1. Backtracking Algorithm

There are certain classes of problems that are considered combinatorial. In solving these problems we may need to conduct an exhaustive search of the problem space satisfying the problem's constraints (i.e. finding a combination of a finite set of objects that satisfies given conditions). On these occasions, we sometimes use a backtracking algorithm (BA), this commits a value to the problem, checks if it satisfies the constraint and later removes the value (i.e. backtracks) and tries another one.

Algorithm 1 presents a pseudocode for backtracking. At line 2 it checks if it reached the end. If not, then at line 3 it computes the list of all possible next values (assign to variable pv). Subsequently, for each p in the list pv, it initially commits p, then recursively call itself to deals with the subsequent positions. At line 7 (post recursive call) it checks to see if it reached the end [1] and if a solution has been found (i.e. the problem is solved) it prints the solution, and then uncommits p in search of (other) solutions, notice this algorithm will find all possible solutions to the problem.

In the algorithm we used the following 2 functions:

1. **possibleValues(n)** this function computes all possible next moves from current position n
2. **iSsolved()** is a Boolean function that checks if a solution is found

as such, both these functions are problem specific and, should be designed to avoid unnecessary computation.

```
   input: n current position
1  Backtracking (n)
2  if n < end then
3      pv ← possibleValues(n)
4      foreach p in pv do
5          commit(p)
6          Backtracking (n + 1)
7          if n+1 == end and iSsolved() then
8              printSolution()
9          end
10         uncommit(p)
11     end
12 end
```

Algorithm 1: Backtracking Algorithm

[1] We assume the Boolean ***and*** operator computes short-circuit evaluation, as such, it is important to have the (n+1 == end) test on the left-hand side, as it is presumably cheaper computationally (both in terms of time and space) to compute than the function iSsolved.

Usually, when we talk about an algorithm or a pseudocode we mean an instruction list that can easily be translated into a programming language code. For example, when we refer to a sorting algorithm such as the bubble sort, it describes how to sort a list in a particular manner. However, the backtracking algorithm presented above is a template and needs to be adapted to a specific problem. Next, we will show how to use the template to write a backtracking algorithm to obtain Latin squares of size 3×3.

2. Latin Square

For a simple concrete example, we can apply the BA to solve a version of the Latin square problem, a square matrix in which the sum of each row, column and the two main diagonals are all equal.

In the following algorithm the vector is a list of size 9, representing the Latin square, we initialised the vector by setting its elements to zero (i.e.$vector = [0, 0, 0, 0, 0, 0, 0, 0, 0]$). This vector is available to both algorithm 3 and 4 (e.g. as a class variable)

```
input: n current position
latinSquare (n)
if n < 9 then
    pv ← possibleValues(n)
    foreach p in pv do
        vector[n] = p
        latinSquare (n + 1)
        if n+1 == 9 and iSsolved() then
            printVector()
        end
        vector[n] = 0
    end
end
```

Algorithm 2: Latin Square

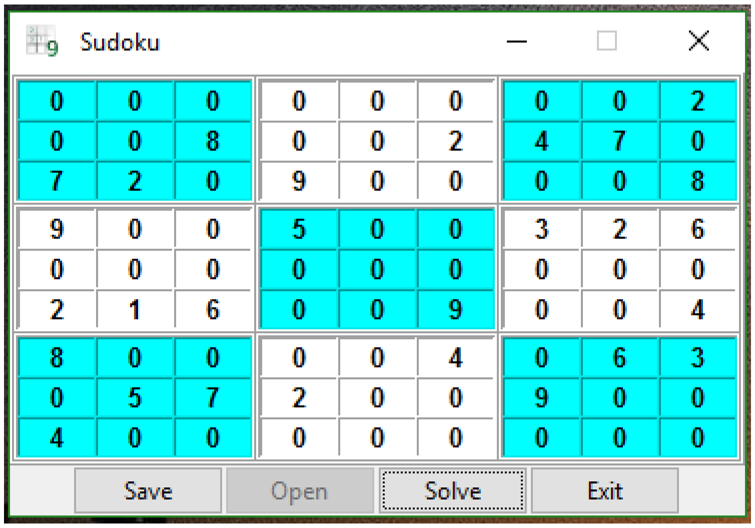
Our main concerns when writing functions ***possibleValues*** and ***isSolved*** are to make sure (other than satisfying the problem's constraints) that they should also be efficient and avoid unnecessary computation. For example, notice how ***isSolved*** is written to stop and return False if any row, column or the two diagonals have a different sum. The function ***possibleValues*** is a simple function for this problem, we merely have to find numbers in the range $1 \leq x \leq 9$ that we have not used so far.

```
input  : $n$ current position
output: $r$ = list of all possible values for position n
1 possibleValues ($n$)
2 r $\leftarrow [x \mid 1 \leq x \leq 9 \, and \, x \notin vector]$
3 return $r$
```

Algorithm 3: possibleValues

```
output: $r$ = returns a Boolean value
1  isSolved ()
2  sum $\leftarrow vector[0] + vector[1] + vector[2]$
3  if if $vector[3] + vector[4] + vector[5] \neq sum$ then
4      return False
5  else if $vector[6] + vector[7] + vector[8] \neq sum$ then
6      return False
7  else if $vector[0] + vector[3] + vector[6] \neq sum$ then
8      return False
9  else if $vector[1] + vector[4] + vector[7] \neq sum$ then
10     return False
11 else if $vector[2] + vector[5] + vector[8] \neq sum$ then
12     return False
13 else if $vector[0] + vector[4] + vector[8] \neq sum$ then
14     return False
15 else if $vector[2] + vector[4] + vector[6] \neq sum$ then
16     return False
17 else
18     return True
19 end
```

Algorithm 4: isSolved



0	0	0	0	0	0	0	0	2
0	0	8	0	0	2	4	7	0
7	2	0	9	0	0	0	0	8
9	0	0	5	0	0	3	2	6
0	0	0	0	0	0	0	0	0
2	1	6	0	0	9	0	0	4
8	0	0	0	0	4	0	6	3
0	5	7	2	0	0	9	0	0
4	0	0	0	0	0	0	0	0

Figure 1: An example of a puzzle

3. Sudoku

We have all seen Sudoku puzzles. To solve a Sudoku puzzle each row, each column and also each 3×3 box must contain numbers from 1 to 9. In this section, we adapt the BA to solve Sudoku. While for the Latin square problem (of 3×3 size) we used a simple data structure to demonstrate how a generic backtracking algorithm can be adapted to find the solutions, for the more complex Sudoku a simple linear list to present the puzzle would get very messy. Instead, we use a 9×9 (2-dimensional array puzzle) to represent the Sudoku. First we initialise every cell of the puzzle by value 0. To solve a particular Sudoku we have to input the given numbers, these given numbers the constraints of the puzzle and in figure 1 all the non-zero entries are the constraints of this particular puzzle.