__Compilation instructions:__
Submission executable “KenKenPuzzleSolver” is compiled in the following environment.

OS and Architecture:
SunOS aludra.usc.edu 5.10 Generic_138888-07 sun4v sparc SUNW,T5240

Compiler version:
g++ 4.2.1

Compiler location:
/usr/usc/bin/g++

Compilation Steps:
```
g++  KenKenPuzzleSolver.c –o KenKenPuzzleSolver
```

Execution Steps:
```
./KenKenPuzzleSolver input.txt output.txt
```

__Problem1:__
a) Briefly explain, describe and illustrate your method of representing KenKen puzzles, including the board and constraints. Which aspects are explicitly represented and which, if any, are implicitly represented?
Following briefs the design of the program:
Important data structures:

- Arrray KenKenBoard[MAXBOARDSIZE][MAXBOARDSIZE] represents the game board.
- MAXBOARDSIZE – represents the maximum size of the board supported. As of now 9*9 is the maximum board size input accepted by the program.
- Array cageOperator[PUZZLEARRAYSIZE] captures the operators found in cages of given KenKen board.
- Array cageIndex[PUZZLEARRAYSIZE][PUZZLEARRAYSIZE] associates the cells with corresponding cage numbers.
- Array cageValue[PUZZLEARRAYSIZE] captures values found in cages of given KenKen board.

Constraints:

- isRowColConstraintSatisfied(int rowIndex,int colIndex,int cellValue) checks uniqueness constraint for a given value across its row and column.
- isCageConstraintSatisfied(int rowIndex,int colIndex,int cellValue) checks whether the given value satisfies the arithmetic operations such as +, -, * and % represented by the corresponding cage.
- exploreGame(int rowIndex,int colIndex,int cellValue) explores the game state for given cellValue and check for both the RowCol and CageConstraints. If both the conditions are satisfied, then the algorithm proceeds to next cell otherwise it backtracks and try different value for the same cell.

Explicit information:
Following aspects are explicitly captured in game description.

- Size of the board and number of elements in the board

- Row and column uniqueness constraint
- Cage arithmetic and . operator constraint
- Initial state of the KenKenBoard which is all cell empty with cage information captured.
- Goal state – a possible arrangement of cell values that satisfies the given constraints
- Successor function is assigning a value to a given cell and check for the constriants

Implicit Information:
Following aspects are implicitly represented.

- A cell value can never be 0
- Values across the diagonal needn't satisfy uniqueness constraint
- There is no limit on number of cages. Entire board could be a cage or each cell could be a cage with . operator everywhere.
- There is no restriction on position of values in order to satisfy the cage operator constraint. For example if the two cells (0,0) and (0,1) are part of a cage with '+' operator and value 5. The two numbers that adds up to 5 can be either in cell(0,0) or cell(0,1) as long as it satisfy the row column uniqueness property.
- If the cage operator is *, then none of the cell values in that cage can be greater than the resultant cage value.
- If the cage operator is *, then resultant cage value must be divisible by each cell in that cage.
- If the cage operator is +, then none of the cell values in that cage can be greater than the resultant cage value.
- If empty cells found in cage with %, then say if A/B = C (where C is resultant cage value), then B * C <=boardSize or C/A == 0
- There are no negative and decimal cell values. All the values are integers.
- Set of next possible states in the game tree depend on the current value being assigned to the cell. Note that the possible states are not totally explicit like a fixed graph.

b) Briefly explain why a naïve search approach such as those represented by the tree-search algorithms in Chapter 3 would be, at best, inefficient for solving a KenKen puzzle (what property of CSPs is being ignored?). Why might a backtracking algorithm provide a better solution?

As given in chapter5, the branching factor for tree-search algorithm like breadth first search is extremely high. The total number of leaves of such a tree would be $n!d^n$. This is not only inefficient in terms of time and space complexity but also unnecessary. The simple reason is CSP has commutativity i.e. the order of applying the variable values has no effect on the outcome. Thus instead of considering all different values for a given variable like BFS, backtracking chose a potentially right value that satisfy the constraint and move in depth exploring the game with that value. If the chosen value doesn't lead to solution, the entire branch would be pruned for further exploration. This saves the time and space.

<u>Problem2:</u>

(a) How many solutions are possible for given puzzle?

There is only one possible solution as given below.

Figure1: Solution for Problem2(a)

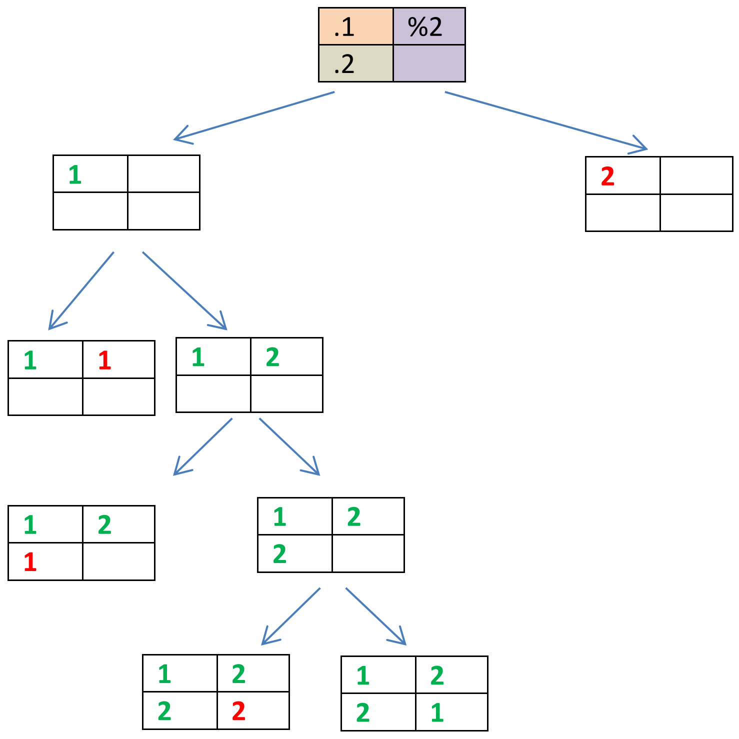
1	2
2	1

(b) Show a complete trace puzzle boards that were considered by the backtracking procedure, from the root node.

Assumptions:

- Root node is the given KenKen input board.
- The given game tree consider all possible values for cell (0,0) in level1, then it considers all possible values for cell (0,1) and so on. Each level one new cell is explored in row order. Refer *Figure2*.

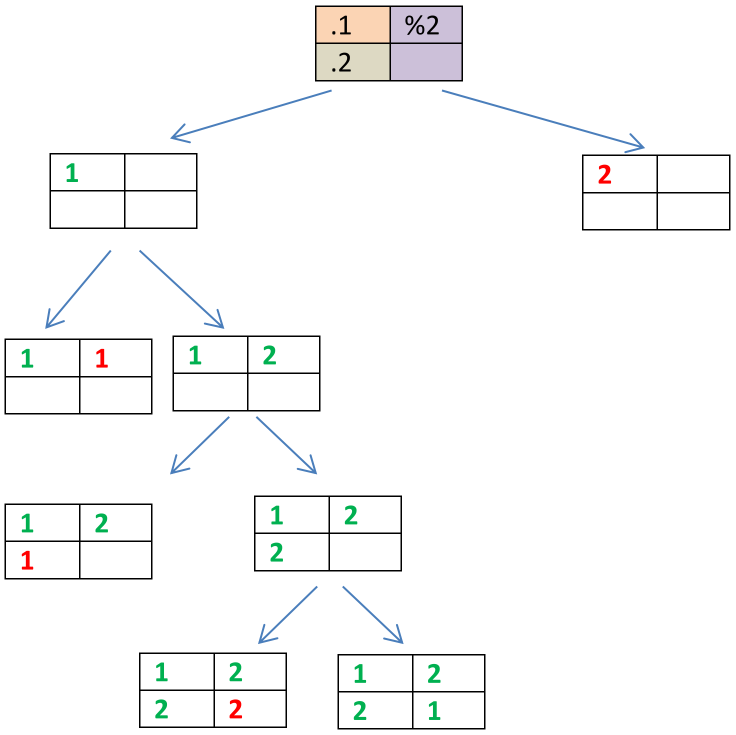
Figure2: Solution for Problem2(b)



(c) *Point out the steps where a branch of the search tree was abandoned because it was found to contain no possible solutions (i.e., the locations where backtracking occurred).*

- The cell values highlighted in red color represent the instances of search in which the entire tree was abandoned for further exploration.
- Looking at the figure, it is very evident to note down the instances of backtracking like level2 for cell(0,1)=1 and level3 cell(1,0)=1. Refer *Figure3*.

Figure3: Solution for Problem2(c)



Problem3:

(a) *Input_q3.txt solution:*

Execution:

./ KenKenPuzzleSolver Input_q3.txt output_q3.txt

```
2 5 4 3 1 6
5 3 2 4 6 1
1 2 3 6 5 4
3 6 1 2 4 5
4 1 6 5 2 3
6 4 5 1 3 2
------------------
```

```
2 5 4 3 1 6
5 3 2 4 6 1
1 2 3 6 5 4
3 6 1 2 4 5
4 1 6 5 3 2
6 4 5 1 2 3
------------------
2 5 4 3 1 6
5 3 2 4 6 1
3 2 1 6 5 4
1 6 3 2 4 5
4 1 6 5 2 3
6 4 5 1 3 2
------------------
2 5 4 3 1 6
5 3 2 4 6 1
3 2 1 6 5 4
1 6 3 2 4 5
4 1 6 5 3 2
6 4 5 1 2 3
------------------
2 5 4 3 6 1
5 3 2 4 1 6
1 2 3 6 5 4
3 6 1 2 4 5
4 1 6 5 2 3
6 4 5 1 3 2
------------------
2 5 4 3 6 1
5 3 2 4 1 6
1 2 3 6 5 4
3 6 1 2 4 5
4 1 6 5 3 2
6 4 5 1 2 3
------------------
2 5 4 3 6 1
5 3 2 4 1 6
3 2 1 6 5 4
1 6 3 2 4 5
4 1 6 5 2 3
6 4 5 1 3 2
------------------
```

```
2 5 4 3 6 1
5 3 2 4 1 6
3 2 1 6 5 4
1 6 3 2 4 5
4 1 6 5 3 2
6 4 5 1 2 3
------------------
```

(b) How many solutions are there to this puzzle?

Totally 8 solutions are possible.

<u>Problem4:</u>

(a) Execute the program as follows.

Execution:

```
./ KenKenPuzzleSolver <input_file.txt> <output_file.txt>
```

<u>Problem5:</u>

The book describes two domain-independent heuristics for CSPs. One is called the "most constrained variable" (or MRV = Minimum Remaining Values) heuristic and the other is the "least constraining value" heuristic. Explain whether these heuristics can be useful for KenKen puzzles and why or why not.

Both Minimum Remaining Value (MRV) and Least Constraining Value (LCV) are very useful heuristics for KenKen puzzle.

Following example demonstrate the use of MRV heuristic in KenKen:

Figure4: Given board for MRV discussion

.1	%2
.2	

There are 3 cages in the above board. After filling cell(0,0) and cell(1,0) with values 1 and 2 respectively, the minimum remaining values for cell(0,1) and cell(1,1) are 2 and 1 respectively. As there are only two values (1 and 2) and already a value per row being assigned, what is left out is 2 and 1. We can directly consider these values for cell(0,1) and cell(1,1) without further of division constraint.

Following example demonstrate the use of LCV heuristic in KenKen:

Figure5: Given board for LCV discussion

+ 6	+ 6	+ 3
		. 3

While exploring this board, consider the following configuration.

Figure6: Exploring solution for board in Figure5

1	2 or 3?	+3
		.3

Figure7: Possible solution for board in Figure5

1	**3**	**2**
3	**2**	**1**
2	**1**	**3**

The cell(0,1) can have two possible values 2 or 3. Which one to select? Look at the next cell(0,2) whose cage value is 3 and operator +. Thus it needs a value less than 3 to satisfy the addition operation. If we assign value 2 for cell(0,1), the game will need to back track attempting cell(0,2) with value 3. Also consider cell(2,2) with cage value 3. Thus cells (0,2) and (1,2) should not contain the value 3. If we consider these two heuristics, then the least constraining value for cell(0,1) is 3. By selecting 3 at this step, we leave the right/good to have value 2 for cell(0,2) avoiding backtracking steps. This is an example of how LCV be useful for KenKen game.

<u>Extra Credit Question 1 (5 points):</u>
Can you think of any domain-specific heuristics that might further limit the number of puzzles considered by backtracking?

- Case1: Very simple domain-specific case is a board with all cells containing '.' operator. In this case, the game should just directly assign the corresponding cage value to the cells without any backtracking business.
- Case2: Refer *Figure8* - Say if the board is 2*2 board with at least one cell containing '.' operator, then there is no need to perform any backtracking operation. Heuristic is to assign remaining values to the cells that fall in same row (cell -0,1) and column (cell -1,0) and same value to last cell (cell -1,1).

Figure8: Board considered for Case2 discussion

.1	+5

.1	2
2	1

<u>Extra Credit Question 2 (15 points):</u>
Implement a CSP heuristic from the book in your algorithm, for instance one suggested in Problem 5 or Extra Credit Question 1, and compare the performance on several puzzles vs. Non-heuristic search.
I wish I could have implemented extra heuristic but I left this question.