Theory and facts.
1)

When a program is executed, it is allocated a memory area. The size of this area depends on various factors, such as the operating system, the number of programs running simultaneously, and the total memory space available. The memory allocated to a program can usually be divided into four parts and are commonly referred to as code segments, data segments, stack segments, and heap segments.

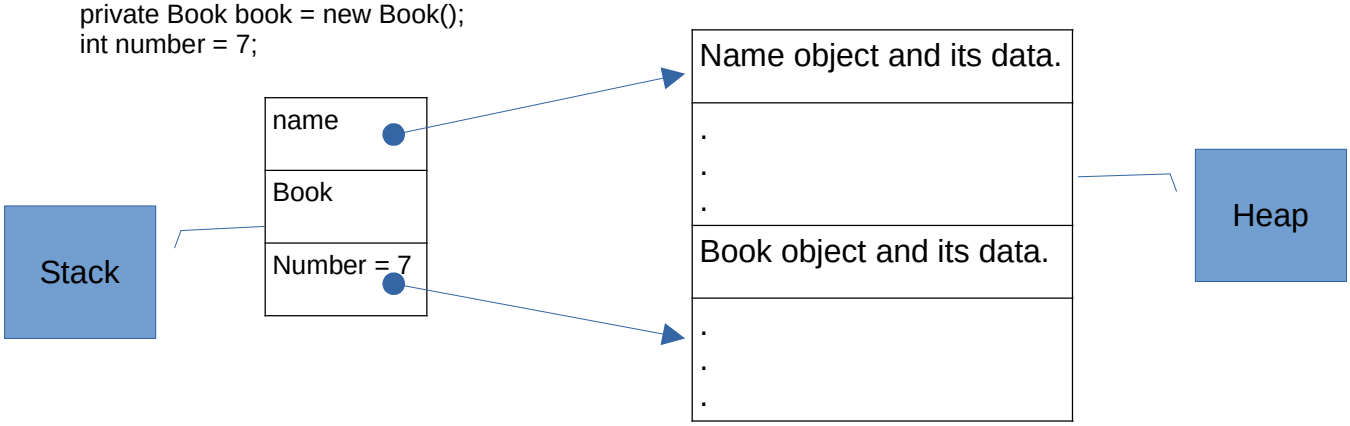
Stack memory:
The stack memory is used for value type variables, local variables, method parameters, and function return values. Local variables are those declared inside a method. The total stack memory is determined per process. The variables must be initialized, i.e. they must have an initial value.
The stack memory works like a pile of plates in a kitchen where you start from the top, take a plate, do something with it, before you reach for the next plate. Another example is a pile of boxes on top of each other. The box on top is taken care of first and then the next and so on.
When a method is called, the method's code (instructions) is placed on the stack memory then the method's parameters. When each line of code in the method is executed, memory is allocated for the local variables. As soon as the method is finished executing, the memory is cleared and the next one is picked.

Heap memory:
Heap is a mechanism for more complex data structures such as strings or classes and memory that can be dynamically allocated with the operator new. An analogy for heap memory can be drawn with a pile of clean clothes on the bed. You can pick any of the clothing, and choose to either ware it, iron it and put it away then choose the next piece of clothing.
The heap memory is managed by the CLR and as a C# programmer you don't need to worry about it.
When more memory is needed, the garbage collector (GC), is activated by the CLR. GC examines the stack and checks if an object is dereferenced and no longer in use. The GC then kills the object and frees its memory. The GC works asynchronous and will choose objects in no particular order. For efficiency you may want to queue an object so that the GC can reclaim memory quicker. To do so mark a value as null or call the object's Dispose() method.

Reference type variables consist of two parts, a reference variable, also known as a pointer, and an object. The pointer variable is a value type that is placed on the stack while the object itself is placed on the heap.
EX.
private string name;
private Book book;

The variables, name and book, are references placed on the stack and point to an null. They are not pointing to an object on the heap.

```
private string name = new sting();
private Book book = new Book();
int number = 7;
```




The reference variables name and book are allocated in memory on the stack.

The objects name and book have been initialised with enough space that their classes require, and have been allocated on the heap.

| |
|---|
| Heap memory<br>Grows down-wards |
| Free memory<br>When the heap and stack meet, memory runs out! |
| Stack memory<br>Grows up-wards |
| Code<br>Program instructions, static variables, C#, Java, C++…<br>When the program is started, the program instructions are loaded into memory. |

2)
A C# data type can be either a value based type, primitive types such as int and double, or a reference based type, i.e. a class type.

Value types:
Value types include numeric types (int, float, double, etc.) as well as enumerators (enum) and structures (struct). A value type variable directly saves its value, which is placed on the stack memory.

Reference types:
A reference type variables save a reference to the memory space where the object's data is saved. This means that a reference type does not save data directly, but an address to the memory location where the value or values belonging to the object are located.
The reference variable itself is placed on the stack memory, while the object is saved in a location controlled by the CLR on the heap memory.

3)
public int ReturnValue()
Reference variables need to initialized in the scope of a method. Thus int x and int y are initialized with new and are on the stack. The variable x is assigned the value 3 there after variable y is assigned the value of variable x so both variables x and y are assigned the value of 3. Next the variable y is assigned with a new value of 4. This means value of x is not equal to the value of y. When return is reached the variable of x is returned from the method thus the value of 3 is returned from the method ReturnValue().

public int ReturnValue2()
Reference variables need to initialized in the scope of a method. Thus int x and int y are initialized with a new class MyInt() and are on the stack. However, both x and y reference variable and are assigned a reference to a memory position on the heap.
x.MyValue is a variable of the class MyInt and it is assigned a value of 3 on the heap. However, y=x this means that variable x reference to the heap is now assigned to the variable y. Thus both variables on the stack are referencing the same memory on the heap. Now y.MyValue = 4 is updating the same heap memory that both variables x and y are referencing to. When then return is reached the variable x returns the value of the referenced heap memory. Thus the method ReturnValue2() returns the value 4.

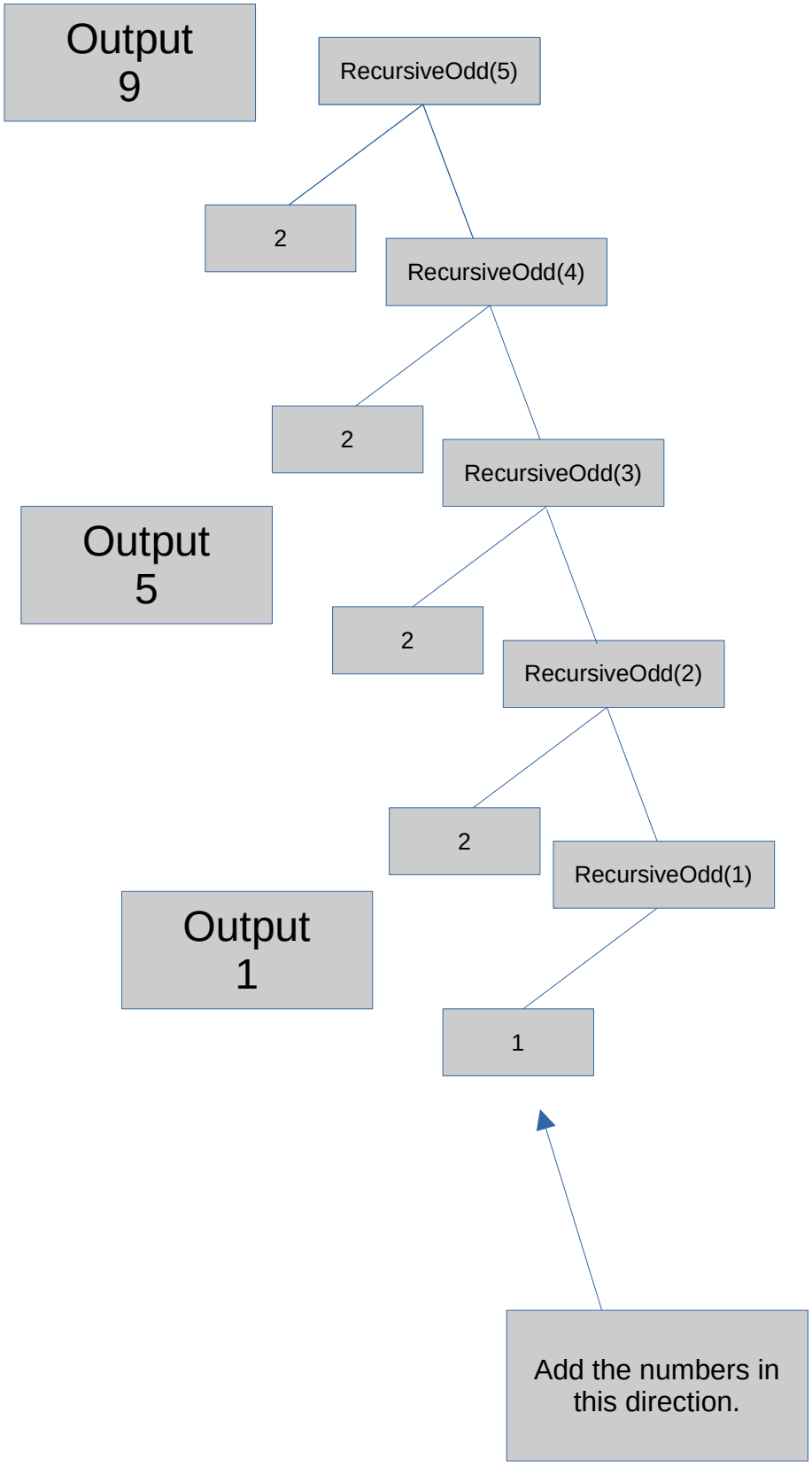
Examine List

Count is the number of elements that are currently in the list.
Capacity is the number of elements that can be stored in the list before
resizing is required.
2) When count exceeds the current capacity then .NET framework resizes the list.
3) The increase in the lists capacity is double the previous capacity.
4) Resizing a list has trade off's. Memory is a resource and needs managing. Resizing the list after an element has been added to the list could increase the overhead of the application. The overhead could be increased hardware resources and or more code to write and maintain. (This question is a little vague. Are we meant to answer why the underlying .NET framework doesn't increase the capacity or why we should adjust the capacity of the list!)
5) No.
6) Arrays are faster when accessing elements of the array because of the predicted size of the array at compile time. Direct access to a specific element in the array using its index. Thus, if you know ahead of time what you want to store in memory then an array has a performance advantage over a list.

| Examine Queue | | | |
|---|---|---|---|
| | | FiFo queue | FIFOQueue.Count |
| ICA öppnar och kön till kassan är tom | `Queue<string> FIFOQueue = new();` | | 0 |
| Kalle ställer sig i kön | `FIFOQueue.Enqueue("Kalle");` | Kalle | 1 |
| Greta ställer sig i kön | `FIFOQueue.Enqueue("Greta");` | Greta, Kalle | 2 |
| Kalle blir expedierad och lämnar kön | `FIFOQueue.Dequeue();` | Greta | 1 |
| Stina ställer sig i kö n | `FIFOQueue.Enqueue("Stina");` | Stina, Greta | 2 |
| Greta blir expedierad och lämnar kön | `FIFOQueue.Dequeue();` | Stina | 1 |
| Olle ställer sig i kö | `FIFOQueue.Enqueue("Olle");` | Olle, Stina | 2 |

| Examine Stack | | | |
|---|---|---|---|
| | | FILO queue | stack.Count |
| ICA öppnar och kön till kassan är tom | `Stack<string> stack = new();` | | 0 |
| Kalle ställer sig i kön | `stack.Push("Kalle");` | Kalle | 1 |
| Greta ställer sig i kön | `stack.push("Greta");` | Greta, Kalle | 2 |
| Kalle blir expedierad och lämnar kön | `stack.pop();` | Kalle | 1 |
| Stina ställer sig i kö n | `stack.push("Stina");` | Stina, Kalle | 2 |
| Greta blir expedierad och lämnar kön | `stack.pop();` | Kalle | 1 |
| Olle ställer sig i kö | `stack.push("Olle");` | Olle, Kalle | 2 |
| A queue based on a stack means that the first person in the queue, ”Kalle”, will remain in the queue for as long the queue continues to grow. Thus if someone arrives at the last minute they will be prioritised and served first. While, Kalle, who was early, will be penalised and as a consequence will receive poor service from ICA. | | | |

| Check Parenthesis | | | | |
|---|---|---|---|---|
| Input | Examine | | FILO Queue | Stack.count |
| ([{}]({}) | | `Stack<char> stack = new();` | | 0 |
| [{}]({}) | '(' | stack.push(')'); | ) | 1 |
| {}]({}) | '[' | stack.push(']'); | ],) | 2 |
| }]({}) | '{' | stack.push('}'); | },],) | 3 |
| ]({}) | '}' | `if(stack.TryPeek(out char testCharacter))`<br>`  if(testCharacter == '}')`<br>`  stack.Pop();` | ],) | 2 |
| ({}) | ']' | `if(stack.TryPeek(out char testCharacter))`<br>`  if(testCharacter == ']')`<br>`  stack.Pop();` | ) | 1 |
| {}) | '(' | stack.push(')'); | ),) | 2 |
| }) | '{' | stack.push('}'); | },),) | 3 |
| ) | '}' | `if(stack.TryPeek(out char testCharacter))`<br>`  if(testCharacter == '}')`<br>`  stack.Pop();` | ),) | 2 |
| | ')' | `if(stack.TryPeek(out char testCharacter))`<br>`  if(testCharacter == ')')`<br>`  stack.Pop();` | ) | 1 |
| | | | ) | 1 |
| Note that the FILO queue is not empty after reading the last parenthesis in the input. Thus ([{}]({}) fails the paranthesis test. | | | | |
| Note: In my implementation when a left handed parenthesis is detected a right handed parenthesis is pushed onto the stack. This makes easier to compare a right handed parenthesis to what is at the head of the stack. | | | | |


Output
9
RecursiveOdd(5)
2
RecursiveOdd(4)
2
RecursiveOdd(3)
Output
5
2
RecursiveOdd(2)
2
RecursiveOdd(1)
Output
1
1
Add the numbers in this direction.