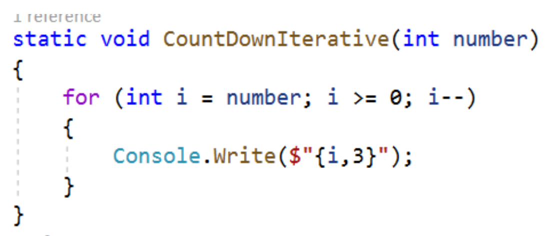
Recursion

Recursion is a key concept in computer science and happens when a method calls itself. Recursion is a powerful design methodology often used by functional programming languages to replace iterative loops. Typically, the implementations are small and elegant, but rather difficult to get your head around. The challenge is not the coding, it is the thinking process.

In iterative looping, we typically repeat a functionality over the whole dataset using indexes or items from an enumerable. In recursion we repeat the exact same functionality over a smaller and smaller (or larger and larger) dataset.

Many algorithms are elegantly implemented using recursion, for example QuickSort, and it is even so that some algorithms can only be implemented using recursion, for example traversing the entire file system in your computer. Combinatorial problems are surprisingly easy to solve using recursion, but almost impossible to solve iterative, for example the StairCase problem. I will show you as we move along.

Let's start with a simple example of a program counting down an integer from 10 to 0. We simply write this iterative in a for-loop as below.

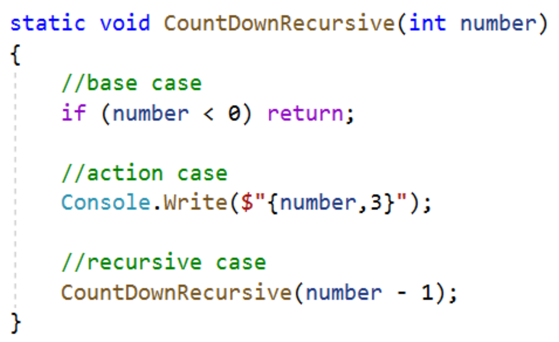


```
1 reference
static void CountDownIterative(int number)
{
    for (int i = number; i >= 0; i--)
    {
        Console.Write($"{i,3}");
    }
}
```

https://github.com/SEIDOAB-BEOP/M3_07/blob/master/M3_07_05/Program.cs

Suppose I want to write this recursively. The dataset is the integer number, and the functionality is to print out the number. Applying the exact same functionality on a smaller and smaller number, is easily done recursively.

However, if we do not stop the recursion the program will keep on printing smaller and smaller numbers until we get an execution error. The decision to stop the recursion is called a base case. Below I stop the recursion at the base case of number equals 0.



```
static void CountDownRecursive(int number)
{
    //base case
    if (number < 0) return;

    //action case
    Console.Write($"{number,3}");

    //recursive case
    CountDownRecursive(number - 1);
}
```

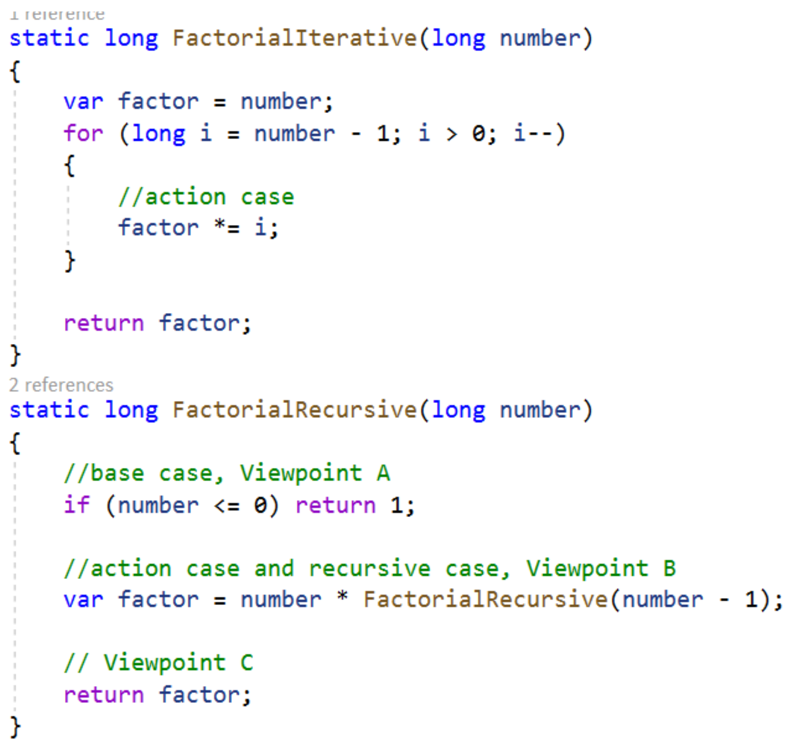
https://github.com/SEIDOAB-BEOP/M3_07/blob/master/M3_07_05/Program.cs

Comparing the two programs, you see both are equally simple in terms of code structure, while the recursive implementation requires a different thought process.

When writing recursive programs, it helps to, in addition to the base case, identify the activity case and recursive case. The recursive case is where the recursion actually takes place, and the activity case is where you do something with the dataset.

Vey often the activity case and the recursive case come together as you will see below in another example from the mathematics. Let’s make a program that calculates the factorial of a number. The factorial of 5, or 5!, equals 5*4*3*2*1 and 3! = 3*2*1 = 6.

Again, such implementation iterative is easily done in a for-loop. Implementing it recursively is as well easily done, keeping in mind that we will recursively multiply smaller and smaller numbers.

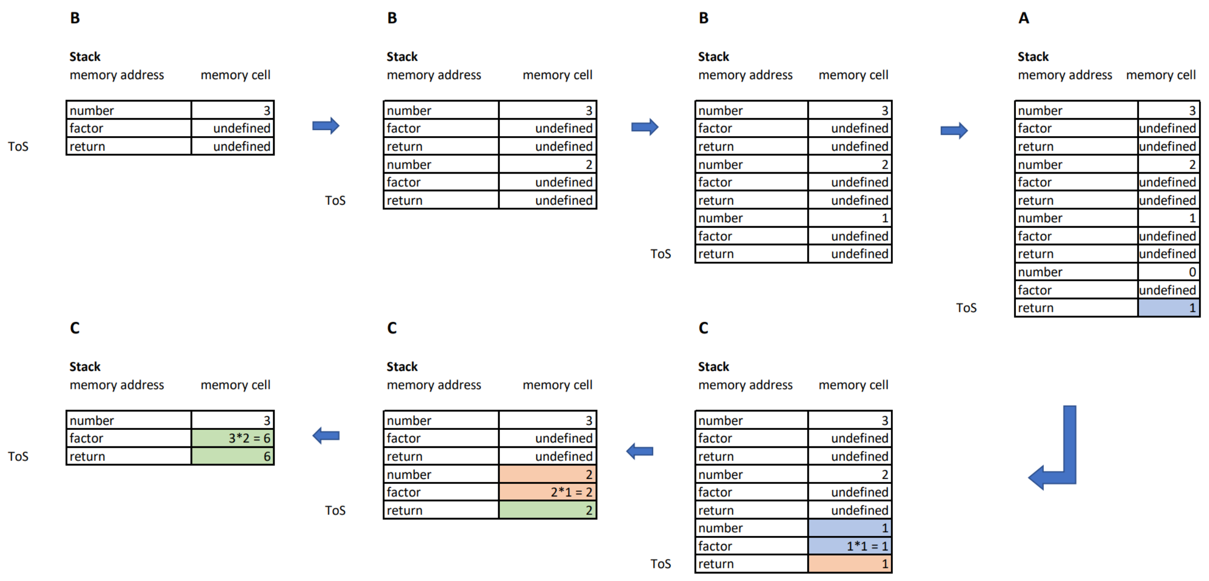


```
1 reference
static long FactorialIterative(long number)
{
    var factor = number;
    for (long i = number - 1; i > 0; i--)
    {
        //action case
        factor *= i;
    }

    return factor;
}
2 references
static long FactorialRecursive(long number)
{
    //base case, Viewpoint A
    if (number <= 0) return 1;

    //action case and recursive case, Viewpoint B
    var factor = number * FactorialRecursive(number - 1);

    // Viewpoint C
    return factor;
}
```

https://github.com/SEIDOAB-BEOP/M3_07/blob/master/M3_07_06/Program.cs

In recursive programs, remember the recursive call is like any other method call thus keeping it’s own local variables on the stack. The stack grows for each call, and shrinks when execution of each call has terminated. Therefore a recursive program without a base case will eventually run into an stack overflow exception.

Let’s go through the stack content of the factorial program when calculating 3!



B

Stack

	memory address	memory cell
	number	3
	factor	undefined
ToS	return	undefined

B

Stack

	memory address	memory cell
	number	3
	factor	undefined
	return	undefined
	number	2
	factor	undefined
ToS	return	undefined

B

Stack

	memory address	memory cell
	number	3
	factor	undefined
	return	undefined
	number	2
	factor	undefined
	return	undefined
	number	1
	factor	undefined
ToS	return	undefined

A

Stack

	memory address	memory cell
	number	3
	factor	undefined
	return	undefined
	number	2
	factor	undefined
	return	undefined
	number	1
	factor	undefined
	return	undefined
	number	0
	factor	undefined
ToS	return	1

C

Stack

	memory address	memory cell
	number	3
	factor	3*2 = 6
ToS	return	6

C

Stack

	memory address	memory cell
	number	3
	factor	undefined
	return	undefined
	number	2
	factor	2*1 = 2
ToS	return	2

C

Stack

	memory address	memory cell
	number	3
	factor	undefined
	return	undefined
	number	2
	factor	undefined
	return	undefined
	number	1
	factor	1*1 = 1
ToS	return	1

M3_07_06 memory usage for 3!

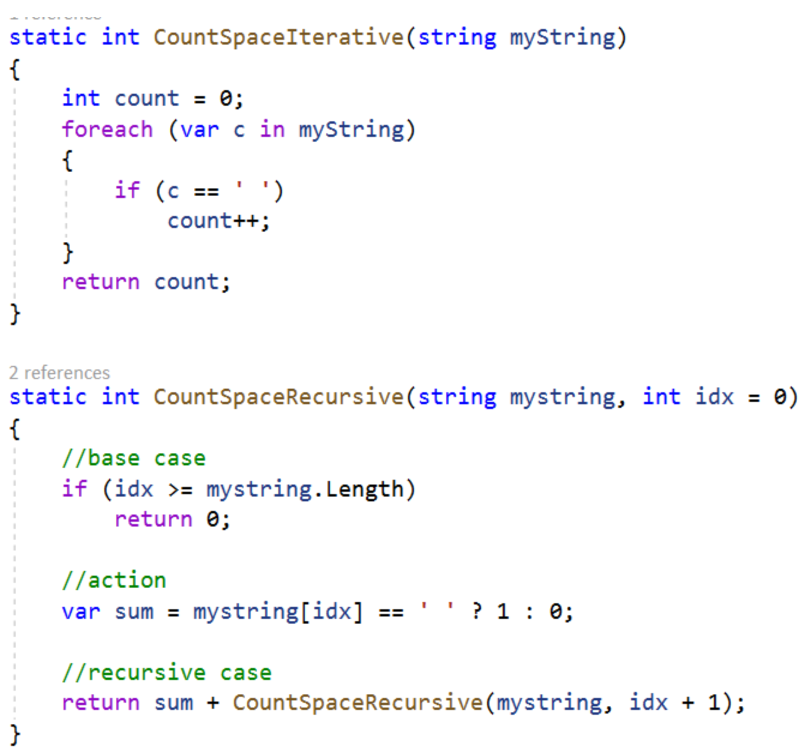
Now that we are warmed up, let’s move into a slightly more challenging example. Let’s write a program that doubles each element in an array of type int as well as calculates the sum or all elements after the doubling.

Below you see an iterative implementation and a recursive implementation where the base case, activity case and recursion case are highlighted.

```
1 reference
static long DoubleSumIterative(long[] array)
{
    long sum = 0;
    for (int i = 0; i < array.Length; i++)
    {
        array[i] *= 2;
        sum += array[i];
    }
    return sum;
}

2 references
static long DoubleSumRecursive(long[] array, int index = 0)
{
    //base case
    if (index >= array.Length) return 0;

    //action case
    array[index] *= 2;

    //recursive case
    long sum = array[index] + DoubleSumRecursive(array, index + 1);
    return sum;
}
```

https://github.com/SEIDOAB-BEOP/M3_07/blob/master/M3_07_07/Program.cs

In yet another example, I calculate the number of blanks in a string iteratively and recursively.



```
static int CountSpaceIterative(string myString)
{
    int count = 0;
    foreach (var c in myString)
    {
        if (c == ' ')
            count++;
    }
    return count;
}

2 references
static int CountSpaceRecursive(string mystring, int idx = 0)
{
    //base case
    if (idx >= mystring.Length)
        return 0;

    //action
    var sum = mystring[idx] == ' ' ? 1 : 0;

    //recursive case
    return sum + CountSpaceRecursive(mystring, idx + 1);
}
```

https://github.com/SEIDOAB-BEOP/M3_07/blob/master/M3_07_08/Program.cs

The filesystem challenge

So far both the recursive and iterative implementations have been rather simple, and you may wonder why recursion?

Let’s look at where recursion really shines. Let’s write a program that write all the directories in my filesystem. Iteratively this is easily done at one directory level. However, to go one level deeper, you need to repeat the same code. Indeed, you need to repeat the same code for each level. A lot of code, and you even don’t know how many levels you need.

```
static void FindDirectoriesIterative(string myDirectory)
{
    //Get first level directories
    Console.WriteLine(myDirectory);
    foreach (var dir1 in Directory.EnumerateDirectories(myDirectory))
    {

        Console.WriteLine($"{dir1}");

        #region get second level directories
        try
        {
            //Get second level directories, where I have access control
            foreach (var dir2 in Directory.EnumerateDirectories(dir1))
            {
                Console.WriteLine($"{dir2}");

                get third level directories
            }
        }
        catch (UnauthorizedAccessException)
        {
            continue;
        }
        #endregion
    }
}
```

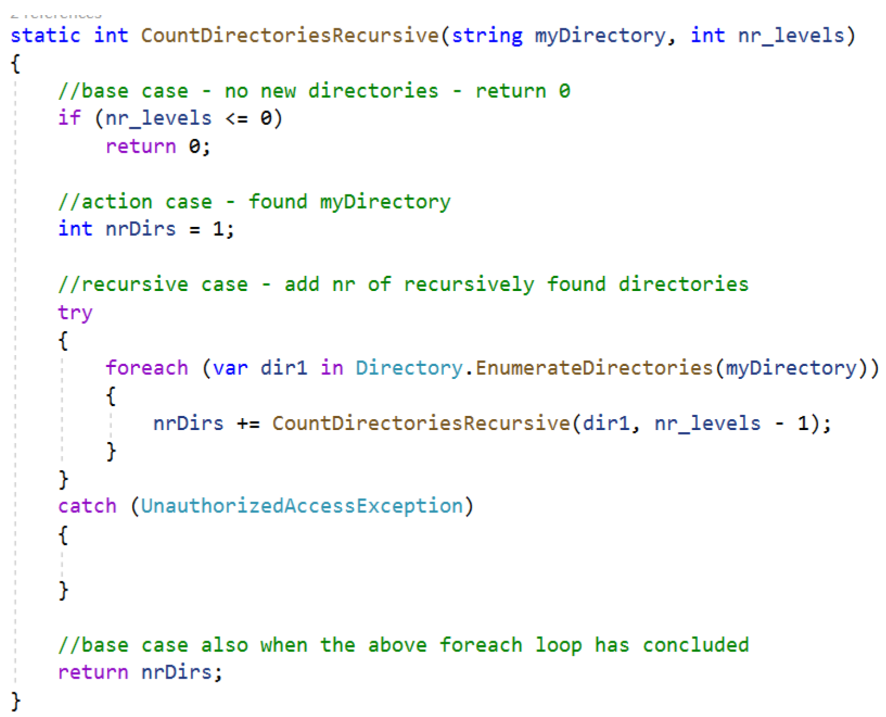
https://github.com/SEIDOAB-BEOP/M3_07/blob/master/M3_07_09/Program.cs

If I implement the same functionality recursively, it becomes a simple, elegant solution, which covers any depth of the file system!

```
static void FindDirectoriesRecursive(string myDirectory, int nr_levels)
{
    //base case
    if (nr_levels <= 0)
        return;

    //action case
    Console.WriteLine(myDirectory);

    //recursive case
    try
    {
        foreach (var dir1 in Directory.EnumerateDirectories(myDirectory))
        {
            FindDirectoriesRecursive(dir1, nr_levels - 1);
        }
    }
    catch (UnauthorizedAccessException)
    {

    }

    //base case also when the above foreach loop has concluded
}
```

https://github.com/SEIDOAB-BEOP/M3_07/blob/master/M3_07_10/Program.cs

In below example, I count the number of directories recursively.

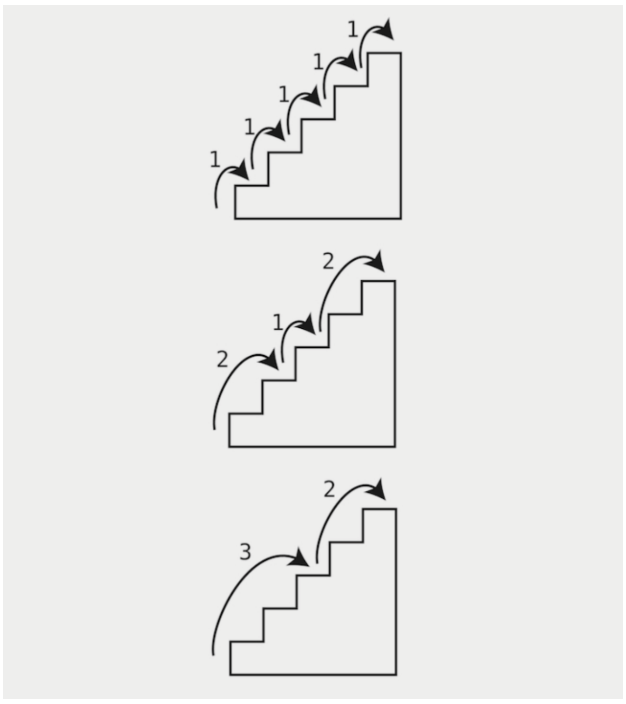


```
static int CountDirectoriesRecursive(string myDirectory, int nr_levels)
{
    //base case - no new directories - return 0
    if (nr_levels <= 0)
        return 0;

    //action case - found myDirectory
    int nrDirs = 1;

    //recursive case - add nr of recursively found directories
    try
    {
        foreach (var dir1 in Directory.EnumerateDirectories(myDirectory))
        {
            nrDirs += CountDirectoriesRecursive(dir1, nr_levels - 1);
        }
    }
    catch (UnauthorizedAccessException)
    {

    }

    //base case also when the above foreach loop has concluded
    return nrDirs;
}
```

https://github.com/SEIDOAB-BEOP/M3_07/blob/master/M3_07_10/Program.cs

The staircase challenge

To show the power of recursion in solving combinatorial problems, consider below challenge. Suppose I have a staircase with N steps. I can take one stair in a step, two stairs in a step or 3 stairs in a step climbing the staircase.



From the book “A common-sense guide to data structures and Algorithms, Jay Wengrow”

The question is, how many combinations of steps can I use to climb the staircase?

For example if I have a staircase with only one stair, I can climb it only with one 1-step which gives combination.

If I have a staircase with two stairs, I can climb it with two 1-steps or one 2-step which equals 2 combinations.

Similarly, if my staircase has 3 stairs, I can climb it with thee 1-steps, one 1-step followed by one 2-step, one 2-step followed by one 1 step, and finally one 3-step. All in all 4 combinations.

But what if my staircase is 30 steps?

Such a problem is quite challenging to solve iteratively, but with recursively it is just a few lines of code.

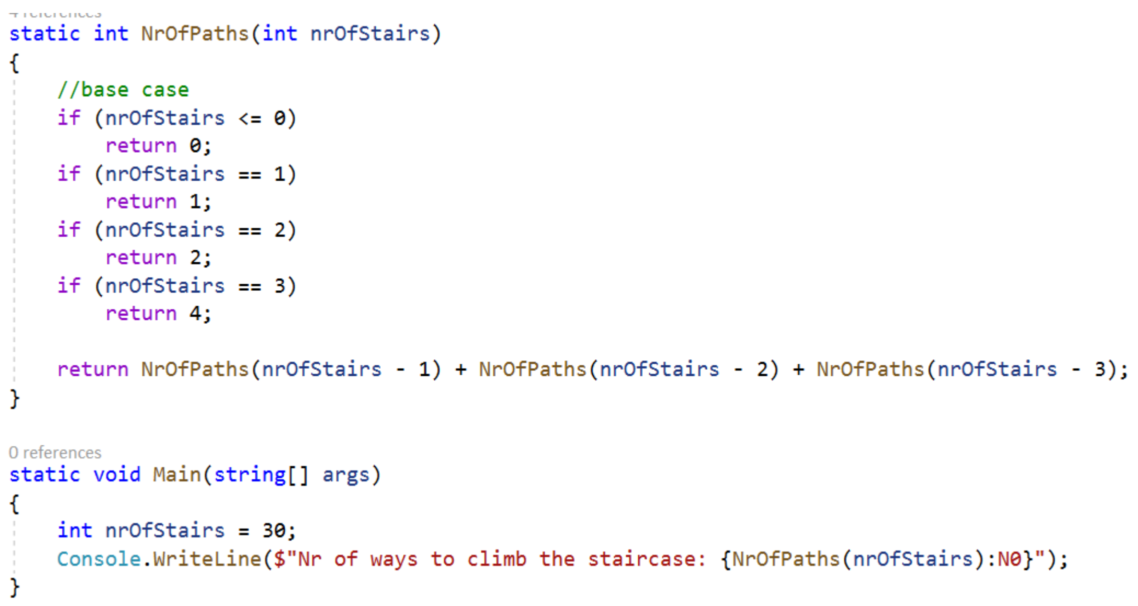
The recursiveness of the problem comes asking yourself:

- "if I start taking one 1-step, how many combinations remain in the staircase?"
- "if I start taking one 2-steps, how many combinations remain in the staircase?"
- "if I start taking one 3-steps, how many combinations remain in the staircase?"

If I have answers to all the above questions, the total number of combinations are indeed the three added together.

Thinking recursively in a smaller and smaller staircase, remember we have the solutions for a staircase or one, two , or three stairs. This gives our base cases.

Hence, the simple solution is below. Btw, for 30 stairs there are 53 798 080 combinations, while for 40 it is 23 837 527 729!



```
static int NrOfPaths(int nrOfStairs)
{
    //base case
    if (nrOfStairs <= 0)
        return 0;
    if (nrOfStairs == 1)
        return 1;
    if (nrOfStairs == 2)
        return 2;
    if (nrOfStairs == 3)
        return 4;

    return NrOfPaths(nrOfStairs - 1) + NrOfPaths(nrOfStairs - 2) + NrOfPaths(nrOfStairs - 3);
}

0 references
static void Main(string[] args)
{
    int nrOfStairs = 30;
    Console.WriteLine($"Nr of ways to climb the staircase: {NrOfPaths(nrOfStairs):N0}");
}
```

https://github.com/SEIDOAB-BEOP/M3_07/blob/master/M3_07_11/Program.cs

QuickSort

Now that I have covered recursion, let's come back to the effective sorting algorithm QuickSort. QuickSort is an algorithm using a combination of partitioning and recursion.

Partitioning is the process to rearrange an array so that at one point in the array, the pivot index, all elements at lower indexes have lower values than the element at the pivot index and all elements at higher indexes have higher values. Note that the values in the lower and higher halves are not necessarily in order, just lower or higher than the pivot point.

The algorithm of QuickSort is then:

- partition the array and get the pivot index.
- Treat the lower half and upper half of the array as subarrays
- Recursively partition each subarray and divide into smaller and smaller halves of subarrays.

With above algorithm, a subarray with just three elements will be in order after partitioning. This is the recursion base, divide the array, make the subarray smaller and smaller, treating each subarray in exactly the same way.

When a subarray has zero or one element do nothing. This becomes our base case

Hoare partitioning

The core of QuickSort is really the partitioning, and here I use a scheme called Hoare partitioning.

Hoare partitioning algorithm is:

- Pick any pivot index in the array.
- Divide the array into an upper and lower half of the pivot index
- Set a lower index at the start of the lower half and upper index at the end of the upper half

do

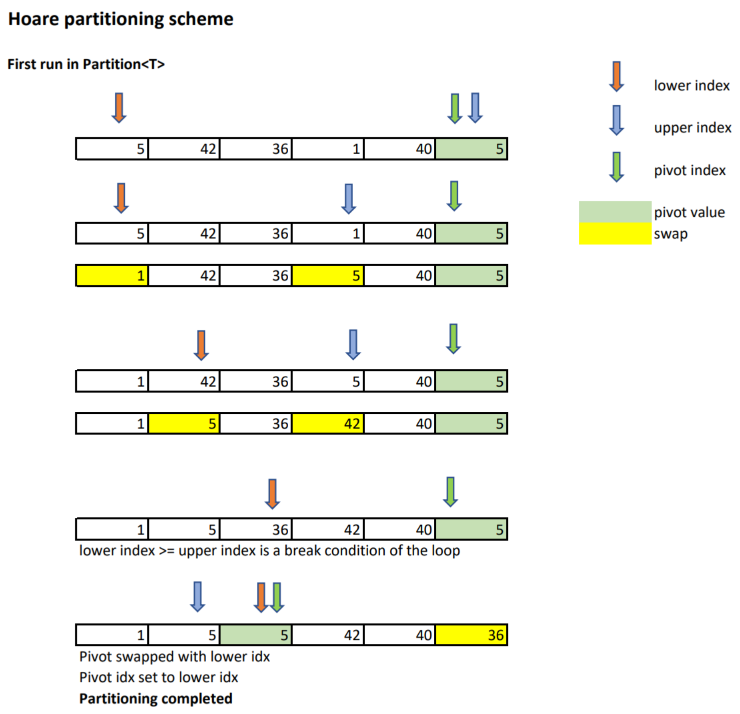
- Increment the lower index until a value higher or equal than the value of the pivot index is found
- Decrement the upper index until a value lower or equal than the value of the pivot index is found
- If lower index and upper index has crossed over each other, no swap needed and break loop.
- Otherwise, swap the values at lower index and upper index
- Increment lower index

While lower index <= upper index

- Swap the value at the pivot index with the value at lower index and increment the lower index.
- Set the pivot index to the lower index

Now, all the elements in the lower half of the array have lower values than the value at the new pivot index and all the element in the upper half have higher values.

Let’s go through it step-by-step starting with the partitioning, assuming an array of 6 elements {5, 42, 36, 1, 40, 5}.

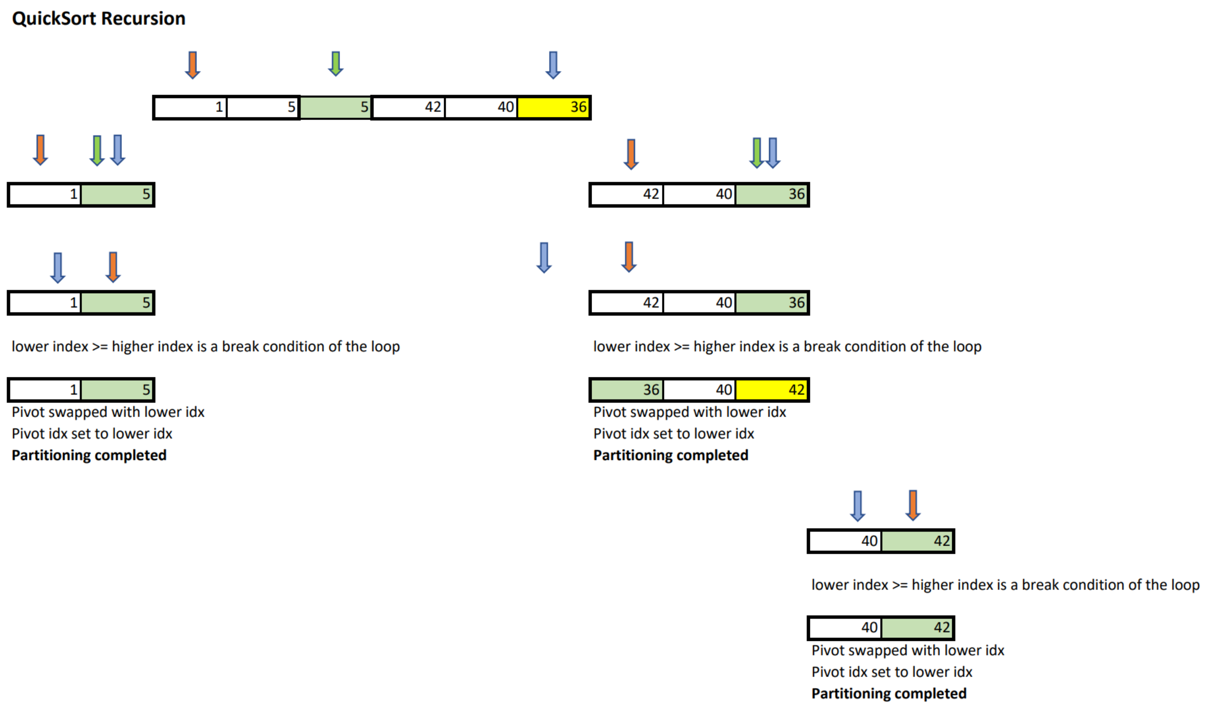

Hoare partitioning scheme
First run in Partition<T>
lower index
upper index
pivot index
pivot value
swap
5 42 36 1 40 5
1 42 36 5 40 5
1 5 36 42 40 5
lower index >= upper index is a break condition of the loop
1 5 5 42 40 36
Pivot swapped with lower idx
Pivot idx set to lower idx
Partitioning completed

M3_08_01 image of the partitioning scheme

```
public static int Partition<T>(T[] array, int lowerIdx, int upperIdx) where T : IComparable
{
    Console.Write($"\n\n----Partition run [{lowerIdx}] to [{upperIdx}]");

    //assume pivot point as set the lower part of the array just after the pivot
    int pivotIdx = upperIdx;
    upperIdx--;

    do
    {
        while (array[lowerIdx].CompareTo(array[pivotIdx]) < 0) { lowerIdx++; }
        while (array[upperIdx].CompareTo(array[pivotIdx]) > 0) { upperIdx--; }

        if (lowerIdx >= upperIdx)
            break;

        WriteArray("Before HiLo swap:", array, pivotIdx, lowerIdx, upperIdx);

        //swap them using tuple, array[lowerIdx] is now in the right place
        (array[lowerIdx], array[upperIdx]) = (array[upperIdx], array[lowerIdx]);

        WriteArray("After HiLo swap:", array, pivotIdx, lowerIdx, upperIdx);

        //array[lowerIdx] is now in the right place so advance lowerIdx
        lowerIdx++;
    }
    while (lowerIdx <= upperIdx);

    WriteArray("Before LoPiv swap:", array, pivotIdx, lowerIdx, upperIdx);

    //as a final step swap the value for the lower idx with the initial pivot and update pivotIdx
    (array[lowerIdx], array[pivotIdx]) = (array[pivotIdx], array[lowerIdx]);
    pivotIdx = lowerIdx;

    WriteArray("After LoPiv swap:", array, pivotIdx, lowerIdx, upperIdx);

    return pivotIdx;
}
```
https://github.com/SEIDOAB-BEOP/M3_08/blob/master/M3_08_01/Program.cs

QuickSort implementation

Now that we have gone through the partitioning, we are ready to recursively repeat the partitioning by dividing the array into smaller pieces. Let’s go through QuickSort.



lower index >= higher index is a break condition of the loop

Pivot swapped with lower idx
Pivot idx set to lower idx
Partitioning completed

lower index >= higher index is a break condition of the loop

Pivot swapped with lower idx
Pivot idx set to lower idx
Partitioning completed

lower index >= higher index is a break condition of the loop

Pivot swapped with lower idx
Pivot idx set to lower idx
Partitioning completed

M3_08_01 Image of Quicksort

```
public static void Sort<T>(T[] array) where T : IComparable
{
    WriteArray("\nUnsorted array:", array);
    Sort(array, 0, array.Length - 1);
    WriteArray("\nSorted array:", array);
}

3 references
private static void Sort<T>(T[] array, int lowerIdx, int upperIdx) where T : IComparable
{
    //base case
    if (lowerIdx >= upperIdx)
        return;

    //action case - arrange a pivot where all elements in the lower part are less
    int pivotIdx = Partition(array, lowerIdx, upperIdx);

    //recursive case
    Sort(array, lowerIdx, pivotIdx - 1);
    Sort(array, pivotIdx + 1, upperIdx);
}
```

https://github.com/SEIDOAB-BEOP/M3_08/blob/master/M3_08_01/Program.cs