2ª Aula Prática – Algoritmos Gananciosos e de Retrocesso – Resolução parcial

1. Labirinto

a.

```
bool Labirinth::findGoal(int x, int y)
{
   initializeVisited();
   return findGoalRec(x,y);
}

// Auxiliary recursive function
bool Labirinth::findGoalRec(int x, int y)
{
   // Check if this position is worth visiting (limits checking could
   // be omitted because the labyrinth is surrounded by walls)
   if (x < 0 || y < 0 || x >= 10 || x >= 10
      || labirinth[y][x] == 0 || visited[y][x])
         return false;

   // Mark as visited
   visited[y][x] = true;

   // Check if the exit was reached
   if (labirinth[y][x] == 2) {
      cout << ": Reached the goal!" << endl;
      return true;
   }

   // Try all the adjacent cells
   return findGoalRec(x-1, y) || findGoalRec(x+1, y)
         || findGoalRec(x, y-1) || findGoalRec(x, y+1);
}
```

b. T(n)=O(n^2) no pior caso, em que n é dimensão do labirinto (neste caso n=10), pois cada célula só é visitada no máximo uma vez.

2. Sudoku

a.

```
/**
 * Solves the Sudoku, that is, fills in on all the empty cells,
   satisfying the Sudoku constraints.
 * Returns true if succeeded and false otherwise.
 * Follows a greedy algorithm with backtracking.
 */
bool Sudoku::solve()
{
   if (isComplete())
      return true; // success, terminate

   // Greedy approach: searches the best cell to fill in
   // (with a minimum number of candidates)
   int i, j;
   if ( ! findBestCell(i, j) )
      return false; // impossible, backtrack
```

```
   // Tries all the possible candidates in the chosen cell
   for (int n = 1; n <= 9; n++)
      if (accepts(i, j, n)) {
         place(i, j, n);
         if (solve())
          return true; // success, terminate
         clear(i, j);
      }

   return false; // impossible, backtrack
}

/**
 * Searches the best cell to fill in - the cell with
 * a minimum number of candidates.
 * Returns true if found and false otherwise (Sudoku impossible).
 */
bool Sudoku::findBestCell(int & best_i, int & best_j)
{
   best_i = -1, best_j = -1;
   int best_num_choices = 10; // above maximum

   for (int i = 0; i < 9 ; i++)
      for (int j = 0; j < 9; j++)
         if (numbers[i][j] == 0) {
            int num_choices = 0;
            for (int n = 1; n <= 9; n++)
               if (accepts(i, j, n))
                  num_choices++;

            if (num_choices == 0)
               return false; // impossible

            if (num_choices < best_num_choices) {
               best_num_choices = num_choices;
               best_i = i;
               best_j = j;
               if (num_choices == 1) // cannot improve
                  return true;
            }
         }

   return best_i >= 0;
}

/**
 * Checks if the cell at line i, column j accepts number n
 */
bool Sudoku::accepts(int i, int j, int n)
{
   return !lineHasNumber[i][n]
          && !columnHasNumber[j][n]
          && !block3x3HasNumber[i / 3][j / 3][n];
}

/**
 * Fills in the cell at line i, column j with number n.
 * Also updates the cell counter.
 */
void Sudoku::place(int i, int j, int n)
{
   numbers[i][j] = n;
```

```
    lineHasNumber[i][n] = true;
    columnHasNumber[j][n] = true;
    block3x3HasNumber[i / 3][j / 3][n] = true;
    countFilled++;
}

/**
 * Clears the cell at line i, column j.
 * Also updates the cell counter.
 */
void Sudoku::clear(int i, int j)
{
    numbers[i][j] = 0;
    lineHasNumber[i][n] = false;
    columnHasNumber[j][n] = false;
    block3x3HasNumber[i / 3][j / 3][n] = false;
    countFilled--;
}

/**
 * Checks if the Sudoku is completely solved.
 */
bool Sudoku::isComplete()
{
    return countFilled == 9 * 9;
}
```