Programming assignment 1

1>
a>The program architecture to solve ken ken puzzle

- A structure to store Inputs, and has the following parameters
    - The value: The value to be got after performing a given operation on one or more grids
    - The operation: “+”, “*”, “-“,”/”,”.”
    - The square grids: The squares that need to satisfy the condition

  The set of inputs are connected in the form of a ***link list*** and are traversed each time by a ***validate*** function.
- A tree-node structure that has the following parameters
    - Array of values: where the size of the array equals to the total number of square grids present i.e., nXn. Each element of the array represents the value of individual squares.
    - The index: The index here represents the particular square grid whose value is under speculation.

  The nodes that pass the validity or constraint for the current situation, ***validate function*** does the required **constraint checks**, are added to a ***Stack. The stack structure is modified using the push_stack and pop_stack functions.***

It all starts with the create_tree function, which then recursively calls the parent_node function.

These are the main structures and functions involved in formulating a solution for a NxN ken ken puzzle.
The algorithm used is a ***variant of DFS, which involves backtracking***, where the backtracking is facilitated by the ***implementation of STACK***.
It’s DFS because the tree traverses by expanding a successor and exploring it before expanding the other peer successors. The characteristic of a STACK, i.e., First in Last Out, helps in bringing out the DFS nature as the rightmost child node expands and then the expansion takes place depth wise
The STACK maintains the history and helps to backtrack to the previous valid node.
When a node passes the validity test, it is pushed into the Stack
When a valid node is expanded, it is popped out of the Stack.

The aspects explicitly represented are the values of the Square grids,
The Stack elements and the Input linked list are implicit.

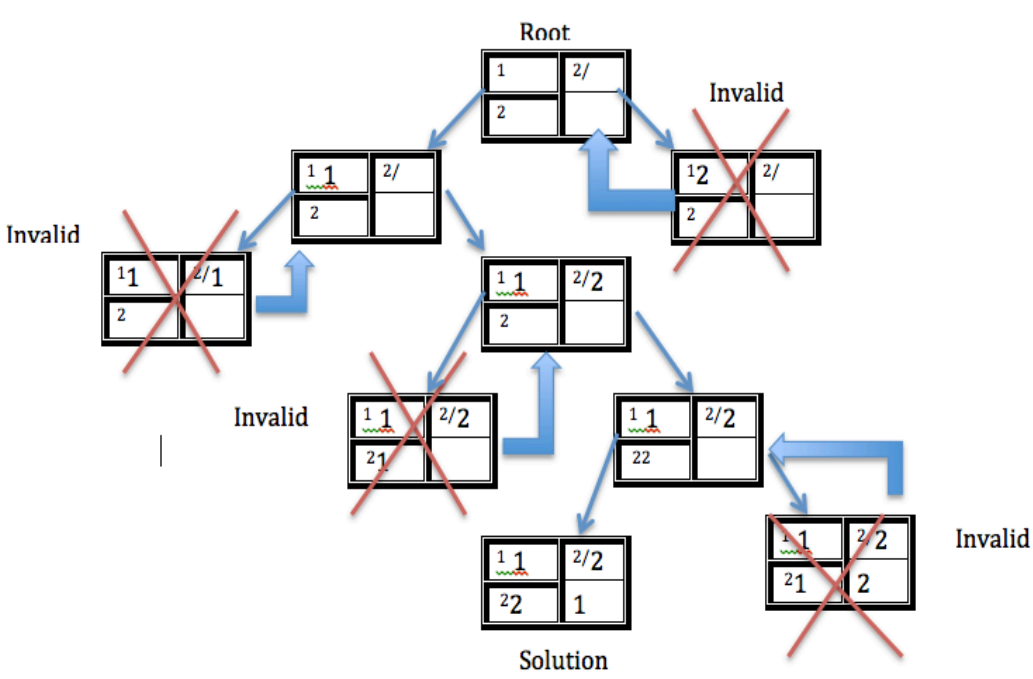
b>
Search like regular BFS or regular DFS would be inefficient, as they would lead to huge branching factors. As any of the values can be assigned to any of the square grids.
The branching factor goes on increasing exponentially and comes close to n! d^n when the assignments possible are d^n. The constraints have not been considered and every value is considered as a feasible value.

If backtracking algorithm was considered where the algorithm would backtrack when ever it finds a node which is failing on the on constraints, then this would prune the tree. It would backtrack to the previous valid node and continue its search from there. Thus this reduces the branching.

Problem 2

a> The given puzzle has just one solution

b> Trace of the puzzle board



According to my program

1>The Invalid nodes are never added to the Stack.

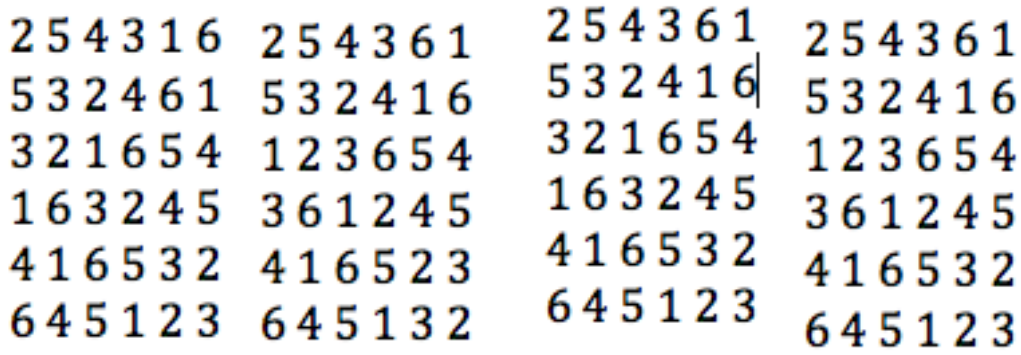
2>Valid nodes are explored till their total depth is reached, i.e., adding valid successors to the Stack. The rightmost valid successor is expanded first and then they are explored depth wise.

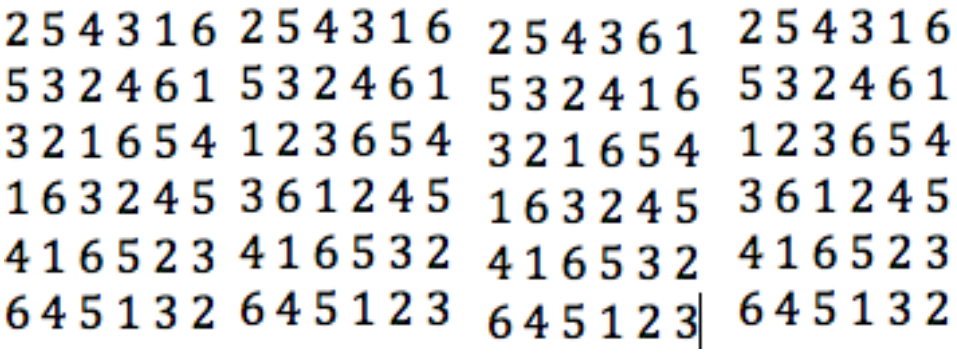
3>When the constraint fail for a node or a complete solution is found (we need to find the next set of solution if it exists), the next valid node is popped from the stack and expanded.

c>In the above figure, all the loops with a red cross on them were abandoned as they did not seem to bring any valid result.

Problem 3

a>The given 6X6 ken ken has the following 8 solutions (Output from the program)





Problem 4:
Program (For a 9X9 matrix the answer was obtained in 2 mins ad 45 seconds).

Problem 5:
The two domain independent heuristics

- Most constrained variable: Is where we choose the variable that has the fewest legal values.
  The ken ken problem could consider using MRV
  - By searching the given input for values that require no operation, i.e., with operation "." . These square grids could be automatically filled with the value they are expected to have.
  - In case of addition or multiplication, the value of each of the square grids should not be more than the final value that needs to be computed

- The least constraining values: It prefers values which rules out the limited choices for its neighbors.
    - If there are values which expect the square grids either vertically or horizontally to compute to their value, and the computation is multiplication, then the rest of the square grids could take values anything other than the factors of that value. E.g., 12 and horizontally square_grids take x, x+1, x+2 need to be multiplied to get us 4 . Then if we know that no other values other than 3,4,1 can give us 12. It would be preferred that the neighbors of these square grids do not take these values.

Ken ken could apply these different heuristics, but as the size of the matrix increases the cost of determining this heuristics may rise.

Extra credit:
Domain-specific heuristics that might further limit the number of puzzles considered by back tracking

There could be restrictions placed on the different values that can be taken by the square grids based on the domain. For e.g.,
1>The domain specific heuristics would depend on the matrix size, such as in case of an 4X4matrix it can be so achieved that a value of 3 got by "-" computation with two squares, must have its square grids values as 1, 4.

2>Finding out the factors of a given value, if the operation is multiplication, and restricting the square grids to only those factors in different combination depending on the size of the matrix.

These could be few of the validation checks related to the domain that could be considered during backtracking.