1 A. Factorial using recursion.

Code attached in file “**QN_1A_Factorial.JAVA**”.
Input N=6.

Recursive Stack operations:

- The function “static int fact()” is called with N=6 initially. (fact(6)). Since the condition “if n<=1” Is not satisfied, the “return n*fact(n-1)” executes as “return 6*fact(5)”. The call stack currently holds the number ‘6’ and is waiting for a return value from fact(5).

- Fact(5) will be called and as the condition “if n<=1” Is not satisfied, the “ return n*fact(n-1)” executes as “return 5*fact(4)”. The call stack currently holds the number ‘5’ and is waiting for a return value from fact(4).

- Fact(4) will be called and as the condition “if n<=1” Is not satisfied, the “ return n*fact(n-1)” executes as “return 4*fact(3)”. The call stack currently holds the number ‘4’ and is waiting for a return value from fact(3).

- Fact(3) will be called and as the condition “if n<=1” Is not satisfied, the “ return n*fact(n-1)” executes as return 3*fact(2). The call stack currently holds the number ‘3’ and is waiting for a return value from fact(2).

- Fact(2) will be called and as the condition “if n<=1” Is not satisfied, the “ return n*fact(n-1)” executes as return 2*fact(1). The call stack currently holds the number ‘2’ and is waiting for a return value from fact(1).

- Fact(1) will be called and as the condition “if n<=1” is satisfied, it returns 1 to the caller fact(1).

- Fact(1) computes 1*1 and returns the answer 1 to fact(2).

- Fact(2) computes 2*1 and returns the answer 2 to the caller fact(3).

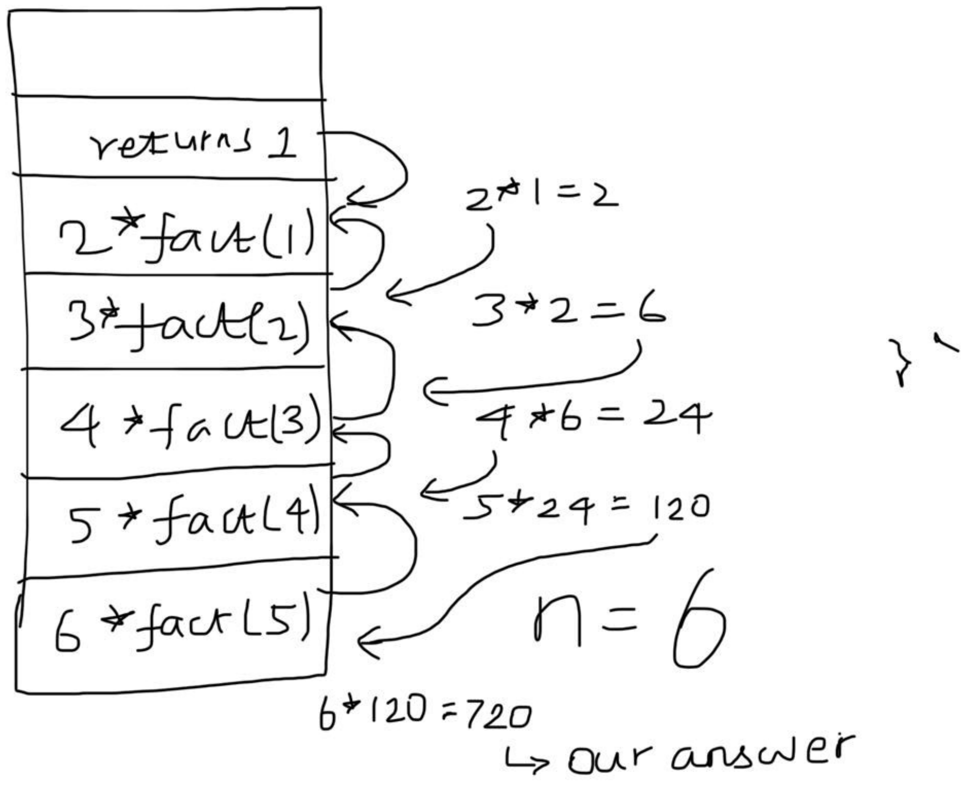
- Fact(3) computes 3*2 and returns the answer 6 to the caller fact(4).

- Fact(4) computes 4*6 and returns the answer 24 to the caller fact(5)

- Fact(5) computes 5*24 and returns the answer 120 to the caller fact(6).

- Fact(6) computes 6*120 and returns the answer 720 to the caller fact(6).

A pictorial representation of the call stack is attached below:



The overall Time complexity is ~ O(N) which is ~ O(6) in our case.

The time complexity for one recursive call would be:

T(n) = T(n-1) + 3 (Assuming that for one arithmetic operation it takes 1ms, we're adding a consent value 3 because we're performing one multiplication, one subtraction and one comparison operation at every function call.)
T(6) = T(5) + 3
T(5) = T(4) + 6
T(4) = T(3) + 9
T(3) = T(2) + 12
T(2) = T(1) + 15
T(1) returns 1 ……. END of recursion.

1 B. Fibonacci series using recursion.

Code attached in file **"QN_1B_Fibonacci.JAVA"**

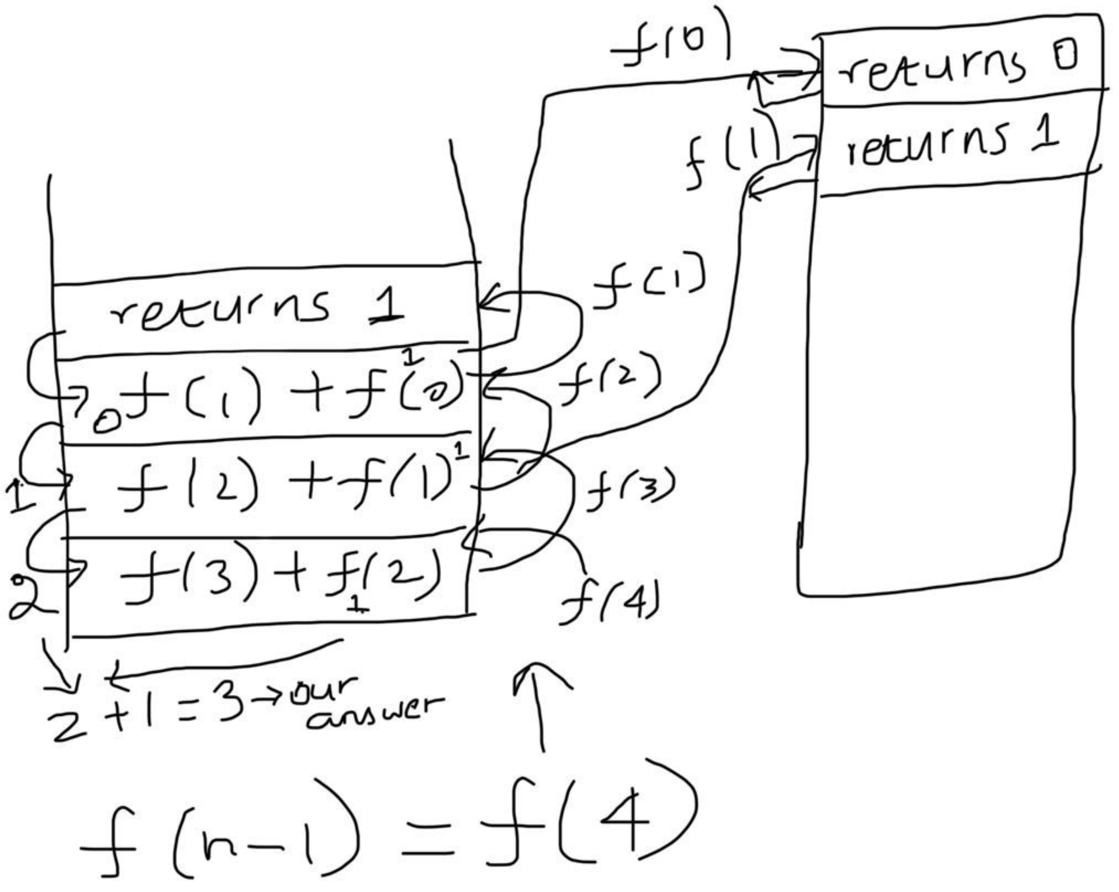
The Fibonacci sequence for N=5 is: **0, 1, 1, 2, 3.**

Therefore the N^{th} (i.e., 5^{th}) Fibonacci number is **3.**

Recursive stack operations:

- The function "static int fibonacci_recursion(n-1)" is called with N=5 initially. (fibonacci_recursion (4)).
- Since the condition "if n<=1" Is not satisfied, the "*return fibonacci_recursion(n-1) + fibonacci_recursion(n-2)"* executes as *"return fibonacci_recursion(3) + fibonacci_recursion(2);"*. The control transfers to *fibonacci_recursion(3)* .
- Since the condition "if n<=1" Is not satisfied, the "*return fibonacci_recursion(n-1) + fibonacci_recursion(n-2)"* executes as *"return fibonacci_recursion(2) + fibonacci_recursion(1);"*. The control transfers to *fibonacci_recursion(2)* .
- Since the condition "if n<=1" Is not satisfied, the "*return fibonacci_recursion(n-1) + fibonacci_recursion(n-2)"* executes as *"return fibonacci_recursion(1) + fibonacci_recursion(0);"*.
- The control transfers to *fibonacci_recursion(1)* . Since the condition "if n<=1" Is satisfied, the "return n" statement executes as "return 1".
- '1' is returned to the caller *fibonacci_recursion(1)*. Now the function proceeds to execute "*fibonacci_recursion(0)"*. This will return the value '0' as the "if" condition is satisfied. Now the "*fibonacci_recursion(2)"* computes 1+0 =1 and returns the value '1' to the caller "*fibonacci_recursion(3)"*.
- The function *fibonacci_recursion(3)* has the value '1' in the stack and proceeds to execute *fibonacci_recursion(1)*. It returns 1. Then this function computes 1+1 =2 and returns '2' to the caller *fibonacci_recursion(4)*.
- This function *fibonacci_recursion(4)* then proceeds to execute *fibonacci_recursion(2)* and it returns 1 (as we have seen in the above scenario). Now the *fibonacci_recursion(4)* computes 2+1 =3 and returns the answer 3 to the caller. We've arrived at our result "5^{th} fibonacci number is 3".

A pictorial representation of the call stack is attached below:



Iterative Algorithm:

1. START
2. Create a method STATIC INT FIBONACCI_ITERATIVE(INT N) and INITIALIZE TWO VARIBALES A and B with values -1 and 0 respectively (inside the body of the method).
3. Provide an IF CONDITION such that it RETURNS 'N' IF N IS LESS THAN OR EQUAL TO 1.
4. Write a "for" loop from i=2 to n and perform the following inside the loop.
   4.1 Add A and B and store it in C. (c=a+b)
   4.2 Assign A=B and B=C.
5. After the end of the loop, return C. (which is our Nth Fibonacci number).

1 c. Tower of Hanoi

Code attached in file **“QN_1C_TowersOfHanoi.JAVA”**

Working of the algorithm:

Input: N=5

1. The method move() is called with parameters n=5, A, C, B representing the number of rods and the name of the rods respectively. (move(5,A,C,B))
2. The function calls itself recursively with N value reduced by 1 each time until the condition n==1 is met.
3. After N equals 1 the message, “Move disk 1 from rod A to rod C” is printed illustrating that the smallest disk is moved from A to C.
4. Then the function returns from the call and prints the message “Move disk 2 from rod A to rod B”, illustrating that the second smallest disk is moved from rod A to rod B. Now the smallest disk is at rod C and the second smallest disk is at rod B.
5. A recursive call is made with N-1 as the parameter (with N==1 for representing the smallest disk) and the message “Move disk 1 from rod C to rod B” illustrating that the smallest disk is moved from rod C to rod B and placed over the second smallest disk (Disk 2).
6. The steps 2 to 5 will be repeated until the rods are transferred one by one to the rod C and the arrangement such that all the smaller disks placed over the larger disk is obtained.

Data structures used:

The character variables used are rodA, rodB and rodC. They are used to represent the three rods A, B, C.

Integer variable N is used to capture input, calling the function recursively by reducing n each time by 1 (n-1) and specifying the exit condition for the recursion.

2. Code attached in file “TestLinkedListQueue.JAVA”

3. Code attached in file “TestArrayQueue.JAVA”

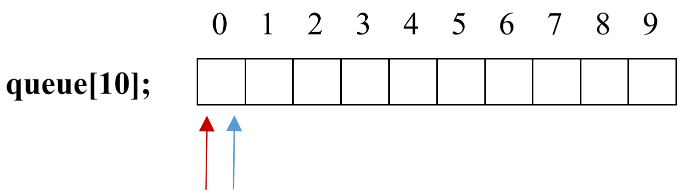
Illustration:

↑ The red arrow represents the head pointer.

↑ The blue arrow represents the tail pointer.

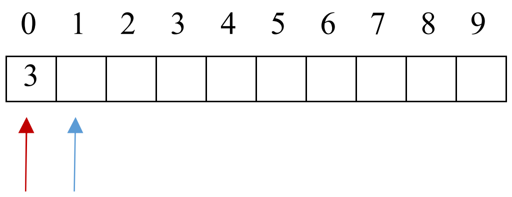
Initially they point to zero.

An Empty Queue of size 10 looks like this:

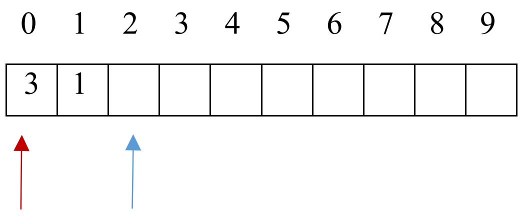


i) **Enqueue**:

Suppose if we Enqueue the element '3' the element will be inserted in the position quque[tail] (where tail =0, in our case). The tail will be incremented by 1. Current tail value =1.

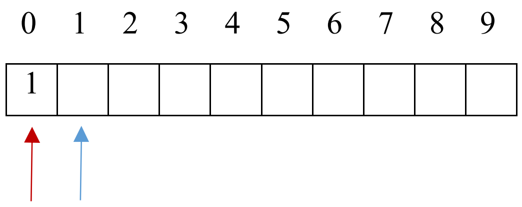


Let us enqueue another element "1". It will be inserted in the position queue[tail] (where tail =0, in our case). The tail will be incremented by 1. Current tail value =2.

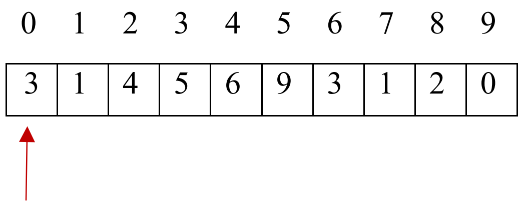


ii) **Dequeue**:

If we call the dequeue() method , the front element 3 is removed from the queue and all the values are left shifted by 1 position. Then the tail pointer is decremented by 1.

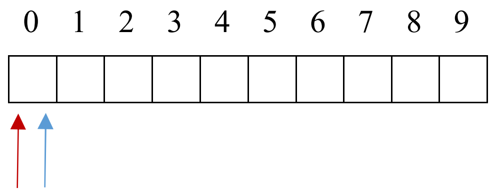


iii) **isFull()**
If the tail pointer equals the size of the queue, it returns true, else it returns false.



iv) isEmpty()

If the head pointer equals the tail pointer, it returns true, else it returns false.



4 A. Difference between equals() , compareTo() and hashCode()

- The "**equals()**" method compared the reference of two objects and returns the Boolean value "**True**" if both the references are same, else it returns "**False**". The **equals()** methods does not compare the values of the data items contained in each object. It simply checks if the object references are same. This type of comparison is known as **Shallow Comparison**. However, a class can define its own **equals()** method (by overriding the default equals() method) to compare the value of the data items of two different objects. This type of comparison is known as **Deep Comparison**. It is recommended to override hashCode() method also , if we override equals() method. This will make sure that the contract between equals() and hashCode() is not violated. **Syntax: Object_1.equals(Object_2);**

- The "**hashCode**()" method returns the integer hash code value of the object that is passed as the argument. If the references of two objects are the same, then the hashCode() method must return the same integer value for both the objects. However, even when both the object do not hold the same reference, the hashCode() method might sometimes produce the same integer value. i.e., It need not produce a unique hashcode value for different object references. This is because the concept of hashing allows more than 1 object to be placed on the same bucket and hence if both the objects are placed in the same bucket, it would return the same integer value for both.
Syntax: int h = sample_object.hashCode();

- The **"compareTo()"** method from the comparable interface compares values and returns an integer value. If the integer value is **positive** it means that the current instance is greater than the one in the argument. If it is **negative**, it means that the current instance is lesser that the instance passed as the argument. If the integer value is **zero**, it means that the two instances are equal. Moreover, using the comparable interface and the comparator, the objects can be compared along with specific properties of the objects. This is known as deep comparison.
Syntax: int ans = obj_1.compareTo(obj_2);

4 B.

The String class's hashCode() method computes hashCode by the following method:

hash=s.charAt(0)*31^n-1+s.charAt(1)*31^n-2+.+s.charAt(n-1)*31^n-n

Print (hash)

Where **S** is the input String and **n** is the length of the String. Java's hashCode() method might return a negative integer. If a string is long enough (like our Input String = "Hello to the World "). The resulting hashcode will be bigger than the largest integer we can store on 32 bits CPU. In this case, due to integer overflow, the value returned by hashCode can be negative.

- **Computing hash value by java hashCode() method from String class:**
  String s = "Hello to the World"
  System.out.print(s.hashCode());

  The method s.hashCode() returns : -9202612

  Java code snippet:

```
String s = "Hello to the World"

int len = s.length();

int n=len;

int hash=0;

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

hash=hash+(s.charAt(i)*(int)Math.pow(31,n-1-i));                    }
System.out.print(hash);
```

  The full Java code is attached in the file " QN_4_computeHash.Java"

- **Computing the hashCode mathematically:**

  Input String = "Hello to the World"

  Hashvalue = H*31^17 + e*31^16 + l*31^15 .... d*31^0

  Hashvalue = -1704027132 (negative sign is because the string is too long and hence the hashcode is bigger than the largest integer we can store on a 32 bit CPU.

5. **Code attached in file "sumOfDigits.JAVA"**

6. **Code attached in file "QN_6.JAVA"**

7. **Code attached in file "VikramKannan_infix2postfix.java"**

   Step by Step progress of the algorithm:

   INPUT: A * B / C + (D + E - (F * (G / H)))

INPUT	STACK	Print to Screen	EXPLANATION
A		A	
*	*	A	
B	*	AB	
/	/	AB*	Both '*' and '/' operators are of same precedence. Hence '*' is popped out of the stack and printed on the screen. The '/' operator is pushed to the stack.
C	/	AB*C	
+	+	AB*C/	The operator '+' has lower precedence than '/'. Hence '/' is popped out of the stack and printed on the screen. The '+' operator is pushed to the stack.
(	+(	AB*C/	
D		AB*C/D	
+	+(+	AB*C/D	
E		AB*C/DE	
-	+(-	AB*C/DE+	Both the '+' and '-' operators are of same precedence. Hence '+' operator is popped out of the stack and printed on the screen. The '-' operator is pushed to the stack.
(	+(-(	AB*C/DE+	
F		AB*C/DE+F	
*	+(-(*	AB*C/DE+F	
(	+(-(*(	AB*C/DE+F	
G	+(-(*(	AB*C/DE+FG	
/	+(-(*(/	AB*C/DE+FG	
H	+(-(*(/	AB*C/DE+FGH	
)	+(-(*	AB*C/DE+FGH/	The '/' operator is popped out of the Stack and printed on the screen. The open parenthesis corresponding to the close parenthesis is removed from the stack.

)	+(-	AB*C/DE+FGH/*-	The '*' operator is popped out of Stack and printed on the screen. The open parenthesis corresponding to the close parenthesis is removed from the stack.
)	+	AB*C/DE+FGH/*-	The '-' operator is popped out of the stack and printed on the screen. The open parenthesis corresponding to the close parenthesis is removed from the stack.
)		AB*C/DE+FGH/*-+	The '+' operator is popped out of the stack and printed on the screen. The open parenthesis corresponding to the close parenthesis is removed from the stack.

The Postfix expression is obtained as: AB*C/DE+FGH/*-+