Primitive Types

Java Bitwise and Shift Operators

In Java, bitwise operators perform operations on integer data at the individual bit-level. Here, the integer data includes byte, short, int, and long types of data.

There are 7 operators to perform bit-level operations in Java.

Operator	Meaning	Description
|	Bitwise OR	Returns 1 if at least one of the operands is 1. Otherwise, returns 0.
&	Bitwise AND	Returns 1 if and only if both the operands are 1. Otherwise, returns 0.
^	Bitwise XOR	Returns 1 if and only if one of the operands is 1. However, if both the operands are 0 or if both are 1, then the result is 0.
~	Bitwise Complement	The bitwise complement operator is a unary operator (works with only one operand). It changes binary digits 1 to 0 and 0 to 1.
<<	Left Shift	The left shift operator shifts all bits towards the left by a certain number of specified bits. As a result, the left-most bit (most-significant) is discarded and the right-most position (least-significant) remains vacant. This vacancy is filled with **0s**.
>>	Signed Right Shift	The signed right shift operator shifts all bits towards the right by a certain number of specified bits. When we shift any number to the right, the least significant bits (rightmost) are discarded and the most significant position (leftmost) is filled with the sign bit, i.e., 0 (for positive sign) or 1 (for negative sign).
>>>	Unsigned Right Shift	Java also provides an unsigned right shift. Here, the vacant leftmost position is filled with **0** instead of the sign bit.

1. Java Bitwise OR Operator (|)

The bitwise OR operator returns 1, if at least one of the operands is 1. Otherwise, it returns 0. The following truth table demonstrates the working of the bitwise OR operator. Let a and b be two operands that can only take binary values i.e. 1 or 0.

a	b	a | b	Example
0	0	0	Let's look at the bitwise OR operation of two integers 12 and 25.
0	1	1	12 = 00001100 (In Binary) 25 = 00011001 (In Binary)
1	0	1	Bitwise OR Operation of 12 and 25
1	1	1	00001100 | 00011001 _____ 00011101 = 29 (In Decimal)

Programme:

```
public class BitwiseOperator {

    public static void main(String[] args) {

        int number1 = 12, number2 = 25, result;

        // bitwise OR between 12 and 25
        result = number1 | number2;
        System.out.println(result);    // prints 29

    }
}
```

2. Java Bitwise AND Operator (&)

The bitwise AND operator returns 1 if and only if both the operands are 1. Otherwise, it returns 0. The following table demonstrates the working of the bitwise AND operator. Let a and b be two operands that can only take binary values i.e. 1 and 0.

a	b	a & b	Example
0	0	0	Let's look at the bitwise AND operation of two integers 12 and 25.
0	1	0	12 = 00001100 (In Binary) 25 = 00011001 (In Binary)
1	0	0	Bitwise AND Operation of 12 and 25
1	1	1	00001100 & 00011001 _____ 00001000 = 8 (In Decimal)

Programme:

```
public class BitwiseOperator {

        public static void main(String[] args) {

                int number1 = 12, number2 = 25, result;

                // bitwise AND between 12 and 25
                result = number1 & number2;
                System.out.println(result);    // prints 8
        }
}
```

3. Java Bitwise XOR Operator (^)

The bitwise XOR operator returns 1 if and only if one of the operands is 1. However, if both the operands are 0 or if both are 1, then the result is 0. The following truth table demonstrates the working of the bitwise XOR operator. Let a and b be two operands that can only take binary values i.e. 1 or 0.

a	b	a ^ b	Example
0	0	0	Let's look at the bitwise OR operation of two integers 12 and 25.
0	1	1	12 = 00001100 (In Binary) 25 = 00011001 (In Binary)
1	0	1	Bitwise XOR Operation of 12 and 25
1	1	0	00001100 ^ 00011001 _____ 00010101 = 21 (In Decimal)

Programme:

```
public class BitwiseOperator {

        public static void main(String[] args) {

                int number1 = 12, number2 = 25, result;

           // bitwise XOR between 12 and 25
           result = number1 ^ number2;
           System.out.println(result);    // prints 21
        }
}
```

4. Java Bitwise Complement Operator (~)

The bitwise complement operator is a unary operator (works with only one operand). It changes binary digits **1** to **0** and **0** to **1**.

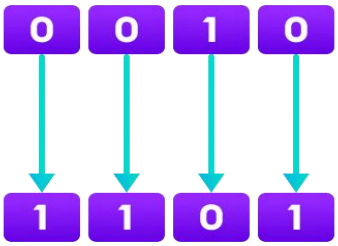


Figure 1: Java Bitwise Complement Operator

It is important to note that the bitwise complement of any integer **N** is equal to **- (N + 1)**. For example, consider an integer **35**. As per the rule, the bitwise complement of **35** should be **- (35 + 1)** = **- 36**. Now let's see if we get the correct answer or not.

```
   00100011 = 35 (In Decimal)
~_________________
   11011100 = -36 (In Decimal) / 220 (In Decimal)
```

In the above example, we get that the bitwise complement of **00100011** (**35**) is **11011100**. Here, if we convert the result into decimal we get **220**. However, it is important to note that we cannot directly convert the result into decimal and get the desired output. This is because the binary result **11011100** is also equivalent to **- 36**.

To understand this we first need to calculate the binary output of **- 36**.

2's Complement

In binary arithmetic, we can calculate the binary negative of an integer using 2's complement.

1's complement changes **0** to **1** and **1** to **0**. And, if we add **1** to the result of the 1's complement, we get the 2's complement of the original number. For example,

```
// compute the 2's complement of 36
36 = 00100100 (In Binary)

1's complement = 11011011

2's complement:

  11011011
+         1
___________
  11011100
```

Here, we can see the 2's complement of **36** (i.e. **-36**) is **11011100**. This value is equivalent to the bitwise complement of **35**. Hence, we can say that the bitwise complement of **35** is **- (35 + 1) = -36**.

Programme:

```
public class BitwiseOperator {

        public static void main(String[] args) {

                int number = 35, result;

                // bitwise complement of 35
                result = ~number;
                System.out.println(result);    // prints -36
        }
}
```

5. Java Left Shift Operator (<<)

The left shift operator shifts all bits towards the left by a certain number of specified bits. As a result, the left-most bit (most-significant) is discarded and the right-most position (least-significant) remains vacant. This vacancy is filled with **0s**.

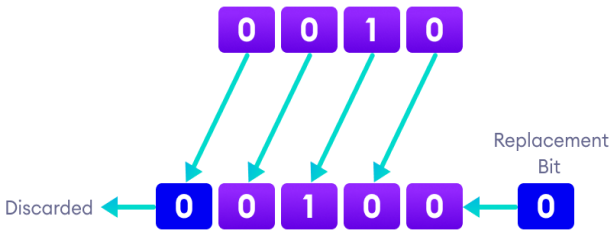


Figure 2: Java 1 bit Left Shift Operator

For example, we have a 4-digit number. When we perform a 1 bit left shift operation on it, each individual bit is shifted to the left by **1** bit. As a result, the left-most bit (most-significant) is discarded and the right-most position (least-significant) remains vacant. This vacancy is filled with **0s**.

Programme:

```
public class ShiftOperator {

        public static void main(String[] args) {

                int number = 2;
```

```
        // 2 bit left shift operation
        int result = number << 2;
        System.out.println(result);    // prints 8
    }
}
```

6. Java Signed Right Shift Operator (>>)

The signed right shift operator shifts all bits towards the right by a certain number of specified bits. When we shift any number to the right, the least significant bits (rightmost) are discarded and the most significant position (leftmost) is filled with the sign bit, i.e., 0 (for positive sign) or 1 (for negative sign).

Example: right shift of 8

In binary, the 8 is 1000.

8 >> 2 (perform 2 bit right shift)

1000 >> 2 = 0010 (equivalent to 2)

Here, we are performing the right shift of **8** (i.e. sign is positive). Hence, there no sign bit. So, the leftmost bits are filled with **0** (represents positive sign).

Example: right shift of -8

In binary, the 8 is 1000.

1's complement = 0111

2's complement:

```
 0111
  + 1
 _____
 1000
```

Signed bit = 1

// perform 2 bit right shift
8 >> 2:
1000 >> 2 = 1110 (equivalent to -2)

Here, we have used the signed bit **1** to fill the leftmost bits.

Programme:

```
public class ShiftOperator {
```

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

                int number1 = 8;
                int number2 = -8;

                // 2 bit signed right shift
                System.out.println(number1 >> 2);    // prints 2
                System.out.println(number2 >> 2);    // prints -2
        }
}
```

7. Java Unsigned Right Shift Operator (>>>)

Java also provides an unsigned right shift. Here, the vacant leftmost position is filled with **0** instead of the sign bit.

For example,

// unsigned right shift of 8
8 = 1000

8 >>> 2 = 0010

// unsigned right shift of -8

-8 >>> 2 = 1073741822

Programme:

```
public class ShiftOperator {

        public static void main(String[] args) {

           int number1 = 8;
           int number2 = -8;

           // 2 bit signed right shift
           System.out.println(number1 >>> 2);    // prints 2
           System.out.println(number2 >>> 2);    // prints 1073741822

        }
}
```

Time and Space Complexity

- Sometimes, there are more than one way to solve a problem.
- Need to compare the performance different algorithms and choose the best one to solve a particular problem.

- For analyzing an algorithm, we mostly consider
  - time complexity
  - space complexity

- Time and space complexity depends on lots of things like hardware, operating system, processors, etc. However, we don't consider any of these factors while analyzing the algorithm. We will only consider the execution time of an algorithm.

Time Complexity of an algorithm

- Time Complexity of an algorithm is the representation of the amount of time taken by the algorithm to complete the execution.

- Time requirements can be denoted or defined as a numerical function t(N), where t(N) can be measured as the number of steps, provided each step takes constant time.

Space complexity of an algorithm

- Space complexity of an algorithm represents the amount of memory space needed the algorithm in its life cycle.

- Space needed by an algorithm is equal to the sum of the following two components

  - A fixed part that is a space required to store certain data and variables (i.e. simple variables and constants, program size etc.), that are not dependent of the size of the problem.

  - A variable part is a space required by variables, whose size is totally dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation etc.

- Space complexity S(p) of any algorithm p is S(p) = A + S(I) Where A is treated as the fixed part and S(I) is treated as the variable part of the algorithm which depends on instance characteristic I.

Example:

Lets start with a simple example, addition of two n-bit integers.

SUM(P, Q)
Step 1 - START
Step 2 - R ← P + Q + 10
Step 3 – Stop

- **Time Complexity**: Addition of two n-bit integers, N steps are taken. Consequently, the total computational time is t(N) = c*n, where c is the time consumed for addition of two bits. Here, we observe that t(N) grows linearly as input size increases.

- **Space Complexity:** Here we have three variables P, Q and R and one constant. Hence S(p) = 1+3. Now space is dependent on data types of given constant types and variables and it will be multiplied accordingly.

Example:

Lets start with a simple example. Suppose you are given an array A with size N. an integer x present/absent in A and have to find if x exists in array A.

Simple solution to this problem is traverse the whole array A and check if the any element is equal to x.

```
for i : 0 to (length of A-1)
        if A[i] is equal to x
                return TRUE
return FALSE
```

- Each of the operation in computer takes approximately constant time c.

- The number of lines of code executed is actually depends on the position of value of x.
- During analyses of algorithm, mostly we will consider worst case scenario, i.e., when x is not present in the array A.

- In the worst case, the if condition will run N times where N is the length of the array A. So in the worst case, total execution time will be (N * c + c). (N * c) for the if condition and for the return statement.

- Space Complexity S(N).

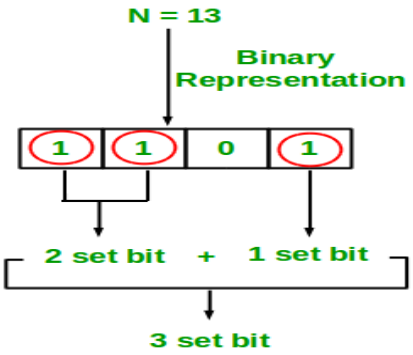
Problem: Write an efficient program to count number of 1s in the binary representation of an integer.

Let an integer number is provided. Usually the computer stores the numbers in binary number format. The binary number is formed as the combination of 0's and 1's. The digit 1 is known as set bit.

Example:

- Let the given integer is 13.

- The binary representation of 13 is 1101 and has 3 set bits or 3 number of 1's present.



Approach 1 - Simple Method: Move through each of the binary bit of an integer, check if a bit is 1, then increment the set bit counter. After passing through all the bits present in the binary number return the counter value. The following program tests one bit at-a-time starting with the least-significant bit. It illustrates shifting and masking.

Programme:

```
import java.util.Scanner;

public class CountSetBitSimple {

        public static void main(String[] args) {

                int x;
                short count;

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

                //Convert Decimal to Binary number
                System.out.println("Binary representation of the Number: " +
                                                                Long.toBinaryString(x));

                count = countBits(x);
                System.out.print("The Number of set bits in word " + x + " is " + count);

        }

        public static short countBits(int x) {

                short numBits = 0;

                while (x != 0) {

                        numBits += (x & 1) ;
                        x >>>= 1;
```

```
            }
            return numBits;
        }
}
```

Output:

Enter an Integer Number: 7
Binary representation of the Number: 111
The Parity of word 7 is 3

Time Complexity:

- Since we perform *0(1)* computation per bit, the time complexity is *0(N)*, where N is the number of bits in the integer word.

Approach 2 - Modulo Method: The idea behind this approach is similar to conversion of decimal number to binary number. Let a given number N. If N%2 ==1, means the list significant bit of that number is 1. Similarly, if N%2 ==0, means the list significant bit of that number is 0. This approach keeps track of number of 1 present in the given number.

Programme:

```
import java.util.Scanner;

public class CountSetBitModulo {

    public static void main(String[] args) {

        int x;
        short count;

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

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the Number: " +
                                                    Long.toBinaryString(x));

        count = countbitsmodulo(x);
        System.out.print("The Number of set bits in word " + x + " is " + count);

    }

    public static short countbitsmodulo(int x) {

        short numBits = 0;
```

```
            while (x != 0) {

                if(x % 2 == 1) {
                    numBits += 1;
                }

                //x = x / 2
                x >>>= 1;
            }
            return numBits;
        }
}
```

Output:

Enter an Integer Number: 6
Binary representation of the Number: 110
The Parity of word 6 is 2

Time Complexity:

- This approach always processing whether the bit is either 0 or 1. For computation of each bit require *0(1)*. Hence, the time complexity of N bits number is *0(N)*.

Approach 3 – Brian Kernighan's Algorithm: By subtracting 1 from integer, an effective observation is that all the bits after the rightmost set bit are flipped including the rightmost set bit.

Example:

$N_1 = 9$ (1001 is Binary representation of 9)
$N_2 = N_1 - 1 = 8$ (1000 is Binary representation of 8)
$N_3 = N_2 - 1 = 7$ (0111 is Binary representation of 7)

In the above example we can see that the rightmost set bit in 8 is now unset and all the bits to its right are flipped. This un-setting of the rightmost set bit is useful at the same time we need suppress the toggling effect which is unwanted. One such algorithm to achieve it is the Brian Kernighan's Algorithm.

Brian Kernighan's Algorithm:

1 Initialize count: = 0

2 If integer n is not zero

(a) Do bitwise & with (n-1) and assign the value back to n
n = n & (n-1)

(b) Increment count by 1

(c) go to step 2

3 Else return count

Programme:

```
import java.util.Scanner;

public class CountSetBitsBrianKernighan {

	public static void main(String[] args) {

		int x;
		short count;

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

		//Convert Decimal to Binary number
		System.out.println("Binary representation of the Number: " +
							Long.toBinaryString(x));

		count = countbitsBrianKernighan(x);
		System.out.print("The Number of set bits in word " + x + " is " + count);

	}

	public static short countbitsBrianKernighan(int x) {

		short numBits = 0;

		while (x != 0) {

			x &= (x - 1);
			numBits += 1;
		}
		return numBits;
	}
}
```

Output:

```
Enter an Integer Number: 9
Binary representation of the Number: 1001
The Parity of word 9 is 2
```

Time Complexity:

- This algorithm goes through as much iteration as there is set bits. Hence, the time complexity is O(k), where k is the number of set bits.

Approach 4 – Lookup Table: This approach processes bits in a group. Here we group some bits together, say 4, and store the corresponding numbers of set bits in a lookup table. For example, we have all the 4 bits integers and the corresponding set bit count store in the lookup table. Now, whenever we are given an integer we can perform constant time lookups.

Here the number of lookups for a given integer is equal to log n that is the number of binary bits in the given integer divided by k where k is the number of bits grouped together, in this case, 4. To calculate the counts of set bits, we can simply add the counts returned by the lookups.

Decimal	Binary	Count Set bits		Decimal	Binary	Count Set bits
0	0000	0		8	1000	1
1	0001	1		9	1001	2
2	0010	1		10	1010	2
3	0011	2		11	1011	3
4	0100	1		12	1100	2
5	0101	2		13	1101	3
6	0110	2		14	1110	3
7	0111	3		15	1111	4

Example:

n = 254 (Binary representation: 1111 1110)

Lower Nibble: 1110 = 14 (3 set bits)

Upper Nibble: 1111 = 15 (4 set bits)

Output = 7

Programme:

```
import java.util.Scanner;

public class CountSetBitsLookupTable {

    public static void main(String[] args) {

        int x;
        short count;

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

```
            //Convert Decimal to Binary number
            System.out.println("Binary representation of the Number: " +
                                                        Long.toBinaryString(x));

            count = countbitsLookupTable(x);
            System.out.print("The Number of set bits in word " + x + " is " + count);

    }

    public static short countbitsLookupTable(int x) {

            short numBits = 0;
             //index          0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
            int[] Table = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2,   2,  3,  2,  3,  3,  4};

            while (x != 0) {

                    numBits += Table[x & 0x0f];  //0x0f means 0000 1111
                    x >>>= 4;
            }
            return numBits;
    }
}
```

Output:

Enter an Integer Number: 234
Binary representation of the Number: 11101010
The Parity of word 234 is 5

Time Complexity:

- The time complexity of this approach is O(N/group size). Here Nis the number of bits in number and group size is the number of bits in group, say 4 here.

Problem: Write an efficient programme to compute the parity of a word.

- Parity of a number refers to whether it contains an odd or even number of 1-bits.
- The number has “odd parity”, if it contains odd number of 1-bits and is “even parity” if it contains even number of 1-bits.
- The parity of a binary word is 1 if the number of 1s in the word is odd; otherwise, it is 0.

    1 → Parity of the set bit is odd.

    0 → Parity of the set bit is even.

- Example:
    - The parity of 1011 is 1, because there are 3 ones.
    - The parity of 10001000 is 0, because there are 2 ones.
- Parity checks are used to detect single bit errors in data storage and communication.

Approach 1 - Brute Force:

- The brute-force algorithm perform bit-wise right shift on the given number iteratively & check the number of 1s seen at least significant bit (LSB).

Programme:

```
import java.util.Scanner;

public class ParityBruteForce {

	public static void main(String[] args) {

		int x;
		short count;

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

		//Convert Decimal to Binary number
		System.out.println("Binary representation of the Number: " +
                                                        Long.toBinaryString(x));

		count = parity (x);
		System.out.print("The Parity of word " + x + " is " + count);
	}

	public static short parity(long x){

		short result = 0;

		while (x != 0) {

			result ^= (x & 1);
			x >>>= 1;
		}

		return result;
	}
}
```

Output:

Enter an Integer Number: 3
Binary representation of the Number: 11
The Parity of word 3 is 0

Advantages:

- The solution is very easy to understand & implement.

Disadvantages:

- Processing all the bits manually, so this approach is hardly efficient at scale.

Time Complexity:

- O(n) where n is the total number of bits in the binary representation of the given number.

Approach 2 - Clear all the set bits one by one:

This approach can just go over only set bits. Let the given number N. In bitwise computation, this approach can clear the rightmost set bit with the following operation:

N = N & (N -1)

Example:

N =	1010	(10 in Decimal)
N – 1 =	1001	(9 in Decimal)
N & (N - 1) =	1000	(8 in Decimal)

In the above example this approach successfully cleared the lowest set bit (2nd bit from the right side in 1010). If repeat the above process, the number N will become 0 at a certain point in time. In this approach need to scan only k bits where k is the total number of set bits in the number & k <= length of the binary representation.

Programme:

```
import java.util.Scanner;

public class ClearSetBits {

    public static void main(String[] args) {

        long x;
        short count;
```

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

            //Convert Decimal to Binary number
            System.out.println("Binary representation of the Number: " +
                                                    Long.toBinaryString(x));

            count = parity(x);
            System.out.print("The Parity of word " + x + " is " + count);
        }

        public static short parity(long x){

            short result = 0;

            while (x != 0) {

                result ^= 1;
                x &= (x - 1); // Drops the lowest set bit of x.
            }
            return result;
        }
}
```

Output:

Enter an Integer Number: 10
Binary representation of the Number: 1010
The Parity of word 10 is 0

Advantages:

- Simple to implement.
- More efficient than brute force solution.

Disadvantages:

- It's not the most efficient solution.

Time Complexity:

- O(k) where k is the total number of set bits in the number. For example, for 10001010, $k = 3$.

Approach 3 - Caching:

The problem statement refers to computing the parity for a very large number of words. The size of a long type number is 64 bits or 8 bytes, so total memory size required is: 2^{64} bits of storage, which is of the order of ten trillion Exabyte.

- To compute the parity of a 64-bit integer by grouping its bits into four non overlapping 16 bit sub words.
- Compute the parity of each 16 sub words, because 2^{16} = 65536 is relatively small, which makes it feasible to cache the parity of all 16-bit words using an array.
- Then XOR parity of these four sub results with each other, since XOR is associative & commutative. The order in which we fetch those groups of bits & operate on them does not even matter.

Example: Illustrate the approach with an 8-bit word 11101010 and with a lookup table for 2-bit words.

- The cache is < 0,1,1, 0 >, these are the parities of (00), (01), (10), (11), respectively.
- To compute the parity of (11101010) we would compute the parities of (11), (10), (10), (10).
- By table lookup we see these are 0, 1, 1, 1 respectively.
- Final result is the parity of 0, 1, 1, 1 which is 0 ^ 1 ^ 1 ^ 1 = 1.

Programme:

```
import java.util.Scanner;

public class ParityCatch {

	public static void main(String[] args) {

		long x;
		short count;

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

		//Convert Decimal to Binary number
		System.out.println("Binary representation of the Number: " +
                                                        Long.toBinaryString(x));

		count = parity(x);
		System.out.print("The Parity of word " + x + " is " + count);

	}

	public static short parity(long x){
```

```
        final int W0RD_SIZE = 2;
        final int BIT_MASK = 3; //00000011 (in Binary)
                          //index  0 1  2 3
        int precomputedParity[] = {0, 1, 1, 0};

        return (short) (
              precomputedParity[(int)((x >>> (3 * W0RD_SIZE)) & BIT_MASK)]
            ^ precomputedParity[(int)((x >>> (2 * W0RD_SIZE)) & BIT_MASK)]
            ^ precomputedParity[(int)((x >>> W0RD_SIZE) & BIT_MASK)]
            ^ precomputedParity[(int)(x & BIT_MASK)]);
    }
}
```

Output:

Enter an Integer Number: 234
Binary representation of the Number: 11101010
The Parity of word 234 is 1

Advantages:

- At the cost of relatively small memory for the cache, we get better efficiency since we are reusing a group of 16-bit numbers across inputs.
- This solution can scale well as we are serving millions of numbers.

Disadvantages:

- If this algorithm needs to be implemented in an ultra-low memory device, the space complexity has to be well thought of in advance in order to decide whether it's worth it to accommodate such amount of space.

Time Complexity:

- O(N / WORD_SIZE) where N is the total number of bits in the binary representation. All right / left shift & bitwise &, |, ~ etc operations are word level operations which are done extremely efficiently by CPU. Hence their time complexity is supposed to be O(1).

Approach 4 - Using XOR & Shifting operations:

The XOR of two bits is 0, if both the bits are 0 or both bits are 1; otherwise it is 1. XOR has the property of being associative, as well as commutative, i.e., the order in which we perform the XORs does not change the result. Since parity occurs when an odd number of set bits are there in the binary representation, we can use XOR operation to check if an odd number of 1 exists there.

- For example, the parity of 64 bit numbers $< b_{63}, b_{62}, b_{61,\ldots,} b_3, b_2, b_1, b_0 >$, equals the parity of the XOR of $< b_{63}, b_{62}, b_{61,\ldots,} b_{35}, b_{34}, b_{33}, b_{32} >$ and $< b_{31}, b_{30}, b_{29,\ldots,} b_3, b_2, b_1, b_0 >$. Hence right shift the number by half of the total number of digits, XOR that shifted

number with the original number, assign the XOR-ed result to the original number. The XOR of these two 32-bit values can be computed with a single shift and a single 32-bit XOR instruction. We repeat the same operation on 32-, 16-, 8-, 4-, 2-, and 1-bit operands to get the final result. Note that the leading bits are not meaningful, and we have to explicitly extract the result from the least-significant bit. The least-significant bit extract by bitwise-AND with (00000001).

Example: Illustrate the approach with an 8-bit word 11010111

1. Split the 8–bit integer into 4–bit chunks and XOR-ed between them:
```
    0111   (considering last 4 bits of  11010111)
^   1101   (considering fast 4 bits of  11010111)
----------------------
    1010   The parity of (11010111) is the same as the parity of (1101) XORed with
           (0111)
```

2. Split the 4–bit integer into 2–bit chunks and XOR-ed between them:
```
    10   (considering last 2 bits of  1010)
^   10   (considering fast 2 bits of  1010)
----------------------
    00       The parity of (1010) is the same as the parity of (10) XORed with (10)
```

3. Split the 2–bit integer into 1–bit chunks and XOR-ed between them:
```
    0   (considering last 1 bits of  00)
^   0   (considering fast 1 bits of  00)
----------------------
    0        The parity of (00) is the same as the parity of (0) XORed with (0)
```

4. The last bit is the result and to extract it, for which bitwise AND with 1
```
    0
&   1
----------------------
    0        The even parity
```

Programme:

```
import java.util.Scanner;

public class ParityXORShifting {

    public static void main(String[] args) {

        long x;
        short count;

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

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the Number: " +
                                                    Long.toBinaryString(x));
```

```
            count = parity(x);
            System.out.print("The Parity of word " + x + " is " + count);

        }

        public static short parity(long x){

            x ^= x >>> 32;
            x ^= x >>> 16;
            x ^= x >>> 8;
            x ^= x >>> 4;
            x ^= x >>> 2;
            x ^= x >>> 1;

            return (short) (x & 0x1);       //means 1
        }
}
```

Output:

```
Enter an Integer Number: 7
Binary representation of the Number: 111
The Parity of word 7 is 1
```

Advantages:

1. No extra space uses word-level operations to compute the result.

Disadvantages:

1. Might be little difficult to understand for developers.

Time Complexity:

- O(log n) where n is the total number of bits in the binary representation.

Problem: Write a programme to Swap bits in a given integer number.

- Given a number x and two positions (indices i and j from the right side) in the binary representation of x, swaps the bits at indices i and j, and returns the result. It is also given that the two sets of bits do not overlap.

- This problem can be solved - First by checking if the two bits at given positions are the same or not.

    - If they are same (i.e., one is 0 and the other is 0) no need to be swap, because the swap does not change the integer.

- If they are not same (i.e., one is 0 and the other is 1), then we can XOR them with 1 << position. This logic will work because:

  XOR with 1 will toggle the bits
  0 ^ 1 = 1
  1 ^ 1 = 0

  XOR with 0 will have no impact
  0 ^ 0 = 0
  1 ^ 0 = 1

- A 64-bit integer can be viewed as an array of 64 bits, with the bit at index 0 corresponding to the least significant bit (LSB), and the bit at index 63 corresponding to the most significant bit (MSB).

- **Example:** bit swapping for an 8-bit integer.

  - **Input:**
    - x = 73 (In binary 01001001)
    - i = 1 (Start from the second bit from the right side)
    - j = 6 (Start from the 7th bit from the right side)

  - **Output:**
    - **11** (In binary 0001011)

  - **Explanation:**

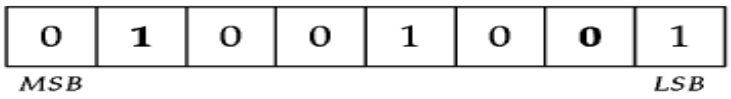


Figure Before Swapping: The 8-bit integer 73 can be viewed as array of bits, with the LSB being at index 0.

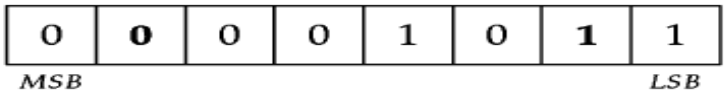


Figure After Swapping: The result of swapping the bits at indices 1 and 6, with the LSB being at index 0. The corresponding integer is 11.

Programme:

```
import java.util.Scanner;

public class SwapBitProg1 {

    public static void main(String[] args) {

        long x, y;
        int i, j;

        System.out.print("Enter an Integer Number: ");
```

```
        Scanner sc = new Scanner(System.in);
        x = sc.nextInt();

        System.out.print("Enter 1st index value i (start from the right side): ");
        i = sc.nextInt();

        System.out.print("Enter 2nd index value j (start from the right side): ");
        j = sc.nextInt();

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the Number (Before Swapping):
                                                    " + Long.toBinaryString(x));

        y = swapBits(x, i, j);
        System.out.println("After Swapping the integer is " + y);
        System.out.println("Binary representation of the Number (After Swapping): "
                                                    + Long.toBinaryString(y));
    }

    public static long swapBits(long x, int i, int j) {

        // Extract the i-th and j-th bits, and see if they differ.
        if( ((x >>> i) & 1) != ((x >>> j) & 1)){

            // We will swap them by flipping their values.
            // Select the bits to flip with bitMask.
            long bitMask = (1L << i) | (1L << j);
            x ^= bitMask;
        }
        return x;
    }
}
```

Output:

```
Enter an Integer Number: 73
Enter 1st index value i (start from the right side): 1
Enter 2nd index value j (start from the right side): 6
Binary representation of the Number (Before Swapping): 1001001
After Swapping the integer is 11
Binary representation of the Number (After Swapping): 1011
```

Time Complexity:

- The time complexity is *0(1),* independent of the word size.

Problem: Write a programme to Reverse bits in a given integer number.

- Write a program that takes a 64-bit word and returns the 64-bit word consisting of the bits of the input word in reverse order.

- For example, if the input is alternating 1s and Os, i.e., (1010...10), the output should be alternating Os and 1s, i.e., (0101...01).

- **Example:** Illustrate the approach with an 8-bit word 10010011 and with a lookup table for 2-bit words.

    - Build an array based lookup-table, rev = ((00), (10), (01), (11)).

    - Let the input is (10010011).

    - Now divide the 8-bit word into 4 groups and each group has 2-bits, such as (10), (01), (00), (11).

    - Its reverse is rev(ll), rev(00), rev(01),rev(10), i.e., (11001001).

Programme:

```
import java.util.Scanner;

public class ReverseBitsProg1 {

    public static void main(String[] args) {

        long x;
        long reverse;

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

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the Number (Before Reverse): "
                                                    + Long.toBinaryString(x));

        reverse = reverseBits(x);
        System.out.println("After Reverse the integer is " + reverse);

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the Number (After Reverse): " +
                                                    Long.toBinaryString(reverse));
    }

    public static long reverseBits(long x) {

        final int WORD_SIZE = 2;
        final int BIT_MASK = 3; //00000011 (in Binary)

                                //index   0  1  2  3
        int precomputedReverse[] = {0, 2, 1, 3};
```

```
        return (precomputedReverse [(int)(x & BIT_MASK)] << (3 * WORD_SIZE)
        | precomputedReverse [(int)((x >>> WORD_SIZE) & BIT_MASK)] << (2 * WORD_SIZE)
        | precomputedReverse[(int)((x >>> (2 * WORD_SIZE)) & BIT_MASK)] << WORD_SIZE
        | precomputedReverse[(int)((x >>> (3 * WORD_SIZE)) & BIT_MASK)]);
        }
}
```

Output:

```
Enter an Integer Number: 147
Binary representation of the Number (Before Reverse): 10010011
After Reverse the integer is 201
Binary representation of the Number (After Reverse): 11001001
```

Time Complexity:

- O(N / WORD_SIZE) where N is the total number of bits in the binary representation. All right / left shift & bitwise &, |, ~ etc operations are word level operations which are done extremely efficiently by CPU. Hence their time complexity is supposed to be O(1).

Given an integer, write a programme to find a closest integer with the same weight.

- The weight of a non-negative integer can be defined as, the number of set bits (i.e., 1's) present in the binary representation of that integer.
    - **Example:** let the given integer is 92. The binary representation of 92 is 1011100. The weight of 92 is 4, because the binary representation of 92 consist 4 numbers of set bits.
- The closest integer with the same weight can be defined as, suppose a given positive integer x and the task is to find an integer y such that:
    - Both x and y weights must be same, i.e., the number of set bits (1's) present in y is equal to the number of set bits (1's) present in x.
    - The integer y which is not equal to integer x, i.e., x != y.
    - The difference between x and y is as small as possible, i.e., |y – x| is minimum.
    - Assume x is not 0 or all 1s.

- **Example:** For integer 7 (0111 in binary), the possible closest integer numbers are 11 (1011 in binary), 13(1101 in binary) and 14 (1110 in binary), because all three have the same weight; but 11 is the closet integer to 7 because the difference between 7 and 11 is minimum.

- **Approach:**

  - The problem can be viewed as "which differing bits to swap in a bit representation of a number, so that the resultant number is closest to the original?"

  - Since the number of bits in both the numbers has to be the same, if a set bit is flipped then an unset bit will also have to be flipped.

  - Now the problem reduces to choosing two bits, say bit at index i and bit at index j for the flipping. Both index i and j are from the LSB (least significant bit) and always i > j.

  - Suppose one bit at index i is flipped and another bit at index j is flipped. To preserve the weight of an integer, the bit at index i has to be different from the bit in j, otherwise the flips lead to an integer with different weight.

  - Then the absolute value of the difference between the original integer and the new one is $2^i - 2^j$. To minimize this, i has to be as small as possible and j has to be as close to i. This means the smallest i can be is the rightmost bit that's different from the LSB, and j must be the very next bit, such that i-j=1. In summary, the correct approach is to swap the two rightmost consecutive bits that differ.

Programme:

```
import java.util.Scanner;

public class ClosestIntegerProg1 {

    public static void main(String[] args) {

        long x;
        long closeInt;

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

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the given integer: " +
                                                    Long.toBinaryString(x));

        closeInt = closestIntSameBitCount(x);
        System.out.println("The closest integer of " + x + " is " + closeInt);
```

```
            //Convert Decimal to Binary number
            System.out.println("Binary representation of the closest integer: " +
                                                    Long.toBinaryString(closeInt));

    }

    public static long closestIntSameBitCount(long x) {

            int NUM_UNSIGN_BITS = 64;
            // x is assumed to be non-negative so we know the leading bit is 0. We
            // restrict to our attention to 63 LSBs.
            for (int i = 0; i < NUM_UNSIGN_BITS - 1; ++i) {

                if ((((x >>> i) & 1) != ((x >>> (i + 1)) & 1))) {

                    x ^= (1L << i) | (1L << (i + 1)); // Swaps bit-i and bit-(i + 1).
                    return x ;
                }
            }
            // Throw error if all bits of x are 0 or 1.
            throw new IllegalArgumentException("All bits are 0 or 1");
    }
}
```

Output:

Enter an Integer Number: 11
Binary representation of the given integer: 1011
The closest integer of 11 is 13
Binary representation of the closest integer: 1101

Time Complexity:

- The time complexity is *0(n)*, where *n* is the integer width.

Write a program that multiplies two nonnegative integers without arithmetic operators.

- The only operators are allowed to use are
    - assignment,
    - the bitwise operators >>, <<, |, &, ~, ^
    - Equality checks and Boolean combinations thereof.
- The constraints imply,
    - Cannot use increment or decrement, or test if $x < y$.

- **Approach:** The algorithm uses shift and add to achieve a much better time complexity.
    - Let x and y are the two non-negative integers.
    - To multiply *x* and *y, first* initialize the result to 0.
    - Then iterate through the bits of *x,*
        - If the k^{th} bit of *x* is 1, and then add $2^k y$ to the result.
        - The value $2^k y$ can be computed by left-shifting *y* by *k.*
        - Since cannot use add operator directly, hence need to implement.
            - Perform binary number addition, i.e., compute the sum bit-by-bit, and "rippling" the carry along.

- **Example:** Multiply two non negative integers 13 and 9.

x = 13 (1101 in binary)
y = 9 (1001 in binary)

- Iterate through the bits of x (such as 1101) from least significant bit (LSB)
- In the first iteration, since the LSB (i.e., 0th bit) of 1101 is 1, set the result to 1001.

result = 1001

- In the second iteration, the second bit (i.e., 1th bit) of 1101 is 0, so no change in result.

result = 1001

- In the third iteration, the third bit (i.e., 2nd bit) is 1, hence shift 1001 to the left by 2 (i.e., 1001 << 2), obtain 100100 and it added (i.e., binary addition performed) to the result (i.e., 1001)

```
    100100          // obtain by 1001 << 2
+     1001          // result (previous),
----------------------------------
    101101          // result (current)
```

- In the fourth or final iteration (because only 4 bits present), the fourth and final bit (i.e., 3rd bit) of 1101 is 1, hence shift 1001 to the left by 3 (i.e., 1001 << 3), obtain 1001000, which is added to the result (i.e., 101101)

```
    1001000         // obtain by 1001 << 3
+    101101         // result (previous),
----------------------------------
    1110101         // result (current),
```

- The result 1110101 is equivalent to integer 117.

Programme:

```
import java.util.Scanner;

public class MultiplicationProg1 {

    public static void main(String[] args) {

        long x, y;
        long result;

        System.out.print("Enter first Integer Number: ");
        Scanner sc = new Scanner(System.in);
        x = sc.nextInt();

        System.out.print("Enter second Integer Number: ");
        y = sc.nextInt();

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the first Integer: " +
                                                    Long.toBinaryString(x));

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the second Integer: " +
                                                    Long.toBinaryString(y));

        result = multiply(x, y);
        System.out.println("The multiplication of " + x + " and " + y + " is " + result);

        //Convert Decimal to Binary number
        System.out.println("Binary representation of multiplication result: " +
                                                    Long.toBinaryString(result));

    }

    public static long multiply(long x, long y){

        long sum = 0;

        while (x != 0) {

            // Examines each bit of x.
            if ((x & 1) != 0) {

                sum = add(sum, y);
            }
```

```
            x >>>= 1;
            y <<= 1;
        }
        return sum;
    }

    private static long add(long a, long b) {

        long sum = 0, carryin = 0, k = 1, tempA = a, tempB = b;

        while (tempA != 0 || tempB != 0) {

            long ak = a & k, bk = b & k ;

            long carryout = (ak & bk) | (ak & carryin) | (bk & carryin);

            sum |= (ak ^ bk ^ carryin);

            carryin = carryout << 1;

            k <<= 1;

            tempA >>>= 1;

            tempB >>>= 1;
        }
        return sum | carryin;
    }
}
```

Output:

```
Enter second Integer Number: 13
Enter second Integer Number: 9
Binary representation of the first Integer: 1101
Binary representation of the second Integer: 1001
The multiplication of 13 and 9 is 117
Binary representation of multiplication result: 1110101
```

Time Complexity:

- The time complexity of addition is 0(n), where n is the width of the operands. Since we do n additions to perform a single multiplication, the total time complexity is $0(n^2)$.

Given two positive integers, such as x and y. Write a programme to compute x/y (their quotient), using only the addition, subtraction, and shifting operators.

Approach:

- Initially find the largest k such that $2^k y <= x$, subtract $2^k y$ from *x* and add 2^k to the quotient.
- In subsequent iterations, test $2^{k-1} y$, $2^{k-2} y$, $2^{k-3} y$... with x, and update the x and quotient.
- The process will stop, when the y > x.

Example: Divide two positive integers 11 and 2.

x = 11 (1011 in binary)
y = 2 (10 in binary)

- The large k value is determined as 2, (i.e. k=2), because $2^2 \times 2 < 11$ and $2^3 \times 2 > 11$. [$\because 2^k y <= x$]
- Then subtract $2^k y$ from x, i.e., $2^2 \times 2 = 8$ (in binary 1000) from 11 (in binary 1011).

```
      1011        // x = 11 (in integer)
  -   1000        // 2^k y=2^2 x 2 = 8
-----------------------------------
      0011        // updating x to 3 (in binary 0011)
```

- Add, $2^k = 2^2 = 4$ (in binary 100) to the quotient, initially the quotient was zero.

```
      0000        // initially the quotient is zero
  +   0100        // 2^k = 2^2 = 4 (in binary 100)
-----------------------------------
      0100        // updating quotient to 4 (in binary 100)
```

- Continue with x = 3 (in binary 0011) and determined the largest k such that $2^k y < 3$ (in binary 0011), hence k=0, show that, $2^\circ \times 2 = 2$ (in binary 0010). Then subtract $2^k y$ from x, i.e., $2^\circ \times 2 = 2$ (in binary 0010) from 3 (in binary 0011).

```
      0011        // x = 3(in integer)
  -   0010        // 2^k y=2^0 x 2 = 2
-----------------------------------
      0001        // updating x to 1 (in binary 0001)
```

- Add, $2^k = 2^0 = 1$ (in binary 0001) to the quotient

```
      0100        // quotient
  +   0001        // 2^k = 2^0 = 1 (in binary 0001)
```

```
---------------------------------
        0101        // updating quotient to 5 (in binary 101)
```

- Now the *y >x, hence stop processing and the* quotient is 5 (in binary101).

Programme:

```
import java.util.Scanner;

public class DivisionProg1 {

public static void main(String[] args) {

        long x, y;
        long quotient;

        System.out.print("Enter first Integer (Dividend): ");
        Scanner sc = new Scanner(System.in);
        x = sc.nextInt();

        System.out.print("Enter second Integer (Divisor): ");
        y = sc.nextInt();

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the first Integer (Dividend): " +
                                                        Long.toBinaryString(x));

        //Convert Decimal to Binary number
        System.out.println("Binary representation of the second Integer (Divisor): " +
                                                        Long.toBinaryString(y));

        quotient = divide(x, y);
        System.out.println("The Division of " + x + " and " + y + " is " + quotient);

        //Convert Decimal to Binary number
        System.out.println("Binary representation of quotient: " +
                                                    Long.toBinaryString(quotient));
    }


    public static long divide(long x, long y) {

        long result = 0;
        int k = 4; //32

        long yPower = y << k ;

        while (x >= y) {

            while (yPower > x) {
```

```
                    yPower >>>= 1;
                    --k ;
                }
                result += 1L << k;
                x -= yPower;
            }
            return result;
        }
}
```

Output:

Enter first Integer (Dividend): 11
Enter second Integer (Divisor): 2
Binary representation of the first Integer (Dividend): 1011
Binary representation of the second Integer (Divisor): 10
The Division of 11 and 2 is 5
Binary representation of quotient: 101

Time Complexity:

- If it takes n bits to represent x/y, there are 0(n) iterations. If the largest k such that $2^k y <= x$ is computed by iterating through k, each iteration has time complexity O(n). This leads to $O(n^2)$ algorithm.

Write a program that takes a double x and an integer y and returns x^y.

Approach:

- To develop an algorithm that works for general *y*, it is instructive to look at the binary representation of *y*, as well as properties of exponentiation, specifically $x^{(y_0 + y_1)} = x^{y_0} * x^{y_1}$

- **Example:** Suppose, y = 5 (101 in binary), then $\mathbf{x^y}$ can be determined as

$$\boldsymbol{x^{(101)_2} = x^{(101)_2 + (1)_2} = x^{(100)_2} \times x = x^{(10)_2} \times x^{(10)_2} \times x}$$

- Assume *y* is nonnegative:
    - If the least significant bit (LSB) of *y* is 0, the result is $(x^{y/2})^2$
    - If the least significant bit (LSB) of *y* is 1, the result is $x * (x^{y/2})^2$
- When *y* is negative, the only changes
    - Replacing *x by 1/x* and *y* by *-y*.

Programme:

```
import java.util.Scanner;

public class PowerProg1 {

        public static void main(String[] args) {

                double x;
                int y;
                double result;

                System.out.print("Enter first Double Number: ");
                Scanner sc = new Scanner(System.in);
                x = sc.nextDouble();

                System.out.print("Enter second Integer Number: ");
                y = sc.nextInt();

                //Convert Decimal to Binary number
                System.out.println("Binary representation of the first Integer: " +
                                                        Long.toBinaryString((long)x));

                //Convert Decimal to Binary number
                System.out.println("Binary representation of the second Integer: " +
                                                                Long.toBinaryString(y));

                result = power(x, y);
                System.out.println("The " + x + "to the power " + y + " is " + result);

                //Convert Decimal to Binary number
                System.out.println("Binary representation of the " + x + " to the power " + y +
                                                        " is " + Long.toBinaryString((long)result));
        }


public static double power(double x, int y){

                double result = 1.0;
                long power = y;

                //check power is positive
                //or negative
                if (y < 0) {
                        power = -power;
                        x = 1.0 / x;
                }
                while (power != 0) {
```

```
                    // If power is odd, multiply
                    // x with result
                    if ((power & 1) != 0) {
                            result *= x;
                    }

                    x *= x ;                // Change x to x^2
                    power >>>= 1;           //power must be even now
            }

            return result;
        }
}
```

Output:

Enter first Integer Number: 2
Enter second Integer Number: 5
Binary representation of the first Integer: 10
Binary representation of the second Integer: 101
The 2.0to the power 5 is 32.0
Binary representation of the 2.0 to the power 5 is 100000

Time Complexity:

- The number of multiplications is at most twice the index of y's MSB, implying an *0(n)* time complexity.

Write a program which takes an integer and returns the integer corresponding to the digits of the input written in reverse order.

Example:
The given integer is of 42, the reverse is 24.
The given integer is of -314, the reverse is -413.

Approach:

- Let the input be x and x > 0.
- First, find the remainder of the given number by using the modulo (%) operator (i.e., x % 10).
- Multiply the variable reverse by 10 and add the remainder into it.
- The subsequent digits are found by dividing the number by 10, (i.e., x % 10).
- Repeat the above steps until the number becomes 0.

- if x < 0, then record its sign, solve the problem for |x|, and apply the sign to the result.

Example: Reverse the number 1234.

- Consider three variables named number (the number to be reversed), remainder = 0 (it stores the remainder), reverse = 0 (it stores the reverse number).

- Iteration 1:

  number = 1234
  remainder = 1234 % 10 = 4
  reverse = 0 * 10 + 4 = 0 + 4 = 4
  number = 1234 / 10 = 123

  Now the value of the number and reverse variable is 123 and 4, respectively.

- Iteration 2:

  number = 123
  remainder = 123 % 10 = 3
  reverse = 4 * 10 + 3 = 40 + 3 = 43
  number = 123 / 10 = 12

  Now the value of the number and reverse variable is 12 and 43, respectively.

- Iteration 3:

  number = 12
  remainder = 12 % 10 = 2
  reverse = 43 * 10 + 2 = 430 + 2 = 432
  number = 12 / 10 = 1

  Now the value of the number and reverse variable is 1 and 432, respectively.

- Iteration 4:

  number = 1
  remainder = 1 % 10 = 1
  reverse = 432 * 10 + 1 = 4320 + 1 = 4321
  number = 1 / 10 = 0

  Now the variable number becomes 0. Hence, we get the reverse number 4321.

Programme:

```
import java.util.Scanner;

public class ReverseProg1 {
```

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

        int x;
        long reverse;

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

        reverse = reverse(x);
        System.out.println("The reverse integer of " + x + " is " + reverse);
    }

    public static long reverse(int x){

        long result = 0;
        long xRemaining = Math.abs(x);

        while (xRemaining != 0) {

            result = result * 10 + xRemaining % 10;
            xRemaining /= 10;
        }

        return x < 0 ? -result : result;
    }
}
```

Output:

Enter an Integer Number: 1234
The reverse integer of 1234 is 4321

Time Complexity:

- The time complexity is ***0(n),*** where ***n*** is the number of digits in ***x.***

Write a program that takes an integer and check that number is palindrome or not.

- A Palindrome Number is a number that remains the same number when it is reversed. For example, 131 is a palindrome, because when its digits are reversed, it remains the same number.

- Examples of palindrome Number:

0, 1, 7, 11, 121, 393, 34043, 111, 555, 48084

Approach:

- If the input is negative, then the given integer cannot be palindrome Number, since it begins with a minus symbol (i.e., -)
- If the input is positive, then iteratively compare pair wise digits, such as the least significant digit (LSB) and the most significant digit (MSB).
- Let x be an inputted integer number and n is the number of digits, then

$$n = \lfloor log_{10} x \rfloor + 1$$

- The least significant digit (LSB) and the most significant digit (MSB) can be extracted from x as follows
    - The least significant digit (LSB) is x%10
    - The most significant digit (MSB) is $x/10^{n-1}$
- If all the pair wise digits (i.e., all compared LSB and MSB digits) are matched, then process return true (i.e., the number is a palindrome).
- If any mismatch occurs between the pair wise digits (i.e., LSB and MSB digits) then stop the process and return false (i.e., the number is not a palindrome).

Example: Suppose the input integer is 157751

- In Integer 157751, the programme compare the leading (i.e., MSB) and trailing (i.e., LSB) digits, 1 and 1. Since these are equal, then update the integer to 5775.
- In Integer 5775, the programme compare the leading (i.e., MSB) and trailing (i.e., LSB) digits, 5 and 5. Since these are equal, so update the integer to 77.
- In Integer 77, the programme compare the leading (i.e., MSB) and trailing (i.e., LSB) digits, 7 and 7. Since these are equal and no more digits, the program return true.

Programme:

```
import java.util.Scanner;

public class PalindromeProg1 {

    public static void main(String[] args) {

        int x;
        boolean result;

        System.out.print("Enter an Integer Number: ");
```

```
		Scanner sc = new Scanner(System.in);
		x = sc.nextInt();

		result = isPalindromeNumber(x);    // true or false

		if(result) {
			System.out.println("The integer " + x + " is palindrome.");
		}
		else {
			System.out.println("The integer " + x + " is not palindrome.");

		}
	}


public static boolean isPalindromeNumber(int x) {

		if (x < 0) {

			return false;
		}

		final int numDigits = (int) (Math . floor (Math . log10(x))) + 1;

		int msdMask = (int)Math .pow(10 , numDigits - 1);

		for (int i = 0; i < (numDigits / 2); ++i) {

			if (x / msdMask != x % 10) {

				return false ;
			}

			x %= msdMask; // Remove the most significant digit of x.

			x /= 10; // Remove the least significant digit of x.

			msdMask /= 100;
		}

		return true ;
	}
}
```

Output:

Case -1: (True)
Enter an Integer Number: 157751
The integer 157751 is palindrome.

Case -2: (True)
Enter an Integer Number: 152751
The integer 152751 is not palindrome.

Time Complexity:

- The time complexity is O(n), and the space complexity is O(1). The space complexity is O(1), because this programme directly extracting the digits from the input.

Write a program which tests if two rectangles have a nonempty intersection. If the intersection is nonempty, return the rectangle formed by their intersection.

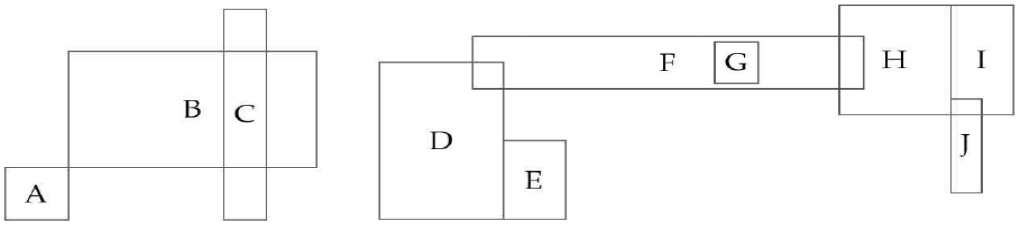


Figure 5.2: Examples of XY-aligned rectangles.

In Figure 5.2, There are many qualitatively different ways in which rectangles can intersect, e.g.,

- The Rectangle A and B share a common corner
- Two rectangles form a cross (B and C)
- The Rectangle D and E share a common side.
- The rectangle D and F have partial overlap
- One contains the other (such as Rectangle F contains G)
- Two rectangles form a tee (such as F and *H)*, etc.

Programme:

```
import java.util.Scanner;

class Rectangle {

    int x, y, width, height;

    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this. width = width;
```

```
            this. height = height;
        }

        void display(){

            System.out.println(" x: " + x  + ", y: " + y + ", width: " + width + ", hight: " + height);
        }
}


public class RectangleIntersectProg1 {

        public static Rectangle intersectRectangle(Rectangle R1 , Rectangle R2) {

            if (!islntersect(R1 , R2)) {
                return new Rectangle(0, 0, -1, -1); // No intersection.
            }

            //At which point rectangle are intersect
            return new Rectangle(

            Math.max(R1.x , R2.x), Math.max(R1.y , R2.y),

            Math.min(R1.x + R1.width, R2.x + R2.width) - Math.max(R1.x , R2.x),

            Math.min(R1.y + R1.height, R2.y + R2.height) - Math.max(R1.y , R2.y));
        }

        //rectangle are intersect or not intersect
        public static boolean islntersect(Rectangle Rl , Rectangle R2) {

            return Rl.x <= R2.x + R2.width && Rl.x + Rl.width >= R2.x
                    && Rl.y <= R2.y + R2.height && Rl.y + Rl.height >= R2.y;
        }

        public  static void main(String[] args) {

            Rectangle R1 = new Rectangle(2, 3, 4, 5);
            R1.display();

            Rectangle R2 = new Rectangle(4, 5, 10, 12);
            R2.display();

            boolean result = islntersect(R1, R2);

            //rectangle are intersect or not intersect
            if(result) {

                System.out.println(" Rectangles are intersect.");
            }
```

```
                else {

                        System.out.println(" Rectangles are not intersect.");
                }

                //At which point rectangle are intersect
                Rectangle R3 = intersectRectangle(R1, R2);
                R3.display();
        }
}
```

Output:

Case -1: (Rectangles are intersect)

```
 x: 2, y: 3, width: 4, hight: 5
 x: 4, y: 5, width: 10, hight: 12
 Rectangles are intersect.
 x: 4, y: 5, width: 2, hight: 3
```

Case -2: (Rectangles are not intersect)

```
x: 2, y: 3, width: 4, hight: 5
x: 7, y: 10, width: 10, hight: 12
Rectangles are not intersect.
x: 0, y: 0, width: -1, hight: -1
```

Time Complexity:

- The time complexity is *0(1)*, since the number of operations is constant.

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

        dutchFlagPartition(pivotIndex , 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 pivotIndex , List<Color> A) {

        Color pivot = A.get(pivotIndex);

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

        dutchFlagPartition(pivotIndex , 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 pivotIndex , List<Color> A) {

        Color pivot = A.get(pivotIndex);
        /*
        * 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 writeIndex = 1;
                for (int i = 1; i < A.size(); ++i) {

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

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

                        }
                }

                return writeIndex;
        }
}
```

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:

```
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 randIdxGen = 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 = randIdxGen.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 $p_o, p_1, \ldots p_n$, 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 $p_o, p_1, \ldots p_n$, 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, $p_0+p_1+p_2$, … , $p_0+p_1+p_2+ \ldots +P_{n-1}$ 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, ... 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).

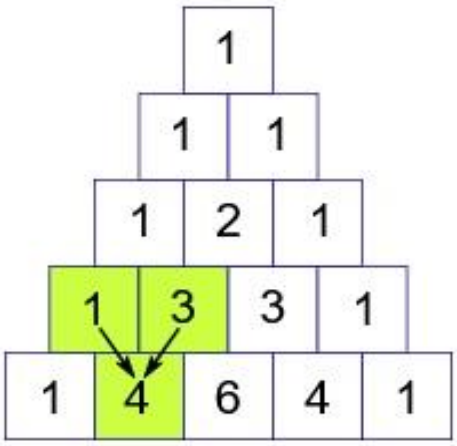


Fig: Pascal's Triangle

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 $(j\text{- }1)^{th}$ and j^{th} entries in the $(i\text{- }1)^{th}$ 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+ …+ *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

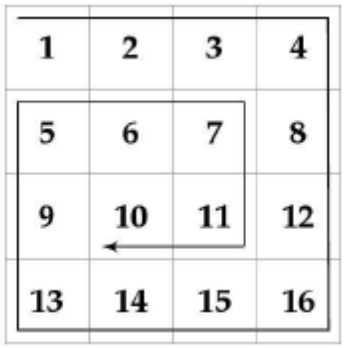
A 2D array can be written as a sequence in several orders:

- 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 $O(1)$.

Strings

- String is a sequence of characters, for e.g. “Hello” is a string of 5 characters.
- String is basically an object that is backed internally by a char array.
- The CharSequence interface is used to represent the sequence of characters.
- We can create strings in java by using three classes.
    - String
        - In java, string is an immutable object which means it is constant and can’t be changed once it has been created.
    - StringBuffer
        - Java StringBuffer class is used to create mutable (modifiable) string.
        - The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.
    - StringBuilder
        - Java StringBuilder class is used to create mutable (modifiable) string.
        - The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized.
        - It is available since JDK 1.5.

Program:

```
public class CreateStringProg1 {

    public static void main(String[] args) {

        //create string using java string literal
        String s1 = "Java String using java string literal ";
        System.out.println(s1);
```

```
        //creating java string using new keyword
        String s2 = new String("Java String using new keyword");
        System.out.println(s2);

        //Create string by converting char array to string
        char ch[]={'H', 'e', 'l', 'l', 'o',' ','I','n','d','i', 'a'};
        String s3 = new String(ch);
        System.out.println(s3);

        //creating java string by using StringBuffer()
        StringBuffer s_buffer=new StringBuffer("Java String using StringBuffer()");
        System.out.println(s_buffer);

        //creating java string by using StringBuilder()
        StringBuilder s_build = new StringBuilder("Java String using StringBuilder()");
        System.out.println(s_build);
    }
}
```

Output:

```
Java String using java string literal
Java String using new keyword
Hello India
Java String using StringBuffer()
Java String using StringBuilder()
```

The key methods in StringBuilder

The key methods in StringBuilder		
Method	**Return Type**	**Description**
append(String s)	StringBuilder	The append() method is used to append the specified string with this string.
insert(int offset, String s)	StringBuilder	The insert() method is used to insert the specified string within a string at the specified position.
replace(int startIndex, int endIndex, String str)	StringBuilder	The replace() method is used to replace the string from specified startIndex and endIndex.
delete(int startIndex, int endIndex)	StringBuilder	The delete() method is used to delete the string from specified startIndex and endIndex.
reverse()	StringBuilder	The reverse() method is used to reverse

		the string.
charAt(int index)	char	The charAt() method is used to return the character at the specified position.
deleteCharAt(int index)	StringBuilder	The deleteCharAt () method is used to delete the character at specified index
length()	int	The length() method is used to return the length of the string i.e. total number of characters.
substring(int beginIndex)	String	The substring() method is used to return the substring from the specified beginIndex.
substring(int beginIndex, int endIndex)	String	The substring() method is used to return the substring from the specified beginIndex and endIndex.
toString()	String	The toString() method to get string representation of an object.

Program:

```
public class KeyMethodsStringBuilderExample {

    public static void main(String[] args) {

        StringBuilder sb1 = new StringBuilder("Hello ");
        sb1.append("Java");                          //now original string is changed
        System.out.println("append() example: " + sb1);      //prints Hello Java

        StringBuilder sb2 = new StringBuilder("Hello ");
        sb2.insert(1,"Java");                        //now original string is changed
        System.out.println("insert() example: " + sb2);      //prints HJavaello

        StringBuilder sb3 = new StringBuilder("Hello");
        sb3.replace(1,3,"Java");
        System.out.println("replace() example: " + sb3);      //prints HJavalo

        StringBuilder sb4 = new StringBuilder("Hello");
        sb4.delete(1,3);
        System.out.println("delete() example: " + sb4);      //prints Hlo

        StringBuilder sb5 = new StringBuilder("Hello");
        sb5.reverse();
        System.out.println("reverse() example: " + sb5);      //prints olleH

        StringBuilder sb6 = new StringBuilder("Hello");
        char ch = sb6.charAt(0);
```

```
        System.out.println("charAt() example: " + ch);        //prints H

        StringBuilder sb7 = new StringBuilder("Hello");
        sb7.deleteCharAt(1);
        System.out.println("deleteCharAt() example: " + sb7);  //prints Hllo

        StringBuilder sb8 = new StringBuilder("Hello World");
        int len = sb8.length();
        System.out.println("length() example: " + len);         //prints 11

        StringBuilder sb9 = new StringBuilder("Hello World");
        String st1= sb9.substring(6);
        System.out.println("substring() with one argument example: " + st1);
                                                                    //prints World

        StringBuilder sb10 = new StringBuilder("Hello World");
        String st2= sb10.substring(2, 8);
        System.out.println("substring() with two argument example: " + st2);
                                                                    //prints llo Wo

        StringBuilder sb11 = new StringBuilder("Hello World");
        String st3= sb11.toString();
        System.out.println("toString() example: " + st3);              //prints Hello World
    }
}
```

Output:

```
append() example: Hello Java
insert() example: HJavaello
replace() example: HJavalo
delete() example: Hlo
reverse() example: olleH
charAt() example: H
deleteCharAt() example: Hllo
length() example: 11
substring() with one argument example: World
substring() with two argument example: llo Wo
toString() example: Hello World
```

Palindrome String

A palindrome string is one which reads the same when it is reversed.

Write the program to checks whether a string is palindrome or not.

Approach:

- Traverses the input string forwards and backwards, and compare the each character.
    - thereby saving space.

Program:

```
import java.util.Scanner;

public class CheckPalindromicStringProg1 {

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);
		System.out.print("Enter a string to check palindrome or not:  ");
		String str = sc.nextLine();

		str = str.toLowerCase();

		boolean flag = isPalindromic(str);

		if(flag)

			System.out.print("String " + str + " is palindrome.");

		else

			System.out.print("String " + str + " is not palindrome.");
	}

	public static boolean isPalindromic(String s){

		for (int i = 0, j = s.length() - 1; i < j; ++i, --j) {

			if (s.charAt(i) != s.charAt(j)) {

				return false;
			}
		}
```

```
                return true ;
        }
}
```

Output:

Case 1:
Enter a string to check palindrome or not: Madam
String madam is palindrome.

Case 2:
Enter a string to check palindrome or not: Sir
String sir is not palindrome.

Time and Space Complexity:

- The time complexity is O(n) and the space complexity is O(1), where n is the length of the string.

INTERCONVERT STRINGS AND INTEGERS

- A string is a sequence of characters.
- A string may encode an integer, e.g., "123" encodes 123.
- Interconvert of strings and integer means
    - Convert a string number to integer
    - Convert an integer to string

Write a program to convert a given string to an integer without using any in-built functions.

For example:

- Case 1: If string is positive number
    - If user inputs "321", then the program should give output 321 as an integer number.
- Case 2: If string is negative number

- If user inputs "-321", then the program should give output -321 as an integer number.

Program:

```
import java.util.Scanner;

public class StringToIntConversionProg1 {

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);
		System.out.print("Enter an Integer Number, Which Stores as String :  ");
		//store as string
		String strNum = sc.nextLine();

		//stringToInt() method convert the integer number to string number
		int intNum = stringToInt(strNum);

		//Display the string number
		System.out.println("The Entered Number in Integer : " + intNum);

	}

	public static int stringToInt(String strNum) {

		//initialize isNegative as false and result as zero
		boolean isNegative = false;
		int result = 0;

		//Iterate till length of the string to Extract
		//the each digit from given string number
		for (int i = 0; i < strNum.length(); i++) {

			if (strNum.charAt(i) == '-') {

				//if the first character is '-', set isNegative as true
				isNegative = true;
			}

			else {

			//converting each character into ASCII format and form the number.
			//Minus the ASCII code of '0' to get the numeric value of the charAt(i)
			final int digit = (int)( strNum.charAt(i) - '0');

			//multiply the partial result by 10 and add the digit.
			result = result * 10 + digit;
```

```
                }
            }

            //if the first character is '-', i.e., isNegative is true, then adds the
            // negative sign to the result and return it otherwise return result
            if (isNegative) {

                return - result;
            }

            else {

                return result;
            }
        }
}
```

Output:

Case 1:

Enter an integer number, which stores as string : 321
The entered number in integer : 321

Case 2 :

Enter an integer number, which stores as string : -321
The entered number in integer : -321

Time and Space Complexity:

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

Write a program to convert a given integer number to string, without using inbuilt method i.e., parseInt() and valueOf() in java.

For example:

- Case 1: If positive integer
    - If user inputs 871, then the program should give output "871" as a string number.
- Case 2: If string is negative number

- If user inputs -871, then the program should give output "-871" as a string number.

Program:

```
import java.util.Scanner;

public class IntToStringConversionProg1 {

	public static void main(String[] args) {

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

		//intToString() method convert the integer number to string number
		String strNum = intToString(intNum);

		//Display the string number
		System.out.println("The Entered Number in String : " + strNum);
	}

	public static String intToString(int num) {

		//initialize isNegative as false
		boolean isNegative = false;

		//if the num is less than 0, set isNegative as true
		if (num < 0) {

			isNegative = true;
		}

		//Create object of mutable string by using StringBuilder()
		StringBuilder sb = new StringBuilder();

		//Extract the each digit from given number (i.e., num)
		//add it to the end
		do {
			//extract one digit at a time from the integer
			// by moving right to left order;
			//starting from the LSB towards MSB
			int digit = num % 10;

			//convert digit to char and append to object of mutable string
			sb.append ((char) ('0' + Math.abs(digit)));
```

```
                //update the num by removing the digit you just extracted.
                num /= 10;

            } while(num != 0);

            //if the num is negative number then adds the negative sign
            if (isNegative) {

                sb.append('-') ;
            }

            //reverse the mutable string to get the computed result
            sb.reverse();

            // Convert the mutable string to immutable string and return
            return sb.toString () ;
        }
}
```

Output:

Case 1:

Enter an Integer Number : 7357
The Entered Number in String : 7357

Case 2 :

Enter an Integer Number : -7357
The Entered Number in String : -7357

Time and Space Complexity:

- The time complexity is O(n) and space complexity is O(n). Here n is the number of digits in input number and/or output string length.

BASE CONVERSION

In Computer Science, quite often, we are required to convert numbers from one base to another. The commonly used number systems are binary numbers system (i.e., bases 2), octal numbers system (i.e., bases 8), decimal numbers system (i.e., bases 10) and hexadecimal numbers system (i.e., bases 16).

Binary numbers include digits 0 and 1, and so they have base 2. Octal numbers have digits from 0 to 7, thus having base 8. Decimal numbers have digits from 0 to 9, thus have base 10. And hexadecimal numbers have digits from 0 to 9 and then A to F, thus having base 16.

In the decimal number system, the position of a digit is used to signify the power of 10 that digit is to be multiplied with. For example, "314" denotes the number 3 X 100 +1 X 10 + 4 X 1. The base *b* number system generalizes the decimal number system: the string *"$a_{n-1}\ a_{n-2}$ ... $a_1\ a_0$"*, where $0 < a_i < b$, denotes in base-b the integer $a_{n-1} X\ b^{n-1} + a_{n-2} X\ b^{n-2} ... a_1 X\ b^1 + a^0 X\ b^0$.

Write a program in Java **to convert a number in any base into another base, limiting the base value up to 16. The input is a string, an integer b_1, and another integer b_2. The string represents be an integer in base b_1. The output should be the string representing the integer in base b_2. Assume $2 \leq b_1, b_2 \leq 16$. Use "A" to represent 10, "B" for 11, ..., and "F" for 15. (For example, if the string is "615", b_1 is 7 and b_2 is 13, then the result should be "1A7", since $6 \times 7^2 + 1 \times 7 + 5 = 1 \times 13^2 + 10 \times 13 + 7$).**

Approach:

- First convert a string in base b_1 to decimal integer value using a sequence of multiplications and additions.
- Then convert that decimal integer value to a string in base b_2 using a sequence of modulus and division operations.

Example:

Program:

```
import java.util.Scanner;

public class BaseConversionProg1 {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        System.out.print("Enter an Integer Number, Which Stores as String :  ");
        //store as string
        String strNumb1 = sc.nextLine();

        System.out.print("Enter Source base (between 2 and 16) : ");
         int b1 = sc.nextInt();

        System.out.print("Enter Destination base (between 2 and 16) : ");
        int b2 = sc.nextInt();

        String strNumb2 = convertBase(strNumb1, b1, b2);
```

```
        System.out.println("The base " + b1 + " equivalent of " + strNumb1 + " in base "
                                                                + b2 + " is " + strNumb2);
}

public static String convertBase(String numAsString , int b1, int b2) {

        //startsWith(), tests the string starts with the specified prefix.
        boolean isNegative = numAsString.startsWith("-");

        //initialization
        int numAsInt = 0;
        int digit = 0;

        //Iterate till length of the string to Extract the each digit
        // the given string number is converted to decimal integer value.
        //convert each character into ASCII format and form the decimal value.
        for (int i = (isNegative ? 1 : 0); i < numAsString.length(); ++i) {

                //Check the specified character in the string is a digit or not
                if (Character.isDigit(numAsString.charAt(i))) {

                        //if character is a digit, then minus the ASCII code
                        //of '0' to get the decimal integer value of the charAt(i)
                         digit = (int)(numAsString.charAt(i) - '0');
                }

                else {

                        //if character is not a digit, then minus the ASCII code
                        //of 'A' to get the decimal integer value of the charAt(i)
                        digit = (int)(numAsString .charAt(i) - 'A' + 10);

                }

                //multiply the partial result by b1 and add the digit.
                numAsInt = numAsInt * b1 + digit;
        }

        //Check the entered number is zero or not
        if (numAsInt == 0)      {

                return ("0");
        }

        else {
```

```
                    //if the first character is '-', i.e., isNegative is true then adds
                    //negative sign to the result and return it; otherwise return result.
                    //The result is found by Converting the decimal value to base b2
                    if (isNegative) {

                            return ("-" + constructFromBase(numAsInt , b2));
                    }

                    else  {

                            return ("" + constructFromBase(numAsInt , b2));
                    }
            }
    }

    // Converting the decimal value to base b2 conversion using recursion:
    private static String constructFromBase(int numAsInt, int base) {

            if (numAsInt == 0) {

                    return "";              // base case
            }

            else {
                    //If the remainder is >= 10 or not
                    if(numAsInt % base >= 10) {

                       //If the remainder is >= 10, add a character with the
                       //corresponding value (i.e., A = 10, B = 11, C = 12, ...)
                      return (constructFromBase(numAsInt / base, base)
                                                    + (char)('A' + numAsInt % base - 10));
                    }

                    else {
                             // If the remainder is a digit < 10, simply add it
                            return (constructFromBase(numAsInt / base, base)
                                                        + (char)('0' + numAsInt % base));
                    }
            }
    }
}
```

Output:

Case 1:

Enter an Integer Number, Which Stores as String : 615

Enter Source base (between 2 and 16) : 7
Enter Destination base (between 2 and 16) : 13
The base 7 equivalent of 615 in base 13 is 1A7

Case 2:

Enter an Integer Number, Which Stores as String : -1A7
Enter Source base (between 2 and 16) : 13
Enter Destination base (between 2 and 16) : 7
The base 13 equivalent of -1A7 in base 7 is -615

Case 3:

Enter an Integer Number, Which Stores as String : 0
Enter Source base (between 2 and 16) : 3
Enter Destination base (between 2 and 16) : 8
The base 3 equivalent of 0 in base 8 is 0

Time and Space Complexity:

- The time complexity is $(n(1 + \log_{b_2} b_1))$, where *n* is the length of string. The reasoning is as follows. First, we perform *n* multiply-and-adds to get d*ecimal integer, i.e., x* from string. Then we perform $\log_{b_2} x$ multiply and adds to get the result. The value *x* is upper-bounded by b_1^n, and $log_{b_2}\ (b_1^n) = n\ log_{b_2}\ b_1$

COMPUTE THE SPREADSHEET COLUMN ENCODING

Spreadsheets or excel sheet often use an alphabetical encoding of the successive columns. Specifically, columns are identified by "A", "B", "C", . . ., "X", "Y", "Z", "AA", "AB", ..., "ZZ", "AAA", "AAB",.... and so on.

Write a program that converts a spreadsheet column id to the corresponding *column number, i.e.*, integer. For Example, if the input (i.e., column id) to the program is A, the output (i.e., column number) is 1. Similarly, if the input is D the output is 4. Suppose the input is AB the program should return output as 28.

Approach:

- This problem is basically the problem of converting a string representing a base-26 number to the corresponding integer, except that "A" corresponds to 1 not 0.

- Each alphabet has a numeric equivalent, i.e., A = 1, B = 2, ... Z = 26.

Example -1: Convert AB,

Use the system where an A = 1, B = 2, ... Z = 26.

Therefore,

$1 \text{ X } 26^1 + 2 \text{ X } 26^0$
$= 1 \text{ X } 26 + 2 \text{ X } 1$
$= 26 + 2$
$= 28$

Example -2: Convert CDA

Use the system where an A = 1, B = 2, ..., Z=26.

Therefore,

$3 \text{ X } 26^2 + 4 \text{ X } 26^1 + 1 \text{ X } 26^0$
$= 3 \text{ X } 676 + 4 \text{ X } 26 + 1 \text{ X } 1$
$= 2028 + 104 + 1$
$= 2133$

Program:

```
import java.util.Scanner;

public class SSColIdtoColNumConversion {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a spreadsheet column id (Upper case Chracter(s) only):  ");
        String colId = sc.nextLine();

        //stringToInt() method convert the integer number to string number
        int colNum = ssDecodeColID(colId);

       //Display the column number
       System.out.println("The equivalent column number is " + colNum);

    }

    public static int ssDecodeColID(final String colId) {

        //Initialization
        int result = 0;
```

```
                //Iterate till length of the string to Extract the each character
                // and compute base-26 number: multiply the partial result by 26
                //and add the digit.
                for (int i = 0; i < colId.length () ; i++) {

                        char ch = colId.charAt (i) ;

                        result = result * 26 + ch - 'A' + 1;
                }

                return result;
        }
}
```

Output:

Enter a spreadsheet column id (Upper case Chracter(s) only): CDA
The equivalent column number is 2133

Time and Space Complexity

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

Write a program that converts a spreadsheet column id to the corresponding *column number, i.e.*, integer. For Example, if the input (i.e., column id) to the program is A, the output (i.e., column number) is 1. Similarly, if the input is D the output is 4. Suppose the input is AB the program should return output as 28.

Given an integer `columnNumber`, return *its corresponding column title as it appears in an Excel sheet*.

Implement
a function that converts an integer to the spreadsheet column id. For example, you
should return "D" for 4, "AA" for 27, and "ZZ" for 702.

Approach:

- This problem is basically the problem of converting a string representing a base-26 number to the corresponding integer, except that "A" corresponds to 1 not 0.

- Each alphabet has a numeric equivalent, i.e., A = 1, B = 2, ... Z = 26.

Example:

1. start n = 26, result = ""
   r = 26 % 26 = 0
   n = (26 / 26) - 1 = 0, result += 'Z'
   n = 0, stop,
   output 'Z'

2. start n = 676, result = ""
   r = 676 % 26 = 0
   n = (676 / 26) - 1 = 25, result += 'Z'

   r = 25 % 26 = 25
   n = (25 / 26) = 0, result += 'A' + (25 - 1) = 'Y'
   n = 0, stop,
   output reverse("ZY") = "YZ"

3. start n = 1000, result = ""
   r = 1000 % 26 = 12
   n = (1000 / 26) = 38, result += 'A' + (12 - 1) = 'L'

   r = 38 % 26 = 12
   n = (38 / 26) = 1, result += 'A' + (12 - 1) = 'L'

   r = 1 % 26 = 1
   n = (1 / 26) = 0, result += 'A' + (1 - 1) = 'A'
   n = 0, stop,
   output reverse("LLA") = "ALL"

REPLACE AND REMOVE

Write a program which takes as input an array of characters, and removes each entry containing 'b' and replaces each 'a' by two 'd's. Specifically, along with the array, you are provided an integer-valued size. Size denotes the number of entries of the array that the operation is to be applied to. You do not have to worry preserving about subsequent entries. Furthermore, the program returns the number of valid entries.

Example, if the array is < *a, b, a, c,' ',' ',' '* > and the size is 4, then the program should return the array < *d, d, d, d, c* >.

Approach:

- First, delete 'b's and compute the final number of valid characters of the string, with a forward iteration through the string.

- Then replace each 'a' by two 'd's, iterating backwards from the end of the resulting string.
- If there are more 'b's than 'a's, the number of valid entries will decrease.
- If there are more 'a's than 'b's the number will increase.

Program:

```
public class ReplaceAndRemoveProg1 {

	public static void main(String[] args) {

		char arr[] = {'a', 'b', 'a', 'c', ' ' , ' ',' ' };

		int size = 4;

		System.out.print("Before processing the array elements : ");
		for (int i=0; i < arr.length; i++) {

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

		System.out.println(" ");

		int finalSize = replaceAndRemove(size, arr);

		System.out.print("After processing the array elements : ");
		for (int i=0; i < arr.length; i++) {

			System.out.print(arr[i] + " ");
		}
		System.out.println(" ");
		System.out.println("The number of valid entries : " + finalSize);
	}

	public static int replaceAndRemove(int size, char[] s) {

		// Forward iteration: remove "b"s and count the number of "a"s.
		int writeldx = 0; //index variable
		int aCount = 0;

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

			if (s[i] != 'b') {
```

```
                    s[writeldx++] = s[i];

                }

                if (s[i] == 'a') {

                    ++aCount ;

                }
            }

            // Backward iteration: replace "a"s with "d d"s starting from the end.
            int curldx = writeldx - 1;

            writeldx = writeldx + aCount - 1;

            final int finalSize = writeldx + 1;

            while (curldx >= 0) {

                if (s[curldx] == 'a') {

                    s[writeldx --] = 'd';

                    s[writeldx --] = 'd';
                }

                else {

                    s[writeldx --] = s[curldx];

                }

                --curldx ;

            }

            return finalSize;
        }
}
```

Output:

Before processing the array elements : a b a c
After processing the array elements : d d d d c
The number of valid entries : 5

Time and Space Complexity:

- The forward and backward iteration search take O*(n)* time, so the total time complexity is *O(n).* No additional space is allocated.

TEST PALINDROMICITY

- A sentence can be defined as to be a palindrome string that reads the same thing forward and backward, after removing all the non-alphanumeric and ignoring case.

- **Example:** palindrome sentence

    - The sentence "A man, a plan, a canal, Panama." is palindrome, because after removing all the non-alphanumeric and ignoring case (here making lowercase) the sentence is turned as “amanaplanacanalpanama”; when you read it forward and backward, it is same.

- **Example:** not palindrome sentence

    - The sentence "Ray a Ray" is not a palindrome, because after removing all the non-alphanumeric and ignoring case (here making lowercase) the sentence is turned as "RayaRay"; when you read it forward and backward, it is not same.

Write a program to check if a sentence is a palindrome or not. You can ignore white spaces and other characters to consider sentences as palindrome.

Approach:

- Use two indices to traverse the string.

- One index is pointing to the 1st character (i.e., i = 0) of the sentence to move forward and another index is pointing to the last character (i.e., j = string length - 1) of the sentence to move backward, skipping non-alphanumeric characters, performing case-insensitive comparison on the alphanumeric characters.

- Return false as soon as there is a mismatch, means sentence is not palindrome. If the indices cross, return true, means sentence is palindrome.

Program:

```
import java.util.Scanner;

public class TestPalindromcityProg1 {
```

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

        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a Sentence : ");
        String str = sc.nextLine();

        boolean  flag = isPalindrome(str);

        if(flag)

                System.out.println("Sentence is palindrome");
        else

                System.out.println("Sentence is not palindrome");
}

// To check sentence is palindrome or not
public static boolean isPalindrome(String str) {

        // i and j are index variable
        // i pointing to the 1st character (i.e., i = 0) of the sentence
        // j pointing to the last character (i.e., i = 0) of the sentence
        int i = 0, j = str.length() - 1;

        // Compares character until they are equal
        while (i < j) {

                // i moves forward; skip non-alphanumeric characters.
                while (!Character.isLetterOrDigit (str.charAt(i)) && i < j) {

                        ++i ;
                }

                // j moves backward; skip non-alphanumeric characters.
                while (!Character.isLetterOrDigit(str.charAt (j)) && i < j) {

                        --j ;
                }

                // If characters are not equal then sentence is not palindrome
                if (Character.toLowerCase(str.charAt(i++)) !=
                                                Character.toLowerCase(str.charAt(j--))) {

                        return false;
                }
```

```
            }

            // Returns true if sentence is palindrome
            return true ;
        }
}
```

Output:

Enter a Sentence : A man, a plan, a canal, Panama.
Sentence is palindrome

Time and Space Complexity:

- Spend 0(1) per character and the sentence has n characters, so the time complexity is O(n). The space complexity is O(1).

REVERSE ALL THE WORDS IN A SENTENCE

A sentence is given which containing a set of words and the words are separated by whitespace. Reverse all the words in a sentence means transform that sentence in such a way so that the words appear in the reverse order.

Example,

Let the sentence is "Alice likes Bob", transforms to "Bob likes Alice".

Approach:

- Reverse the whole sentence first.
    - By reversing sentence the words appeared correctly in their relative positions.
    - However, the words with more than one character, their letters appear in reverse order.
- Finally to get the correct words, reverse the individual word.

Example:

- Let the given sentence is “Alice likes Bob”.
- Reversing the whole sentence, the obtained reverse sentence is “boB sekil ecilA”.

- By reversing each word in reverse sentence, obtain the final result, “Bob likes Alice”.

Program:

```
import java.util.Scanner;

public class ReverseAllWordsInSentenceProg1 {

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);
		System.out.print("Enter a Sentence : ");
		String str = sc.nextLine();

		// toCharArray() method converts a string
		// to a new character array.
		char[] chArray = str.toCharArray();

		//Calling function which reverse all words in a sentence
		reverseWords(chArray);

		// valueOf() method returns the string representation
		//of the char array argument.
		String Output = String.valueOf(chArray);

		// Display the resultant String
		System.out.println("After reversing all words, the sentence : " +  Output);
	}

	//reverse all words in a sentence
	public static void reverseWords(char[] input) {

		// Reverses the whole string first.
		reverse(input, 0, input.length);

		int start = 0;

		//In the reverse string, it find the
		// index of first word boundary
		int end = find(input, ' ' , start);

		//reverses each individual word in the
		// reverse string, except last word
		while (end != -1) {

			// Reverses each word in the reverse string.
```

```
                reverse( input , start, end);
                start = end + 1;

                //it find the index of next word boundary
                end = find(input, ' ' , start);
            }

            // Reverses the last word of reverse string.
            reverse(input, start, input.length);
        }

        public static void reverse(char[] array, int start, int stoplndex) {

            if (start >= stoplndex) {

                return ;
            }

            int last = stoplndex - 1;

            for (int i = start; i <= start + (last - start) / 2; i++) {

                char tmp = array[i];

                array[i] = array[last - i + start];

                array[last - i + start] = tmp;
            }
        }

        public static int find(char[] array, char c, int start){

            for (int i = start; i < array.length ; i++) {

                if (array[i] == c) {

                    return i ;
                }
            }

            return -1;
        }
}
```

Output:

Enter a Sentence : Alice likes Bob
After reversing all words, the sentence : Bob likes Alice

Time and Space Complexity:

- The time complexity is O*(n)*, where *n* is the length of the of the string. The additional space complexity is *O(1)*.

CONVERT FROM ROMAN TO DECIMAL

- The Romans used Latin letters for writing numbers.
- Roman numerals are a number system developed in ancient Rome where letters represent numbers.
- The modern use of Roman numerals involves the letters I, V, X, L, C, D, and M.
    - In Roman numeral system I is 1, V is 5, X is 10, L is 50, C is 100, D is 500, M is 1000.
- Roman numeral system is classic example non-positional numeral systems
    - The value of a figure is independent of its positions.
        - For example, number 3, is written as III in Roman numerals.
    - Although, it's a non-positional numeral system but there is an additional rule that modify the value of a digit according to its place.
        - That rule forbids the use of the same digit 3 times in a row. That's why 3 is III
- The Roman number symbols to be a valid Roman Number, if symbols appear in non increasing order with the following exceptions allowed:
    - I placed before V or X indicates one less, so 4 is IV (one less than 5) and 9 is IX (one less than 10).
    - X placed before L or C indicates ten less, so 40 is XL (10 less than 50) and 90 is XC (ten less than a hundred).
    - C placed before D or M indicates a hundred less, so 400 is CD (100 less than 500) and 900 is CM (100 less than 1000).
- **Note:** Back-to-back exceptions are not allowed, e.g., IXC and CDM are invalid.

- A valid complex Roman number string represents the integer which is the sum of the symbols that do not correspond to exceptions; for the exceptions, add the difference of the larger symbol and the smaller symbol.

Write a program which takes as input a valid Roman number string and returns the corresponding integer.

Approach:

- Performs the right-to-left iteration, and if the symbol after the current one is greater than it, then subtract the current symbol.

Program:

```
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

public class ConvertRomanNumberToDecimalProg1 {

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);
		System.out.print("Enter a Roman Number : ");
		String RomNum = sc.nextLine();

		int DecNum =  romanToInteger(RomNum);

		System.out.print("Roman Number " + RomNum + " is equivalent to Decimal
                                                                Number " + DecNum);
	}

	public static int romanToInteger(String RomNum){

		// Convert String to Upper Case
		RomNum = RomNum.toUpperCase();

		Map<Character , Integer> HM = new HashMap<Character, Integer>();

                        HM.put('I', 1);
                        HM.put('V', 5);
                        HM.put('X', 10);
                        HM.put('L', 50);
                        HM.put('C', 100);
                        HM.put('D', 500);
```

```
                HM.put('M', 1000);

        // get() returns the value to which the specified key is mapped,
        // otherwise return null, if no mapping for the key.
        // initialize the sum
        int sum = HM.get(RomNum.charAt(RomNum.length() - 1));

        // Performs the right-to-left iteration
        for (int i = RomNum.length() - 2 ; i >= 0; --i) {

            // if the symbol after the current one is greater than it,
            // then subtract the current symbol. otherwise add.
            if (HM.get(RomNum.charAt(i)) < HM.get(RomNum.charAt(i + 1))) {

                sum -= HM.get(RomNum.charAt(i));

            } else {

                sum += HM.get(RomNum.charAt(i));

            }
        }

        return sum;
    }
}
```

Output:

Enter a Roman Number : XC
Roman Number XC is equivalent to Decimal Number 90

Time and Space Complexity:

- The overall time complexity is O(n), because for each character of string is processed in O(1) time, where n is the length of the string. The Space complexity is O(1).

WRITE A STRING SINUSOIDALLY

- Write a string "Hello World!" in sinusoidal fashion, which as shown as below:

```
     e       ␣       l
H    l   o   W   r   d
      l       o       !
```

Here ␣ denotes a blank

- From the sinusoidal fashion of the string, it is possible to write the snakestring.
- To write the snakestring of the string, write the sequence from left to right and from top to bottom in which the characters appear in the sinusoidal fashion of the string.
- Example :
    - The snakestring for "Hello␣World!" is "e␣lHloWrdlo!".

Write a program which takes as input a string and returns the snakestring of the inputted string.

Approach:

- It can be observed that, write the given string in a sinusoidal manner in the array, which is shown below. Finally, for snakestring read out the non-null characters in row-manner.

0			e				␣				l	
1	H			l			o		W			r			d		
2						l					o						!
	0	1	2	3	4	5	6	7	8	9	10	11

Here ␣ denotes a blank

- However, it observe that the result begins with the characters s[l], s[5], s[9],..., followed by s[0],s[2], s[4],..., and then s[3], s[7], s[11],…
- Hence, create snakestring directly (without writing string in a sinusoidal manner) with 3 iterations through given input string.

Program:

```
import java.util.Scanner;

public class SinusoidalStringProg1 {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a string : ");
        String str = sc.nextLine();

        String output = snakeString(str);

        System.out.println("The sinsoida structure of given string : " + output);
    }
```

```
    public static String snakeString(String s) {

        //Create object of mutable empty string by using StringBuilder()
        StringBuilder result = new StringBuilder ();

        // Outputs the first row, i.e., s[l], s[5], s[9], ...
        for (int i = 1; i < s.length(); i += 4) {

            result.append (s . charAt (i) ) ;
        }

        // Outputs the second row, i.e., s[<9], s[2], s[4], ...
        for (int i = 0; i < s.length(); i += 2) {

            result . append (s . charAt (i) ) ;
        }

        // Outputs the third row, i.e., s[3], s[7], s[ll], ...
        for (int i = 3; i < s.length(); i += 4) {

            result . append (s . charAt (i) ) ;
        }

        return result.toString ();
    }
}
```

Output:

Enter a string : Hello World!
The sinsoida structure of given string : e lHloWrdlo!

Time and Space Complexity:

- Let *n* be the length of s. Each of the three iterations takes*0(n)* time, implying an*0(n)*time complexity.

THE LOOK-AND-SAY PROBLEM

- The look-and-say sequence starts with 1.
- Subsequent numbers are derived by describing the previous number in terms of consecutive digits.

- Specifically, to generate an entry of the sequence from the previous entry, read off the digits of the previous entry, counting the number of digits in groups of the same digit.
- The look-and-say sequence is the sequence of integers beginning as follows:

  1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211, …

  - The first term is "1"
  - Second term is "11", generated by reading first term as "One 1"
  - Third term is "21", generated by reading second term as "Two 1"
  - Fourth term is "1211", generated by reading third term as "One 2 One 1"
  - Fifth term is “111221” generated by reading the forth term as “one 1 one 2 two 1”
  - and so on.

Write a program that takes as input an integer *n* and returns the nth integer in the look-and-say sequence. Return the result as a string.

Example:

- Let n = 4, then return the 4^{th} term of the look and say sequence. Hence the sequence is 1211.

Program:

```
import java.util.Scanner;

public class LookAndSayProg1 {

    public static void main(String[] args) {

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

        String result = lookAndSay(num);

        System.out.println(" ");

      System.out.println("The " + num + " term of the look and say sequence is " + result);
```

```
}

public static String lookAndSay(int n) {

        String str = "1";

        System.out.print("The look and say sequence are " + str + ' ');

        for (int i = 1; i < n; ++i) {

                str = nextNumber(str);

                System.out.print(str + ' ');

        }

        return str;
}

private static String nextNumber(String str) {

        //Create object of mutable empty string by using StringBuilder()
        StringBuilder result = new StringBuilder();

        for (int i = 0; i < str.length(); ++i) {

                // initialize the count as 1
                // count store how many times a digit occurred
                int count = 1 ;

                // Compares current digit and the next digit and counts
                // the number of times the current digit is repeated
                while (i + 1 < str.length() && str.charAt(i) == str.charAt(i + 1)) {

                        ++i ;
                        ++count ;
                }

                // Once a different digit is encountered,
                // the count and the current digit are appended to result.
                result.append(count);
                result.append(str.charAt(i));
        }

        return result.toString();
}
```

}

Output:
Enter an Integer : 4
The look and say sequence are 1 11 21 1211
The 4 term of the look and say sequence is 1211

Time and Space Complexity:

- Since there are *n* iterations and the work in each iteration is proportional to the length of the number computed in the iteration, a simple bound on the time complexity is $0(n^2)$.

IMPLEMENT RUN-LENGTH ENCODING

- Run Length encoding (RLE) is a lossless data compression algorithm.
- RLE works by reducing the physical size of a repeating string of characters.
- This repeating string, called a run, is typically encoded into two bytes.
- The first byte represents the number of characters in the run and is called the run count.
- The second byte is the value of the character in the run and is called the run value.
- **Example:**
    - Let consider a string "BBBBBBBBBBBBBBB" and its run length encoding is 15B. In 15B, the first byte, 15, is the run count and contains the number of repetitions. The second byte, B, is the run value and contains the actual repeated value in the run.
- Every encoded string is a repetition of a string of digits followed by a single character. The string of digits is the decimal representation of a positive integer.

Run length decoding is a process of interpretation and translation of coded information into a comprehensible form.

To generate the decoded string, need to convert this sequence of digits into its integer equivalent and then write the character that many times. do this for each character.

Approach:

1. Start with an empty string for the code, and start traversing the string.
2. Pick the first character and append it to the code.

3. Now count the number of subsequent occurances of the currently chosen character and append it to the code.
4. When you encounter a different character, repeat the above steps.
5. In a similar manner, traverse the entire string and generate the code.

Given an input string, write a function that returns the Run Length Encoded string for the input string. Assume the string to be encoded consists of letters of the alphabet, with no digits.

Program:

```
import java.util.Scanner;

public class RunLengthEncodingProg1 {

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);
		System.out.print("Enter a string for run-length encoding : ");
		String str = sc.nextLine();

		String encodestr = encoding(str);

		System.out.println("The run-length encoding is " + encodestr);
	}

	public static String encoding(String str) {

		//Create object of mutable empty string by using StringBuilder()
		StringBuilder sb = new StringBuilder();

		// Count occurrences of current character
		int count = 1;

		//Iterate till length of the string
		for (int i = 1; i <= str.length(); ++i) {

			// Compares current  and the previous character
			// if current character is a new character, write
			// the count of previous character.
			if (i==str.length() || str.charAt(i) != str.charAt(i - 1)) {
```

```
                        sb.append(count);
                        sb.append(str.charAt(i - 1));

                        //set count equal to 1, for new character
                        count = 1;
                    }

                    // Compares current and the previous character and counts
                    // the number of times the previous character is repeated
                    else { // str.charAt(i) == str.charAt(i - 1).

                        ++count ;
                    }
                }

                return sb.toString();
            }
}
```

Output:

Enter a string for run-length encoding : aaaabcccaa
The run-length encoding is 4a1b3c2a

Time and Space Complexity:

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

Given an input Run Length Encoded string, consisting of digits and lowercase alphabet characters write a function that returns the decoded string. The decoded string is guaranteed not to have numbers in it.

Program:

```
import java.util.Scanner;

public class RunLengthDecoding {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        System.out.print("Enter a string for run-length decoding : ");
        String str = sc.nextLine();

        String decodestr = decoding(str);
```

```
            System.out.println("The run-length decoded string is " + decodestr);
        }

        public static String decoding(String s) {

            int count = 0;

            StringBuilder result = new StringBuilder () ;

            for (int i = 0; i < s.length(); i++) {

                char c = s.charAt(i);

                if (Character . isDigit(c)) {

                    count = count * 10 + c - '0' ;

                } else { // c is a letter of alphabet.


                    while (count > 0){ // Appends count copies of c to result.

                        result.append(c);

                        count -- ;
                    }
                }
            }

            return result.toString();
        }
}
```

Output:

Enter a string for run-length decoding : 3e4f2e
The run-length decoded string is eeeffffee

Time and Space Complexity:

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

COMPUTE ALL VALID IP ADDRESSES

- Internet Protocol (**IPv4 or IP)** addresses are canonically represented in dot-decimal notation, which consists of four decimal numbers, each ranging from 0 to 255, separated by periods and cannot have leading zeros.

- Examples of valid IP addresses are 0.1.2.201 and 192.168.1.1

- Examples of invalid IP addresses are 0.011.255.245 and 192.168.1.312

- The IP address 0.011.255.245 is invalid, due to leading 0 present in the second decimal number, i.e., 011.

- The IP address 192.168.1.312 is invalid, because the fourth decimal number, i.e., 312 is beyond the range 0 to 255.

Let the given decimal string. A decimal string is a string consisting of digits between 0 and 9. Write a program that determines where to add periods to a decimal string so that the resulting string is a valid IP address. There may be more than one valid IP address corresponding to a string, hence return all possible valid IP addresses that can be obtained.

Example:

- Let the input is “19216811”

- The output returns 9 different IP addresses as an array

    - [92.168.1.11, 19.2.168.11, 19.21.68.11, 19.216.8.11, 19.216.81.1, 192.68.1.11, 192.16.8.11, 192.16.81.1, 192.168.1.1]

Approach:

- There are three periods in a valid **IP** address, hence enumerate all possible placements of these periods, and check whether all four corresponding substrings are between 0 and 255. It is possible to reduce the number of placements by spacing the periods 1 to 3 characters apart. It is also potential to lead by stopping as soon as a substring is invalid.

- **Example**:

    - Let the string is "19216811", one could put the first period after "1", "19" and "192". If the first part is "1", after placing the second period, the second part might be "9", "92" and "921". Amongst these, "921" is illegal, so do not go on with it.

Program:

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

```
import java.util.Scanner;

public class ComputeAllValidIPAddressesProg1 {

        public static void main(String[] args) {

                Scanner sc = new Scanner(System.in);
                System.out.print("Enter a decimal string : ");
                String str = sc.nextLine();

                List<String> result = getValidIpAddress(str);

                System.out.println("All possible valid IP addresses :");
                for(String ip : result)

                        System.out.println(ip);

         }

// This method calculate different combinations and
// uses isValidPart to valid IP address part or substring
 public static List<String> getValidIpAddress(String s) {

        List<String> result = new ArrayList();

        for (int i = 1; i < 4 && i < s.length(); ++i) {

            final String first = s.substring(0, i) ;

            if (isValidPart(first)) {

                for (int j = 1; i + j < s.length() && j < 4; ++ j ) {

                    final String second = s.substring(i, i + j);

                    if (isValidPart(second)) {

                        for(int k = 1; i + j + k < s.length() && k < 4 ; ++k) {

                            final String third = s.substring(i + j, i + j + k) ;

                            final String fourth = s.substring(i + j + k) ;

                            if ( isValidPart(third) && isValidPart( fourth) ) {

                                    result . add(first + '.' + second + '.' + third + '.' + fourth);
```

```
                                    } //end if
                                } //end for
                            } //end if
                        } //end for
                    } //end if
                } //end for

        return result;
    }

        //Check if this string forms valid IP address part.
        private static boolean isValidPart (String s) {


                // if string length is greater than 3,
                // then that string is not a part of valid IP address.
                if (s.length() > 3) {

                        return false;
                }

                // If the string start with "0" and length greater than 1,
                // then that string is not a part of valid IP address.
                // "00", "000" , "01", etc. are not valid, but "0" is valid.
                if (s.startsWith("0") && s.length() > 1) {

                        return false;
                }

                //Parses the string argument as a signed decimal integer.
                int val = Integer.parseInt(s) ;

                // return value if it's ranging from 0 to 255
                return val <= 255 && val >= 0;
        }
}
```

Output:

```
Enter a decimal string : 19216811
All possible valid IP addresses :
1.92.168.11
19.2.168.11
19.21.68.11
19.216.8.11
```

19.216.81.1
192.1.68.11
192.16.8.11
192.16.81.1
192.168.1.1

Time and Space Complexity:

- The total number of IP addresses is a constant (232), implying an (9(1) time complexity for the above algorithm.

Linked List

- A linked list is a linear data structure.
- Linked list is the second most-used data structure after array.
- Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.
- There are many variants of linked lists: singly linked list, doubly linked list and circular linked list. Singly linked list

Singly linked list

- A singly linked list is a data structure that contains a sequence of nodes such that each node contains an object and a reference to the next node in the list.
- A node contains two fields, one is data field and another is next field.
    - Data field : which store a data called an element or an object.
    - Next field : which contains the address of the next node in the list.
- The first node is referred to as the head.
- The last node is referred to as the tail; the tail's next field is null.
- The singly linked list can be traversed only in one direction. Because each node contains only next pointer, therefore traversing the list in the reverse direction is not possible.
- The structure of a singly linked list is shown in below figure.

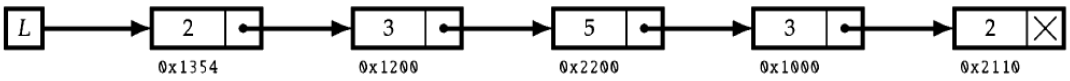


Figure 1: Example of a singly linked list.

Doubly linked list

- Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the list.
- A node contains three fields, data field, previous field and next field.
    - Data field : which store a data called an element or an object.
    - Next field : which contains the address of the next node in the list.
    - Previous field : which contains the address of the previous node in the list.
- The first node is referred to as the head; the head's previous field is null.
- The last node is referred to as the tail; the tail's next field is null.
- The structure of a doubly linked list is shown in below figure.

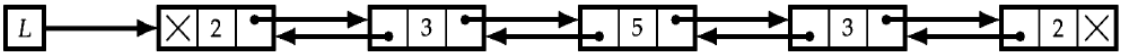


Figure 2: Example of a doubly linked list.

Program:

```
package SimpleLinkedList;

// Linked list Node
public class Node<T>{

        public T data;
        public Node<T> next;

        //Constructor
        Node(T d){

                data = d;
                next = null;
        }
}
```

```
package SimpleLinkedList;

public class LinkedList {

	// Create a new Linked List node and insert the nodes at first
	public static Node<Integer> insert(int d, Node node){

		Node<Integer> newNode = new Node<Integer>(d);
		newNode.next = node.next;
		node.next = newNode;
		return node;
	}

	// Insert newNode after given node.
	public static void insertAfter(Node<Integer> node, Node<Integer> newNode) {

		newNode.next = node.next;
		node.next = newNode;
	}


	// This function delete the an element next to head
	public static void deleteList(Node aNode) {

		aNode.next = aNode.next.next;
	}

	// returns the position of the first occurrence of the given
	// value (-1 if not found)
	public static int search (Node L, Integer key) {
		int index = 0;
		Node current = L;
		while (current != null) {
			if (current.data == key) {
				return index;
			}
			current = current.next;
			index++;
		}
		return -1;
	}
```

```
//return the size of the linked list
public static int size(Node head) {

        // initialize the count
        int count = 0;

        // if no element in the list, re
        if(head == null) {

                return count;
        }
        else {

                //count = 0;
                Node node = head;
                while(node.next != null) {

                        count++;
                        node = node.next;
                }
                return count;
        }
}

//Display the list
public static void display(Node node) {

        // display the linked list header/number,
        // for example, LinledList 1 or LinledList 2
        System.out.print("LinkedList " + node.data + " : " );

        //Display the data present in the list
        while(node.next != null) {

                System.out.print(node.next.data + " --> ");
                node = node.next;
        }

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

public static void main(String[] args) {

        //Create head node in Linked list with header/number,
        // for example, LinledList 1 or LinledList
        Node<Integer> head = new Node<>(1);
```

```
                //insert 4 element in the list
                for(int i=0;i<4;i++) {

                        head=insert(i+2,head);
                }

                System.out.println("The elements in the linked list: " );
                display(head);

                int length = size(head);
                System.out.println("The size of the linked list " + length);

                // Create a node and its data is 7. Insert the node
                // after second  node (position 3 in list) from head
                System.out.println("Insert a new node after a specified node in the linked list: ");
                Node<Integer> newNode = new Node<Integer>(7);
                insertAfter(head.next.next, newNode);
                display(head);

                //element next to head will be deleted
                System.out.println("The linked list, after deletion of element next to head: ");
                deleteList(head);
                display(head);

                //returns the position of the first occurrence of searching element
                int position = search(head, 3);
                if(position > 0)
                        System.out.println("The searching element present in Linked List at
                                                                position: " + position);
                else
                        System.out.println("The searching element is not present in Linked List.");

        }
}
```

Output:

```
The elements in the linked list:
LinkedList 1 : 5 --> 4 --> 3 --> 2 --> null
The size of the linked list 4
Insert a new node after a specified node in the linked list:
LinkedList 1 : 5 --> 4 --> 7 --> 3 --> 2 --> null
The linked list, after deletion of element next to head:
LinkedList 1 : 4 --> 7 --> 3 --> 2 --> null
The searching element present in Linked List at position: 3
```

Time and Space Complexity:

- Insert and delete are local operations and have O*(1)* time complexity. Search requires traversing the entire list, e.g., if the key is at the last node or is absent, so its time complexity is *0(n),* where *n* is the number of nodes.

MERGE TWO SORTED LISTS

- Consider two singly linked lists in which each node holds a number. Assume the lists are sorted, i.e., numbers in the lists appear in ascending order within each list.

- The merge of the two lists is a list consisting of the nodes of the two lists in which numbers appear in ascending order.

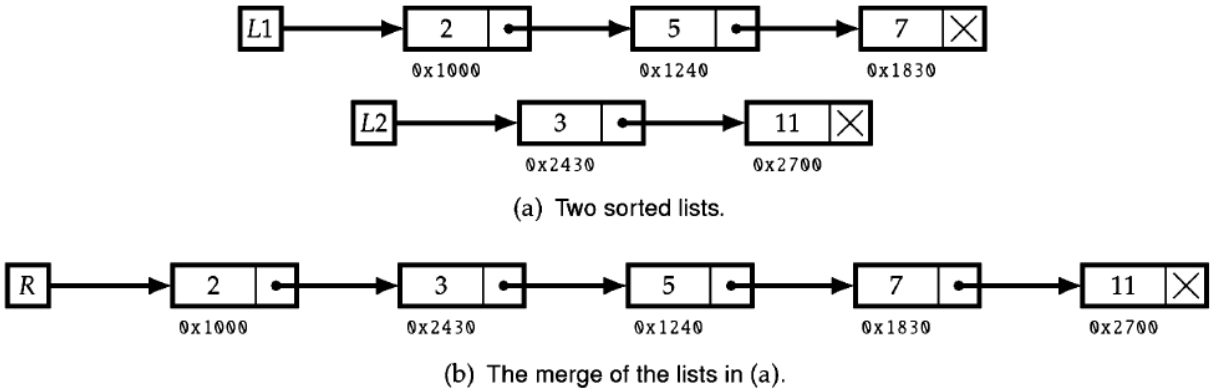


Figure 2: Merging sorted lists.

Write a program that takes two sorted linked lists, and returns their merge. The merge list also a sorted list. The list should be made by splicing together the nodes of the first two lists.

Example:

- Let consider the following two sorted linked list for input to the program

  LinkedList 1: 1 --> 3 --> 5 --> 7 --> null

  LinkedList 2: 2 --> 5 --> 8 --> null

- The output is a merged list:

  LinkedList 3: 1 --> 2 --> 3 --> 5 --> 5 --> 7 --> 8 --> null

Program:

```
package MergeTwoSortedList;

//Linked list Node
public class Node<T>{

	public T data;
	public Node<T> next;

	//Constructor
	Node(T d){

		data = d;
		next = null;
	}
}
```

```
package MergeTwoSortedList;

public class MergeTwoSortedListProg1 {

	// Create a new Linked List node and insert the nodes at first
	public static Node<Integer> insert(int d, Node node){

		Node<Integer> newNode = new Node<Integer>(d);
		newNode.next = node.next;
		node.next = newNode;
		return node;
	}

	//Display the list
	public static void display(Node node) {

		// display the linked list header/number,
		// for example, LinledList 1 or LinledList 2
		System.out.print("LinkedList " + node.data + " : " );

		//Display the data present in the list
		while(node.next != null) {

			System.out.print(node.next.data + " --> ");
```

```
            node = node.next;
        }

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

public static Node<Integer>mergeTwoSortedLists(Node l1, Node l2){

        // Create a dummy node to hang the result on
        Node<Integer> dummy = new Node<>(3);

        // current pointer points to the last result node
        Node<Integer> current = dummy;

        // index p1 point to linked list 1
        Node<Integer> p1 = l1.next;

        // index p2 point to linked list 2
        Node<Integer> p2 = l2.next;

        //traverse the two lists
        while(p1 != null && p2 != null) {

                // Compare the data of the two lists whichever
                // lists' data is smaller, append it into tail
                // and advance the curr pointer to the next Node
                 if(p1.data < p2.data) {

                        current.next = p1;
                        p1 = p1.next;
                }

                else {

                        current.next = p2;
                        p2 = p2.next;
                }
                current = current.next;
        }

        // Appends the remaining nodes of pi or p2.
        if(p1 != null) {

                current.next = p1;
        }
```

```
            else {

                current.next = p2 ;
            }

            return dummy;
        }

        public static void main(String[] args) {

            // create first sorted linked, i.e.,, LinledList 1
            // insert 4 element in the list, ascending order
            Node<Integer> head1=new Node<>(1);
            for(int i = 7; i > 0 ; i-=2) {
                head1=insert(i,head1);
            }
            // Display first linked list
            display(head1);

            // create second sorted linked, i.e.,, LinledList 2
            // insert 3 element in the list, ascending order
            Node<Integer> head2=new Node<>(2);
            for(int j = 8; j > 0 ; j-=3) {
                head2=insert(j,head2);
            }

            // Display second linked list
            display(head2);

            //Create list3
            Node<Integer> head3=new Node<>(3);
            head3 = mergeTwoSortedLists(head1, head2);
            display(head3);
        }
}
```

Output:

```
LinkedList 1 : 1 --> 3 --> 5 --> 7 --> null
LinkedList 2 : 2 --> 5 --> 8 --> null
LinkedList 3 : 1 --> 2 --> 3 --> 5 --> 5 --> 7 --> 8 --> null
```

Time and Space Complexity:

- The worst-case, the time complexity is *0(n + m)*. Since we reuse the existing nodes, the space complexity is *0(1)*.

REVERSE A SINGLE SUBLIST

Program:

```
package ReverseSingleSublist;

//Linked list Node
public class Node<T>{

	public T data;
	public Node<T> next;

	//Constructor
	Node(T d){

		data = d;
		next = null;
	}
}
```

```
package ReverseSingleSublist;

public class ReverseSingleSublistProg1 {

	// Create a new Linked List node and insert the nodes at first
	public static Node<Integer> insert(int d, Node node){

		Node<Integer> newNode = new Node<Integer>(d);
		newNode.next = node.next;
		node.next = newNode;
		return node;
	}

	//Display the list
	public static void display(Node node) {

		// display the linked list header/number,
		// for example, LinledList 1 or LinledList 2
		System.out.print("LinkedList " + node.data + ": " );

		//Display the data present in the list
		while(node.next != null) {

			System.out.print(node.next.data + " --> ");
			node = node.next;
```

```
        }

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


// reverseSublist() method used to reverse a linked list
public static Node<Integer> reverseSublist(Node<Integer> L, int start, int finish) {

        // No need to reverse since start == finish.
        if (start == finish) {

                return L;
        }

        Node<Integer> dummyHead = new Node<>(0);
        dummyHead.next = L;
        Node<Integer> sublistHead = dummyHead;
        int k = 1;
        while (k++ < start) {

                sublistHead = sublistHead.next ;
        }

        // Reverse sublist.
        Node<Integer> sublistIter = sublistHead.next ;

        while (start++ < finish) {

                Node<Integer> temp = sublistIter.next ;
                sublistIter.next = temp.next;
                temp.next = sublistHead.next ;
                sublistHead.next = temp;
        }

        return dummyHead.next ;
}

public static void main(String[] args) {

        // create first sorted linked, i.e.,, LinledList 1
        // insert 4 element in the list, ascending order
        Node<Integer> head1= new Node<>(1);
        for(int i = 14; i > 0 ; i-=2) {
                head1=insert(i,head1);
        }
```

```
                // Display the elements in linked list
                System.out.println("The elements in the linked list: " );
                display(head1);

                // select node 2 to node 5 as sublist and reverse
                System.out.println("After reverring sublist, the elements in the linked list: " );
                reverseSublist(head1.next, 2, 5);
                // Display first linked list
                display(head1);
        }
}
```

Output:

The elements in the linked list:
LinkedList 1: 2 --> 4 --> 6 --> 8 --> 10 --> 12 --> 14 --> null
After reverring sublist, the elements in the linked list:
LinkedList 1: 2 --> 10 --> 8 --> 6 --> 4 --> 12 --> 14 --> null

Time and Space Complexity:

- The time complexity is dominated by the search for the end node (let it's position m in the linked list), hence the time complexity is *0(m).*

DELETE A NODE FROM A SINGLY LINKED LIST

Write a program which deletes a node in a singly linked list. It is guaranteed that the node to be deleted is not a tail node in the list.

Example:

- The elements in the linked list:

  LinkedList 1: 0 --> 1 --> 3 --> 8 --> 5 --> null

- After Deleting kth elements from start of the linked list:

  LinkedList 1: 0 --> 3 --> 8 --> 5 --> null

Approach:

- Instead traverses the linked list from head to find the node to be deleted, direct access to the node to be deleted.

- After access to the node to be deleted, first copy the data from the next node to the node to be deleted and delete the next node.

Program:

```
package DeleteNodeFromLinkedList;

//Linked list Node
public class Node<T>{

	public T data;
	public Node<T> next;

	//Constructor
	Node(T d){

		data = d;
		next = null;
	}
}
```

```
package DeleteNodeFromLinkedList;

import java.util.Random;

public class DeleteNodeFromLinkedListProg1 {

		// Create a new Linked List node and insert the nodes at first
		public static Node<Integer> insert(int d, Node node){

			Node<Integer> newNode = new Node<Integer>(d);
			newNode.next = node.next;
			node.next = newNode;
			return node;
		}

		//Display the list
		public static void display(Node node) {

			// display the linked list header/number,
			// for example, LinledList 1 or LinledList 2
			System.out.print("LinkedList " + node.data + ": " );

			//Display the data present in the list
			while(node.next != null) {
```

```
                System.out.print(node.next.data + " --> ");
                node = node.next;
            }

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

        // Assumes nodeToDelete is not tail.
        public static void deletionFromList(Node<Integer> nodeToDelete) {

            nodeToDelete.data = nodeToDelete.next.data ;
            nodeToDelete.next = nodeToDelete.next.next ;
        }


        public static void main(String[] args) {

            // create instance of Random class
            Random rand = new Random();

            // create linked list, i.e.,, LinledList 1
            // insert 5 element in the list
            Node<Integer> head1= new Node<>(1);
            for(int i = 0; i < 5 ; i++) {

                int num = rand.nextInt(10);
                head1=insert(num,head1);
            }
            // Display the elements in linked list
            System.out.println("The elements in the linked list: " );
            display(head1);

           //Delete kth element
            deletionFromList(head1.next.next);
            // Display the elements in linked list
            System.out.println("After Deleting kth node from start of the linked list:  " );
            display(head1);
        }
}
```

Output:

```
The elements in the linked list:
LinkedList 1: 0 --> 1 --> 3 --> 8 --> 5 --> null
After Deleting kth elements from start of the linked list:
LinkedList 1: 0 --> 3 --> 8 --> 5 --> null
```

Time and Space Complexity:

- The time complexity is O(1).

REMOVE THE Kth LAST ELEMENT FROM A LIST

Given a singly linked list and an integer *k*, write a program to remove the kth last element from the list.

Example:

- Let consider the following linked list and the k = 3, for input to the program

```
LinkedList 1: 8 --> 2 --> 6 --> 5 --> 1 --> null
```

- After remove 3rd element (i.e., 6) from the end of the list, the output

```
LinkedList 1: 8 --> 2 --> 5 --> 1 --> null
```

Approach:

- This approach, use two iterators to traverse the list.
- The first iterator is advanced by *k* steps, and then the second iterators advance in tandem.
- When the first iterator reaches the tail, the second iterator is at the $(k + 1)^{th}$ last node, and then remove the Kth node.

Program:

```
package RemoveKthLastElementFromList;

//Linked list Node
public class Node<T>{

	public T data;
	public Node<T> next;

	//Constructor
	Node(T d){

		data = d;
		next = null;
	}
}
```

```
package RemoveKthLastElementFromList;

import java.util.Random;

public class RemoveKthLastElementFromListProg1 {

	// Create a new Linked List node and insert the nodes at first
	public static Node<Integer> insert(int d, Node node){

		Node<Integer> newNode = new Node<Integer>(d);
		newNode.next = node.next;
		node.next = newNode;
		return node;
	}

	//Display the list
	public static void display(Node node) {

		// display the linked list header/number,
		// for example, LinledList 1 or LinledList 2
		System.out.print("LinkedList " + node.data + ": " );

		//Display the data present in the list
		while(node.next != null) {

			System.out.print(node.next.data + " --> ");
			node = node.next;
		}

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

	// Assumes L has at least k nodes, deletes the k-th last node in L.
	public static Node<Integer> removeKthLast(Node <Integer> L, int k) {

		Node <Integer> dummyHead = new Node<>(0);
		dummyHead.next = L;

		Node <Integer> first = dummyHead.next ;

		while (k-- > 0) {

			first = first.next;
		}
```

```
        Node <Integer> second = dummyHead;

        while (first != null) {

            second = second. next;
            first = first.next;
        }

        // second points to the (k + l)-th last node, deletes its successor.
        second.next = second.next.next ;
        return dummyHead.next ;
    }


    public static void main(String[] args) {

        // create instance of Random class
        Random rand = new Random();

        // create linked list, i.e.,, LinledList 1
        // insert 5 element in the list.
        Node<Integer> head1= new Node<>(1);
        for(int i = 0; i < 5 ; i++) {

            int num = rand.nextInt(10);
            head1=insert(num,head1);
        }

        // Display the elements in linked list
        System.out.println("The elements in the linked list: " );
        display(head1);


        System.out.println("After removing kth elements from end of the linked list: " );
        removeKthLast(head1.next, 3);
        // Display the linked list
        display(head1);
    }
}
```

Output:

The elements in the linked list:
LinkedList 1: 8 --> 2 --> 6 --> 5 --> 1 --> null
After removing kth elements from end of the linked list:
LinkedList 1: 8 --> 2 --> 5 --> 1 --> null

Time and Space Complexity:

- The time complexity is that of list traversal, i.e.,*O(n)*,where *n* is the length of the list. The space complexity is*0(1)*,since there are only two iterators.

REMOVE DUPLICATES FROM A SORTED LIST

- This problem is concerned with removing duplicates from a sorted list of integers.

Write a program that takes as input a singly linked list of integers in sorted order, and removes duplicates from it. The list should be sorted.

- Let the input the sorted linked list:

  LinkedList 1 : 2 --> 5 --> 7 --> 7 --> 11 --> null

- The output is deletion of the duplicate elements from the linked list:

  LinkedList 1 : 2 --> 5 --> 7 --> 11 --> null

Approach:

- Exploit the sorted nature of the list.
- As traverse the list, remove all successive nodes with the same value as the current node.

Program:

```
//Linked list Node
public class Node<T>{

	public T data;
	public Node<T> next;

	//Constructor
	Node(T d){

		data = d;
		next = null;
	}
}
```

```
public class RemoveDuplicatesFromSortedListProg1 {

    // Create a new Linked List node and insert the nodes at first
    public static Node<Integer> insert(int d, Node node){

        Node<Integer> newNode = new Node<Integer>(d);
        newNode.next = node.next;
        node.next = newNode;
        return node;
    }

    //Display the list
    public static void display(Node node) {

        // display the linked list header/number,
        // for example, LinledList 1 or LinledList 2
        System.out.print("LinkedList " + node.data + " : " );

        //Display the data present in the list
        while(node.next != null) {

            System.out.print(node.next.data + " --> ");
            node = node.next;
        }

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



    public static Node <Integer> removeDuplicates (Node <Integer> L) {

        Node <Integer> iter = L;

        while (iter != null) {

            // Uses nextDistinct to find the next distinct value.
            Node <Integer> nextDistinct = iter. next;

            while (nextDistinct != null && nextDistinct . data == iter.data) {

                nextDistinct = nextDistinct . next ;
            }

            iter.next = nextDistinct ;
            iter = nextDistinct;
```

```
            }

            return L;
        }

        public static void main(String[] args) {

            // create sorted linked, i.e.,, LinledList 1
            // insert 5 element in the list, ascending order
            Node<Integer> head1=new Node<>(1);
            head1=insert(11, head1);
            head1=insert(7, head1);
            head1=insert(7, head1);
            head1=insert(5, head1);
            head1=insert(2, head1);

            System.out.println("The elements in linked list: ");
            // Display linked list
            display(head1);

           System.out.println("After deletion of the duplicate elements from the linked list: ");
           removeDuplicates(head1);
           // Display linked list
           display(head1);
        }
}
```

Output:

The elements in linked list:
LinkedList 1 : 2 --> 5 --> 7 --> 7 --> 11 --> null
After deletion of the duplicate elements from the ascending order sorted linked list:
LinkedList 1 : 2 --> 5 --> 7 --> 11 --> null

Time and Space Complexity:

- The linked list is traversed once, hence the time complexity is O*(n)*. The space complexity is O*(1)*.

TEST WHETHER A SINGLY LINKED LIST IS PALINDROMIC

- A palindromic list is the one which is equivalent to the reverse of itself.

- The list given in the below figure is a palindrome since it is equivalent to its reverse list, i.e., 2, 3, 5, 3, 2.

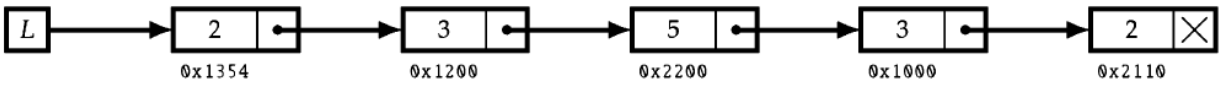


Figure 3: Palindromic List

Write a program that tests whether a singly linked list is palindromic.

Example:

Approach:

- First find the middle of the linked list.
    - Traverse the linked list by using two pointers, i.e., fast and slow.
    - Move fast pointer by one step and the slow pointer by two.
    - When the fast pointer reaches the end slow pointer will reach the middle of the linked list.
    - Now the list is divided into two parts or sub list, i.e., first half and second half.
- Reverse the second half of the list
- If the first half and the reversed second half are equal, then the linked list is palindrome otherwise not palindrome.
- To restore the original list reverse the reversed second half.

Stacks

- A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle.
- Stack has one end
    - Whenever an element is added in the stack, it is added on the top of the stack.
    - Whenever an element is removed from the stack, it is removed from the top of the stack.
- A stack can be implemented by means of Array and Linked List.
    - You can implement stack can either be a fixed size (use Array) or Stack can either be a dynamic size (Linked List).
- Some Basic Operations of Stacks:
    - push():
        - When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
        - Some implementing classes may throw an IllegalStateException if the capacity limit is exceeded or a NullPointerException if the element being inserted is null.
        - To avoid the IllegalStateException, first test with isfull().
            - isFull(): It determines whether the stack is full or not.
        - LinkedList has no capacity limit and allows for null entries, though as we will see you should be very careful when adding null.
    - pop():
        - When we delete an element from the stack, the operation is known as a pop.
        - It will remove and return the element at the top of the stack.

- If the stack is empty means that no element exists in the stack, this state is known as an underflow state.

- Some implementing classes may throw a NoSuchElementException if the deque is empty. To avoid the exception, first test with isEmpty().

    - isEmpty(): It determines whether the stack is empty or not.

- peek():

    - It will retrieve the element at the top of the stack, but does not remove.

    - If the stack is empty, it returns null. Since null may be a valid entry, this leads to ambiguity. Therefore a better test for an empty stack is isEmpty().

- When the stack is implemented using a linked list, both push and pop operations have 0(1) time complexity. If it is implemented using an array, both push and pop operations have still 0(1) time complexity.

IMPLEMENT A STACK WITH MAX API

Design a stack that includes a max operation, in addition to push and pop. The max method should return the maximum value stored in the stack.

Approach:

- Use additional storage to track the maximum value.

- Suppose we use a single auxiliary variable, M, to record the element that is maximum in the stack. Updating M on pushes is easy: M = max(M,e), where e is the element being pushed.

- However, updating M on pop is very time consuming. If M is the element being popped, we have no way of knowing what the maximum remaining element is, and are forced to consider all the remaining elements.

- To overcome this problem, for each entry in the stack, we cache the maximum stored at or below that entry. Now when we pop, we evict the corresponding cached value.

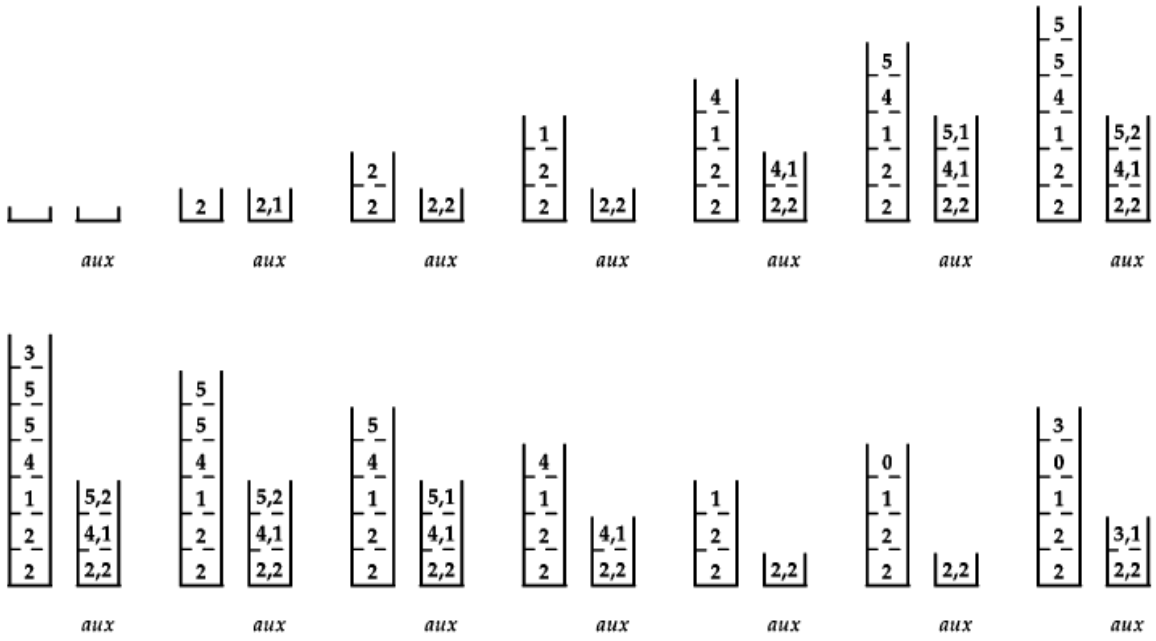


Figure 9.2: The primary and auxiliary stacks for the following operations: push 2, push 2, push 1, push 4, push 5, push 5, push 3, pop, pop, pop, pop, push 0, push 3. Both stacks are initially empty, and their progression is shown from left-to-right, then top-to-bottom. The top of the auxiliary stack holds the maximum element in the stack, and the number of times that element occurs in the stack. The auxiliary stack is denoted by *aux*.

Program:

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

public class StackWithMax {

    // @include
    private static class ElementWithCachedMax {

        public Integer element;
        public Integer max;

        public ElementWithCachedMax(Integer element, Integer max) {

            this.element = element;
            this.max = max;
        }
    }
```

```
public static class Stack {

        // Stores (element, cached maximum) pair.
        private Deque<ElementWithCachedMax> elementWithCachedMax
                                                        =  new LinkedList<>();

        public boolean empty() { return elementWithCachedMax.isEmpty(); }

        public Integer max() {

                if (empty()) {

                        throw new IllegalStateException("max(): empty stack");
                }
                return elementWithCachedMax.peek().max;
        }

        public Integer pop() {

                if (empty()) {

                        throw new IllegalStateException("pop(): empty stack");
                }

                return elementWithCachedMax.removeFirst().element;
        }

        public void push(Integer x) {

                elementWithCachedMax.addFirst (new
                        ElementWithCachedMax(x, Math.max(x, empty() ? x : max())));
        }
}
// @exclude

public static void check(boolean condition, String msg) {

        if (!condition) {

                System.err.println(msg);
                System.exit(-1);
        }
}

public static void missedMaxException() {
```

```
        System.err.println("Should have seen an exception, max() on empty stack!");
        System.exit(-1);
}

public static void missedPopException() {

        System.err.println("Should have seen an exception, pop() on empty stack!");
        System.exit(-1);
}

public static void main(String[] args) {

        Stack s = new Stack();
        s.push(1);
        s.push(2);
        check(s.max() == 2, "failed max() call with stack created by push 1, push 2");
        System.out.println(s.max()); // 2
        System.out.println(s.pop()); // 2
        check(s.max() == 1, "failed max() call with stack created by push 1, push 2,
                                                                                pop");
        System.out.println(s.max()); // 1
        s.push(3);
        s.push(2);
        check(s.max() == 3, "failed max() call with stack created by push 1, push 2, pop,
                                                                    push 3, push 2");
        System.out.println(s.max()); // 3
        s.pop();
        check(s.max() == 3, "failed max() call with stack created by push 1, push 2, pop,
                                                                push 3, push 2, pop");
        System.out.println(s.max()); // 3


        s.pop();
        check(s.max() == 1,  "failed max() call with stack created by push 1, push 2, pop,
                                                                push 3, push 2, pop, pop");
        System.out.println(s.max()); // 1
        s.pop();

        try {

                s.max();
                missedMaxException();

        } catch (RuntimeException e) {
```

```
                System.out.println("Got expected exception calling max() on an empty
                                                                stack:" + e.getMessage());
            }

            try {

                s.pop();
                missedPopException();

            } catch (RuntimeException e) {

                System.out.println("Got expected exception calling pop() on an empty
                                                                stack:" + e.getMessage());
            }
        }
}
```

Output:

```
2
2
1
3
3
1
Got expected exception calling max() on an empty stack:max(): empty stack
Got expected exception calling pop() on an empty stack:pop(): empty stack
```

Time and Space Complexity:

- The time complexity for each specified method is still *0(1).*
- In the best-case, the additional space complexity is less, *0(1)*, which occurs when the number of distinct keys is small, or the maximum changes infrequently.
- The worst-case additional space complexity is *0(n),* which occurs when each key pushed is greater than all keys in the primary stack.

EVALUATE Reverse Polish Notation (RPN) EXPRESSIONS

- Reverse Polish Notation is postfix notation which in terms of mathematical notion signify the operators following operands.

Write a program that takes an arithmetical expression in RPN and returns the number that the expression evaluates to.

Example:

- Let an arithmetical expression in RPN is inputted as string, i.e., “3,4,+,2,*,1,+”
- After evaluation of RPN arithmetical expression, the output is 15

Approach:

The basic approach for the problem is using the **stack.**

- Accessing all elements in the array, if the element is not matching with the special character (‘+’, ‘-‘,’*’, ‘/’) then push the element to the stack.
- Then whenever the special character is found then pop the first two-element from the stack and perform the action and then push the element to stack again.
- Repeat the above two process to all elements in the array.
- At last pop the element from the stack and print the Result.

Program:

```
import java.util.Deque;
import java.util.LinkedList;
import java.util.Scanner;

public class EvaluateRPNExpressionsProg1 {

	public static void main(String[] args) {

		Scanner sc = new Scanner(System.in);
		System.out.print("Enter an arithmetical expression in RPN, each character
                                                    separated by comma (,) : ");
		String str = sc.nextLine();

		int result = eval(str);

		System.out.print("The evaluation result of RPN arithmetical expression : " +
                                                                    result);
	}

	public static int eval(String RPNExpression) {

		Deque<Integer> intermediateResults = new LinkedList<>();

		String delimiter = ",";
		String[] symbols = RPNExpression.split(delimiter);
```

```
            for (String token : symbols) {

                if (token.length() == 1 && "/*+-".contains(token)){

                    final int y = intermediateResults.removeFirst();
                    final int x = intermediateResults.removeFirst();

                    switch (token.charAt(0)){

                        case '+':

                            intermediateResults.addFirst(x + y);
                            break ;

                        case '-':

                            intermediateResults.addFirst(x - y);
                            break ;

                        case '*':

                            intermediateResults.addFirst(x * y);
                            break ;

                        case '/':

                            intermediateResults.addFirst(x / y);
                            break ;

                        default:

                            throw new IllegalArgumentException("Malformed
                                                                RPN at :" + token);
                    }

                } else { // token is a number.

                    intermediateResults.addFirst(Integer.parseInt(token));

                }
            }

            return intermediateResults.removeFirst();
        }
}
```

Output:

Enter an arithmetical expression in RPN, each character separated by comma (,) : 3,4,+,2,*,1,+
The evaluation result of RPN arithmetical expression : 15

Time and Space Complexity:

- Since we perform *0(1)* computation per character of the string, the time complexity is *0(n)*, where *n* is the length of the string. The space complexity is O(1).

Queues

- A Queue is a linear structure that follows the First In First Out (FIFO) principle, , i.e., the data item stored first will be accessed first.

- A queue is open at both its ends :

    - One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).

- A queue supports two basic operations:

    - **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

    - **Dequeue:** Removes an item from the queue. If the queue is empty, then it is said to be an Underflow condition. If the queue is empty, dequeue typically returns null or throws an exception.

    - Elements are added (enqueued) and removed (dequeued) in first-in, first-out order.

- The most recently inserted element is referred to as the tail or back element, and the item that was inserted least recently is referred to as the head or front element.

- A Queue can be implemented by means of Array and Linked List.

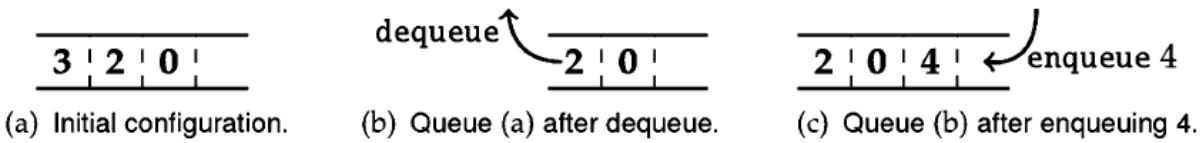


Figure 9.4: Examples of enqueuing and dequeuing.

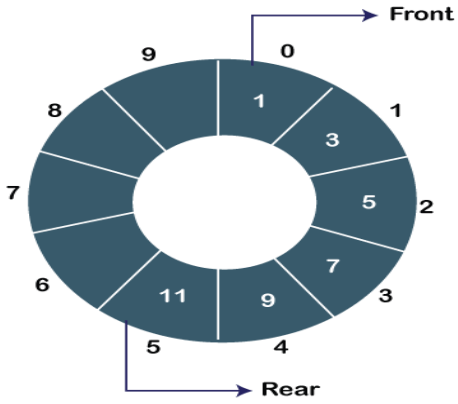
- The preferred way to manipulate queues is via the Deque interface.

- The LinkedList class is a doubly linked list that implements this interface, and provides efficient (0(1) time) queue (and stack) functionality.

- The key queue-related methods in the Deque interface are addLast(3.14), removeFirst(), getFirst().
    - addLast (e):
        - addLast (e) enqueues an element. Some classes implementing Deque have capacity limits and/or preclude null from being enqueued, but this is not the case for LinkedList.
    - removeFirst ()
        - removeFirst() retrieves and removes the first element of this deque, throwing NoSuchElementException if the deque is empty.
    - getFirst()
        - getFirst() retrieves, but does not remove, the first element of this deque, throwing NoSuchElementException if the deque is empty.
- **The time complexity**
    - The time complexity of enqueue and dequeue are O(1).
    - The time complexity of finding the maximum is 0(n), where n is the number of entries.

IMPLEMENT A CIRCULAR QUEUE

- A queue can be implemented using an array and two additional fields, the beginning/front and the end/rear indices.
- In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element.
- The circular queue can be represented as:



Implement a queue API using an array for storing elements. Your API should include a constructor function, which takes as argument the initial capacity of the queue, enqueue and dequeue functions, and a function which returns the number of elements stored. Implement dynamic resizing to support storing an arbitrarily large number of elements.

Program:

```
import java.util.Arrays;
import java.util.Collections;
import java.util.NoSuchElementException;

public class CircularQueue {

    // @include
    public static class Queue {

        private int head = 0, tail = 0, numQueueElements = 0;
        private static final int SCALE_FACTOR = 2;
        private Integer[] entries;

        public Queue(int capacity) { entries = new Integer[capacity]; }

        public void enqueue(Integer x) {

            if (numQueueElements == entries.length) { // Need to resize.

                // Makes the queue elements appear consecutively.
                Collections.rotate(Arrays.asList(entries), -head);
                // Resets head and tail.
                head = 0;
                tail = numQueueElements;
                entries = Arrays.copyOf(entries, numQueueElements *
                                                        SCALE_FACTOR);
            }
```

```
                entries[tail] = x;
                tail = (tail + 1) % entries.length;
                ++numQueueElements;
            }

            public Integer dequeue() {

                if (numQueueElements != 0) {
                --numQueueElements;
                Integer ret = entries[head];
                head = (head + 1) % entries.length;
                return ret;
                }
                throw new NoSuchElementException("Dequeue called on an empty
                                                                    queue.");
            }

            public int size() { return numQueueElements; }

        }  // @exclude


        private static void assertDequeue(Queue q, Integer t) {

            Integer dequeue = q.dequeue();
            assert(t.equals(dequeue));
        }

        public static void main(String[] args) {

            Queue q = new Queue(5);
            q.enqueue(1);
            q.enqueue(2);
            q.enqueue(3);

            int totalele = q.size();
            System.out.println("The total elements in Circular queue: " + totalele);

            q.enqueue(4);
            q.enqueue(5);
            q.enqueue(6);
            int totalele1= q.size();
            System.out.println("The total elements in Circular queue: " + totalele1);
        }
}
```

Output:

The total elements in Circular queue: 3
The total elements in Circular queue: 6

Time and Space complexity:

- The time complexity of dequeue is *0(1)*, and the amortized time complexity of enqueue is *0(1).*

IMPLEMENT A QUEUE USING STACKS

- Queue is a linear data structure that follows FIFO (First In First Out) principle in which insertion is performed from the rear end and the deletion is done from the front end.

- Stack is a linear data structure that follows LIFO (Last In First Out) principle in which both insertion and deletion are performed from the top of the stack.

Write a program to implement a Queue using Stacks.

Approach:

- A queue can be implemented using two stacks.

- Use the first stack for enqueue and the second for dequeue.

Program:

```
import java.util.Deque;
import java.util.LinkedList;
import java.util.NoSuchElementException;

public class ImplementQueueUsingStacks {

    public static class Queue {

        private Deque <Integer> enq = new LinkedList<>();
        private Deque <Integer> deq = new LinkedList<>();

        public void enqueue(Integer x) { enq.addFirst(x); }

        public Integer dequeue() {

            if (deq.isEmpty()){

                // Transfers the elements from enq to deq.
```

```
                while (!enq.isEmpty()){

                    deq.addFirst(enq.removeFirst());
                }
            }

            if (!deq.isEmpty()){

                return deq.removeFirst();
            }

            throw new NoSuchElementException("Cannot pop empty queue");
        }
    }

    // Driver code
    public static void main(String[] args) {

        Queue q = new Queue();

        q.enqueue(1);
        q.enqueue(2);
        q.enqueue(3);

        System.out.println(q.dequeue());
        System.out.println(q.dequeue());
        System.out.println(q.dequeue());
        //System.out.println(q.dequeue());
    }
}
```

Output:

```
1
2
3
```

Time and Space complexity:

- This approach takes *0(m)* time for *m* operations, which can be seen from the fact that each element is pushed no more than twice and popped no more than twice.

IMPLEMENT A QUEUE WITH MAX API

Implement a queue with enqueue, dequeue, and max operations. The max operation returns the maximum element currently stored in the queue.

Approach:

- Maintain two queues one for entries, i.e., originalQueue and another for max candidate, i.e., max_queue
- In the max_queue, head will be pointing to the current max element.
- When an element is enqueued to the originalQueue
    - Check the tail of max_queue, if tail is greater than element, then add the element at tail.
    - If tail is less than element, then remove from tail iteratively until max_queue is empty or tail element is greater than or equal to the element, then add the element at tail.
- When an element is dequeued from originalQueue
    - Check the element is same as max_queue's head or not, if it is same then remove the max_queue's head, otherwise the max_queue remains unchanged
- Note: max_queue must contain elements in decreasing order.

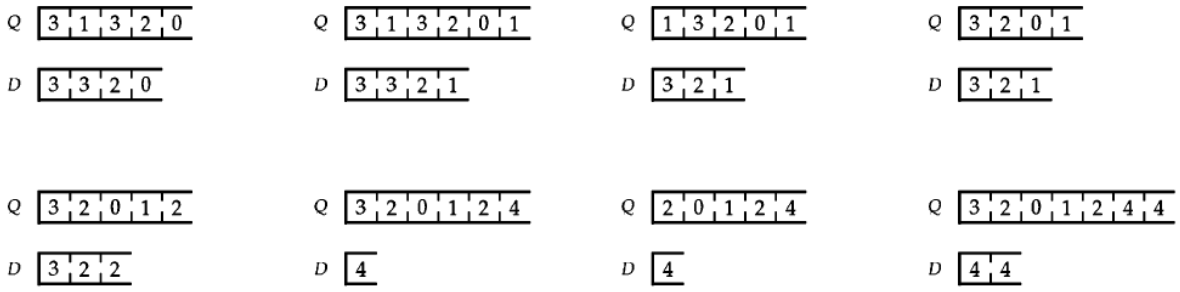


Figure 9.5: The queue with max for the following operations: enqueue 1, dequeue, dequeue, enqueue 2, enqueue 4, dequeue, enqueue 4. The queue initially contains $3, 1, 3, 2$, and 0 in that order. The deque D corresponding to queue Q is immediately below Q. The progression is shown from left-to-right, then top-to-bottom. The head of each queue and deque is on the left. Observe how the head of the deque holds the maximum element in the queue.

Program:

```
import java.util.Deque;
import java.util.LinkedList;
import java.util.NoSuchElementException;
import java.util.Queue;

public class QueueWithMax <T extends Comparable<T>> {

	//Maintain two queues one for entries and another for max candidate
	private Queue<T> entries = new LinkedList<>();
	private Deque<T> candidateForMax = new LinkedList<>();

	/* While inserting new element in the queue, check if the newly added
	* element is bigger than the last element on the candidateForMax if
	* yes remove it all small elements are removed then add the new
	* element to candidateForMax queue
	*/

	public void enqueue(T x){

		entries.add(x);

		while (!candidateForMax.isEmpty()){

			if(candidateForMax.getLast().compareTo(x) >= 0) {

				break;
			}
			candidateForMax.removeLast();
		}
		candidateForMax.addLast(x);
	}


	// Remove a entry from entries queue and check if its same as that of
	// the max candidate if yes remove that entry from max candidate as well
	public T dequeue(){

		if(!entries.isEmpty()){

			T result  = entries.remove();

			if(result.equals(candidateForMax.getFirst())){

					candidateForMax.removeFirst();
```

```
                }
                return result;
        }
        throw new NoSuchElementException("Called dequeue on empty row");
}

public T max(){

        if(!candidateForMax.isEmpty()) {

                return candidateForMax.getFirst();
        }
        throw new NoSuchElementException("Called dequeue on empty row");
}

//Display the original Queue elements
public void displayEntries() {

        System.out.print("The Queue elements are: ");

        for(T Qelement : entries) {

                System.out.print(Qelement + " ");
        }
        System.out.println(" ");
}


//Display the MaxQueue candidate for Max elements
public void displaycandidateForMax() {

        System.out.print("The candidate for Max elements are: ");

        for(T maxelement : candidateForMax) {

                System.out.print(maxelement + " ");
        }
        System.out.println(" ");
}


// Driver code
public static void main(String[] args) {

        QueueWithMax q = new QueueWithMax();
```

```
            //The queue initially contains 3,1,3,2, and 0 in that order.
            q.enqueue(3);
            q.enqueue(1);
            q.enqueue(3);
            q.enqueue(2);
            q.enqueue(0);

            //Display the original Queue elements
            q.displayEntries();
            //Display the MaxQueue candidate for Max elements
            q.displaycandidateForMax();
            System.out.println("The max element in queue is " + q.max());

            q.enqueue(1);
            q.dequeue();
            q.dequeue();
            q.enqueue(2);
            q.enqueue(4);
            q.dequeue();
            q.enqueue(4);
            //Display the original Queue elements
            q.displayEntries();
            //Display the MaxQueue candidate for Max elements
            q.displaycandidateForMax();
            System.out.println("The max element in queue is " + q.max());
        }
}
```

Output:

```
The Queue elements are: 3 1 3 2 0
The candidate for Max elements are: 3 3 2 0
The max element in queue is 3
The Queue elements are: 2 0 1 2 4 4
The candidate for Max elements are: 4 4
The max element in queue is 4
```

Time and Space Complexity:

- Each dequeue operation has time *0(1)* complexity. A single enqueue operation may entail many ejections from the max_queue. However, the amortized time complexity of *n* enqueues and dequeues is *0(n),* since an element can be added and removed from the max_queue no more than once. The max operation is *0(1)* since it consists of returning the element at the head of the max_queue.