Arrays

- The array is a simplest data structure.
- It is a contiguous block of memory.
- Given an array *A, A[i]* denotes the *(i*+1)th object stored in the array.
- Retrieving and updating, A[i] takes O*(1)* time.
- Insertion into a full array can be handled by resizing, i.e., allocating a new array with additional memory and copying over the entries from the original array.
    - This increases the worst-case time of insertion.
- Deleting an element from an array entails moving all successive elements one over to the left to fill the vacated space.
    - The time complexity to delete the element at index *i* from an array of length *n* is O*(n- i).*

Write a program to define an array, which take the array input from the user. Then reorder its entries so that the even entries appear first.

Approach:

- For this problem, partition the array into three subarrays: Even, Unclassified, and Odd, appearing in that order.
- Initially Even and Odd are empty, and Unclassified is the entire array.
- We iterate through Unclassified, moving its elements to the boundaries of the Even and Odd subarrays via swaps, thereby expanding Even and Odd, and shrinking Unclassified.

Programme:

```
import java.util.Scanner;

public class EvenOddProg1 {

	public static void main(String[] args) {

		int size;
		Scanner s = new Scanner(System.in);

		System.out.print("Enter the size of the array: ");
		size = s.nextInt();
		int A[] = new int[size];

		System.out.println("Enter " + size + " the elements:");
		for(int i = 0; i < size; i++) {

			A[i] = s.nextInt();
		}


		System.out.print("The array elements are ");
		for(int i = 0; i < size; i++) {

			System.out.print(A[i] + " ");
		}

		System.out.println();

		evenOdd(A);

		System.out.print("The array elements (after reorder) are ");
		for(int i = 0; i < size; i++) {

			System.out.print(A[i] + " ");
		}

	}

	public static void evenOdd(int[] A){

		int nextEven =0 , nextOdd = A.length - 1;

		while (nextEven < nextOdd) {

			if (A[nextEven] % 2 == 0) {

				nextEven++;
			}
			else {

				int temp = A[nextEven];
```

```
                        A[nextEven] = A[nextOdd];

                        A[nextOdd--] = temp;
                    }
                }       //end of while
        } //end of evenOdd()
}       //end of EvenOddProg1
```

Output:

```
Enter the size of the array: 5
Enter 5 the elements:
11
24
15
14
42
The array elements are 11 24 15 14 42
The array elements (after reorder) are 42 24 14 15 11
```

Time Complexity:

- A constant amount of processing per entry, so the time complexity is *0(n).*

Space Complexity:

- It uses *O(n)* space, where *n* is the length of the array. The additional space complexity is clearly O(1) - a couple of variables that hold indices, and a temporary variable for performing the swap.

The DUTCH NATIONAL FLAG Problem

- The Dutch national flag (DNF) problem is one of the most popular programming problems proposed by Edsger Dijkstra. The flag of the Netherlands consists of three colors: white, red, and blue.

- The Dutch national flag (DNF) is closely related to the *partition* operation of quick sort.

    - The quicksort algorithm for sorting arrays proceeds recursively - it selectsan element (the "pivot"), reorders the array to make all the elements less than or equal to the pivot appear first, followed by all the elements greater than the pivot. The two subarrays are then sorted recursively.

- In the Dutch national flag (DNF) problem, write a program that takes an array A and an index *i* into A, and rearranges the elements such that all elements less than *A[i]* (the "pivot") appear first, followed by elements equal to the pivot, followed by elements greater than the pivot.

- **Example:** Suppose A = (0, l, 2, 0, 2, l, l)
  - Let the pivot index is 3. Then A[3] = 0,
    - so (0, 0,1, 2, 2,1,1) is a valid partitioning.
  - For the same array, if the pivot index is 2, then A[2] = 2,
    - so the arrays (0, 1, 0, 1, 1, 2, 2) as well as (0, 0, 1, 1, 1, 2, 2) are valid partitionings.

Programme 1:

```
import java.util.*;

public class DutchFlagPartitionProg1 {

    public static enum Color { RED, WHITE, BLUE }

    public static void main(String[] args) {

        Color RED = Color.RED;

        Color WHITE = Color.WHITE;

        Color BLUE = Color.BLUE;

         //Creating a List
         List<Color> A = new ArrayList<Color>();

         //Adding elements in the List
         A.add(RED);
         A.add(WHITE);
         A.add(BLUE);
         A.add(WHITE);
         A.add(RED);
         A.add(BLUE);
        // A.add(RED);
        // A.add(BLUE);
        // A.add(WHITE);

         System.out.println("List Before Sorting");
         //Iterating the List element using for-each loop
```

```
             for(Color C : A)
                        System.out.print( C + " ");

             int pivotlndex = (A.size() - 1);

             dutchFlagPartition(pivotlndex , A);

             System.out.println(" ");

             System.out.println("List After Sorting");
             //Iterating the List element using for-each loop
             for(Color C : A)
                         System.out.print( C + " ");
}

public static void dutchFlagPartition(int pivotlndex , List<Color> A) {

            Color pivot = A.get(pivotlndex);

            // First pass: group elements smaller than pivot.
            for (int i = 0; i < A.size(); ++i) {

                        // Look for a smaller element.
                        for (int j = i + 1; j < A.size(); ++j){

                                    //The ordinal() method of Enum class returns
                                    //the ordinal of this enumeration constant.
                                    if (A.get(j).ordinal() < pivot.ordinal()){

                                                Collections.swap(A , i, j);
                                                break ;
                                    }
                        }
            }

            // Second pass: group elements larger than pivot.
            for (int i = A.size() - 1 ; i >= 0 && A.get(i).ordinal() >= pivot.ordinal(); --i) {

                        // Look for a larger element. Stop when we reach an element less
                        // than pivot, since first pass has moved them to the start of A.
                        for (int j = i - 1; j >= 0 && A.get(j).ordinal() >= pivot.ordinal(); --j) {

                                    if (A.get(j).ordinal() > pivot.ordinal()){

                                                Collections.swap(A , i, j);
                                                break ;
```

```
                                        }
                              }
                    }
          }
}
```

Output:

```
List Before Sorting :
RED WHITE BLUE WHITE RED BLUE

List After Sorting :
WHITE WHITE RED RED BLUE BLUE
```

Time and Space Complexity:

- The additional space complexity is now O(1), but the time complexity is O(n^2)

Programme 2:

```
import java.util.*;

public class DutchFlagPartitionProg2 {

          public static enum Color { RED, WHITE, BLUE }

          public static void main(String[] args) {

                    Color RED = Color.RED;

                    Color WHITE = Color.WHITE;

                    Color BLUE = Color.BLUE;

                     //Creating a List
                     List<Color> A = new ArrayList<Color>();

                     //Adding elements in the List
                     A.add(RED);
                     A.add(WHITE);
                     A.add(BLUE);
                     A.add(WHITE);
                     A.add(RED);
                     A.add(BLUE);
                    // A.add(RED);
                    // A.add(BLUE);
```

```
            // A.add(WHITE);

             System.out.println("List Before Sorting : ");
             //Iterating the List element using for-each loop
             for(Color C : A)
                         System.out.print( C + " ");

             int pivotIndex = (A.size() - 1);

             dutchFlagPartition(pivotIndex , A);

             System.out.println(" ");
             System.out.println(" ");

             System.out.println("List After Sorting : ");
             //Iterating the List element using for-each loop
             for(Color C : A)
                          System.out.print( C + " ");
}

public static void dutchFlagPartition(int pivotIndex, List<Color> A){

            Color pivot = A.get(pivotIndex);

            // First pass: group elements smaller than pivot.
            int smaller = 0;

            for (int i = 0; i < A.size(); ++i){

                        if (A.get(i).ordinal() < pivot.ordinal()){

                                    Collections.swap(A , smaller++, i);
                        }
            }


            // Second pass: group elements larger than pivot.
            int larger = A.size() - 1;

            for (int i = A.size() - 1; i >= 0 && A.get(i).ordinal() >= pivot.ordinal(); --i){

                        if (A.get(i).ordinal() > pivot.ordinal()){

                                   Collections.swap(A , larger--, i);
                        }
            }
```

```
        }
}
```

Output:

```
List Before Sorting :
RED WHITE BLUE WHITE RED BLUE

List After Sorting :
RED WHITE WHITE RED BLUE BLUE
```

Time and Space Complexity:

- The time complexity is O(n) and the space complexity is O(1).

Programme 3:

```
import java.util.*;

public class DutchFlagPartitionProg3 {

        public static enum Color { RED, WHITE, BLUE }

        public static void main(String[] args) {

                Color RED = Color.RED;

                Color WHITE = Color.WHITE;

                Color BLUE = Color.BLUE;

                 //Creating a List
                 List<Color> A = new ArrayList<Color>();

                 //Adding elements in the List
                 A.add(RED);
                 A.add(WHITE);
                 A.add(BLUE);
                 A.add(WHITE);
                 A.add(RED);
                 A.add(BLUE);
//               A.add(RED);
//               A.add(BLUE);
//               A.add(WHITE);

                 System.out.println("List Before Sorting");
```

```
            //Iterating the List element using for-each loop
            for(Color C : A)
                        System.out.print( C + " ");

            int pivotlndex = (A.size() - 1);

            dutchFlagPartition(pivotlndex , A);

            System.out.println(" ");

            System.out.println("List After Sorting");
            //Iterating the List element using for-each loop
            for(Color C : A)
                         System.out.print( C + " ");
}

public static void dutchFlagPartition(int pivotlndex , List<Color> A) {

           Color pivot = A.get(pivotlndex);
           /*
           * Keep the following invariants during partitioning:
           * bottom group: A .subList (SI , smaller) .
           * middle group: A .subList (smaller , equal).
           * unclassified group: A .subList (equal , larger).
           * top group: A .subList (larger , A . size ()) .
           */
           int smaller = 0, equal = 0, larger = A.size();

           // Keep iterating as long as there is an unclassified element.
           while (equal < larger) {

                      // A .get (equal) is the incoming unclassified element.
                      if (A.get(equal).ordinal() < pivot.ordinal()){

                                 Collections.swap(A , smaller++, equal++);

                      } else if (A.get(equal).ordinal() == pivot.ordinal()){

                                 ++equal ;

                      } else { // A . get (equal) > pivot.

                                 Collections.swap(A , equal, --larger);
                      }
           }
}
```

```
}
```

Output:

```
List Before Sorting :
RED WHITE BLUE WHITE RED BLUE

List After Sorting :
RED WHITE WHITE RED BLUE BLUE
```

Time and Space Complexity:

- The time spent within each iteration is O(1),implying the time complexity is O(n). The space complexity is clearly O(1).

INCREMENT AN ARBITRARY-PRECISION INTEGER

Write a program which takes as input an array of digits encoding a decimal number D and updates the array to represent the number D + 1.

Example: if the input is (1,2,9) then you should update the array to (1,3,0).

- For this, we would update 9 to 0 with a carry-out of 1.
- We update 2 to 3 (because of the carry-in).
- There is no carry-out, hence the result is (1, 3, 0).

Approach :

To add one to number to the integer digits,

- check If the last element (i.e., Least Significant Digit) is 9, then make it 0 and carry = 1 otherwise make element value + 1.
- For the next digits in the given integer traverse from last digit, check If the element is 9, then make it 0 and carry = 1 otherwise make element value + 1.
- If the MSD is 10, set current MSD as 0 and need additional digit as the MSD (i.e., increase the List size) append 1 in the beginning.

Programme :

```
import java.util.*;
```

```
public class IncreamentArbitaryPrecisionIntProg1 {

        public static void main(String[] args) {


                //Creating a List
                List<Integer> A = new ArrayList<Integer>();

                //Adding elements in the List
                A.add(9);
                A.add(9);
                A.add(9);

                System.out.print("The enter Integer Number :  ");
                //Iterating the List element using for-each loop
                for(Integer num : A) {
                        System.out.print( num + " " );
                }
                System.out.print( " " + "means ");
                for(Integer num : A) {
                        System.out.print(num);
                }

                System.out.println(" ");
                System.out.println(" ");

                plusOne(A);

                System.out.print("After Increament by One the Integer Number is ");
                //Iterating the List element using for-each loop
                for(Integer num : A)
                         System.out.print( num );
        }

        public static List<Integer> plusOne (List<Integer> A) {

                //index of last digit
                int n = A.size() - 1;

                // Add 1 to last digit. set(int index, element) method
                //replace the element at the specified position in the
                // list with the specified element.
                // index- index of the element to replace
                // element- element to be stored at the specified position
                A.set(n, A.get(n) + 1);
                //Traverse from last digit find carry and add to other digit.
                for (int i = n; i > 0 && A.get(i) == 10; --i) {

                        A.set (i , 0); //set 0 at index i

                        A.set(i - 1, A.get(i - 1) + 1); // add carry to other digit.
                }

                // if the MSD is 10, set current MSD as 0 and need additional digit as
                // the MSD (i.e., increase the List size) append 1 in the beginning.
                if (A.get (0) == 10) {

                        A.set (0 , 0);    // set current MSD as 0

                        A.add (0, 1);             //increase the List size
                                                  //append 1 in the beginning.
                }

                return A;
        }
}
```

Output:

```
The enter Integer Number :  9 9 9  means 999

After Increament by One the Integer Number is 1000
```

Time and Space Complexity:

- The time complexity is 0(n), where n is the length of A.

MULTIPLY TWO ARBITRARY-PRECISION INTEGERS

Write a program that takes two arrays representing integers, and returns an integer representing their product.

- Use arrays to represent integers, e.g., with one digit per array entry
  - the most significant digit appearing first
    - For example, (1,9,3, 7,0,7, 7, 2,1) represents 193707721
  - a negative leading digit denoting a negative integer.
    - For example, (-7,6,1,8,3,8, 2,5,7, 2,8, 7) represents -761838257287.

- if the inputs are (1,9,3, 7,0,7, 7, 2,1} and (-7,6,1,8,3,8, 2,5,7, 2,8, 7), your function should return (-1,4, 7,5,7,3, 9,5, 2,5,8, 9,6, 7,6, 4,1, 2, 9, 2,7).

Approach:

- From a space perspective, it is better to incrementally add the terms rather than compute all of them individually and then add them up.
- The number of digits required for the product is at most *n* + *m* for *n* and *m* digit operands,
    - hence use an array of size *n*+ *m* for the result.
- **Example:** when multiplying 123 with 987, (All numbers shown are represented using arrays of digits.)
    - First find, 7 X 123 = 861
    - Then compute 8 X 123 X 10 = 9840
    - Add 9840 with 861 to get 10701
    - Then compute 9 X 123 X 100 = 110700
    - Add 110700 with 10701 to get the final result 121401

Programme :

```
import java.util.*;

public class MultiplyTwoArbitraryPrecisionIntProg1 {

	public static void main(String[] args) {

	//-----Enter 1st Integer and display--------

		//Creating a List
		List<Integer> num1 = new ArrayList<Integer>();

		//Adding elements in the List
		//num1.add(1);
		num1.add(2);
		num1.add(3);

		System.out.print("The 1st Integer Number :  ");
		//Iterating the List element using for-each loop
```

```
        for(Integer num : num1) {
                System.out.print( num + " " );
        }
        System.out.print( " " + "means ");
        for(Integer num : num1) {
                System.out.print(num);
        }

        System.out.println(" ");
        System.out.println(" ");

//-----Enter 2nd Integer and display--------

        //Creating a List
        List<Integer> num2 = new ArrayList<Integer>();

        //Adding elements in the List
        //num2.add(9);
        num2.add(8);
        num2.add(7);

        System.out.print("The 2nd Integer Number :  ");
        //Iterating the List element using for-each loop
        for(Integer num : num2) {
                System.out.print( num + " " );
        }
        System.out.print( " " + "means ");
        for(Integer num : num2) {
                System.out.print(num);
        }

        //call the method multiply() and store the result
        List<Integer> result = multiply (num1, num2);

        System.out.println(" ");
        System.out.println(" ");

//----- Display Multiplication Result--------

        System.out.print("After Multiplication the result is ");
        //Iterating the List element using for-each loop
        for(Integer num : result) {
                System.out.print( num + " " );
        }
        System.out.print( " " + "means ");
        for(Integer num : result) {
                System.out.print(num);
        }
}

public static List <Integer> multiply(List <Integer> num1 , List <Integer > num2) {

        //find or decide the sign of the result and store
        final int sign = num1.get (0) < 0 ^ num2.get(0) < 0 ? -1 : 1;

        //if the num1 and num2 are -ve, then make its to +ve,
        //by using abs() of element at index 0
        num1.set (0 , Math . abs (num1 .get(0)));
        num2.set (0 , Math . abs (num2 .get(0)));

        //find the size of num1 and num2
        int size1 = num1.size();
        int size2 = num2.size();

        //Create result array with size3 and initialize the each cell with zero
        int size3 = size1 + size2;
        List<Integer> result = new ArrayList <> (Collections.nCopies(size3, 0));

        // Multiply with current digit of first number
        // and add result to previously stored product
        // at current position.
        for (int i = size1-1; i >= 0; --i) {

                for (int j = size2 - 1; j >= 0 ; --j) {

                        int k = i + j;

                        result.set (k + 1, result.get(k + 1) + num1.get(i) * num2.get(j));

                        result.set(k, result.get(k) + result.get(k + 1) / 10);

                        result.set(k + 1, result. get(k + 1) % 10);
                }
        }

        // Remove the leading zeroes.
        int first_not_zero = 0;
        while (first_not_zero < result.size() && result.get(first_not_zero) == 0) {

                ++first_not_zero ;
        }
        result = result. subList (first_not_zero, result . size()) ;
```

```
                //if the result ArrayList is empty
                if (result.isEmpty ()) {

                        return Arrays.asList (0) ;
                }

                //attach the sign of the result, which earlier find
                result.set (0, result.get(0) * sign);
                return result;
        }
}
```

Output:

```
The 1st Integer Number :  2 3  means 23

The 2nd Integer Number :  8 7  means 87

After Multiplication the result is 2 0 0 1  means 2001
```

Time and Space Complexity:

- we perform O(1) operations on each digit in each partial product. O(m * n), where m and n are length of two number that need to be multiplied.

DELETE DUPLICATES FROM A SORTED ARRAY

Given a sorted array that has some unique as well as some duplicate elements. Write a program which remove all duplicate elements from the given array and return the array that is not having any duplicate elements in it.

This problem is concerned with deleting repeated elements from a sorted array.

Since the array is sorted, repeated elements must appear one-after-another, so we do not need an auxiliary data structure to check if an element has appeared already. We move just one element, rather than an entire subarray, and ensure that we move it just once.

Example:

- Let the given array, A = [2,3,5,5,7, 11] and the size is 6.
- then after deletion of duplicate element, i.e., 5, the array A = [2, 3, 5, 7, 11, 11].
    - After deleting repeated elements, there are 5 valid entries.

    - There are no requirements as to the values stored beyond the last valid element.

- Hence, we need to display first 5 entries in array A; i.e., 2, 3, 5, 7, 11.

Approach:

- We need to modify the array in-place and the size of the final array would potentially be smaller than the size of the input array. So, we ought to use a two-pointer approach here. One, that would keep track of the current element in the original array and another one for just the unique elements.
    - Use a separate index in same array for pointing the unique elements.
- Essentially, once an element is encountered, simply need to bypass its duplicates and move on to the next unique element.
    - shift the elements in the array by comparing two adjacent elements, if they are not equal then consider one as unique and put it at the index pointer's location and we will again check the other element with its adjacent one.
- This shifting will help in only having unique elements in the array.

For the given example, (2,3,5,5,7,11), when processing the A[3], since we already have a 5 (which we know by comparing A[3] with A[2]), we advance to A[4], Since this is a new value, we move it to the first vacant entry, namely A[3]. Now the array is (2,3,5,7,7,11,11,11,13), and the first vacant entry is A[4]. We continue from A[5],

Programme :

```
import java.util.*;

public class DeleteDuplicateFromSortedArrayProg1 {

        public static void main(String[] args) {

                //Creating a List
                List<Integer> A = new ArrayList<Integer>();

                //Adding elements in the List
                A.add(2);
                A.add(3);
                A.add(5);
                A.add(5);
                A.add(7);
                A.add(11);
                System.out.print("The enter Integer Number :  ");
                //Iterating the List element using for-each loop
                for(Integer num : A) {
                        System.out.print( num + " " );
                }


                System.out.println(" ");
                System.out.println(" ");

                int writeIndex = deleteDuplicates(A);     5

                System.out.print("After Deleting the Duplicate element from Array : ");
                //Iterating the List element using for-each loop
                for(int i=0; i<writeIndex; i++)
                        System.out.print(A.get(i) + " ");
}


        //Returns the number of valid entries after deletion.
        public static int deleteDuplicates(List<Integer> A) {

                if (A.isEmpty()) {

                        return 0;
                }

                int writelndex = 1;
                for (int i = 1; i < A.size(); ++i) {

                        if (!A.get(writelndex - 1).equals(A.get (i))) {

                                A.set(writelndex++, A.get(i));

                        }
                }

                return writelndex;
        }
}
```

Output:

```
The enter Integer Number :  2 3 5 5 7 11
```

After Deleting the Duplicate element from Array : 2 3 5 7 11

Time and Space Complexity:

- The time complexity is O(n),and the space complexity is O(1), since all that is needed is the two additional variables.

ADVANCING THROUGH AN ARRAY

- In a particular board game, a player has to try to advance through a sequence of positions.
- Each position has a nonnegative integer associated with it, representing the maximum you can advance from that position in one move.
- You begin at the first position, and win by getting to the last position.

Example:

- Let *A* = < 3, 3, 1,0, 2, 0, 1 > represent the board game, i.e., the ith entry in A is the maximum we can advance from *i.*
- Then the game can be won by the following sequence of advances through *A:*
    - take 1 step from A[0] to *A[*1],
    - then 3 steps from A[l] to A[4],
    - then 2 steps from A[4] to A[6], which is the last position.

Example:

- Let *A* = < 3, 2, 0,0, 2, 0,1> represent the board game, i.e., the ith entry in A is the maximum we can advance from *i.*
- Then the game can take the following sequence of advances through *A:*
    - take 3 step from A[0] to *A[*3],
    - The A[3], which contains 0, hence it would not possible to advance past position 3, so the game cannot be won.
- Alternatively:
    - take 1 step from A[0] to *A[*1],
    - The A[1], which contains 2, which leads to index 3,
    - The A[3], which contains 0, hence it would not possible to advance past position 3, so the game cannot be won.
- Alternatively:
    - take 1 step from A[0] to *A[*1], which contains 2.
    - Then take 1 step from A[1], which contains 0, hence it would not possible to advance past position 3, so the game cannot be won.

Example, let *C* = < 2,4,1,1,0, 2,3> represent the board game, i.e., the ith entry in C is the maximum we can advance from *i.*

- Take 2 step from C[0] to *C[*2],
- *C[*2] which contains 1, which leads to index 3, after which it cannot progress.

- Alternative:
  - Take 1 step from C[0] advancing to index 1, i.e., C[2],
  - C[2], which contains a 4 lets us proceed to index, i.e., C[5]
  - By Taking 1 step from C[5] we can advance to index 6. which is the last position. The game own.

Write a program which takes an array of *n* integers, where *A[i]* denotes the maximum you can advance from index i, and returns whether it is possible to advance to the last index starting from the beginning of the array.

Approach:

- Iterating through all entries in Array.
- As iterate through the array, track the furthest index which can advance to.
- The furthest can advance from index *i* is *i* + *A[i], where i is index processed.*
- If, for some *i* before the end of the array, *i* is the furthest index that we can advance to, we cannot reach the last index. Otherwise, we reach the end.

Programme:

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

public class AdvancingThroughArrayProg1 {

	public static void main(String[] args) {

	System.out.println("CASE - 1:  ");

		//Creating a List
		List<Integer> A = new ArrayList<Integer>();

		//Adding elements in the List
		A.add(3);
		A.add(3);
		A.add(1);
		A.add(0);
		A.add(2);
		A.add(0);
```

```
        A.add(1);

        System.out.print("The Array of Integer Numbers :  ");
        //Iterating the List element using for-each loop
        for(Integer num : A) {
                System.out.print( num + " " );
        }

        System.out.println(" ");

        boolean reach1 = canReachEnd(A);

        if (reach1)

                System.out.println("possible to reach the last index");

         else

                System.out.println("not possible to reach the last index");

        System.out.println(" ");
        System.out.println(" ");

System.out.println("CASE - 2:  ");

        //Creating a List
        List<Integer> B = new ArrayList<Integer>();

        //Adding elements in the List
        B.add(3);
        B.add(2);
        B.add(0);
        B.add(0);
        B.add(2);
        B.add(0);
        B.add(1);

        System.out.print("The Array of Integer Numbers :  ");
        //Iterating the List element using for-each loop
        for(Integer num : B) {
                System.out.print( num + " " );
        }

        System.out.println(" ");

        boolean reach2 = canReachEnd(B);
```

```
            if (reach2)

                    System.out.println("possible to reach the last index");

             else

                    System.out.println("not possible to reach the last index");
    }

    public static boolean canReachEnd(List<Integer> maxAdvanceSteps) {

            int furthestReachSoFar = 0;
            int lastIndex = maxAdvanceSteps.size() - 1;

         for (int i = 0; i <= furthestReachSoFar && furthestReachSoFar < lastIndex; ++i) {

                    furthestReachSoFar = Math.max(furthestReachSoFar,
                                                                      i + maxAdvanceSteps.get(i));
            }

            if (furthestReachSoFar >= lastIndex)

                    return true;

            else

                    return false;
    }
}
```

Output:

```
CASE - 1:
The Array of Integer Numbers :  3 3 1 0 2 0 1
possible to reach the last index

 CASE - 2:
The Array of Integer Numbers :  3 2 0 0 2 0 1
not possible to reach the last index
```

Time and Space Complexity:

- The time complexity is *O(n)*, and the additional space complexity (beyond what is used for A) is three integer variables, i.e., *O(1).*

BUY AND SELL A STOCK ONCE

Write a program that takes an array denoting the daily stock price, and returns the maximum profit that could be made by buying and then selling one share of that stock.

Explanation:

- Suppose an array of integers, i.e., prices = [310, 315, 275, 295, 260, 270, 290, 230, 255, 250]
    - The items (array elements), i.e., prices[i] correspond to stock prices of a given stock on the i+1th day.
    - The indexes of the array, i.e., i correspond to sequential days.
- You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.
    - calculates the maximum profit from this transaction (by buying once and selling once).
    - If you cannot achieve any profit, return 0.
- You can only sell a stock on a day after you bought it.
    - You can’t sell before you buy
    - You can’t sell on the same day that you buy

Example:

- In array, prices = [310, 315, 275, 295, 260, 270, 290, 230, 255, 250]
- The maximum profit that can be made with one buy and one sell is 30
    - Buy on day 5 (index, i = 4) at price 260
    - Sell on day 7 (index, i = 6) at price 290
    - Hence, profit = 290 - 260 = 30.
- **Note:**
    - 260 is not the lowest price, nor 290 the highest price.

- buying on day 5 (at price 260) and selling on day 2 (at price 315) is not allowed because you must buy before you sell.

Approach:

- This approach assumes that each item (array elements) present in the given array, i.e., prices[i], is the potential selling price for the stock.
- Iterate through array element, keeping track of the minimum element *m* seen thus far.
- If the difference of the current element and *m* is greater than the maximum profit recorded so far, update the maximum profit.

Programme:

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

public class BuySellStockOnce {

        public static void main(String[] args) {

                //Creating a List
                List<Double> Prices = new ArrayList<Double>();

                //Adding elements in the List
                Prices.add(310.0);
                Prices.add(315.0);
                Prices.add(275.0);
                Prices.add(295.0);
                Prices.add(260.0);
                Prices.add(270.0);
                Prices.add(290.0);
                Prices.add(230.0);
                Prices.add(255.0);
                Prices.add(250.0);


                System.out.print("The Stock Prices :  ");
                //Iterating the List element using for-each loop
                for(Double sprice: Prices) {
                        System.out.print( sprice + " " );
                }

                                System.out.println(" ");

                Double maxProfit  = computeMaxProfit(Prices);
```

```
            System.out.print( "Profit : " + " " + maxProfit);
        }


        public static double computeMaxProfit(List<Double> prices){

            double minPrice = Double.MAX_VALUE; //(somewhere around, 1.7*10^308)
            double maxProfit = 0.0;

            //Iterating the List element using for-each loop
            for (Double price : prices) {

                //find the profit
                maxProfit = Math.max(maxProfit , price - minPrice);

                //decide buying price of share
                minPrice = Math.min(minPrice , price);
            }

            return maxProfit;
        }
}
```

Output:

```
The Stock Prices :  310.0 315.0 275.0 295.0 260.0 270.0 290.0 230.0 255.0 250.0
Profit :  30.0
```

Time and Space Complexity:

- This algorithm performs a constant amount of work per array element, leading to an *0(n)* time complexity. It uses two float-valued variables (the minimum element and the maximum profit recorded so far) and an iterator, i.e., ***0(1)*** additional space.

BUY AND SELL A STOCK TWICE

Write a program that takes an array denoting the daily stock price, then computes the maximum profit that can be made by buying and selling a share at most twice. The second buy must be made on another date after the first sale.

Example:

- In array, prices = [30, 30, 50, 10, 10, 30, 11, 40]
    - For 1st Buy

- Buy on day 4 (index, i = 3) at price 10
- Sell on day 6 (index, i = 5) at price 30
- Hence, profit = 30 - 10 = 20.

- For 2nd Buy
    - Buy on day 7 (index, i = 6) at price 11
    - Sell on day 8 (index, i = 7) at price 40
    - Hence, profit = 40 - 11 = 29.
- Therefore, Total Profit = 20 + 29 = 49

Programme:

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

public class BuySellStockTwiceProg1 {

        public static void main(String[] args) {

                List <Double> A =new ArrayList<Double>();

                Scanner sc = new Scanner(System.in);
                System.out.print("Enter the size of the array list :  ");
                int n = sc.nextInt();

                System.out.println("Enter the " + n + "Integers");
                for(int i =0; i<n; i++) {
                        A.add(sc.nextDouble());
                }

                System.out.print("The Stock Prices :  ");
                //Iterating the List element using for-each loop
                for(Double num : A) {
                        System.out.print( num + " " );
                }

                System.out.println(" ");

                double maxTotalProfit = buyAndSellStockTwice(A);
```

```
            System.out.print( "Profit : " + " " + maxTotalProfit);

        }

        public static double buyAndSellStockTwice(List<Double> prices) {


            double maxTotalProfit = 0.0;

            //Create an ArrayList to store first Buy and Sale Profit
            List<Double> firstBuySellProfits = new ArrayList<>();

            double minPriceSoFar = Double.MAX_VALUE;

            // Forward phase. For each day, we record maximum profit if we
            // sell on that day.
            for (int i = 0; i < prices.size(); ++i) {

                minPriceSoFar = Math.min(minPriceSoFar , prices.get(i));

                maxTotalProfit = Math.max(maxTotalProfit , prices.get(i) - minPriceSoFar);

                firstBuySellProfits.add(maxTotalProfit);
            }

            // Backward phase. For each day, find the maximum profit if we make
            // the second buy on that day.
            double maxPriceSoFar = Double.MIN_VALUE ;

            for (int i = prices.size() - 1; i > 0; --i) {

                 maxPriceSoFar = Math.max(maxPriceSoFar , prices.get(i));

                maxTotalProfit = Math.max( maxTotalProfit ,
                        maxPriceSoFar - prices.get(i) + firstBuySellProfits.get(i - 1));
            }

            return maxTotalProfit;
        }
}
```

Output:

```
Enter the size of the array list :  9
Enter the 9Integers
12
```

```
11
13
9
12
8
14
13
15
The Stock Prices :  12.0 11.0 13.0 9.0 12.0 8.0 14.0 13.0 15.0
Profit :  10.0
```

Time and Space Complexity:

- The time complexity is *0(n)*, and the additional space complexity is *O(n)*, which is the space used to store the best solutions for the subarrays.

ENUMERATE ALL PRIMES TO *n*

A natural number is called a prime if it is bigger than1and has no divisors other than 1and itself. In otherwords, we can say a number is prime if it has only two factors i.e 1 and the number itself.

Example: 2, 3, 5 are prime numbers.

Note: The first prime number is 2.

Write a program that takes an integer argument and returns all the primes between1 and that integer.

Example:

- If the input is 10
- The program should return 2, 3, 5, 7.

1. **Brute Force Approach.**

In brute force approach, we will simply run a loop from 2 to N and for each number in this range(let say X), we will check whether it is prime or not which will take √X steps for each X in range(2,N). The complexity of the algorithm comes out to be O(N*√N).

2. Sieve of Eratosthenes

- As the word ‘Sieve’ means filtering out dirt elements.
- It was first discovered by a Greek mathematician.

- This approach use a Boolean array to encode the candidates.
- if the ith entry in the array is true, then i is potentially a prime.
- Initially, every number greater than or equal to 2 is a candidate.
- This approach filter out the composite numbers leaving the prime numbers.
- The steps of the algorithm is as follows:
    - Create an Boolean array A of size N+1, for consecutive integers from 0 to N.
    - Initialize all the values in array A as true; otherthan indices 0 and 1
        - Entries 0 and 1 are false, since 0 and 1 are not primes.
    - Take p=2, i.e., if A[p] = true and make all the multiples of 2 in array (indices which are multiples of 2) as false.
    - Increment p and if A[p] = true, then mark all multiples of p in array as false.
    - Repeat this until $p<\sqrt{N}$
    - The elements left unmarked (i.e., true) after the above steps in the array are the prime numbers.

Example: Let say n = 10, it means to find out all the primes less than 10.

- The candidate array with size 11, is initialized to [F, F, T, T, T, T, T, T, T, T, T].
    - Where *T* is true and F is false. Entries 0 and 1 are false, since 0 and 1 are not primes.
- We begin with index 2. Since the corresponding entry is True, we add 2 to the list of primes, and sieve out its multiples.
    - The array is now [F, F, T, T, F, T, F, T, F, T, F].
- The next nonzero entry is 3, so we add it to the list of primes, and sieve out its multiples.
    - The array is now [F, F, T, T, F, T, F, T, F, F, F].
- The next nonzero entry are 5 and 7, and neither of them can be used to sieve out more entries.

Programme:

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

public class GeneratePrimeNumbersProg1 {

        public static void main(String[] args) {

                System.out.print("Enter an Integer Number : ");
                Scanner sc = new Scanner(System.in);
                int n = sc.nextInt();

                //Create ArrayList to store all primes up to and including n.
                List <Integer> Primes = new ArrayList<Integer>();

                //return all primes up to and including n.
                Primes = generatePrimes(n);

                // Print all prime numbers
                System.out.print("Prime numbers less than " + n + " are ");
                for(int pnum : Primes) {

                        System.out.print(pnum + " ");
                }
        }

        // Given n, return all primes up to and including n.
        public static List<Integer> generatePrimes(int n){

                //Create ArrayList to store all primes up to and including n.
                List<Integer> Primes = new ArrayList<>();

                // Create a boolean ArrayList, i.e., isPrime and
                 // initialize all entries set as true, except 0 and 1.
                List<Boolean> isPrime = new ArrayList<>(Collections.nCopies(n + 1, true));
                isPrime.set(0, false);
                isPrime.set(1, false);

                for (int p = 2; p <= n; ++p) {

                        //isPrime.get(p) represents if p is prime or not.
                        if(isPrime.get(p)){ // p is true

                                Primes.add(p); //Add p to Primes ArrayList
```

```
                                        // Sieve out p's multiples.
                                        for (int j = p; j <= n; j += p){

                                                isPrime.set(j, false);
                                        }
                              }
                    }

                    return Primes;
          }
}
```

Output:

```
Enter an Integer Number : 10
Prime numbers less than 10 are 2 3 5 7
```

Time and Space Complexity:

- The time complexity is O(n*log(log(n))) and The space complexity is dominated by the storage for P, i.e., *0(n).*

PERMUTE THE ELEMENTS OF AN ARRAY

- A permutation is a rearrangement of members of a sequence into a new sequence.
- A permutation of a set is a rearrangement of its elements.
- A set which consists of *n* elements has *n!* permutations.
- Here *n!* is the factorial, which is the product of all positive integers smaller or equal to *n*.

Example:

- Let us now consider the [a,b,c,d].
- the total number of permutations of all four letters are 4! = 4 * 3 * 2 * 1 = 24;
- some of these are [b,a,d,c], [d,a,b,c], and [a,d,b,c].

Given an array *A* of *n* elements and a permutation P, apply P to *A*.

Example:

- *Let given array is A = [a, b, c, d] and the* permutation can be specified by an array P = [2, 0, 1, 3]
    - where P[i] represents the location of the element at *i* in the permutation.
- Apply P = [2, 0, 1, 3] to an array *A = [a, b, c, d], it means*
    - move the element at index 0 in array A *(i.e., a)* to index 2.
    - move the element at index 1 in array A *(i.e., b)* to index 0.
    - move the element at index 2 in array A *(i.e., c)* to index 1.
    - keep the element at index 3 in array A (i.e., d) unchanged.
    - Now all elements have been moved according to the permutation, and the result is [b, c, a, d]

Programme:

```java
import java.util.*;

public class PermuteElementsOfArrayProg1 {

	public static void main(String[] args) {

		//Creating a ArrayList A and Add elements
		List<Character> A = new ArrayList<Character>();
		A.add('a');
		A.add('b');
		A.add('c');
		A.add('d');

		System.out.print("The Given Character Array A :  ");
		//Iterating the List element using for-each loop
		for(Character ch : A) {
			System.out.print( ch + " " );
		}

		System.out.println(" ");

		//Creating a ArrayList P, which is and Add elements
		//where P[i] represents the location of the element
		//at i in the permutation.
```

```
        List <Integer> P = new ArrayList<Integer>();
        P.add(2);
        P.add(0);
        P.add(1);
        P.add(3);

        System.out.print("The Permutation Array P :  ");
        //Iterating the List element using for-each loop
        for(Integer num : P) {
                System.out.print( num + " " );
        }

        System.out.println(" ");

        applyPermutation(P, A);

        System.out.print("After applying P to A, the result : ");
        //Iterating the List element using for-each loop
        for(Character ch : A) {
                System.out.print( ch + " " );
        }
}

public static void applyPermutation(List<Integer> P, List<Character> A){

        for (int i = 0; i < A.size(); ++i){

                // Check if the element at index i has not been moved by checking if
                // perm. get (i) is nonnegative .
                int next = i;

                while (P.get(next) >= 0){

                        Collections.swap(A , i, P.get(next));

                        int temp = P.get(next);

                        // Subtracts perm.size() from an entry in P to make it negative ,
                        // which indicates the corresponding move has been performed .
                        P.set(next , P.get(next) - P.size());
                        next = temp;
                }
        }

        // Restore ArrayList Elements in P
        for (int i = 0; i < P.size(); i++) {
```

```
                    P.set(i, P.get(i) + P.size());
                }
        }
}
```

Output:

```
The Given Character Array A :  a b c d
The Permutation Array P :  2 0 1 3
After applying P to A, the result : b c a d
```

Time and Space Complexity:

- The program above will apply the permutation in *0(n)* time. The space complexity is *0(1),*assumingwecan temporarily modify the sign bit from entries in the permutation array.

COMPUTE THE NEXT PERMUTATION

- There exist exactly *n* permutations of *n* elements. These can be totally ordered using the *dictionary ordering.*

Write a program that takes as input a permutation, and returns the next permutation under dictionary ordering. If the permutation is the last permutation, return the empty array.

- The general algorithm for computing the next permutation is as follows:
  - Find *k* such that *p[k]* < *p[k* + 1] and entries after index *k* appear in decreasing order.
  - Find the smallest *p[l]* such that *p[l]* > *p[k]* (such an / must exist since *p[k] <p[k+l]).*
  - Swap *p[l]* and *p[k]* (note that the sequence after position *k* remains in decreasing order).
  - Reverse the sequence after position *k.*

Programme:

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

```
public class ComputeNextPermutationProg1 {

public static void main(String[] args) {

                List <Integer> perm = new ArrayList<Integer>();

                Scanner sc = new Scanner(System.in);
                System.out.print("Enter the Number of Elements in Permutation Array :  ");
                int n = sc.nextInt();

                System.out.println("Enter the " + n + " Integers : ");
                for(int i =0; i<n; i++) {
                        perm.add(sc.nextInt());
                }

                System.out.print("The Entered Permutation Array :  ");
                //Iterating the List element using for-each loop
                for(Integer num : perm) {
                        System.out.print( num + " " );
                }

                System.out.println(" ");

                nextPermutation(perm);

                System.out.print("The Next Permutation Array :  ");
                //Iterating the List element using for-each loop
                for(Integer num : perm) {
                        System.out.print( num + " " );
                }
        }

        public static List<Integer> nextPermutation(List<Integer> perm) {

                int k = perm.size() - 2;

                while (k >= 0 && perm.get(k) >= perm.get(k + 1)) {

                        --k;
                }

                if (k == -1) {

                        return Collections.emptyList(); // perm is the last permutation.
                }
                // Swap the smallest entry after index k that is greater than perm[k] . We
                // exploit the fact that perm .subList (k + 1, perm.sizeO) is decreasing so
                // if we search in reverse order, the first entry that is greater than
                // perm[k ] is the smallest such entry.
                for (int i = perm.size() - 1; i > k; --i) {

                        if (perm.get(i) > perm.get(k)) {

                            Collections.swap(perm, k, i);
                            break ;
                        }
                }

                //Since perm . subList[k + 1, perm.size()) is in decreasing order, we can
                //build the smallest dictionary ordering of this subarray by reversing it.
                Collections . reverse (perm . subList (k + 1, perm . size ())) ;
                return perm;
        }
}
```

Output:

```
Enter the Number of Elements in Permutation Array :  7
Enter the 7 Integers :
6
2
1
5
4
3
0
The Entered Permutation Array :  6 2 1 5 4 3 0
The Next Permutation Array :  6 2 3 0 1 4 5
```

Time and Space Complexity:

- Each step is an iteration through an array, so the time complexity is *O(n).* All that we use are a few local variables, so the additional space complexity is *O(1).*

SAMPLE OFFLINE DATA

- A sample data set contains a part, or a subset, of a whole data set.

- The size of a sample is always less than the size of the whole data set from which it is taken.

- Let A be an array of n distinct elements.
- Design an algorithm that returns a subset of k elements of A.
- All subsets should be equally likely.

Implement an algorithm that takes as input an array of distinct elements and a size, and returns a subset of the given size of the array elements. All subsets should be equally likely. Return the result in input array itself.

Approach:

- Let A be an array of n distinct elements and we need to select k elements.
- In first iteration, generate one random number between 0 to A.length; let the random number is r, then swap A[0] with *A[r],* The entry A[0] now partn of the result.
- In second iteration, generate one random number between 1 to A.length; let the random number is s. then swap A[1] with *A[s],* The entry A[1] now part of the result.
- Repeat the above procedure k times. Eventually, the random subset occupies the slots A[0 : *k* - 1] and the remaining elements are in the last *n*-*k* slots.
- Return the result (i.e, A[0 : *k* - 1]) in the same input array .

Example:

- let the input is *A* = *[*3, 7,5,11] and the size be 3 (the sample size is 3).
- In the first iteration,
    - use the random number generator to pick a random integer in the interval [0,3],
    - Let the returned random number be 2.
    - swap A[0] with *A[*2], now the array is [5,7,3,11].
- In the Second iteration
    - use the random number generator to pick a random integer in the interval [1,3].
    - Let the returned random number be 3.
    - swap A[l] with A[3], now the resulting array is [5,11,3,7].
- In the Third iteration
    - use the random number generator to pick a random integer in the interval [2,3].
    - Let the returned random number be 2.
    - When swap A[2] with itself the resulting array is unchanged.
- The random subset consists of the first three entries, i.e., [5,11,3].

Programme:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;
import java.util.Scanner;

public class SampleOfflineDataProg1 {

public static void main(String[] args) {

                List <Integer> A = new ArrayList<Integer>();

                Scanner sc = new Scanner(System.in);
                System.out.print("Enter the Number of Elements in Array :  ");
                int n = sc.nextInt();

                System.out.println("Enter the " + n + " Integers : ");
                for(int i =0; i<n; i++) {
                        A.add(sc.nextInt());
                }

                System.out.print("The Entered Array Elements :  ");
                //Iterating the List element using for-each loop
                for(Integer num : A) {
                        System.out.print( num + " " );
                }

                System.out.println(" ");

                System.out.print("Enter the size of  Sample Array :  ");
                int k = sc.nextInt();

                randomsampling(k, A);

                System.out.print("The Sample Array :  ");
                //Iterating the List element
                for(int i = 0; i < k; i++) {
                        System.out.print( A.get(i) + " " );
                }
        }

        public static void randomsampling(int k, List<Integer> A) {
                Random gen = new Random ();

                for (int i = 0; i < k; ++i) {
                        // Generate a random int in [i, A.size() - 1].
                        Collections . swap (A , i, i + gen.nextInt(A.size () - i));
                }
        }
}
```

Output:

```
Enter the Number of Elements in Array :  4
Enter the 4 Integers :
3
7
5
11
The Entered Array Elements :  3 7 5 11
Enter the size of  Sample Array :  3
The Sample Array Elements :  5 3 7
```

Time and Space Complexity:

- The algorithm clearly runs in additional *0(1)* space. The time complexity is *0(k)* to select the elements.

SAMPLE ONLINE DATA

- Sample Online Data or Reservoir sampling is a family of randomized algorithms.
- Here, randomly choosing *k* samples from a list of *n* items, where *n* is either a very large or unknown number.
- Typically *n* is large enough that the list doesn't fit into main memory.

Design a program that takes as input a size *k*,and reads packets, continuously maintaining a uniform random subset of size *k* of the read packets.

Algorithm:

- Create an array *runningSample[0..k-1]* and copy first *k* items of sequence*[]* to it.
- Now one by one consider all items from *(k+1)*th item to *n*th item.
    - Generate a random number from 0 to *i*.

        - where *i* is the index of the current item in *stream[]*.
    - Let the generated random number is *j*.
    - If *j* is in range 0 to *k-1*, replace *reservoir[j]* with stream*[i]*
    - Repeat the above procedure when i = n.
- Return the result array, i.e., *runningSample[0..k-1]*

Programme:

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Random;
import java.util.Scanner;

public class SampleOnlineDataProg1 {

	public static void main(String[] args) {

		List <Integer> A = new ArrayList<Integer>();

		Scanner sc = new Scanner(System.in);
		System.out.print("Enter the Number of Elements in Array :  ");
		int n = sc.nextInt();

		System.out.println("Enter the " + n + " Integers : ");
		for(int i =0; i<n; i++) {
			A.add(sc.nextInt());
		}

		System.out.print("The Entered Array Elements :  ");
		//Iterating the List element using for-each loop
		for(Integer num : A) {
			System.out.print( num + " " );
		}

		System.out.println(" ");

		System.out.print("Enter the size of  Running Sample Array :  ");
		int k = sc.nextInt();

		// Create an iterator for the list
		// using iterator() method
```

```
        Iterator  B = A.iterator();

        List<Integer> runningSample = new ArrayList<>(k);
        runningSample = onlineRandomSample(B, k);

        System.out.print("The Running Sample Array Elements :  ");
         for(int i = 0; i < k; i++) {
                System.out.print( runningSample.get(i) + " " );
        }
}

//Assumption: there are at least k elements in the stream.
public static List<Integer> onlineRandomSample(Iterator<Integer> sequence, int k) {

        List<Integer> runningSample = new ArrayList<>(k);

        //Stores the first k elements.
        for (int i = 0; sequence.hasNext() && i < k ; ++i) {

                runningSample.add(sequence.next());
        }

        //Have read the first k elements.
        int numSeenSoFar = k;

        Random randldxGen = new Random();

        while (sequence.hasNext()){

                Integer x = sequence.next();

                ++numSeenSoFar ;

                //Generate a random number in [0, numSeenSoFar], and if this number
                //is in [0, k - 1], we replace that element from the sample with x.
                final int idxToReplace = randldxGen.nextInt(numSeenSoFar);

                if (idxToReplace < k) {

                        runningSample.set(idxToReplace, x);

                }
        }

        return runningSample;
}
```

```
}
```

Output:

```
Enter the Number of Elements in Array :  7
Enter the 7 Integers :
5
3
7
2
8
6
1
The Entered Array Elements :  5 3 7 2 8 6 1
Enter the size of  Sample Array :  3
The Sample Array Elements :  2 3 8
```

Time and Space Complexity:

- The time complexity is proportional to the number of elements in the stream, since we spend *0(1)* time per element, hence the Time complexity is O(n). The space complexity is *0(k).*

COMPUTE A RANDOM PERMUTATION

- A random permutation is a random ordering of a set of objects, that is, a permutation-valued random variable.

- A good example of a random permutation is the shuffling of a deck of cards: this is ideally a random permutation of the 52 cards.

- The use of random permutations is often fundamental to fields that use randomized algorithms such as coding theory, cryptography, and simulation.

Design an algorithm that creates uniformly random permutations of {0, 1, …, N-1}. Youaregiven a random numbergenerator that returnsintegersin theset {0, 1, …, N-1} with equal probability; use as few calls to it as possible.

Example:

- Create an array of N elements and initialize the elements.
  - Let *n* = 4 and A = [0, 1, 2, 3].
  - To selct the each element randomly, we need to perform N iteration. Here N = 4.

- In the 1st Iteration
  - The first random number is chosen between 0 and 3, inclusive.

- Suppose it is1.
- We update the array by swapping A[0] with A[1]. Hence the Updated array is A[1,0, 2,3].

- In the 2nd Iteration
  - The second random number is chosen between 1 and 3, inclusive.
  - Suppose it is 3.
  - We update the array by swapping A[1] with A[3]. Hence the Updated array is A[1, 3, 2, 0].

- In the 3rd Iteration
  - The second random number is chosen between 2 and 3, inclusive.
  - Suppose it is 3.
  - We update the array by swapping A[2] with A[3]. Hence the Updated array is A[1, 3, 0, 2].

- Hence the result is A[1, 3, 0, 2].

Programme:

```
package ComputeRandomPermutaion;

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

public class ComputeRandomPermutaionProg1 {

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);
		System.out.print("Enter the Number of Elements in Array :  ");
		int n = sc.nextInt();

		List <Integer> permutation = new ArrayList<>();
		permutation = computeRandomPermutation(n);

		System.out.print("The Random Permutation Array :  ");
		//Iterating the List element using for-each loop
		for(Integer num : permutation) {
					System.out.print( num + " " );

		}
	}
	public static List<Integer> computeRandomPermutation(int n) {
```

```
                //Create permutation[] with size n
                List <Integer> permutation = new ArrayList<>(n) ;

                //Adding the n elements in permutation[]
                for (int i = 0; i < n; ++i) {

                        permutation . add(i) ;
                }

                //Calling randomSampling(), which defined OfflineSampling class
                OfflineSampling.randomSampling(permutation.size(), permutation);

                return permutation;
        }
}


package ComputeRandomPermutaion;

import java.util.Collections;
import java.util.List;
import java.util.Random;

public class OfflineSampling {

        static void randomSampling(int k, List<Integer> A) {

                //Create object of Random Class
                Random gen = new Random ();

                //
                for (int i = 0; i < k; ++i) {

                        // Generate a random int in [0, A.size() - 1].
                        int randNo = gen.nextInt(A.size () - i);

                        //i + randNo which make the random number between
                        //i to A.size() - 1, then swap A[i] with A[i+randNo]
                        Collections . swap (A, i, i + randNo);
                }
        }
}
```

Output:

```
Enter the Number of Elements in Array :  4
```

The Random Permutation Array : 1 3 0 2

Time and Space Complexity:

- The time complexity is *0(n),* and, as an added bonus, no storage outside of that needed for the permutation array itself is needed.

COMPUTE A RANDOM SUBSET

Write a program that takes as input a positive integer *n* and a size *k* < *n*,and returns a size-*k* subset of {0,1, 2,...,*n*-1}. The subset should be represented as an array. All subsets should be equally likely and, in addition, all permutations of elements of the array should be equally likely. You may assume you have a function which takes as input a nonnegative integer *t* and returns an integer in the set {0, 1, . . . , *t* – 1} with uniform probability.

Example:

- Create a HashMap H, which is empty. In HasMap H whose keys and values are from (0,1, …, n-1).
  Let n be the total number of elements in array and k is the subset size. Here k < = n.
  Here, n =5 and k = 3.

- To selct one subset randomly from the k subset, we need to perform k iteration.

- In the 1st Iteration
  - The first random number is chosen between 0 and n-1, inclusive.
  - Suppose it is 4.
  - We update *H* to (0, 4), (4, 0).
    - This means that H[0] is 4 and H[4] is 0
  - *Hence, the H is*
    key: 0 value: 4
    key: 4 value: 0

- In the 2nd Iteration
  - The second random number is chosen between 1 and 4, inclusive.
  - Suppose it is also 4.
  - We update *H* to (1, 4), (4, 1).
    - This means that H[1] is 4 and H[4] is 1
  - *Hence, the H is*
    key: 0 value: 4
    key: 1 value: 0 //(1, 4) and (4, 0) => (1, 0)
    key: 4 value: 1

- In the 3rd Iteration
  - The second random number is chosen between 2 and 4, inclusive.
  - Suppose it is 3.

- We update *H* to (2, 3), (3, 2).
  - This means that H[2] is 3 and H[3] is 2
- *Hence, the H is*

  key: 0 value: 4
  key: 1 value: 0
  key: 2 value: 3
  key: 3 value: 2
  key: 4 value: 1

- The random subset is the 3 elements corresponding to indices 0,1, 2, i.e., [4, 0, 3].

Programme:

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.Scanner;

public class ComputeRandomSubsetProg1 {

        public static void main(String[] args) {

                Scanner sc = new Scanner(System.in);
                System.out.print("Enter the Number of Elements in Array :  ");
                int n = sc.nextInt();

                System.out.print("Enter the Number of Elements in Sub-Set Array :  ");
                int k = sc.nextInt();

                List<Integer> result = new ArrayList<>(k);

                result = randomSubset(n, k);

                System.out.print("The Subset is " );
                for (int i = 0; i < k; ++i) {

                        System.out.print(result.get(i) + " ");
                }

        }

        // Returns a random k-sized subset of {Q, 1, n - 1}.
        public static List<Integer> randomSubset(int n, int k) {
```

```
//Create the HashMap
Map<Integer , Integer> changedElements = new HashMap<>();

//Create object of Random Class
Random randldxGen = new Random();

for (int i = 0; i < k; ++i) {

        // Generate random number between i to n - 1.
        int randldx = i + randldxGen.nextInt(n-i);

        Integer ptrl = changedElements.get(randldx);
        Integer ptr2 = changedElements.get(i);

        if (ptrl == null && ptr2 == null) {

                changedElements.put(randldx , i);
                changedElements.put(i, randldx);

        } else if (ptrl == null && ptr2 != null) {

                changedElements.put(randldx , ptr2);
                changedElements.put(i, randldx);

        } else if (ptrl != null && ptr2 == null) {

                changedElements.put(i , ptrl);
                changedElements.put(randldx , i);

        } else {

                changedElements.put(i , ptrl);
                changedElements.put(randldx , ptr2);
        }

}


List<Integer> result = new ArrayList<>(k);

for (int i = 0; i < n; ++i) {

        result.add(changedElements.get(i));
}

return result ;
```

```
        }
}
```

Output:

```
Enter the Number of Elements in Array :  5
Enter the Number of Elements in Sub-Set Array :  3
The Subset is 4 0 3
```

Time and Space Complexity:

- The time complexity is O(k), since we perform a bounded number of operations per iteration. The space complexity is also O(k), since H and the result array never contain more than k entries.

GENERATE NONUNIFORM RANDOM NUMBERS

If numbers are 3, 5, 7, 11, and the probabilities are 9/18, 6/18, 2/18, 1/18, then in 1000000 calls to the program, 3 should appear 500000 times, 5 should appear roughly 333333 times, 7 should appear roughly 111111 times and 11 should appear roughly 55555 times.

You are given *n* numbers as well as probabilities p_o, p_1, … p_n , which sum up to 1. Given a random number generator that produces values in [0,1] uniformly, how would you generate one of the *n* numbers according to the specified probabilities?

Approach:

- You are given *n* numbers as well as probabilities p_o, p_1, ... p_n , which sum up to 1.
- For the case where the probabilities are not same, the problem can be solved by partitioning the interval [0, 1] into n disjoint segments.
    - So that the length of the jth interval is proportional to Pj.
- To create these intervals is to use p_0, p_0+p_1, $p_0+p_1+p_2$, … , $p_0+p_1+p_2+ … +P_{n-1}$ as the endpoints.
    - Here the interval array < p0, p0+p1, p0+p1+p2, … , p0+p1+p2+ … +Pn-1 > is sorted.
- To select a number uniformly, generate a random number between 0 to 1.
- Find the index of the interval that randomly generated number falls in.
    - use the searching technique, i.e., the interval array is sorted, hence use Binary search.

- Return the number corresponding to the interval the randomly generated number falls in.

Example:

- Let *n* numbers as well as probabilities $p_o, p_1, \ldots p_n$ are given, *the sum of the probabilities is* up to 1.
- If numbers are 3, 5, 7, 11, and the probabilities are 9/18, 6/18, 2/18, 1/18
  - values = [3, 5, 7, 11] and probabilities=[0.5, 0.333, 0.111, 0.055]
- To create these intervals is to use p0, p0+p1, p0+p1+p2, … , p0+p1+p2+ … +Pn-1 as the endpoints. Here the interval array < p0, p0+p1, p0+p1+p2, … , p0+p1+p2+ … +Pn-1 > is sorted.
  - intervalarray = [0.0, 0.5, 0.833, 0.944, 0.999], which is a sorted array.
    - the four intervals are [0.0, 5.0], [0.5, 0.0.8333), [0.833, 0.944), [0.944, 1.0].
- To select a number uniformly, generate a random number between 0 to 1.
  - if the generated random number is 0.873.
- Use the searching technique, i.e., binary search to find the index of the interval that randomly generated number falls in.
  - science o.873 lies in [0.833, 0.944), which is the 3rd interval (i.e., index 2) in intervalarray = [0.0, 0.5, 0.833, 0.944, 0.999]
- Return the number corresponding to the interval the randomly generated number falls in.
  - return the third number (i.e., elemet at index 2) in values = [3, 5, 7, 11], which is 7.

Programme:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;
import java.util.Scanner;

public class GenerateNonUniformRandomNumbers {
```

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


        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the size of the array list :  ");
        int n = sc.nextInt();

        List<Integer> values = new ArrayList<>();
        System.out.println("Enter the " + n + " Integer Numbers : ");
        for(int i =0; i<n; i++)
                values.add(sc.nextInt());

        List<Double> probabilities = new ArrayList<>();
        System.out.println("Enter the " + n + " Probabilities (between 0 to 1) : ");
        for(int i =0; i<n; i++)
                probabilities.add(sc.nextDouble());

        int Number = nonUniformRandomNumberGeneration(values, probabilities);
        System.out.println("The non uniform generated random number is " + Number);

}

public static int nonUniformRandomNumberGeneration(List<Integer> values,
                                                                List<Double> probabilities){

        List<Double> prefixSumofprobabilities = new ArrayList<>();
        prefixSumofprobabilities.add(0.0);

        //Creating end points for the intervals corresponding to the probabilities
        for(double p : probabilities) {

                prefixSumofprobabilities.add(
                    prefixSumofprobabilities.get(prefixSumofprobabilities.size() - 1) + p);

        }

        //create random class object
        Random r = new Random();

        //get a random number between 0.0 to 1.0
        final double uniform01 = r.nextDouble();


        //find the index of the interval that uniform01 lies in.
        int it = Collections.binarySearch(prefixSumofprobabilities, uniform01);
```

```
                if(it < 0) {

                        final int intervalldx = (Math . abs ( it) - 1) - 1;

                        return values.get ( intervalldx) ;

                } else {

                        return values.get(it) ;
                }
        }
}
```

Output:

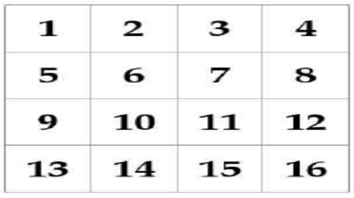
```
Enter the size of the array list :  4
Enter the 4 Integer Numbers :
3
5
7
11
Enter the 4 Probabilities (between 0 to 1) :
0.5
0.333
0.111
0.055
The non uniform generated random number is 3
```

Time and Space Complexity:

- The time complexity to compute a single value is O(*n*),which is the time to create the array of intervals. This array also implies an O(*n*) space complexity. Once the array is constructed, computing each additional result entails one call to the uniform random number generator, followed by a binary search, i.e., O*(log n).*

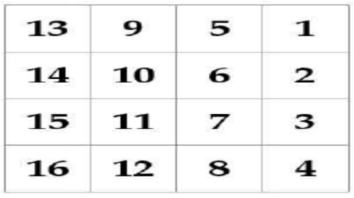
ROTATE A 2D ARRAY

Write a function that takes as input an *n* X *n* 2D array, and rotates the array by 90 degrees clockwise.



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

(a) Initial 4 × 4 2D array.



13	9	5	1
14	10	6	2
15	11	7	3
16	12	8	4

(b) Array rotated by 90 degrees clockwise.

Brute-force approach

- Let the Original array is A

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

- Allocate a new *n* X *n* 2D array, i.e., B; here n = 4.

- writing rows of the original matrix into the columns of the new matrix B.

13	9	5	1
14	10	6	2
15	11	7	3
16	12	8	4

- then copying the new array B to the original array A
  - since the problem says to update the original array.

- The time and additional space complexity are both $O(n^2)$.

Alternate Approach:

- It perform the rotation in a layer-by-layer fashion
  - different layers can be processed independently

- Furthermore, within a layer, we can exchange groups of four elements at a time to perform the rotation,
  - Example:
    - In Layer 1:
      - send 1 to 4's location
      - send 4 to 16's location
      - send 16 to 13's location
      - send 13 to 1's location
    - In Layer 2:
      - send 2 to 8's location,
      - send 8 to 15's location
      - send 15 to 9's location

- send 9 to 2's location

- Repeat the procedure for Other 2 layers.

Programme:

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

public class Rotate2DArrayProg2 {

        public static void main(String[] args) {

                 /*Declaring 2D ArrayList<E>*/
                ArrayList<ArrayList<Integer>> squareMatrix =
                                                    new ArrayList<ArrayList<Integer>>();

                /*Adding values to 1st row*/
                squareMatrix.add(new ArrayList<Integer>(Arrays.asList(1, 2, 3, 4)));

                /*Adding values to 2nd row*/
                squareMatrix.add(new ArrayList<Integer>(Arrays.asList(5, 6, 7, 8)));

                /*Adding values to 3rd row*/
                squareMatrix.add(new ArrayList<Integer>(Arrays.asList(9, 10, 11, 12)));

                /*Adding values to 4th row*/
                squareMatrix.add(new ArrayList<Integer>(Arrays.asList(13, 14, 15, 16)));

                System.out.println("Contents of 2D ArrayList(Nested ArrayList):");
                //squareMatrix.size() gives count of rows
                for (int i = 0; i <squareMatrix.size(); i++) {

                        //squareMatrix.get(i).size() gives count of columns for particular row
                        for (int j = 0; j <squareMatrix.get(i).size(); j++) {

                                System.out.print(squareMatrix.get(i).get(j) + " ");
                        }

                        System.out.println();
                }

                rotateMatrix(squareMatrix);
        System.out.println("After Rotation the Contents of 2D ArrayList(Nested
                                                                  ArrayList):");

        //squareMatrix.size() gives count of rows
        for (int i = 0; i <squareMatrix.size(); i++) {

            //squareMatrix.get(i).size() gives count of columns for particular row
            for (int j = 0; j <squareMatrix.get(i).size(); j++) {

                System.out.print(squareMatrix.get(i).get(j) + " ");
            }

            System.out.println();
        }
    }

    public static void rotateMatrix(ArrayList<ArrayList<Integer>> squareMatrix) {

        //squareMatrix.size() gives count of rows
        final int matrixSize = squareMatrix.size() - 1;

        for (int i = 0; i < (squareMatrix.size()/2); ++i) {

            for (int j = i; j < matrixSize - i; ++j) {

                // Perform a 4-way exchange.
                int temp1 = squareMatrix.get(matrixSize - j).get(i);

                int temp2 = squareMatrix.get(matrixSize - i).get(matrixSize - j);

                int temp3 = squareMatrix.get(j).get(matrixSize - i);

                int temp4 = squareMatrix.get(i).get(j);

                squareMatrix.get(i).set(j, temp1);

                squareMatrix.get(matrixSize - j).set(i, temp2);

                squareMatrix.get(matrixSize - i).set(matrixSize - j, temp3);

                squareMatrix.get(j).set(matrixSize - i, temp4);
            }
        }
    }
}
```

Output:

```
Contents of 2D ArrayList(Nested ArrayList):
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
After Rotation the Contents of 2D ArrayList(Nested ArrayList):
13 9 5 1
14 10 6 2
15 11 7 3
16 12 8 4
```

Time and Space Complexity:

- The time complexity is $O(n^2)$ and the additional space complexity is $O(1)$.

COMPUTE ROWS IN PASCAL'S TRIANGLE

The Pascal's Triangle, as follows:

1. Put a 1 at the top of the triangle. Each row will have one entry more than the previous row.

2. Each row has 1 as the first and last entries.

3. In each spot of a row enter the sum of the two entries immediately above to the left and to the right in the previous row.

4. Because each row is built from the previous row by step 3, the rows are symmetric (i.e. reversing the row produces the same numbers).

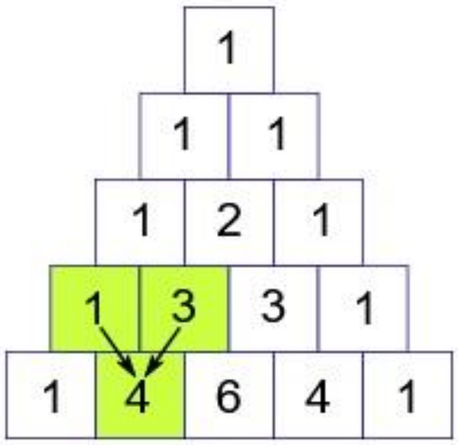


Fig: Pascal's Triangle

Write a program which takes as input a nonnegative integer *n* and returns the first *n* rows of Pascal's triangle.

Approach:

- keep the arrays left-aligned, that is the first entry is at location 0.
    - If j = 0 or *j* = *i*, the j^th^ entry in the i^th^ row is1.
    - Otherwise, it is the sum of the *(j*- l)th and jth entries in the *(i*- l)th row.

Example:

- The first row *R0* is < 1 >.
- The second row *R!* is < 1, 1 >.
- The third row *R2* is < 1, R_1[0] + R_1[1] = 2, 1 >.
- The fourth row R3 is < l, R_2[0] + R_2[l] =3, R_2[1] + R_2[2] = 3,1 >.
- The fifth row R4 is < l, R_3[0] + R_3[l] = 4, R_3[1] + R_3[2] = 6, R_3[2] + R_3[3] = 4, 1 >.

Programme

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

public class PascalTriangleProg1 {

        public static void main(String[] args) {

                Scanner sc = new Scanner(System.in);
                System.out.print("Enter the Number of rows in Pascals Triangle :  ");
                int numRows = sc.nextInt();

                /*Declaring 2D ArrayList<>*/
                List<List<Integer>> PascalTriangle = new ArrayList<List<Integer>>();

                PascalTriangle = generatePascalTriangle(numRows);

                System.out.println("The Generated Pascal's Triangle :");

                //PascalTriangle.size() gives count of rows
                for (int i = 0; i <PascalTriangle.size(); i++) {

                        //PascalTriangle.get(i).size() gives count of columns for
                        particular row
                        for (int j = 0; j <PascalTriangle .get(i).size(); j++) {

                                System.out.print(PascalTriangle .get(i).get(j) + " ");
                        }

                        System.out.println();
                }
        }

        public static List<List<Integer>> generatePascalTriangle (int numRows) {

                /*Declaring 2D ArrayList<>*/
                List<List<Integer>> pascalTriangle = new ArrayList<>();

                for (int i = 0; i < numRows; ++i) {

                        List<Integer> currRow = new ArrayList<>();

                        for (int j = 0; j <= i ; ++j){

                                if ((j > 0 && j < i))

                                        currRow.add(pascalTriangle.get(i - 1).get(j - 1)

                                                        + pascalTriangle.get(i - 1).get(j));
                                else
                                        currRow.add(1);
                        }

                        pascalTriangle.add(currRow);
                }

                return pascalTriangle ;
        }
}
```

Output:

```
Enter the Number of rows in Pascals Triangle :  5
The Generated Pascal's Triangle :
1
1 1
```

1 2 1
1 3 3 1
1 4 6 4 1

Time and Space Complexity:

- Since each element takes O*(1)* time to compute, the time complexity is $O(1+ 2+ \ldots + n) = O(n(n + l)/2) = O(n^2)$. Similarly, the space complexity is $0(n^2)$.

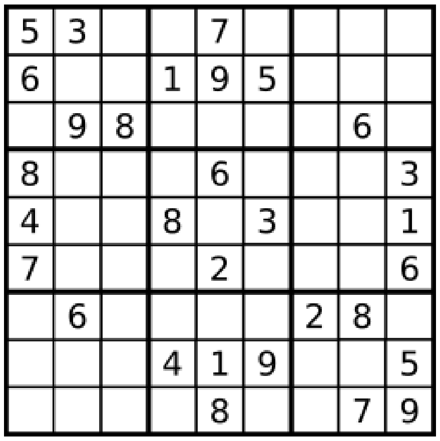
THE SUDOKU CHECKER PROBLEM

What is Sudoku?

- Sudoku is a number-placement puzzle where the objective is to fill a square grid of size 'n' with numbers between 1 to 'n'.
- The numbers must be placed so that each column, each row, and each of the sub-grids (if any) contains all of the numbers from 1 to 'n'.
- The most common Sudoku puzzles use a 9x9 grid. The grids are partially filled (with hints) to ensure a solution can be reached.

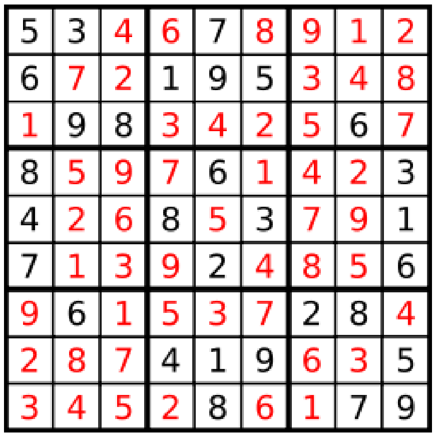
Problem Description: You are given a **Sudoku puzzle** and you need to fill the empty cells without violating any rules. A sudoku solution must satisfy all of the following rules:

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



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

Figure: A given sudoku puzzle



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

Figure: Solved sudoku puzzle. Solution numbers are marked in red.

Check whether a 9 X 9 2D array representing a partially completed Sudoku is valid.

Problem Note:

- Specifically, check that no row, column, or 3 X 3 2D subarray contains duplicates.
- A 0-value in the 2D array indicates that entry is blank.
- every other entry is in [1,9].

Algorithm

- Check if the rows and columns contain values 1-9, without repetition.
- If any row or column violates this condition, the Sudoku board is invalid.
- Check to see if each of the 9 sub-squares contains values 1-9, without repetition. If they do, the Sudoku board is valid; otherwise, it is invalid.

Programme:

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

public class SUDOKUCheckerProblemProg1 {

	public static void main(String[] args) {

		/*Declaring 2D ArrayList< >*/
		List<List<Integer>> squareMatrix = new ArrayList<List<Integer>>();

		/*Adding values to 1st row*/
		squareMatrix.add(new ArrayList<Integer>(Arrays.asList(5, 3, 0, 0, 7, 0, 0, 0, 0)));

		 /*Adding values to 2nd row*/
		squareMatrix.add(new ArrayList<Integer>(Arrays.asList(6, 0, 0, 1, 9, 5, 0, 0, 0)));

		/*Adding values to 3rd row*/
		squareMatrix.add(new ArrayList<Integer>(Arrays.asList(0, 9, 8, 0, 0, 0, 0, 6, 0)));

		 /*Adding values to 4th row*/
		squareMatrix.add(new ArrayList<Integer>(Arrays.asList(8, 0, 0, 0, 6, 0, 0, 0, 3)));
```

```
        /*Adding values to 5th row*/
        squareMatrix.add(new ArrayList<Integer>(Arrays.asList(4, 0, 0, 8, 0, 3, 0, 0, 1)));

        /*Adding values to 6th row*/
        squareMatrix.add(new ArrayList<Integer>(Arrays.asList(7, 0, 0, 0, 2, 0, 0, 0, 6)));

         /*Adding values to 7th row*/
        squareMatrix.add(new ArrayList<Integer>(Arrays.asList(0, 6, 0, 0, 0, 0, 2, 8, 0)));

        /*Adding values to 8th row*/
        squareMatrix.add(new ArrayList<Integer>(Arrays.asList(0, 0, 0, 4, 1, 9, 0, 0, 5)));

        /*Adding values to 8th row*/
        squareMatrix.add(new ArrayList<Integer>(Arrays.asList(0, 0, 0, 0, 8, 0, 0, 7, 9)));

        System.out.println("Partial Sudoku configurations : ");
        //squareMatrix.size() gives count of rows
        for (int i = 0; i <squareMatrix.size(); i++) {

                //squareMatrix.get(i).size() gives count of columns for particular row
                for (int j = 0; j <squareMatrix.get(i).size(); j++) {

                        System.out.print(squareMatrix.get(i).get(j) + " ");
                }

                System.out.println();
        }


        boolean result = isValidSudoku(squareMatrix);

        if(result){

                System.out.println("The board is valid.");
        }

        else{

                System.out.println("The board is invalid.");
        }
}

// Check if a partially filled matrix has any conflicts.
public static boolean isValidSudoku(List<List<Integer>> partialAssignment){

        //Check if there are any duplicates in given row
        for (int i = 0; i < partialAssignment.size(); ++i) {

            if (hasDuplicate(partialAssignment , i, i + 1, 0, partialAssignment.size())){

                return false;
            }
        }

        //Check if there are any duplicates in the give column
        for (int j = 0; j < partialAssignment.size(); ++j) {

            if (hasDuplicate(partialAssignment , 0, partialAssignment.size(), j, j + 1)) {

                return false;
            }
        }

        //Check if there are any duplicates in the 3*3 matrix
        int regionSize = (int)Math.sqrt(partialAssignment.size());

        for (int I = 0 ; I < regionSize; ++I) {

            for (int J = 0 ; J < regionSize; ++J) {

                if (hasDuplicate(partialAssignment , regionSize * I,
                regionSize * (I + 1), regionSize * J, regionSize * (J + 1))) {

                    return false;
                }
            }
        }

        return true ;
    }

//This function takes one set at a time and checks if there is a duplicate in it.
// First it creates a boolean array of size 9 and then starts iterating through
//all the records for every digit it checks if that digit is already set to true
//in the array if yes then that means there is a duplicate if not, it sets that
//digit to true
private static boolean hasDuplicate (List <List<Integer>> partialAssignment, int
                startRow, int endRow, int startCol, int endCol ) {

        List <Boolean> isPresent = new ArrayList<>(
        Collections . nCopies (partialAssignment . size () + 1, false));
```

```
            for (int i = startRow; i < endRow; ++i) {

                for (int j = startCol; j < endCol; ++j) {

                    if (partialAssignment.get(i).get(j) != 0 &&
                        isPresent.get(partialAssignment.get(i).get (j))) {

                        return true ;
                    }

                    isPresent.set(partialAssignment.get(i).get(j), true);
                }
            }

            return false;
        }
}
```

Output:

```
Partial Sudoku configurations :
5 3 0 0 7 0 0 0 0
6 0 0 1 9 5 0 0 0
0 9 8 0 0 0 0 6 0
8 0 0 0 6 0 0 0 3
4 0 0 8 0 3 0 0 1
7 0 0 0 2 0 0 0 6
0 6 0 0 0 0 2 8 0
0 0 0 4 1 9 0 0 5
0 0 0 0 8 0 0 7 9

The board is valid.
```

Time and Space Complexity:

The time complexity of this algorithm for an *nXn* Sudoku grid with *n X n* subgrids is O*(n2)* + $O(n^2)$ + $O(n^2/(\sqrt{n})^2 X(\sqrt{n})^2)$ = $0(n^2)$; the terms correspond to the complexity to check *n* row constraints, the *n* column constraints, and the *n* subgrid constraints, respectively. The memory usage is dominated by the bit array used to check the constraints, so the space complexity is *0(n).*

COMPUTE THE SPIRAL ORDERING OF A 2D ARRAY

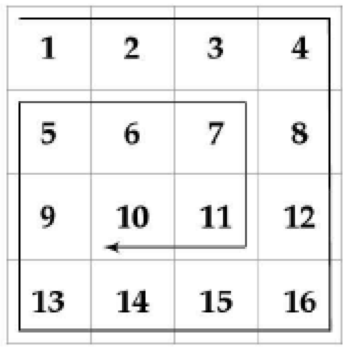
A 2D array can be written as a sequence in several orders:

- row-by-row
- column-by-column

- spiral order.

Example:

- the spiral ordering for the 2D array shown in Figure.
- the spiral ordering is <1, 2, 3, 4,8,12,16,15,14,13, 9,5, 6, 7,11,10>



Write a program which takes an *nxn* 2D array and returns the spiral ordering of the array.

Programme:

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

public class SpiralOrderingProg1 {

public static void main(String[] args) {

    /*Declaring 2D ArrayList<E>*/
    List<List<Integer>> squareMatrix = new ArrayList<List<Integer>>();

    /*Adding values to 1st row*/
    squareMatrix.add(new ArrayList<Integer>(Arrays.asList(1, 2, 3, 4)));

    /*Adding values to 2nd row*/
    squareMatrix.add(new ArrayList<Integer>(Arrays.asList(5, 6, 7, 8)));

    /*Adding values to 3rd row*/
    squareMatrix.add(new ArrayList<Integer>(Arrays.asList(9, 10, 11, 12)));

    /*Adding values to 4th row*/
    squareMatrix.add(new ArrayList<Integer>(Arrays.asList(13, 14, 15, 16)));
```

```
        //Display the Input Matrix
        System.out.println("The Input Matrix : ");
        //squareMatrix.size() gives count of rows
        for (int i = 0; i <squareMatrix.size(); i++) {

            //squareMatrix.get(i).size() gives count of columns for particular row
          for (int j = 0; j <squareMatrix.get(i).size(); j++) {

                System.out.print(squareMatrix.get(i).get(j) + " ");
          }

          System.out.println();
        }

        List<Integer> spiralOrdering = new ArrayList <>();

        spiralOrdering = matrixInSpiralOrder(squareMatrix);

        //Display the spiral ordering of elements in a matrix
        System.out.println("The spiral ordering of elements in the givrn matrix");
        for(int num : spiralOrdering) {

                    System.out.print(num + " ");
        }
    }

        public static List<Integer> matrixInSpiralOrder(List<List<Integer>> squareMatrix){

                List<Integer> spiralOrdering = new ArrayList <>();

                for (int offset = 0; offset < Math.ceil(0.5 * squareMatrix.size()); ++offset) {

                        matrixLayerlnClockwise(squareMatrix , offset, spiralOrdering);
                }

                return spiralOrdering ;
        }


        private static void matrixLayerlnClockwise(List<List<Integer>> squareMatrix,
                                                      int offset, List<Integer> spiralOrdering) {

                if (offset == squareMatrix.size() - offset - 1) {

                        // squareMatrix has odd dimension, and we are at its center.
                        spiralOrdering.add(squareMatrix.get(offset).get(offset));
                return ;
            }

            for (int j = offset; j < squareMatrix.size() - offset - 1; ++j) {

                spiralOrdering.add(squareMatrix.get(offset).get(j));
            }

            for (int i = offset; i < squareMatrix.size() - offset - 1; ++i) {

               spiralOrdering.add(squareMatrix.get(i).get(squareMatrix.size() - offset - 1));

            }

            for (int j = squareMatrix.size() - offset - 1; j > offset; --j) {

              spiralOrdering.add(squareMatrix.get(squareMatrix.size() - offset - 1).get(j));

            }

            for (int i = squareMatrix.size() - offset - 1; i > offset; --i) {

                spiralOrdering.add(squareMatrix.get(i).get(offset));

            }
        }
}
```

Output:

```
The Input Matrix :
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

The spiral ordering of elements in the givrn matrix
1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10
```

Time and Space Complexity:

The time complexity is $O(n^2)$ and the space complexity is $O(1)$.