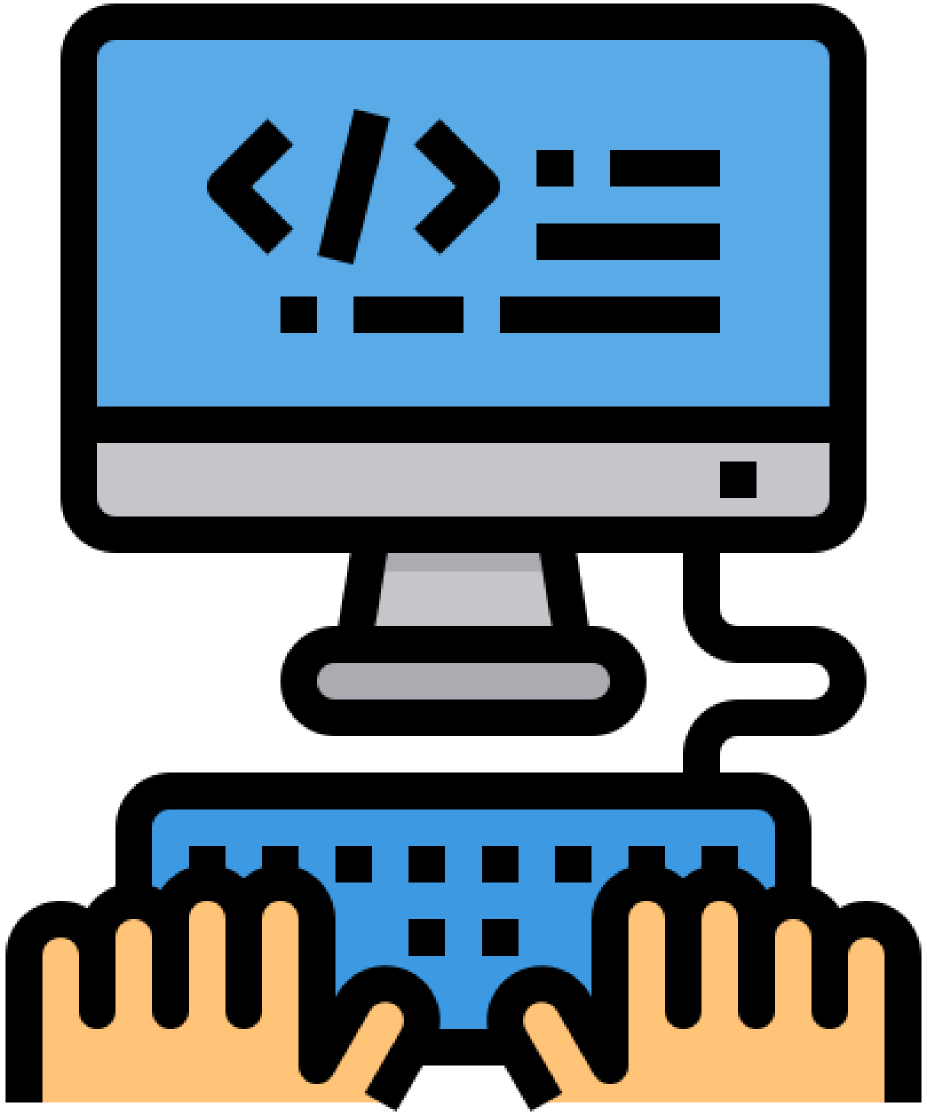
```java
public class RunningIndex
{
    static int[] a = {1, 2, 3, 4, 5};

    public static int sum(int[] a){
         return sum(a, 0);
      }
    public static int sum(int[] a, int idx){
        if (idx >=a.length) return 0;
        return sum(a, idx+1)+a[idx];
      }

    public static void main(String[] args){
        System.out.println(sum(a));
      }
}
```

## Running Index Recursion

# Recursion with Accumulator



```java
public class Accumulator
{
    static int[] a= {1, 2, 3, 4,5};

    public static int sum(int[] a){
        return sum(a, 0, 0);
    }

    public static int sum(int[] a, int idx, int s){
      if (idx >=a.length) return s;
      return sum(a, idx+1, s+a[idx]);
    }

    public static void main(String[] args){
        System.out.println(sum(a));
    }
}
```



## Recursive Function Over Array

Section 4

## Palindrome Examples

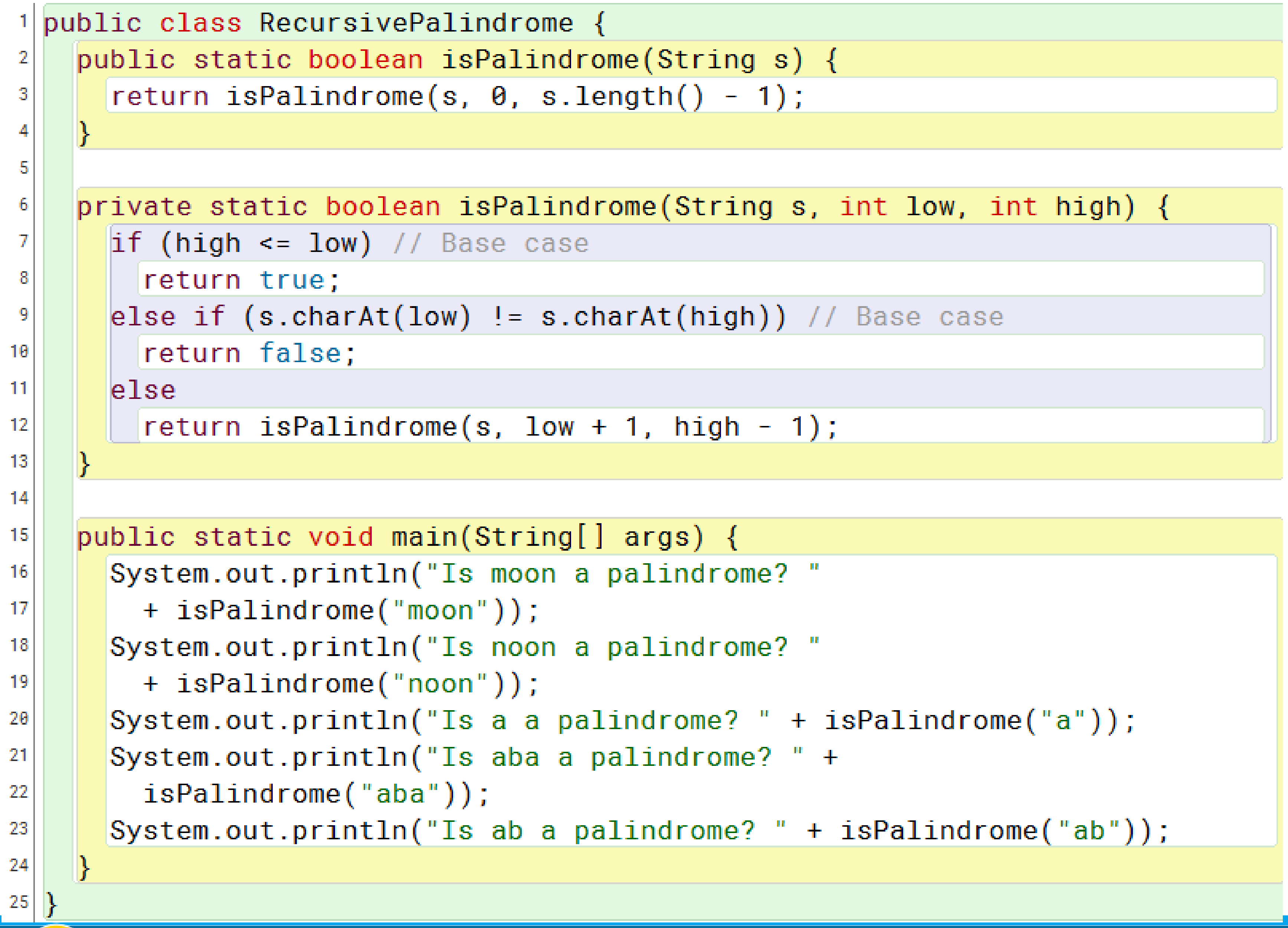
- A palindrome is a word, phrase, number or sequence of words that reads the same backwards as forwards. Punctuation and spaces between the words or lettering is allowed (or not in CS).

Single Word Palindromes

| | | |
|---|---|---|
| •Anna | •Radar | •Stats |
| •Civic | •Redder | •Tenet |
| •Kayak | •Refer | •Wow |
| •Level | •Repaper | |
| •Madam | •Rotator | |
| •Mom | •Rotor | |
| •Noon | •Sagas | |
| •Racecar | •Solos | |

## Recursive Palindrome

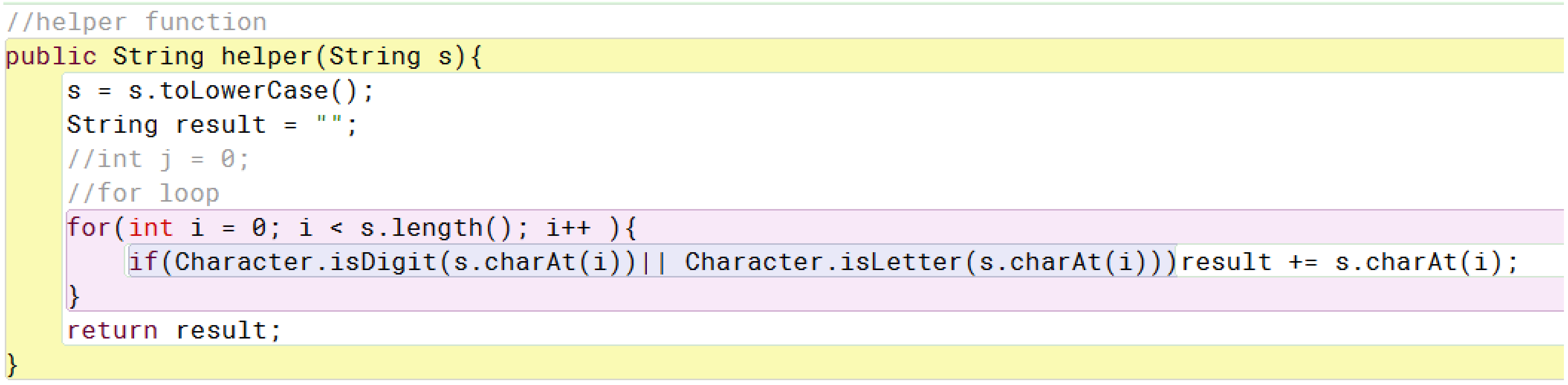
- Recursively check the string’s characters layer by layer.  If two characters of the same layer are matched, then continue to check otherwise reporting false.
- If no characters (or one characters left) then declare that the input string is indeed a palindrome word.



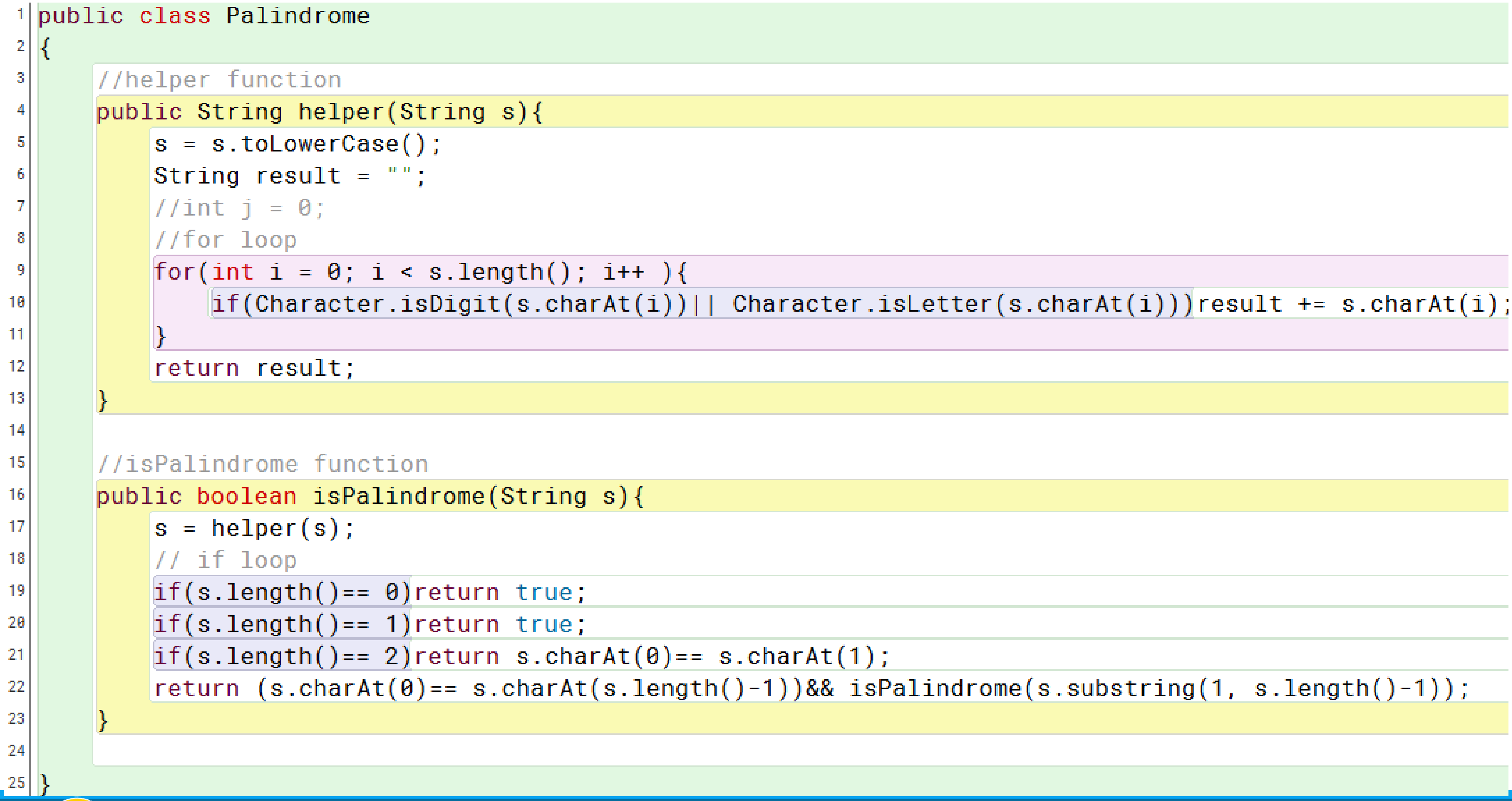
```java
public class RecursivePalindrome {
  public static boolean isPalindrome(String s) {
    return isPalindrome(s, 0, s.length() - 1);
  }

  private static boolean isPalindrome(String s, int low, int high) {
    if (high <= low) // Base case
      return true;
    else if (s.charAt(low) != s.charAt(high)) // Base case
      return false;
    else
      return isPalindrome(s, low + 1, high - 1);
  }

  public static void main(String[] args) {
    System.out.println("Is moon a palindrome? "
      + isPalindrome("moon"));
    System.out.println("Is noon a palindrome? "
      + isPalindrome("noon"));
    System.out.println("Is a a palindrome? " + isPalindrome("a"));
    System.out.println("Is aba a palindrome? " +
      isPalindrome("aba"));
    System.out.println("Is ab a palindrome? " + isPalindrome("ab"));
  }
}
```

## Adding a Helper to Remove Unrelated Characters and Allow Case-Insensitiveness

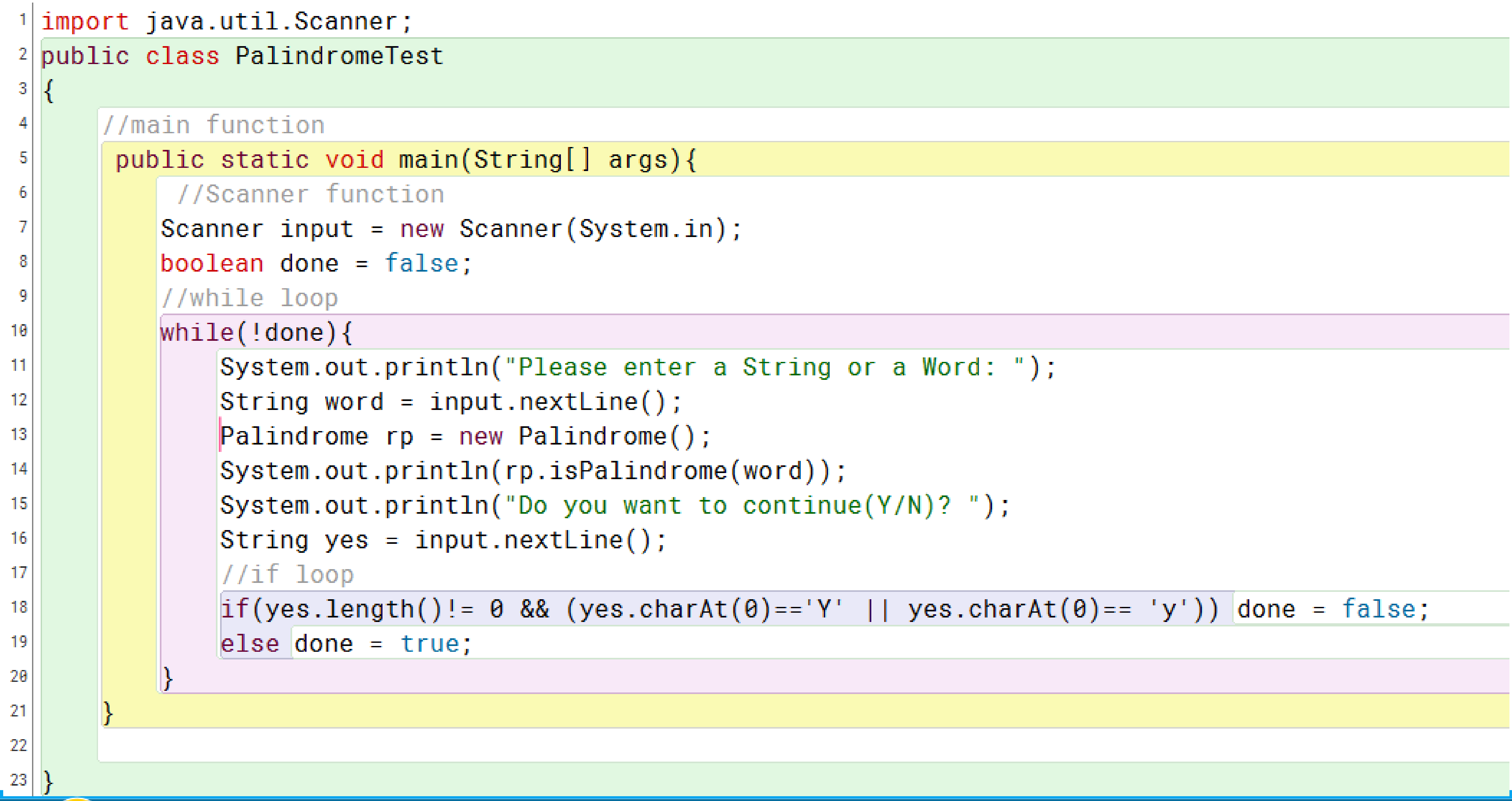
- Convert all letters to lowercase (check palindrome under case-insensitive criteria)
- Remove non-digit and non-letter characters.



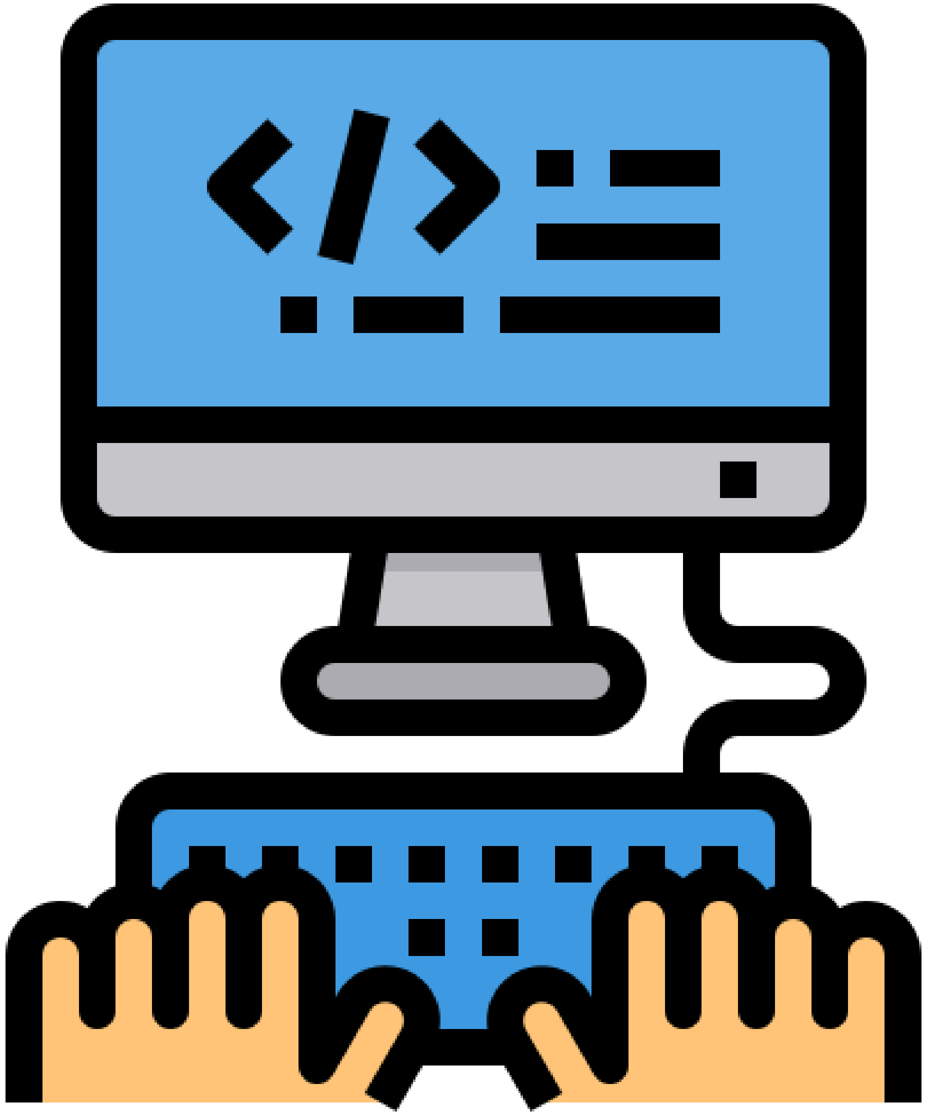
```
//helper function
public String helper(String s){
    s = s.toLowerCase();
    String result = "";
    //int j = 0;
    //for loop
    for(int i = 0; i < s.length(); i++ ){
        if(Character.isDigit(s.charAt(i))|| Character.isLetter(s.charAt(i)))result += s.charAt(i);
    }
    return result;
}
```



```java
public class Palindrome
{
    //helper function
    public String helper(String s){
        s = s.toLowerCase();
        String result = "";
        //int j = 0;
        //for loop
        for(int i = 0; i < s.length(); i++ ){
            if(Character.isDigit(s.charAt(i))|| Character.isLetter(s.charAt(i)))result += s.charAt(i);
        }
        return result;
    }

    //isPalindrome function
    public boolean isPalindrome(String s){
        s = helper(s);
        // if loop
        if(s.length()== 0)return true;
        if(s.length()== 1)return true;
        if(s.length()== 2)return s.charAt(0)== s.charAt(1);
        return (s.charAt(0)== s.charAt(s.length()-1))&& isPalindrome(s.substring(1, s.length()-1));
    }

}
```



```
import java.util.Scanner;
public class PalindromeTest
{
	//main function
	 public static void main(String[] args){
		 //Scanner function
		Scanner input = new Scanner(System.in);
		boolean done = false;
		//while loop
		while(!done){
			System.out.println("Please enter a String or a Word: ");
			String word = input.nextLine();
			Palindrome rp = new Palindrome();
			System.out.println(rp.isPalindrome(word));
			System.out.println("Do you want to continue(Y/N)? ");
			String yes = input.nextLine();
			//if loop
			if(yes.length()!= 0 && (yes.charAt(0)=='Y' || yes.charAt(0)== 'y')) done = false;
			else done = true;
		}
	}

}
```



# Recursion on 2D Grid

Section 5

# Traverse a given Matrix using Recursion

Given a matrix mat[][] of size n x m, the task is to traverse this matrix using recursion.

**Examples:**

**Input:**
```
int[][] mat = {{1, 2, 3},
                {4, 5, 6},
                {7, 8, 9}};
```

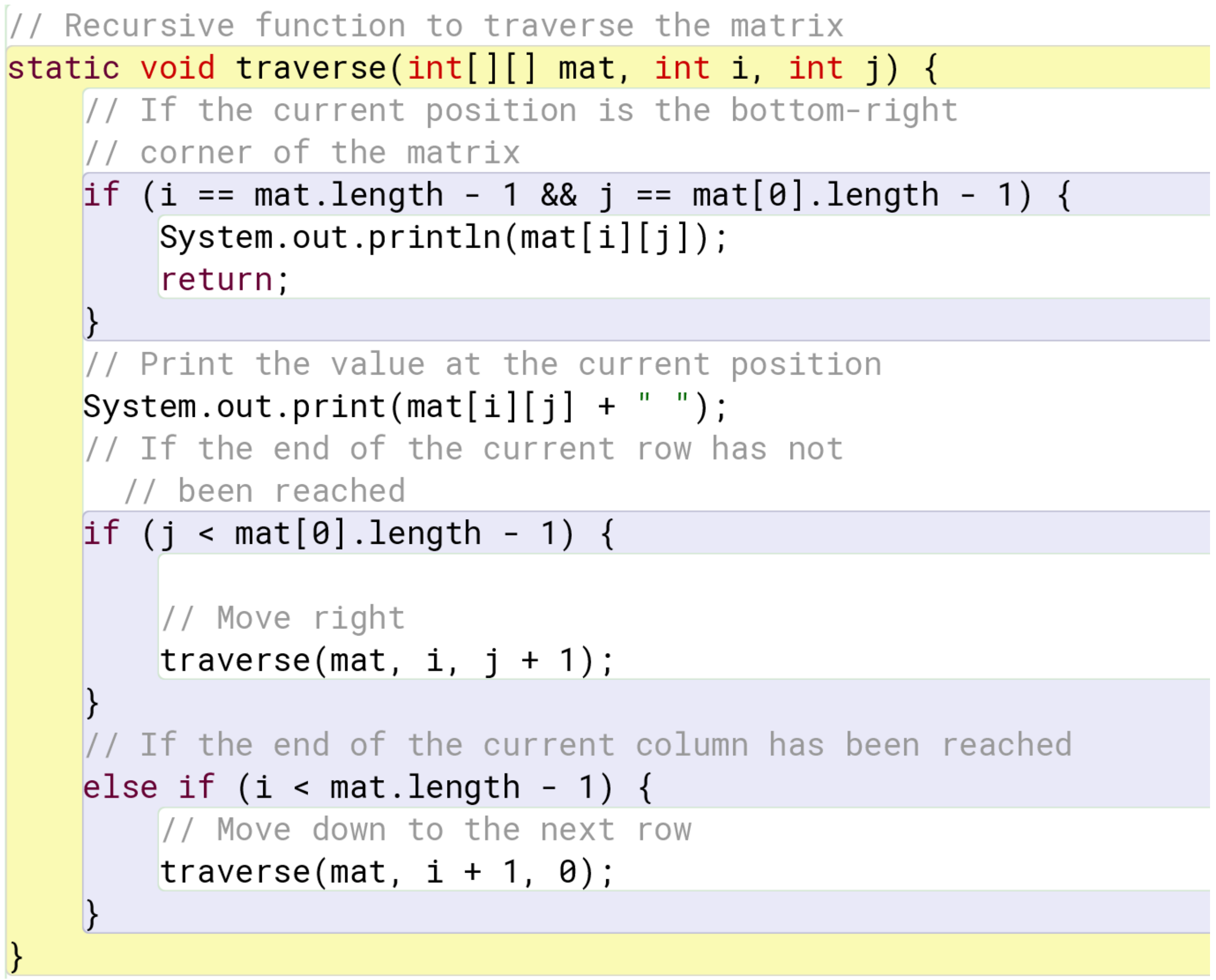
**Output :**
```
1 2 3 4 5 6 7 8 9
```

## Traverse a given Matrix using Recursion

**Input:**
```
int[][] mat = {{11, 12, 13},
               {14, 15, 16},
               {17, 18, 19}};
```

**Output:**
```
11 12 13 14 15 16 17 18 19
```

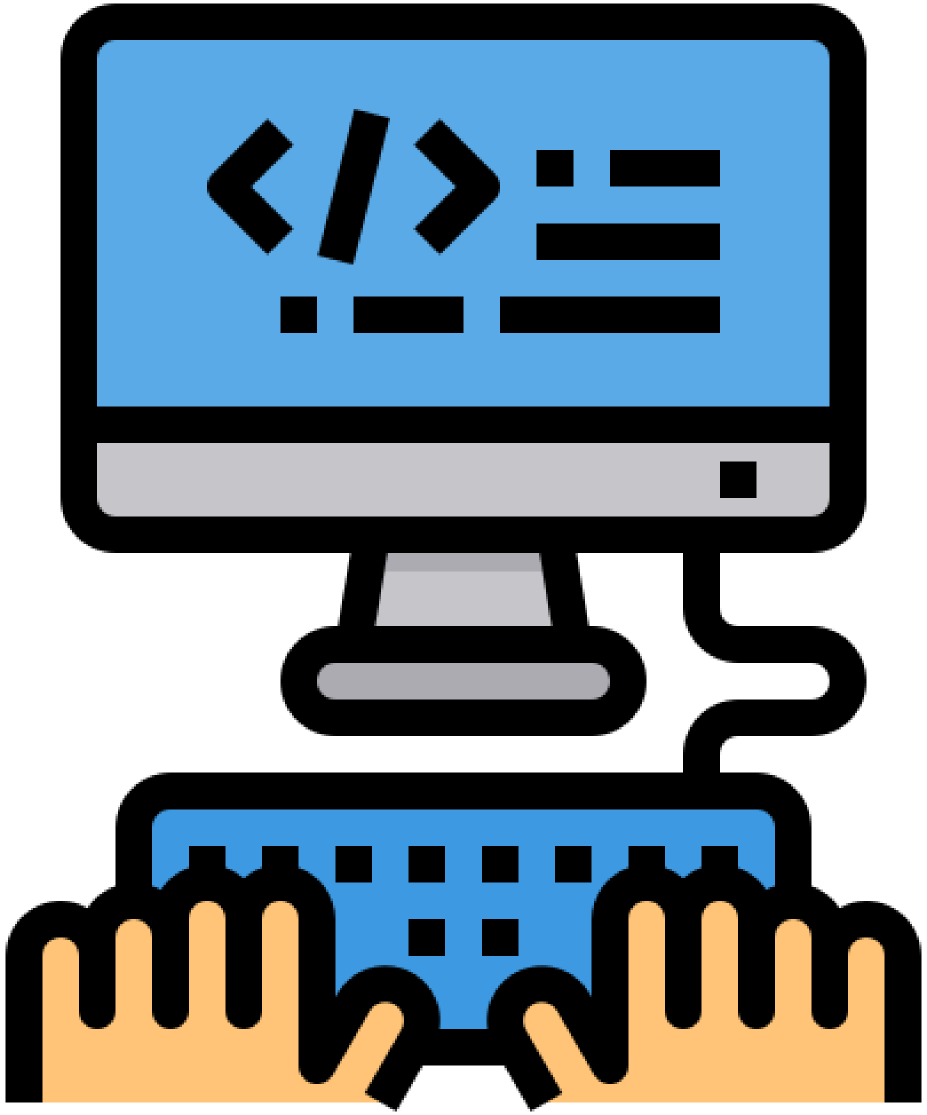
# Recursive Traversal Over 2D Array



```
// Recursive function to traverse the matrix
static void traverse(int[][] mat, int i, int j) {
    // If the current position is the bottom-right
    // corner of the matrix
    if (i == mat.length - 1 && j == mat[0].length - 1) {
        System.out.println(mat[i][j]);
        return;
    }
    // Print the value at the current position
    System.out.print(mat[i][j] + " ");
    // If the end of the current row has not
      // been reached
    if (j < mat[0].length - 1) {

        // Move right
        traverse(mat, i, j + 1);
    }
    // If the end of the current column has been reached
    else if (i < mat.length - 1) {
        // Move down to the next row
        traverse(mat, i + 1, 0);
    }
}
```

## Top Level: **GfG.java**

```java
public static void main(String[] args) {
    int[][] mat = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
    traverse(mat, 0, 0);
}
```



# Recursive Paradigm (Divide and Conquer, DP)

Section 6

## Divide and Conquer Algorithm

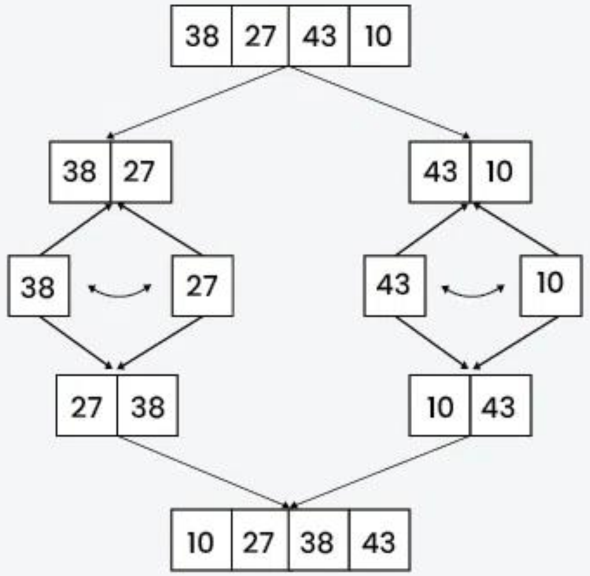
Divide and Conquer algorithm is a problem-solving strategy that involves.

- **Divide :** Break the given problem into smaller non-overlapping problems.
- **Conquer :** Solve Smaller Problems
- **Combine :** Use the Solutions of Smaller Problems to find the overall result.

Examples of Divide and Conquer are Merge Sort, Quick Sort, Binary Search and Closest Pair of Points.

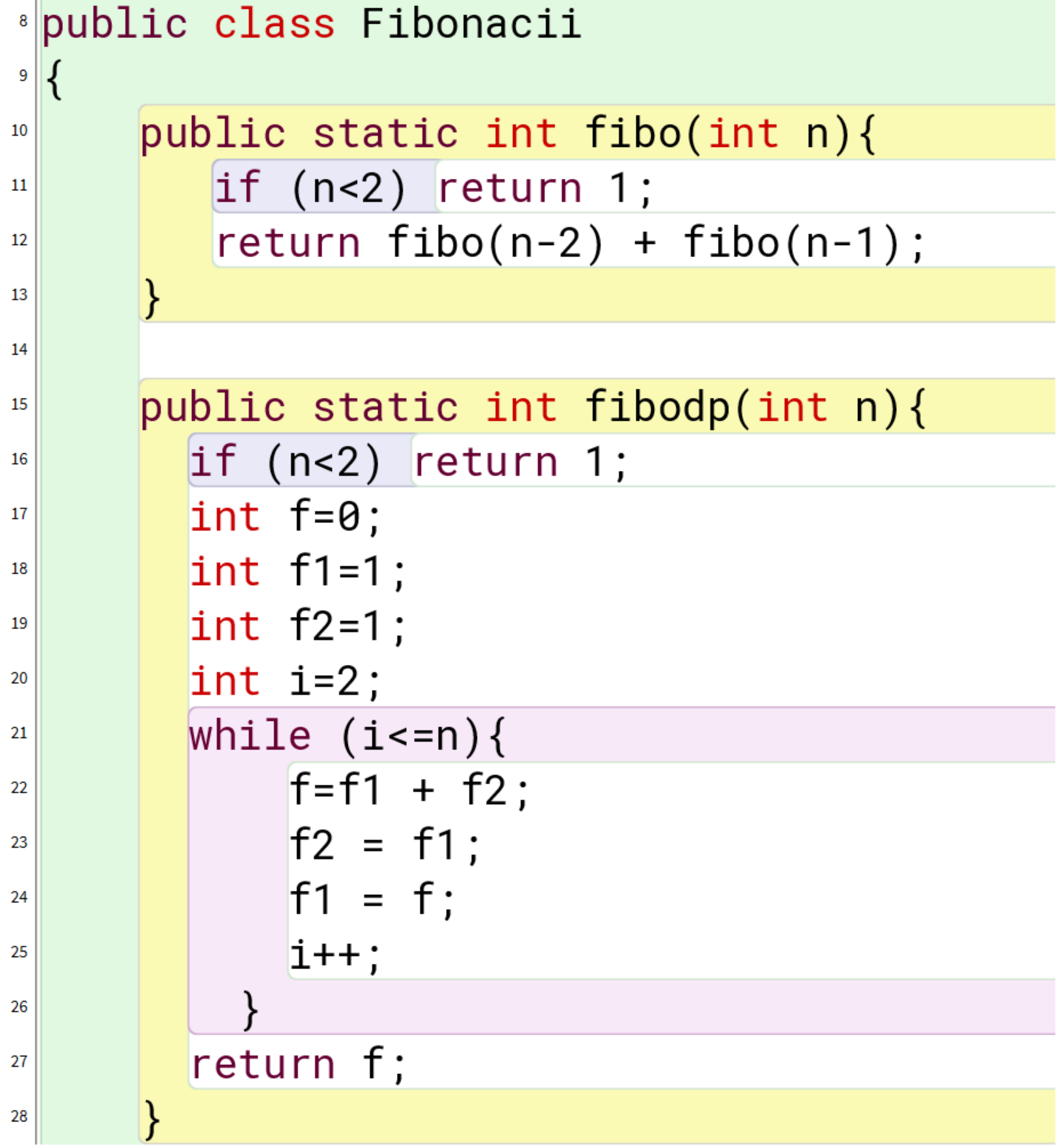
## Merge Sort

Divide and Conquer



# Dynamic Programming or DP

- Dynamic Programming is an algorithmic technique with the following properties.
    - It is mainly an **optimization** over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using **Dynamic Programming**.
    - The idea is to simply store the results of subproblems so that we do not have to re-compute them when needed later. **This simple optimization typically reduces time complexities from exponential to polynomial**.
    - Some popular problems solved using Dynamic Programming are **Fibonacci Numbers**, Diff Utility (Longest Common Subsequence), Bellman–Ford Shortest Path, Floyd Warshall, Edit Distance and Matrix Chain Multiplication.



```java
public class Fibonacii
{
    public static int fibo(int n){
        if (n<2) return 1;
        return fibo(n-2) + fibo(n-1);
    }

    public static int fibodp(int n){
      if (n<2) return 1;
      int f=0;
      int f1=1;
      int f2=1;
      int i=2;
      while (i<=n){
          f=f1 + f2;
          f2 = f1;
          f1 = f;
          i++;
        }
      return f;
    }
```

## Top-Down Approach (Memoization):

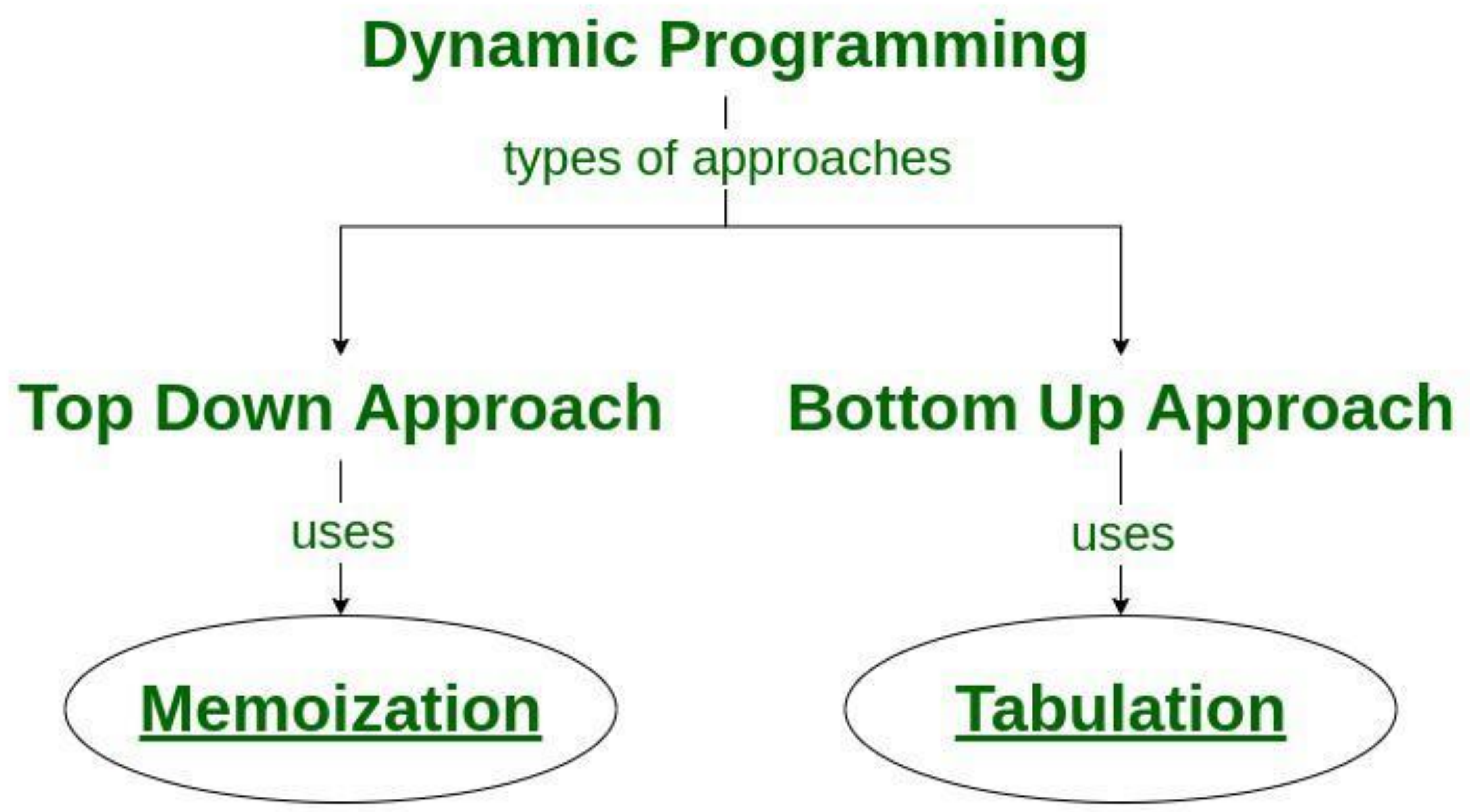
In the top-down approach, also known as **memoization**, we keep the solution recursive and add a memoization table to avoid repeated calls of same subproblems.

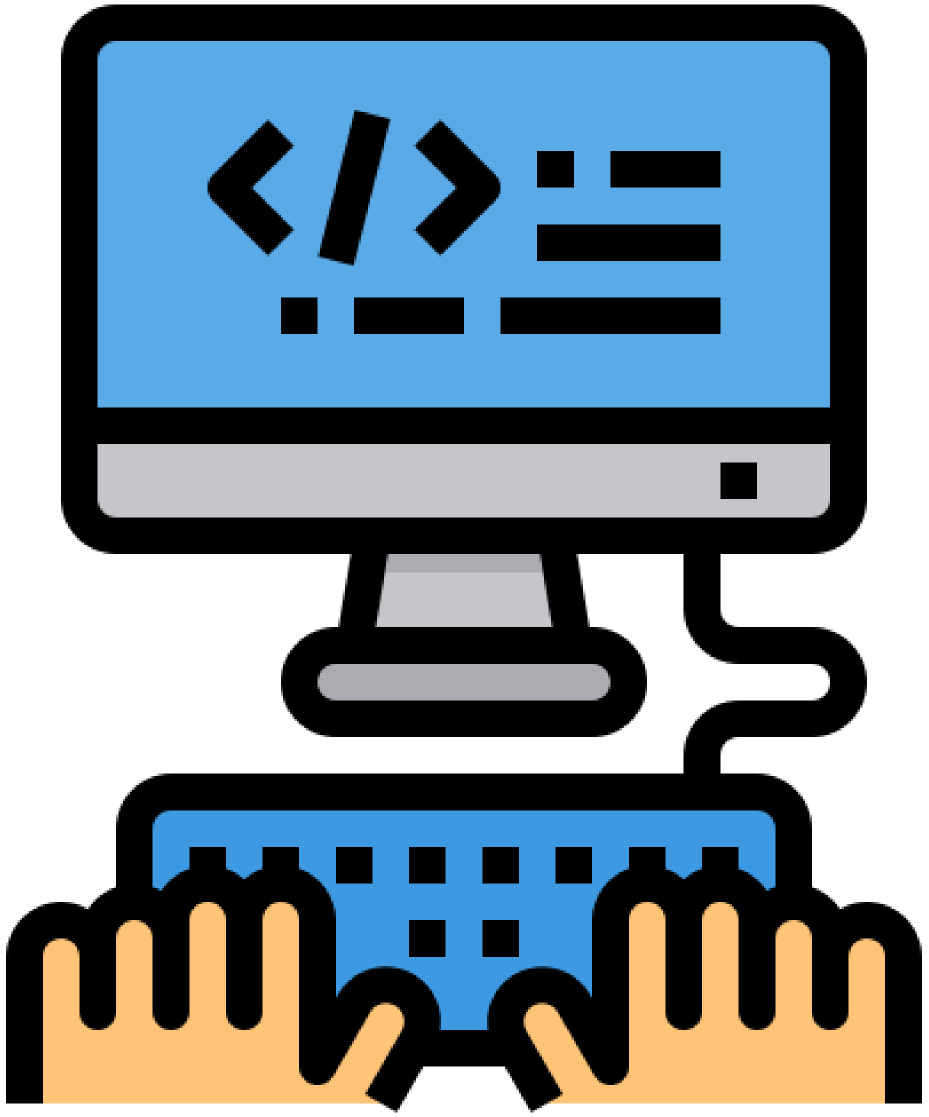
- Before making any recursive call, we first check if the memoization table already has solution for it.
- After the recursive call is over, we store the solution in the memoization table.

## Bottom-Up Approach (Tabulation):

In the bottom-up approach, also known as **tabulation**, we start with the smallest subproblems and gradually build up to the final solution.

- We write an iterative solution (avoid recursion overhead) and build the solution in bottom-up manner.
- We use a dp table where we first fill the solution for base cases and then fill the remaining entries of the table using recursive formula.
- We only use recursive formula on table entries and do not make recursive calls.


Dynamic Programming
types of approaches
Top Down Approach
Bottom Up Approach
uses
uses
Memoization
Tabulation



## Recursion versus Iteration

Section 7

## Summary

- Tail recursion will improve program performance.
- All iterative function can be written into recursive function and vice versa.
- A program can be written into several different ways.  Which is the best way to write it is an algorithm analysis problem.