Chapter 10. Recursion

In This Chapter

In this chapter we are going to get familiar with **recursion and its applications**. Recursion represents a powerful programming technique in which a **method makes a call to itself** from within its own method body. By means of recursion we can solve complicated **combinatorial problems**, in which we can easily exhaust different combinatorial configurations, e.g. **generating permutations** and **variations** and simulating **nested loops**. We are going to demonstrate many examples of correct and incorrect usage of recursion and convince you how useful it can be.

What Is Recursion?

We call an object **recursive** if it contains itself, or if it is defined by itself.

Recursion is a programming technique in which **a method makes a call to itself** to solve a particular problem. Such methods are called **recursive**.

Recursion is a programming technique whose correct usage leads to elegant solutions to certain problems. Sometimes its usage could considerably simplify the programming code and its readability.

Example of Recursion

Let's consider the **Fibonacci numbers**. These are the elements of the following sequence:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Each element of the sequence is formed by the sum of the previous two elements. The first two elements are equal to 1 by definition, i.e. the next two rules apply:

$F_1 = F_2 = 1$

$F_i = F_{i-1} + F_{i-2}$ (for $i > 2$)

Proceeding directly from the definition, we can implement the following **recursive method for finding the nth Fibonacci number**:

```
static long Fib(int n)
{
  if (n <= 2)
  {
      return 1;
  }
  return Fib(n - 1) + Fib(n - 2);
}
```

This example shows how simple and natural the implementation of a solution can be when using recursion.

On the other hand, it can serve as an example of how attentive we have to be while programming with recursion. Although it is intuitive, the present solution is one of the **classical examples when the usage of recursion is highly inefficient** as there are many excessive calculations (of one and the same element of the sequence) due to the recursive calls.

We are going to consider the advantages and the disadvantages of using recursion later in this chapter.

Direct and Indirect Recursion

When in the body of a method there is a call to the same method, we say that the method is **directly recursive**.

If method A calls method B, method B calls method C, and method C calls method A we call the methods A, B and C **indirectly recursive** or **mutually recursive**.

Chains of calls in indirect recursion can contain multiple methods, as well as branches, i.e. in the presence of one condition one method to be called, and provided a different condition another to be called.

Bottom of Recursion

When using recursion, we have to be totally sure that after a certain count of steps we get a concrete result. For this reason we should have one or more cases in which the solution could be found directly, without a recursive call. These cases are called **bottom of recursion**.

In the example with Fibonacci numbers the bottom of recursion is when n is less than or equal to 2. In this base case we can directly return result without making recursive calls, because by definition the first two elements of the sequence of Fibonacci are equal to 1.

If a recursive method has no base case, i.e. bottom, it will become **infinite** and the result will be **`StackOverflowException`**.

Creating Recursive Methods

When we create recursive methods, it is necessary that we break the task we are trying to solve in **subtasks**, for the solution of which we can use the same algorithm (recursively). The combination of solutions of all subtasks should lead to the solution of the initial problem.

In each recursive call the problem area should be limited so that at some point the **bottom of the recursion** is reached, i.e. breaking of each subtask must lead eventually to the bottom of the recursion.

Recursive Calculation of Factorial

The usage of recursion we will illustrate with a classic example – recursive calculation of factorial.

Factorial of **n** (written **n!**) is the product of all integers between 1 and **n** inclusive. By definition 0! = 1.

n! = 1.2.3…n

Recurrent Definition

When creating our solution, it is much more convenient to use the corresponding recurrent definition of factorial:

n! = 1, for n = 0

n! = n.(n-1)!, for n>0

Finding a Recurrent Dependence

The presence of recurrent dependence is not always obvious. Sometimes we have to find it ourselves. In our case we can do this by analyzing the problem and calculating the values of the factorial for the first few integers.

```
0! = 1
1! = 1 = 1.1 = 1.0!
2! = 2.1 = 2.1!
3! = 3.2.1 = 3.2!
4! = 4.3.2.1 = 4.3!
5! = 5.4.3.2.1 = 5.4!
```

From here you can easily see the recurrent dependability:

```
n! = n.(n-1)!
```

Algorithm Implementation

The bottom of our recursion is the simplest case n = 0, in which the value of the factorial is 1.

In the other cases we solve the problem for n-1 and multiply the result by n. Thus after a certain count of steps we are definitely going to reach the bottom of the recursion, because between 0 and n there is a certain count of integer numbers.

Once we have these substantial conditions we can write a method, which computes factorial:

```
static decimal Factorial(int n)
{
   // The bottom of the recursion
   if (n == 0)
   {
      return 1;
   }
   // Recursive call: the method calls itself
   else
   {
      return n * Factorial(n - 1);
   }
}
```

By using this method we can create an application, which reads an integer from the console computes its factorial and then prints the obtained value:

RecursiveFactorial.cs

```
using System;

class RecursiveFactorial
{
   static void Main()
   {
      Console.Write("n = ");
      int n = int.Parse(Console.ReadLine());

      decimal factorial = Factorial(n);
      Console.WriteLine("{0}! = {1}", n, factorial);
   }

   static decimal Factorial(int n)
   {
      // The bottom of the recursion
      if (n == 0)
      {
         return 1;
      }
      // Recursive call: the method calls itself
```

```
    else
    {
      return n * Factorial(n - 1);
    }
  }
}
```

Here is what the result of the execution of the application would be like if we enter 5 for n:

```
n = 5
5! = 120
```

Recursion or Iteration?

The calculation of factorial is often given as an example when explaining the concept of recursion, but in this case, as in many others, recursion is not the best approach.

Very often, if we are given a recurrent definition of the problem, the **recurrent** solution is intuitive and not posing any difficulty, while **iterative** (consecutive) solution is not always obvious.

In this particular case the implementation of the iterative solution is as short and simple, but is a bit **more efficient**:

```
static decimal Factorial(int n)
{
  decimal result = 1;

  for (int i = 1; i <= n; i++)
  {
    result = result * i;
  }

  return result;
}
```

We are going to consider the advantages and disadvantages of using recursion and iteration later in this chapter.

For the moment we should remember that before proceeding with recursive implementation we should think about an iterative variant, after which we should choose the better solution according to the situation.

Let's look at another example where we could use recursion to solve the problem. This time we are going to consider an iterative solution, too.

Simulation of N Nested Loops

Very often we have to write **nested loops**. It is very easy when they are two, three or any number previously assigned. However, if their count is not known in advance, we have to think of an alternative approach. This is the case with the following task.

Write a program that simulates the execution of **N nested loops** from **1** to **K**, where **N** and **K** are entered by the user. The result of the performance of the program should be equivalent to the execution of following fragment:

```
for (a1 = 1; a1 <= K; a1++)
  for (a2 = 1; a2 <= K; a2++)
    for (a3 = 1; a3 <= K; a3++)
      …
      for (aN = 1; aN <= K; aN++)
        Console.WriteLine("{0} {1} {2} … {N}",
          a1, a2, a3, …, aN);
```

For example, when N = 2 and K = 3 (which is equivalent to 2 nested loops from 1 to 3) and when N = 3 and K = 3, the results would be as follows:

```
                1 1                 1 1 1
                1 2                 1 1 2
                1 3                 1 1 3
   N = 2        2 1     N = 3       1 2 1
   K = 3   ->   2 2     K = 3  ->   …
                2 3                 3 2 3
                3 1                 3 3 1
                3 2                 3 3 2
                3 3                 3 3 3
```

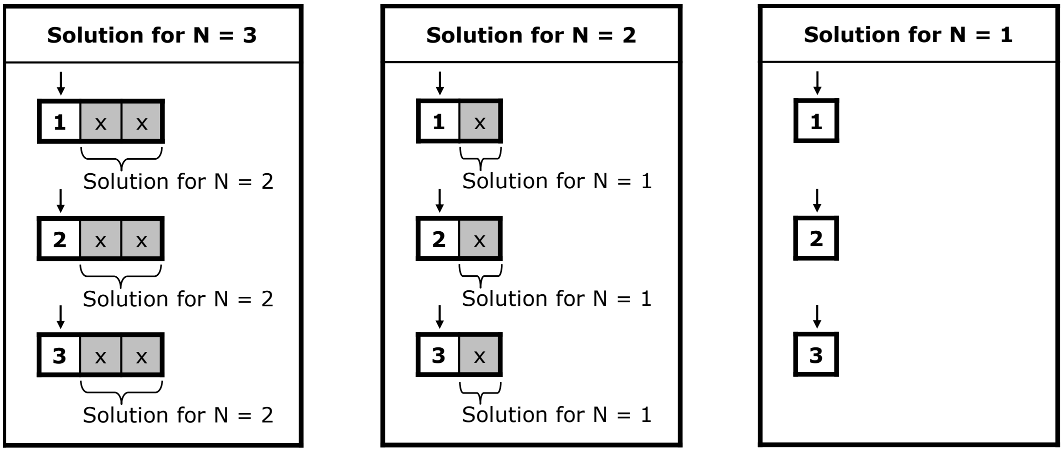
The algorithm for solving this problem is not as obvious as in the previous example. Let's consider two different solutions – one **recursive**, and one **iterative**.

Each row of the result can be regarded as ordered sequence of N numbers. The first one represents the current value of the counter of the loop, the second one – of the second loop, etc. On each position we can have value between 1 and K. The solution of our task boils down to finding all ordered sequences of N elements for N and K given.

Nested Loops – Recursive Version

If we are looking for a recursive solution to the problem, the first problem we are going to face is finding a recurrent dependence. Let's look more carefully at the example from the assignment and put some further consideration.

Notice that, if we have calculated the answer for N = 2, then the answer for N = 3 can be obtained if we put on the first position each of the values of K (in this case from 1 to 3), and on the other two positions we put each of the couples of numbers, produced for N = 2. We can check that this rule applies for numbers greater than 3.



This way we have obtained the following dependence – starting from the first position, we put on the current position each of the values from **1** to **K** and continue **recursively** with the next position. This goes on until we reach position N, after which we print the obtained result (**bottom of the recursion**). Here is how the method looks implemented in C#:

```
static void NestedLoops(int currentLoop)
{
   if (currentLoop == numberOfLoops)
   {
      PrintLoops();
      return;
   }

   for (int counter=1; counter<=numberOfIterations; counter++)
   {
      loops[currentLoop] = counter;
      NestedLoops(currentLoop + 1);
   }
}
```

We are going to keep the sequence of values in an array called `loops`, which would be printed on the console by the method **`PrintLoops()`** when needed.

The method **`NestedLoops(…)`** takes one parameter, indicating the position in which we are going to place values.

In the loop we place consecutively on the current position each of the possible values (the variable **numberOfIterations** contains the value of **K** entered by the user), after which we call recursively the method **NestedLoops(…)** for the next position.

The bottom of the recursion is reached when the current position becomes **N** (the variable **numberOfIterations** contains the value of **N**, entered by the user). In this moment we have values on all positions and we print the sequence.

Here is a complete implementation of the **recursive nested loops** solution:

RecursiveNestedLoops.cs

```
using System;

class RecursiveNestedLoops
{
  static int numberOfLoops;
  static int numberOfIterations;
  static int[] loops;

  static void Main()
  {
    Console.Write("N = ");
    numberOfLoops = int.Parse(Console.ReadLine());

    Console.Write("K = ");
    numberOfIterations = int.Parse(Console.ReadLine());

    loops = new int[numberOfLoops];

    NestedLoops(0);
  }

  static void NestedLoops(int currentLoop)
  {
    if (currentLoop == numberOfLoops)
    {
      PrintLoops();
      return;
    }

    for (int counter=1; counter<=numberOfIterations; counter++)
    {
      loops[currentLoop] = counter;
```

```
            NestedLoops(currentLoop + 1);
        }
    }

    static void PrintLoops()
    {
        for (int i = 0; i < numberOfLoops; i++)
        {
            Console.Write("{0} ", loops[i]);
        }
        Console.WriteLine();
    }
}
```

If we run the application and enter for **N** and **K** respectively 2 and 4 as follows, we are going to obtain the following result:

```
N = 2
K = 4
1 1
1 2
1 3
1 4
2 1
2 2
2 3
2 4
3 1
3 2
3 3
3 4
4 1
4 2
4 3
4 4
```

In the **Main()** method we enter values for N and K, create an array in which we are going to keep the sequence of values, after which we call the method **NestedLoops(…)**, starting from the first position.

Notice that as a parameter of the array we give 0 because we keep the sequence of values in an array, and as we already know, counting of array elements starts from 0.

The method **PrintLoops()** iterates all elements of the array and prints them on the console.

Nested Loops – Iterative Version

For the implementation of an **iterative solution of the nested loops** we can use the following algorithm, which finds the next sequence of numbers and prints it at each iteration:

1. In the beginning on each position place the number 1.
2. Print the current sequence of numbers.
3. Increment with 1 the number on position N. If the obtained value is greater than K replace it with 1 and increment with 1 the value on position N – 1. If its value has become greater then K, too, replace it with 1 and increment with 1 the value on position N – 2, etc.
4. If the value on the first position has become greater than K, the algorithm ends its work.
5. Go on with step 2.

Below we propose a straightforward implementation of the described **iterative nested loops algorithm**:

IterativeNestedLoops.cs

```
using System;

class IterativeNestedLoops
{
  static int numberOfLoops;
  static int numberOfIterations;
  static int[] loops;

  static void Main()
  {
    Console.Write("N = ");
    numberOfLoops = int.Parse(Console.ReadLine());

    Console.Write("K = ");
    numberOfIterations = int.Parse(Console.ReadLine());

    loops = new int[numberOfLoops];

    NestedLoops();
  }

  static void NestedLoops()
  {
    InitLoops();
```

```
        int currentPosition;

        while (true)
        {
            PrintLoops();

            currentPosition = numberOfLoops - 1;
            loops[currentPosition] = loops[currentPosition] + 1;

            while (loops[currentPosition] > numberOfIterations)
            {
                loops[currentPosition] = 1;
                currentPosition--;

                if (currentPosition < 0)
                {
                    return;
                }
                loops[currentPosition] = loops[currentPosition] + 1;
            }
        }
    }

    static void InitLoops()
    {
        for (int i = 0; i < numberOfLoops; i++)
        {
            loops[i] = 1;
        }
    }

    static void PrintLoops()
    {
        for (int i = 0; i < numberOfLoops; i++)
        {
            Console.Write("{0} ", loops[i]);
        }
        Console.WriteLine();
    }
}
```

The methods **Main()** and **PrintLoops()** are the same as in the implementation of the recursive solution.

The **NestedLoops()** method is different. It now implements the algorithm for iterative solution of the problem and for this reason does not get any parameters, unlike in the recursive version.

In the very beginning of this method we call the method **InitLoops()**, which iterates the elements of the array and places in each position 1.

The steps of the algorithm we perform in an infinite loop, from which we are going to escape in an appropriate moment by ending the execution of the methods via the operator **return**.

The way we implement step 3 of the algorithm is very interesting. The verification of the values greater than K, their substitution with 1 and the incrementing with 1 the value on the previous position (after which we make the same verification for it too) we implement by using one while loop, which we enter only if the value is greater than K.

For this purpose we first replace the value of the current position with 1. After that the position before it becomes current. Next we increment the value on the new position with 1 and go back to the beginning of the loop. These actions continue until the value on the current position is not less than or equal to K (the variable **numberOfIterations** contains the value of K), which is when we escape the loop.

When the value on the first position becomes greater than K (this is the moment when we have to end the execution), on its place we put 1 and try to increment the value on the previous position. In this moment the value of the variable **currentPosition** becomes negative (as the first position of the array is 0) and we end the execution of the method using the operator **return**. This is the end of our task.

We can now test it whit N = 3 and K = 2, for example:

```
N = 3
K = 2
1 1 1
1 1 2
1 2 1
1 2 2
2 1 1
2 1 2
2 2 1
2 2 2
```

Which is Better: Recursion or Iteration?

If the algorithm solving of the problem is recursive, the implementation of recursive solution can be much more readable and elegant than iterative solution to the same problem.

Sometimes defining equivalent algorithm is considerably more difficult and it is not easy to be proven that the two algorithms are equivalent.

In certain cases by using recursion we can accomplish **much simpler, shorter and easy to understand solutions**.

On the other hand, recursive calls can consume much **more resources** (CPU time and memory). On each recursive call in the stack new memory is set aside for arguments, local variables and returned results. If there are too many recursive calls, a stack overflow could happen because of lack of memory.

In certain situations the recursive solutions can be much **more difficult to understand** and follow than the relevant iterative solutions.

Recursion is powerful programming technique, but we have to think carefully before using it. If used incorrectly, it can lead to inefficient and tough to understand and maintain solutions.

If by using recursion we reach a simpler, shorter and easier for understanding solution, not causing inefficiency and other side effects, then we can prefer recursive solution. Otherwise, it is better to think of iteration.

Fibonacci Numbers – Inefficient Recursion

Let's go back to the example with **finding the nth Fibonacci number** and look more carefully at the recursive solution:

```
static long Fib(int n)
{
   if (n <= 2)
   {
      return 1;
   }
   return Fib(n - 1) + Fib(n - 2);
}
```

This solution is intuitive, short and easy to understand. At first sight it seems that this is a great example for applying recursion. The truth is that this is one of the classical examples of **inappropriate usage of recursion**. Let's run the following application:

RecursiveFibonacci.cs

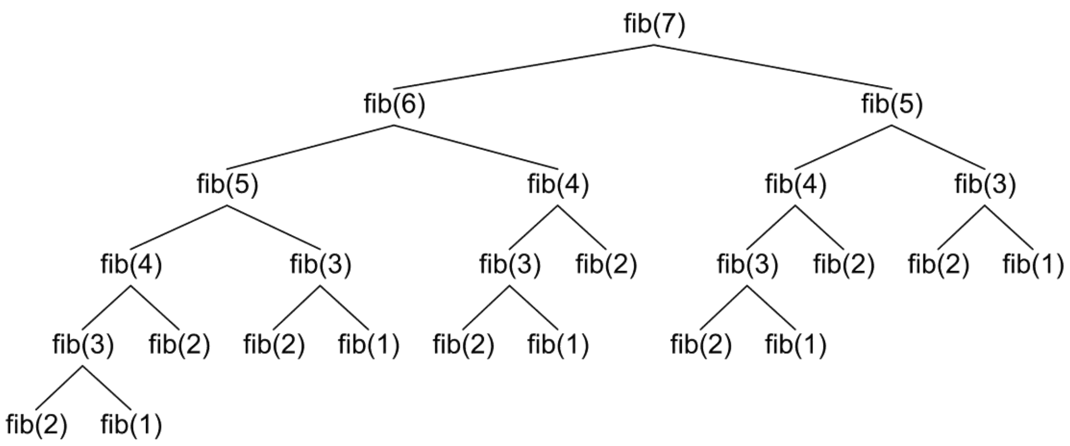
```
using System;

class RecursiveFibonacci
```

```
{
   static void Main()
   {
      Console.Write("n = ");
      int n = int.Parse(Console.ReadLine());

      long result = Fib(n);
      Console.WriteLine("fib({0}) = {1}", n, result);
   }

   static long Fib(int n)
   {
      if (n <= 2)
      {
         return 1;
      }
      return Fib(n - 1) + Fib(n - 2);
   }
}
```

If we set the value of n = 100, the calculations would take so much time that no one would wait to see the result. The reason is that similar implementation is extremely inefficient. Each recursive call leads to two more calls and each of these calls causes two more calls and so on. That's why the tree of calls **grows exponentially** as shown on the figure below.

The count of steps for computing of **fib(100)** is of the order of 1.6 raised to the power 100 (this could be mathematically proven), whereas, if the solution is linear, the count of steps would be only 100.

The problem comes from the fact that there are a lot of excessive calculations. You can notice that **fib(2)** appears below many times on the **Fibonacci tree**:



Fibonacci Numbers – Efficient Recursion

We can **optimize the recursive method** for calculating the Fibonacci numbers by remembering (saving) the already calculated numbers in an array and making recursive call only if the number we are trying to calculate has not been calculated yet. Thanks to this small **optimization technique** (also known in computer science and dynamic optimization as **memoization** (not to be confused with **memorization**) the recursive solution would work for linear count of steps. Here is a sample implementation:

RecursiveFibonacciMemoization.cs

```
using System;

class RecursiveFibonacciMemoization
{
   static long[] numbers;

   static void Main()
   {
      Console.Write("n = ");
      int n = int.Parse(Console.ReadLine());

      numbers = new long[n + 2];
      numbers[1] = 1;
      numbers[2] = 1;

      long result = Fib(n);
      Console.WriteLine("fib({0}) = {1}", n, result);
   }

   static long Fib(int n)
   {
      if (0 == numbers[n])
      {
         numbers[n] = Fib(n - 1) + Fib(n - 2);
      }

      return numbers[n];
   }
}
```

Do you notice the difference? While with the initial version if n = 100 it seems like the computation goes on forever, with the optimized solution we get an answer instantly. As we will learn later in chapter "Algorithm Complexity", the first solution runs in **exponential time** while the second is **linear**.

```
n = 100
fib(100) = 3736710778780434371
```

Fibonacci Numbers – Iterative Solution

It is not hard to notice that we can solve the problem without using recursion, by calculating the Fibonacci numbers consecutively. For this purpose we are going to keep only the last two calculated elements of the sequence and use them to get the next element. Bellow you can see an implementation of the **iterative Fibonacci numbers calculation algorithm**:

IterativeFibonacci.cs

```
using System;

class IterativeFibonacci
{
   static void Main()
   {
      Console.Write("n = ");
      int n = int.Parse(Console.ReadLine());

      long result = Fib(n);
      Console.WriteLine("fib({0}) = {1}", n, result);
   }

   static long Fib(int n)
   {
      long fn = 0;
      long fnMinus1 = 1;
      long fnMinus2 = 1;

      for (int i = 2; i < n; i++)
      {
         fn = fnMinus1 + fnMinus2;

         fnMinus2 = fnMinus1;
         fnMinus1 = fn;
      }

      return fn;
   }
}
```

This solution is as short and elegant, but does not hide risks of using recursion. Besides, it is efficient and does not require extra memory.

Concluding the previous examples we can give you the next recommendation:

⚠	**Avoid recursion, unless you are certain about how it works and what has to happen behind the scenes. Recursion is a great and powerful weapon, with which you can easily shoot yourself in the leg. Use it carefully!**

If you follow this rule, you considerably will reduce the possibility of incorrect usage of recursion and the consequences, created by it.

More about Recursion and Iteration

Generally, when we have **a linear computational process**, we do not have to use recursion, because iteration can be constructed easily and leads to simple and **efficient calculations**. An example of linear computational process is the calculation of factorial. In it we calculate the elements of the sequence in which every next element depends only on the previous ones.

What is distinctive about the linear computational processes is that on each step of the calculating **recursion is called only once**, only in one direction. Schematically, a linear computational process we can describe as follows:

```
void Recursion(parameters)
{
  do some calculations;
  Recursion(some parameters);
  do some calculations;
}
```

In such a process, when we have only one recursive call in the body of the recursive method, it is not necessary to use recursion, because **the iteration is obvious**.

Sometimes, however, we have a **branched computational process** (like a tree). For example, the imitation of N nested loops cannot be easily replaced with iteration. You have probably noticed that our iterative algorithm, which imitates nested loops, works in a completely different principle. Try to implement the same without recursion and you will see it is not easy.

Ordinarily each recursion could **boil down to iteration by using a stack** of the calls (which is created through program execution), but this is complicated and there is no benefit from doing this. Recursion has to be used when it provides simple, easy-to-understand and efficient solution to a problem, for which we have no obvious iterative solution.

In **tree-like (branched) computational processes** on each step of the recursion a couple of recursive calls are made and the scheme of calculations could be visualized as a **tree** (and not as a list like in linear calculations). For example, we saw what the tree of recursive calls would be like when we calculate the Fibonacci numbers.

A typical scheme of a tree computational process could be described with a pseudo-code in the following way:

```
void Recursion(parameters)
{
  do some calculations;
  Recursion(some parameters);
  …
  Recursion(some other parameters);
  do some calculations;
}
```

Tree computational processes could not be directly boiled down to recursive (unlike the linear processes). The case of Fibonacci is simple, because each next number is calculated via the previous, which we can calculate in advance. Sometimes, however, each next number is calculated not only via the previous, but via the next, and the recursive dependence is not so simple. In this case recursion turns out very efficient, if **implemented correctly** by avoiding duplicated calculations through **memoization**.

Use recursion for branched recursive calculations (and ensure each value is calculated only once). For linear recursive calculations prefer using iteration.

We are going to demonstrate the last statement with one classic example.

Searching for Paths in a Labyrinth – Example

We are given a **labyrinth** with a rectangular shape, consisting of N*M squares. Each square is either passable or impassable. An adventurer enters the labyrinth from its top left corner (there is the entrance) and has to reach the bottom right corner of the labyrinth (there is the exit). At each turn the adventurer can move up, down, left or right with one position and he has no right to go outside the binderies of the labyrinth, or step on impassable square. Passing through one and the same position is also forbidden (it is considered that the adventurer is lost if after a several turns he goes back to a position he has already been).

Write a computer program, which prints **all possible paths** from the beginning of the labyrinth to the exit.

This is a typical example of a problem, which can be easily solved with recursion, while with iteration the solution will be more complex and harder to implement.

Let's first draw an example in order to illustrate the problem and think about finding a solution:

s			▨			
▨	▨		▨		▨	
	▨	▨	▨	▨	▨	
						e

You can see that there are **3 different paths** from the starting position to the end, which meets the requirements of the task (movement only on passable squares and not passing twice through any of the squares). Here you can see how these three paths look like:

s	**1**	**2**	▨			
▨	▨	**3**	▨		▨	
6	**5**	**4**				
7	▨	▨	▨	▨	▨	
8	**9**	**10**	**11**	**12**	**13**	**14**

s	**1**	**2**	▨	**8**	**9**	**10**
▨	▨	**3**	▨	**7**	▨	**11**
		4	**5**	**6**		**12**
	▨	▨	▨	▨	▨	**13**
						14

s	**1**	**2**	▨			
▨	▨	**3**	▨		▨	
		4	**5**	**6**	**7**	**8**
	▨	▨	▨	▨	▨	**9**
						10

On the figure above with numbers from 1 to 14 are marked the numbers of the corresponding turns of the paths.

Paths in a Labyrinth – Recursive Algorithm

How can we solve the problem? We can consider searching from a position in the labyrinth to the end of the labyrinth as a **recursive process** as follows:

- Let the current position in the labyrinth be (row, col). In the beginning we go from the **starting position** (0, 0).
- If the current position is the searched position (N-1, M-1), then we have **found a path** and we should print it.
- If the current position is **impassable**, we **go back** (we have no right to step on it).
- If the current position is already **visited**, we **go back** (we have no right to step on it twice.
- Otherwise, we **look for a path in four possible directions**. We search recursively (with the same algorithm) a path to the exit from the labyrinth by trying to go in all possible directions:
    - We try left: position (row, col-1).
    - We try up: position (row-1, col).
    - We try right: position (row, col+1).
    - We try down: position (row+1, col).

In order to reach this algorithmic solution we **think recursively**. We have the problem "searching for a path from given position to the exit". It can be boiled down to the following four sub problems:

- searching for a path from the position on the **left** from the current position to the exit;
- searching for a path from the position **above** the current position to the exit;
- searching for a path from the position on the **right** from the current position to the exit;
- searching for a path from the position **below** the current position to the exit.

If from each possible position, which we reach, we check the four possible directions and do not move in a circle (avoid passing through positions, on which we have already stepped on), we should find a path to the exit sooner or later (if such exists).

This time the recursion is not as simple as in the previous problems. On each step we have to check whether we have reached the exit and whether we are on a forbidden position; after that we should mark the position as visited and recursively call searching in the four directions. After returning from the recursive calls we have to mark as unvisited the starting point. In informatics such crawl is known as searching with **backtracking**.

Paths in a Labyrinth – Implementation

For the implementation of the algorithm we need to represent the labyrinth in a suitable way. We are going to use a two-dimensional array of characters, as in it we are going to mark with the character ' ' (space) the passable positions, with '**e**' the exit from the labyrinth and with '*****' the impassable positions. The starting position is marked as passable position. The positions we have already visited we are going to mark with the character '**s**'. Here is how the definition of the labyrinth is going to look like for our example:

```
static char[,] lab =
{
    {' ', ' ', ' ', '*', ' ', ' ', ' '},
    {'*', '*', ' ', '*', ' ', '*', ' '},
    {' ', ' ', ' ', ' ', ' ', ' ', ' '},
    {' ', '*', '*', '*', '*', '*', ' '},
    {' ', ' ', ' ', ' ', ' ', ' ', 'e'},
};
```

Let’s try to implement the recursive method for searching in a labyrinth. It should be something like this:

```
static char[,] lab =
{
   {' ', ' ', ' ', '*', ' ', ' ', ' '},
   {'*', '*', ' ', '*', ' ', '*', ' '},
   {' ', ' ', ' ', ' ', ' ', ' ', ' '},
   {' ', '*', '*', '*', '*', '*', ' '},
   {' ', ' ', ' ', ' ', ' ', ' ', 'e'},
};

static void FindPath(int row, int col)
{
   if ((col < 0) || (row < 0) ||
      (col >= lab.GetLength(1)) || (row >= lab.GetLength(0)))
   {
      // We are out of the labyrinth
      return;
   }

   // Check if we have found the exit
   if (lab[row, col] == 'e')
   {
      Console.WriteLine("Found the exit!");
   }

   if (lab[row, col] != ' ')
   {
      // The current cell is not free
      return;
   }

   // Mark the current cell as visited
   lab[row, col] = 's';

   // Invoke recursion to explore all possible directions
   FindPath(row, col - 1); // left
   FindPath(row - 1, col); // up
   FindPath(row, col + 1); // right
   FindPath(row + 1, col); // down

   // Mark back the current cell as free
   lab[row, col] = ' ';
}

static void Main()
```

```
{
  FindPath(0, 0);
}
```

The implementation strictly follows the description from the above. In this case the size of the labyrinth is not stored in variables N and M, but is derived from the two-dimensional array lab, which stores the labyrinth: the count of the columns is **lab.GetLength(1)**, and the count of the rows is **lab.GetLength(0)**.

When entering the recursive method for searching, firstly we check if we go outside the labyrinth. In this case the searching is terminated, because going outside the boundaries of the labyrinth is forbidden.

After that we **check whether we have found the exit**. If we have, we print an appropriate message and the searching from the current position onward is terminated.

Next, we check if the current square is **available**. The square is available if the position is passable and we have not been on it on some of the previous steps (if it is not part of the current path from the starting position to the current cell of the labyrinth).

If the cell is available, we step on it. This is performed by marking it as visited (with the character '**s**'). After that we recursively search for a path in the four possible directions. After returning from the recursive search of the four possible directions, we step back from the current cell and mark it as available.

The **marking back** of the current position as available when leaving the current position is **substantial** because, when we go back, it is not a part of the current path. If we skip this action, not all paths to the exit would be found, but only some of them.

This is how the recursive method for searching for the exit from the labyrinth looks like. We should now only call the method from the **Main()** method, beginning the search from the starting position (0, 0).

If we run the program, we are going to see the following result:

```
Found the exit!
Found the exit!
Found the exit!
```

You can see that the exit has been found exactly three times. It seems that the algorithm works correctly. However, we are missing the printing of the path as a sequence of positions.

Paths in a Labyrinth – Saving the Paths

In order to print the paths we have found by our recursive algorithm, we can use an array, in which at every step we keep the direction taken (**L** – left, **U** – up, **R** – right, **D** – down). This array will keep in every moment the current path from the start of the labyrinth to the current position.

We are going to need an **array of characters** and a **counter** for the steps we have taken. The counter will keep how many times we have moved to the next position recursively, i.e. the current depth of recursion.

In order to work correctly, our program has to increment the counter when entering recursion and save the direction we have taken in the position in the array. When returning from a recursion, the counter should be reduced by 1. When an exit I found, the path can be printed (it consists of all the characters in the array from 0 to the position pointed by the counter).

What should be the **size of the array**? The answer to this question is easy; since we can enter one cell at most once, than the path would never be longer than the count of all cells (N*M). In our case the size of the maze is 7*5, i.e. the size of the array has to be 35.

Note: if you know the **List<T>** data structure is might be more appropriate to use **List<char>** instead of the array of chars. We will learn about lists in the chapter "Linear Data Structures".

This is an example implementation of the described idea:

```
static char[,] lab =
{
  {' ', ' ', ' ', '*', ' ', ' ', ' '},
  {'*', '*', ' ', '*', ' ', '*', ' '},
  {' ', ' ', ' ', ' ', ' ', ' ', ' '},
  {' ', '*', '*', '*', '*', '*', ' '},
  {' ', ' ', ' ', ' ', ' ', ' ', 'e'},
};

static char[] path =
  new char[lab.GetLength(0) * lab.GetLength(1)];
static int position = 0;

static void FindPath(int row, int col, char direction)
{
  if ((col < 0) || (row < 0) ||
    (col >= lab.GetLength(1)) || (row >= lab.GetLength(0)))
  {
    // We are out of the labyrinth
    return;
  }
```

```
    // Append the direction to the path
    path[position] = direction;
    position++;

    // Check if we have found the exit
    if (lab[row, col] == 'e')
    {
        PrintPath(path, 1, position - 1);
    }

    if (lab[row, col] == ' ')
    {
        // The current cell is free. Mark it as visited
        lab[row, col] = 's';

        // Invoke recursion to explore all possible directions
        FindPath(row, col - 1, 'L'); // left
        FindPath(row - 1, col, 'U'); // up
        FindPath(row, col + 1, 'R'); // right
        FindPath(row + 1, col, 'D'); // down

        // Mark back the current cell as free
        lab[row, col] = ' ';
    }

    // Remove the last direction from the path
    position--;
}

static void PrintPath(char[] path, int startPos, int endPos)
{
    Console.Write("Found path to the exit: ");
    for (int pos = startPos; pos <= endPos; pos++)
    {
        Console.Write(path[pos]);
    }
    Console.WriteLine();
}

static void Main()
{
    FindPath(0, 0, 'S');
}
```

To make it easier we added one more parameter to the recursive method for searching path to the exit of the labyrinth: the direction we have taken to in order to reach the current position. This parameter has no meaning when going from the starting position. For this reason in the beginning we put a meaningless value '**S**'. After that, when printing, we skip the first element of the path.

If we start the program, we are going to get the three possible paths from the beginning to the end of the labyrinth:

```
Found path to the exit: RRDDLLDDRRRRRR
Found path to the exit: RRDDRRUURRDDDD
Found path to the exit: RRDDRRRRDD
```

Paths in a Labyrinth – Testing the Program

It seems like the algorithm works properly. It remains to test it with some more examples in order to make sure we have not made a stupid mistake. We can test the program with an empty labyrinth with size 1x1, with an empty labyrinth with size 3x3, or for instance with a labyrinth in which there is no path to the exit, and in the end with an enormous labyrinth, where there are a lot of paths.

If we run the tests, we are going to be convinced that in each case the program is working correctly.

Example input (labyrinth 1 x 1):

```
static char[,] lab =
{
  {'e'},
};
```

Example output:

```
Found path to the exit:
```

You can see that the output is correct, but the path is empty (with length 0), because the starting position coincides with the exit. We could improve the visualization in this case (for example print "**Empty path**"). Example input (empty labyrinth 3x3):

```
static char[,] lab =
{
  {' ', ' ', ' '},
  {' ', ' ', ' '},
  {' ', ' ', 'e'},
};
```

Example output for the above labyrinth:

```
Found path to the exit: RRDLLDRR
Found path to the exit: RRDLDR
Found path to the exit: RRDD
Found path to the exit: RDLDRR
Found path to the exit: RDRD
Found path to the exit: RDDR
Found path to the exit: DRURDD
Found path to the exit: DRRD
Found path to the exit: DRDR
Found path to the exit: DDRUURDD
Found path to the exit: DDRURD
Found path to the exit: DDRR
```

You can check that the output is correct – these are **all the paths** to the exit.

Let's try another example input (labyrinth 5x3 without a path to the exit):

```
static char[,] lab =
{
   {' ', '*', '*', ' ', ' '},
   {' ', ' ', ' ', '*', ' '},
   {'*', ' ', ' ', '*', 'e'},
};
```

Example output:

```
(there is no output)
```

You can see that the output is correct, but again we could add a more friendly message (for example "**No exit!**"), instead of any output.

Now we have to check what would happen when we have an **enormously big labyrinth**. Here is a sample input (labyrinth with size 15x9):

```
static char[,] lab =
{{' ','*',' ',' ',' ',' ','*',' ',' ',' ',' ','*','*',' ',' '},
 {' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ','*','*','*',' ','*',' ',' ',' ',' ',' ','*','*','*','*'},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ','e'}};
```

We run the program and it starts typing paths to the exit, but it **does not end** because there are **too many paths**. Here is how a small part of the output looks like:

```
Found path to the exit:
DRDLDRRURUURRDLDRRURURRRDLLDLDRRURRURRURDDLLDLLDLLLDRRDLDRDRRURDRR
Found path to the exit:
DRDLDRRURUURRDLDRRURURRRDLLDLDRRURRURRURDDLLDLLDLLLDRRDLDRDRRRURRD
Found path to the exit:
DRDLDRRURUURRDLDRRURURRRDLLDLDRRURRURRURDDLLDLLDLLLDRRDLDRDRRRURDR
…
```

Now, let's try one last example – labyrinth with big size (15x9), in which there is no path to the exit:

```
static char[,] lab =
{
 {' ','*',' ',' ',' ',' ','*',' ',' ',' ',' ','*','*',' ',' '},
 {' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ',' ',' '},
 {' ','*','*','*',' ','*',' ',' ',' ',' ',' ','*','*','*','*'},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ','*','*',' ',' '},
 {' ',' ',' ',' ',' ','*',' ',' ',' ',' ',' ',' ',' ','*','e'},
};
```

We run the program and it **hangs, without printing anything**. It actually works very long for us to wait for it. It seems like there is a problem.

What is the problem? The problem is that the possible paths, analyzed by the algorithm are **too many** and their research takes **too much time**. Let's think how many these paths are. If a path to the exit is average 20 steps long and on each step there are 4 possible directions to be take, then 4^{20} paths have to be researched, which is a very big number. This evaluation of the count of possibilities is very inaccurate, but it gives orientation on the approximate order of possibilities.

What is the conclusion? The **backtracking** method **does not work**, when the variants are too many, and the fact they are too many can be easily concluded.

We are not going to torture you by making you find solution to the task. The problem of **searching all paths in a labyrinth has no efficient solution** for big labyrinths.

The problem has an efficient solution if it is formulated in a slightly different way: **find at least one exit from the labyrinth**. This task is far easier and can be solved with one very small correction in the sample code: when escaping the recursion, we do not mark the current cell as available. This means to delete the following lines from the code:

```
// Mark back the current cell as free
lab[row, col] = ' ';
```

We can convince ourselves that after this change the program finds out very quickly if there is no path to the exit, and if there is, it very quickly finds one of them. It is not the shortest or longest, just the first path found.

Using Recursion – Conclusions

The general conclusion from the problem searching a path in a labyrinth is already formulated: **if you do not understand how recursion works, avoid using it**!

Be careful when you write recursive methods. Recursion is a **powerful programming technique** for solving combinatorial problems (problems in which we have to go through all variants), but **it is not for everyone**. We can easily make mistakes when using recursion. You may make the program "hang", or cause stack overflow with bottomless recursion. Always look for iterative solutions, unless you deeply understand how to use recursion.

As to the problem searching shortest path in a labyrinth you can solve it elegantly without recursion with the so called **BFS (breadth-first search)**, also known as **the wavefront algorithm**, which is elementary implemented with a queue. You can read more about the "**BFS**" algorithm in this article in Wikipedia: http://en.wikipedia.org/wiki/Breadth-first_search.

Exercises

1. Write a program to simulate **n nested loops** from **1** to **n**.

2. Write a program to generate **all variations with duplicates** of **n** elements class **k**. Sample input:

```
n = 3
k = 2
```

   Sample output:

```
(1 1), (1 2), (1 3), (2 1), (2 2), (2 3), (3 1), (3 2), (3 3)
```

   Think about and implement an iterative algorithm for the same task.

3. Write a program to generate and print **all combinations with duplicates** of **k** elements from a set with **n** elements. Sample input:

```
n = 3
k = 2
```

Sample output:

```
(1 1), (1 2), (1 3), (2 2), (2 3), (3 3)
```

Think about and implement an iterative algorithm for the same task.

4. You are given a **set of strings**. Write a **recursive program**, which **generates all subsets**, consisting exactly **k** strings chosen among the elements of this set. Sample input:

```
strings = {'test', 'rock', 'fun'}
k = 2
```

Sample output:

```
(test rock), (test fun), (rock fun)
```

Think about and implement an **iterative algorithm** as well.

5. Write a **recursive program**, which prints **all subsets of a given set** of **N** words. Example input:

```
words = {'test', 'rock', 'fun'}
```

Example output:

```
(), (test), (rock), (fun), (test rock), (test fun),
(rock fun), (test rock fun)
```

Think about and implement an **iterative algorithm** for the same task.

6. Implement the **merge-sort algorithm recursively**. In it the initial array is divided into two equal in size parts, which are sorted (recursively via merge-sort) and after that the two sorted parts are merged in order to get the whole sorted array.

7. Write a recursive program, which generates and prints **all permutations of the numbers 1, 2, …, n**, for a given integer **n**. Example input:

```
n = 3
```

Example output:

```
(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)
```

Try to find an **iterative solution** for generating permutations.

8. You are given an array of integers and a number **N**. Write a recursive program that finds **all subsets** of numbers in the array, which have a **sum N**. For example, if we have the array **{2, 3, 1, -1}** and **N=4**, we can obtain **N=4** as a sum in the following two ways: **4=2+3-1; 4=3+1**.

9. You are given an array of **positive** integers. Write a program that checks whether there is one or more numbers in the array **(subset), whose sum is equal to S**. Can you solve the task **efficiently for large arrays**?

10. You are given a **matrix** with passable and impassable cells. Write a recursive program that finds **all paths between two cells** in the matrix.

11. Implement the algorithm **BFS** (breadth-first search) for finding the **shortest path in a labyrinth**.

12. Modify the previous program to check **whether a path exists between two cells** without finding all possible paths. Test the program with a matrix 100x100 filled only with passable cells.

13. You are given a matrix with passable and impassable cells. Write a program that finds the **largest area of neighboring passable cells**.

14. Write a recursive program that **traverses the whole hard disk C:\ recursively** and prints all folders and files.

Solutions and Guidelines

1. Create a **recursive method Loops(int k)**, perform a **for**-loop from **1** to **n** and make a recursive call **Loops(k-1)** in the loop. The bottom of the recursion is when **k < 0**. Initially invoke **Loops(n-1)**.

2. The **recursive solution** is to modify the algorithm for **generating N nested loops**. In fact you need **k nested loops from 1 to n**.

   The **iterative solution** is as follows: start from the **first variation** in the lexicographical order: **{1, 1, …, 1} k times**. To **obtain the next variation**, **increase the last number**. If it becomes greater than **n**, change it to 1 and increase the next number on the left. Do the same on the left until the first number goes greater than **n**.

3. Modify the algorithms from **the previous problem** and always keep each number equal or greater than the number on the left of it. The easiest way to achieve this is to **generate k nested loops from 1 to n** and print only these combinations in which each number is greater or equal than the number on its left. You may optimize this approach to get generate directly an increasing sequence for better performance.

4. Let the strings' count be **n**. Use the implementation of **k nested loops** (recursive or iterative) with additional limitation that **each number is greater than the previous one**. Thus you will generate all different subsets of **k** elements in the range **[0…n-1]**. For each set consider the numbers from it as indices in the array of strings and print for each

number the corresponding string. For the example above, the set **`{0, 2}`** corresponds to the strings at position 0 and position 2, i.e. **`(test, fun)`**.

The **iterative algorithm** is similar to the iterative algorithm for generating **n nested loops**, but is more complicated because it needs to guarantee that each number is greater than the number on its left.

5. You can **use the previous task** and **call it N times** in order to generate consequently the empty set (**k**=0), followed by the all subsets with one element (**k**=1), all subsets with 2 elements (**k**=2), all subsets with 3 elements (**k**=3), etc.

   The problem has another **very smart iterative solution**: run a **loop from 0 to 2^N-1** and convert each of these numbers to **binary numeral system**. For example, for N=3 you will have the following binary representations of the numbers between 0 to 2^N-1:

   ```
   000, 001, 010, 011, 100, 101, 110, 111
   ```

   Now for each binary representation take those words from the subset for which **have bit 1 on the corresponding position in the binary representation**. For instance, for the binary representation "101" take the first and the last string (at these positions there is 1) and omit the second string (at this position there is 0). Smart, isn't it?

6. In case you have any difficulties **search in Internet for "merge sort"**. You are going to find hundreds of implementations, including in C#. The challenge is to avoid allocating a new array for the result at each recursive call, because this is inefficient, and to **use only three arrays in the whole program**: two arrays to be merged merge and a third for the result from the merging. You will have to implement merging of two ranges of an array into a range of another array.

7. **Recursive algorithm**: suppose that the method **`Perm(k)`** permutes in all possible ways the elements of the array **`p[]`** at positions from **0** to **k-1** (inclusive). Firstly, initialize the array **`p`** with the numbers from 1 to N. **Implement recursively `Perm(k)`** in the following way:

   1. If **`k == 0`**, print the current permutation and exit the recursion (bottom of the recursion).
   2. Call **`Perm(k-1)`**.
   3. For each position **`i`** from **`0` to `k-1`** do the following:
      a. Swap **`p[i]`** with **`p[k]`**.
      b. Recursively call **`Perm(k-1)`**.
      c. Swap back **`p[i]`** with **`p[k]`**.

   In the beginning call **`Perm(n-1)`** to start the recursive generation.

Iterative algorithm: read in Wikipedia how to generate from given permutation the next permutation in the lexicographic order iteratively: en.wikipedia.org/wiki/Permutation#Generation_in_lexicographic_order.

8. The problem is not very different from the task with **finding all subsets among a given list of strings**. Shall it work fast enough with 500 numbers? Pay attention that we have to print **all subsets with sum N** which can be really big amount if N is very big and proper numbers exist in the array. For this reason **the task has no efficient solution**.

9. If we approach the problem by the method of generating of all possibilities, the solution **will not work for more than 20-30 numbers**. That's why we may approach it in a very different way in case the elements of the array are only positive, or are **limited in a certain range** (for example **[-50…50]**). Then we could use the following optimized algorithm based on **dynamic programming**:

   Assume we are given an array of numbers **p[]**. Let's denote by **possible(k, sum)** whether we could obtain **sum** by using only the numbers first **k** numbers (**p[0]**, **p[1]**, …, **p[k]**). Then, the following **recurrent dependencies** are valid:

   - **possible(0, sum) = true** if **p[0] == sum**
   - **possible(k, sum) = true** if **possible[k-1, sum] == true** or **possible[k-1, sum-p[k]] == true**

   The formula above shows that we can obtain **sum** from the elements of the array at positions **0** to **k** if one of the following two statements remains:

   - The element **p[k]** does not participate in the **sum** and the **sum** is obtained from the rest of the elements (from **0** to **k-1**);
   - The element **p[k]** participates in **sum** and the remainder **sum-p[k]** is obtained from the rest of the elements (from **0** to **k-1**).

   The implementation is not complex. Just calculate the recursive formulas by **recursive method**. We should be careful and not let already calculated values from the two-dimensional array **possible[,]** to be calculated twice. For this purpose we should keep for each possible **k** and **sum** the value **possible[k, sum]**. Otherwise the algorithm will not work for more than 20-30 elements.

   The regeneration of the numbers, which compose the found sum, may be implemented if we **go backwards from the sum n**, obtained from the first **k** numbers. At each step we examine how this sum can be obtained from the first **k–1** numbers (by taking the **k**th number or omitting it).

   Bear in mind that in the general case all possible sums of the numbers from the input array may be an awful lot. For instance, possible sums of 50 **int** numbers in the range [**Int32.MinValue** … **Int32.MaxValue**] are enough so that we could not sum them in whatever data structure. If, however, all numbers in the input array are positive (as in our case), we could keep the sums in the range **[1…S]** because from the rest we could not obtain sum **S** by adding one or more numbers from the input array.

If the numbers in the input array are not mandatory positive, but are **limited in a range**, then all possible sums are limited in some range too and we could use the algorithm described above. For example, if the range of numbers is from -50 to 50, then the least sum is -50*S and the greatest is 50*S.

If the numbers in the input array are random and not limited in a range, then **the problem has no efficient solution**.

You could read more about this classical optimization problem in computer science called "**Subset Sum Problem**" in the following article in Wikipedia: http://en.wikipedia.org/wiki/Subset_sum_problem.

10. Follow the algorithms described in the section "Searching for Paths in a Labyrinth". Note that you need to find **all possible paths** (not just one of them) so don't expect your program to run fast for large input data.

11. Read the article about **BFS** in Wikipedia: http://en.wikipedia.org/wiki/Breadth-first_search. There are enough explanations and sample code. In order to implement a queue in C#, just an array or the .NET system class **`System.Collections.Generics.Queue<T>`**. For the elements of the queue you could use your own structure **Point**, containing **x** and **y** coordinates, or use two queues (one for each of the coordinates). You may also check the section BFS in the chapter "Trees and Graphs".

12. Follow the algorithms described in the section "Searching for Paths in a Labyrinth". You should run some graph traversal algorithm like **Depth-First Search (DFS)** or **Breath-First Search (BFS)**. You may read about them in Internet or check the sections about DFS and BFS in the chapter "Trees and Graphs". Your program should visit each cell at most once and should be fast, even on large matrices (like 1,000 x 1,000).

13. The same like the **previous exercise**: use DFS or BFS. By a recursive traversal or BFS traversal, find the areas of neighbor cells in the matrix one after another and mark each area's cells as visited. Do not visit again a visited cell. From all the areas found, remember the largest.

14. For each folder (starting from **`C:\`**) print the name and the files from the current folder and **call a recursion** for each subfolder. The problem is solved as example in the sections DFS and BFS in the chapter "Trees and Graphs". Your program may crash with **`UnauthorizedAccessException`** in case you do not have access permissions for some folders on the hard disk. This is typical for some Windows installations so you could start the traversal from another directory or catch the exception (see the "Catching Exceptions" section in the "Exception Handling" chapter).