ASSIGNMENT 3

Jaspreet Kaur

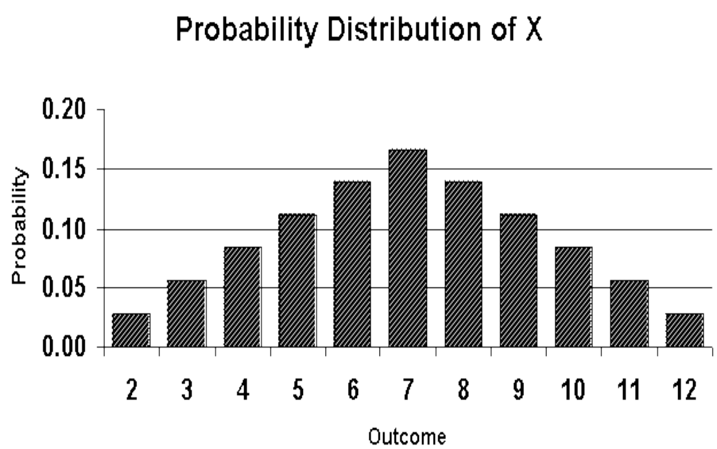
11529746

1. What is the time complexity of the following code, and why?

```
public makeSentence ( String[] words) {
        String sentence=“”;
        for (String w:words) {
                sentence+=w;
        }
        return sentence;
}
```

Answer – The time complexity is O(n^2) where n is the number of strings in the sentence. Each time a string is appended to sentence – a new copy of sentence is created and run through all the letters to copy them over. If we iterate through n characters each time in the loop and you are looping n times –
n * n ~ O(n^2).

2. Suppose the customers enter a Bank has the following histogram:



a) What is Random variable?

b) What are the probabilities for this distribution throwing two dices?

c) Calculate the Mean and Standard Deviation of this Probability distribution?

d) Explain the observed statistics for a Bank system.

Answer – a) A random variable, usually written X, is a variable whose possible values are numerical outcomes of a random phenomenon. There are 2 types of random – variables -> discrete(take countable values) and continuous(take infinite number of possible values).

Let the random variable in above case is X which can take values – 2,3,4,5,6,7,8,9,10,11.

b) Probability for the distribution throwing two dices is 1/36 because there is only 1 way in which we can obtain 2 when we throw 2 dices – (1,1) and total possibilities 6*6 = 36 which gives us 1/36.

c) Mean is the weighted for discrete random variables –

P(X=2) = 1/36

P(X=3) = 2/36

P(X=4) = 3/36

P(X=5) = 4/36

P(X=6) = 5/36

P(X=7) = 6/36

P(X=8) = 5/36

P(X=9) = 3/36

P(X=10) = 3/36

P(X=11) = 2/36

P(X=12) = 1/36

Mean = 2(1/36) + 3(2/36) + 4(3/36) + 5(4/36) + 6(5/36) + 7(6/36) + 8(5/36) + 9(4/36) + 10(3/36) + 11(2/36) + 12(1/36) = 252/36 = 7

Variance = (((2-7)^2)(1/36)) + (((3-7)^2)(2/36)) + (((4-7)^2)(3/36)) + (((5-7)^2)(4/36)) + (((6-7)^2)(5/36)) + (((7-7)^2)(6/36)) + (((8-7)^2)(5/36)) + (((9-7)^2)(3/36)) + (((10-7)^2)(3/36)) + ((11-7)^2)(2/36)) +

(((12-7)^2)(1/36)) = 25/36 + 32/36 + 27/36 + 16/36 + 5/36 + 1/36 + 16/36 + 27/36 + 32/36 + 25/36 + 50/36 + 64/36 + 54/36 + 32/36 + 6/26 = 5.833

Standard Deviation = sqrt(variance) = sqrt(5.7) = 2.415

d) The probability distribution follows a bell curve which is known as Normal Distribution or Gaussian Distribution. It is symmetric about the mean. In the above bank system, the average of many samples of random variable with finite mean and variance is itself a random variable where distribution converges to a normal distribution.

3. Write code that results to the following running time. The 3-Sum Triple loop has the following running time estimate.

A) Do Not prove Math. Just want to explain the math. What does the math do represent and why the result is 1/6 N^3?

$$\sum_{i=1}^{N}\sum_{j=i}^{N}\sum_{k=j}^{N} 1 \;\sim\; \int_{x=1}^{N}\int_{y=x}^{N}\int_{z=y}^{N} dz\,dy\,dx \;\sim\; \frac{1}{6}N^3$$

Answer - Let's consider the 3-sum problem –

```
for (int i = 0; i < N; i++) {
        for (int j = i + 1; i < N; j++) {
            for (int k = j + 1; k < N; k++) {
                if (a[i] + a[j] + a[k] == 0) {
                    count++;
                }
            }
        }
    }
```

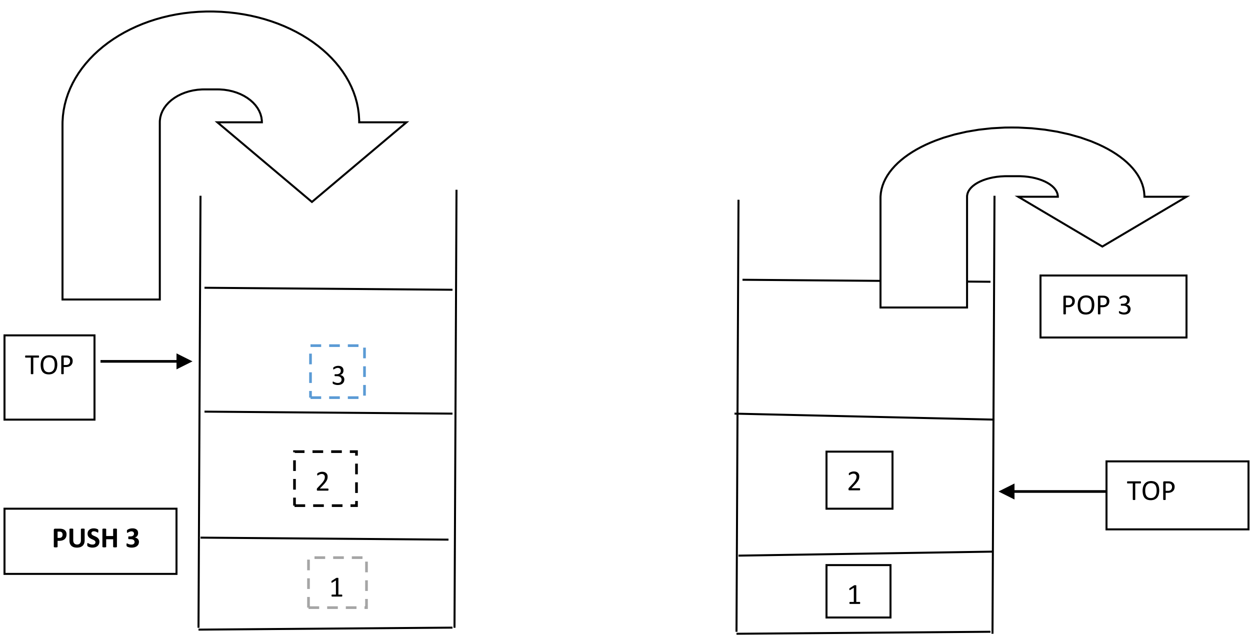
In the algorithm – i, j, k can take values from 0 to N where N is the total number of elements present in the array. The important point is that i, j and k are different from each other i.e they have different indexes. Therefore, the outermost 'i' loop can take N number of values, 'j' loop can take 'N-1' number of values and 'k' loop can take 'N-2' number of values.

Let's say we have 3 numbers 3,-1,-2 There are 6 ways, these numbers can form a combination – (3,-1,-2) (3,-2,-1) (-2,3,-1) (-1,-2,3) (-2,-1,3) (-1,3,-2), But in the code, we want to count it only once – that is why we need to divide by 6, therefore number operations will e equivalent to ~ N(N-1)(N-2)/6.

4. What is Stack data structure, and stack operations? Explain

Stack is an Abstract type data structure because it can be used for any data type.

REPRESENTATION OF STACK IS AS FOLLOWS –



<u>Properties of Stack –</u>

1. LIFO – Stack follows Last in First Out i.e., LIFO. The element that is inserted last will be taken out(deleted/popped) first.
2. Example – Pile of plates follows a real-world example of stack.
3. Top – Top of stack always points to the last element in the stack.
4. Deletion of element – Element can be deleted from the Top.

Operations on Stack –

1. Push() – If we push an element in a stack – its called push.
2. Pop() – If we delete an element from a stack – its called pop. It also returns the popped element.
3. Peek() – It retrieves the top element of stack – without deleting it.
4. isFull() – It checks if the stack is full.
5. isEmpty() – It check if stack is empty.

Implementation of Operations on Stack –

1. **push() –**

```
public void push(int x){
    if(top == 99){
      System.out.println("STACK FULL");
    }
    else{
      top++;
      items[top] = x;
    }
  }
```

2. **pop() –**

```
public int pop(){

  if(top == -1){

    System.out.println("No Element to Delete");

    return -1;

  }

  else{

    int element = items[top];

    System.out.println("TOP BEFORE DELETE "+top);

    top--;
```

```
            System.out.println("TOP AFTER DELETE "+top);
            return element;
        }
    }
```

3. **isEmpty() –**

```
public boolean isEmpty(){
        return (top == -1) ? true:false;
    }
```

4. **isFull() –**

```
public boolean isFull(){
        return (top == arr.length) ? true:false;
    }
```

5. **peek() –**

```
public int peek(){
        if(top!=-1){
            return items[top];
        }
        return 0;
    }
```

<u>CHECKS on Stack –</u>

1. isFull() – It checks if stack is full and prevents the element from getting inserted if it returns true.
2. isEmpty() – It checks if stack is empty and prevents from deleting an element from the stack.

For example – if we implement stack using array arr[] -

int arr[] = new arr[5];

arr.pop();

ERROR – because no element to delete

arr.push(1);

arr.push(2);

arr.push(3);

arr.push(4);

arr.push(5);

arr.push(6);

ERROR – because stack is already full

Qs 5. Consider String “It was the best of time”. Start with the first word, design a Stack such that when you read back the words, the order of string does not change. Provide code for all necessary operations of Stack. Compile and run the code.

Answer – Qs5_Stack_String

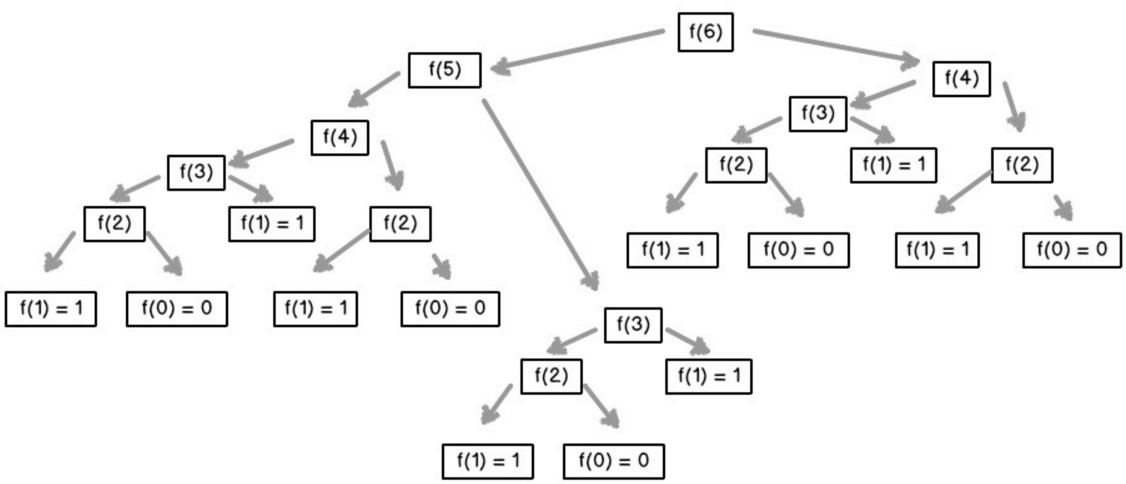
```
public static void main(String[] args) {
    Stack<String> stack1 = new Stack<String>();
    Stack<String> stack2 = new Stack<String>();
    String input = "It was the best of time";
    String[] arr = input.split(" ");
    int size = arr.length;
    for(int i=0; i<size; i++) {
      stack1.push(arr[i]);
    }
    String[] res = new String[size];
    for(int i=0; i<size; i++) {
      stack2.push(stack1.pop());
    }
```

```
for(int i=0; i<size; i++) {
   res[i] = stack2.pop();
   System.out.print(res[i]+" ");   }}
```

Ques6 The Recursive operations for Factorial and Fibonacci sequence was discussed in class.

A) For Fibonacci sequence with n=7, the following diagram shows its Tree Structure

a) Is this diagram iterative or recursive?
b) What data structure is used to implement recursion?
c) Provide Tree Structure for n=5 step-by-step. What differences do you see in diagrams between n=6 and n=7?
d) What are Pros and Cons between iterative and recursive Algorithms?
e) Write recursive Java code for both n=6 and n=7



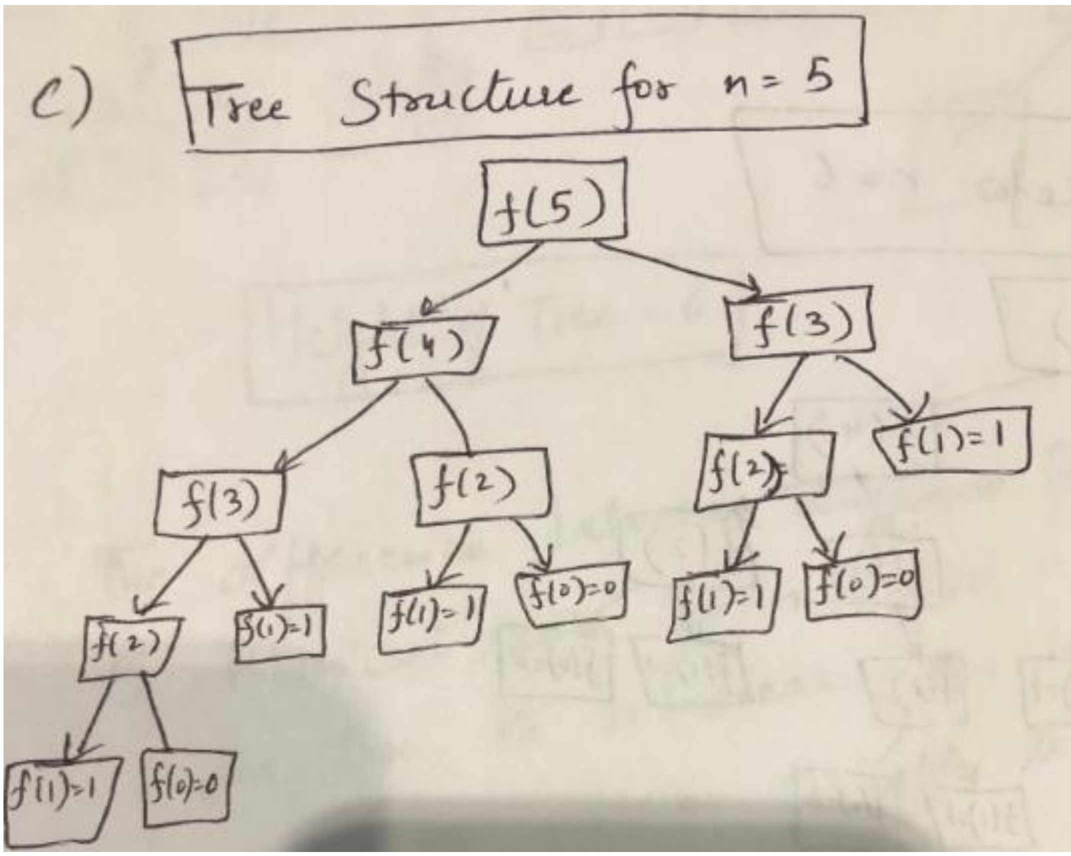
B) For factorial 8! a) Show recursive stack operations, provide details step-by-step, b) Walk through your stack operations and provide the result. c) Write Java code with input factorial 6! d) Compile and run your program, what is the running time of your algorithm?

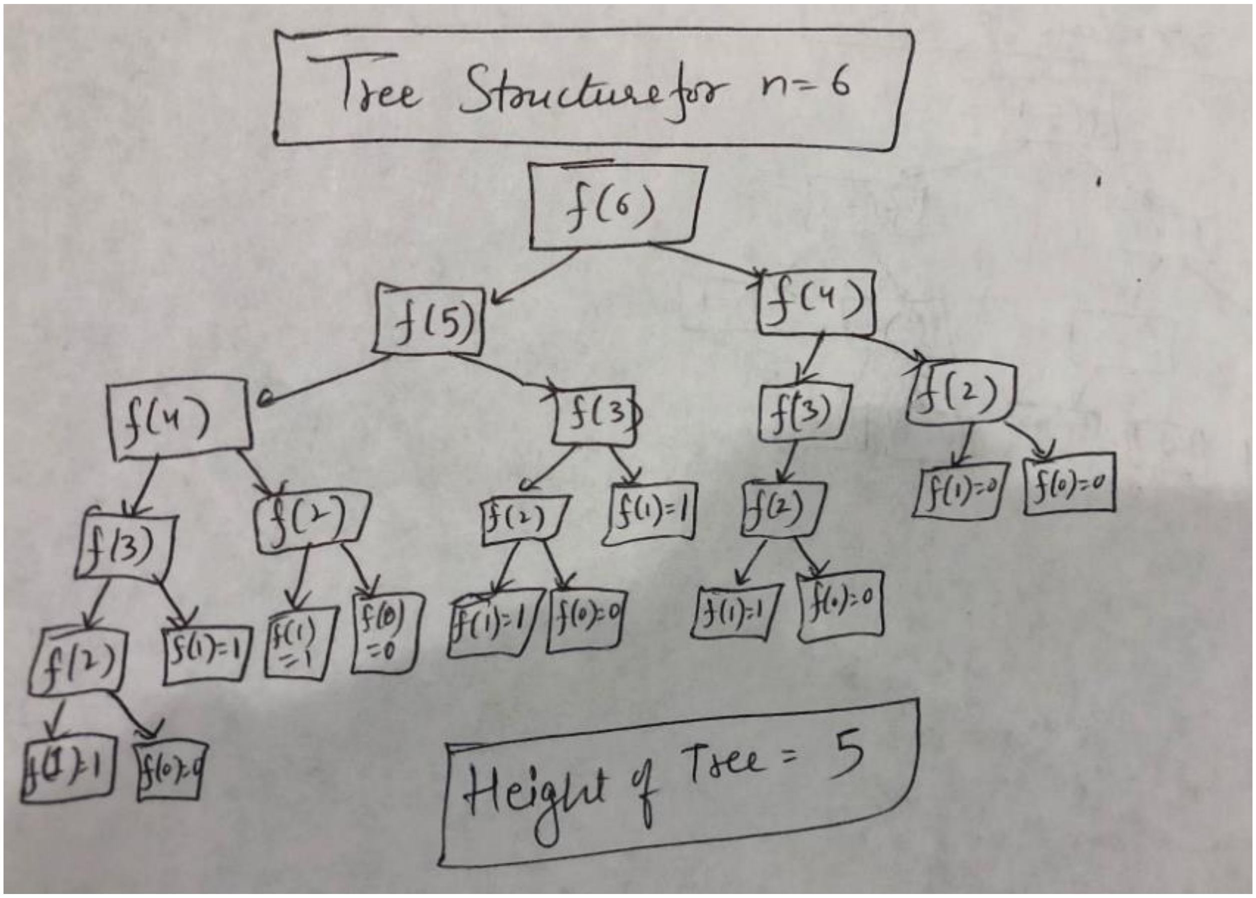
Answer –

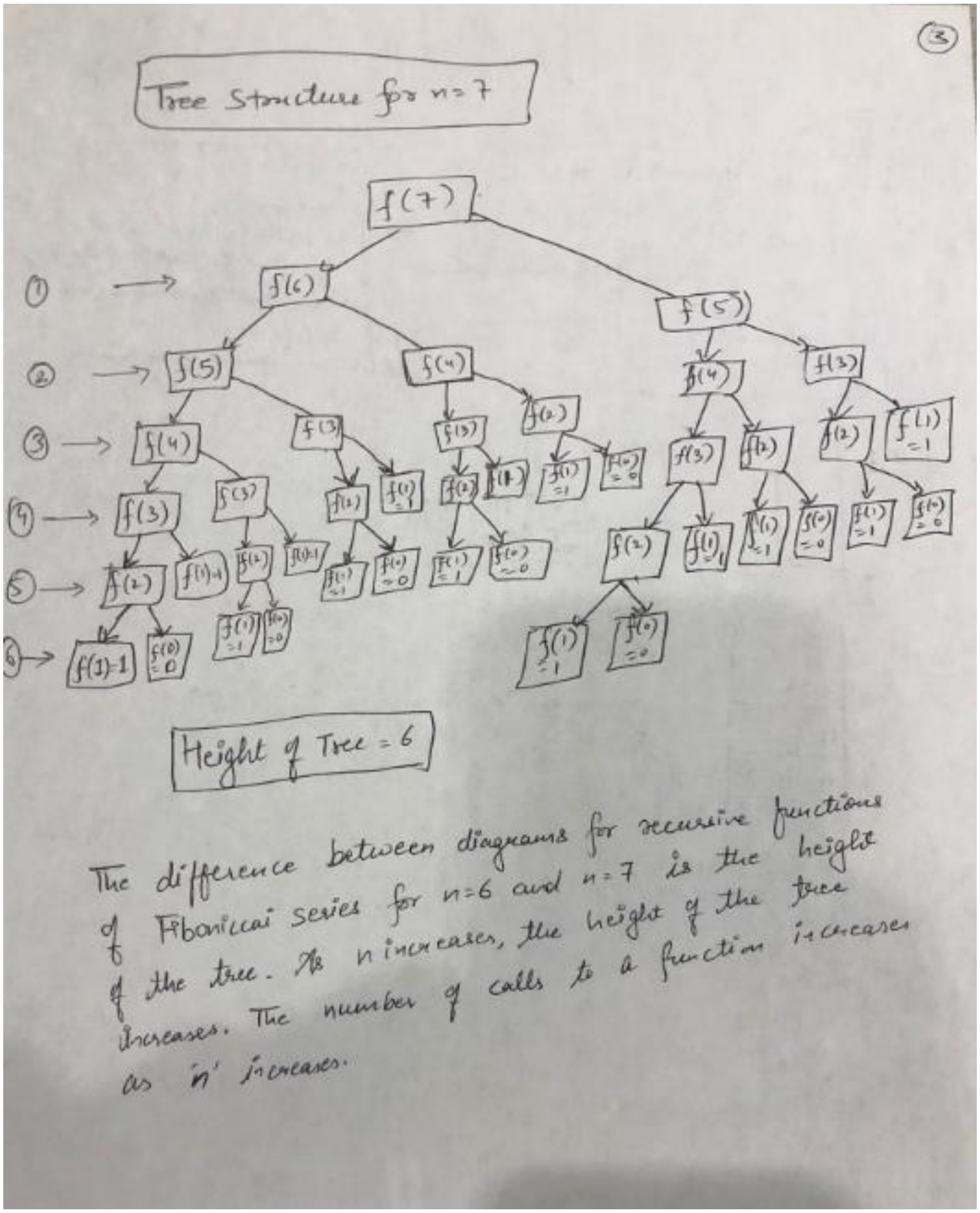
A) a) The diagram is recursive because at every step same function is being called.

b) Stack is used to implement recursion because the value that is being pushed to the stack the first time is popped out the last.

c)




Tree Structure for n=6
f(6)
f(5)
f(4)
f(4)
f(3)
f(3)
f(2)
f(3)
f(2)
f(2)
f(1)=1
f(2)
f(1)=0
f(0)=0
f(2)
f(1)=1
f(1)=1
f(0)=0
f(1)=1
f(0)=0
f(1)=1
f(0)=0
f(1)=1
f(0)=0
Height of Tree = 5



d) Difference between iterative and recursive call is tabulated below -

Iterative Call	Recursive Call
ADVANTAGE OF ITERATIVE OVER RECURSIVE	
1. Faster than recursion	**1. It is slower.**
2. Use less memory than recursion (depending on input size).	**2. Use more memory as input size increases.**

3. Relatively Lower space-complexity.	**3. Relatively higher space-complexity.**
4. No need to calculate value for particular input again and again(depending upon the code).	**4. Some values are calculated again and again – which leads to introduction to Dynamic Programming.**
ADVANTAGE OF RECURSIVE OVER ITERATIVE	
5. The code size is large.	**4. The code size is small.**
6. Performance is not great with Trees Algorithms.	**5. Performs better with tree algorithms.**

e) Implementation – Qs6A(e)_Fibonacci

```
 public int fibonacciRecursion(int n) {
        if (n == 0) {
           return 0;
        }
        if (n == 1 || n == 2) {
           return 1;
        }
        return fibonacciRecursion(n - 2) + fibonacciRecursion(n - 1);
}
```

f) Results –

```
run:
Fiboniccai Series for n=6
0
1
1
2
3
5

Fiboniccai Series for n=7
0
1
1
2
3
5
8
BUILD SUCCESSFUL (total time: 1 second)
```

B) a) and b) parts - TOGETHER

B) a) 8!.

```
factorial (int n)
{
   if (n==1)
   {
     return 1;
   }
   else
   {
   return n * factorial(n-1)
   }
}
```

```
main ()
{
  int res = factorial(8)
  System.out.print (res + " ")
}
```

Step1

8 * fact(7)	main ( )

Step2

7 * fact(6)	fact (7)
8 * fact(7)	main ( )

Step3

6 * fact(5)	fact(6)
7 * fact(6)	fact(7)
8 * fact(7)	main ( )

Step4

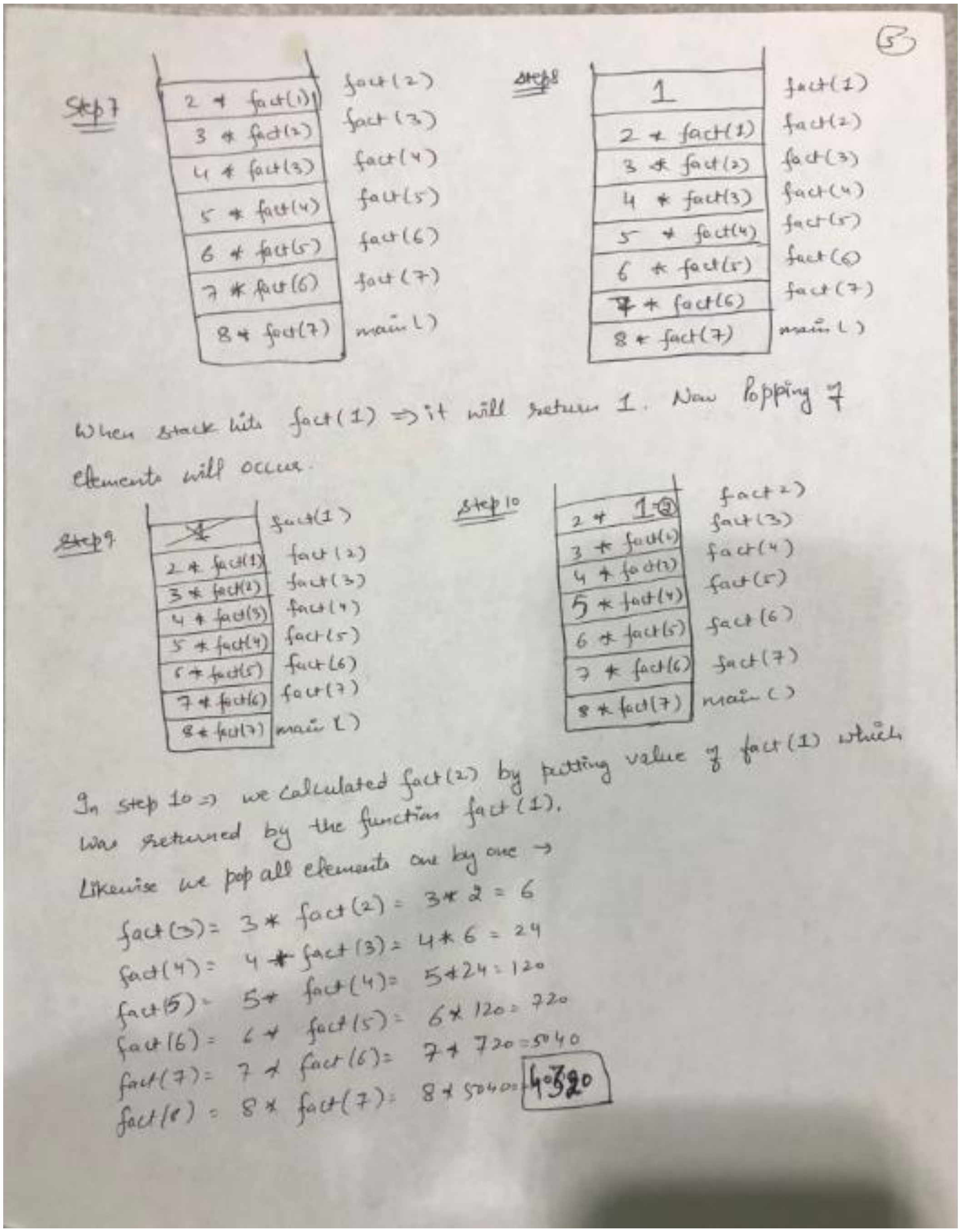
5 * fact(4)	fact(5)
6 * fact(5)	fact(6)
7 * fact(6)	fact (7)
8 * fact(7)	main ()

Step5

4 * fact(3)	fact (4)
5 * fact(4)	fact (5)
6 * fact(5)	fact (6)
7 * fact(6)	fact (7)
8 * fact(7)	main ( )

Step6

3 * fact(2)	fact (3)
4 * fact (3)	fact(4)
5 * fact(4)	fact(5)
6 * fact(5)	fact (6)
7 * fact(6)	fact (7)
8 * fact (7)	main ( )



c) Implementation – Qs6B(c)_Factorial

```
public long multiplyNumbers(int num)
{
  if (num >= 1)
    return num * multiplyNumbers(num - 1);
  else
```

```
        return 1;
    }
```

d) The time-complexity for Factorial 6 program is O(n) as the program will iterate from num=6 to num=1. The function is called recursively n times.

Qs 7. Consider following data to build Stack with:

A) LinkedList implementation

B) Array implementation

	A	B	C	D
1	**ID**	**First Name**	**Last Name**	**Course**
2	1	Jack	Irwan	Software Engineering
3	2	Billy	Mckao	Requirement Engineering
4	3	Nat	Mcfaden	Multivariate Calculus
5	4	Steven	Shwimmer	Software Architecture
6	5	Ruby	jason	Relational DBMS
7	6	Mark	Dyne	PHP development
8	7	Philip	namdaf	Microsoft Dot Net Platform
9	8	Erik	Bawn	HTMl & Scripting
10	9	Ricky	ben	Data communication
11	10	Van	Miecky	Computer Networks
12				

a) Create file “Input.txt” with this data

b) Read input.data into an ArrayList

c) Create Stack with LinkedList implementation

d) Write Node data structure of your input data

e) Stack must support all operations of stack: push, pop. is-empty, is-full

f) Write a Test program to test your linked implementation of Stack:

—push 4 elements into stack

—pop 5 elements from stack

—push all elements into stack

—push

11 john henry “software development”

12 justin morgan “engineering statistics”

—pop all elements from stack

—push 8 elements into stack

—pop 9 elements from stack

—push all elements into stack

—pop all elements from stack

—Print stack with the goal:
i) reverse order ii) original order as was first read into array list

g) Compile and Run your program

h) what is Stack LinkedList time-complexity?

i) Repeat (a)—(h) with Stack fixed Array Implementation

j) What are the consequences of oversizing or undersizing fixed array size?

Answer -

a) to g) Qs7_StackLinkedImpl

i) Qs7_Array

h) Time Complexity for Stack With Linked List Implementation –

- push() – This operation will take O(1) complexity for item to be inserted at the end.
- Pop() – This operation will take O(1) complexity for item to be deleted from the end.
- Peek() – O(1) – because item can be taken out from the top in one step.

Time Complexity for Stack With Array Implementation –

- push() – O(1) – because item can be pushed using indexing.
- Pop() – O(1)) – because item can be popped using indexing.
- Peek() – O(1)) – because item can be taken out from the top in one step.

j) Oversizing – We need to copy all the items to a new array. It will take time to copy the number of items from old array to new array and corrosponds to time-complexity of O(n).

```
public void push(Item item)
{
        // Add item to top of stack.
        if (N == a.length) resize(2*a.length);
        a[N++] = item;
}

private void resize(int max)
```

```
{
        // Move stack to a new array of size max.
        Item[] temp = (Item[]) new Object[max];
        for (int i = 0; i < N; i++)
        temp[i] = a[i];
        a = temp;
}
```

Undersizing – Undersizing also requires O(n) time – complexity.

```
public Item pop()
        {  //
        Remove item from top of stack.
        Item item = a[--N];
        a[N] = null;
        if (N > 0 && N == a.length/4) resize(a.length/2);
        return item;
}
```

Resizing an array is cumbersome task and performance issue arises as the size of array changes.

Qs 8 Consider following Algorithm to "Evaluate Infix Expressions" with Two arrays:

Test data:

A * B / C + (D + E - (F * (G / H)))
(1 + 3 + ((4 / 2) * (8 * 4)))
(4 + 8) * (6 - 5)/((3 - 2) * (2 + 2))

A) Step through algorithm to develop a Stack Table for for each Infix expression

B) Write Java code to test each Infix Expression

C) Compile and Run

ANSWER

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

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

Element	Operation	Operator Stack	Evaluation
A	Push A		A
*	Push *	*	A
B	Push B	*	A B
/	Pop * Push A*B Push /	/	A*B
C	Push C	/	A*B C
+	Pop / PUSH A*B/C PUSH +	+	A*B/C
(	Push (	+(	A*B/C
D	Push D	+(	A*B/C D
+	Push +	+(+	A*B/C D
E	Push E	+(+	A*B/C DE
-	Pop + Push D+E Push -	+(-	A*B/C D+E
(	Push (	+(-(	A*B/C D+E
F	Push F	+(-(	A*B/C D+E F
*	Push *	+(-(*	A*B/C D+E F
(	Push (	+(-(*(	A*B/C D+E F
G	Push G	+(-(*(	A*B/C D+E F G

Element	Operation	Operator Stack	Evaluation
/	PUSH /	+(-(*(/	A * B/C D+E F G
H	Push H	+(-(*(/	A * B/C D+E F G H
)	Pop / Push G/H	+(-(*	A*B/C D+E F G/H
)	Pop * Push F*G/H	+(-	A*B/C D+E F*G/H
)	Pop - Push D+E-F*G/H	+	A*B/C D+E-F*G/H
	Pop+ PUSH A*B/C+D+E-F*G/H		A*B/C+D+E-F*G/H

(ii) (4 + 8) * (6 - 5)/((3 - 2) * (2 + 2))

8) (4+8) * (6-5) / ((3-2) * (2+2))

Element	Operation	Stack operand	Stack Operator
(	Push (		(
4	Push 4	4	
+	Push +	4	(+
8	Push 8	4 8	(+
)	Pop +	12	
*	Push *	12	*
(	Push (	12	*(
6	Push 6	12 6	*(
-	Push -	12 6	*(-
5	Push 5	12 6 5	*(-
)	Pop - / PUSH 1	12 1	*
/	Pop * / PUSH /	12	/
(	Push (	12	/(
(	Push (	12	/((
3	Push 3	12 3	/((
-	Push -	12 3	/((-
2	Push 2	12 3 2	/((-
)	Pop - Push 1	12 1	/(
*	Push *	12 1	/(*
(	Push (	12 1	/(*(
2	Push 2	12 1 2	/(*(
+	Push +	12 1 2	/(*+
2	Push 2	12 1 2 2	

Element	Operation	Stack operand	Stack operator
)	Pop + Push 4	12 1 4	/ (*
)	Pop * Push 4	12 4	/ (
/	Pop / Push 3	3	

(iii) (1 + 3 + ((4 / 2) * (8 * 4)))

8) 1+ 3+ ((4/2) * (8 * 4))

Element	Operation	Stack operand	Operator Stack
1	Push 1	1	
+	Push +	1	+
3	Push 3	1 3	+
+	Pop + Push + Push 4	4	+
(	Push (	4	+(
(	Push (	4	+((
4	Push 4	4 4	+((
/	Push /	4 4	+((/
2	Push 2	4 4 2	+((/
)	Pop / Push 4/2=2	4 2	+(
*	Push *	4 2	+(*
(	Push (	4 2	+(*(
8	Push 8	4 2 8	+(*(
*	Push *	4 2 8	+(*(*
4	Push 4	4 2 8 4	+(*(*
)	Pop * Push 8*4=32	4 2 32	+(*
)	Pop * Push 2*32=64	4 64	+
	Pop + Push 4+64=68	68	

C) Implementation –> Qs8_InfixEvaluation

Qs 9. Consider the following Algorithm to convert Infix expression to Postfix.

A) Infix expression example: (A + B) * C + D / (E + F * G) - H
B) Apply Algorithm to Infix example, show step-by-step
C) Write Java code for the algorithm to convert Infix to Postfix expression

Algorithm:
while there are more symbols to read
read the next symbol
case:
operand --> output it.
'(' --> push it on the stack.
')' --> pop operators from the stack to output until a '(' is popped; do not output either of the parentheses.
operator --> pop higher- or equal-precedence operators from the stack to the output; stop before popping a lower-precedence operator or a '('. Push the operator on the stack.
end case
end while
pop the remaining operators from the stack to the output

Answer – (A) and (B)

Q1. (A+B) * C+D/(E+F*G)-H

Element	Operation	Operators	Output
(	Push (	(	
A	Push A	(	A
+	Push +	(+	A
B	Push B	(+	AB
)	Pop +		AB+
*	Push *	*	AB+
C	Push C	*	AB+C
+	Poop *	+	AB+C*
	Push +		
D	Push D	+	AB+C*D
/	Push /	+/	AB+C*D
(	Push (	+/(	AB+C*D
E	Push E	+/(	AB+C*DE
+	Push +	+/(+	AB+C*DE
F	Push F	+/(+	AB+C*DEF
*	Push *	+/(+*	AB+C*DEF
G	Push G	+/(+*	AB+C*DEFG
)	Pop *	+/	AB+C*DEFG*+
	Pop +		

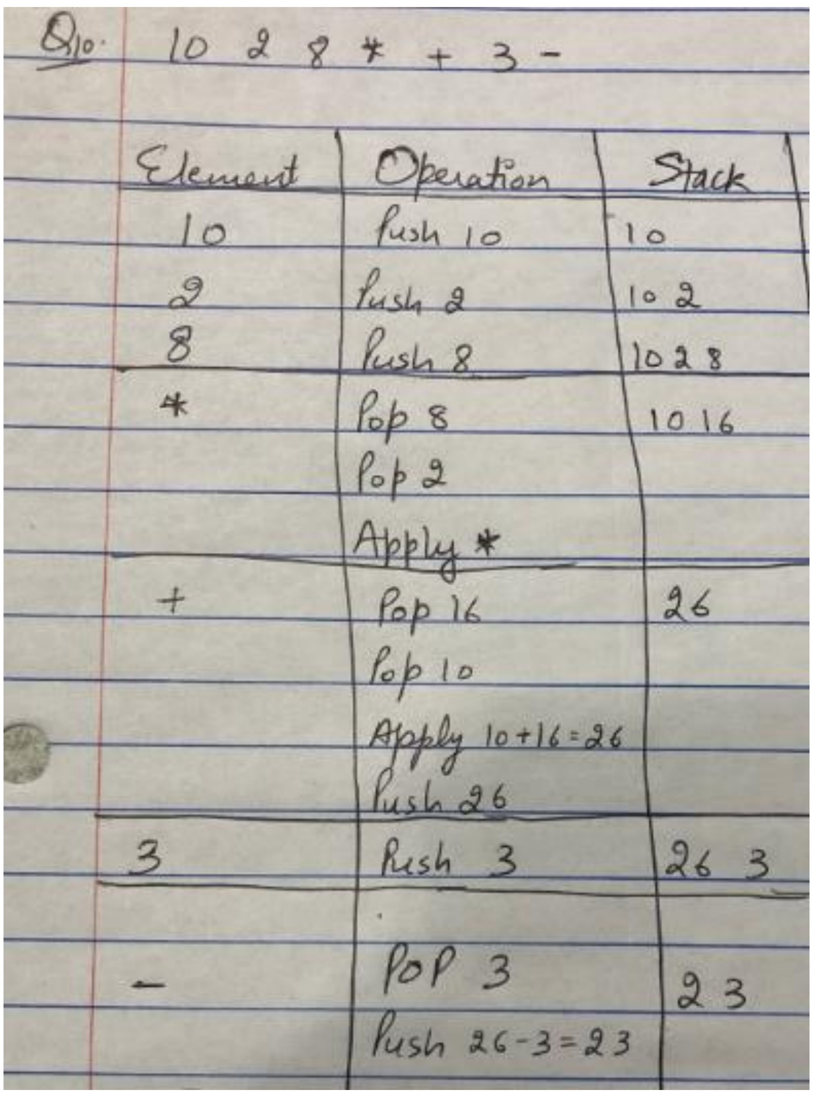
Element	Operation	Operators	Evaluation
−	Pop /	−	AB+C*DEFG*+/+
	Pop +		
	Push −		
H	Push H	−	AB+C*DEFG*+/+H
	Pop −		AB+C*DEFG*+/+H−

C) Implementation – Qs9_InfixToPostfix

Qs 10. Consider this Algorithm to “Evaluate Postfix Expression”: 10 2 8 * + 3 -
Algorithm: Maintain a stack and scan the postfix expression from left to right – When we get a number, output it – When we get an operator, pop the top element in the stack until there is no operator having higher priority than this operator, and then push (operator) into the stack – When the expression is ended, pop all the operators remain in the stack:

A) Show Stack step-by-step
B) Write Java code to compute postfix expression

Answer – A)



Q10. 10 2 8 * + 3 -

Element	Operation	Stack
10	Push 10	10
2	Push 2	10 2
8	Push 8	10 2 8
*	Pop 8 Pop 2 Apply *	10 16
+	Pop 16 Pop 10 Apply 10+16=26 Push 26	26
3	Push 3	26 3
-	Pop 3 Push 26-3=23	23

B) Implementation – Qs10_PostfixEvaluation

Qs 11. Consider the following code with Array Stack implementation

A) Explain this code
B) Why would an application need such a code, Explain
C) What code change would you make to this code to correct over-sizing?

```
public ResizingArrayStackOfStrings()
{  s = new String[1];  }

public void push(String item)
{
   if (N == s.length) resize(2 * s.length);
   s[N++] = item;
}

private void resize(int capacity)
{
   String[] copy = new String[capacity];
   for (int i = 0; i < N; i++)
      copy[i] = s[i];
   s = copy;
```

A) This code is used for resizing the length of the array. If the length hits the current length [N == s.length] => then, a copy of array of string is created with capacity double the size of current length and current array is assigned to copied array with new capacity.

B) This code is helpful to prevent undersizing of an array by doubling the current capacity.

C) Preventing oversizing of an array is crucial from the perspective of memory utilization. To check oversizing – While deleting an element from array, we will check whether the array size is 4 times the current number of elements and if it is true, we will resize the array by resizing it to half size.

```
Public String pop(){
   String item = s[--N];
   s[N] = null;
   if (N > 0 && N == s.length/4)
      resize(s.length/2); return item;
}
```