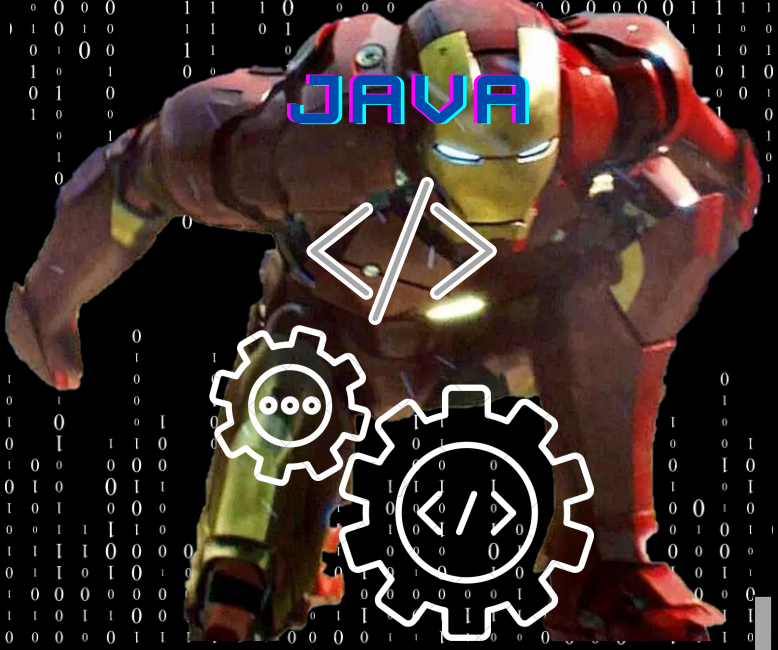
2023

Java

DSA
PART-2

OBZTRON

JAVA



(THIRD PART)

T.I.M. HAMIM

Index: DSA < Part - 2 >

1.Maths for DSA

2.Recursion

3.Searching using Recursion

4.Sorting using Recursion

5.Pattern using Recursion

6.Recursion Subset, Subsequence, String

7.Backtracking

8.LinkedList

9.----------

10.----------

11.----------

12.----------

1. Maths for DSA

Introduction :

- **Number System :**

The number system is a mathematical notation for representing numbers. It encompasses various types of numbers and the rules for performing arithmetic operations on them. There are several commonly used number systems, including:

- **Decimal Number System (Base-10):** This is the most familiar number system, where each digit can take on values from 0 to 9. It is also known as the base-10 system because it is based on powers of 10. The rightmost digit represents ones, the next digit to the left represents tens, and so on.

- **Binary Number System (Base-2):** In this system, each digit can take on values of either 0 or 1. It is widely used in computer science and digital electronics, as it directly corresponds to the on-off states of electronic devices. Each digit represents powers of 2.

- **Octal Number System (Base-8):** This system uses digits from 0 to 7. Each digit represents powers of 8. Octal numbers are often used in computer programming for representing groups of three binary digits.

- **Hexadecimal Number System (Base-16):** This system uses digits 0-9 and letters A-F to represent values from 0 to 15. Hexadecimal is also commonly used in computing to represent binary-coded values in a more compact and human-readable format.

Each of these number systems follows similar rules for addition, subtraction, multiplication, and division, although the representations of numbers and the mechanics of carrying and borrowing differ due to the varying bases.

Converting between number systems can be done using a process of division and remainder, where you repeatedly divide the number by the base and record the remainders. This gives you the digits in the desired base. For example, converting a decimal number to binary involves repeatedly dividing by 2 and recording the remainders.

Number systems are a fundamental concept in mathematics and play a crucial role in various fields such as computer science, engineering, and everyday life.

- **Converting Decimal to Binary :**

```
import java.util.Scanner;

public class Minian {
   public static void main(String[] args) {
      Scanner scan = new Scanner(System.in);
      int a, rem, k, s = 0;
      System.out.print("Enter decimal number : ");
      a = scan.nextInt();
      for (k = 1; a > 0; a = a / 2, k = k * 10) {
         rem = a % 2;
         s = s + rem * k;
      }
      System.out.printf("\nBinary form = %d", s);
      scan.close();
   }
}
```

Output:

```
Enter decimal number : 10

Binary form = 1010
```

- **Converting Decimal to Octal :**

```
import java.util.Scanner;

public class Minian {
  public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    int a, rem, k, s = 0;
    System.out.print("Enter a decimal number: ");
    a = scan.nextInt();
    for (k = 1; a > 0; a = a / 8, k = k * 10) {
      rem = a % 8;
      s = s + rem * k;
    }
    System.out.print("\nOctal form = "+ s);
    scan.close();
  }
}
```

Output:

```
Enter a decimal number: 10

Binary form = 12
```

- **Converting Decimal to Hexadecimal :**

```
import java.util.Scanner;

public class Test {
  public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    int num;
    System.out.print("Enter a decimal number: ");
    num = scan.nextInt();

    char hex[] = new char[50];
    int i;

    for (i = 0; num > 0; num /= 16, i++) {
      int rem = num % 16;

      if (rem >= 0 && rem <= 9) {
        hex[i] = (char)(rem + 48);
      } else if (rem >= 10 && rem <= 15) {
        hex[i] = (char)(rem + 55);
      }
    }

    System.out.printf("Hexadecimal: ");
    for (i = i - 1; i >= 0; i--) {
      System.out.printf("%c", hex[i]);
    }
  }

}
```

or,

```
import java.util.Scanner;

public class Test {
   public static void main(String[] args) {
      Scanner scan = new Scanner(System.in);
      int num;
      System.out.print("Enter a decimal number: ");
      num = scan.nextInt();

      char hex[] = new char[50];
      int i;

      for (i = 0; num > 0; num /= 16, i++) {
         int rem = num % 16;

         if (rem >= 0 && rem <= 9) {
            hex[i] = (char)(rem + 48);
         } else if (rem == 10)
         {
            hex[i] = 'A';
         }
         else if (rem == 11)
         {
            hex[i] = 'B';
         }
         else if (rem == 12)
         {
            hex[i] = 'C';
         }
```

```
            else if (rem == 13)
            {
                hex[i] = 'D';
            }
            else if (rem == 14)
            {
                hex[i] = 'E';
            }
            else if (rem == 15)
            {
                hex[i] = 'F';
            }
        }

        System.out.printf("Hexadecimal: ");
        for (i = i - 1; i >= 0; i--) {
            System.out.printf("%c", hex[i]);
        }
    }

}
```

Output:

Enter a decimal number: 122
Hexadecimal: 7A

- **Bit-manipulation :**

Introduction :

- **Make a program that will check whether a number is odd or not.**

```
public class Test {
   public static void main(String[] args) {
      int n = 67;
      System.out.println(isOdd(n));
   }

   public static boolean isOdd(int n){
      return (n & 1) == 1;
   }

}
```

Output:
true

- **Make a program where you are given an integer array. In the array, all elements are twice except one element and you need to find the element.**

```
public class Test {
  public static void main(String[] args) {
    int[] arr = {2, 3, 3, 4, 2, 6, 4};
    System.out.println(ans(arr));
  }

  public static int ans(int[] arr){
    int unique = 0;
    for (int i : arr) {
      unique ^= i;
    }
    return unique;
  }

}
```

Output:
6

- **Make a program where you are given an integer array. In the array, all elements are twice but in positive and negative numberexcept one element and you need to find the element.**

```
public class Test {
  public static void main(String[] args) {
    int[] arr = {2, -3, 3, 4, -2, 6, -4};
    System.out.println(ans(arr));
  }

  public static int ans(int[] arr){
    int unique = 0;
    for (int i : arr) {
      unique += i;
    }
    return unique;
  }

}
```

Output:
6

- **Find i no. bit of a number.**

```
public class Test {
  public static void main(String[] args) {
    int a = 7;
    System.out.println(ans(7, 2));
  }

  public static boolean ans(int a, int b){
    int mask = 1;
    return (a & mask << (b-1)) !=0 ;
  }

}
```

Output:
true

- **Set the ith no. bit into 1 of a number.**

```
public class Test {
  public static void main(String[] args) {
    int a = 7;
    System.out.println(ans(9, 2));
  }

  public static int ans(int a, int b){
    int mask = 1;
    return a | mask << (b-1) ;
  }

}
```

Output:
11

- **Reset the ith no. bit of a number. It means if ith no. bit was 1 then it would be 0 and 0 would be 0.**

```
public class Test {
  public static void main(String[] args) {
    int a = 7;
    System.out.println(ans(7, 2));
  }

  public static int ans(int a, int b){
    int mask = 1;
    return a & (~ (mask << (b-1))) ;
  }

}
```

Output:
5

2. Recursion

Recursion is a programming technique in which a function calls itself in order to solve a problem. Recursive functions can be used to solve problems that can be broken down into smaller subproblems of the same type. When writing recursive functions, it's important to define base cases that specify when the recursion should stop and when the function should start returning values.

- It helps us in solving bigger/ complex problems in a simple way.
- You can convert recursive solution into iteration and vice versa.
- Space complexity is not constant because of recursive calls.
- It helps us in breaking down bigger problems into smaller problems.

when to use Recursion :

- Identify if you can break the problem into smaller problems.
- Write the recursive relation if needed.
- Draw the recursive tree.
- About the tree :

1. See the flow of functions, how they are getting in stack.
2. Identify and focus on left tree walls and right walls.

- See how the values and what types of values are returned at each step.
- See where the function call will come out of the main function
- Make sure to return the result of a function call 0f the return type

Variables using recursion:

1. Arguments
2. Return type
3. Body of the function

Types of recurrence relation:

1. Linear recurrence relation (Fibonacci number),
2. Divide and conquer recurrence relation (Binary search)
   (Search space reduce by a factor)

1. Linear recurrence relation:
    1. Factorial
    2. Fibonacci Number

2. Divide and Conquer:
    1. Binary Search
    2. Finding Maximum and Minimum
    3. Merge Sort
    4. Quick Sort
    5. Strassem's Matrix Multiplication

- **Iteration vs Recursion :**

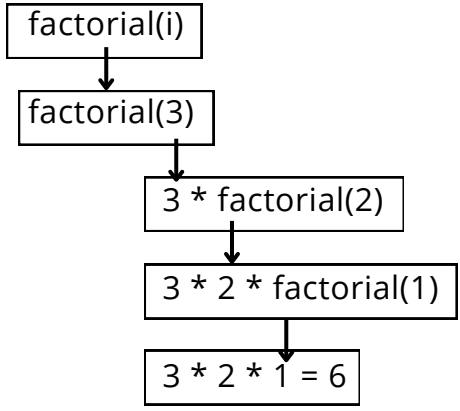
Recursion	**Iteration**
1.Use selection structure(if, else, swtich)	1.Use repetition structure(for, while, do while)
2.Terminates when base case is satisfied.	2.Terminates whene condition fails.
3.It's slow	3.It's fast
4.Code is smaller	4.Code is bigger

- **Make a program which will calculates factorial for a given number using a recursive function.**

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

  public static void main(String[] args) {
    int num = 5;
    int result = factorial(num);
    System.out.println("Factorial of " + num + " is " +
result);
  }
}
```

Output:
Factorial of 3 is 6

Here,



Here,

- This is called tail recursion
- This is the last function call

- **What is the output of the following Java program fragment:**

```
public class Test {
  public static void main(String[] args) {
    System.out.println(message(5));;
  }

  static String message (int a){
    if(a==1){
      return "Hamim Jim ";

    }

    return "Hamim Jim \n" + message(a-1);
  }
}
```

Output:
Hamim Jim
Hamim Jim
Hamim Jim
Hamim Jim
Hamim Jim

- **What is the output of the following Java program fragment:**

```
public class Test {
  public static void main(String[] args) {
    System.out.println(message(5));
  }

  static String message (int a){
    if(a==1){
      return 1+"";
    }

    return a +"\n" + message(a-1);
  }
}
```

Output:

5
4
3
2
1

- **What is the output of the following Java program fragment:**

```
public class Test {
  public static void main(String[] args) {
    System.out.println(message(1));
  }

  static String message(int a) {
    if (a == 5) {
      return 5 + "";
    }

    return a + "\n" + message(a + 1);
  }
}
```

Output:

1
2
3
4
5

- **What is the output of the following Java program fragment:**

```
public class Test {
  public static void main(String[] args) {
    System.out.println(message(1,5));
  }

  static String message (int a, int b){
    if(a==b){
      return b+"";
    }

    return a +"\n" + message(a+1,b);
  }
}
```

Output:

1
2
3
4
5

- **What is the output of the following Java program fragment:**

```
public class Test {
  public static void main(String[] args) {
    print(1);
  }

  static void print(int a) {
    if(a==5){
      System.out.println(a);
      return;
    }

    System.out.println(a);
    print(a+1);
  }
}
```

Output:
1
2
3
4
5

Here,
return is used for break the Recursion.

- **What is the output of the following Java program fragment:**

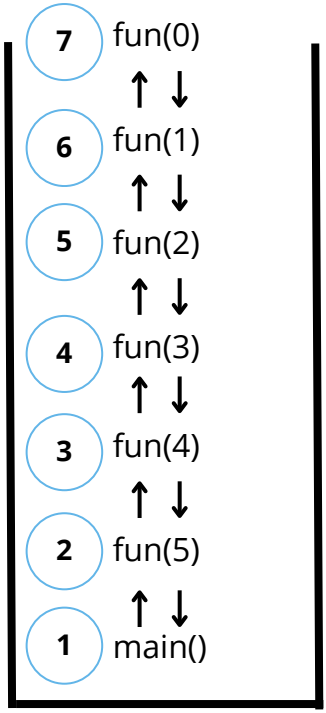
```
public class Test {
 public static void main(String[] args) {
  fun(5);
 }

 static void fun (int n){
  if(n==0){
   return;
  }

  System.out.println(n);
  fun(n-1);
 }
}
```

Output:

5
4
3
2
1

Here,
return is used for break the Recursion.

Here,



Stack memory

- **What is the output of the following Java program fragment:**

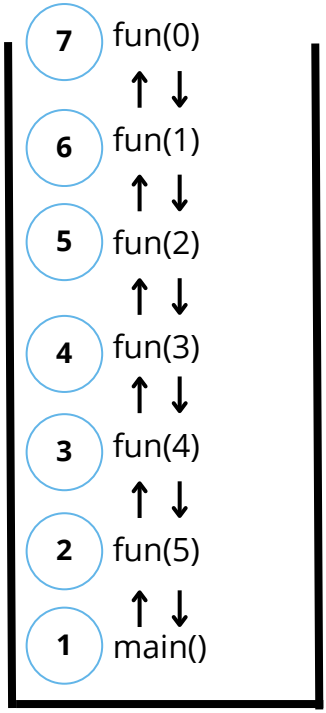
```
public class Test {
 public static void main(String[] args) {
  funRev(5);
 }

 static void funRev(int n){
  if(n==0){
   return;
  }

  funRev(n-1);
  System.out.println(n);
 }
}
```

Output:

1
2
3
4
5

Here,
return is used for break the Recursion.

Here,



Stack memory

- **Make a program that will show the sum of the digits of any number.**

```
public class Test {
  public static void main(String[] args) {
    int ans = sum(1342);
    System.out.println(ans);
  }

  static int sum(int n){
    if(n==0){
      return 0;
    }

    return (n%10) + sum(n/10);
  }
}
```

Output:
10

- **Make a program that will show the multiplication of the digits of any number.**

```
public class Test {
 public static void main(String[] args) {
  int ans = sum(55);
  System.out.println(ans);
 }

 static int sum(int n){
  if(n%10==n){
   return n;
  }

  return (n%10) * sum(n/10);
 }
}
```

Output:
55

- **Make a program that will reverse a number.**

```
import java.util.Scanner;

public class Test {

  // Recursive function to reverse a number
  public static int reverseNumber(int num, int
reversedNum) {
    if (num == 0) {
      return reversedNum;
    }

    int lastDigit = num % 10;
    reversedNum = reversedNum * 10 + lastDigit;

    return reverseNumber(num / 10, reversedNum);
  }

  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter a number: ");
    int num = scanner.nextInt();

    int reversedNum = reverseNumber(num, 0);
    System.out.println("Reversed number: " +
reversedNum);
  }
}
```

or,

```
import java.util.Scanner;

public class Test {
   // Recursive function to reverse a number
   public static int reverseNumber(int num) {
      return reverse(num, 0);
   }

   public static int reverse(int num, int reversedNum){
      if (num == 0) {
         return reversedNum;
      }

      int lastDigit = num % 10;
      reversedNum = reversedNum * 10 + lastDigit;

      return reverse(num / 10, reversedNum);
   }

   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter a number: ");
      int num = scanner.nextInt();

      int reversedNum = reverseNumber(num);
      System.out.println("Reversed number: " +
reversedNum);
   }
}
```

Output:

Enter a number: 1824
Reversed number: 4281

- **Make a program that will reverse a number.**

```
public class Test {

   static int sum = 0;

   public static void reverseNumber(int num) {
      if (num == 0) {
         return;
      }
      int rem = num % 10;
      sum = sum * 10 + rem;
      reverseNumber(num / 10);
   }

   public static void main(String[] args) {
      reverseNumber(1234);
      System.out.println(sum);
   }
}
```

or,

```
public class Test {

   public static int reverseNumber(int num) {
      int digits = (int)(Math.log10(num));
      return helper(num, digits);
   }

   public static int helper(int n, int digits){
      if(n%10==n){
         return n;
      }
      int rem = n%10;
      return rem * (int) (Math.pow(10, digits)) +
helper(n/10, digits-1);
   }

   public static void main(String[] args) {

      System.out.println(reverseNumber(1234));
   }
}
```

Output:

4321

- **Make a program that will count number of zeros(0) of a number.**

```
public class Test {

   public static int countZero(int num) {
      return helper(num, 0);
   }

   public static int helper(int n, int c){
      if(n==0){
         return c;
      }
      int rem = n%10;
      if(rem==0){
         return helper(n/10, c+1);
      }
      return helper(n/10, c);
   }

   public static void main(String[] args) {

      System.out.println(countZero(12049040));
   }
}
```

Output:
3

- **Given an integer num, return the number of steps to reduce it to zero.**
  In one step, if the current number is even, you have to divide it by 2 , otherwise, you have to subtract 1 from it.

```
public class Test {

  public static int numberOfSteps(int num) {
    return helper(num, 0);
  }

  public static int helper(int num, int steps){
    if(num==0){
      return steps;
    }
    if(num%10==0){
      return helper(num/10, steps+1);
    }
    return helper(num-1, steps+1);
  }

  public static void main(String[] args) {

    System.out.println(numberOfSteps(14));
  }
}
```

Output:

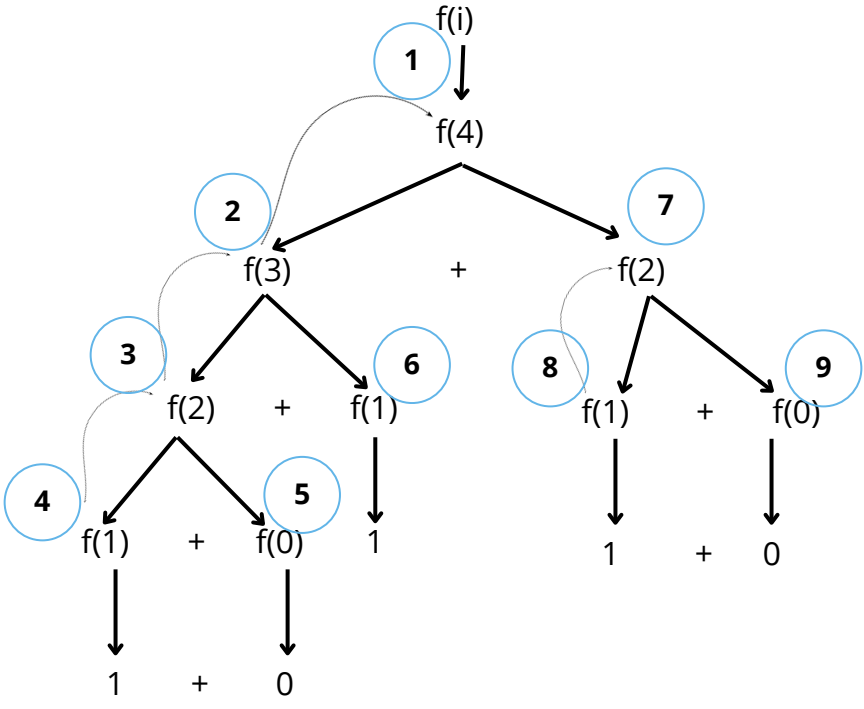
6

- **Make a program which will generates Fibonacci series for a given number using a recursive function**

```
public class Test {
  public static int fibonacci(int i) {
    if (i < 2) {
      return i;
    }

    return fibonacci(i - 1) + fibonacci(i - 2);
  }

  public static void main(String[] args) {
    System.out.println(fibonacci(4));
  }
}
```

Output:
3

Here,



- **Make a program which will generates Fibonacci series for a given number using a recursive function**

```
public class Test {
 public static int fibonacci(int i) {
  if (i == 0) {
   return 0;
  }

  if (i == 1) {
   return 1;
  }
  return fibonacci(i - 1) + fibonacci(i - 2);
 }

 public static void main(String[] args) {
  System.out.println(fibonacci(5));
 }
}
```

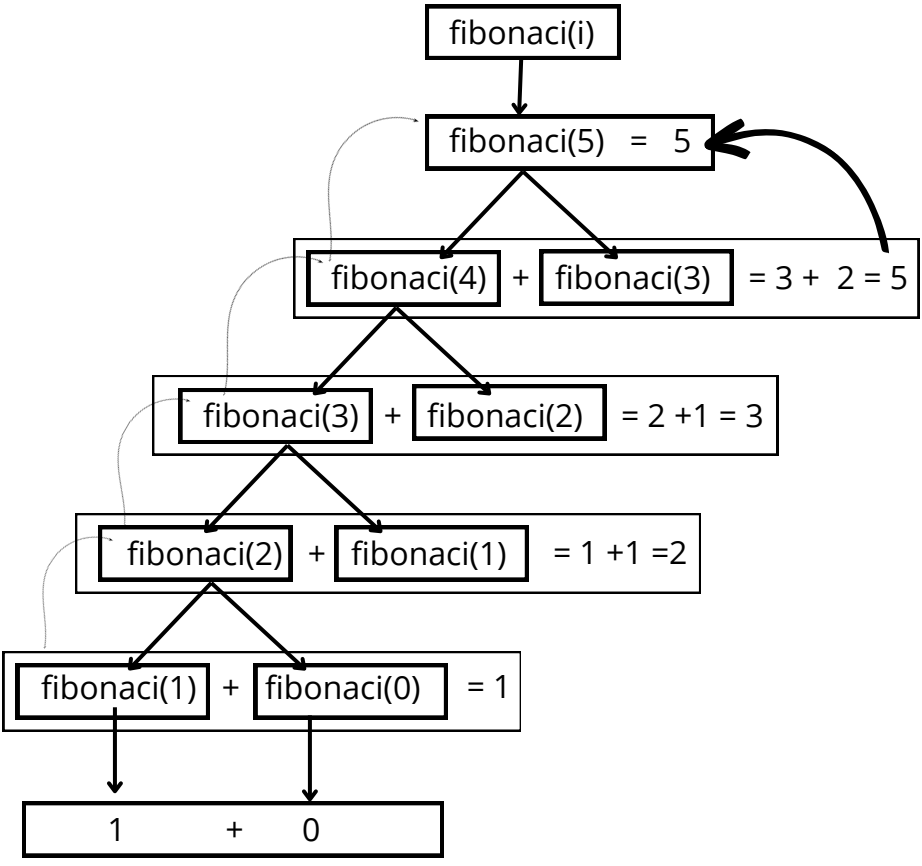
or,

```
public class Test {
 public static int fibonacci(int i) {
  if (i < 2) {
   return i;
  }

  return fibonacci(i - 1) + fibonacci(i - 2);
 }

 public static void main(String[] args) {
  System.out.println(fibonacci(5));
 }
}
```

Output:

5

Here,


fibonaci(i)
fibonaci(5) = 5
fibonaci(4) + fibonaci(3) = 3 + 2 = 5
fibonaci(3) + fibonaci(2) = 2 +1 = 3
fibonaci(2) + fibonaci(1) = 1 +1 =2
fibonaci(1) + fibonaci(0) = 1
1 + 0

- **Make a program which will generates Fibonacci series for a given number using a recursive function**

```
public class Test {
  public static int fibonacci(int i) {
    if (i == 0) {
      return 0;
    }
    if (i == 1) {
      return 1;
    }
    return fibonacci(i - 1) + fibonacci(i - 2);
  }

  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
      System.out.print(fibonacci(i) + "\t");
    }
  }
}
```

Output:

```
0	1	1	2	3	5	8	13	21	34
```

- **Make a program which will generates Fibonacci series for a given number using a recursive function**

```
public class Test {
  public static int fibonacci(int i) {
    if (i < 2) {
      return i;
    }

    return fibonacci(i - 1) + fibonacci(i - 2);
  }

  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
      System.out.print(fibonacci(i) + "\t");
    }
  }
}
```

Output:

```
0	1	1	2	3	5	8	13	21	34
```

- **Make a binary search function using recursion.**

```
public class Test {
   public static int binarySearch(int[] arr, int target, int s,
int e) {
      if (s > e) {
         return -1;
      }
      int m = s + (e - s) / 2;
      if (arr[m] == target) {
         return m;
      }
      if (target < arr[m]) {
         return binarySearch(arr, target, s, m - 1);
      }
      return binarySearch(arr, target, m + 1, e);

   }

   public static void main(String[] args) {
      int[] arr = { 1, 2, 3, 4, 55, 66, 78 };
      int target = 4;
      System.out.println(binarySearch(arr, target, 0,
arr.length));
   }
}
```

Output:

3

Here,

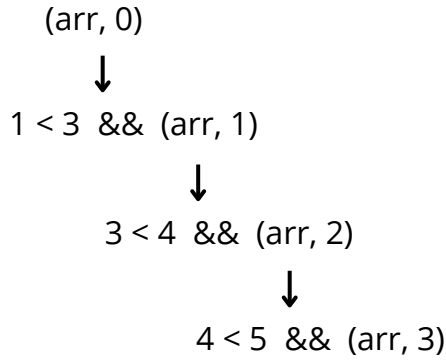
- **Find if an array is sorted or not.**

```
public class Test {

  static boolean arraySorted(int[] arr){
    return helper(arr, 0);
  }

  static boolean helper(int[] arr, int index){
    if(arr.length-1==index){
      return true;
    }
    return (arr[index] < arr[index+1]) && helper(arr,
index+1);
  }

  public static void main(String[] args) {
    int[] arr = {1,3,4,5};
    System.out.println(arraySorted(arr));
  }
}
```

Output:
true

Here,

arr = {1, 3, 4, 5}

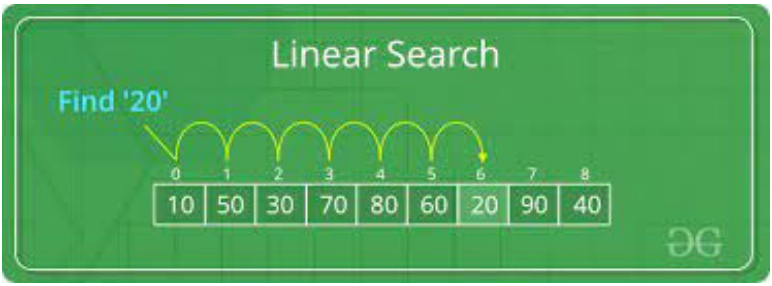


3. Searching using Recursion

1. Linear Search
2. Binary Search
3. Modified Binary Search
4. Binary Search on 2D Arrays

1. Linear Search :

Linear search is a simple searching algorithm that iterates through an array or a list to find a specific element. It sequentially checks each element until the target element is found or until the end of the list is reached.



- **Linear search using recursion.**

```
public class Test {

  static boolean linearSearch(int[] arr, int target){
    return helper(arr, target, 0);
  }

  static boolean helper(int[] arr, int target, int index)
{
    if(arr.length-1 < index){
      return false;
    }
    if(arr[index] == target){
      return true;
    }
    return helper(arr, target, index + 1);
  }

  public static void main(String[] args) {
    int[] arr = {7,3,4,9,5};
    int target = 4;
    System.out.println(linearSearch(arr, target));
  }
}
```

or,

```
public class Test {

   static boolean arraySorted(int[] arr, int target){
      return helper(arr, target, 0);
   }

   static boolean helper(int[] arr, int target, int
index){
      if(arr.length-1 < index){
         return false;
      }

      return arr[index] == target || helper(arr,
target, index + 1);
   }

   public static void main(String[] args) {
      int[] arr = {7,3,4,9,5};
      int target = 4;
      System.out.println(arraySorted(arr, target));
   }
}
```

Output:

true

- **Linear search using recursion.**
  Return type integer(index of target number).
  < Left index>

```
public class Test {

  static int LinearSearch(int[] arr, int target){
    return findIndex(arr, target, 0);
  }

  static int findIndex(int[] arr, int target, int index){
    if(arr.length - 1 < index){
      return -1;
    }
    if(arr[index] == target){
      return index;
    } else{
      return findIndex(arr, target, index + 1);
    }
  }

  public static void main(String[] args) {
    int[] arr = {7,3,4,9,4,5};
    int target = 4;
    System.out.println(LinearSearch(arr, target));
  }
}
```

Output:

2

- **Linear search using recursion.**
  Return type integer(index of target number).
  <Right index>

```
public class Test {

 static int LinearSearch(int[] arr, int target){
  return findIndex(arr, target, arr.length-1);
 }

 static int findIndex(int[] arr, int target, int index){
  if(index == -1){
   return -1;
  }
  if(arr[index] == target){
   return index;
  } else{
   return findIndex(arr, target, index - 1);
  }
 }

 public static void main(String[] args) {
  int[] arr = {7,3,4,9,4,5};
  int target = 4;
  System.out.println(LinearSearch(arr, target));
 }
}
```

Output:

4

- **Make a function that will take an array and that will search a target element from the array. After finding the elements that are equal to the target element from the array it will return the indexes of them by an array.**

```
import java.util.ArrayList;
import java.util.List;

public class Test {

  public static int[] searchIndexes(int[] inputArray, int
target) {
    List<Integer> indexList = new ArrayList<>();
    return searchIndexesRecursive(inputArray, target, 0,
indexList);
  }

  public static int[] searchIndexesRecursive(int[]
inputArray, int target, int currentIndex, List<Integer>
indexList) {
    if (currentIndex >= inputArray.length) {
      int[] indexes = new int[indexList.size()];
      for (int i = 0; i < indexes.length; i++) {
        indexes[i] = indexList.get(i);
      }
      return indexes;
    }
```

```
 if (inputArray[currentIndex] == target) {
 indexList.add(currentIndex);
 }

 return searchIndexesRecursive(inputArray, target,
currentIndex + 1, indexList);
 }

 public static void main(String[] args) {
 int[] array = {4, 2, 6, 3, 2, 7, 2, 8};
 int target = 2;

 int[] indexes = searchIndexes(array, target);

 System.out.println("Target Element: " + target);
 System.out.print("Indexes of target element: ");
 for (int index : indexes) {
 System.out.print(index + " ");
 }
 System.out.println();
 }
}
```

Output:

Target Element: 2
Indexes of target element: [1, 4, 6]

or,

```
import java.util.ArrayList;
import java.util.List;

public class Test {

   public static List<Integer> searchIndexesRecursive(int[]
inputArray, int target, int currentIndex, List<Integer>
indexList) {
      if (currentIndex >= inputArray.length) {
         return indexList;
      }

      if (inputArray[currentIndex] == target) {
         indexList.add(currentIndex);
      }

      return searchIndexesRecursive(inputArray, target,
currentIndex + 1, indexList);
   }

   public static List<Integer> searchIndexes(int[]
inputArray, int target) {
      List<Integer> indexList = new ArrayList<>();
      return searchIndexesRecursive(inputArray, target, 0,
indexList);
   }

   public static void main(String[] args) {
      int[] array = {4, 2, 6, 3, 2, 7, 2, 8};
      int target = 2;
```

```
        List<Integer> indexes = searchIndexes(array, target);

        System.out.println("Target Element: " + target);
        System.out.print("Indexes of target element: ");
        for (int index : indexes) {
            System.out.print(index + " ");
        }
        System.out.println();
    }
}
```

Output:

Target Element: 2
Indexes of target element: 1 4 6

or,

```
import java.util.ArrayList;
import java.util.List;

public class Test {

   public static List<Integer> searchIndexesRecursive(int[]
inputArray, int target, int currentIndex, List<Integer>
indexList) {
      if (currentIndex >= inputArray.length) {
         return indexList;
      }

      if (inputArray[currentIndex] == target) {
         indexList.add(currentIndex);
      }

      return searchIndexesRecursive(inputArray, target,
currentIndex + 1, indexList);
   }

   public static List<Integer> searchIndexes(int[]
inputArray, int target) {
      List<Integer> indexList = new ArrayList<>();
      return searchIndexesRecursive(inputArray, target, 0,
indexList);
   }

   public static void main(String[] args) {
      int[] array = {4, 2, 6, 3, 2, 7, 2, 8};
      int target = 2;
```

```
        System.out.println("Target Element: " + target);
        System.out.print("Indexes of target element:
"+searchIndexes(array, target));
    }
}
```

Output:

```
Target Element: 2
Indexes of target element: [1, 4, 6]
```

or,

```
import java.util.ArrayList;
import java.util.List;

public class Test {

    public static ArrayList<Integer>
LinearSearchIndexes(int[] inputArray, int target, int
currentIndex) {
        ArrayList<Integer> indexList = new ArrayList<>();
        if (currentIndex == inputArray.length) {
            return indexList;
        }

        // This will contain answer for that function call
only
        if (inputArray[currentIndex] == target) {
            indexList.add(currentIndex);
        }

        ArrayList<Integer> ansFromBelowCalls =
LinearSearchIndexes(inputArray, target, currentIndex
+ 1);
        indexList.addAll(ansFromBelowCalls);
        return indexList;
    }

    public static void main(String[] args) {
        int[] array = {4, 2, 6, 3, 2, 7, 2, 8};
        int target = 2;
```

```
        System.out.println("Target Element: " + target);
        System.out.print("Indexes of target element:
"+LinearSearchIndexes(array, target, 0));
    }
}
```

Output:

Target Element: 2
Indexes of target element: [1, 4, 6]

or,

```
import java.util.ArrayList;

public class Test {

   public static ArrayList<Integer>
searchIndexesRecursive(int[] inputArray, int target, int
currentIndex, ArrayList<Integer> indexList) {
      if (currentIndex >= inputArray.length) {
         return indexList;
      }
      if (inputArray[currentIndex] == target) {
         indexList.add(currentIndex);
      }

      return searchIndexesRecursive(inputArray, target,
currentIndex + 1, indexList);
   }

   public static void main(String[] args) {
      int[] array = {4, 2, 6, 3, 2, 7, 2, 8};
      int target = 2;
      System.out.println("Target Element: " + target);
      System.out.print("Indexes of target element:
"+searchIndexesRecursive(array, target, 0, new
ArrayList<>()));
   }
}
```

Output:

```
Target Element: 2
Indexes of target element: [1, 4, 6]
```

or,

```
import java.util.ArrayList;

public class Test {
  public static ArrayList<Integer>
searchIndexesRecursive(int[] inputArray, int target, int
currentIndex, ArrayList<Integer> indexList) {
    if (currentIndex >= inputArray.length) {
      return indexList;
    }

    if (inputArray[currentIndex] == target) {
      indexList.add(currentIndex);
    }

    return searchIndexesRecursive(inputArray, target,
currentIndex + 1, indexList);
  }

  public static void main(String[] args) {
    int[] array = {4, 2, 6, 3, 2, 7, 2, 8};
    int target = 2;
    ArrayList<Integer> list = new ArrayList<>();
    System.out.println("Target Element: " + target);
    System.out.print("Indexes of target element:
"+searchIndexesRecursive(array, target, 0, list));
  }
}
```

Output:

```
Target Element: 2
Indexes of target element: [1, 4, 6]
```

or,

```
import java.util.ArrayList;
import java.util.List;

public class Test {

   static ArrayList<Integer> indexList = new ArrayList<>();

   public static void searchIndexes(int[] inputArray, int
target) {
      searchIndexesRecursive(inputArray, target, 0);
   }

   public static void searchIndexesRecursive(int[] arr, int
target, int index) {
      if(index == arr.length){
         return;
      }
      if(arr[index] == target){
         indexList.add(index);
      }
      searchIndexesRecursive(arr, target, index + 1);
   }

   public static void main(String[] args) {
      int[] array = {4, 2, 6, 3, 2, 7, 2, 8};
      int target = 2;
      System.out.println("Target : "+target);
      searchIndexes(array, target);
      System.out.println(indexList);
   }
}
```

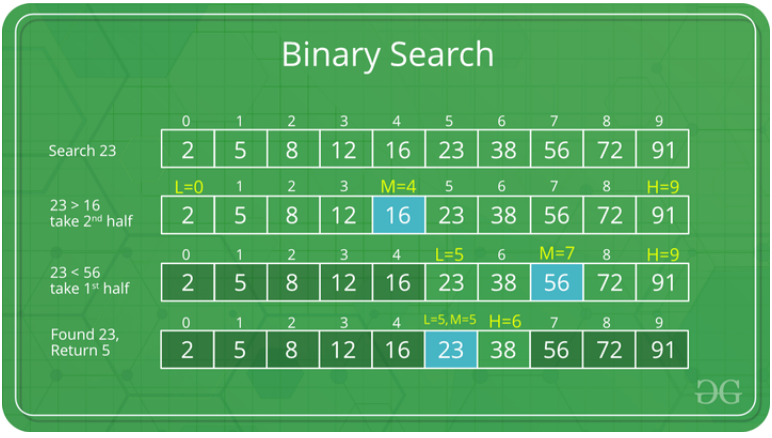
Output:

Target : 2

[1, 4, 6]

2. Binary Search :

Binary search is a search algorithm that finds the position of a target value within a sorted array. It works by repeatedly dividing the search space in half until the target value is found or it is determined that the target value does not exist in the array.



- **Binary search from sorted array.**

```
public class Test {
  public static int binarySearch(int[] arr, int target) {
    return binarySearch(arr, target, 0, arr.length - 1);
  }

  private static int binarySearch(int[] arr, int target,
int left, int right) {
    if (left <= right) {
      int mid = left + (right - left) / 2;

      if (arr[mid] == target) {
        return mid;
      } else if (arr[mid] < target) {
        return binarySearch(arr, target, mid + 1, right);
      } else {
        return binarySearch(arr, target, left, mid - 1);
      }
    }

    return -1;
  }

  public static void main(String[] args) {
    int[] arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int target = 6;
    int result = binarySearch(arr, target);
```

```
    if (result != -1) {
      System.out.println("Element " + target + " found at
index " + result);
    } else {
      System.out.println("Element " + target + " not found
in the array.");
    }
  }
}
```

Output:

Element 6 found at index 5

- **Binary search from sorted array.**

```
public class Test {
  public static int binarySearch(int[] arr, int target) {
    return binarySearch(arr, target, 0, arr.length - 1);
  }

  private static int binarySearch(int[] arr, int target,
int left, int right) {

    if (left > right) {
      return -1;
    }

    int mid = left + (right - left) / 2;

    if (arr[mid] == target) {
      return mid;
    } else if (arr[mid] < target) {
      return binarySearch(arr, target, mid + 1, right);
    } else {
      return binarySearch(arr, target, left, mid - 1);
    }

  }

  public static void main(String[] args) {
    int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int target = 69;
    int result = binarySearch(arr, target);
```

```
    if (result != -1) {
      System.out.println("Element " + target + " found at
index " + result);
    } else {
      System.out.println("Element " + target + " not
found in the array.");
    }
  }
}
```

Output:

Element 6 found at index 5

- **Binary search(First occurrence) from sorted array.**

```
public class Minian {
  static int binarySearch(int arr[], int left, int right,
int target) {
    if (right >= left) {
      int mid = left + (right - left) / 2;

      if (arr[mid] == target) {
        if (mid == 0 || arr[mid - 1] < target) {
          return mid;
        }
        else {
          return binarySearch(arr, left, mid - 1, target);
        }
      }
      if (arr[mid] > target) {
        return binarySearch(arr, left, mid - 1, target);
      }
      return binarySearch(arr, mid + 1, right, target);
    }

    return -1;
  }

  public static void main(String args[]) {
    int arr[] = {1, 2, 2, 2, 3, 4, 4, 5, 6};
    int target = 4;
    int n = arr.length;

    int result = binarySearch(arr, 0, n - 1, target);
```

```
 if (result != -1) {
 System.out.println("First occurrence of " + target +
" is at index " + result);
 } else {
 System.out.println(target + " not found in the
array");
 }
 }
}
```

Output:

First occurrence of 4 is at index 5

- **Binary search(Last occurrence) from sorted array.**

```
public class Minian {
  static int binarySearchLastOccurrence(int arr[], int
left, int right, int target) {
    if (right >= left) {
      int mid = left + (right - left) / 2;
      if (arr[mid] == target) {
        if (mid == right || arr[mid + 1] > target) {
          return mid;
        } else {
          return binarySearchLastOccurrence(arr, mid
+ 1, right, target);
        }
      }
      if (arr[mid] > target) {
        return binarySearchLastOccurrence(arr, left,
mid - 1, target);
      }
      return binarySearchLastOccurrence(arr, mid +
1, right, target);
    }
    return -1;
  }

  public static void main(String args[]) {
    int arr[] = {1, 2, 2, 2, 3, 4, 4, 5, 6};
    int target = 4;
    int n = arr.length;
```

```
 int result = binarySearchLastOccurrence(arr, 0, n -
1, target);

 if (result != -1) {
 System.out.println("Last occurrence of " + target +
" is at index " + result);
 } else {
 System.out.println(target + " not found in the
array");
 }
 }
}
```

Output:

Last occurrence of 4 is at index 6

- **Binary search(First and Last occurrence) from sorted array.**

```
import java.util.Arrays;

public class Minian {
  static int[] findFirstAndLastOccurrences(int arr[],
int left, int right, int target) {
    int[] result = {-1, -1};

    if (left <= right) {
      int mid = left + (right - left) / 2;

      if (arr[mid] == target) {
        if (mid == 0 || arr[mid - 1] < target) {
          result[0] = mid;
        }

        result[1] = mid;
        int[] leftResult =
findFirstAndLastOccurrences(arr, left, mid - 1,
target);
        int[] rightResult =
findFirstAndLastOccurrences(arr, mid + 1, right,
target);

        if (leftResult[0] != -1) {
          result[0] = leftResult[0];
        }

        if (rightResult[1] != -1) {
```

```
 result[1] = rightResult[1];
 }
 }
 else if (arr[mid] > target) {
 return findFirstAndLastOccurrences(arr, left, mid -
1, target);
 }
 else {
 return findFirstAndLastOccurrences(arr, mid + 1,
right, target);
 }
 }

 return result;
 }

public static void main(String args[]) {
    int arr[] = {1, 2, 2, 2, 3, 4, 4, 5, 6};
    int target = 6;
    int n = arr.length;
    int[] result = findFirstAndLastOccurrences(arr, 0,
n - 1, target);
    System.out.println("Target : "+target);
    if (result[0] != -1) {
      System.out.println("Indexes of first and last
occurrence : "+Arrays.toString(result));
    } else {
      System.out.println(target + " not found in the
array");
    }
  }
}
```

Output:

Target : 6

Indexes of first and last occurrence : [8, 8]

- **Roted binary search from roted sorted array.**

```
import java.util.ArrayList;
import java.util.List;

public class Test {

  public static int rotedBinarySearch(int[] arr, int
target, int s, int e){
   if(s > e){
      return -1;
   }

   int m = s + (e-s) / 2;

   if(arr[m] == target){
      return m;
   }

   if(arr[s] <= arr[m]){
      if(target >= arr[s] && target <= arr[m]){
         return rotedBinarySearch (arr, target, s, m-
1);
      } else {
         return rotedBinarySearch (arr, target, m+1,
e);
      }
   }
```

```
    if(target >= arr[m] && target <= arr[e]){
        return rotedBinarySearch (arr, target, m+1, e);
    }
    return rotedBinarySearch (arr, target, s, m-1);
   }

    public static void main(String[] args) {
        int[] array = {5, 6, 7, 8, 9, 1, 2, 3};
        int target = 4;
        System.out.println("Target Element: " + target);
        System.out.print("Indexe of target element:
"+rotedBinarySearch(array, target, 0, array.length-1));
    }
}
```

Output:

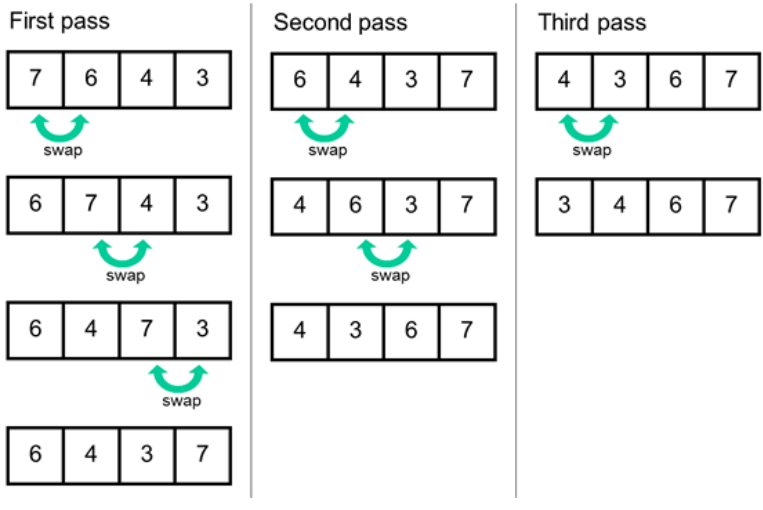
Target Element: 2
Indexe of target element: 6

4. Sorting using Recursion

1. Bubble Sort
2. Selection Sort
3. Merge Sort
4. Quick Sort

1. Bubble Sort :

Bubble sort is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the entire list is sorted.



- **What is the output of the following Java program fragment:**

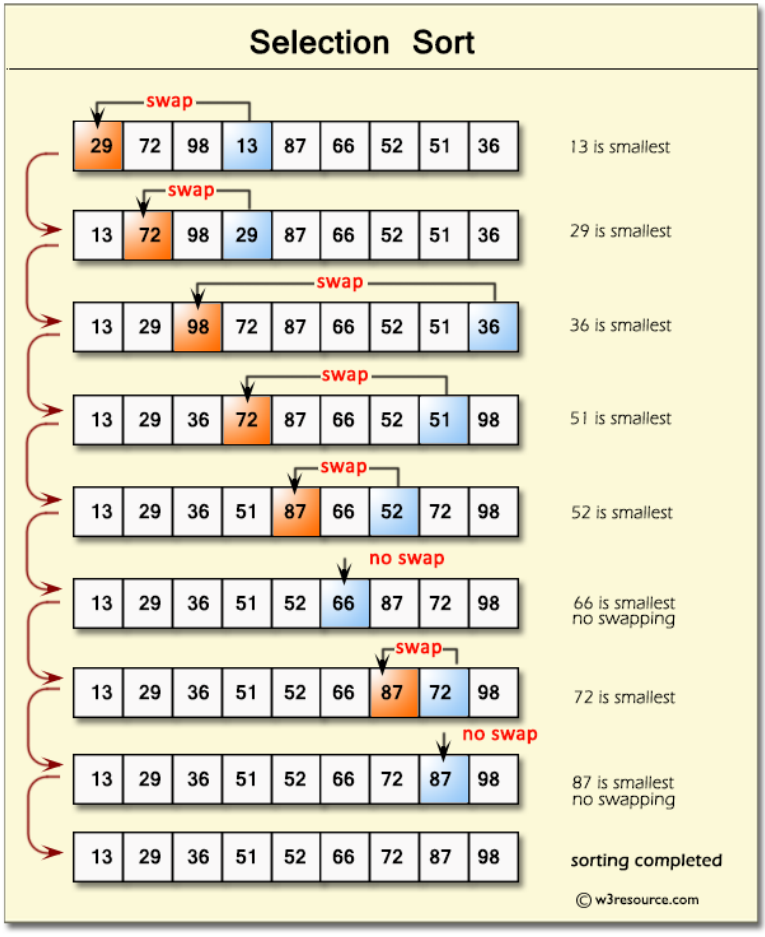
```
import java.util.Arrays;

public class Test{

  static void BubbleSort(int[] arr, int arrLength, int c){
    if(arrLength == 0){
      return;
    }
    if(arrLength==c){
      BubbleSort(arr, arrLength - 1, 0);
    } else{
      if(arr[c] > arr[c+1]){
        //swap
        int temp = arr[c];
        arr[c] = arr[c+1];
        arr[c+1] = temp;
      }
      BubbleSort(arr, arrLength, c+1);
    }
  }
  public static void main(String[] args) {
    int[] arr = {4, 6, 4, 2, 8};
    System.out.println("Before sort : "+Arrays.toString(arr));
    BubbleSort(arr, arr.length -1, 0);
    System.out.println("After sort : "+Arrays.toString(arr));
  }
}
```

Output:

Before sort : [4, 6, 4, 2, 8]
After sort : [2, 4, 4, 6, 8]

2. Selection Sort :

Selection Sort is a simple sorting algorithm that works by repeatedly finding the minimum element from the unsorted part of the array and swapping it with the first unsorted element. It effectively divides the array into two parts: the sorted part on the left and the unsorted part on the right.



- **What is the output of the following Java program fragment:**

```
import java.util.Arrays;

public class Test{

   static void BubbleSort(int[] arr, int arrLength, int c, int
max){
      if(arrLength == 0){
         return;
      }
      if(arrLength==c){
         int temp = arr[max];
         arr[max] = arr[arrLength-1];
         arr[arrLength-1] = temp;
         BubbleSort(arr, arrLength - 1, 0, 0);
      } else{
         if(arr[c] > arr[max]){
            BubbleSort(arr, arrLength, c+1, c);
         }
         else{
            BubbleSort(arr, arrLength, c+1, max);
         }
      }
   }
   public static void main(String[] args) {
      int[] arr = {4, 6, 4, 2, 8};
      System.out.println("Before sort :
"+Arrays.toString(arr));
```

```
        BubbleSort(arr, arr.length, 0, 0);
        System.out.println("After sort : "+Arrays.toString(arr));
    }
}
```

Output:

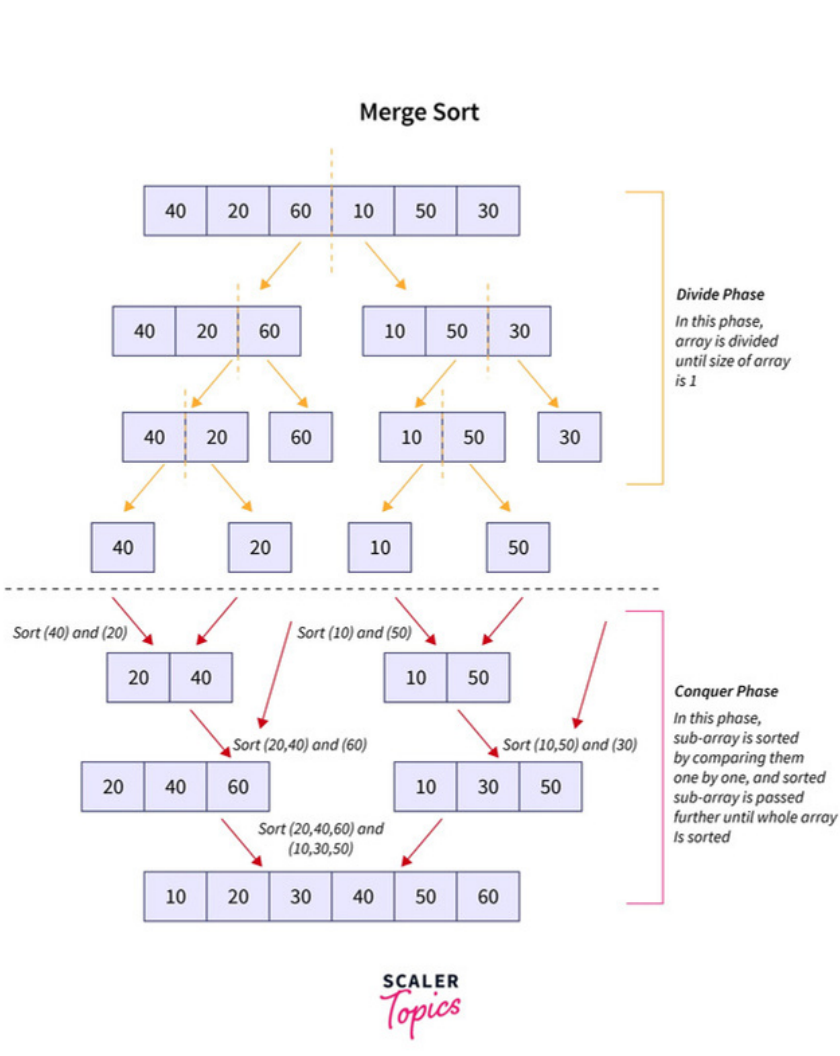
```
Before sort : [4, 6, 4, 2, 8]
After sort : [2, 4, 4, 6, 8]
```

3. Merge Sort :

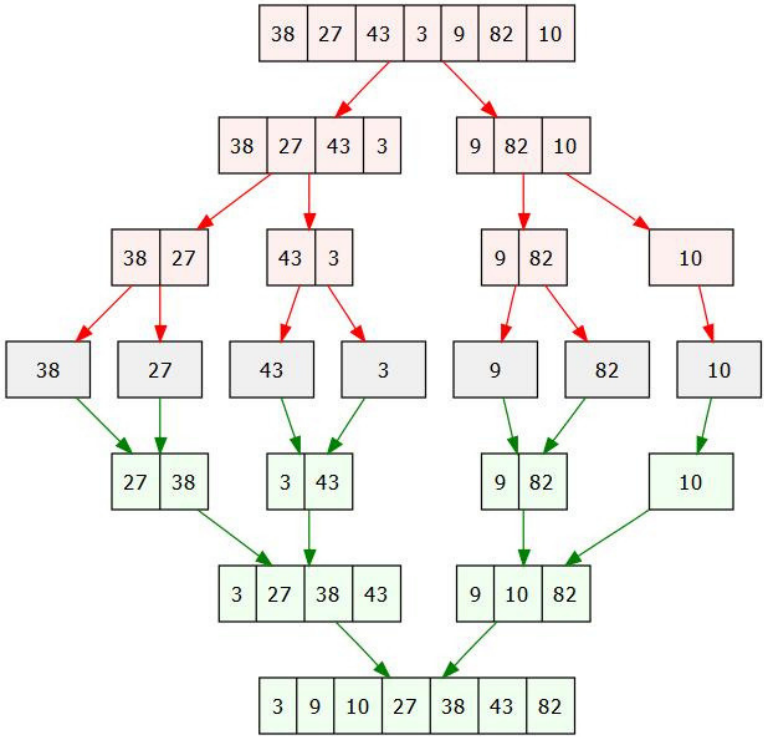
Merge Sort is a widely used comparison-based sorting algorithm that follows the divide-and-conquer strategy. It's known for its stability, consistent time complexity, and efficient performance for large datasets. The main idea behind Merge Sort is to break down a large array into smaller subarrays, sort them individually, and then merge them back together to obtain a sorted array.

The algorithm works as follows:

1. Divide: The unsorted array is divided into two equal halves (or as close to equal as possible).
2. Conquer: Each half is recursively sorted using the Merge Sort algorithm.
3. Merge: The sorted halves are then merged back together in a way that maintains the sorted order. This involves comparing the elements of the two halves and placing them in the correct order in a temporary array. Once all elements are merged, the temporary array becomes the sorted array.



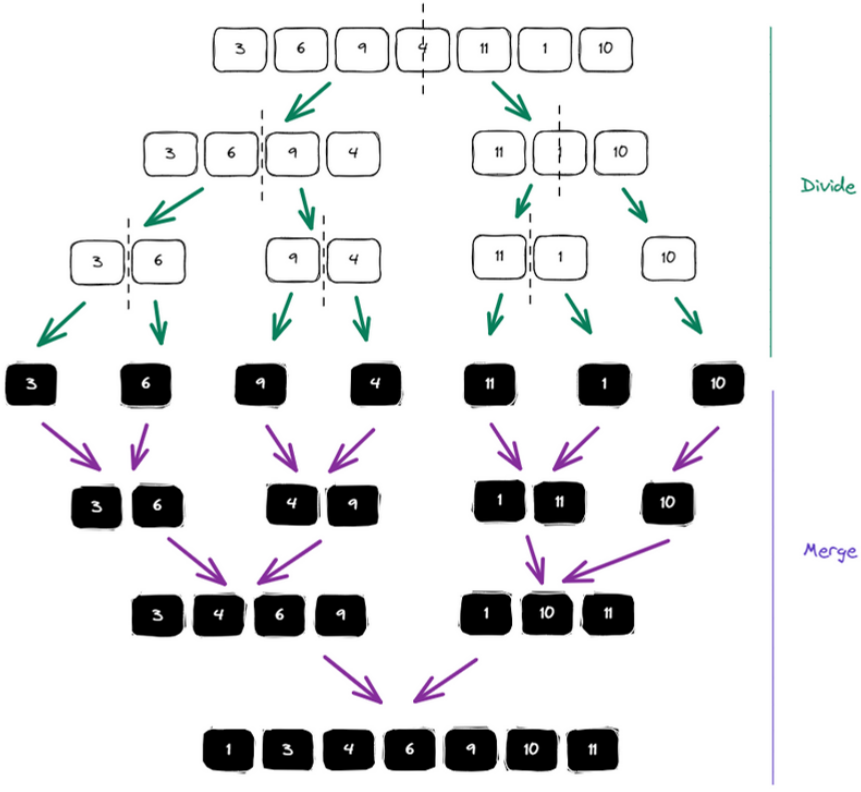
SCALER Topics


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Time Complexity:
Best case: O(n logn)
Worst case: O(n logn)

Where n is the size of the array.

Space Complexity:
O(log n)


3 6 9 4 11 1 10
3 6 9 4
11 1 10
3 6
9 4
11 1
10
3
6
9
4
11
1
10
3 6
4 9
1 11
10
3 4 6 9
1 10 11
1 3 4 6 9 10 11
Divide
Merge

- **What is the output of the following Java program fragment:**

```
public class Minian {

 static void mergeSort(int arr[], int l, int r) {
  if (l < r) {
    int m = l + (r - l) / 2;
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    merge(arr, l, m, r);
  }
 }

 private static void merge(int[] arr, int s, int m, int e){
  int[] mix = new int[(e - s)+1];
  int i = s;
  int j = m+1;
  int k = 0;

  while(i <= m && j <= e){
   if(arr[i] < arr[j]){
     mix[k] = arr[i];
     i++;
   } else{
     mix[k] = arr[j];
     j++;
   }
   k++;
  }
```

```
 // it may be possible that one of the arrays is not
complete
 // copy the remaining elements
 while(i <= m){
 mix[k] = arr[i];
 i++;
 k++;
 }

 while(j <= e){
 mix[k] = arr[j];
 j++;
 k++;
 }

 for(int l=0; l<mix.length; l++){
 arr[s+l] = mix[l];
 }
 }

 public static void main(String args[]) {
 int arr[] = {12, 11, 13, 5, 6, 7};
 int arr_size = arr.length;

 System.out.print("Original array: ");
 for (int i = 0; i < arr_size; i++)
 System.out.print(arr[i] + " ");
 System.out.println();

 mergeSort(arr, 0, arr_size - 1);
```

```
 System.out.print("Sorted array: ");
 for (int i = 0; i < arr_size; i++)
 System.out.print(arr[i] + " ");
 System.out.println();
 }
}
```

Output:

Original array: 12 11 13 5 6 7
Sorted array: 5 6 7 11 12 13

- **What is the output of the following Java program fragment:**

```
import java.util.Arrays;

public class Test{

   public static void main(String[] args) {
      int[] arr = {5, 4, 3, 2, 1};
      System.out.println("Before sort :
"+Arrays.toString(arr));
      mergeSort(arr, 0, arr.length);
      System.out.println("After sort : "+Arrays.toString(arr));
   }

   static void mergeSort(int[] arr, int s, int e){
      if(e - s == 1){
         return;
      }

      int mid = (s + e) / 2;

      mergeSort(arr,  s, mid);
      mergeSort(arr, mid, e);

      merge(arr, s, mid, e);
   }

   private static void merge(int[] arr, int s, int m, int e){
      int[] mix = new int[e - s];
      int i = s;
```

```
    int j = m;
    int k = 0;

    while(i < m && j < e){
        if(arr[i] < arr[j]){
            mix[k] = arr[i];
            i++;
        } else{
            mix[k] = arr[j];
            j++;
        }
        k++;
    }

    // it may be possible that one of the arrays is not
complete
    // copy the remaining elements
    while(i < m){
        mix[k] = arr[i];
        i++;
        k++;
    }

    while(j < e){
        mix[k] = arr[j];
        j++;
        k++;
    }
```

```
        for(int l=0; l<mix.length; l++){
            arr[s+l] = mix[l];
        }
    }
}
```

Output:

Before sort : [5, 4, 3, 2, 1]
After sort : [1, 2, 3, 4, 5]

- **What is the output of the following Java program fragment:**

```
import java.util.Arrays;

public class Test{

   public static void main(String[] args) {
      int[] arr = {5, 4, 3, 2, 1};
      System.out.println("Before sort :
"+Arrays.toString(arr));
      arr = mergeSort(arr);
      System.out.println("After sort : "+Arrays.toString(arr));
   }

   static int[] mergeSort(int[] arr){
      if(arr.length == 1){
         return arr;
      }

      int mid = arr.length / 2;

      int[] left = mergeSort(Arrays.copyOfRange(arr, 0,
mid));
      int[] right = mergeSort(Arrays.copyOfRange(arr, mid,
arr.length));
      return merge(left, right);
   }

   private static int[] merge(int[] first, int[] second){
      int[] mix = new int[first.length + second.length];
```

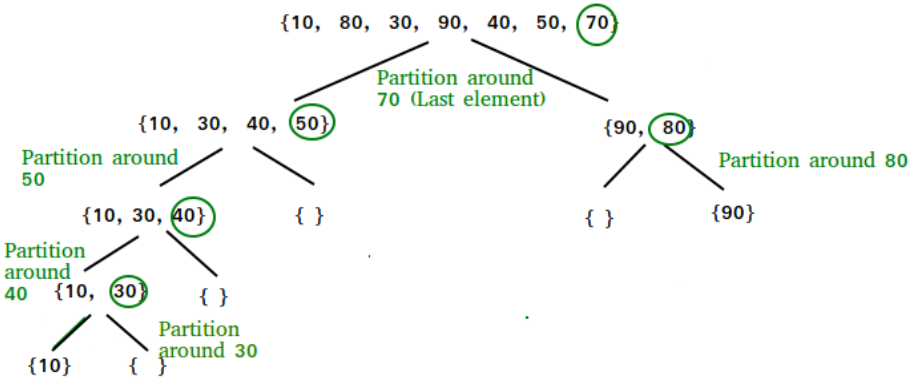
```
        int i = 0;
        int j = 0;
        int k = 0;

        while(i < first.length && j < second.length){
            if(first[i] < second[j]){
                mix[k] = first[j];
                i++;
            } else{
                mix[k] = second[j];
                j++;
            }
            k++;
        }

        // it may be possible that one of the arrays is not
complete
        // copy the remaining elements
        while(i < first.length){
            mix[k] = first[i];
            i++;
            k++;
        }

        while(j < second.length){
            mix[k] = second[j];
            j++;
            k++;
        }

        return mix;
    }
}
```

Output:

```
Before sort : [5, 4, 3, 2, 1]
After sort : [1, 2, 3, 4, 5]
```

4. Quick Sort :

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.



The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

Time Complexity:
Best Case: Ω (N log (N))
Average Case: θ (N log (N))
Worst Case: O(N^2)

Auxiliary Space Complexity:
O(1)

If we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make O(N).

1. Best Case:-
The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.

In this case, the algorithm will make balanced partitions, leading to efficient Sorting.

The Best case time complexity of quicksort is O(n log n).

2. Average Case:-

Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.
The Average Case Time-complexity is also O(n log n).

3. worst Case:-

The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions.

When the array is already sorted and the pivot is always chosen as the smallest or largest element.

To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort) to shuffle the element before sorting.

Quicksort Worst-case Time Complexity is O(n^2).

- **What is the output of the following Java program fragment:**

```
import java.util.Arrays;

public class Test{

   public static void main(String[] args) {
      int[] arr = {5, 4, 3, 2, 1};
      System.out.println("Before sort :
"+Arrays.toString(arr));
      QuickSort(arr, 0, arr.length-1);
      System.out.println("After sort : "+Arrays.toString(arr));
   }

   static void QuickSort(int[] nums,  int low, int high){
      if(low >= high){
         return;
      }

      int s = low;
      int e = high;
      int m = s + (e - s) /2;
      int pivot = nums[m];

      while (s <= e){

         // It's also reason whyif its already sorted it will not
swap
         while (nums[s] < pivot){
            s++;
         }
```

```
            while (nums[e] > pivot){
                e--;
            }

            if (s <= e){
                int temp = nums[s];
                nums[s] = nums[e];
                nums[e] = temp;
                s++;
                e--;
            }
        }

        // now my pivot is at correct index, please sort two
halves now
        QuickSort(nums, low, e);
        QuickSort(nums, s, high);
    }
}
```

Output:

Before sort : [5, 4, 3, 2, 1]
After sort : [1, 2, 3, 4, 5]

5. Pattern using Recursion

- **What is the output of the following Java program fragment:**

```
public class Test{

   static void TrianglePattern(int r, int c){
      if(r==0){
         return;
      }
      if(r==c){
         System.out.println();
         TrianglePattern(r-1, 0);
      } else{
         System.out.print("*");
         TrianglePattern(r, c+1);
      }



   }
   public static void main(String[] args) {
      TrianglePattern(4, 0);
   }
}
```

Output:

```
****
***
**
*
```

or,

```
public class Test{

   static void TrianglePattern(int r, int c){
      if(r==0){
         return;
      }
      if(c < r){
         System.out.print("*");
         TrianglePattern(r, c+1);
      } else{
          System.out.println();
         TrianglePattern(r-1, 0);
      }


   }
   public static void main(String[] args) {
      TrianglePattern(4, 0);
   }
}
```

Output:

```
****
***
**
*
```

- **What is the output of the following Java program fragment:**

```
public class Test{

   static void TrianglePattern(int r, int c){
      if(r==0){
         return;
      }
      if(c < r){
         TrianglePattern(r, c+1);
         System.out.print("*");
      } else{
         TrianglePattern(r-1, 0);
         System.out.println();
      }

   }
   public static void main(String[] args) {
      TrianglePattern(4, 0);
   }
}
```

Output:

```
*
**
***
****
```

```
public class Test{

   static void TrianglePattern(int r, int c){
      if(r==0){
         return;
      }
      if(r==c){
         TrianglePattern(r-1, 0);
         System.out.println();
      } else{
         TrianglePattern(r, c+1);
         System.out.print("*");
      }


   }
   public static void main(String[] args) {
      TrianglePattern(4, 0);
   }
}
```

Output:

```
*
**
***
****
```

6. Recursion Subset, Subsequence, String Questions:

String in recursion:

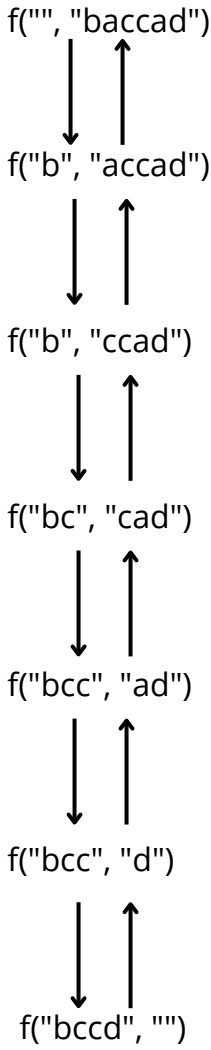
- **Make a program that will take a string and remove all 'a' character from it after then it will simply print the string.**

```
public class Test{
   public static void main(String[] args) {
      skip("", "baccdah");
   }

   static void skip(String p, String up){
      if(up.isEmpty()){
         System.out.println(p);
         return;
      }

      char ch = up.charAt(0);

      if(ch == 'a'){
         skip(p, up.substring(1));
      } else{
         skip(p + ch, up.substring(1));
      }
   }
}
```

Output:

bccdh

Here,

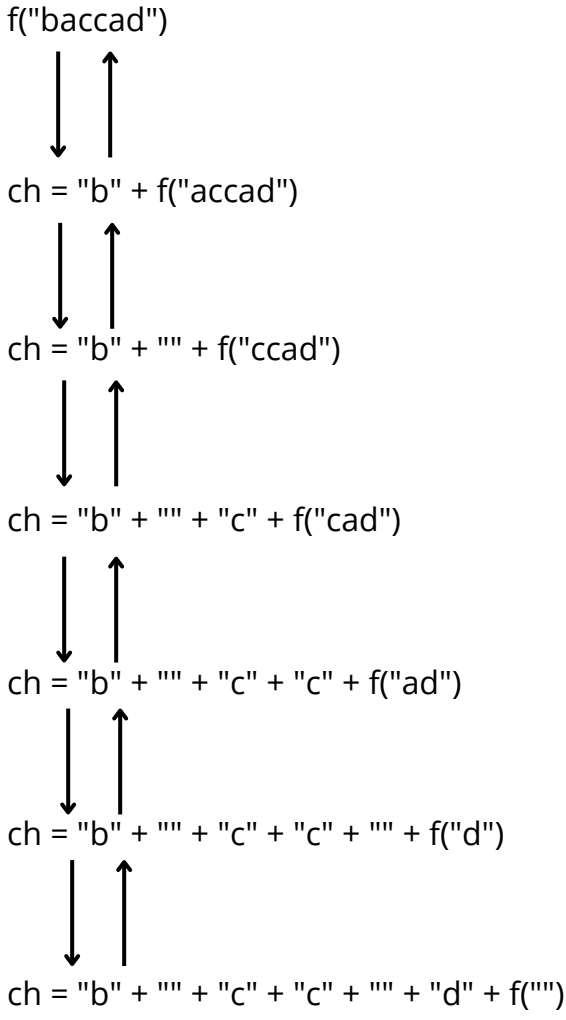


- **Make a program that will take a string and remove all 'a' character from it after then it will simply print the string.**

```
public class Test{
  public static void main(String[] args) {
    System.out.println(skip("baccdah"));;
  }

  static String skip( String up){
    if(up.isEmpty()){
      return "";
    }

    char ch = up.charAt(0);

    if(ch == 'a'){
      return skip(up.substring(1));
    } else{
      return ch + skip(up.substring(1));
    }
  }
}
```

Output:
bccdh

Here,


f("baccad")
ch = "b" + f("accad")
ch = "b" + "" + f("ccad")
ch = "b" + "" + "c" + f("cad")
ch = "b" + "" + "c" + "c" + f("ad")
ch = "b" + "" + "c" + "c" + "" + f("d")
ch = "b" + "" + "c" + "c" + "" + "d" + f("")

- **Make a program that will take a string and remove all "apple" from it after that it will simply print the string.**

```
public class Test{
 public static void main(String[] args) {

System.out.println(skipApple("bappleccappledah"));
;
 }

 static String skipApple( String up){
  if(up.isEmpty()){
   return "";
  }

  if(up.startsWith("apple")){
   return skipApple(up.substring(5));
  } else{
   return up.charAt(0) + skipApple(up.substring(1));
  }
 }
}
```

Output:
bccdah

- **Make a program that will take a string and remove all "app" only not "apple" from it after that it will simply print the string.**

```
public class Test{
 public static void main(String[] args) {

System.out.println(skipApple("bapplecappcappleda
h"));;
 }

 static String skipApple( String up){
  if(up.isEmpty()){
   return "";
  }

  if(up.startsWith("app") && !up.startsWith("apple"))
{
   return skipApple(up.substring(3));
  } else{
   return up.charAt(0) + skipApple(up.substring(1));
  }
 }
}
```

Output:
bappleccappledah

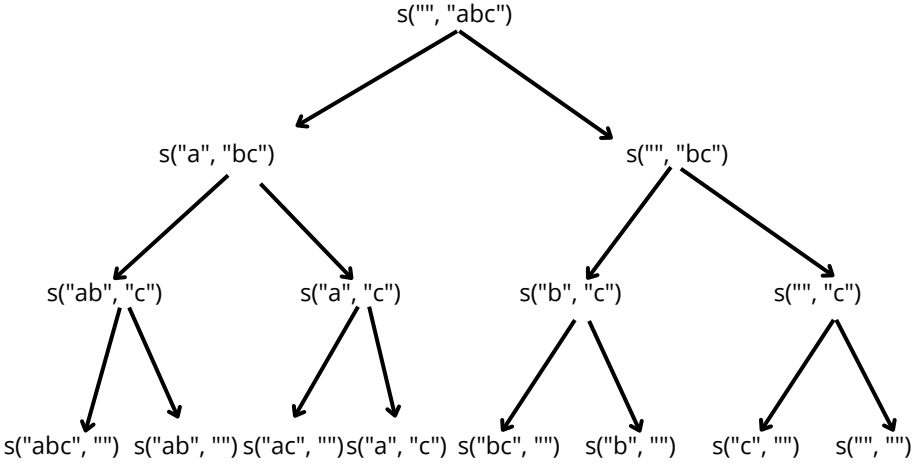
Subsets:

- **Permutation and combination:**

Subset ----> Non-adjacent collection.
[3, 5, 9] ----> [3], [3,5], [3,9], [3,5,9], [5,9], [5], [9]

str = "ABC"
ans = ["a", "b", "c", "ab", "ac", "bc", "abc"]

- **combination:**



- **Subsequence:**

- **Make a program that will show the subsequences of String.**

```
public class Test{
   public static void main(String[] args) {
      subSeq("", "abc");
   }

   public static void subSeq(String p, String str){
      if(str.isEmpty()){
         System.out.println(p+" ");
         return;
      }

      char ch = str.charAt(0);

      subSeq(p + ch, str.substring(1));
      subSeq(p, str.substring(1));

   }
}
```

Output:

```
abc
ab
ac
a
bc
b
c
```

- **Make a program that will show the subsequences of String.**

```
public class Test{
   public static void main(String[] args) {
      subSeq("", "abc");
   }

   public static void subSeq(String p, String str){
      if(str.isEmpty()){
         System.out.print(p+" ");
         return;
      }

      char ch = str.charAt(0);

      subSeq(p + ch, str.substring(1));
      subSeq(p, str.substring(1));

   }
}
```

Output:

```
c
b
bc
a
ac
ab
abc
```

- **Make a program that will show the subsequences of String.**

```
import java.util.ArrayList;

public class Test{
  public static void main(String[] args) {
    System.out.println(subSeq("", "abc"));;
  }

  public static ArrayList<String> subSeq(String p,
String str){
    if(str.isEmpty()){
      ArrayList<String> list = new ArrayList<>();
      list.add(p);
      return list;
    }

    char ch = str.charAt(0);

    ArrayList<String> left = subSeq(p + ch,
str.substring(1));
    ArrayList<String> right =subSeq(p,
str.substring(1));

    left.addAll(right);
    return left;
  }
}
```

Output:

[abc, ab, ac, a, bc, b, c,]

- **Make a program that will show the subsequences of String with ASCII value.**

```
public class Test{
  public static void main(String[] args) {
    subSeq("", "abc");
  }

  public static void subSeq(String p, String str){
    if(str.isEmpty()){
      System.out.println(p);
      return;
    }

    char ch = str.charAt(0);

    subSeq(p + ch, str.substring(1));
    subSeq(p , str.substring(1));
    subSeq(p + (ch+0), str.substring(1));

  }
}
```

Output:

abc
ab
ab99
ac
a
a99
a98c
a98
a9899
bc
b
b99
c

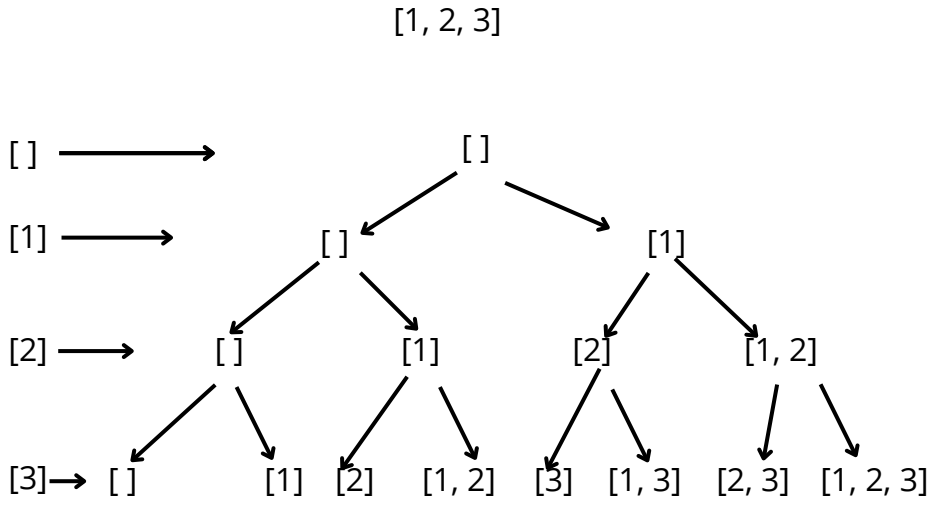
99
98c
98
9899
97bc
97b
97b99
97c
97
9799
9798c
9798
979899

- **Make a program that will show the subsequences of String with ASCII value.**

```
import java.util.ArrayList;

public class Test{
 public static void main(String[] args) {
  System.out.println(subSeqAscii("", "abc"));;
 }

 public static ArrayList<String> subSeqAscii(String p,
String str){
  if(str.isEmpty()){
   ArrayList<String> list = new ArrayList<>();
   list.add(p);
   return list;
  }

  char ch = str.charAt(0);

  ArrayList<String> first = subSeqAscii(p + ch,
str.substring(1));
  ArrayList<String> second = subSeqAscii(p,
str.substring(1));
  ArrayList<String> third =subSeqAscii(p + (ch+0),
str.substring(1));

  first.addAll(second);
  first.addAll(third);
  return first;
 }
}
```

Output:
[abc, ab, ab99, ac, a, a99, a98c, a98, a9899, bc, b, b99, c, , 99, 98c, 98, 9899, 97bc, 97b, 97b99, 97c, 97, 9799, 9798c, 9798, 979899]

- **Subset:**



Time Complexity:
Worst Case: O(n * 2^n)

Auxiliary Space Complexity:
O(2^n * n)

- **Make a program that will show subset of [1, 2, 3].**

```
import java.util.ArrayList;
import java.util.List;

public class Test{
  public static void main(String[] args) {
    int[] arr = {1, 2, 3};
    List<List<Integer>> ans = subset(arr);
    for (List<Integer> list : ans) {
       System.out.println(list);
    }
  }

  public static List<List<Integer>> subset(int[] arr){
    List<List<Integer>> outer = new ArrayList<>();
    outer.add(new ArrayList<>());

    for(int num : arr){
       int n = outer.size();
       for(int i = 0; i<n; i++){
          List<Integer> internal = new ArrayList<>
(outer.get(i));
          internal.add(num);
          outer.add(internal);
       }
    }
    return outer;
  }

}
```

Output:

[ ]
[1]
[2]
[1, 2]
[3]
[1, 3]
[2, 3]
[1, 2, 3]

- **Make a program that will show subsequencees of a string without duplicate elements.**

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Test{
 public static void main(String[] args) {
  int[] arr = {1, 2, 2};
  List<List<Integer>> ans = subset(arr);
  for (List<Integer> list : ans) {
    System.out.println(list);
  }
 }

 public static List<List<Integer>> subset(int[] arr){
   Arrays.sort(arr);
  List<List<Integer>> outer = new ArrayList<>();
  outer.add(new ArrayList<>());
  int start = 0;
  int end = 0;
  for(int i=0; i<arr.length; i++){
    start = 0;
    // if current and previous elemnts is same, s = e + 1
    if(i>0 && arr[i] == arr[i-1]){
      start = end + 1;
    }
    end = outer.size() - 1;
    int n = outer.size();
    for(int j = start; j<n; j++){
```

```
    List<Integer> internal = new ArrayList<>(outer.get(j));
    internal.add(arr[i]);
    outer.add(internal);
   }
  }
  return outer;
 }

}
```

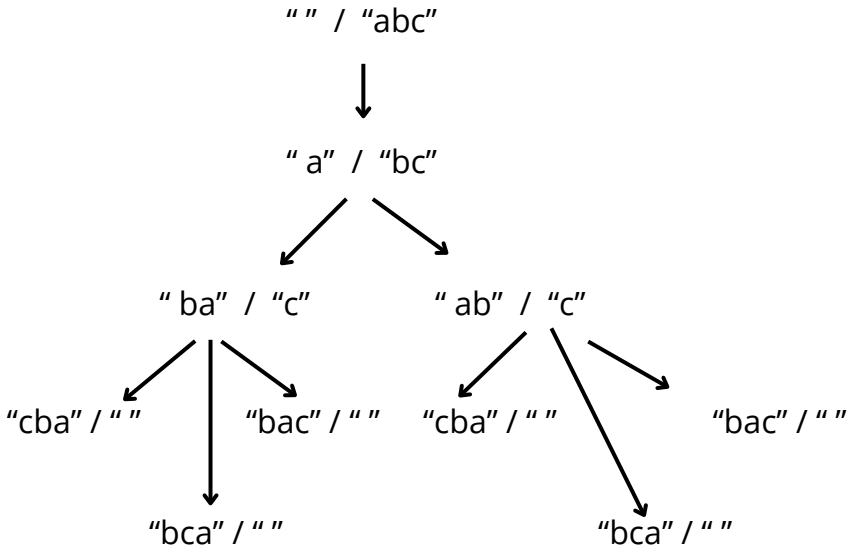
Output:

```
[ ]
[1]
[2]
[1, 2]
[2, 2]
[1, 2, 2]
```

Here,
When you find a duplicate element, only add it in the newly created subset of the previous step.

Because of the above point, duplicates have to be together.

- **Permutations:**



- **What is the output of the following Java program fragment:**

```
class Test{
   public static void main(String[] args) {
      permutations("","abc");
   }

   static void permutations(String p, String up){
      if(up.isEmpty()){
         System.out.println(p);
         return;
      }
      char ch=up.charAt(0);
      for(int i=0; i<=p.length(); i++){
         String f = p.substring(0, i);
         String s = p.substring(i, p.length());
         permutations(f + ch + s, up.substring(1));
      }
   }
}
```

Output:

```
cba
bca
bac
cab
acb
abc
```

- **What is the output of the following Java program fragment:**

```
import java.util.ArrayList;

class Test{
   public static void main(String[] args) {
      ArrayList<String> ans = permutationList("", "abc");
      System.out.println(ans);
   }

   static ArrayList<String> permutationList(String p,
String up){
      if(up.isEmpty()){
         ArrayList<String> list=new ArrayList<>();
         list.add(p);
         return list;
      }

      char ch = up.charAt(0);

      // local to this call
      ArrayList<String> ans = new ArrayList<>();

      for(int i=0; i<= p.length(); i++){
         String f = p.substring (0, i);
         String s = p.substring(i, p.length());
         ans.addAll(permutationList(f + ch + s,
up.substring(1)));
      }
      return ans;
   }
}
```

Output:
[cba, bca, bac, cab, acb, abc]

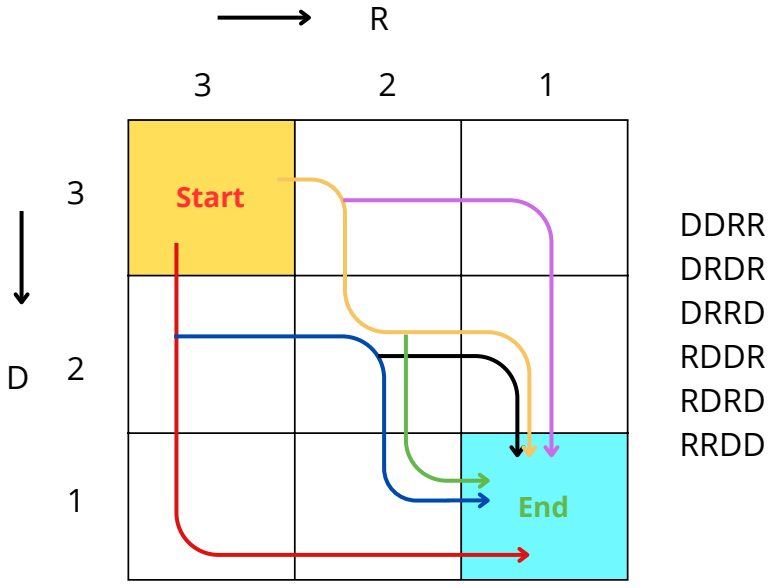
- **What is the output of the following Java program fragment:**

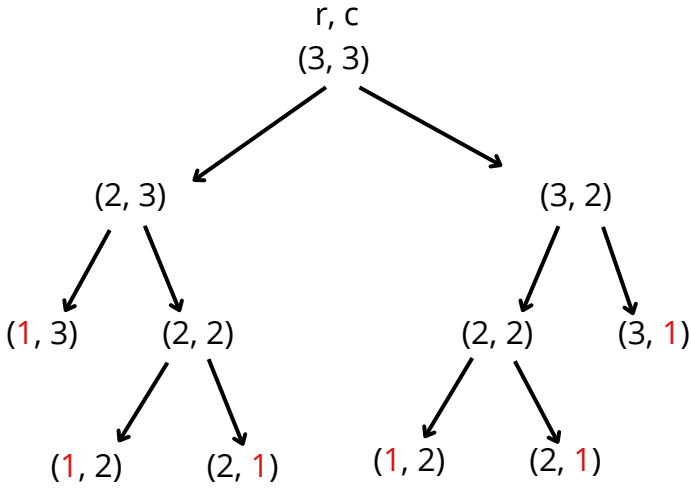
```
class Test{
  public static void main(String[] args) {
    System.out.println(permutationsCount("", "abc"));
  }

  static int permutationsCount(String p, String up){
    if(up.isEmpty()){
      return 1;
    }

    int count = 0;
    char ch = up.charAt(0);
    for(int i=0; i<=p.length(); i++){
      String f = p.substring(0, i);
      String s = p.substring(i, p.length());
      count = count + permutationsCount(f + ch + s,
up.substring(1));
    }

    return count;
  }
}
```

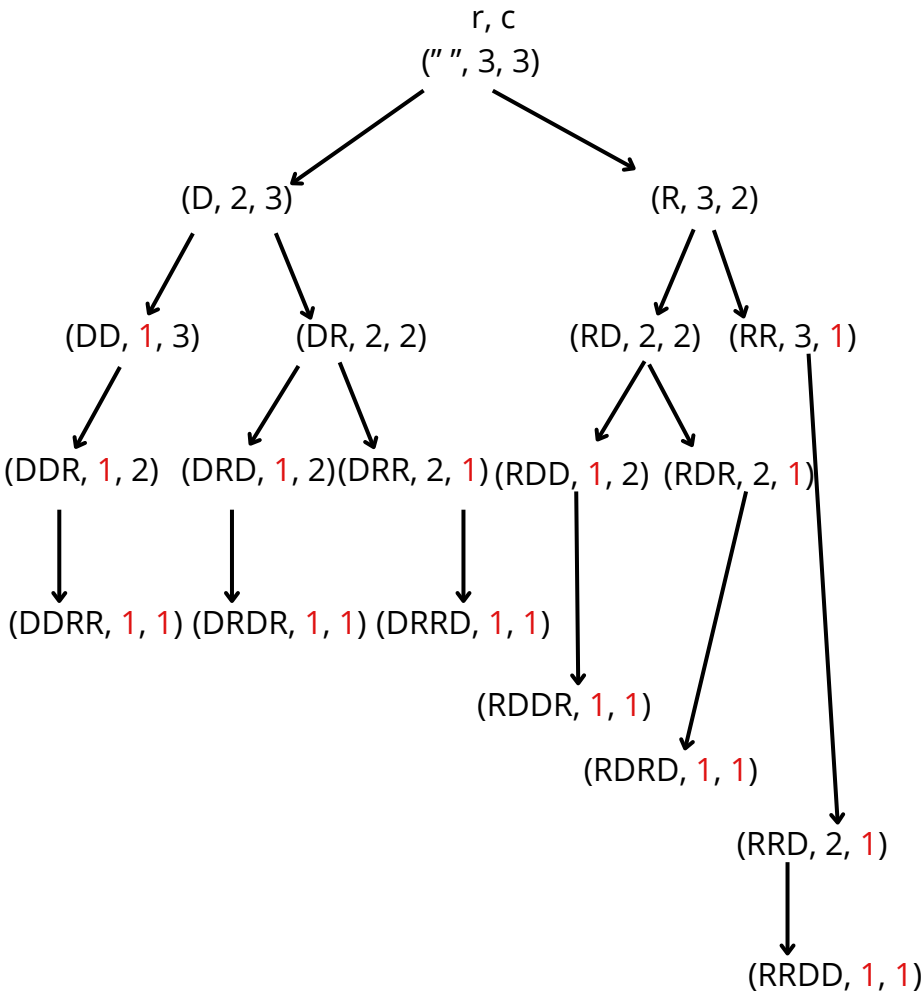
Output:
6

7. Backtracking

- **Backtracking (Maze Problem) :**






r, c
(" ", 3, 3)
(D, 2, 3)
(R, 3, 2)
(DD, 1, 3)
(DR, 2, 2)
(RD, 2, 2)
(RR, 3, 1)
(DDR, 1, 2)
(DRD, 1, 2)
(DRR, 2, 1)
(RDD, 1, 2)
(RDR, 2, 1)
(DDRR, 1, 1)
(DRDR, 1, 1)
(DRRD, 1, 1)
(RDDR, 1, 1)
(RDRD, 1, 1)
(RRD, 2, 1)
(RRDD, 1, 1)

- **What is the output of the following Java program fragment:**

```
class Test{
   public static void main(String[] args) {
      System.out.println(count(3, 3));
   }

   static int count(int r, int c){
      if(r == 1 || c == 1){
         return 1;
      }

      int left = count(r-1, c);
      int right = count(r, c-1);
      return left + right;
   }
}
```

Output:
6

- **What is the output of the following Java program fragment:**

```
class Test{
  public static void main(String[] args) {
    path("", 3, 3);
  }

  static void path(String p, int r, int c){
    if(r == 1 && c == 1){
      System.out.println(p);
      return;
    }

    if(r > 1){
      path(p + 'D', r-1, c);
    }

    if(c > 1){
      path(p + 'R', r, c-1);
    }
  }
}
```

Output:

DDRR
DRDR
DRRD
RDDR
RDRD
RRDD

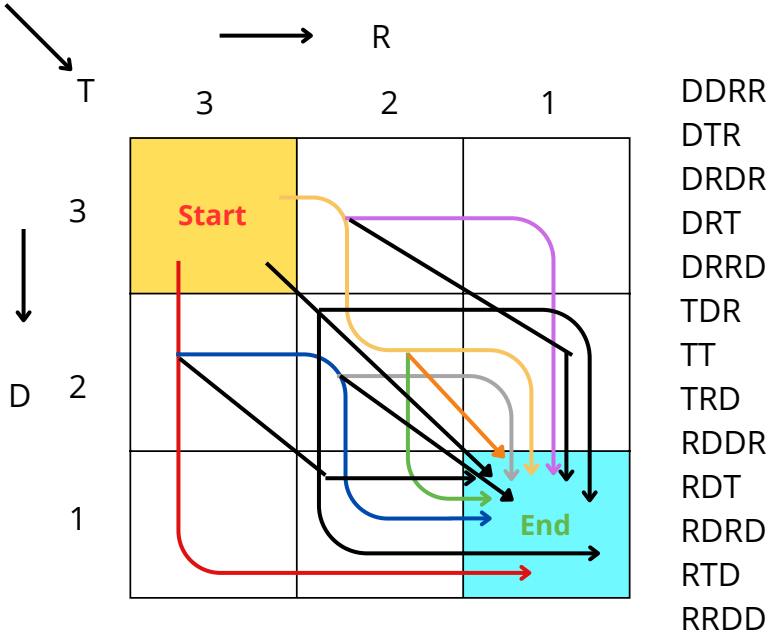
- **What is the output of the following Java program fragment:**

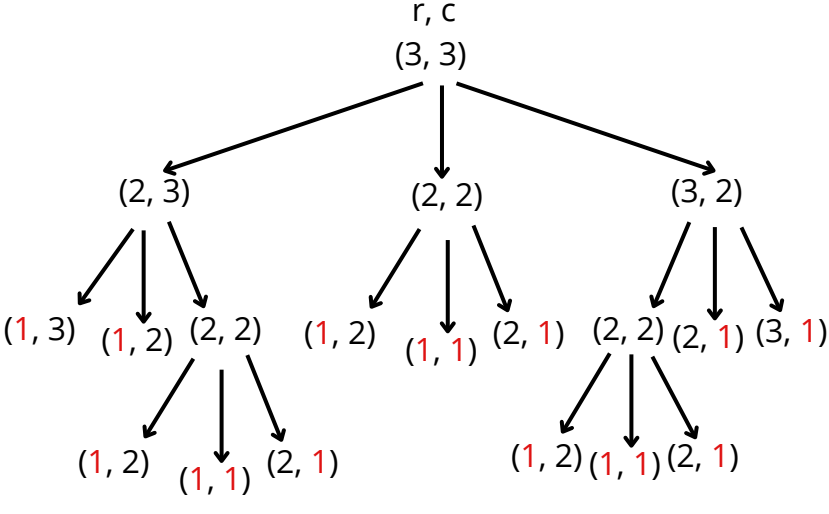
```
import java.util.ArrayList;

class Test{
  public static void main(String[] args) {
    System.out.println(pathRet("", 3, 3));
  }

  static ArrayList<String> pathRet(String p, int r, int
c){
    if(r == 1 && c == 1){
      ArrayList<String> list = new ArrayList<>();
      list.add(p);
      return list;
    }

    ArrayList<String> list = new ArrayList<>();

    if(r > 1){
      list.addAll(pathRet(p + 'D', r-1, c));
    }

    if(c > 1){
      list.addAll(pathRet(p + 'R', r, c-1));
    }

    return list;
  }
}
```

Output:

[DDRR, DRDR, DRRD, RDDR, RDRD, RRDD]

- **Backtracking (Maze Problem--Diagonal) :**





- **What is the output of the following Java program fragment:**

```
class Test{
  public static void main(String[] args) {
    System.out.println(count(3, 3));
  }

  static int count(int r, int c){
    if(r == 1 || c == 1){
      return 1;
    }

    int left = count(r-1, c);
    int middel = count(r-1, c-1);
    int right = count(r, c-1);
    return left + middel + right;
  }
}
```

Output:
13

- **What is the output of the following Java program fragment:**

```
class Test{
 public static void main(String[] args) {
  path("", 3, 3);
 }

 static void path(String p, int r, int c){
  if(r == 1 && c == 1){
   System.out.println(p);
   return;
  }

  if(r > 1){
   path(p + 'D', r-1, c);
  }

  if(r > 1 && c>1){
   path(p + 'T', r-1, c-1);
  }

  if(c > 1){
   path(p + 'R', r, c-1);
  }
 }
}
```

Output:

DDRR
DTR
DRDR
DRT
DRRD
TDR
TT
TRD
RDDR
RDT
RDRD
RTD
RRDD

- **What is the output of the following Java program fragment:**

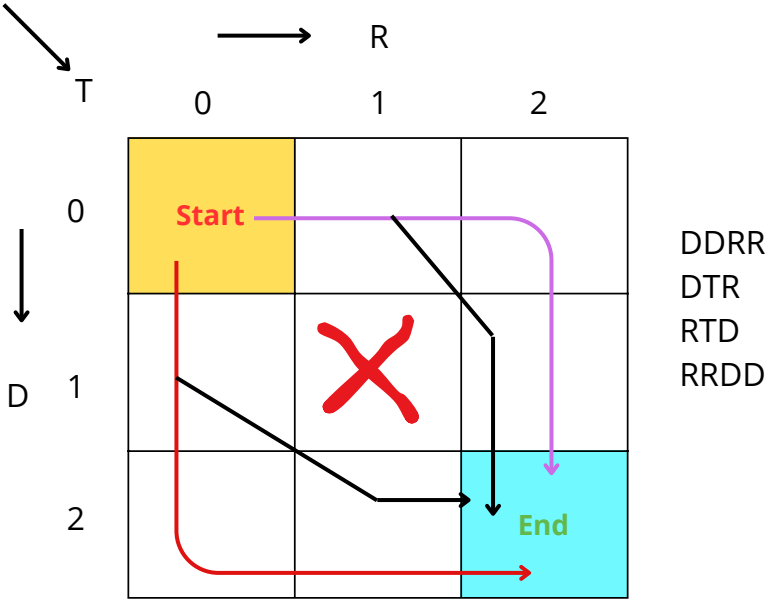
```
import java.util.ArrayList;

class Test{
  public static void main(String[] args) {
    System.out.println(pathRetDiagonal("", 3, 3));
  }

  static ArrayList<String> pathRetDiagonal(String p,
int r, int c){
    if(r == 1 && c == 1){
      ArrayList<String> list = new ArrayList<>();
      list.add(p);
      return list;
    }

    ArrayList<String> list = new ArrayList<>();

    if(r > 1){
      list.addAll(pathRetDiagonal(p + 'D', r-1, c));
    }

    if(c > 1 && r>1){
      list.addAll(pathRetDiagonal(p + 'T', r-1, c-1));
    }

    if(c > 1){
      list.addAll(pathRetDiagonal(p + 'R', r, c-1));
    }
```

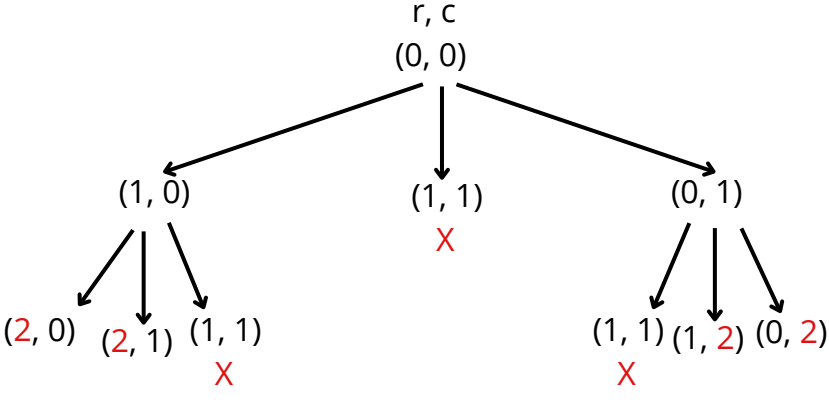
```
    return list;
  }
}
```

Output:

[DDRR, DTR, DRDR, DRT, DRRD, TDR, TT, TRD, RDDR, RDT, RDRD, RTD, RRDD]

- **Backtracking (Maze Problem) :**





- **What is the output of the following Java program fragment:**

```
class Test{
 public static void main(String[] args) {
  System.out.println(count(3, 3));
 }

 static int count(int r, int c){
  return countStart(r, c, 0, 0);
 }

 static int countStart(int r, int c, int a, int b){
  if(a == r-1 || b == c-1){
   return 1;
  }

  if(a==1 && b==1){
    return 0;
  }

  int left = countStart(r, c, a+1, b);
  int middel = countStart(r, c, a+1, b+1);
  int right = countStart(r, c, a, b+1);
  return left + middel + right;
 }
}
```

Output:

4

- **What is the output of the following Java program fragment:**

```
class Test{
 public static void main(String[] args) {
  boolean[][] board = {
    {true, true, true},
    {true, false, true},
    {true, true, true},
  };

  pathRestrictions("", board, 0, 0);
 }

 static void pathRestrictions(String p, boolean[][]
maze, int r, int c){
  if(r == maze.length - 1 && c == maze[0].length - 1){
    System.out.println(p);
    return;
  }

  if(maze[r][c] == false){
    return;
  }

  if(r < maze.length - 1){
    pathRestrictions(p + 'D', maze, r+1, c);
  }
```

```
  if(r < maze.length - 1 && c < maze[0].length - 1){
    pathRestrictions(p + 'T', maze, r+1, c+1);
  }

  if(c < maze[0].length - 1){
    pathRestrictions(p + 'R', maze, r, c+1);
  }

 }
}
```

Output:

DDRR
DTR
RTD
RRDD

- **What is the output of the following Java program fragment:**

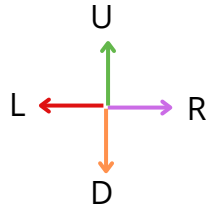
```
import java.util.ArrayList;

class Test{
 public static void main(String[] args) {
  boolean[][] board = {
    {true, true, true},
    {true, false, true},
    {true, true, true},
  };

  System.out.println(pathRestrictions("", board, 0,
0));
 }

 static ArrayList<String> pathRestrictions(String p,
boolean[][] maze, int r, int c){
  if(r == maze.length - 1 && c == maze[0].length - 1){
    ArrayList<String> list = new ArrayList<String>();
    list.add(p);
    return list;
  }

  if(maze[r][c] == false){
    ArrayList<String> list = new ArrayList<String>();
    return list;
  }

  ArrayList<String> ans = new ArrayList<>();
```

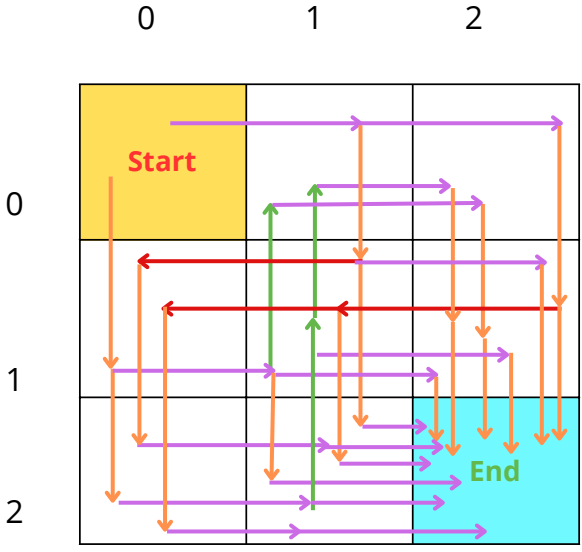
```
  if(r < maze.length - 1){
    ans.addAll(pathRestrictions(p + 'D', maze, r+1, c));
  }

  if(r < maze.length - 1 && c < maze[0].length - 1){
    ans.addAll(pathRestrictions(p + 'T', maze, r+1, c+1));
  }

  if(c < maze[0].length - 1){
    ans.addAll(pathRestrictions(p + 'R', maze, r, c+1));
  }

  return ans;
 }
}
```

Output:

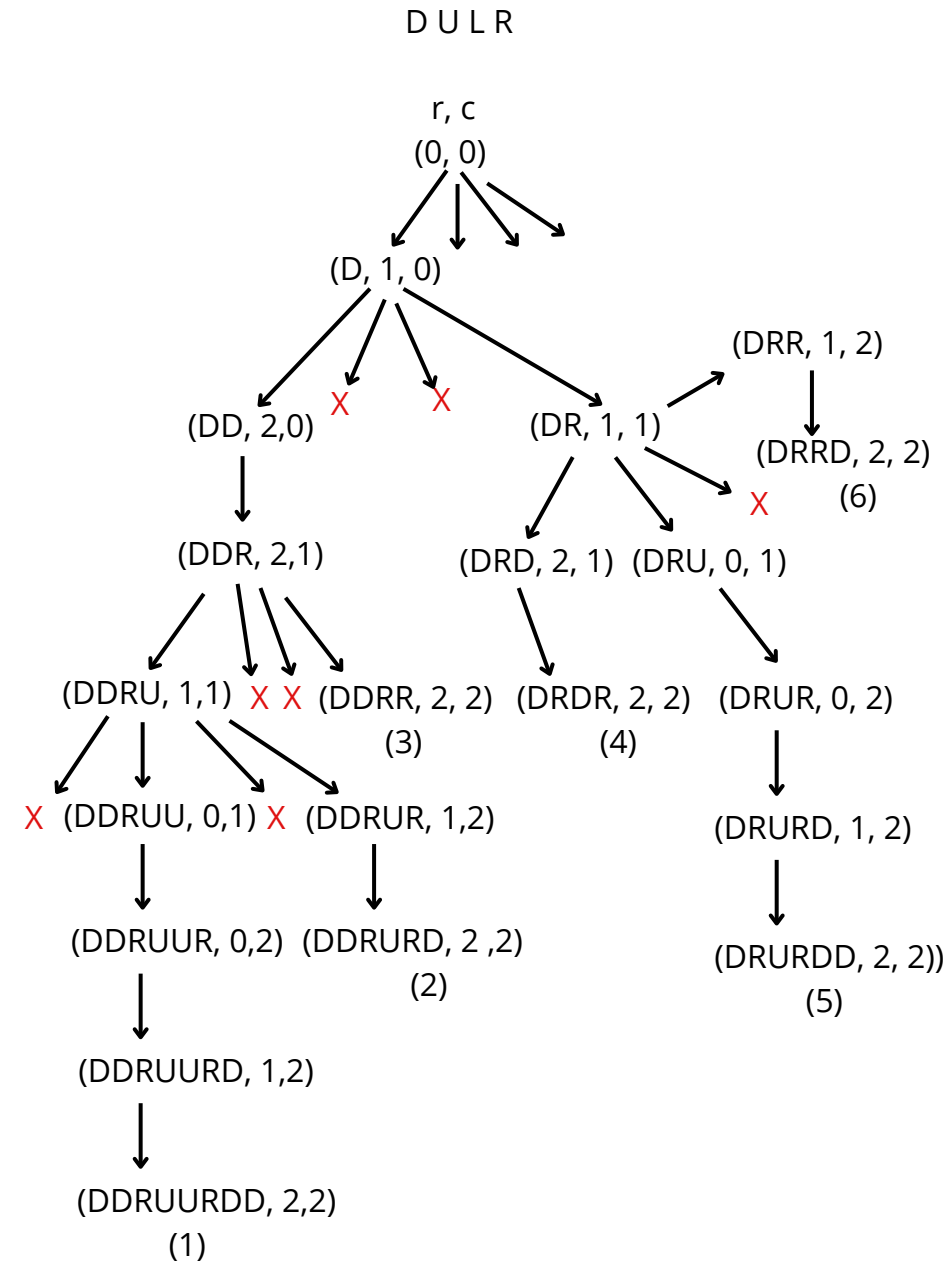
[DDRR, DTR, RTD, RRDD]

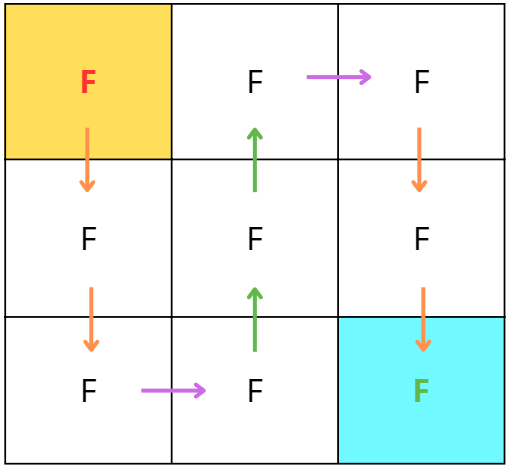
- **Backtracking (Maze Problem--Four Direction) :**





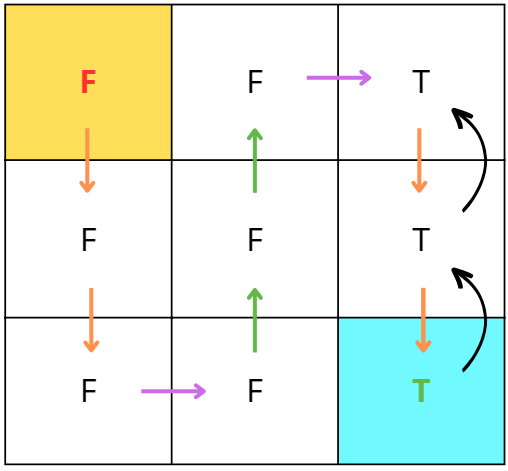
DDRUURDD
DDRURD
DDRR
DRDR
DRURDD
DRRD
RDDR
RDLDRR
RDRD
RRDD
RRDLDR
RRDLLDRR


D U L R
r, c
(0, 0)
(D, 1, 0)
X
X
(DD, 2,0)
(DR, 1, 1)
(DRR, 1, 2)
(DRRD, 2, 2)
(6)
X
(DDR, 2,1)
(DRD, 2, 1)
(DRU, 0, 1)
(DDRU, 1,1)
X
X
(DDRR, 2, 2)
(3)
(DRDR, 2, 2)
(4)
(DRUR, 0, 2)
X
(DDRUU, 0,1)
X
(DDRUR, 1,2)
(DRURD, 1, 2)
(DDRUUR, 0,2)
(DDRURD, 2 ,2)
(2)
(DRURDD, 2, 2))
(5)
(DDRUURD, 1,2)
(DDRUURDD, 2,2)
(1)



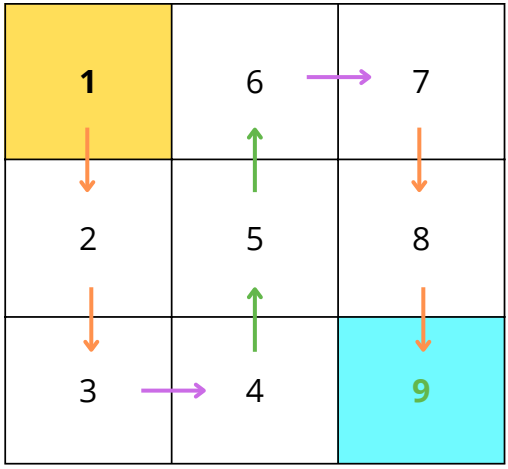
(DDRUURDD, 2, 2) ---> 1st Path

Here,
Making false means I have that cell in my current path. So, when that path is over for example you are in another cell, those cells should not be false.



(DDRUUR, 0, 2) ---> 1st Path

While you are moving back, you restore the maze as it was.



(DDRUURDD, 2, 2) ---> 1st Path

DDRUURDD
[1, 6, 7]
[2, 5, 8]
[3, 4, 9]

- **What is the output of the following Java program fragment:**

```
import java.util.ArrayList;

class Test{
 public static void main(String[] args) {
  boolean[][] board = {
    {true, true, true},
    {true, true, true},
    {true, true, true},
  };

  System.out.println(countPath(0, board, 0, 0));
 }

 static int countPath(int n, boolean[][] maze, int r,
int c){
  if(r == maze.length - 1 && c == maze[0].length - 1){
    n++;
    return n;
  }

  if(maze[r][c] == false){
    return n;
  }

  // i am considering this block in my path
  maze[r][c] = false;
```

```
  if(r < maze.length - 1){
    n = countPath(n, maze, r+1, c);
  }

  if(r > 0){
    n = countPath(n, maze, r-1, c);
  }

  if(c > 0){
    n = countPath(n, maze, r, c-1);
  }

  if(c < maze[0].length - 1){
    n = countPath(n, maze, r, c+1);
  }

  /* this line is whre the function will be over
    so before the function gets removed, also
    remove the changes that were made by that function */
  maze[r][c] = true;

  return n;
 }
}
```

Output:

12

- **What is the output of the following Java program fragment:**

```
import java.util.ArrayList;

class Test{
 public static void main(String[] args) {
  boolean[][] board = {
    {true, true, true},
    {true, true, true},
    {true, true, true},
  };

  path("", board, 0, 0);
 }

 static void path(String p, boolean[][] maze, int r, int
c){
  if(r == maze.length - 1 && c == maze[0].length - 1){
    System.out.println(p);
    return;
  }

  if(maze[r][c] == false){
    return ;
  }

  // i am considering this block in my path
  maze[r][c] = false;
```

```
  if(r < maze.length - 1){
    path(p + 'D', maze, r+1, c);
  }

  if(r > 0){
    path(p + 'U', maze, r-1, c);
  }

  if(c > 0){
    path(p + 'L', maze, r, c-1);
  }

  if(c < maze[0].length - 1){
    path(p + 'R', maze, r, c+1);
  }

  /* this line is whre the function will be over
    so before the function gets removed, also
    remove the changes that were made by that function */
  maze[r][c] = true;

 }
}
```

Output:

DDRUURDD
DDRURD
DDRR
DRDR
DRURDD
DRRD
RDDR
RDLDRR
RDRD
RRDD
RRDLDR
RRDLLDRR

- **What is the output of the following Java program fragment:**

```
import java.util.ArrayList;

class Test{
 public static void main(String[] args) {
  boolean[][] board = {
    {true, true, true},
    {true, true, true},
    {true, true, true},
  };

  System.out.println(path("", board, 0, 0));
 }

 static ArrayList<String> path(String p, boolean[][]
maze, int r, int c){
  if(r == maze.length - 1 && c == maze[0].length - 1){
    ArrayList<String> list = new ArrayList<String>();
    list.add(p);
    return list;
  }

  if(maze[r][c] == false){
    ArrayList<String> list = new ArrayList<String>();
    return list;
  }

  // i am considering this block in my path
  maze[r][c] = false;
```

```
  ArrayList<String> ans = new ArrayList<>();

  if(r < maze.length - 1){
    ans.addAll(path(p + 'D', maze, r+1, c));
  }

  if(r > 0){
    ans.addAll(path(p + 'U', maze, r-1, c));
  }

  if(c > 0){
    ans.addAll(path(p + 'L', maze, r, c-1));
  }

  if(c < maze[0].length - 1){
    ans.addAll(path(p + 'R', maze, r, c+1));
  }

  /* this line is whre the function will be over
    so before the function gets removed, also
    remove the changes that were made by that function */
  maze[r][c] = true;

  return ans;
 }
}
```

Output:

[DDRUURDD, DDRURD, DDRR, DRDR, DRURDD, DRRD, RDDR, RDLDRR, RDRD, RRDD, RRDLDR, RRDLLDRR]

- **What is the output of the following Java program fragment:**

```
import java.util.ArrayList;
import java.util.Arrays;

class Test{
 public static void main(String[] args) {
 boolean[][] board = {
  {true, true, true},
  {true, true, true},
  {true, true, true},
 };
 int[][] pathStep = new int[board.length]
[board[0].length];
 path("", board, 0, 0, pathStep, 1);
 }

 static void path(String p, boolean[][] maze, int r, int
c, int[][] pathStep, int step){
 if(r == maze.length - 1 && c == maze[0].length - 1){
  System.out.println(p);
  pathStep[r][c] = step;
  for (int[] arr : pathStep) {
    System.out.println(Arrays.toString(arr));
  }
  return;
 }

 if(maze[r][c] == false){
  return ;
 }
```

```
// i am considering this block in my path
maze[r][c] = false;
pathStep[r][c] = step;

if(r < maze.length - 1){
 path(p + 'D', maze, r+1, c, pathStep, step+1);
}

if(r > 0){
 path(p + 'U', maze, r-1, c, pathStep, step+1);
}

if(c > 0){
 path(p + 'L', maze, r, c-1, pathStep, step+1);
}

if(c < maze[0].length - 1){
 path(p + 'R', maze, r, c+1, pathStep, step+1);
}

/* this line is whre the function will be over
 so before the function gets removed, also
 remove the changes that were made by that function */
maze[r][c] = true;
pathStep[r][c] = 0;
}
}
```

Output:
DDRUURDD
[1, 6, 7]
[2, 5, 8]
[3, 4, 9]

DDRURD
[1, 0, 0]
[2, 5, 6]
[3, 4, 7]

DDRR
[1, 0, 0]
[2, 0, 0]
[3, 4, 5]

DRDR
[1, 0, 0]
[2, 3, 0]
[0, 4, 5]

DRURDD
[1, 4, 5]
[2, 3, 6]
[0, 0, 7]

DRRD
[1, 0, 0]
[2, 3, 4]
[0, 0, 5]

RDDR
[1, 2, 0]
[0, 3, 0]
[0, 4, 5]

RDLDRR
[1, 2, 0]
[4, 3, 0]
[5, 6, 7]

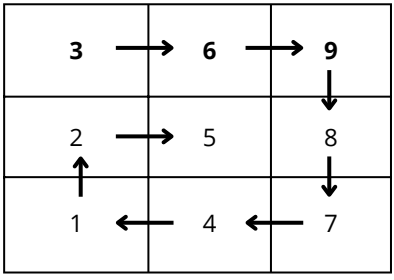
RDRD
[1, 2, 0]
[0, 3, 4]
[0, 0, 5]

RRDD
[1, 2, 3]
[0, 0, 4]
[0, 0, 5]

RRDLDR
[1, 2, 3]
[0, 5, 4]
[0, 6, 7]

RRDLLDRR
[1, 2, 3]
[6, 5, 4]
[7, 8, 9]

- **Write a program that takes input of the number of rows and columns of a matrix, followed by the values of the matrix. The program should have to print the matrix in a specific pattern that we provided using recursion.**



Input:
Enter row : 3
Enter column : 3
Enter value:
3 6 9
2 5 8
1 4 7

Output:
3 6 9 8 7 4 1 2 5

```
import java.util.Scanner;

public class Test {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the number of rows: ");
        int rows = sc.nextInt();

        System.out.print("Enter the number of columns: ");
        int cols = sc.nextInt();

        int[][] matrix = new int[rows][cols];

        System.out.println("Enter the values of the matrix:");

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = sc.nextInt();
            }
        }

        printMatrixInPattern(matrix, 0, 0, rows, cols);
    }

    public static void printMatrixInPattern(int[][] matrix, int
startRow, int startCol, int numRows, int numCols) {
        if (numRows <= 0 || numCols <= 0) {
            return;
        }
```

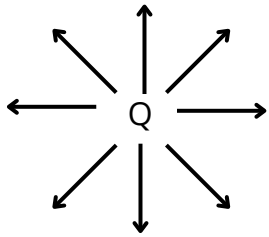
```
        for (int j = 0; j < numCols; j++) {
            System.out.print(matrix[startRow][startCol + j] +
" ");
        }

        for (int i = 1; i < numRows; i++) {
            System.out.print(matrix[startRow + i][startCol +
numCols - 1] + " ");
        }

        if (numRows > 1) {
            for (int j = numCols - 2; j >= 0; j--) {
                System.out.print(matrix[startRow + numRows
- 1][startCol + j] + " ");
            }
        }

        if (numCols > 1) {
            for (int i = numRows - 2; i > 0; i--) {
                System.out.print(matrix[startRow + i][startCol]
+ " ");
            }
        }

        printMatrixInPattern(matrix, startRow + 1, startCol
+ 1, numRows - 2, numCols - 2);
    }
}
```

Output:

```
Enter the number of rows: 3
Enter the number of columns: 3
Enter the values of the matrix:
3 6 9
2 5 8
1 4 7
3 6 9 8 7 4 1 2 5
```

- **N-Queens:**

n = 4
4 Queens

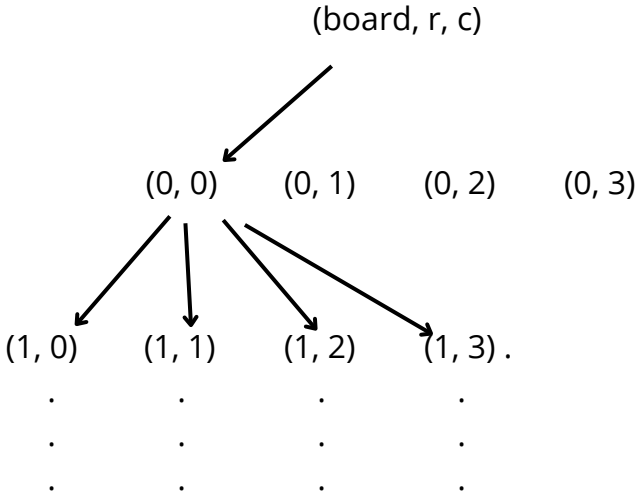


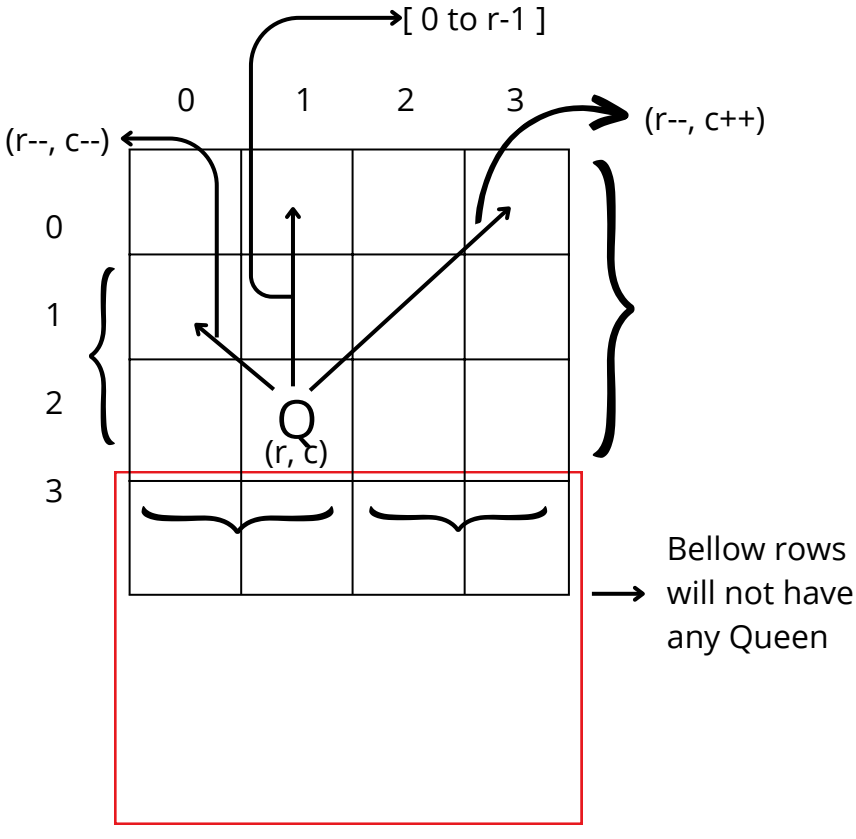
Q			
		Q	

❎

	Q		
			Q
Q			
		Q	

✅


(board, r, c)
(0, 0)
(0, 1)
(0, 2)
(0, 3)
(1, 0)
(1, 1)
(1, 2)
(1, 3)


[0 to r-1]
0
1
2
3
(r--, c--)
(r--, c++)
0
1
2
3
Q
(r, c)
Bellow rows
will not have
any Queen

- **What is the output of the following Java program fragment:**

```
import java.util.Scanner;

public class Test {

  public static void main(String[] args) {
    int n = 4;
    boolean[][] board = new boolean[n][n];
    System.out.println(queens(board, 0));
  }

  static int queens(boolean[][] board, int row){
    if(row == board.length){
      display(board);
      System.out.println();
      return 1;
    }

    int count = 0;

    // placing the queen and checking for every row
and col
    for(int col=0; col<board.length; col++){
      // place the queen if it is safe
      if(isSafe(board, row, col)){
        board[row][col] = true;
        count += queens(board, row + 1);
        board[row][col] = false;
      }
    }
```

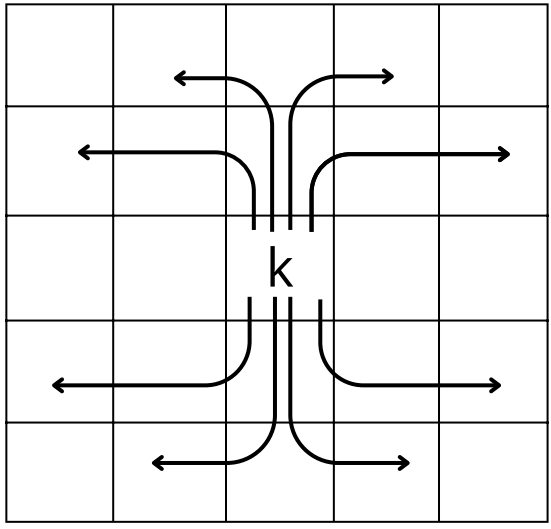
```
    return count;
  }

  private static boolean isSafe(boolean[][] board, int row,
int col) {
    // check vertical row
    for(int i=0; i<row; i++){
      if(board[i][col]){
        return false;
      }
    }

    // diagonal left
    int maxLeft = Math.min(row, col);
     for(int i=1; i<=maxLeft; i++){
      if(board[row-i][col-i]){
        return false;
      }
    }

    //diagonal right
    int maxRight = Math.min(row, board.length-col-1);
    for(int i=1; i<=maxRight; i++){
      if(board[row-i][col+i]){
        return false;
      }
    }

    return true;
  }
```

```
  private static void display(boolean[][] board){
    for(boolean[] row : board){
      for(boolean element : row){
        if(element){
          System.out.print("Q ");
        } else {
          System.out.print("X ");
        }
      }
      System.out.println();
    }
  }
}
```

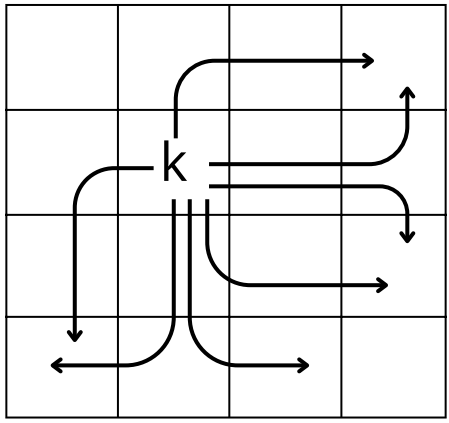
Output:

```
X Q X X
X X X Q
Q X X X
X X Q X

X X Q X
Q X X X
X X X Q
X Q X X

2
```

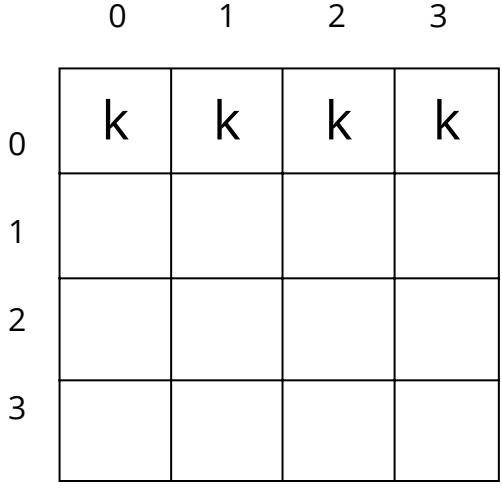
- **N-Knights:**

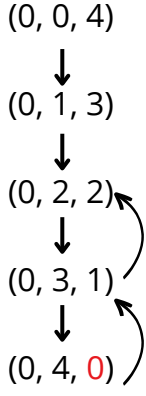


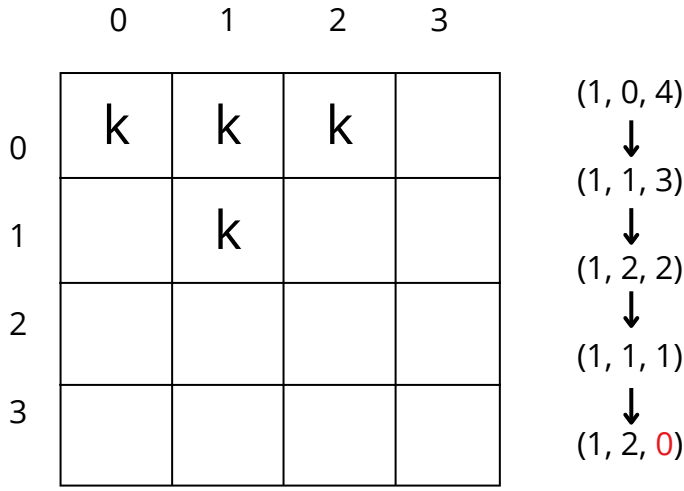
$$\begin{bmatrix} r-2, & c-1 \\ r-2, & c+1 \\ r-1, & c+2 \\ r-1, & c-2 \end{bmatrix}$$



$$\begin{bmatrix} r-2, & c-1 \\ r-2, & c+1 \\ r-1, & c+2 \\ r-1, & c-2 \end{bmatrix}$$






0
1
2
3
k
k
k
k
(1, 0, 4)
(1, 1, 3)
(1, 2, 2)
(1, 1, 1)
(1, 2, 0)

- **What is the output of the following Java program fragment:**

```
//package com.kunal.backtracking;

public class Hamim {
    public static void main(String[] args) {
        int n = 4;
        boolean[][] board = new boolean[n][n];
        knight(board, 0, 0, 4);
    }

    static void knight(boolean[][] board, int row, int col, int
knights) {
        if (knights == 0) {
            display(board);
            System.out.println();
            return;
        }

        if (row == board.length - 1 && col == board.length) {
            return;
        }

        if (col == board.length) {
            knight(board, row + 1, 0, knights);
            return;
        }

        if (isSafe(board, row, col)) {
            board[row][col] = true;
            knight(board, row, col + 1, knights - 1);
            board[row][col] = false;
```

```
    }
      knight(board, row, col + 1, knights);
    }

    private static boolean isSafe(boolean[][] board, int row,
int col) {
        if (isValid(board, row - 2, col - 1)) {
            if (board[row - 2][col - 1]) {
                return false;
            }
        }

        if (isValid(board, row - 1, col - 2)) {
            if (board[row - 1][col - 2]) {
                return false;
            }
        }

        if (isValid(board, row - 2, col + 1)) {
            if (board[row - 2][col + 1]) {
                return false;
            }
        }

        if (isValid(board, row - 1, col + 2)) {
            if (board[row - 1][col + 2]) {
                return false;
            }
        }

        return true;
    }
```

```
    // do not repeat yourself, hence created this function
    static boolean isValid(boolean[][] board, int row, int col) {
        if (row >= 0 && row < board.length && col >= 0 && col
< board.length) {
            return true;
        }
        return false;
    }

    private static void display(boolean[][] board) {
        for(boolean[] row : board) {
            for(boolean element : row) {
                if (element) {
                    System.out.print("K ");
                } else {
                    System.out.print("X ");
                }
            }
            System.out.println();
        }
    }
}
```

Output:

K K K K
X X X X
X X X X
X X X X

K K K X
X K X X
X X X X
X X X X

K K K X
X X X X
X X X X
K X X X

.....
.....
.....

- **Sudoku Solver :**

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy all of the following rules:

1. Each of the digits 1-9 must occur exactly once in each row.
2. Each of the digits 1-9 must occur exactly once in each column.
3. Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

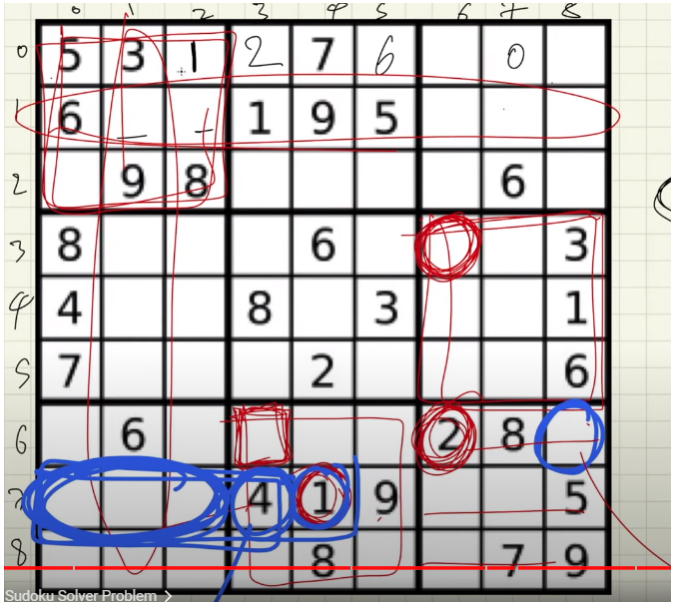
The '.' character indicates empty cells.

Input:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Output:

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9



- **What is the output of the following Java program fragment:**

```
public class Test {
   public static void main(String[] args) {
      int[][] board = new int[][]{
            {3, 0, 6, 5, 0, 8, 4, 0, 0},
            {5, 2, 0, 0, 0, 0, 0, 0, 0},
            {0, 8, 7, 0, 0, 0, 0, 3, 1},
            {0, 0, 3, 0, 1, 0, 0, 8, 0},
            {9, 0, 0, 8, 6, 3, 0, 0, 5},
            {0, 5, 0, 0, 9, 0, 6, 0, 0},
            {1, 3, 0, 0, 0, 0, 2, 5, 0},
            {0, 0, 0, 0, 0, 0, 0, 7, 4},
            {0, 0, 5, 2, 0, 6, 3, 0, 0}
      };

      if (solve(board)) {
         display(board);
      } else {
         System.out.println("Cannot solve");
      }

   }

   static boolean solve(int[][] board) {
      int n = board.length;
      int row = -1;
      int col = -1;

      boolean emptyLeft = true;
```

```
        // this is how we are replacing the r,c from arguments
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (board[i][j] == 0) {
                    row = i;
                    col = j;
                    emptyLeft = false;
                    break;
                }
            }
            // if you found some empty element in row, then
break
            if (emptyLeft == false) {
                break;
            }
        }

        if (emptyLeft == true) {
            return true;
            // soduko is solved
        }

        // backtrack
        for (int number = 1; number <= 9; number++) {
            if (isSafe(board, row, col, number)) {
                board[row][col] = number;
                if (solve(board)) {
                    // found the answer
                    return true;
                } else {
                    // backtrack
```

```
                board[row][col] = 0;
            }
        }
    }
    return false;
}

private static void display(int[][] board) {
for(int[] row : board) {
for(int num : row) {
System.out.print(num + " ");
}
System.out.println();
}
}


static boolean isSafe(int[][] board, int row, int col,
int num) {
// check the row
for (int i = 0; i < board.length; i++) {
// check if the number is in the row
if (board[row][i] == num) {
return false;
}
}

// check the col
for (int[] nums : board) {
// check if the number is in the col
```

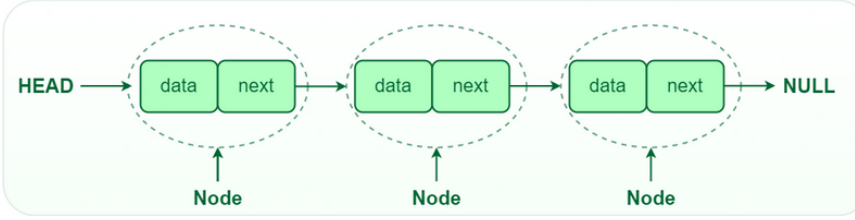
```
if (nums[col] == num) {
return false;
}
}

int sqrt = (int)(Math.sqrt(board.length));
int rowStart = row - row % sqrt;
int colStart = col - col % sqrt;

for (int r = rowStart; r < rowStart + sqrt; r++) {
for (int c = colStart; c < colStart + sqrt; c++) {
if (board[r][c] == num) {
return false;
}
}
}
return true;
}
}
```

Output:

3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9

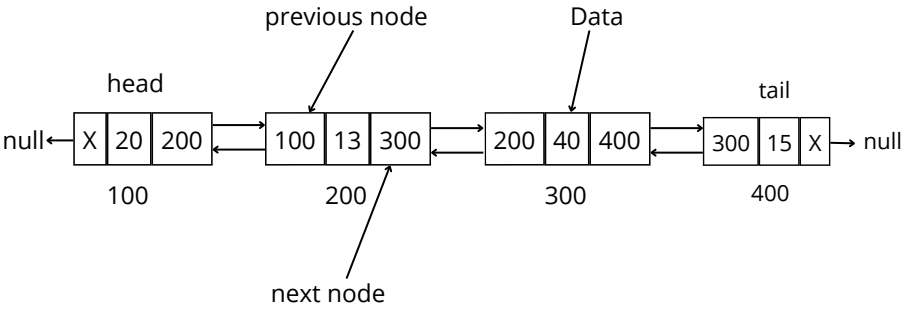
5. LinkedList

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers/reference variables as shown in the below image:

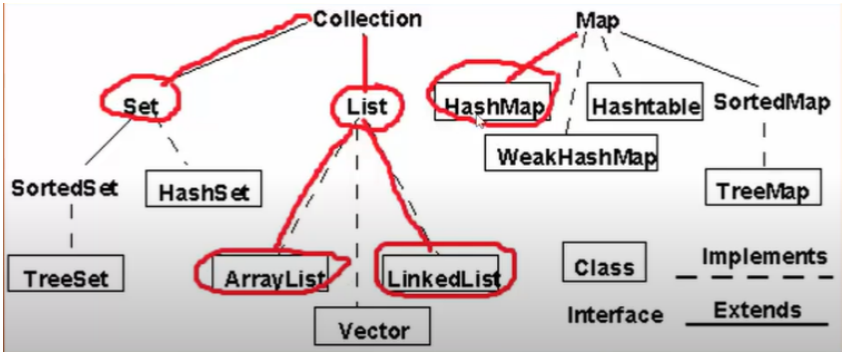


In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

- **Node Structure:** A node in a linked list typically consists of two components:
- **Data:** It holds the actual value or data associated with the node.
- **Next Pointer:** It stores the memory address (reference) of the next node in the sequence.
- **Head and Tail:** The linked list is accessed through the head node, which points to the first node in the list. The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the tail node.



1. LinkedList class uses double linked list to store the elements.
2. manipulating of data is fast here (deleting or inserting data)
3. can contain duplicate elements.



Why linked list data structure needed?

Here are a few advantages of a linked list that is listed below, it will help you understand why it is necessary to know.

- **Dynamic Data structure:** The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
- **Ease of Insertion/Deletion:** The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
- **Efficient Memory Utilization:** As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
- **Implementation:** Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

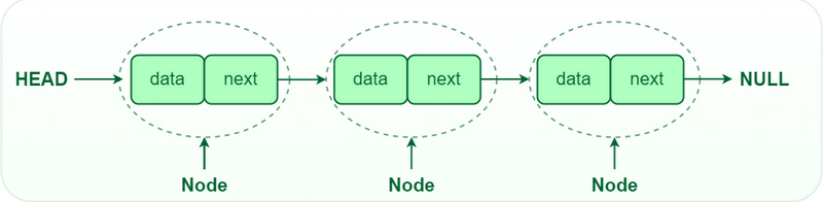
Types of linked lists:

There are mainly three types of linked lists:

- Single-linked list
- Double linked list
- Circular linked list

1. Single-linked list:

In a singly linked list, each node contains a reference to the next node in the sequence. Traversing a singly linked list is done in a forward direction.

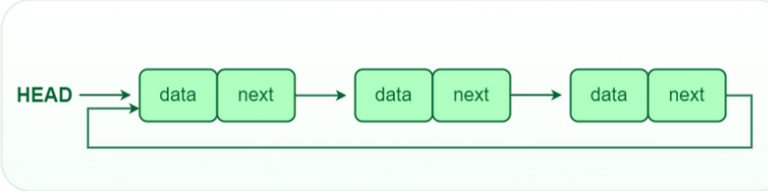


2. Double-linked list:

In a doubly linked list, each node contains references to both the next and previous nodes. This allows for traversal in both forward and backward directions, but it requires additional memory for the backward reference.

3. Circular linked list:

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



Operations on Linked Lists

1. **Insertion:** Adding a new node to a linked list involves adjusting the pointers of the existing nodes to maintain the proper sequence. Insertion can be performed at the beginning, end, or any position within the list
2. **Deletion:** Removing a node from a linked list requires adjusting the pointers of the neighboring nodes to bridge the gap left by the deleted node. Deletion can be performed at the beginning, end, or any position within the list.
3. **Searching:** Searching for a specific value in a linked list involves traversing the list from the head node until the value is found or the end of the list is reached.

Advantages of Linked Lists:

- **Dynamic Size:** Linked lists can grow or shrink dynamically, as memory allocation is done at runtime.
- **Insertion and Deletion:** Adding or removing elements from a linked list is efficient, especially for large lists.
- **Flexibility:** Linked lists can be easily reorganized and modified without requiring a contiguous block of memory.

Disadvantages of Linked Lists:

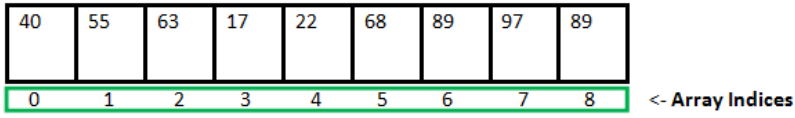
- **Random Access:** Unlike arrays, linked lists do not allow direct access to elements by index. Traversal is required to reach a specific node.
- **Extra Memory:** Linked lists require additional memory for storing the pointers, compared to arrays.

Linked lists are versatile data structures that provide dynamic memory allocation and efficient insertion and deletion operations. Understanding the basics of linked lists is essential for any programmer or computer science enthusiast. With this knowledge, you can implement linked lists to solve various problems and expand your understanding of data structures and algorithms.

Linked List vs Array:

Array:

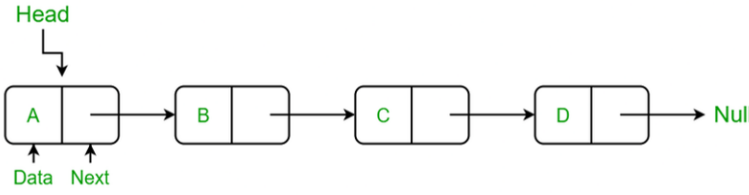
Arrays store elements in contiguous memory locations, resulting in easily calculable addresses for the elements stored and this allows faster access to an element at a specific index.



Array Length = 9
First Index = 0
Last Index = 8

Linked List:

Linked lists are less rigid in their storage structure and elements are usually not stored in contiguous locations, hence they need to be stored with additional tags giving a reference to the next element.



Major differences between array and linked-list are listed below:

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

Basic Operations:

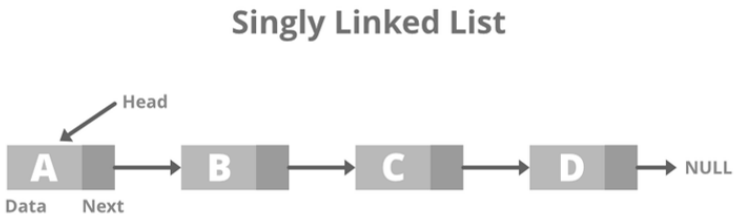
- Linked List Insertion
- Search an element in a Linked List (Iterative and Recursive)
- Find Length of a Linked List (Iterative and Recursive)
- Reverse a linked list
- Linked List Deletion (Deleting a given key)
- Linked List Deletion (Deleting a key at given position)
- Write a function to delete a Linked List
- Write a function to get Nth node in a Linked List
- Nth node from the end of a Linked List

Types Of Linked List:

1. Singly Linked List:

It is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.

The node contains a pointer to the next node means that the node stores the address of the next node in the sequence. A single linked list allows the traversal of data only in one way. Below is the image for the same:



- **Singly LinkedList implementation in Java :**

```
// package com.kunal;

public class LL {

    private class Node {
        private int value;
        private Node next;

        public Node(int value) {
            this.value = value;
        }

        public Node(int value, Node next) {
            this.value = value;
            this.next = next;
        }
    }

    private Node head;
    private Node tail;
    private int size;

    public LL() {
        this.size = 0;
    }

    public void insertFirst(int val) {
        Node node = new Node(val);
        node.next = head;
        head = node;
```

```
        if (tail == null) {
            tail = head;
        }
        size += 1;
    }

    public void insertLast(int val) {
        if (tail == null) {
            insertFirst(val);
            return;
        }
        Node node = new Node(val);
        tail.next = node;
        tail = node;
        size++;
    }

  public void insert(int val, int index) {
    if (index == 0) {
      insertFirst(val);
      return;
    }
    if (index == size) {
      insertLast(val);
      return;
    }

    Node temp = head;
    for (int i = 1; i < index; i++) {
      temp = temp.next;
    }
```

```
  Node node = new Node(val, temp.next);
  temp.next = node;

  size++;
}

// insert using recursion
public void insertRec(int val, int index) {
  head = insertRec(val, index, head);
}

private Node insertRec(int val, int index, Node node) {
  if (index == 0) {
    Node temp = new Node(val, node);
    size++;
    return temp;
  }

  node.next = insertRec(val, index-1, node.next);
  return node;
}


public int deleteLast() {
  if (size <= 1) {
    return deleteFirst();
  }

  Node secondLast = get(size - 2);
  int val = tail.value;
```

```
  tail = secondLast;
  tail.next = null;
  size--;
  return val;
}

public int delete(int index) {
  if (index == 0) {
    return deleteFirst();
  }
  if (index == size - 1) {
    return deleteLast();
  }

  Node prev = get(index - 1);
  int val = prev.next.value;

  prev.next = prev.next.next;
  size--;
  return val;
}

public Node find(int value) {
  Node node = head;
  while (node != null) {
    if (node.value == value) {
      return node;
    }
    node = node.next;
  }
  return null;
}
```

```
public Node get(int index) {
  Node node = head;
  for (int i = 0; i < index; i++) {
    node = node.next;
  }
  return node;
}

public int deleteFirst() {
  int val = head.value;
  head = head.next;
  if (head == null) {
    tail = null;
  }
  size--;
  return val;
}

public void display() {
  Node temp = head;
  while (temp != null) {
    System.out.print(temp.value + " -> ");
    temp = temp.next;
  }
  System.out.println("END");
}


public static void main(String[] args) {
  LL list = new LL();
```

```
    list.insertFirst(3);
    list.insertFirst(2);
    list.insertFirst(8);
    list.insertFirst(17);
    list.insertLast(99);
    list.insert(100, 3);
    list.display();
    System.out.println(list.deleteFirst());
    list.display();
    System.out.println(list.deleteLast());
    list.display();
  }

}
```

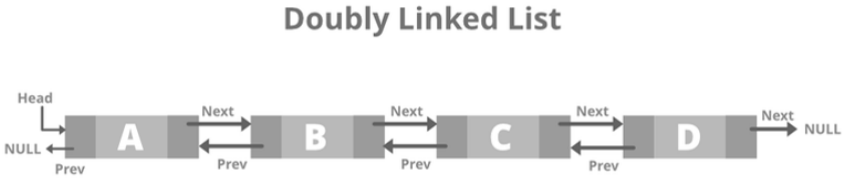
Output:

```
17 -> 8 -> 2 -> 100 -> 3 -> 99 -> END
17
8 -> 2 -> 100 -> 3 -> 99 -> END
99
8 -> 2 -> 100 -> 3 -> END
```

2. Doubly Linked List:

A doubly linked list or a two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in sequence.

Therefore, it contains three parts of data, a pointer to the next node, and a pointer to the previous node. This would enable us to traverse the list in the backward direction as well. Below is the image for the same:



- **Doubly LinkedList implementation in Java :**

```
//package com.kunal;

public class DLL {

    private class Node {
        int val;
        Node next;
        Node prev;

        public Node(int val) {
            this.val = val;
        }

        public Node(int val, Node next, Node prev) {
            this.val = val;
            this.next = next;
            this.prev = prev;
        }
    }

    private Node head;

    public void insertFirst(int val) {
        Node node = new Node(val);
        node.next = head;
        node.prev = null;
        if (head != null) {
            head.prev = node;
        }
```

```
    head = node;
}

public void display() {
    Node node = head;
    Node last = null;
    while (node != null) {
        System.out.print(node.val + " -> ");
        last = node;
        node = node.next;
    }
    System.out.println("END");

    System.out.println("Print in reverse");
    while (last != null) {
        System.out.print(last.val + " -> ");
        last = last.prev;
    }
    System.out.println("START");
}

public void insertLast(int val) {
    Node node = new Node(val);
    Node last = head;

    node.next = null;

    if (head == null) {
        node.prev = null;
        head = node;
        return;
    }
```

```
    while (last.next != null) {
        last = last.next;
    }

    last.next = node;
    node.prev = last;
}

public Node find(int value) {
    Node node = head;
    while (node != null) {
        if (node.val == value) {
            return node;
        }
        node = node.next;
    }
    return null;
}

public void insert(int after, int val) {
    Node p = find(after);

    if (p == null) {
        System.out.println("does not exist");
        return;
    }

    Node node = new Node(val);
    node.next = p.next;
    p.next = node;
    node.prev = p;
```

```
        if (node.next != null) {
            node.next.prev = node;
        }
    }

    public static void main(String[] args) {
        DLL list = new DLL();
        list.insertFirst(3);
        list.insertFirst(2);
        list.insertFirst(8);
        list.insertFirst(17);
        list.insertLast(99);
        list.insert(8, 65);

        list.display();

    }

}
```

Output:

```
17 -> 8 -> 65 -> 2 -> 3 -> 99 -> END
Print in reverse
99 -> 3 -> 2 -> 65 -> 8 -> 17 -> START
```

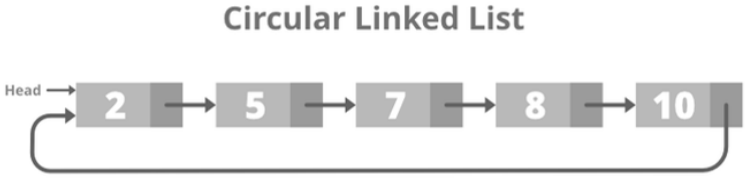
Time Complexity:
The time complexity of the push() function is O(1) as it performs constant-time operations to insert a new node at the beginning of the doubly linked list. The time complexity of the printList() function is O(n) where n is the number of nodes in the doubly linked list. This is because it traverses the entire list twice, once in the forward direction and once in the backward direction. Therefore, the overall time complexity of the program is O(n).

Space Complexity:
The space complexity of the program is O(n) as it uses a doubly linked list to store the data, which requires n nodes. Additionally, it uses a constant amount of auxiliary space to create a new node in the push() function. Therefore, the overall space complexity of the program is O(n).

3. Circular Linked List:

A circular linked list is that in which the last node contains the pointer to the first node of the list.

While traversing a circular linked list, we can begin at any node and traverse the list in any direction forward and backward until we reach the same node we started. Thus, a circular linked list has no beginning and no end. Below is the image for the same:



- **Circular LinkedList implementation in Java :**

```
// package com.kunal;

public class CLL {

    private class Node {
        int val;
        Node next;

        public Node(int val) {
            this.val = val;
        }
    }

    private Node head;
    private Node tail;

    public CLL() {
        this.head = null;
        this.tail = null;
    }

    public void insert(int val) {
        Node node = new Node(val);
        if (head == null) {
            head = node;
            tail = node;
            return;
        }
```

```
    tail.next = node;
    node.next = head;
    tail = node;
}

public void display() {
    Node node = head;
    if (head != null) {
        do {
            System.out.print(node.val + " -> ");
            if (node.next != null) {
                node = node.next;
            }
        } while (node != head);
    }
    System.out.println("HEAD");
}

public void delete(int val) {
    Node node = head;
    if (node == null) {
        return;
    }

    if (head == tail){
        head = null;
        tail = null;
        return;
    }
```

```
        if (node.val == val) {
            head = head.next;
            tail.next = head;
            return;
        }

        do {
            Node n = node.next;
            if (n.val == val) {
                node.next = n.next;
                break;
            }
            node = node.next;
        } while (node != head);

    }

    public static void main(String[] args) {
        CLL list = new CLL();
        list.insert(23);
        list.insert(3);
        list.insert(19);
        list.insert(75);
        list.display();
        list.delete(19);
        list.display();
    }
}
```

Output:

23 -> 3 -> 19 -> 75 -> HEAD
23 -> 3 -> 75 -> HEAD

Time Complexity:

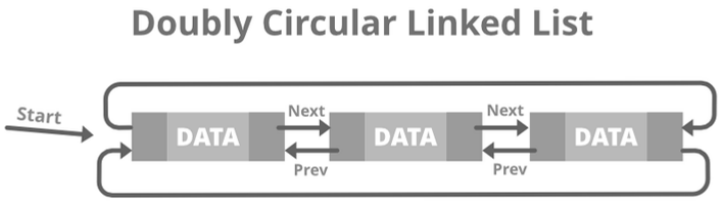
Insertion at the beginning of the circular linked list takes O(1) time complexity.
Traversing and printing all nodes in the circular linked list takes O(n) time complexity where n is the number of nodes in the linked list.
Therefore, the overall time complexity of the program is O(n).

Auxiliary Space:

The space required by the program depends on the number of nodes in the circular linked list.
In the worst-case scenario, when there are n nodes, the space complexity of the program will be O(n) as n new nodes will be created to store the data.
Additionally, some extra space is required for the temporary variables and the function calls.
Therefore, the auxiliary space complexity of the program is O(n).

4. Doubly Circular linked list:

A Doubly Circular linked list or a circular two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in the sequence. The difference between the doubly linked and circular doubly list is the same as that between a singly linked list and a circular linked list. The circular doubly linked list does not contain null in the previous field of the first node. Below is the image for the same:



- **Doubly Circular LinkedList implementation in Java :**

Time Complexity:

Insertion at the beginning of a doubly circular linked list takes O(1) time complexity.
Traversing the entire doubly circular linked list takes O(n) time complexity, where n is the number of nodes in the linked list.
Therefore, the overall time complexity of the program is O(n).

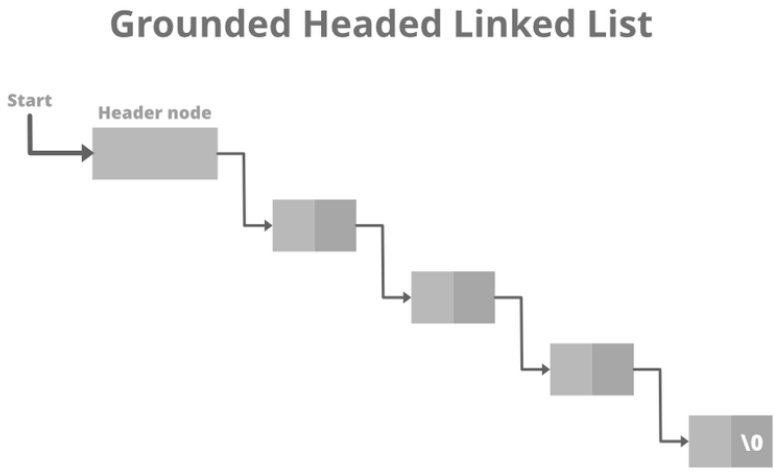
Auxiliary space:

The program uses a constant amount of auxiliary space, i.e., O(1), to create and traverse the doubly circular linked list.
The space required to store the linked list grows linearly with the number of nodes in the linked list.
Therefore, the overall auxiliary space complexity of the program is O(1).

5. Header Linked List:

A header linked list is a special type of linked list that contains a header node at the beginning of the list.

So, in a header linked list START will not point to the first node of the list but START will contain the address of the header node. Below is the image for Grounded Header Linked List:



- **Header LinkedList implementation in Java :**

Time Complexity:
The time complexity of creating a new node and inserting it at the end of the linked list is O(1).
The time complexity of traversing the linked list to display its contents is O(n), where n is the number of nodes in the list.
Therefore, the overall time complexity of creating and traversing a header linked list is O(n).

Auxiliary Space:
The space complexity of the program is O(n), where n is the number of nodes in the linked list. This is because we are creating n nodes, each with a fixed amount of space required for storing the node information and a pointer to the next node.
Therefore, the overall auxiliary space complexity of the program is O(n).

- **Additional Types:**

Multiply Linked List:
Multiply Linked List is a data structure in which each node of the list contains multiple pointers. It is a type of linked list which has multiple linked lists in one list. Each node has multiple pointers which can point to different nodes in the list and can also point to nodes outside the list. The data stored in a Multiply Linked List can be easily accessed and modified, making it a very efficient data structure. The nodes in a Multiply Linked List can be accessed in any order, making it suitable for applications such as graphs, trees, and cyclic lists.

Built in LinkedList :

There are already built in LinkedList. We don’t need to implement them. We can simply use their functionalities without doing anything.

All methods related to LinkedList:

- **add(element):** Adds an element to the end of the list.
- **add(index, element):** Inserts an element at the specified index.
- **addAll(collection):** Adds all elements from a collection to the end of the list.
- **addAll(index, collection):** Inserts all elements from a collection at the specified index.
- **addFirst(element):** Inserts an element at the beginning of the list.
- **addLast(element):** Adds an element to the end of the list.
- **clear():** Removes all elements from the list.
- **clone():** Returns a shallow copy of the list.
- **contains(element):** Checks if the list contains the specified element.
- **descendingIterator():** Returns an iterator over the elements in reverse order.
- **get(index):** Retrieves the element at the specified index.
- **getFirst():** Returns the first element in the list.
- **getLast():** Returns the last element in the list.
- **indexOf(element):** Returns the index of the first occurrence of the specified element.
- **lastIndexOf(element):** Returns the index of the last occurrence of the specified element.

- **listIterator():** Returns a list iterator over the elements in the list.
- **remove(index):** Removes the element at the specified index.
- **remove(element):** Removes the first occurrence of the specified element.
- **removeFirst():** Removes and returns the first element in the list.
- **removeFirstOccurrence(element):** Removes the first occurrence of the specified element.
- **removeLast():** Removes and returns the last element in the list.
- **removeLastOccurrence(element):** Removes the last occurrence of the specified element.
- **size():** Returns the number of elements in the list.
- **toArray():** Returns an array containing all the elements in the list.
- **toArray(array):** Returns an array containing all the elements in the list, using the specified array if it is large enough.
- **toString():** Returns a string representation of the list.
- **subList(fromIndex, toIndex):** Returns a view of the portion of the list between the specified fromIndex (inclusive) and toIndex (exclusive).
- **equals():** It is used to compare the contents of two LinkedList objects to check if they are equal. It returns true if the two lists have the same elements in the same order, and false otherwise.
- **Collections.sort(LinkedList variable):**

- **add(element):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        // Create a LinkedList
        LinkedList<String> linkedList = new LinkedList<>();

        // Adding elements to the LinkedList using
add(element)
        linkedList.add("Apple");
        linkedList.add("Banana");
        linkedList.add("Orange");

        // Displaying the LinkedList
        System.out.println("LinkedList: " + linkedList);
    }
}
```

Output:
LinkedList: [Apple, Banana, Orange]

- **add(index, element):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
   public static void main(String[] args) {

       LinkedList<String> linkedList = new LinkedList<>();

       linkedList.add("Apple");
       linkedList.add("Banana");
       linkedList.add("Orange");
       System.out.println("Original LinkedList: " +
linkedList);

       linkedList.add(1, "Mango");
       System.out.println("After adding Mango at index
1: " + linkedList);
   }
}
```

Output:
Original LinkedList: [Apple, Banana, Orange]
After adding Mango at index 1: [Apple, Mango, Banana, Orange]

- **addAll(collection):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;
import java.util.ArrayList;

public class LinkedListExample {
   public static void main(String[] args) {

      LinkedList<Integer> linkedList = new LinkedList<>();

      ArrayList<Integer> numbers = new ArrayList<>();
      numbers.add(1);
      numbers.add(2);
      numbers.add(3);

      linkedList.addAll(numbers);

      System.out.println("LinkedList: " + linkedList);
   }
}
```

Output:
LinkedList: [1, 2, 3]

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;
import java.util.Arrays;

public class LinkedListExample {
   public static void main(String[] args)

      LinkedList<String> linkedList = new LinkedList<>();

      String[] fruits = {"Apple", "Banana", "Orange"};

      linkedList.addAll(Arrays.asList(fruits));

      System.out.println("LinkedList: " + linkedList);
   }
}
```

Output:
LinkedList: [Apple, Banana, Orange]

- **addAll(index, collection):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;
import java.util.ArrayList;

public class LinkedListExample {
    public static void main(String[] args) {

        LinkedList<String> linkedList = new LinkedList<>();

        linkedList.add("Apple");
        linkedList.add("Banana");
        linkedList.add("Orange");

        ArrayList<String> fruits = new ArrayList<>();

        fruits.add("Mango");
        fruits.add("Grapes");

        linkedList.addAll(1, fruits);

        System.out.println("LinkedList: " + linkedList);
    }
}
```

Output:
LinkedList: [Apple, Mango, Grapes, Banana, Orange]

- **addFirst(element):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {

        LinkedList<String> linkedList = new LinkedList<>();

        linkedList.add("Banana");
        linkedList.add("Orange");

        linkedList.addFirst("Apple");

        System.out.println("LinkedList: " + linkedList);
    }
}
```

Output:
LinkedList: [Apple, Banana, Orange]

addLast(element):

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {

        LinkedList<String> linkedList = new LinkedList<>();

        linkedList.add("Apple");
        linkedList.add("Banana");

        linkedList.addLast("Orange");

        System.out.println("LinkedList: " + linkedList);
    }
}
```

Output:
LinkedList: [Apple, Banana, Orange]

- **clear():**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {

        LinkedList<String> linkedList = new LinkedList<>();

        linkedList.add("Apple");
        linkedList.add("Banana");
        linkedList.add("Orange");

        linkedList.clear();

        System.out.println("LinkedList after clearing: " +
linkedList);
    }
}
```

Output:
LinkedList after clearing: []

- **clone():**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
   public static void main(String[] args) {

       LinkedList<String> originalList = new LinkedList<>();

       originalList.add("Apple");
       originalList.add("Banana");
       originalList.add("Orange");

       LinkedList<String> clonedList = (LinkedList<String>)
originalList.clone();

       System.out.println("Original LinkedList: " +
originalList);
       System.out.println("Cloned LinkedList: " + clonedList);
   }
}
```

Output:
Original LinkedList: [Apple, Banana, Orange]
Cloned LinkedList: [Apple, Banana, Orange]

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
   public static void main(String[] args) {

      LinkedList<Integer> originalList = new LinkedList<>();

      originalList.add(1);
      originalList.add(2);
      originalList.add(3);

      LinkedList<Integer> clonedList =
(LinkedList<Integer>) originalList.clone();

      clonedList.add(4);
      clonedList.removeFirst();

      System.out.println("Original LinkedList: " +
originalList);
      System.out.println("Cloned LinkedList: " + clonedList);
   }
}
```

Output:
Original LinkedList: [1, 2, 3]
Cloned LinkedList: [2, 3, 4]

- **contains(element):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
   public static void main(String[] args) {

      LinkedList<String> linkedList = new LinkedList<>();

      linkedList.add("Apple");
      linkedList.add("Banana");
      linkedList.add("Orange");

      boolean containsBanana =
linkedList.contains("Banana");

      boolean containsGrapes =
linkedList.contains("Grapes");

      System.out.println("LinkedList contains 'Banana': " +
containsBanana);
      System.out.println("LinkedList contains 'Grapes': " +
containsGrapes);
   }
}
```

Output:

```
LinkedList contains 'Banana': true
LinkedList contains 'Grapes': false
```

- **descendingIterator():**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;
import java.util.Iterator;

public class LinkedListExample {
   public static void main(String[] args) {

      LinkedList<String> linkedList = new LinkedList<>();

      linkedList.add("Apple");
      linkedList.add("Banana");
      linkedList.add("Orange");

      Iterator<String> descendingIterator =
linkedList.descendingIterator();

      System.out.println("Elements in reverse order:");
      while (descendingIterator.hasNext()) {
         String element = descendingIterator.next();
         System.out.println(element);
      }
   }
}
```

Output:

```
Elements in reverse order:
Orange
Banana
Apple
```

- **get(index):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
   public static void main(String[] args) {

      LinkedList<String> linkedList = new LinkedList<>();

      linkedList.add("Apple");
      linkedList.add("Banana");
      linkedList.add("Orange");

      String element = linkedList.get(1);

      System.out.println("Element at index 1: " + element);
   }
}
```

Output:
Element at index 1: Banana

- **getFirst():**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {

        LinkedList<String> linkedList = new LinkedList<>();

        linkedList.add("Apple");
        linkedList.add("Banana");
        linkedList.add("Orange");

        String firstElement = linkedList.getFirst();

        System.out.println("First element: " + firstElement);
    }
}
```

Output:
First element: Apple

- **getLast():**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {

        LinkedList<String> linkedList = new LinkedList<>();

        linkedList.add("Apple");
        linkedList.add("Banana");
        linkedList.add("Orange");

        String lastElement = linkedList.getLast();

        System.out.println("Last element: " + lastElement);
    }
}
```

Output:
Last element: Orange

- **indexOf(element):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {

        LinkedList<String> linkedList = new LinkedList<>();

        linkedList.add("Apple");
        linkedList.add("Banana");
        linkedList.add("Orange");

        int index = linkedList.indexOf("Banana");

        System.out.println("Index of 'Banana': " + index);
    }
}
```

Output:
Index of 'Banana': 1

- **lastIndexOf(element):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {

        LinkedList<String> linkedList = new LinkedList<>();

        linkedList.add("Apple");
        linkedList.add("Banana");
        linkedList.add("Orange");
        linkedList.add("Banana");

        int lastIndex = linkedList.lastIndexOf("Banana");

        System.out.println("Last index of 'Banana': " +
lastIndex);
    }
}
```

Output:
Last index of 'Banana': 3

- **listIterator():**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;
import java.util.ListIterator;

public class LinkedListExample {
   public static void main(String[] args) {

      LinkedList<String> linkedList = new LinkedList<>();

      linkedList.add("Apple");
      linkedList.add("Banana");
      linkedList.add("Orange");

      ListIterator<String> iterator = linkedList.listIterator();

      System.out.println("Elements in forward direction:");
      while (iterator.hasNext()) {
         String element = iterator.next();
         System.out.println(element);
      }

      System.out.println("\nElements in reverse
direction:");
      while (iterator.hasPrevious()) {
         String element = iterator.previous();
         System.out.println(element);
      }
   }
}
```

Output:

```
Elements in forward direction:
Apple
Banana
Orange

Elements in reverse direction:
Orange
Banana
Apple
```

- **remove(index):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
   public static void main(String[] args) {

       LinkedList<String> linkedList = new LinkedList<>();

       linkedList.add("Apple");
       linkedList.add("Banana");
       linkedList.add("Orange");

       System.out.println("Original LinkedList: " + linkedList);
       String removedElement = linkedList.remove(1);

       System.out.println("Removed element: " +
removedElement);

       System.out.println("Updated LinkedList: " +
linkedList);
   }
}
```

Output:
Original LinkedList: [Apple, Banana, Orange]
Removed element: Banana
Updated LinkedList: [Apple, Orange]

- **remove(element):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
   public static void main(String[] args) {

       LinkedList<String> linkedList = new LinkedList<>();

       linkedList.add("Apple");
       linkedList.add("Banana");
       linkedList.add("Orange");

       System.out.println("Original LinkedList: " + linkedList);

       boolean removed = linkedList.remove("Banana");

       System.out.println("Element removed: " + removed);

       System.out.println("Updated LinkedList: " +
linkedList);
   }
}
```

Output:
Original LinkedList: [Apple, Banana, Orange]
Element removed: true
Updated LinkedList: [Apple, Orange]

- **removeFirst():**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
   public static void main(String[] args) {

       LinkedList<String> linkedList = new LinkedList<>();

       linkedList.add("Apple");
       linkedList.add("Banana");
       linkedList.add("Orange");

       System.out.println("Original LinkedList: " + linkedList);

       String removedElement = linkedList.removeFirst();

       System.out.println("Removed element: " +
removedElement);

       System.out.println("Updated LinkedList: " +
linkedList);
   }
}
```

Output:

```
Original LinkedList: [Apple, Banana, Orange]
Removed element: Apple
Updated LinkedList: [Banana, Orange]
```

- **removeFirstOccurrence(element):**
- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
   public static void main(String[] args) {

      LinkedList<String> linkedList = new LinkedList<>();

      linkedList.add("Apple");
      linkedList.add("Banana");
      linkedList.add("Orange");
      linkedList.add("Banana");

      System.out.println("Original LinkedList: " + linkedList);

      boolean removed =
linkedList.removeFirstOccurrence("Banana");

      System.out.println("Element removed: " + removed);

      System.out.println("Updated LinkedList: " +
linkedList);
   }
}
```

Output:

```
Original LinkedList: [Apple, Banana, Orange, Banana]
Element removed: true
Updated LinkedList: [Apple, Orange, Banana]
```

- **removeLast():**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
   public static void main(String[] args) {

      LinkedList<String> linkedList = new LinkedList<>();

      linkedList.add("Apple");
      linkedList.add("Banana");
      linkedList.add("Orange");

      System.out.println("Original LinkedList: " + linkedList);

      String removedElement = linkedList.removeLast();

      System.out.println("Removed element: " +
removedElement);

      System.out.println("Updated LinkedList: " +
linkedList);
   }
}
```

Output:

```
Original LinkedList: [Apple, Banana, Orange]
Removed element: Orange
Updated LinkedList: [Apple, Banana]
```

- **removeLastOccurrence(element):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class Test {
   public static void main(String[] args) {

      LinkedList<String> linkedList = new LinkedList<>();

      linkedList.add("Apple");
      linkedList.add("Banana");
      linkedList.add("Orange");
      linkedList.add("Banana");
      linkedList.add("Lemon");

      System.out.println("Original LinkedList: " + linkedList);

      boolean removed =
linkedList.removeLastOccurrence("Banana");

      System.out.println("Element removed: " + removed);

      System.out.println("Updated LinkedList: " +
linkedList);
   }
}
```

Output:

Original LinkedList: [Apple, Banana, Orange, Banana, Lemon]
Element removed: true
Updated LinkedList: [Apple, Banana, Orange, Lemon]

- **size():**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {

        LinkedList<String> linkedList = new LinkedList<>();

        linkedList.add("Apple");
        linkedList.add("Banana");
        linkedList.add("Orange");

        int size = linkedList.size();

        System.out.println("Size of the LinkedList: " + size);
    }
}
```

Output:
Size of the LinkedList: 3

- **toArray():**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class Test1 {
   public static void main(String[] args) {

      LinkedList<String> linkedList = new LinkedList<>();

      linkedList.add("Apple");
      linkedList.add("Banana");
      linkedList.add("Orange");

      Object[] array = linkedList.toArray();

      System.out.println("Array elements:");
      for (Object element : array) {
         System.out.println(element);
      }
   }
}
```

Output:
Array elements:
Apple
Banana
Orange

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class Test {
   public static void main(String[] args) {

      LinkedList<Integer> linkedList = new LinkedList<>();

      linkedList.add(1);
      linkedList.add(2);
      linkedList.add(2);

      Object[] array = linkedList.toArray();

      System.out.println("Array elements:");
      for (Object element : array) {
         System.out.println(element);
      }
   }
}
```

Output:

```
Array elements:
1
2
2
```

- **toArray(array):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
   public static void main(String[] args) {

      LinkedList<String> linkedList = new LinkedList<>();

      linkedList.add("Apple");
      linkedList.add("Banana");
      linkedList.add("Orange");

      String[] array = new String[linkedList.size()];

      array = linkedList.toArray(array);

      System.out.println("Array elements:");
      for (String element : array) {
         System.out.println(element);
      }
   }
}
```

Output:
Array elements:
Apple
Banana
Orange

- **toString():**
  - **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {

        LinkedList<String> linkedList = new LinkedList<>();

        linkedList.add("Apple");
        linkedList.add("Banana");
        linkedList.add("Orange");

        String stringRepresentation = linkedList.toString();

        System.out.println("String representation of the
LinkedList: " + stringRepresentation);
    }
}
```

Output:
String representation of the LinkedList: [Apple, Banana, Orange]

- **subList(fromIndex, toIndex):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;
import java.util.List;

public class Test {
   public static void main(String[] args) {

      LinkedList<String> linkedList = new LinkedList<>();

      linkedList.add("Apple");
      linkedList.add("Banana");
      linkedList.add("Orange");
      linkedList.add("Grape");
      linkedList.add("Mango");

      List<String> subList = linkedList.subList(1, 4);

      System.out.println("SubList elements:");
      for (String element : subList) {
         System.out.println(element);
      }
   }
}
```

Output:

```
SubList elements:
Banana
Orange
Grape
```

- **equals():**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class Test {
   public static void main(String[] args) {
      LinkedList<String> list1 = new LinkedList<>();
      LinkedList<String> list2 = new LinkedList<>();

      list1.add("Apple");
      list1.add("Banana");
      list1.add("Orange");

      list2.add("Apple");
      list2.add("Hamim");
      list2.add("Orange");

      boolean areEqual = list1.get(1).equals(list2.get(1));

      System.out.println("Are the lists equal? " + areEqual);
   }
}
```

Output:
Are the lists equal? false

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

public class LinkedListExample {
   public static void main(String[] args) {

      LinkedList<String> list1 = new LinkedList<>();
      LinkedList<String> list2 = new LinkedList<>();

      list1.add("Apple");
      list1.add("Banana");
      list1.add("Orange");

      list2.add("Apple");
      list2.add("Banana");
      list2.add("Orange");

      boolean areEqual = list1.equals(list2);

      System.out.println("Are the lists equal? " + areEqual);
   }
}
```

Output:
Are the lists equal? true

- **Collections.sort(LinkedList variable):**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;
import java.util.Collections;

public class Test1 {
   public static void main(String[] args) {
      LinkedList<String> linkedList = new LinkedList<>();
      linkedList.add("Orange");
      linkedList.add("Ant");
      linkedList.add("Banana");
      linkedList.add("Apple");
      linkedList.add("Cherry");

      System.out.println("Before sorting: " + linkedList);

      // Sorting the LinkedList using Collections.sort()
      Collections.sort(linkedList);

      System.out.println("After sorting: " + linkedList);
   }
}
```

Output:
Before sorting: [Orange, Ant, Banana, Apple, Cherry]
After sorting: [Ant, Apple, Banana, Cherry, Orange]

- **What is the output of the following Java program fragment:**

```
package FirstPackage;

import java.util.LinkedList;

public class Test {
   public static void main(String[] args)
   {
      LinkedList<String> countryName = new
LinkedList<>();

      countryName.add("Bangladesh");
      countryName.add("India");
      countryName.add("Pakisthan");
      countryName.add("Nepal");

      System.out.println(countryName);
   }
}
```

Output:
[Bangladesh, India, Pakisthan, Nepal]

- **What is the output of the following Java program fragment:**

```
package FirstPackage;

import java.util.LinkedList;

public class Test {
   public static void main(String[] args)
   {
      LinkedList<String> countryName = new
LinkedList<>();

      countryName.add("Bangladesh");
      countryName.add("India");
      countryName.add("Pakisthan");
      countryName.add("Nepal");

      for (String country : countryName){
         System.out.println(country);
      }
   }
}
```

Output:
Bangladesh
India
Pakisthan
Nepal

- **What is the output of the following Java program fragment:**

```
package FirstPackage;

import java.util.LinkedList;

public class Test {
   public static void main(String[] args)
   {
      LinkedList<String> countryName = new
LinkedList<>();

      countryName.add("Bangladesh");
      countryName.add("India");
      countryName.add("Pakisthan");
      countryName.add("Nepal");
      countryName.add(3, "London");

      for (String country : countryName)
      {
         System.out.println(country);
      }

      System.out.println("Size of the linkedList :
"+countryName.size());
   }
}
```

Output:
Bangladesh
India
Pakisthan
Nepal

- **What is the output of the following Java program fragment:**

```
package FirstPackage;

import java.util.LinkedList;

public class Test {
   public static void main(String[] args)
   {
      LinkedList<String> countryName = new
LinkedList<>();

      countryName.add("Bangladesh");
      countryName.add("India");
      countryName.add("Pakisthan");
      countryName.add("Nepal");
      countryName.add(3, "London");
      countryName.addFirst("Australia");
      countryName.addLast("Japan");

      for (String country : countryName)
      {
         System.out.println(country);
      }
```

```
        System.out.println("Size of the linkedList :
"+countryName.size());
    }
}
```

- **What is the output of the following Java program fragment:**

```
package FirstPackage;

public class Student {
    String name, className;
    int id;

    Student(int id, String name, String className)
{
        this.id = id;
        this.name = name;
        this.className = className;
    }
}
```

```
package FirstPackage;

import java.util.LinkedList;

public class StudentListDemo {
    public static void main(String[] args) {
```

```
        LinkedList<Student> list = new
LinkedList<Student>();

        // student create
        Student s1 = new Student(5369,"HaMeem","1st
year");
        Student s2 = new Student(5379,"Jim","1st
year");
        Student s3 = new Student(5367,"Rahim","1st
year");
        Student s4 = new Student(5364,"Mithu","1st
year");

        // adding student to the student list
        list.add(s1);
        list.add(s2);
        list.add(s3);
        list.add(s4);

        //information display
        for (Student s: list){
            System.out.println(s.id+" "+s.name+"
"+s.className);
        }
    }
}
```

Output:

5369 HaMeem 1st year
5379 Jim 1st year
5367 Rahim 1st year
5364 Mithu 1st year

- **Creating multiple objects using LinkedList:**

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

class Person {
    private String name;
    private int age;
    private String address;

    public Person(String name, int age, String address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
```

```
    public String getAddress() {
        return address;
    }
}

public class Test {
    public static void main(String[] args) {

        LinkedList<Person> personList = new LinkedList<>();

        personList.add(new Person("John", 25, "123 Main St"))
        personList.add(new Person("Jane", 30, "456 Elm St"));
        personList.add(new Person("Mike", 35, "789 Oak St"));

        for (int i = 0; i < personList.size(); i++) {
            Person person = personList.get(i);

            System.out.println("Name: " + person.getName());
            System.out.println("Age: " + person.getAge());
            System.out.println("Address: " +
person.getAddress());
            System.out.println("----------------");
        }
    }
}
```

Output:

Name: John
Age: 25
Address: 123 Main St

Name: Jane
Age: 30
Address: 456 Elm St

Name: Mike
Age: 35
Address: 789 Oak St

- **What is the output of the following Java program fragment:**

```
import java.util.LinkedList;

class Person {
    private String name;
    private int age;
    private String address;

    public Person(String name, int age, String address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public String getAddress() {
        return address;
    }
}
```

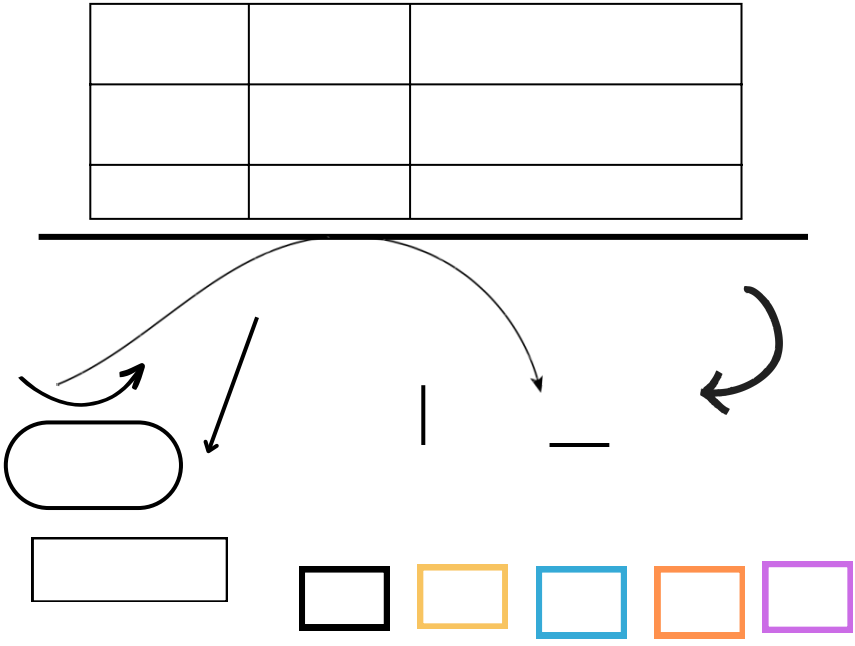
```
public class Test {
   public static void main(String[] args) {

      LinkedList<Person> personList = new LinkedList<>();

      personList.add(new Person("John", 25, "123 Main St"));
      personList.add(new Person("Jane", 30, "456 Elm St"));
      personList.add(new Person("Mike", 35, "789 Oak St"));

      for (Person person : personList) {
         System.out.println("Name: " + person.getName());
         System.out.println("Age: " + person.getAge());
         System.out.println("Address: " +
person.getAddress());
         System.out.println("----------------");
      }
   }
}
```

Output:

```
Name: John
Age: 25
Address: 123 Main St
----------------
Name: Jane
Age: 30
Address: 456 Elm St
----------------
Name: Mike
Age: 35
Address: 789 Oak St
----------------
```

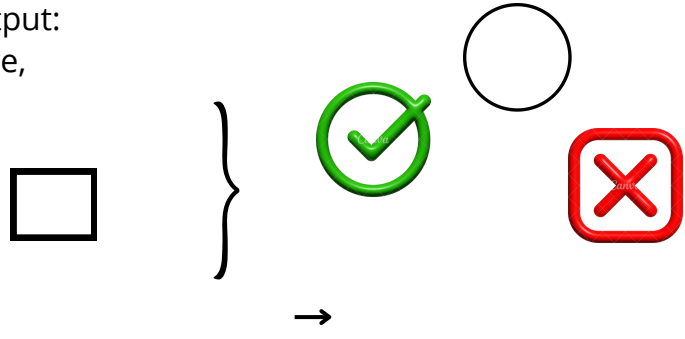
Operator Precedence and Associativity:

- **Make a program that will show String palindrome.**



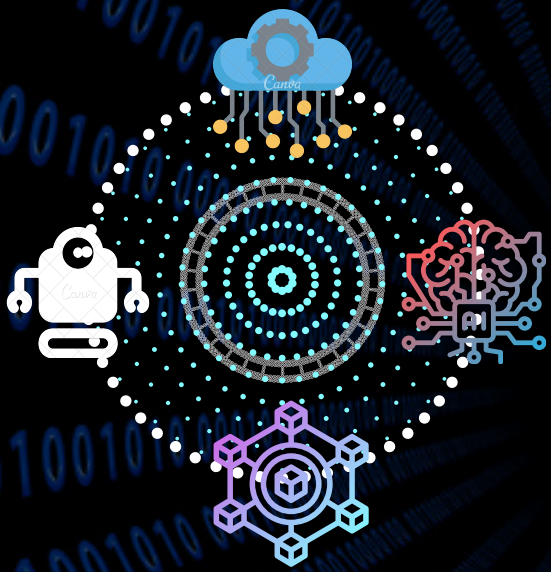
Output:
Here,

Jim



- **What is the output of the following Java program fragment:**

Operator



JAVA
(FOURTH PART)

T.I.M. HAMIM

ABC
PROKASHONI