Question_1:
You must have heard about Object-Oriented Programming. Where the concept of object is introduced, which binds data and methods. How you're going to describe a circle and a rectangle by referencing the concepts of OOP. And please also tell us about runtime polymorphism and ways to achieve it.

Answer 1:

```
class Shape
{
	int length = 10;
	int width  = 5 ;

	void draw()
	{
		System.out.println("The length and width in given ") ;
	}

}

class Rectangle extends Shape
{
	void draw()
	{
		int areaRec = length * width ;
		System.out.println("The area of a rectangle is "  + areaRec) ;
	}
}

class Circle extends Shape
{
	void draw()
	{
		double areaCircle = 2 * 3.1416 * width ;
		System.out.println("The area of a rectangle is "  + areaCircle) ;
	}
}

class Question_1
{
	public static void main(String args[])
	{
		Shape s;
		s=new Rectangle();
		s.draw() ;
```

```
                s=new Circle();
                s.draw();
        }
}
```

In the above code, child class Rectangle and class Circle inherit parent class Shape. class Rectangle and class Circle overrides the method draw() which is called method overriding. It is mainly Polymorphism in OOP. Both Rectangle and Circle class have specific implementations of the draw() method. Runtime polymorphism, also known as Dynamic Method Dispatch, is a process that resolves a call to an overridden method at runtime rather than compile-time. In this process, an overridden method is called through the reference variable of a superclass.

Question_2:
Please describe stack and heap memory. Assume you need a very large array (100MB) and your preferred language allows you to both allocate memory on heap and stack. Where do you want to allocate memory for your array and why?

Answer_2:
A stack is a type of memory area in a computer that stores temporary variables created by a function. Variables are declared, stored, and initialized on the stack during runtime. It is a type of temporary memory. When the computing task is finished, the variable's memory is automatically erased. Methods, local variables, and reference variables are mostly found in the stack section.
At runtime, each thread is given a physical place (in RAM) called stack memory. When a thread is created, it produces this object. Because the stack is globally accessible, memory management follows the LIFO (Last-In-First-Out) principle.

The heap is a type of memory that computer languages employ to keep track of global variables. All global variables are stored in heap memory space by default. It has the ability to allocate memory dynamically. The heap isn't managed automatically for you, and the CPU doesn't have as much control over it. It's more like a memory's free-floating section. In JAVA it is produced when the JVM starts up and is used by the application for the duration of its execution. It saves JRE classes and objects. When we construct objects, they take up space in the heap memory, while their references are stored in the stack. It is not arranged in the same way as the stack. The memory blocks are managed dynamically. It means we won't have to deal with the memory by hand. Java has a garbage collector that automatically manages memory by deleting things that are no longer in use.
I will store the large array (100 MB) in Heap memory. Because the heap has no limit on memory. Size. On the other hand, Stack memory is very limited. So I will prefer to store the array in Heap memory rather than the stack.

Question_3:
Imagine there's two matrix A, B with the same dimension. You want to perform element-wise matrix multiplication. So, you iterate two for loops and perform element-wise multiplication.

```
for i := 0; i < Length; i++{
        for j = 0; j < width; j++{
                ans[i][j] = A[i][j] * B[i][j]
```

but this is slow if the matrix size is huge (million elements or more). Is there any way to improve or speedup this process? Just share your ideas to improve this solution.
**hint: we live in a world full of ryzen cores :P

Answer_3:
In Matlab/Python/R we can perform the element-wise matrix multiplication by multiplying both the matrix A and B. In matlab, above operation can be done by the following way for efficiently.
For example : C = A . *B
Here, the multiplication is done parallelly. Ryzen processor has multi-core in its processor. So this type of multiplication is done in multicore processors parallely in high level programming language like Python, Matlab, R, etc. But in the question, the multiplication is done sequentially. It will speed up the process if the multiplication is done by multiplying them both in the above way.

Question_4:
Imagine you have to traverse a tree, in which you need to count the total number of nodes. So, it's easy right. You can traverse any way (in-order, pre, post). All you need is a global scope counter variable and a single line: count++ with recursion.But there's a catch, the tree is huge (millions of nodes and heavily left-skewed).So, you wrote a recursive function, like below,

```
struct Node{
        Node* left;
        Node* right;
}
int traverse(Node* node){
        if (node == null)   return 0;   //base case
        count++;
        count += traverse (node->left)    //recursive call
        count += traverse (node->right)   //recursive call
        return count ;
}
```

Can you please confirm that this recursive approach is going to work? if not please explain why and share any alternative ways to count the nodes.

Answer_4:

The above approach will not work properly. It will work in the following way:

```
int count = 0 ;
struct node
{
        node *left, *right ;
} ;
void traverse(node* node)
{
   if (node == NULL) return ;
   count++ ;
   traverse(node->left) ;
   traverse(node->right) ;
}
```

In the question, count variable in incremented more than one time in each recursion. But it will increase only once when the function calls.