peopleTECH

A Quest Global Company

# INTERNSHIP REPORT

## WEEK 1 - Data Structures and Algorithms

SUBMITTED TO

PEOPLE TECH GROUP INC

GOVIND SHARMA

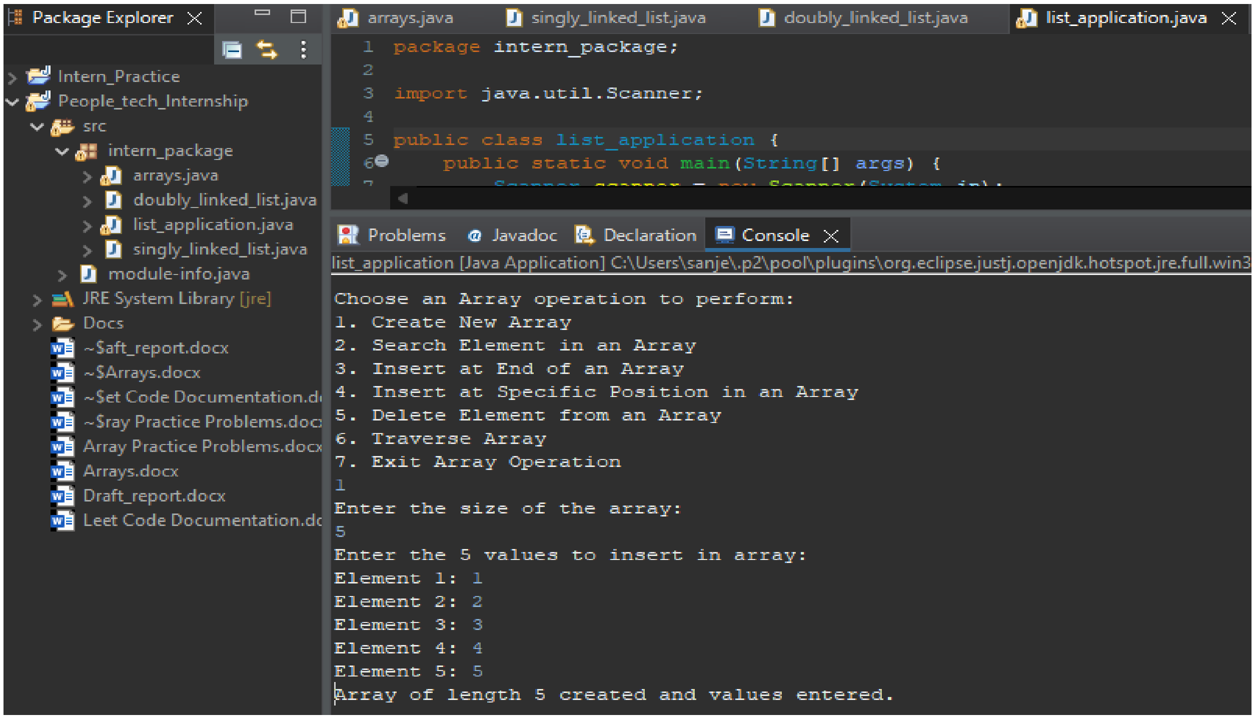
RECRUITMENT LEAD

SUBMITTED BY

SANJEEV REDDY SIRIPINANE

## 1. Introduction to arrays:

### Array Creation:

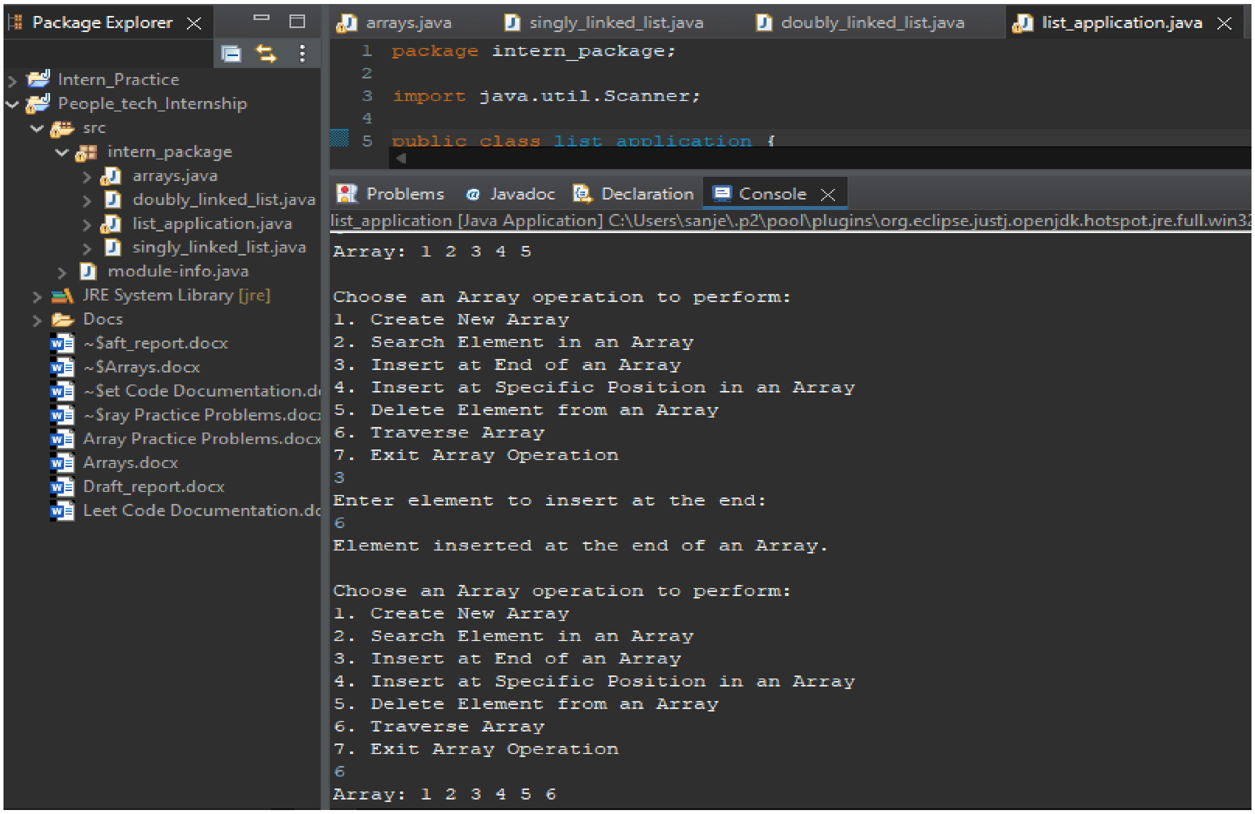


```java
package intern_package;

import java.util.Scanner;

public class list_application {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
```

list_application [Java Application] C:\Users\sanje\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win3

```
Choose an Array operation to perform:
1. Create New Array
2. Search Element in an Array
3. Insert at End of an Array
4. Insert at Specific Position in an Array
5. Delete Element from an Array
6. Traverse Array
7. Exit Array Operation
1
Enter the size of the array:
5
Enter the 5 values to insert in array:
Element 1: 1
Element 2: 2
Element 3: 3
Element 4: 4
Element 5: 5
Array of length 5 created and values entered.
```

### Array Insertion at the end:

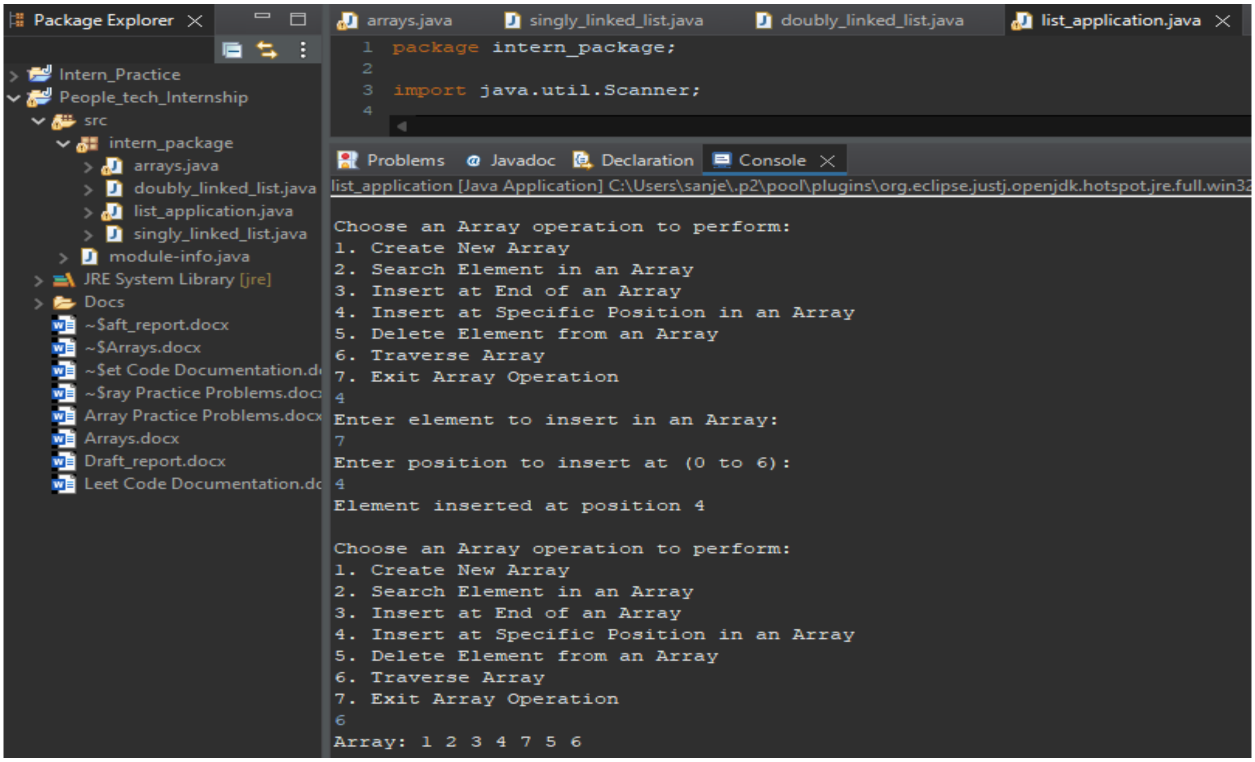


```java
package intern_package;

import java.util.Scanner;

public class list_application {
```

list_application [Java Application] C:\Users\sanje\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32

```
Array: 1 2 3 4 5

Choose an Array operation to perform:
1. Create New Array
2. Search Element in an Array
3. Insert at End of an Array
4. Insert at Specific Position in an Array
5. Delete Element from an Array
6. Traverse Array
7. Exit Array Operation
3
Enter element to insert at the end:
6
Element inserted at the end of an Array.

Choose an Array operation to perform:
1. Create New Array
2. Search Element in an Array
3. Insert at End of an Array
4. Insert at Specific Position in an Array
5. Delete Element from an Array
6. Traverse Array
7. Exit Array Operation
6
Array: 1 2 3 4 5 6
```

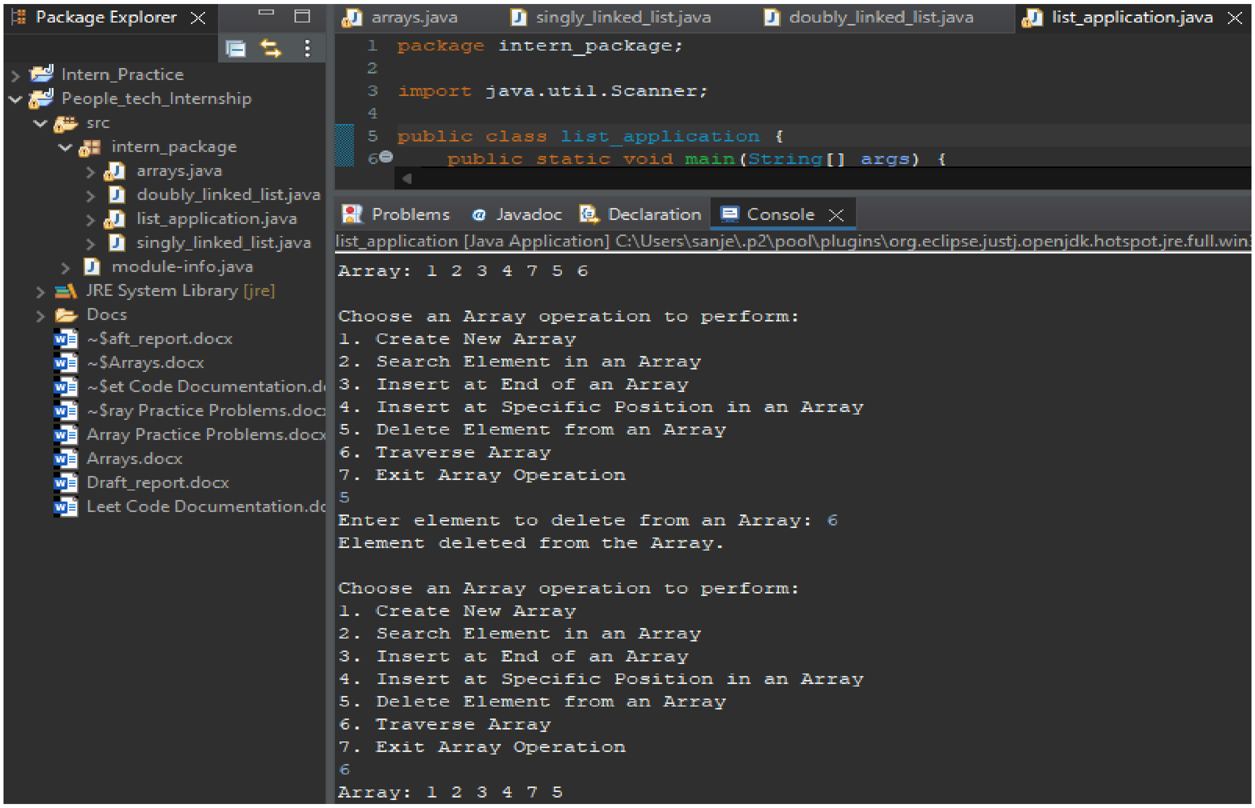
## Array Insertion at Specific Position:



```
package intern_package;

import java.util.Scanner;

```

```
Choose an Array operation to perform:
1. Create New Array
2. Search Element in an Array
3. Insert at End of an Array
4. Insert at Specific Position in an Array
5. Delete Element from an Array
6. Traverse Array
7. Exit Array Operation
4
Enter element to insert in an Array:
7
Enter position to insert at (0 to 6):
4
Element inserted at position 4

Choose an Array operation to perform:
1. Create New Array
2. Search Element in an Array
3. Insert at End of an Array
4. Insert at Specific Position in an Array
5. Delete Element from an Array
6. Traverse Array
7. Exit Array Operation
6
Array: 1 2 3 4 7 5 6
```

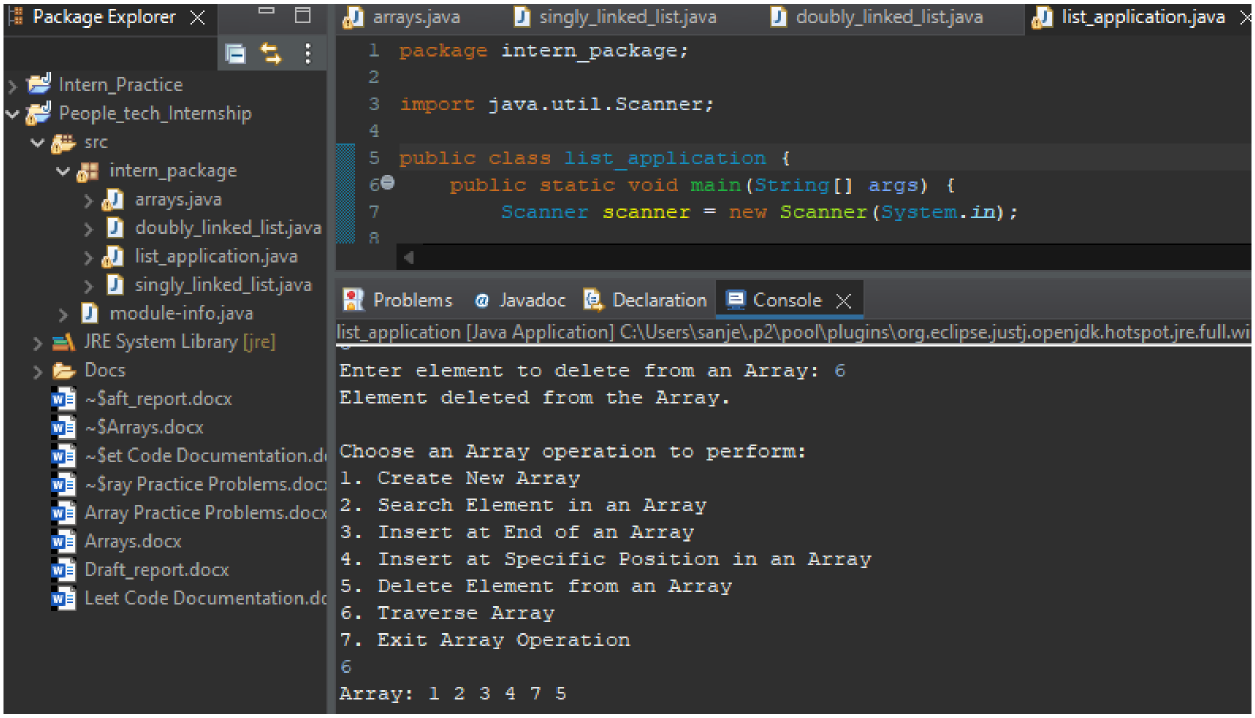
## Array Deletion:



```
package intern_package;

import java.util.Scanner;

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

```
Array: 1 2 3 4 7 5 6

Choose an Array operation to perform:
1. Create New Array
2. Search Element in an Array
3. Insert at End of an Array
4. Insert at Specific Position in an Array
5. Delete Element from an Array
6. Traverse Array
7. Exit Array Operation
5
Enter element to delete from an Array: 6
Element deleted from the Array.

Choose an Array operation to perform:
1. Create New Array
2. Search Element in an Array
3. Insert at End of an Array
4. Insert at Specific Position in an Array
5. Delete Element from an Array
6. Traverse Array
7. Exit Array Operation
6
Array: 1 2 3 4 7 5
```

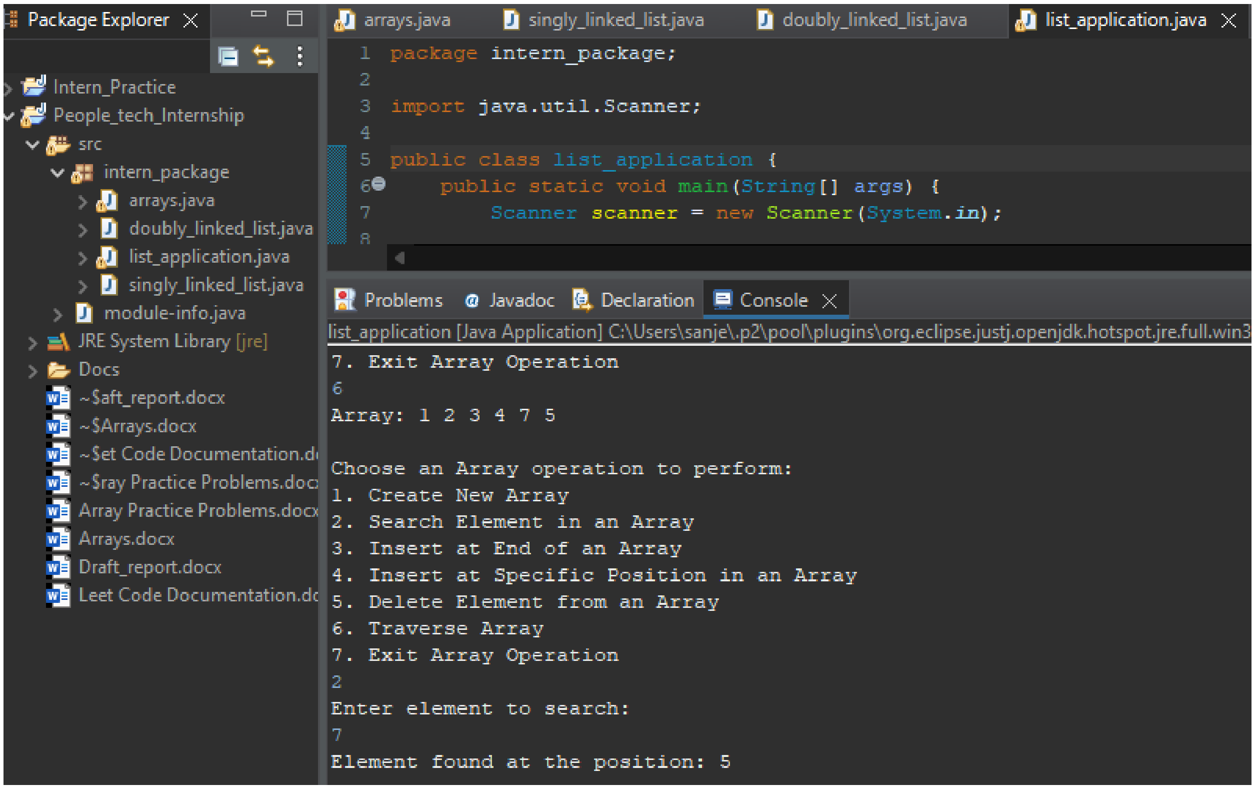
## Array Traverse:



```java
package intern_package;

import java.util.Scanner;

public class list_application {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
```

```
Enter element to delete from an Array: 6
Element deleted from the Array.

Choose an Array operation to perform:
1. Create New Array
2. Search Element in an Array
3. Insert at End of an Array
4. Insert at Specific Position in an Array
5. Delete Element from an Array
6. Traverse Array
7. Exit Array Operation
6
Array: 1 2 3 4 7 5
```

## Search for an Element in an Array:



```java
package intern_package;

import java.util.Scanner;

public class list_application {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
```

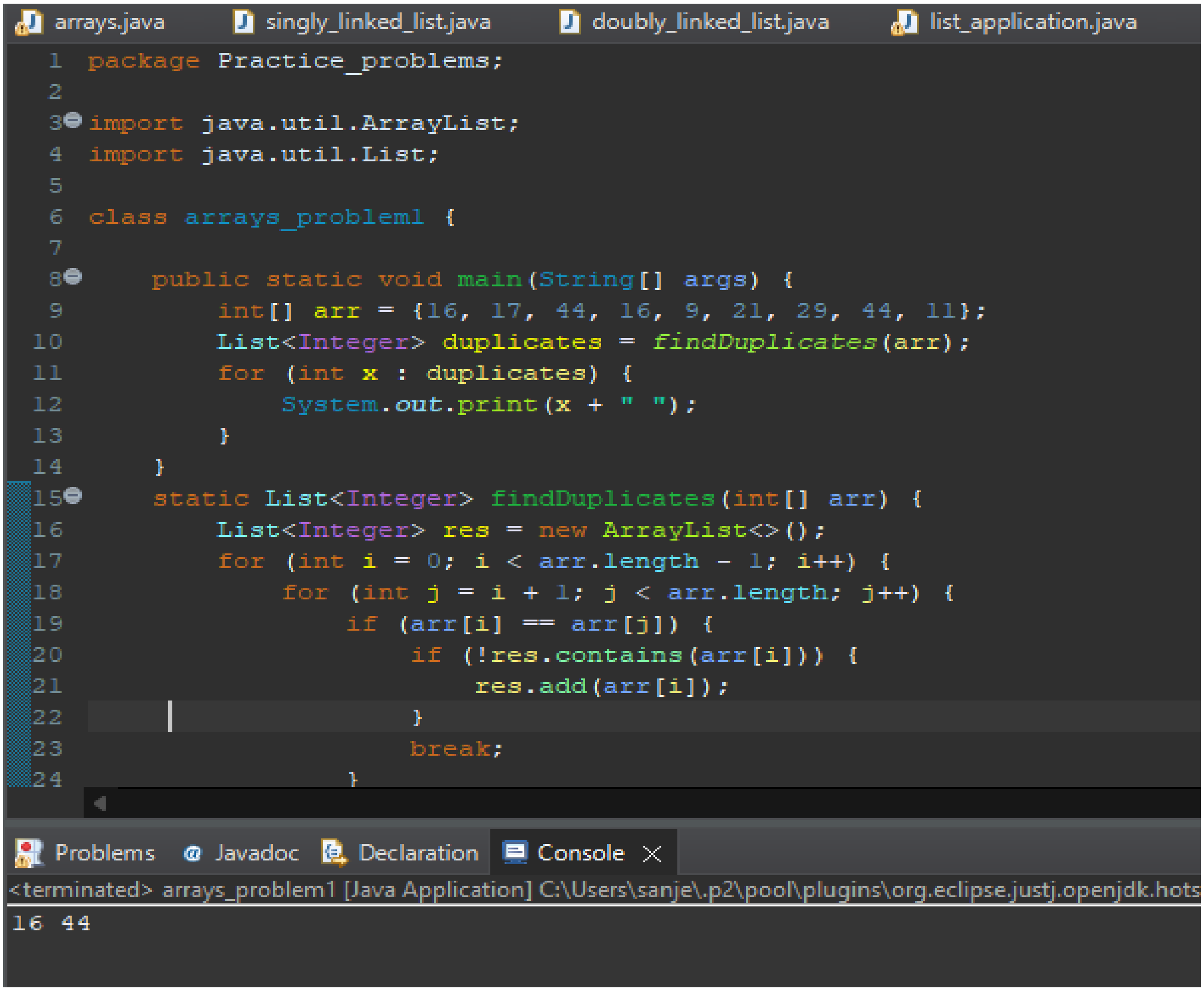
```
7. Exit Array Operation
6
Array: 1 2 3 4 7 5

Choose an Array operation to perform:
1. Create New Array
2. Search Element in an Array
3. Insert at End of an Array
4. Insert at Specific Position in an Array
5. Delete Element from an Array
6. Traverse Array
7. Exit Array Operation
2
Enter element to search:
7
Element found at the position: 5
```

**Code Flow- Chart for Array Operations:**

```
                                        Start
                                          |
                                          v
                                     [Main Menu]
                                          |
                                          v
                               [User Selects Operation]
                   |----------------------------|--------------------|-----------------------|
                         |                   |              |                 |
         [Create Array]        [Search Element]     [Insert at End]     [Insert at Position]
                         |                   |              |                 |
  [Input Array Size]       [Ask Element]      [Ask for Element]   [Ask for Element & Position]
                         |                   |              |                 |
             [Initialize Array]     [Search Array]      [Check Full]        [Check Full]
                         |                   |              |                 |
            [Input Values]         [Display Result]   [Resize if Full]   [Resize if Full]
                         |                   |              |                 |
     [Operation Complete]   [Return to Main Menu]   [Insert at End]     [Shift Elements]
                         |                   |              |                 |
                         v                   v              v                 v
[Return to Main Menu]   [Return to Main Menu] [Return to Main Menu]  [Return to Main Menu]
                                          |
                                          v
                           [User Selects Another Operation]
                                          |
                                          v
                                        [Exit]
```

## Practice Problems Related to Arrays:

### Duplicates in an Array:



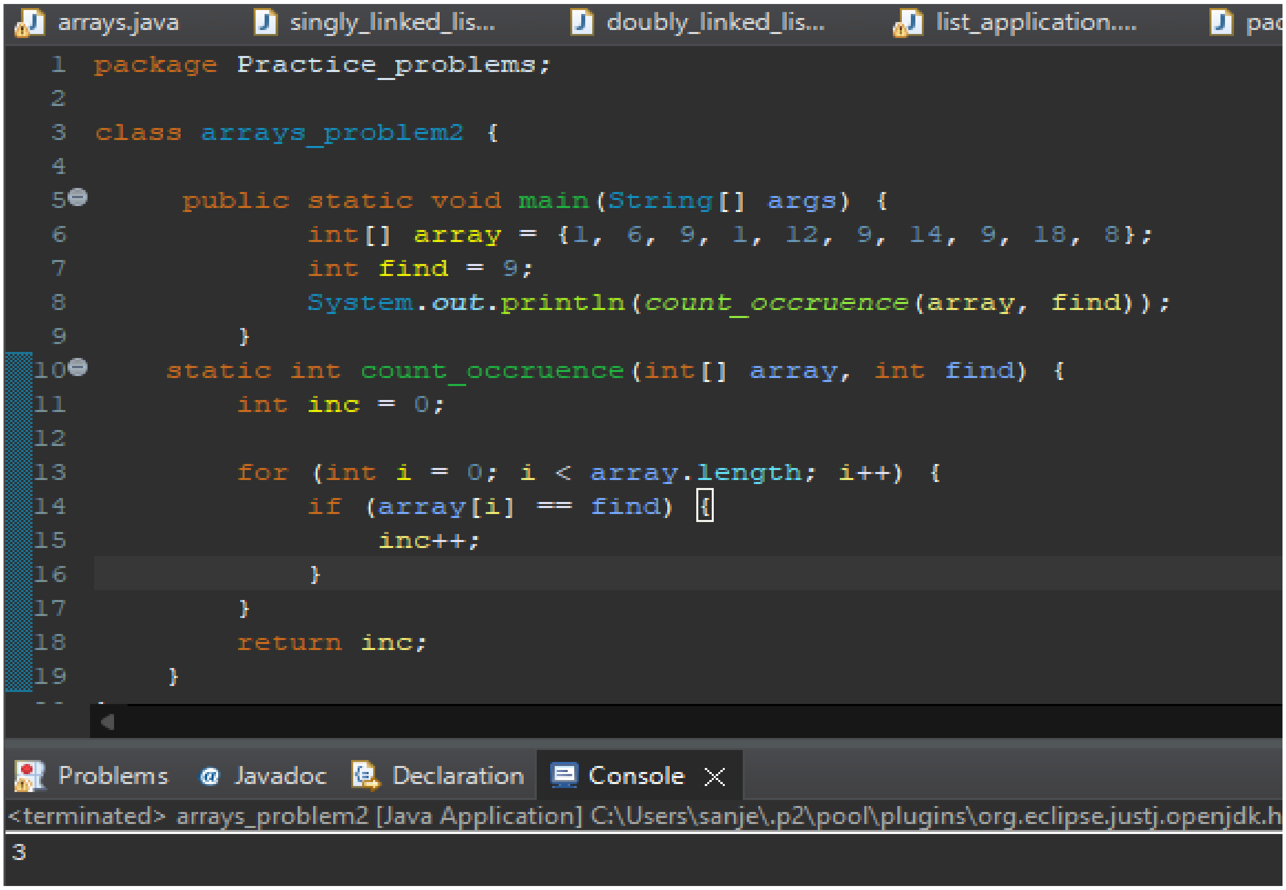
```java
package Practice_problems;

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

class arrays_problem1 {

    public static void main(String[] args) {
        int[] arr = {16, 17, 44, 16, 9, 21, 29, 44, 11};
        List<Integer> duplicates = findDuplicates(arr);
        for (int x : duplicates) {
            System.out.print(x + " ");
        }
    }
    static List<Integer> findDuplicates(int[] arr) {
        List<Integer> res = new ArrayList<>();
        for (int i = 0; i < arr.length - 1; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] == arr[j]) {
                    if (!res.contains(arr[i])) {
                        res.add(arr[i]);
                    }
                    break;
                }
```

```
<terminated> arrays_problem1 [Java Application] C:\Users\sanje\.p2\pool\plugins\org.eclipse.justj.openjdk.hots
16 44
```

**Approach:** To discover duplicates in an array, we use two nested loops: the first loop examines each item, and the second loop checks the items following it for matches. We use if (!res.contains(arr[i])) to avoid the same duplicate from being added multiple times. While this strategy is straightforward, it is not the most efficient because it requires several comparisons, particularly with bigger arrays. The res.contains() check adds complexity, resulting in a "quadratic" time complexity (O($n^2$)). As the array size rises, the solution slows down dramatically.

**Challenge:** The biggest challenge I faced was the inefficiency of using nested loops and the contains() function to check for duplicates, which leads to slow performance as the array size grows (O($n^2$) time complexity). Additionally, ensuring that each duplicate was added only once to the result list required extra care. Handling edge cases, like arrays with no duplicates or arrays where all elements are the same, also added complexity, making it difficult to balance simplicity with performance.

### Occurrences of Integer in an Array:



```
package Practice_problems;

class arrays_problem2 {

     public static void main(String[] args) {
            int[] array = {1, 6, 9, 1, 12, 9, 14, 9, 18, 8};
            int find = 9;
            System.out.println(count_occruence(array, find));
        }
    static int count_occruence(int[] array, int find) {
        int inc = 0;

        for (int i = 0; i < array.length; i++) {
            if (array[i] == find) {
                inc++;
            }
        }
        return inc;
    }
```

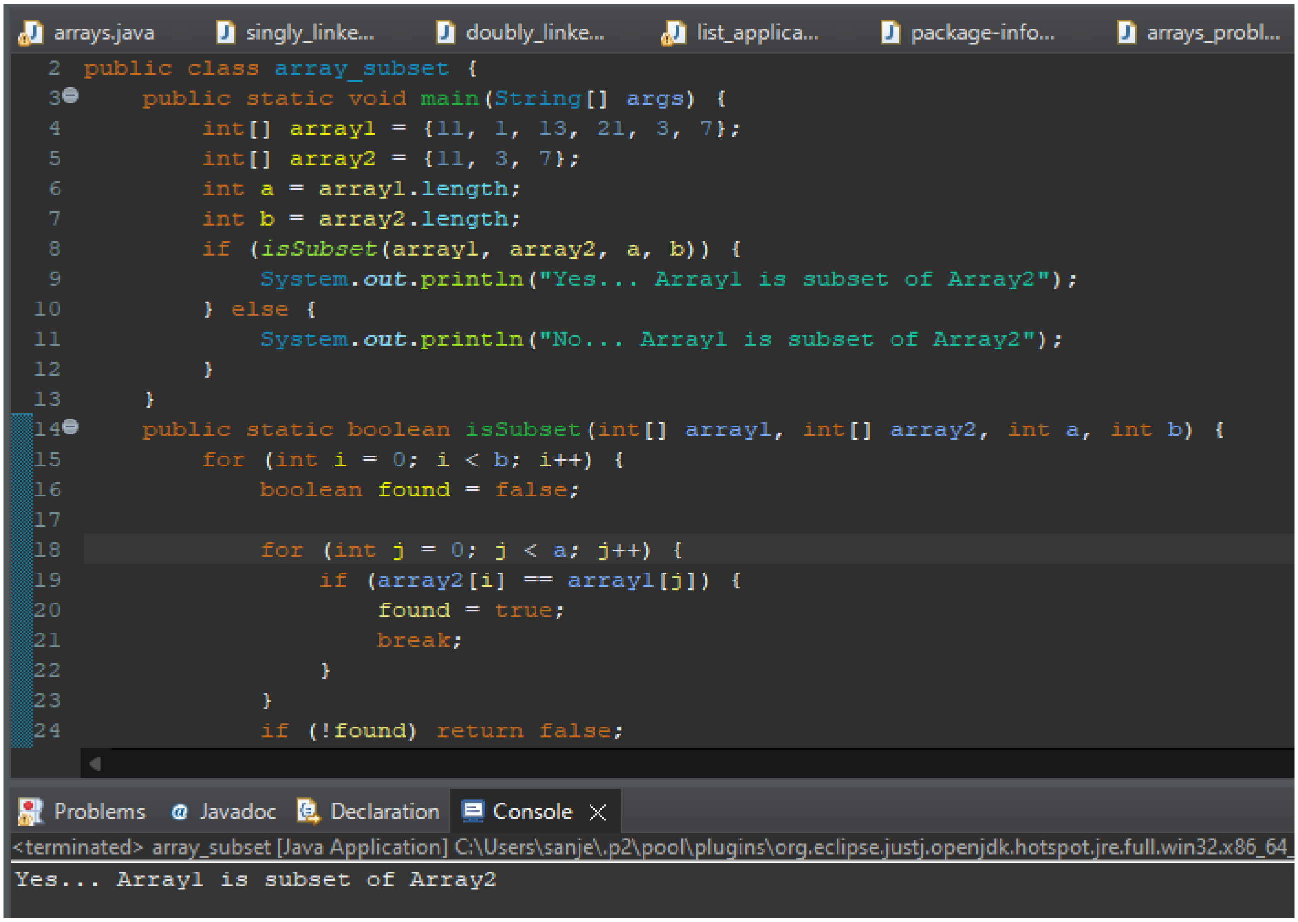
```
<terminated> arrays_problem2 [Java Application] C:\Users\sanje\.p2\pool\plugins\org.eclipse.justj.openjdk.h
3
```

### Approach:

The goal of this task is to count how many times a specific number appears in an array. The approach is simple: we loop through the array and check each element to see if it matches the target number. Every time we find a match, we increment a counter. Once the loop finishes, the counter holds the total count of how many times the target number appeared in the array, which we then return as the result. While this method is easy to understand and implement, its efficiency can decrease for larger arrays, as it requires checking every element individually.

**Challenge**: Making sure the code accurately counts the instances of the desired number in the array was the biggest issue I encountered when writing this code. I had to verify that the count was correct by closely examining each component. For small arrays, the technique works well; but, as the array size increases, the procedure slows down since each element must be examined separately. In addition, I had to deal with edge circumstances, such as when the target number wasn't in the array and ensure that the code was clear and simple without becoming too complicated.

### Subset of an Array:



```java
public class array_subset {
    public static void main(String[] args) {
        int[] array1 = {11, 1, 13, 21, 3, 7};
        int[] array2 = {11, 3, 7};
        int a = array1.length;
        int b = array2.length;
        if (isSubset(array1, array2, a, b)) {
            System.out.println("Yes... Array1 is subset of Array2");
        } else {
            System.out.println("No... Array1 is subset of Array2");
        }
    }
    public static boolean isSubset(int[] array1, int[] array2, int a, int b) {
        for (int i = 0; i < b; i++) {
            boolean found = false;

            for (int j = 0; j < a; j++) {
                if (array2[i] == array1[j]) {
                    found = true;
                    break;
                }
            }
            if (!found) return false;
```

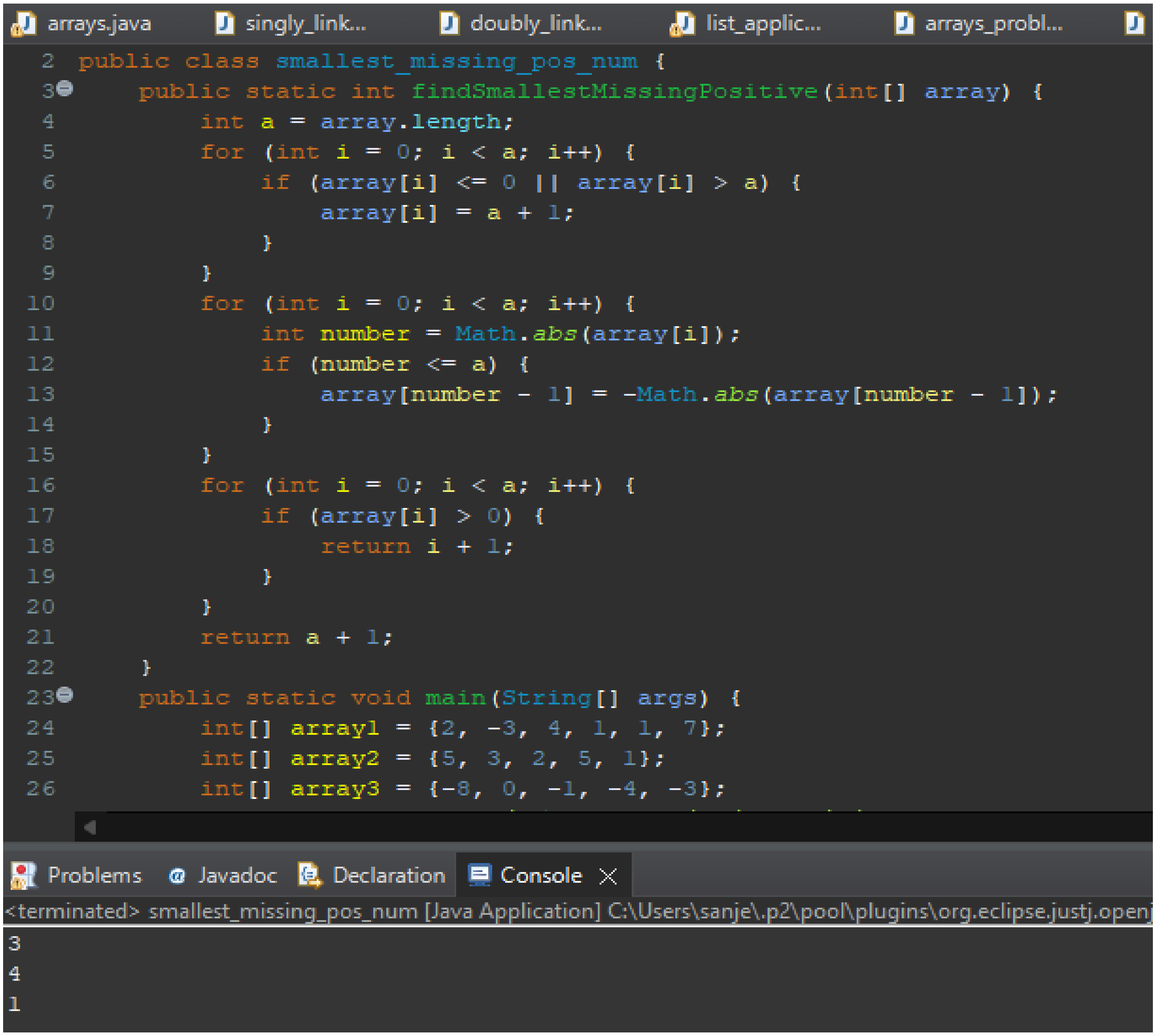
```
<terminated> array_subset [Java Application] C:\Users\sanje\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_
Yes... Array1 is subset of Array2
```

### Approach :

The code checks if all elements of array2 are present in array1, essentially determining if array2 is a subset of array1. It loops through each element of array2 and checks for a match in array1. If any element in array2 isn't found in array1, it returns false. If all elements of array2 are found, it returns true. The final result is printed as "Yes" if array2 is a subset of array1, and "No" if it's not. The approach is straightforward but may be inefficient for large arrays due to the nested iteration.

**Challenge**: The main challenge in this problem was ensuring the logic correctly checks if all elements of array2 are in array1 while managing the nested loop structure. As the arrays grow, the approach becomes slower due to inefficiency in the nested iterations. Handling edge cases, such as an empty array2 or arrays with duplicates, also proved difficult. Additionally, careful management of the boolean flag for each element in array2 was necessary to ensure the code terminates early when a match isn't found.

**Smallest Positive Missing Number:**



```java
public class smallest_missing_pos_num {
    public static int findSmallestMissingPositive(int[] array) {
        int a = array.length;
        for (int i = 0; i < a; i++) {
            if (array[i] <= 0 || array[i] > a) {
                array[i] = a + 1;
            }
        }
        for (int i = 0; i < a; i++) {
            int number = Math.abs(array[i]);
            if (number <= a) {
                array[number - 1] = -Math.abs(array[number - 1]);
            }
        }
        for (int i = 0; i < a; i++) {
            if (array[i] > 0) {
                return i + 1;
            }
        }
        return a + 1;
    }
    public static void main(String[] args) {
        int[] array1 = {2, -3, 4, 1, 1, 7};
        int[] array2 = {5, 3, 2, 5, 1};
        int[] array3 = {-8, 0, -1, -4, -3};
```

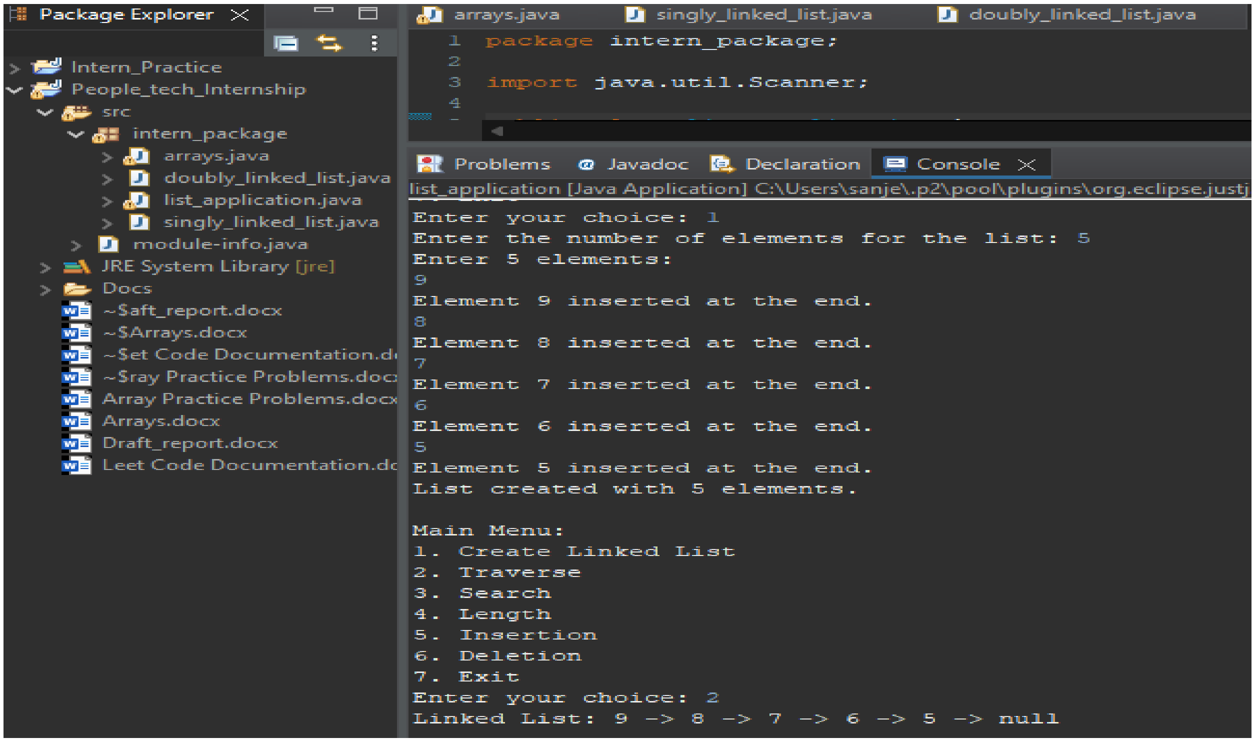
```
<terminated> smallest_missing_pos_num [Java Application] C:\Users\sanje\.p2\pool\plugins\org.eclipse.justj.openj
3
4
1
```

**Approach:** To find the smallest missing positive number, we first replace any out-of-range values (negative or greater than the array length) with a number like array.length + 1. Then, for each valid number, we mark its corresponding index as "seen" by making the value negative. For example, if we encounter the number 3, we make the value at index 2 negative. After scanning the array, the first index with a positive value indicates the missing number. If all values are marked, the smallest missing number is array.length + 1. This approach efficiently tracks the missing number without using extra space, modifying the array itself.

**Challenge:** Since we use the array itself to record "seen" numbers, one issue in putting this technique into practice is carefully handling in-place adjustments without losing track of the original values. It's critical to manage negative numbers appropriately and refrain from erasing important data. Furthermore, after marking, it might be challenging to accurately interpret the "unseen" numbers, particularly when all of the values may fall outside of the desired range.

## 2. Introduction to Linked List

### Singly Linked List Creation:

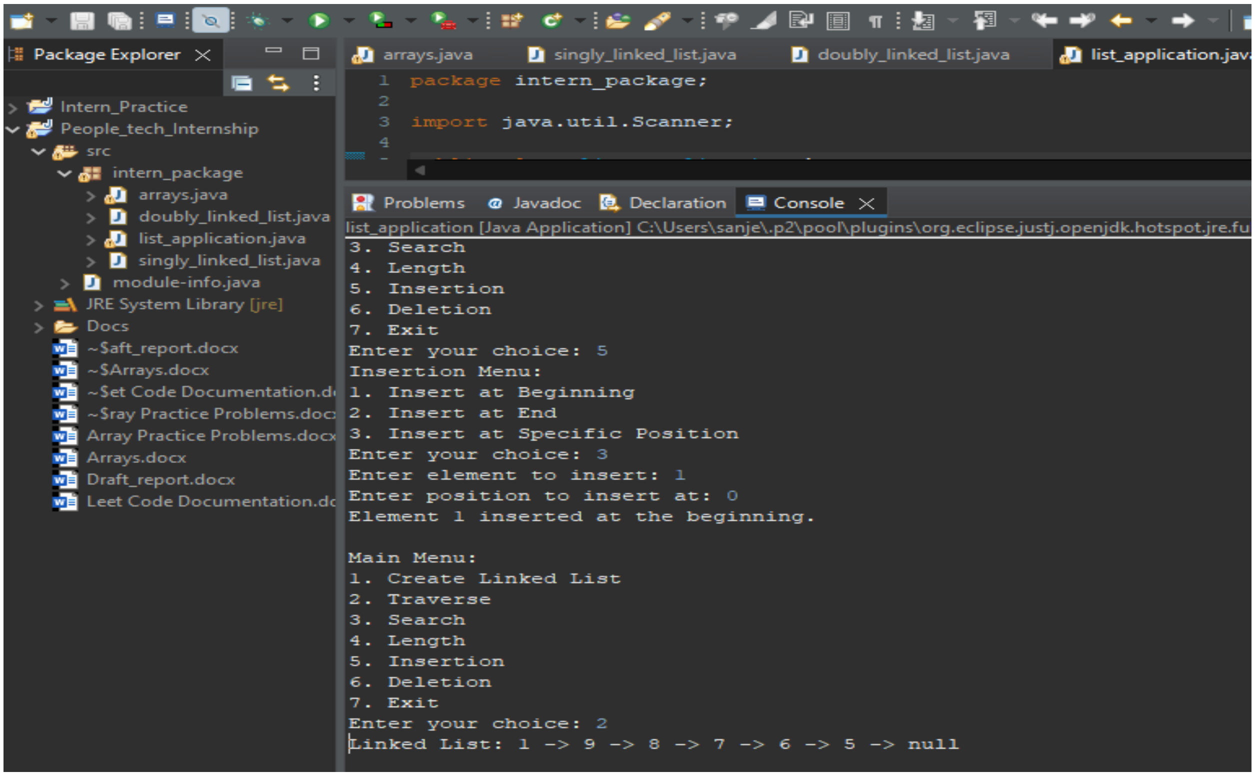


```
package intern_package;

import java.util.Scanner;
```

list_application [Java Application] C:\Users\sanje\.p2\pool\plugins\org.eclipse.justj

```
Enter your choice: 1
Enter the number of elements for the list: 5
Enter 5 elements:
9
Element 9 inserted at the end.
8
Element 8 inserted at the end.
7
Element 7 inserted at the end.
6
Element 6 inserted at the end.
5
Element 5 inserted at the end.
List created with 5 elements.

Main Menu:
1. Create Linked List
2. Traverse
3. Search
4. Length
5. Insertion
6. Deletion
7. Exit
Enter your choice: 2
Linked List: 9 -> 8 -> 7 -> 6 -> 5 -> null
```

Singly Linked List Insertion at specific position:

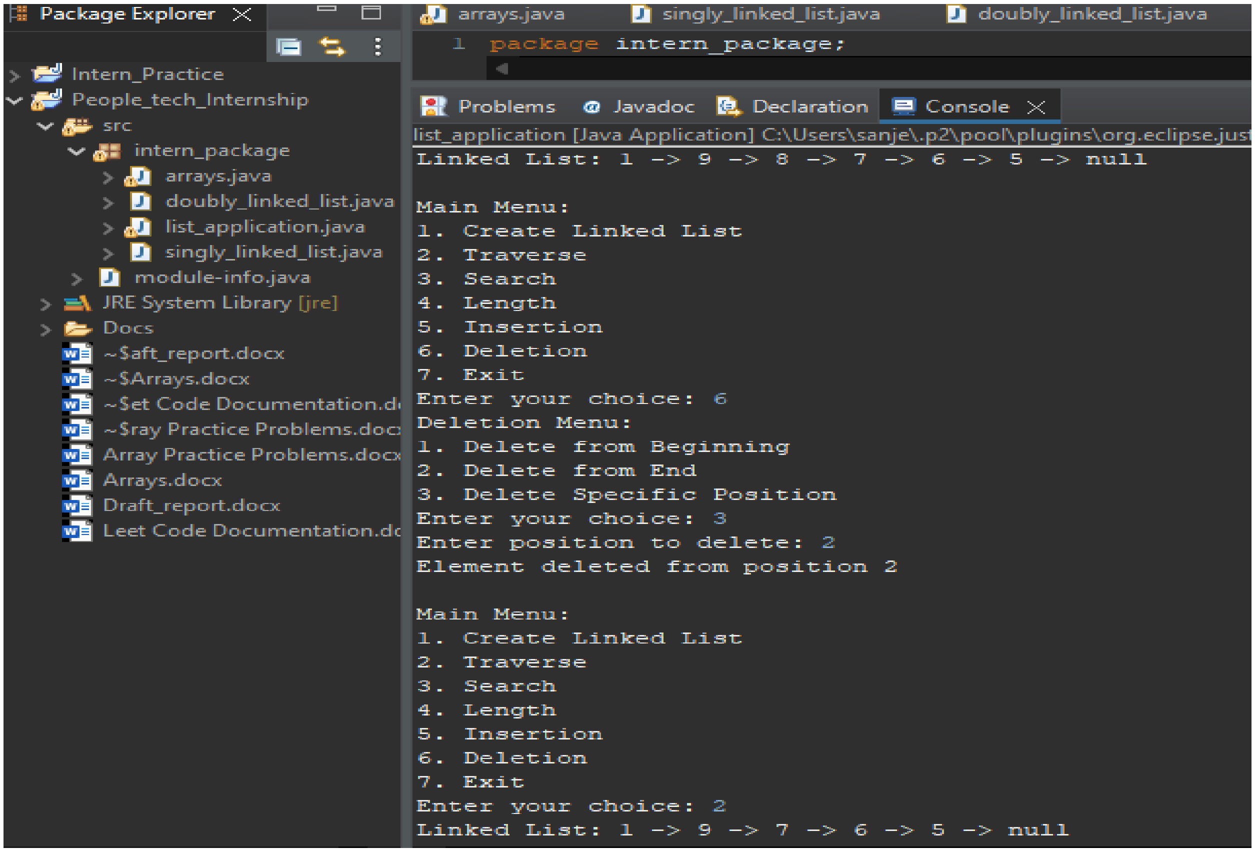


```
package intern_package;

import java.util.Scanner;
```

list_application [Java Application] C:\Users\sanje\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.fu

```
3. Search
4. Length
5. Insertion
6. Deletion
7. Exit
Enter your choice: 5
Insertion Menu:
1. Insert at Beginning
2. Insert at End
3. Insert at Specific Position
Enter your choice: 3
Enter element to insert: 1
Enter position to insert at: 0
Element 1 inserted at the beginning.

Main Menu:
1. Create Linked List
2. Traverse
3. Search
4. Length
5. Insertion
6. Deletion
7. Exit
Enter your choice: 2
Linked List: 1 -> 9 -> 8 -> 7 -> 6 -> 5 -> null
```

Singly Linked List Deletion at specific position:

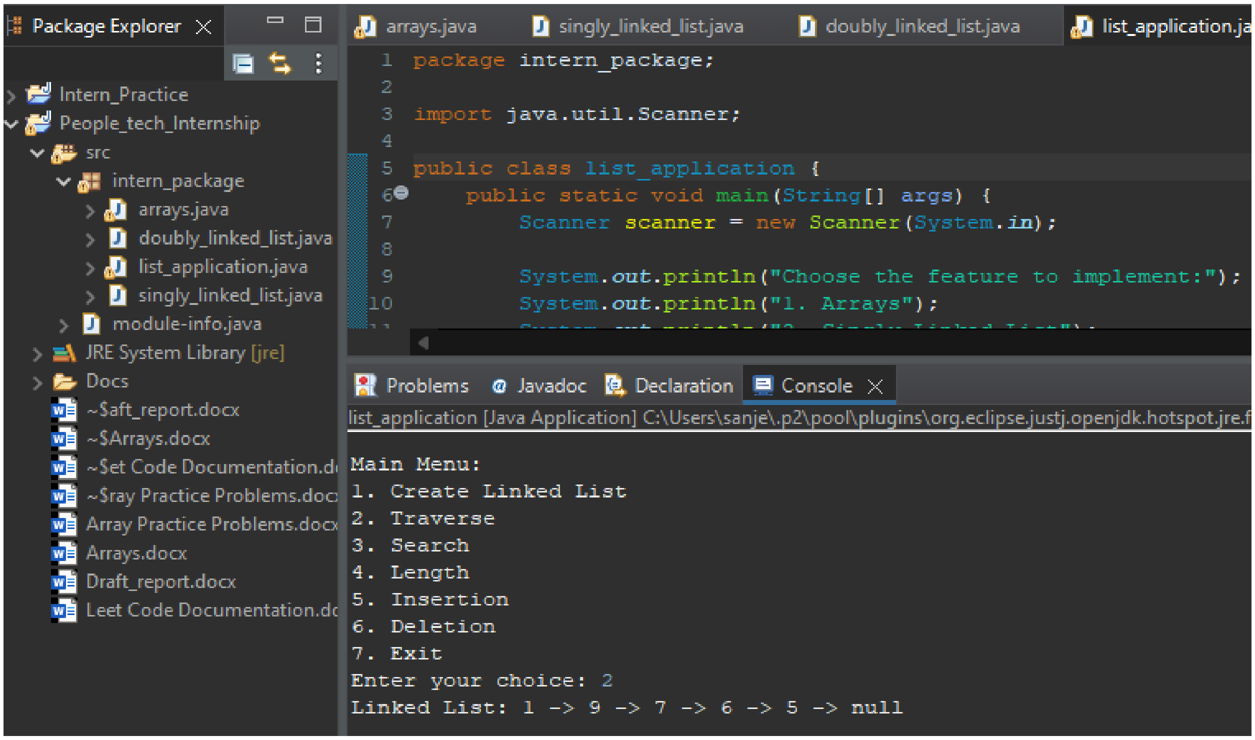


```java
package intern_package;
```

list_application [Java Application] C:\Users\sanje\.p2\pool\plugins\org.eclipse.just

```
Linked List: 1 -> 9 -> 8 -> 7 -> 6 -> 5 -> null

Main Menu:
1. Create Linked List
2. Traverse
3. Search
4. Length
5. Insertion
6. Deletion
7. Exit
Enter your choice: 6
Deletion Menu:
1. Delete from Beginning
2. Delete from End
3. Delete Specific Position
Enter your choice: 3
Enter position to delete: 2
Element deleted from position 2

Main Menu:
1. Create Linked List
2. Traverse
3. Search
4. Length
5. Insertion
6. Deletion
7. Exit
Enter your choice: 2
Linked List: 1 -> 9 -> 7 -> 6 -> 5 -> null
```

Singly Linked List Traversal:

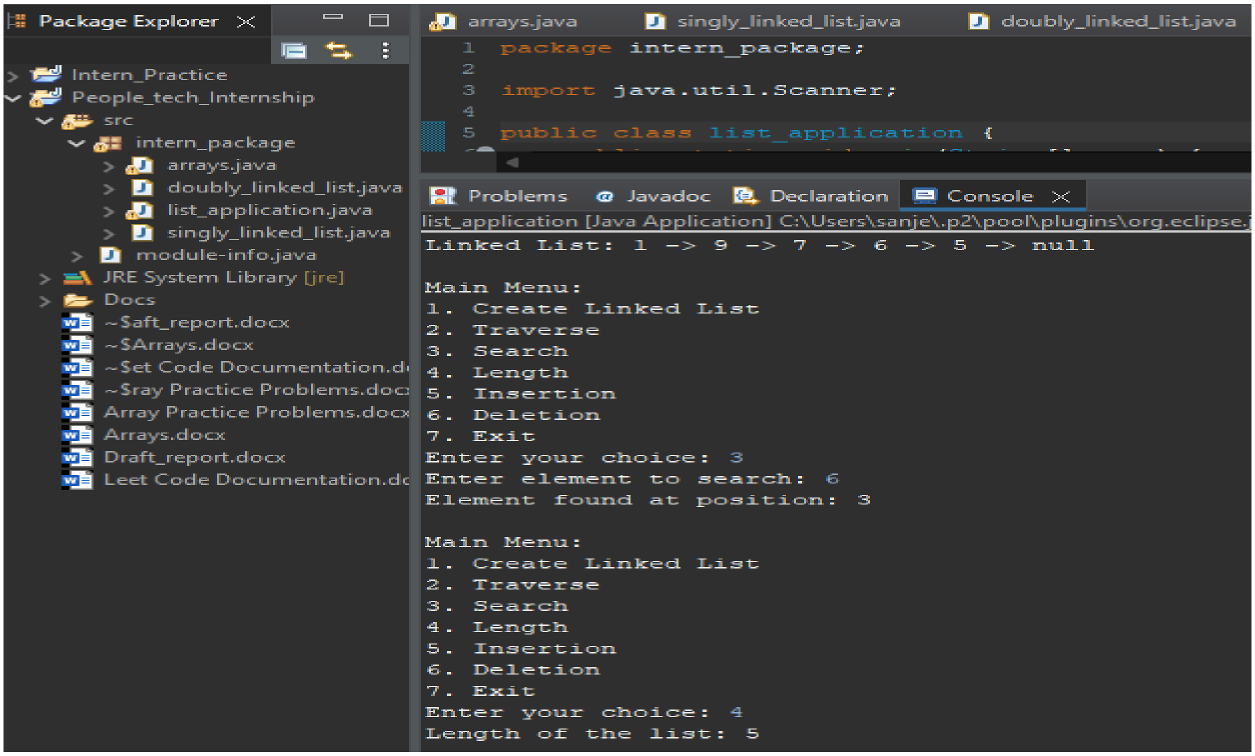


```java
package intern_package;

import java.util.Scanner;

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

        System.out.println("Choose the feature to implement:");
        System.out.println("1. Arrays");
```

list_application [Java Application] C:\Users\sanje\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.f

```
Main Menu:
1. Create Linked List
2. Traverse
3. Search
4. Length
5. Insertion
6. Deletion
7. Exit
Enter your choice: 2
Linked List: 1 -> 9 -> 7 -> 6 -> 5 -> null
```

Singly Linked List Search and Length of List:



```java
package intern_package;

import java.util.Scanner;

public class list_application {
```

```
Linked List: 1 -> 9 -> 7 -> 6 -> 5 -> null

Main Menu:
1. Create Linked List
2. Traverse
3. Search
4. Length
5. Insertion
6. Deletion
7. Exit
Enter your choice: 3
Enter element to search: 6
Element found at position: 3

Main Menu:
1. Create Linked List
2. Traverse
3. Search
4. Length
5. Insertion
6. Deletion
7. Exit
Enter your choice: 4
Length of the list: 5
```

## Linked List Code Flow- chart:

```
                                        Start
                                          v
                                  [Show Main Menu]
                                          v
                              [User Selects Operation]

              |------------------------|----------------------------|-----------------------|

         [Create Linked List]      [Traverse List]     [Search Element]      [Length of List]

              |                          |                          |                       |

   [Enter Number of Elements] [Display Linked List] [Ask for Element to Search] [Display Length]

              |                          |                          |                       |

       [Create List]              [Show Elements]      [Search List for Element] [Display Length]

              |                          |                          |                       |

   [List Created]             [Return to Main Menu] [Display Search Result]   [Return to Main Menu]

              |                          |                          |                       |

[Return to Main Menu]    [User Selects Another Operation]          [User Selects Another Operation]

                                          v

                              [User Selects Insert Operation]

                |-----------------------------------|------------------------------------|
         [Insert at Beginning]               [Insert at End]            [Insert at Specific Position]
                        |                               |                          |
             [Ask for Element]                [Ask for Element]        [Ask for Element & Position]
                        |                               |                          |
            [Insert at Beginning]               [Insert at End]              [Insert at Position]
                        |                               |                          |
         [Operation Complete]            [Operation Complete]          [Operation Complete]
                        |                               |                          |
        [Return to Main Menu]           [Return to Main Menu]         [Return to Main Menu]

                                          v

                            [User Selects Delete Operation]

              |---------------------------------------|--------------------------------------|

           [Delete from Beginning]            [Delete from End]     [Delete from Specific Position]

                            |                               |                   |

            [Operation Complete]             [Operation Complete]     [Operation Complete]

                            |                               |                   |

           [Return to Main Menu]           [Return to Main Menu]    [Return to Main Menu]

                                          v

                            [User Selects Another Operation]

                                          v

                                        [Exit]
```

### Compare and contrast arrays and linked lists in terms of performance and use cases.

**Arrays**:

- Arrays provide quicker access to an element at a given index by storing elements in contiguous memory regions, which produce readily calculable addresses for the elements stored.

**Advantages**:

- **Random Access:** We can access ith item in O(1) time. Sequential access makes it an O(n) operation in the case of linked lists.
- **Cache friendliness:** Arrays are cache friendly because their items (or item references) are kept in contiguous places.
- **Simple to use:** Arrays are a fundamental component of computer languages and are comparatively simple to utilize.
- **Less Overhead:** We don't need to hold extra references or pointers with each item, unlike a linked list.

**Dis – Advantages**:

- **Fixed Size:** A static array's size cannot be altered after it has been declared.
- **Inefficient Insertions and Deletions:** For big arrays, it might be expensive to relocate other items in order to add or remove elements from the middle of the array.
- **Wasted Space:** The array may have wasted memory if some of its items are not used.

**Use-case of Arrays:**

**1. Needs for Random Access:** When you require quick, continuous access to components based on their index.
**2. Fixed Size Requirements:** If your data structure's size is known in advance and won't alter.
For applications where cache speed is crucial, cache-sensitive operations are used.
**4. Mathematical Operations:** Especially in linear algebra, arrays are ideally suited for mathematical operations.
**5. Sorting methods:** Arrays may be used with a variety of effective sorting methods.

**Applications of Arrays in the Real World**

- **Image Processing**: 2D arrays of pixels are frequently used to represent images.
- **Spreadsheets**: 2D arrays are commonly used to build the grid layout of spreadsheets.
- **Sensor Data Collection**: Time-series data from sensors is stored in arrays.

**Database Indexing**: Arrays are used by certain database systems to facilitate effective indexing and searching.

**Linked List**:

- Linked List 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.

**Advantages**:

- **Effective insertion and deletion**: To add (or remove) an item in the middle, we just need to modify a small number of pointers (or references). Any point in a linked list can be added or removed in O(1) time. In contrast, insertions and deletions in the center of an array data structure require O(n) time.
- **Queue and Deque implementation**: A basic array approach is completely inefficient. For efficient implementation, we need to utilize a circular array, which is complicated. However, using a linked list makes things simple and uncomplicated. For this reason, most language libraries implement these data structures internally using linked lists.
- **Space Efficient in Certain Situations:** When we are unable to predict the number of elements ahead of time, linked lists may prove to be more space efficient than arrays. When arrays are used, 100% of the memory is allotted to each item. even with dynamically sized arrays, such as an array list in Java, a list in Python, or a vector in C++. When insertions exceed the existing capacity, the internal workings entail de-allocating the whole memory and allocating a larger piece.
- **Circular List with Deletion/Addition**: Because circular linked lists allow for fast deletion/insertion in a circular fashion, they are helpful for implementing CPU round robin scheduling and other similar needs in the real world.

**Dis- Advantages**:

- **Random Access**: Since you must go through the list from the beginning, accessing elements by index is slower than using arrays.
- **Extra Memory Usage**: To store the reference to the node after it, each node needs extra memory.
- **Cache Performance**: In some situations, linked lists' low cache locality may affect performance.

**Use-case Linked Lists:**

- **Dynamic Size Requirements**: When a data structure that may expand or contract without reallocation is required.
- **Frequently Inserted/Deleted**: If your application has to include or remove items frequently, particularly at the start or middle of the list.
- **Use of Other Data Structures**: When constructing data structures such as queues, stacks, or specific kinds of trees.
- **Memory Management**: When memory is an issue, and you wish to distribute memory as required.

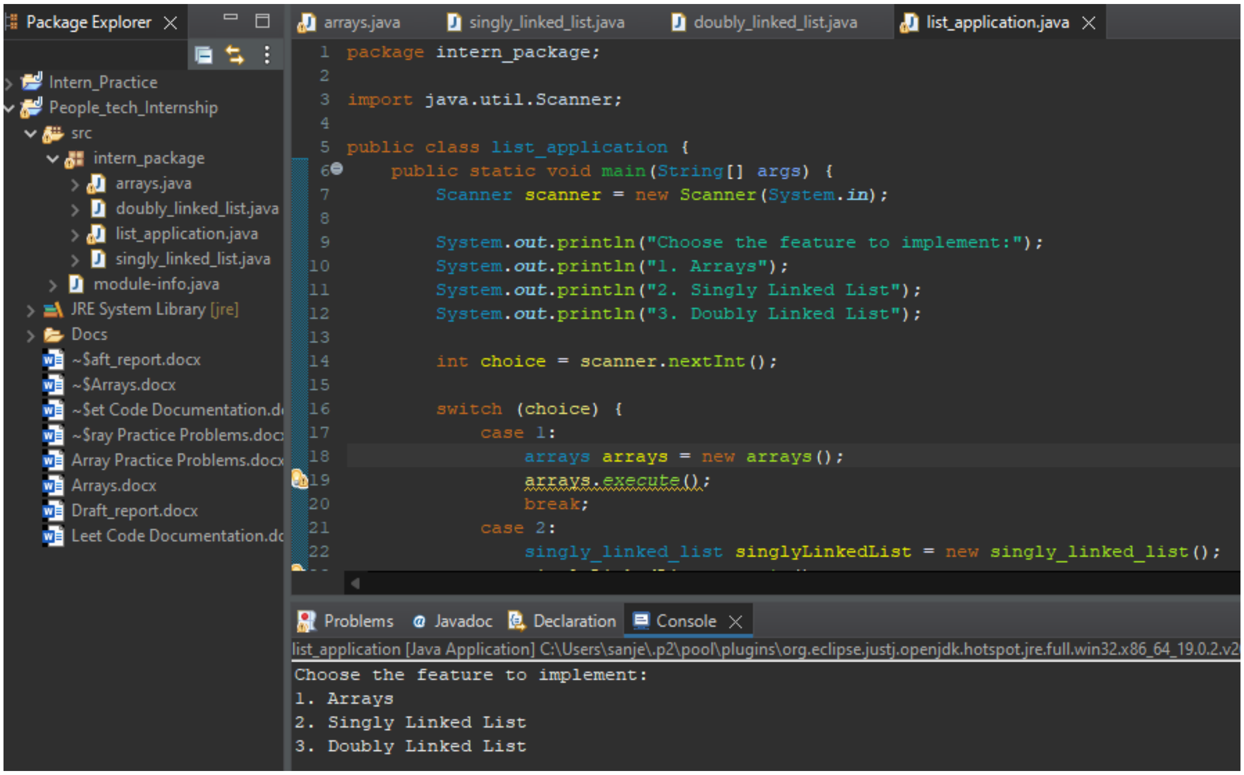
- **Polynomial Manipulation**: Polynomials are frequently represented by linked lists, in which each node stands for a word.

**Applications of Linked Lists in the Real World**:

- **Music Player Playlists**: The "next" and "previous" features in music players may be implemented using linked lists.
- **Undo Functionality in Applications**: Using a linked list, a stack may effectively handle undo actions.
- **Hash Tables with Chaining**: To manage collisions, hash tables are implemented using linked lists.
- **Operating System Memory Management**: Linked lists assist operating systems in managing free memory blocks.

| **Aspect** | **Linked Lists** | **Arrays** |
|---|---|---|
| Memory Allocation | Dynamic | Static (for fixed arrays) or Dynamic (for resizable arrays) |
| Element Access | O(n) – Sequential access | O(1) – Random access |
| Insertion at Beginning | O(1) | O(n) – Requires shifting elements |
| Insertion at End | O(n) for singly linked list, O(1) for doubly linked list with tail pointer | O(1) amortized for dynamic arrays, O(n) if resizing is needed |
| Deletion at Beginning | O(1) | O(n) – Requires shifting elements |
| Deletion at End | O(n) for singly linked list, O(1) for doubly linked list | O(1) |
| Memory Overhead | Higher (due to storage of next/prev pointers) | Lower |
| Cache Performance | Poor | Good |

3. **Hands-On Practice:**



```java
package intern_package;

import java.util.Scanner;

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

        System.out.println("Choose the feature to implement:");
        System.out.println("1. Arrays");
        System.out.println("2. Singly Linked List");
        System.out.println("3. Doubly Linked List");

        int choice = scanner.nextInt();

        switch (choice) {
            case 1:
                arrays arrays = new arrays();
                arrays.execute();
                break;
            case 2:
                singly_linked_list singlyLinkedList = new singly_linked_list();
```

```
Choose the feature to implement:
1. Arrays
2. Singly Linked List
3. Doubly Linked List
```

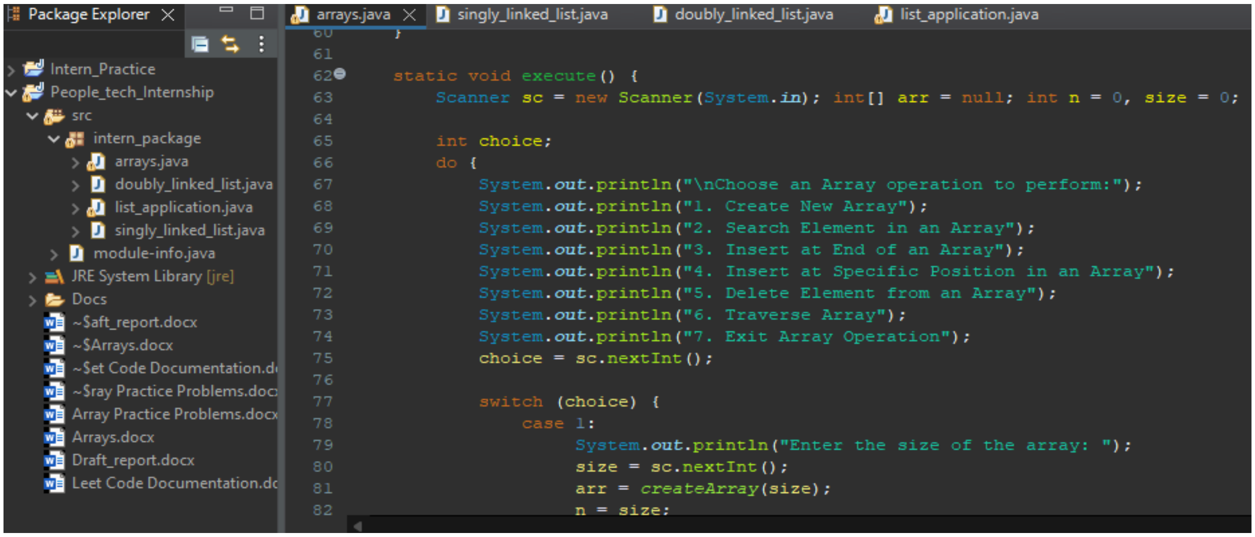
In this code, i've written a primary application (list_application class) that serves as an entry point for selecting and conducting certain operations from three separate classes: arrays, singly_linked_list, and doubly_linked_list. The user picks the type of data structure with which to operate, and the operations of the associated class are carried out depending on that option.

The switch statement analyzes the choice variable to determine which option the user chose.

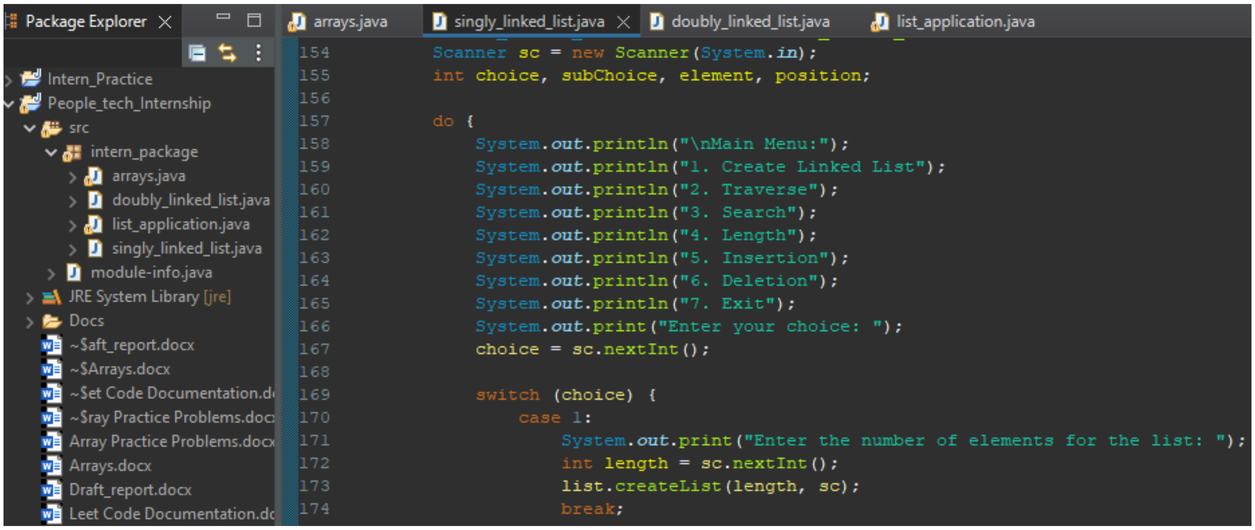
Case 1: If the user picks 1, the arrays class is instantiated and the execute function is called. The execute method in arrays should include all array-related actions.

Case 2: If the user picks 2, an instance of the singly_linked_list class is created and the execute function is called. The execute method should include operations related to singly linked lists.

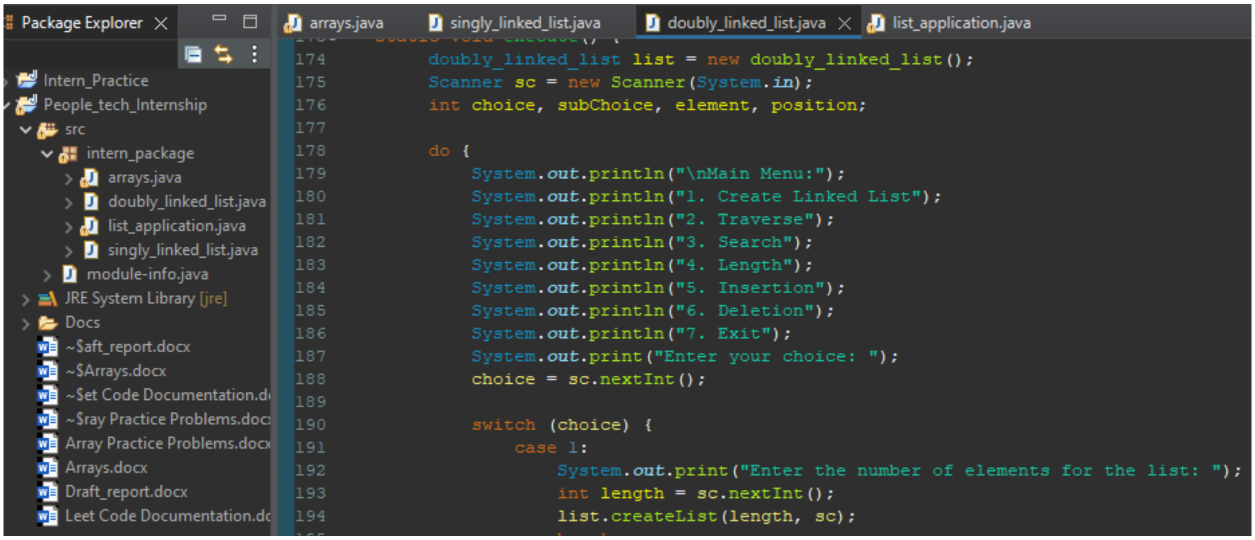
Case 3: If the user picks 3, the doubly_linked_list class is instantiated and the execute function is invoked. This method should include all actions related to doubly linked lists.



```java
    }

    static void execute() {
        Scanner sc = new Scanner(System.in); int[] arr = null; int n = 0, size = 0;

        int choice;
        do {
            System.out.println("\nChoose an Array operation to perform:");
            System.out.println("1. Create New Array");
            System.out.println("2. Search Element in an Array");
            System.out.println("3. Insert at End of an Array");
            System.out.println("4. Insert at Specific Position in an Array");
            System.out.println("5. Delete Element from an Array");
            System.out.println("6. Traverse Array");
            System.out.println("7. Exit Array Operation");
            choice = sc.nextInt();

            switch (choice) {
                case 1:
                    System.out.println("Enter the size of the array: ");
                    size = sc.nextInt();
                    arr = createArray(size);
                    n = size;
```



```java
        Scanner sc = new Scanner(System.in);
        int choice, subChoice, element, position;

        do {
            System.out.println("\nMain Menu:");
            System.out.println("1. Create Linked List");
            System.out.println("2. Traverse");
            System.out.println("3. Search");
            System.out.println("4. Length");
            System.out.println("5. Insertion");
            System.out.println("6. Deletion");
            System.out.println("7. Exit");
            System.out.print("Enter your choice: ");
            choice = sc.nextInt();

            switch (choice) {
                case 1:
                    System.out.print("Enter the number of elements for the list: ");
                    int length = sc.nextInt();
                    list.createList(length, sc);
                    break;
```



```java
    public void execute() {
        doubly_linked_list list = new doubly_linked_list();
        Scanner sc = new Scanner(System.in);
        int choice, subChoice, element, position;

        do {
            System.out.println("\nMain Menu:");
            System.out.println("1. Create Linked List");
            System.out.println("2. Traverse");
            System.out.println("3. Search");
            System.out.println("4. Length");
            System.out.println("5. Insertion");
            System.out.println("6. Deletion");
            System.out.println("7. Exit");
            System.out.print("Enter your choice: ");
            choice = sc.nextInt();

            switch (choice) {
                case 1:
                    System.out.print("Enter the number of elements for the list: ");
                    int length = sc.nextInt();
                    list.createList(length, sc);
```

In this application, we’ve created a simple interface that lets users choose to perform operations on different data structures—arrays, singly linked lists, or doubly linked lists—based on their input. The list_application class serves as the entry point, where we prompt the user to select one of the three data structures by displaying a menu with numbered options. After taking the user’s input, we use a switch statement to determine which data structure’s operations to execute. The switch statement checks the user's choice, and based on the chosen option, it creates an instance of the appropriate class (arrays, singly_linked_list, or doubly_linked_list). Then, it calls the execute method on the selected instance, which contains all the related operations for that data structure.Arrays Class: Contains array operations such as adding, deleting, sorting, and searching. The execute function in this class may prompt the user to select amongst several array operations.

**Approach Overview:**

The code structure is designed in a way that each data structure (array, singly linked list, doubly linked list) has its own set of operations. These operations are encapsulated into methods that are called based on user choices from the main menu. The core approach is modular, with dedicated methods for creating arrays or lists, performing operations like search, insertion, and deletion, and managing the flow of the program.

1. **Main Menu and Flow Control**:

- The user starts by selecting the data structure (array, singly linked list, or doubly linked list).
- After choosing the data structure, the relevant menu is displayed for selecting operations specific to that structure (like inserting elements, traversing the structure, etc.).
- The program continues to loop until the user chooses to exit.

2. **Switch-Case Structure**:

- The switch-case structure handles routing the user to the corresponding functionality for each data structure.
- It ensures that the logic for each data structure is compartmentalized and easy to manage.

For example:

- **Array Operations**: Involves operations like creating an array, searching for an element, inserting at the end, inserting at a specific position, deleting elements, and traversing the array.

- **Singly Linked List Operations**: Operations include creating a linked list, traversing the list, searching for an element, inserting and deleting nodes, and checking the length of the list.
- **Doubly Linked List Operations**: Similar to the singly linked list, but includes additional operations such as traversing the list in reverse and managing the previous and next node pointers for both directions.

**Challenges Faced During Implementation:**

1. **Handling Switch-Case Logic for Different Data Structures**:

- **Challenge**: The use of a switch-case for routing operations can lead to a lot of repetition if not handled properly. Each structure (array, singly linked list, doubly linked list) needs a separate set of options, which can make the code cumbersome as the number of operations grows.
- **Solution**: To mitigate this, I ensured that each data structure has its own well-defined methods. I used smaller helper methods for each operation (e.g., createArray(), insertAtEnd(), searchElement()) to keep the code clean and organized.

2. **Complexity of Operations Between Data Structures**:

- **Challenge**: Each operation for arrays and linked lists has a different time complexity and implementation logic. For example, arrays involve direct indexing, while linked lists require traversal to find nodes for insertion or deletion. Managing these differences and ensuring that each structure's logic is implemented correctly within the switch-case structure is challenging.
- **Solution**: For arrays, direct indexing was used, while for linked lists, I made sure to handle node pointers correctly. I kept separate helper methods for traversal, insertion, and deletion specific to each data structure to maintain clarity.

3. **Error Handling**:

- **Challenge**: Handling edge cases such as invalid inputs (e.g., searching for an element in an empty list or array, or inserting/deleting at invalid positions) can cause runtime errors. For example, when inserting into an array, if the user specifies a position greater than the array length, or when deleting from an empty linked list, the program should not crash.
- **Solution**: I added checks for edge cases like verifying array bounds before performing insertions and deletions and checking if a list is empty before trying to delete or search for elements. For each operation, appropriate error messages were displayed to guide the user.

4. **Array vs Linked List Specific Operations**:

- **Challenge**: The operations for arrays (such as insertion at a specific index) are simple as arrays use indexes for direct access. But for linked lists, especially doubly linked lists, the insertion and deletion operations require additional logic, including managing the next and prev pointers. Ensuring consistency in the implementation of these operations, especially for the doubly linked list, was a challenge.
- **Solution**: I separated the logic for each structure. For singly linked lists, I used the next pointer to traverse and modify nodes. For doubly linked lists, I had to handle both next and prev pointers, which required additional care to ensure that the previous node correctly points to the new node during insertions or deletions.

5. **Memory Management and Efficiency**:

- **Challenge**: Arrays have a fixed size, meaning they require resizing operations (copying the entire array to a new array with more space) for insertions at capacity. Linked lists do not have this limitation but require careful memory management due to dynamic node allocation.
- **Solution**: I ensured that resizing was handled correctly when inserting into a full array, and linked list operations dynamically adjusted pointers without causing memory leaks.

6. **User Experience**:

- **Challenge**: Providing a clear, easy-to-understand user interface is essential. The menu should be intuitive, and users should get clear feedback about the results of their actions (such as successful insertions, deletions, or errors).
- **Solution**: I used clear prompts and outputs to guide the user, displaying the results of each operation and handling invalid inputs with meaningful error messages.

**1. Stacks**:

**1. Introduction to Stacks.**
A stack is a linear data structure that adheres to the LIFO principle, which stands for Last In, First Out. This indicates that the final piece added to the stack will be the first one deleted.

**Key Characteristics:** LIFO Order: Elements are added and withdrawn from the top of the stack.

**Restricted Access**: Elements in the center of the stack cannot be accessed directly; operations take place at the top.

**Basic Operations**:

1. **Push**: Adds an element to the top of the stack.
2. **Pop**: Removes the element from the top of the stack.
3. **Peek (or Top)**: Retrieves the element at the top without removing it.
4. **isEmpty**: Checks if the stack contains no elements.
5. **isFull**: Checks if the stack has reached its maximum capacity (in fixed-size implementations).

**Examples**:

- A stack of plates: The last plate placed on top is the first one removed.
- Undo functionality in text editors.

2. **Array implementation of stacks**: This implementation represents a stack with a fixed-size array. The array's elements are used to mimic stack operations.

**Characteristics**:

- The array size specifies the stack's maximum capacity.
- The index variable represents the top of the stack.

**Advantages**:

- Easy to implement.
- Push, pop, and peek operations have constant temporal complexity ($O$(1 ) O(1)).

**Disadvantages**:

- Fixed size limits flexibility, which might lead to overflow.
- Inefficient memory utilization occurs when the array size is excessively big and not completely utilized.

**Use Case:**

- Suitable when the stack's maximum size is predetermined and does not vary dynamically.

**3. Linked List Implementation of Stacks:** In this implementation, the stack is represented using a linked list. Each node contains data and a pointer to the next node, with the top of the stack pointing to the last inserted node.

**Characteristics**:

- The stack grows dynamically as new elements are added.
- Memory is allocated on demand.

**Advantages**:

- No fixed size: The stack can grow or shrink based on the number of elements.
- Efficient memory utilization as only required memory is used.

**Disadvantages**:

- Requires extra memory for pointer/reference storage.
- Slightly slower than arrays due to pointer manipulation overhead.

**Use Case**:

- Useful when the size of the stack is unknown or changes frequently

**4. Applications for Stacks**: Stacks are flexible and commonly utilized in computer science and real-world applications. Key applications include:

**Expression Evaluation and Conversion:**

- Convert infix expressions (e.g., $A$+$B$) to postfix or prefix notation.
- Evaluate postfix expressions.

**Function Call Management**:

- A call stack is used by programming languages to organize function calls and returns, particularly in recursion.

**Parenthesis matching and syntax checking:**

- Stacks are used to ensure that balanced parentheses or other symbols in code are correct.

**Undo/Redo Functionality:**

- Text editors use stacks for undo and redo operations to keep track of action history.

**Browser Navigation:**

- Web browsers employ stacks to implement back and forward navigation.

**Backtracking**:

- Stacks are used to recall prior states by algorithms that solve mazes, puzzles, and decision trees.

**Key Differences Between Implementations:**

| Aspect | Array Implementation | Linked List Implementation |
|---|---|---|
| **Size** | Fixed size, determined at creation. | Dynamic size, grows as needed. |
| **Memory Usage** | Pre-allocated memory, can lead to wastage. | Memory allocated as needed. |
| **Complexity** | Simple and faster due to direct indexing. | Slightly complex due to pointer usage. |
| **Overflow** | It can occur if stack exceeds array size. | No overflow unless memory is exhausted. |

**2. Queue**:

1. **Introduction to Queues.**
A queue is a linear data structure that adheres to the First In, First Out (FIFO) principle. This means that the first item put to the queue is the first to be withdrawn. Queues are commonly used in scheduling, resource sharing, and other real-world applications.

**Key Characteristics:**

- FIFO Order: Elements are added to one end (rear) and deleted from the other (front).
- Restricted Access: Elements may only be removed from the front or added from the back.

**Basic operations:**

- Enqueue: Add an element to the back of the queue.
- Dequeue means to remove an element from the queue's front.
- Front/Peek retrieves the front element without deleting it.
- isEmpty: Determine whether the queue is empty.
- isFull (for fixed-size implementations): Check if the queue is full.

**Real-Life Examples:**

- A queue of clients at a bank or ticket desk.
- Print queue on a printer.

**2. Array Implementation of Queues**

In this implementation, the queue is represented using an array.

**Characteristics**:

- Elements are stored in a linear array.
- Two indices, front and rear, track the positions for removing and adding elements.

**Advantages**:

- Simple and easy to implement.
- Constant time operations (O(1)O(1)O(1)) for enqueue and dequeue.

**Disadvantages**:

- Fixed size may lead to overflow.
- If not managed carefully, memory can be wasted as elements are dequeued.

**3. Implementation of Queues using Linked Lists**: In this implementation, the queue is represented as a linked list. Each node has data and a reference to the next node.

**Characteristics**:

- The linked list's head represents the front of the queue.
- The linked list's tail represents the queue's rear.

**Advantages**:

- The queue's size is dynamic, meaning it can increase or shrink depending on the quantity of entries.
- Efficient memory management: Memory is allocated only when needed.

**Disadvantages**:

- Requires additional memory for pointers.
- Slightly slower due to pointer manipulations.

**Use Case**:

- Useful when the size of the queue is unknown or changes frequently.

4. **Implementing a stack Using a single queue:**

- To create a stack with a single queue, we alter the enqueue and dequeue processes so that the last element added behaves as the stack's top.

**Approach**:

- Push elements into the queue as normal.
- To simulate the stack's LIFO behavior, after adding a new element, rotate all prior elements in the queue to the rear.

**5. Level Order Traversal in Spiral Form:** Level order traversal in spiral (or zig-zag) form is a binary tree traversal strategy in which nodes are visited level by level in an order that alternates between left-to-right and right-to-left.

**Approach**:

- Use two stacks or a deque to switch between the two traversal orders.
- Push child nodes in the opposite order for each level to create the zig-zag pattern.

**6. Check Whether a Binary Tree is Complete**

A binary tree is considered complete if all levels except possibly the last are fully filled, and all nodes are as far left as possible.

**Approach**:

1. Perform a level order traversal using a queue.
2. After encountering a node with no children, ensure that no subsequent node in the traversal has children.

**7. Implementing a Stack with Two Queues**

- Push Operation: Add a new element to the empty queue, then move all elements from the other queue to the current one.
- Pop Operation: Simply delete the first member of the current queue.

**8. Circular Queue:** A circular queue overcomes the restrictions of a linear queue by utilizing the empty space left after dequeuing entries.

**Characteristics**:

- When the rear pointer hits the end of the array, it returns to the beginning.
- Efficient memory use as items is added and discarded.

**Advantages**:

- There is no waste of space, as observed in a basic array-based queue.
- Suitable for real-time applications, such as task scheduling.

**Disadvantages**:

- Due to the wrapping behavior, implementation is little more difficult.

9. **Reversing the Queue**: Reversing a queue involves placing its members in reverse order.

**Approach**:

**Using A Stack**:

- Dequeue elements from the queue and add them to a stack.
- Remove elements from the stack and place them back in the queue.

**Using recursion**:

- Dequeue the first element, then reverse the remaining queue and finally enqueue the dequeued element at the end.

**3. Hashing:**

**1. Hashing**: Hashing is a technique used to map data elements (keys) to indices in a fixed-size array, called a hash table, using a hash function. It enables efficient insertion, deletion, and lookup operations, often in O (1) average time complexity.

**Key concepts**:

- Hash Function: A function that calculates the index of a key.
- A hash table is an array-like data structure used to store data.
- Collision occurs when two keys map to the same index.
- The Load Factor is the ratio of the number of elements in the hash table to its size.

2**. Separate chaining:** Separate chaining is a strategy for handling collisions in hashing that involves keeping a linked list (or other data structure) at each index of the hash table.

**How It Works:**

- If a collision occurs, all entries that correspond to the same index are put in a linked list at that position.
- During insertion, the element is added to the linked list.
- During the search, the linked list is traversed to locate the specified element.

**Advantages**:

- Simple to implement.
- There is no limit to the number of items that may be stored, as long as memory is sufficient.

**Disadvantages**:

- Performance worsens as the load factor rises, resulting in larger linked lists.
- Pointers in linked lists take up more memory.

**3. Open Addressing:** Open addressing resolves collisions by finding another available slot within the hash table.

**Techniques**:

1. **Linear Probing**:
   - If a collision occurs, the algorithm checks subsequent indices (in a linear fashion) until an empty slot is found.
   - Example: index=(hash+i)%table_sizeindex = (hash + i) \% table_sizeindex=(hash+i)%table_size, where iii increments linearly.
2. **Quadratic Probing**:

- Instead of linear steps, indices are checked with a quadratic offset.
- Example: index=(hash+i2)%table_sizeindex = (hash + i^2) \% table_sizeindex=(hash+i2)%table_size.

3. **Double Hashing**:

- A secondary hash function determines the step size.
- Example: index=(hash1+i×hash2)%table_sizeindex = (hash1 + i \times hash2) \% table_sizeindex=(hash1+i×hash2)%table_size.

**Advantages**:

- Memory-efficient as no additional structures like linked lists are required.
- Cache-friendly due to contiguous memory storage.

**Disadvantages**:

- Performance can degrade significantly with a high load factor.
- Requires careful handling of deletions to avoid breaking the search chain.

4. Looking for Patterns: Hashing is also employed in pattern matching algorithms, such as locating substrings inside a text.

**Example of Rabin-Karp Algorithm:**

- The pattern and substrings of the text are hashed.
- For efficiency, hash values are compared rather than substrings themselves.
- If a hash match is discovered, the actual substring and pattern are matched to prevent false positives (hash collisions).

### 5. Advantages of BST over Hash Table

While hash tables offer O (1)O(1)O(1) average time complexity for most operations, **Binary Search Trees (BSTs)** have unique advantages:

| **Feature** | **BST** | **Hash Table** |
|---|---|---|
| **Order Maintenance** | Elements are stored in sorted order, allowing in-order traversal. | No order is maintained. |
| **Range Queries** | Efficient for range queries (e.g., find all keys between two values). | Inefficient for range queries. |
| **Memory Usage** | Does not require additional memory for storing hash functions or handling collisions. | Requires extra memory for collisions. |
| **Dynamic Size** | It can dynamically grow without rehashing. | Rehashing is needed when the load factor is high. |
| **Key Comparison** | Keys are compared directly, avoiding hash collisions. | Hash collisions can degrade performance. |
| **Performance in Worst Case** | O(log n)O(\log n)O(logn) for balanced BSTs. | O(n)O(n)O(n) in the worst case due to collisions. |

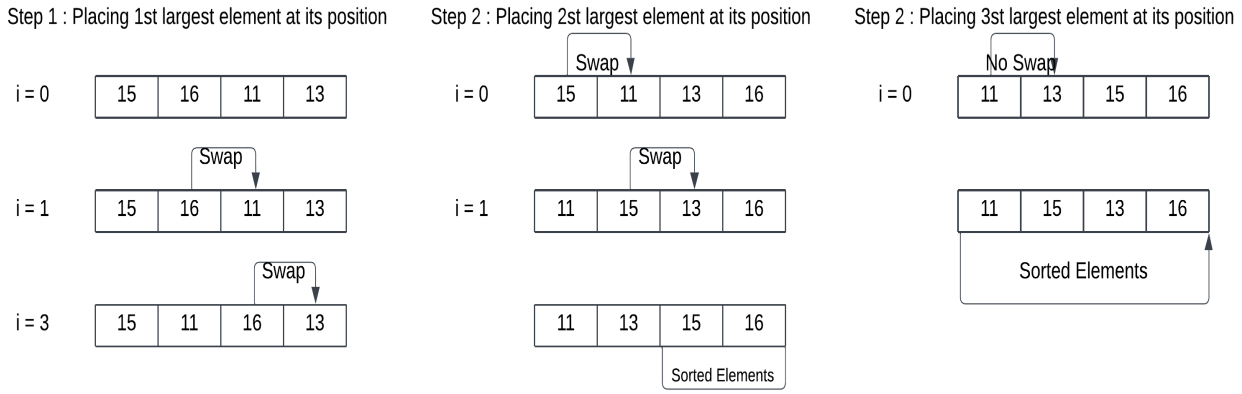
# ALGORITHAMS

1. **Understanding Sorting and Searching Algorithms:**

Bubble Sort Algorithm:

## Bubble Sort:

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where **n** is the number of items.



### Bubble Sort Pseudocode

Time Complexity : O (n^2)

```
voidbubbleSort(int numbers[], intarray_size){
  int  i,  j,  temp;
  for ( i = ( array_size - 1 ) ; i >= 0; i - -)
  for ( j = 1; j < = i ; j + +)
  if ( numbers [ j-1 ] > numbers[ j ] ) {
    temp = numbers[ j - 1 ];
    numbers[ j - 1 ] = numbers[ j ] ;
    numbers[ j ] = temp ;
  }
}
```

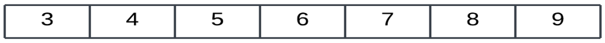
Binary Search Algorithm:

# Binary Search Algoritham

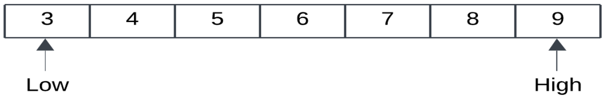
An effective method for locating an item in a sorted list of elements is the Binary Search Algorithm. It divides the search interval in half repeatedly. The search proceeds in the lower half if the target value is less than the value in the middle of the interval, or in the upper half if the target value is greater. Until the value is located or the search period is empty, the procedure is repeated.

Element to find be 4

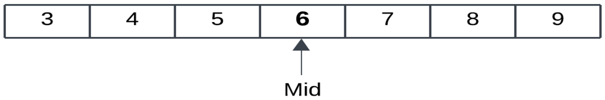
Initial Array



Set two pointers `low` and `high` at the lowest and the highest positions respectively.



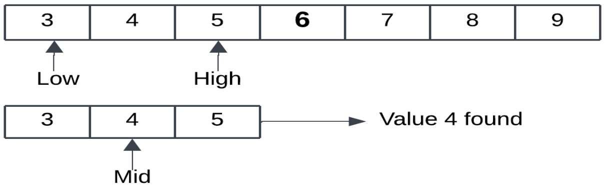
Find the middle position `mid` of the array ie. `mid = (low + high)/2` and `arr[mid] = 6`.



If `x == arr[mid]`, then return `mid`. Else, compare the element to be searched with `arr[mid]`.

If `x > arr[mid]`, compare `x` with the middle element of the elements on the right side of `arr[mid]`. This is done by setting `low` to `low = mid + 1`

Else, compare `x` with the middle element of the elements on the left side of `arr[mid]`. This is done by setting `high` to `high = mid - 1`.



Time Complexities

- Best case complexity: `O(1)`
- Average case complexity: `O(log n)`
- Worst case complexity: `O(log n)`

**Pseudocode :**

**Iterative Method**

```
do until the pointers low and
high meet each other.
    mid = (low + high)/2
    if (x == arr[mid])
        return mid
    else if (x > arr[mid])
        low = mid + 1
    else
        high = mid - 1
```

**Recursive Method**

```
binarySearch(arr, x, low, high)
    if low > high
        return False
    else
        mid = (low + high) / 2
        if x == arr[mid]
            return mid
        else if x > arr[mid]
          return binarySearch(arr, x,
mid + 1, high)
        else
            return binarySearch(arr,
x, low, mid - 1)
```

Linear Search Algorithm:

## Linear Search Algoritham:

Iterating through the array's elements, Linear Search determines whether the current element is equal to the target element. Return the current element's index if any element is determined to be equal to the target element. Otherwise, return -1 if the element cannot be located if no element is equal to the target element. Sequential search is another name for linear search.

### Time Complexity:

Best Case: O(1)
Worst Case:O(N) where N is the size of the list.
Average Case: O(N)

### Pseudocode:

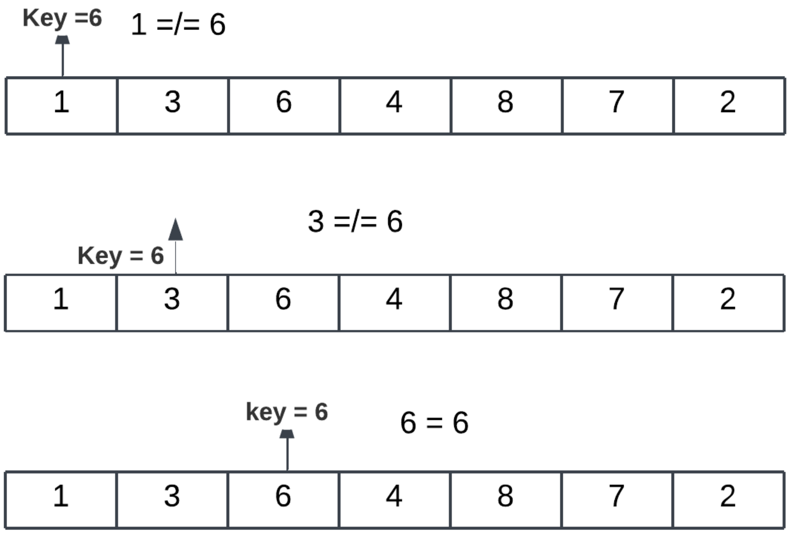
```
procedure linear_search (list, value)
  for each item in the list
    if match item == value
      return the item's location
    end if
  end for
end procedure
```

### Example :

Compare the key with each element one by one starting from the 1st element

Compare the key with each 2nd element which is not equal

Compare the key with each 3nd element which is equal so stop the search



## Merge Sort:

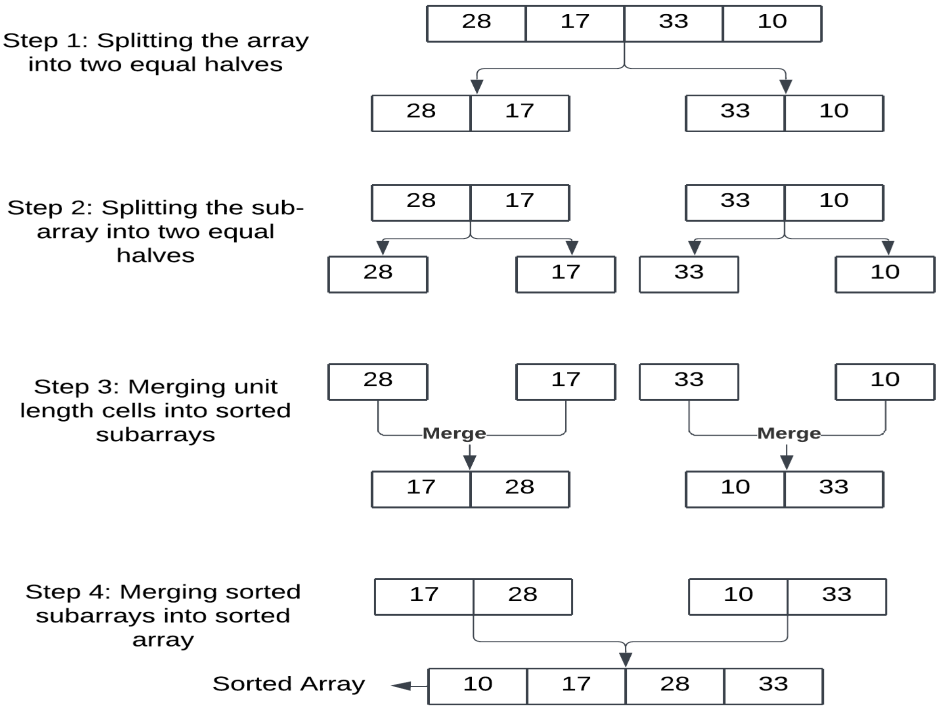
### Merge Sort Algoritham:

A well-liked sorting algorithm with a reputation for stability and efficiency is merge sort. It sorts a specified array of elements using the divide-and-conquer strategy.

This is a detailed breakdown of how merge sort operates:

Divide: Until the list or array can no longer be divided, divide it recursively into two halves.
Conquer: The merge sort method is used to sort each subarray separately.
Merge: The subarrays that have been sorted are combined once more in the same order. Until every element from both subarrays has been combined, the operation is repeated.

Example:


Step 1: Splitting the array into two equal halves
28
17
33
10
Step 2: Splitting the sub-array into two equal halves
Step 3: Merging unit length cells into sorted subarrays
Merge
Merge
Step 4: Merging sorted subarrays into sorted array
Sorted Array
10
17
28
33

Pseudocode:

```
function MERGE-SORT(A, left, right)
    if left < right
        middle = (left + right) / 2
        MERGE-SORT(A, left, middle)
        MERGE-SORT(A, middle + 1, right)
        MERGE(A, left, middle, right)

function MERGE(A, left, middle, right)
```

**Complexity Analysis of Merge Sort:**

- **Time Complexity:**
  - **Best Case:** O(n log n), When the array is already sorted or nearly sorted.
  - **Average Case:** O(n log n), When the array is randomly ordered.
  - **Worst Case:** O(n log n), When the array is sorted in reverse order.

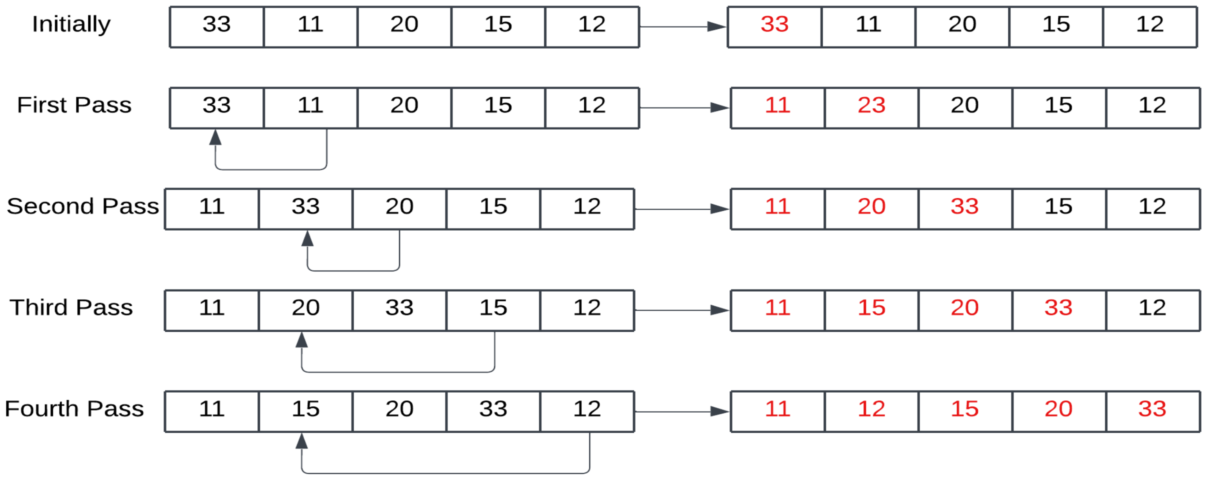
## Insertion Sort:

### Insertion Sort :

Iteratively placing each member of an unsorted list in the appropriate location within a sorted section of the list is how the straightforward sorting algorithm known as "insertion sort" operates. Sorting playing cards in your hands is similar to that. Sorted cards and unsorted cards are the two categories into which you divide the cards. A card is then selected from the unsorted group and placed in the appropriate location within the sorted group.

Since the array's initial element is presumed to be sorted, we begin with the second member.
Check if the second element is less than the first by comparing them, then switch them.
The complete array has been sorted.

Example:



Pseudocode :

```
procedure insertionSort(array,N )
array – array to be sorted
N- number of elements
begin
int freePosition
int insert_val

for i = 1 to N -1 do:

insert_val = array[i]
freePosition = i
while freePosition > 0 and array[freePosition -1] > insert_val do:
array [freePosition] = array [freePosition -1]
freePosition = freePosition -1
end while
array [freePosition] = insert_val
end for
end procedure
```

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

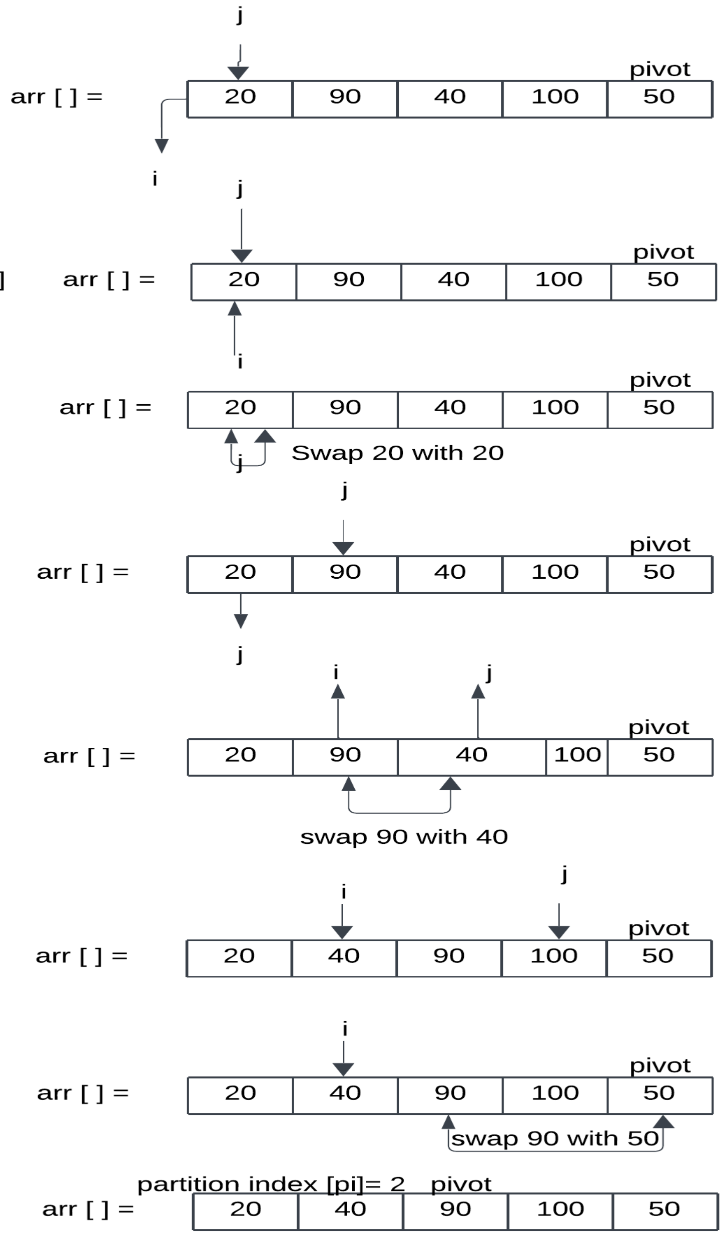
## Quick Sort Algoritham:

### Quick Sort Algoritham:

A sorting method called QuickSort, which is based on the Divide and Conquer strategy, selects an element as a pivot and divides the provided array around the pivot by positioning it correctly within the sorted array.

The algorithm consists of three primary steps:
Select a Pivot: Choose a pivot element from the array. One can choose a different pivot (e.g., median, random element, first element, or final element).
Divide up the array: The array should be rearranged around the pivot. All items larger than the pivot will be on its right after partitioning, while all elements less than the pivot will be on its left. Once the pivot is in the proper place, we can determine its index.
Recursively Make a call: The two partitioned sub-arrays (left and right of the pivot) should undergo the same procedure recursively.

Example :



Pivot selection : The last element arr[4] = 40 is choosen as the pivot initial pointers i = -1 and j = 0

| arr [ ] = | 20 | 90 | 40 | 100 | 50 |
|---|---|---|---|---|---|

Since , arr [ j ] < pivot [ 10 < 40 ] increment i to 0 and swap arr [ i ] with arr [ j ], increment j by 1

| arr [ ] = | 20 | 90 | 40 | 100 | 50 |
|---|---|---|---|---|---|

| arr [ ] = | 20 | 90 | 40 | 100 | 50 |
|---|---|---|---|---|---|

Since arr [ j ] > pivot [ 90 < 50 ] no swap needed increment j by 1

| arr [ ] = | 20 | 90 | 40 | 100 | 50 |
|---|---|---|---|---|---|

Since arr [ j ] > pivot [ 40  < 50 ] increment i by 1 and swap arr[ i ] with arr[ j ]. Increment j by 1

| arr [ ] = | 20 | 90 | 40 | 100 | 50 |
|---|---|---|---|---|---|

arr [ j ] > pivot[ 100 < 40] no swap needed increment j by 1

| arr [ ] = | 20 | 40 | 90 | 100 | 50 |
|---|---|---|---|---|---|

SInce traversal of j ended now move pivot to its current position swap arr [ i+1 } = aee [ 2 ] with arr [ 4 ] = 50

| arr [ ] = | 20 | 40 | 90 | 100 | 50 |
|---|---|---|---|---|---|

| arr [ ] = | 20 | 40 | 90 | 100 | 50 |
|---|---|---|---|---|---|

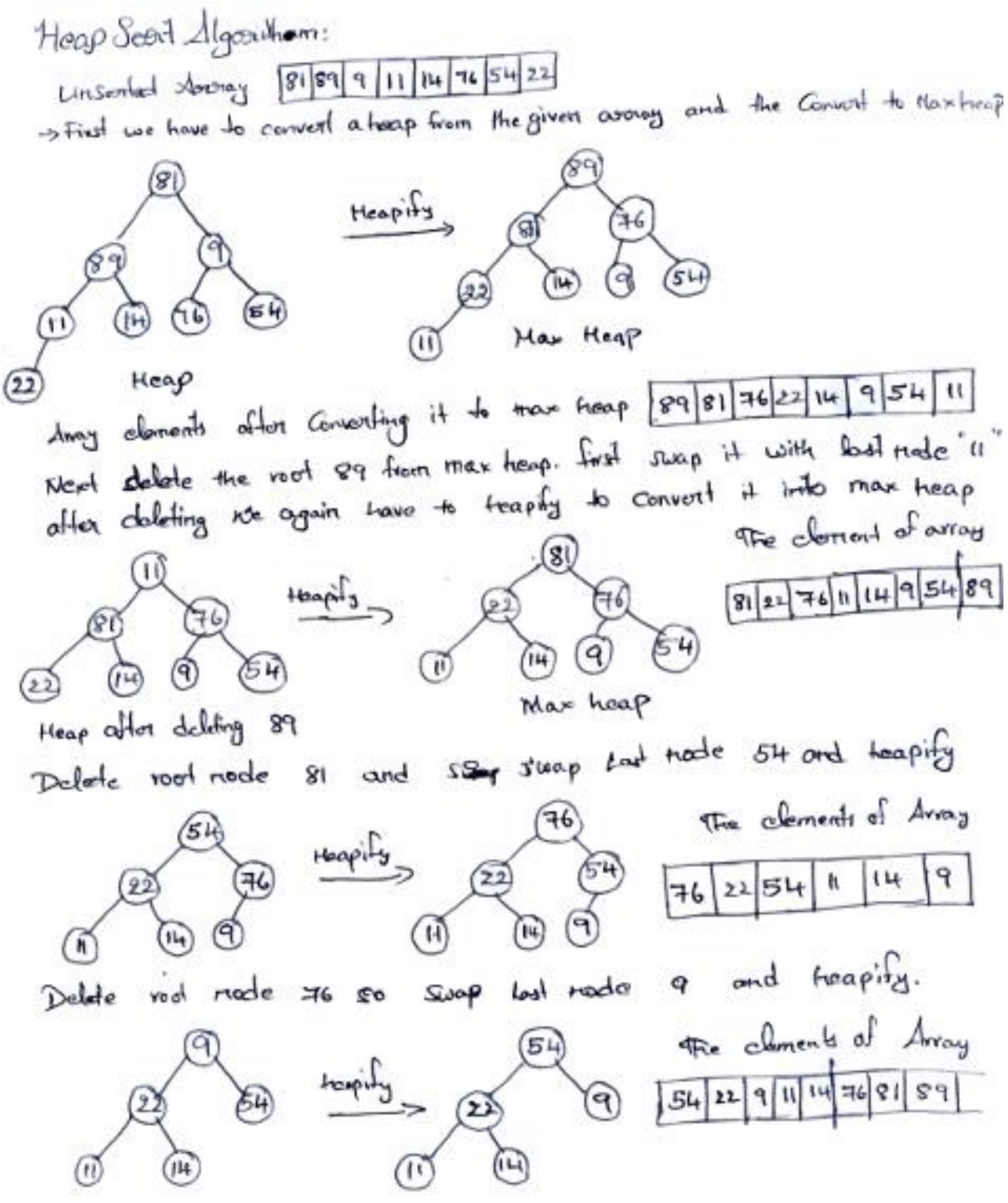
Pseudocode :

```
function partitionFunc(left, right, pivot)
  leftPointer = left
  rightPointer = right - 1

  while True do
    while A[++leftPointer] < pivot do
    end while

    while rightPointer > 0 && A[--rightPointer] > pivot do
    end while

    if leftPointer >= rightPointer
      break
    else
      swap leftPointer,rightPointer
    end if
  end while

  swap leftPointer,right
  return leftPointer
end function
```

**Time Complexity:**

- **Best Case:** ($\Omega(n \log n)$), Occurs when the pivot element divides the array into two equal halves.
- **Average Case** ($\theta(n \log n)$), On average, the pivot divides the array into two parts, but not necessarily equal.
- **Worst Case:** ($O(n^2)$), Occurs when the smallest or largest element is always chosen as the pivot

**Heap Sort:**



Heap Sort Algoritham:

Unsorted Array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 81 | 89 | 9 | 11 | 14 | 76 | 54 | 22 |

→ First we have to convert a heap from the given array and the Convert to Max heap

Array elements after Converting it to max heap

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 89 | 81 | 76 | 22 | 14 | 9 | 54 | 11 |

Next delete the root 89 from max heap. first swap it with last node "11" after deleting we again have to heapify to convert it into max heap

The element of array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 81 | 22 | 76 | 11 | 14 | 9 | 54 | 89 |

Delete root node 81 and ~~S~~ swap last node 54 and heapify

The elements of Array

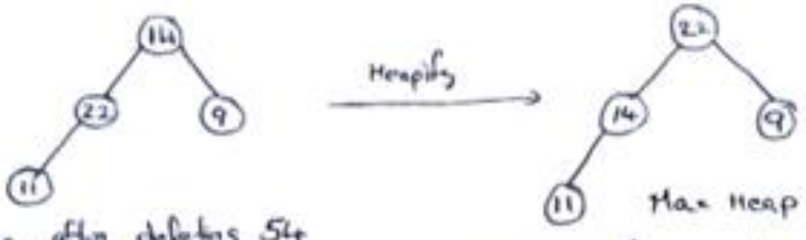
| | | | | | |
|---|---|---|---|---|---|
| 76 | 22 | 54 | 11 | 14 | 9 |

Delete root node 76 so swap last node 9 and heapify.

The elements of Array

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 54 | 22 | 9 | 11 | 14 | 76 | 81 | 89 |

Delete root element 54 So swap it with last element 14.



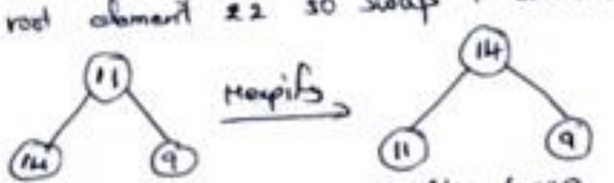
Heap after deleting 54

Max Heap

The elements of Array

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 22 | 14 | 9 | 11 | 54 | 76 | 81 | 89 |

Delete root element 22 so swap it with last node 11

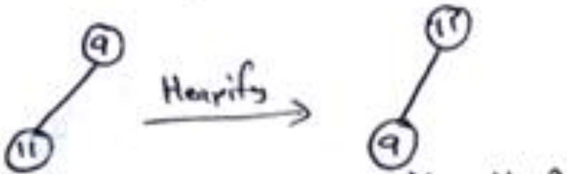


Heap after deleting 22

. Max heap

The elements of Array

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 14 | 11 | 9 | 22 | 54 | 76 | 81 | 89 |



Heap after deleting 14

Max Heap

After swapping the array elements m with 9 Converting the heap into max-heap the elements are

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 11 | 9 | 14 | 22 | 54 | 76 | 81 | 89 |

Delete root element 11 so swap it with last element 9

⑨ $\xrightarrow{\text{Heapify}}$ ⑨

Heap after del 11 Max heap

The array elements are

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 9 | 11 | 14 | 22 | 54 | 76 | 81 | 89 |

heap has only one element left Heap will be empty.

⑨ $\xrightarrow{\text{Remove 9}}$ empty

Sorted Array →

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 9 | 11 | 14 | 22 | 54 | 76 | 81 | 89 |

Time Complexity Best Case = $O(n \log n)$

Average case = $O(n \log n)$

worst case = $O(n \log n)$

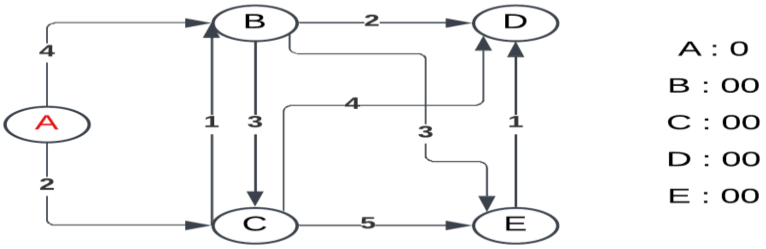
### 3. Algorithms Tutorials:

**Dijkstra’s Algoritham:**

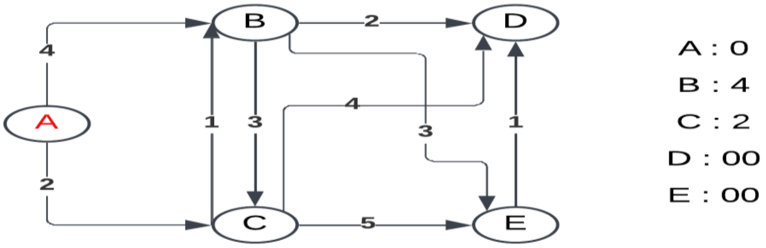
The idea is to generate a **SPT (shortest path tree)** with a given source as a root. Maintain an Adjacency Matrix with two sets,

- one set contains vertices included in the shortest-path tree,
- The other set includes vertices not yet included in the shortest-path tree.

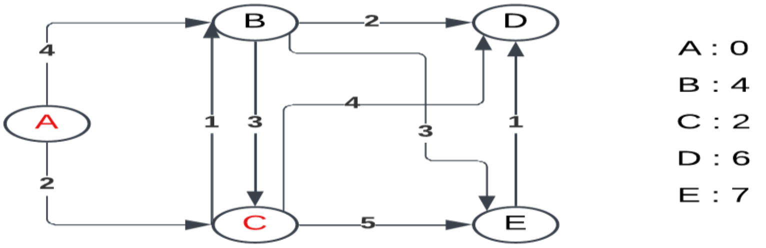
At every step of the algorithm, find a vertex that is in the other set (set not yet included) and has a minimum distance from the source.



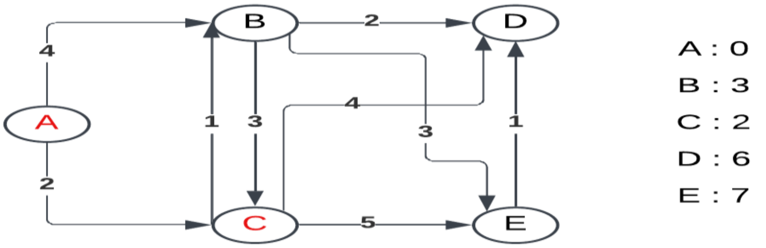
Unvisited Nodes = {A, B, C, D, E}



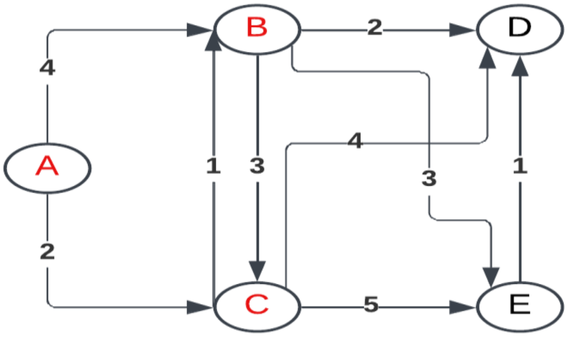
Unvisited Nodes = {A, B, C, D, E}



Unvisited Nodes = {A, B, C, D, E}



Unvisited Nodes = {A, B, C, D, E}



A : 0

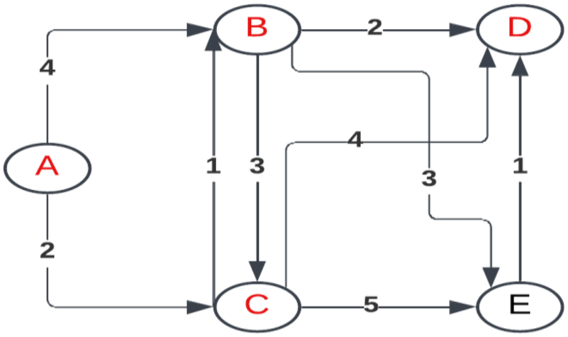
B : 3

C : 2

D : 5

E : 6

Unvisited Nodes = {A, B, C, D, E}



A : 0

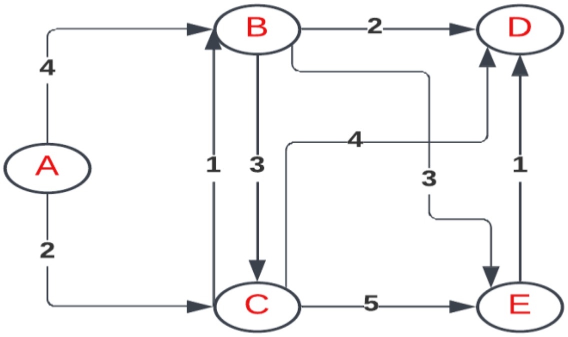
B : 3

C : 2

D : 5

E : 6

Unvisited Nodes = {A, B, C, D, E}



A : 0

B : 3

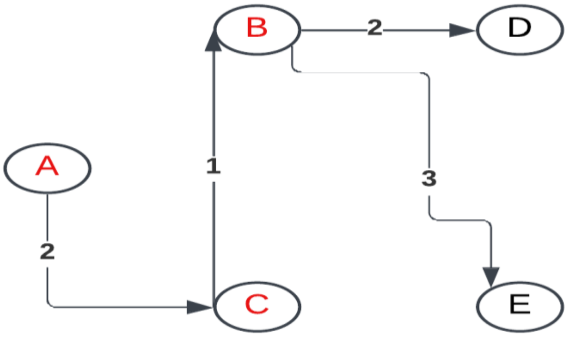
C : 2

D : 5

E : 6

Unvisited Nodes = {A, B, C, D, E}

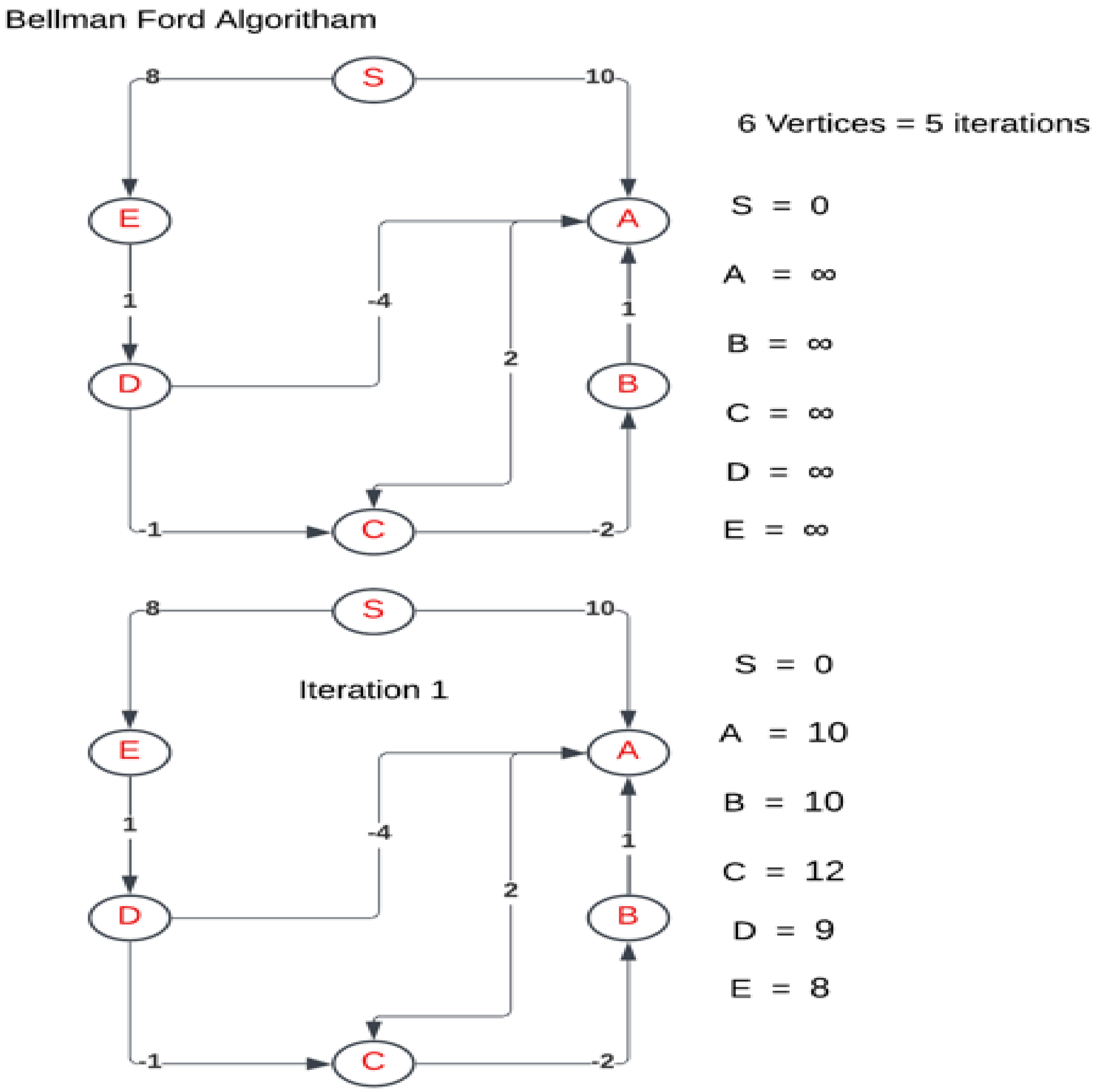
Shortest Path



**Bellman Ford Algoritham:** The Bellman-Ford algorithm is best suited to find the shortest paths in a directed graph, with one or more negative edge weights, from the source vertex to all other vertices. It does so by repeatedly checking all the edges in the graph for shorter paths, as many times as there are vertices in the graph (minus 1).

The Bellman-Ford algorithm can also be used for graphs with positive edges (both directed and undirected), like we can with Dijkstra's algorithm, but Dijkstra's algorithm is preferred in such cases because it is faster.

Using the Bellman-Ford algorithm on a graph with negative cycles will not produce a result of shortest paths because in a negative cycle we can always go one more round and get a shorter path.



6 Vertices = 5 iterations

S = 0

A = ∞

B = ∞

C = ∞

D = ∞

E = ∞

S = 0

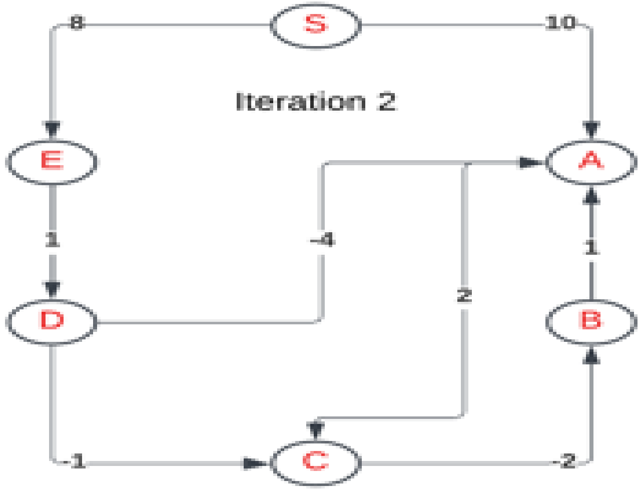
A = 10

B = 10

C = 12

D = 9

E = 8



S = 0

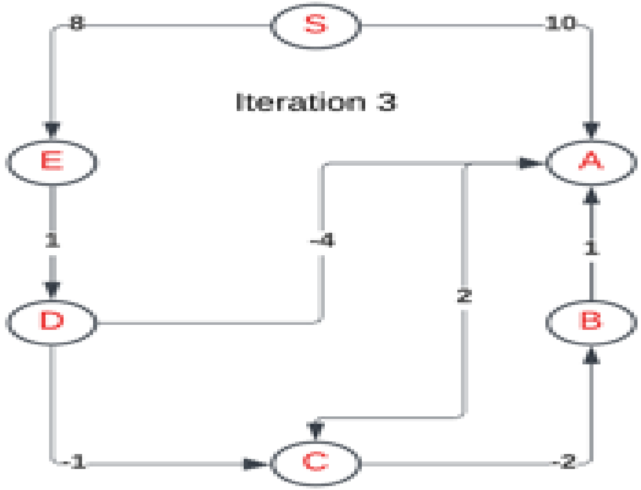
A = 5

B = 10

C = 8

D = 9

E = 8



S = 0

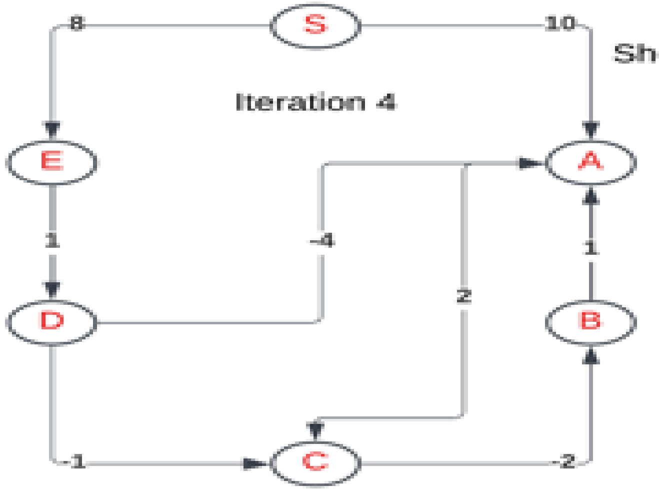
A = 5

B = 5

C = 7

D = 9

E = 8



Shortest path from S to
all other nodes

S = 0

A = 5

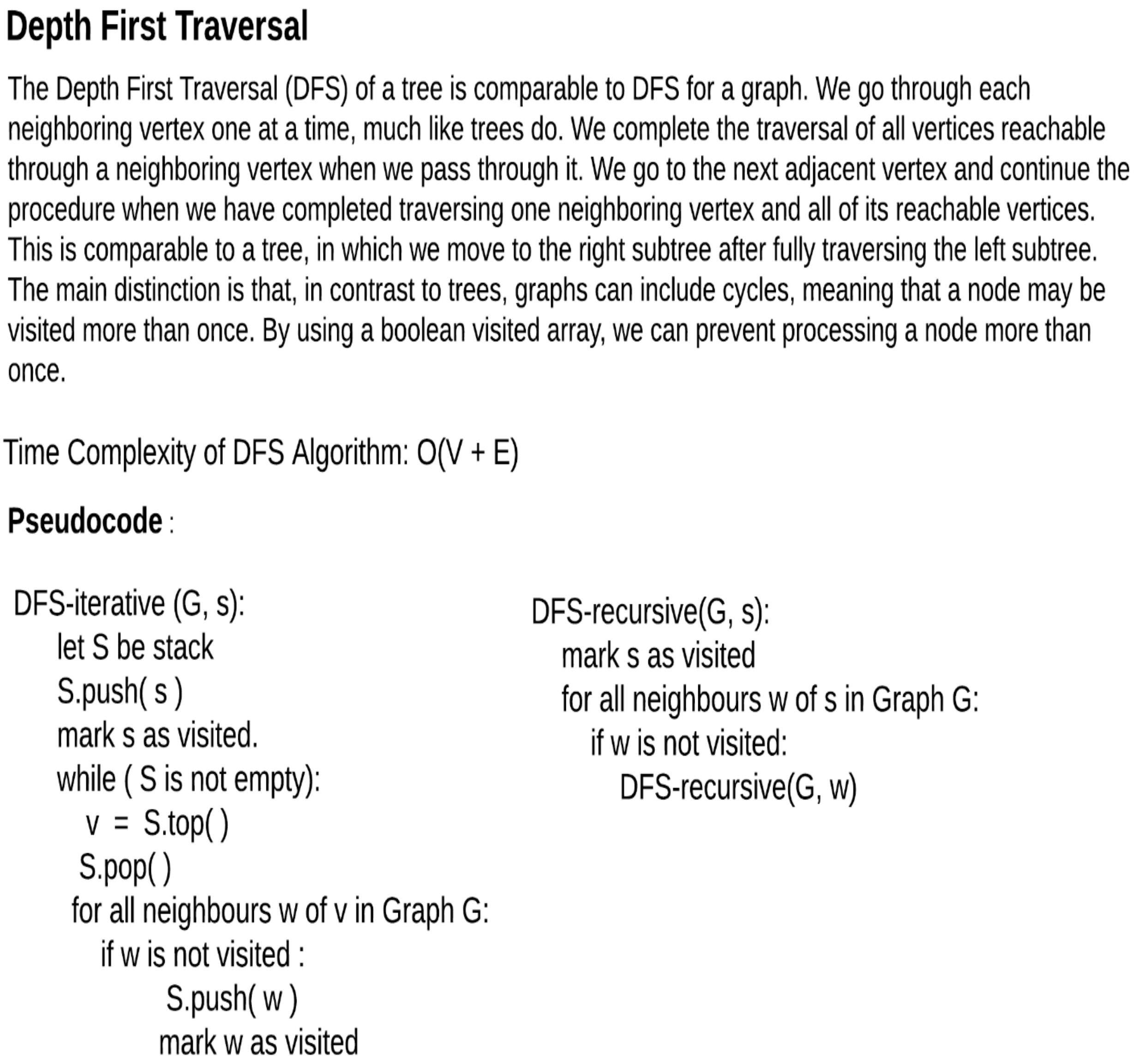
B = 10

C = 8

D = 9

E = 8

Depth First Search:



## Depth First Traversal

The Depth First Traversal (DFS) of a tree is comparable to DFS for a graph. We go through each neighboring vertex one at a time, much like trees do. We complete the traversal of all vertices reachable through a neighboring vertex when we pass through it. We go to the next adjacent vertex and continue the procedure when we have completed traversing one neighboring vertex and all of its reachable vertices. This is comparable to a tree, in which we move to the right subtree after fully traversing the left subtree. The main distinction is that, in contrast to trees, graphs can include cycles, meaning that a node may be visited more than once. By using a boolean visited array, we can prevent processing a node more than once.

Time Complexity of DFS Algorithm: O(V + E)

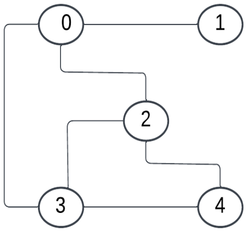
**Pseudocode** :

```
DFS-iterative (G, s):
    let S be stack
    S.push( s )
    mark s as visited.
    while ( S is not empty):
       v  =  S.top( )
      S.pop( )
     for all neighbours w of v in Graph G:
        if w is not visited :
              S.push( w )
             mark w as visited
```

```
DFS-recursive(G, s):
   mark s as visited
   for all neighbours w of s in Graph G:
      if w is not visited:
         DFS-recursive(G, w)
```

Example:

Initially stack and visited array are empty

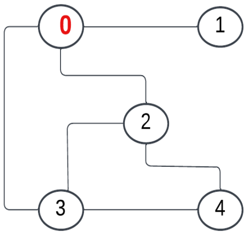


| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Visited | | | | | |
| DFS | | | | | |

| |
|---|
| |
| |
| |

empty stack

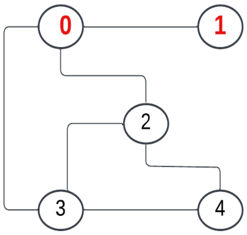
Visit 0 and put its adjacent nodes which are not visited into the stack



| | | | | | |
|---|---|---|---|---|---|
| Visited | T | T | T | T | |
| DFS | 0 | | | | |

| |
|---|
| 1 |
| 2 |
| 3 |

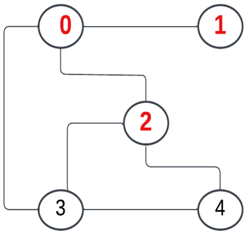
Node 1 at top of stack , so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack



| | | | | | |
|---|---|---|---|---|---|
| Visited | T | T | T | T | |
| DFS | 0 | 1 | | | |

| |
|---|
| |
| 2 |
| 3 |

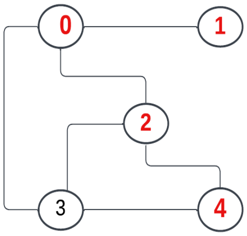
Node 2 at top of stack , so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack



| | | | | | |
|---|---|---|---|---|---|
| Visited | T | T | T | T | T |
| DFS | 0 | 1 | 2 | | |

| |
|---|
| |
| 4 |
| 3 |

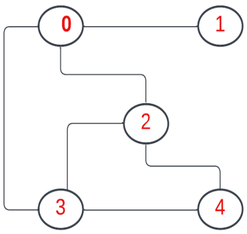
Node 4 at top of stack , so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack



| | | | | | |
|---|---|---|---|---|---|
| Visited | T | T | T | T | T |
| DFS | 0 | 1 | 2 | 4 | |

| |
|---|
| |
| |
| 3 |

Node 3 at top of stack , so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack

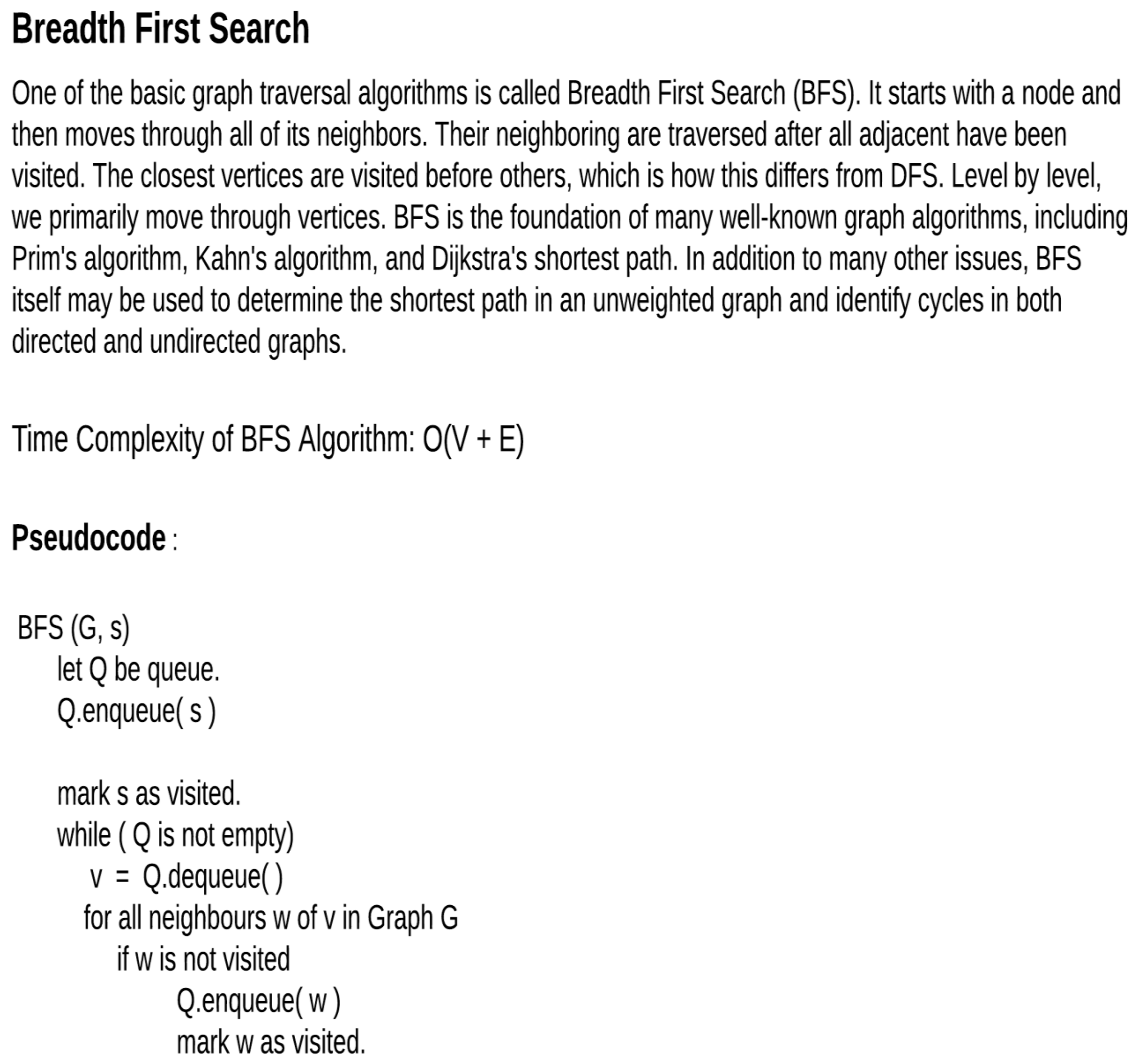


| | | | | | |
|---|---|---|---|---|---|
| Visited | T | T | T | T | T |
| DFS | 0 | 1 | 2 | 4 | 3 |

| |
|---|
| |
| |
| |

empty stack means we have visited all the nodes

Breadth First Search:



## Breadth First Search

One of the basic graph traversal algorithms is called Breadth First Search (BFS). It starts with a node and then moves through all of its neighbors. Their neighboring are traversed after all adjacent have been visited. The closest vertices are visited before others, which is how this differs from DFS. Level by level, we primarily move through vertices. BFS is the foundation of many well-known graph algorithms, including Prim's algorithm, Kahn's algorithm, and Dijkstra's shortest path. In addition to many other issues, BFS itself may be used to determine the shortest path in an unweighted graph and identify cycles in both directed and undirected graphs.

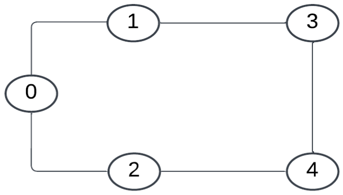
Time Complexity of BFS Algorithm: O(V + E)

**Pseudocode** :

```
BFS (G, s)
    let Q be queue.
    Q.enqueue( s )

    mark s as visited.
    while ( Q is not empty)
        v  =  Q.dequeue( )
      for all neighbours w of v in Graph G
          if w is not visited
                Q.enqueue( w )
                mark w as visited.
```

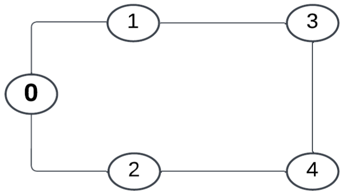
Example:

Initially queue and visited array are empty



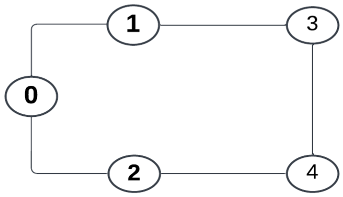
| | | | | | |
|---|---|---|---|---|---|
| Visited | | | | | |
| Queue | | | | | |
| | Front | | | | |

Push 0 into queue and mark visited



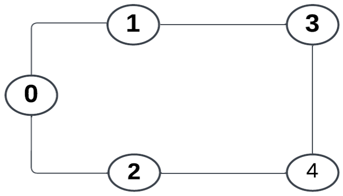
| | | | | | |
|---|---|---|---|---|---|
| Visited | 0 | | | | |
| Queue | 0 | | | | |
| | Front | | | | |

Remove 0 from the front of queue and visit the unvisited neighbour and push them into queue



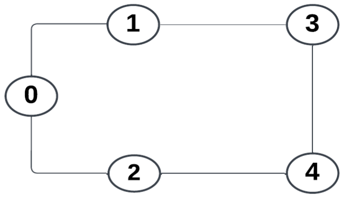
| | | | | | |
|---|---|---|---|---|---|
| Visited | 0 | 1 | 2 | | |
| Queue | 0 | 1 | 2 | | |
| | | Front | | | |

Remove node 1 from the front of queue and visit the unvisited neighbour and push them into queue



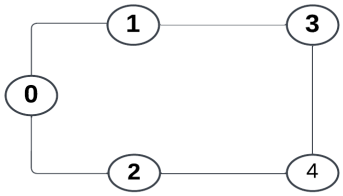
| | | | | | |
|---|---|---|---|---|---|
| Visited | 0 | 1 | 2 | 3 | |
| Queue | 1 | 2 | 3 | | |
| | | Front | | | |

Remove 2 node from the front of queue and visit the unvisited neighbour and push them into queue



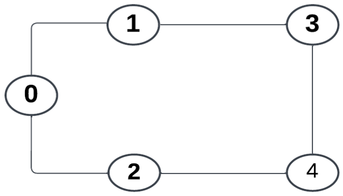
| | | | | | |
|---|---|---|---|---|---|
| Visited | 0 | 1 | 2 | 3 | 4 |
| Queue | 2 | 3 | 4 | | |
| | | Front | | | |

Remove 3 node from the front of queue and visit the unvisited neighbour and push them into queue



| | | | | | |
|---|---|---|---|---|---|
| Visited | 0 | 1 | 2 | 3 | 4 |
| Queue | 3 | 4 | | | |
| | | Front | | | |

Remove 4 node from the front of queue and visit the unvisited neighbour and push them into queue

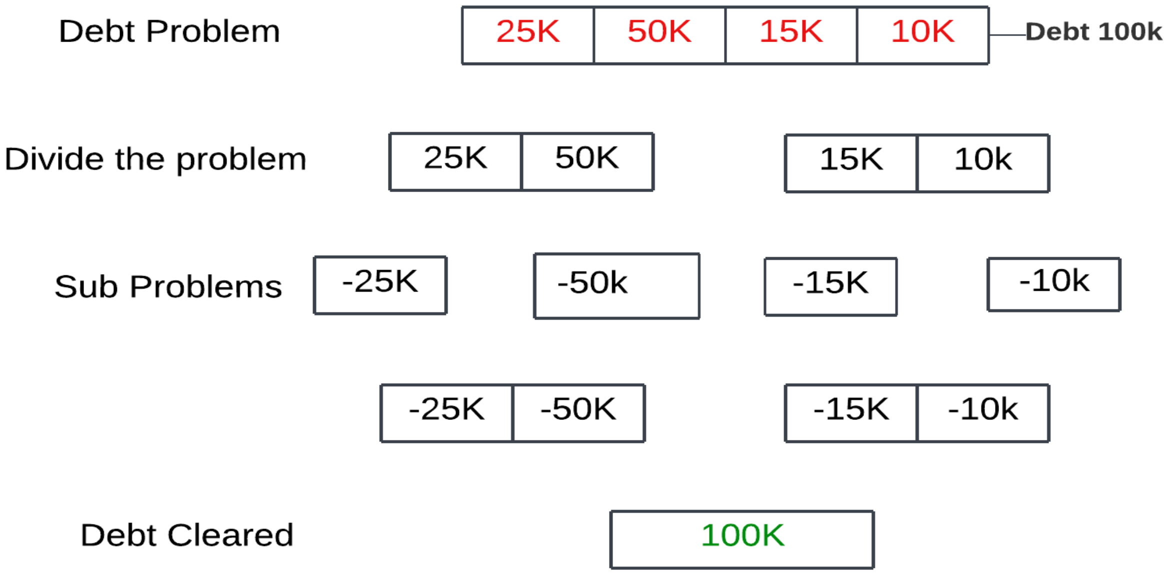


| | | | | | |
|---|---|---|---|---|---|
| Visited | 0 | 1 | 2 | 3 | 4 |
| Queue | 4 | | | | |
| | | Front | | | |

## Divide and Conquer:

Divide and Conquer:

The Divide and Conquer algorithm is a method of problem-solving that entails dividing a difficult issue into smaller, easier-to-manage components, resolving each component separately, and then integrating the answers to resolve the original issue. It is a popular algorithmic method in mathematics and computer science.



**Approach:**

Divide:

- Dissect the primary issue into more manageable subissues.
- Every subproblem ought to be a component of the main issue.
- Dividing the problem until it can no longer be divided is the aim.

Conquer:

- Address each of the more manageable subissues separately.
- We solve a subproblem directly, without additional recursion, if it is sufficiently small.
- The objective is to solve these subproblems on your own.

Merge:

- To obtain the ultimate answer to the entire problem, combine the subproblems.
- The bigger problem's solution is obtained by recursively combining the solutions of the smaller subproblems.
- The objective is to combine the findings from the subproblems to create a solution for the main issue.

## Knapsack Algorithm:

Fractional knapsack Problem:

Given a bag with a capacity of W, meaning it can store a maximum of W weight, and N objects, each of which has a weight and profit connected with it. The goal is to pack the things in the bag so that the total profit from them is as high as it can be.

Let's run our algorithm on the following data: n = 4 (number of items) M = 5 (knapsack capacity = maximum weight) (wi, pi): (2, 3), (3, 4), (4, 5), (5, 6)

| i/w | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | | | | | |
| 2 | 0 | | | | | |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |

Compute C[ 2, 5]

| i/w | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 3 | 3 | 3 | 3 |
| 2 | 0 | 0 | 3 | 4 | 4 | 7 |
| 3 | 0 | | | | | |
| 4 | 0 | | | | | |

Compute C[ 4, 5]

| i/w | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 3 | 3 | 3 | 3 |
| 2 | 0 | 0 | 3 | 4 | 4 | 7 |
| 3 | 0 | 0 | 3 | 4 | 5 | 7 |
| 4 | 0 | 0 | 3 | 4 | 5 | 7 |

To find the actual items in the knapsack

- All of the information we need is in the table.
- C[n, M] is the maximal value of items that can be placed in the Knapsack.
- Let i = n and k = M if C[i, k] 6= C[ i − 1, k] then mark the i-th item as in the knapsack i = i − 1, k = k − w i . else i = i − 1

Compute C[ 4, 5]

| i/w | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 3 | 3 | 3 | 3 |
| 2 | 0 | 0 | 3 | 4 | 4 | 7 |
| 3 | 0 | 0 | 3 | 4 | 5 | 7 |
| 4 | 0 | 0 | 3 | 4 | 5 | 7 |

| i/w | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 3 | 3 | 3 | 3 |
| 2 | 0 | 0 | 3 | 4 | 4 | 7 |
| 3 | 0 | 0 | 3 | 4 | 5 | 7 |
| 4 | 0 | 0 | 3 | 4 | 5 | 7 |

Solution : { 1, 1, 0, 0 }