Team group

- **Ahmet Hatip**
- **Zihong Chen**
- **Ignacio Piera Fernández de Retana**

Ahmet Hatip, Zihong Chen, and Ignacio Piera Fernández de Retana together developed all three of our programs. We brainstormed together and come up with the way to do all of the solutions. Ahmet contributes the most to the simple backtracking search and the complex backtracking search; Piera Fernández de Retana contributes the most to the local search, and Zihong Chen contributes the most to the documentation.

Simple backtracking solutions

Input:

6

ABBCDD

AEECFD

GGHHFD

GGIJKK

LLIJJM

NNNOOM

A:11+

B:2/

C:20*

D:6*

E:3-

F:3/

G:240*

H:6*

I:6*

J:7+

K:30*

L:6*

M:9+

N:8+

O:2/

Important Variables:

letter_array: a 2-dimensional array storing letters:

[['A', 'B', 'B', 'C', 'D', 'D'],
 ['A', 'E', 'E', 'C', 'F', 'D'],
 ['G', 'G', 'H', 'H', 'F', 'D'],
 ['G', 'G', 'I', 'J', 'K', 'K'],
 ['L', 'L', 'I', 'J', 'J', 'M'],
 ['N', 'N', 'N', 'O', 'O', 'M']]

letter_dict: a dictionary letter that stores restrictions like '11+' for a given letter; it also stores locations of the given letter in letter_array above. For example, letter A is on [0, 0], [1, 0] of letter_array, and it has a restriction of “11+.”

{'A': ['11+', [0, 0], [1, 0]],
 'B': ['2/', [0, 1], [0, 2]],
 'C': ['20*', [0, 3], [1, 3]],
 'D': ['6*', [0, 4], [0, 5], [1, 5], [2, 5]],
 'E': ['3-', [1, 1], [1, 2]],
 'F': ['3/', [1, 4], [2, 4]],
 'G': ['240*', [2, 0], [2, 1], [3, 0], [3, 1]],
 'H': ['6*', [2, 2], [2, 3]],
 'I': ['6*', [3, 2], [4, 2]],
 'J': ['7+', [3, 3], [4, 3], [4, 4]],
 'K': ['30*', [3, 4], [3, 5]],
 'L': ['6*', [4, 0], [4, 1]],
 'M': ['9+', [4, 5], [5, 5]],
 'N': ['8+', [5, 0], [5, 1], [5, 2]],

'O': ['2/', [5, 3], [5, 4]]}

Functions:

row-checker(row, matrix):

this function checks if there are repeated numbers in rows. First, it copies the given row and does a set operation on the copy; then it compares the length of the copy with the original row. It returns true if the length of the copy is the same as the length of the original row.

column-checker(column,matrix):

this function checks if there are repeated numbers in columns. First, it copies the given column and does a set operation on the copy; then it compares the length of the copy with the original column. It returns true if the length of the copy is the same as the length of the original column.

letter-check(location,matrix,letter_array,letter_dict):

this function checks when a placed number works with parameters of Kenken puzzles;

check_no_zero(location,matrix):

this function checks if matrix[location] is zero or not;

check_no_zero(location,matrix):

this function checks if the whole matrix satisfies the Kenken puzzle;

simple_back(letter_array,matrix,letter_dict):

p.s. The solution matrix is initially filled with zeros.
While (True)
{

iteration ++
We first run check_all, which checks if the whole matrix satisfies letter, column, and row constraints. If the function returns true, we return the matrix since we find the solution to Kenken.

if check_all is not true{

if (matrix[row, column] == 0){ it means that we haven't encountered this number yet; thus we assign this value to 1. }

if(matrix[row, column] now meets row, letter, column constraints) {
we move to the next element of the matrix by updating the column index by adding 1. If the column is equal to the length of the matrix, we update the row index by adding 1 and assign 0 to the column index. }

else if(matrix[row, column] doesn't meet row, letter, column constraints)
{

we update one to matrix[row, column] and go back to the beginning of the while loop.
As long as the matrix[row, column] is not bigger than the dimension of the Kenken puzzle, we keep updating it until it meets row, letter, column constraints.

if matrix[row, column] is bigger than the length of the matrix, it means that we hit a dead end and have to backtrack. Thus, we update the column index by subtracting 1. If the column index is 0, we update the row index by subtracting 1 and assign (the length of the matrix -1) to the column index. Then we go back to the beginning of the while loop. If we go all the way back to matrix[0][0] and also reach the length of the matrix, we return NO SOLUTION.
}

}

Complex back-tracking solutions

Data:

ABB

ACC

DDE

A:2/

B:1-

C:3/

D:3/

E:2

prime_factors(n):
A function that receives a number n and returns a list of its prime factors, with 1 included inside;
3

get_choices_letters(solution,letter_dict):
A function that creates choices is an 2D array: each of its elements is a list showing possible solutions to each tile. For the data above, it creates an array:

[[[1, 2], [1, 2, 3], [1, 2, 3]],
 [[1, 2], [1, 3], [1, 3]],
 [[1, 3], [1, 3], [2]]]

This means that for the Kenken puzzle above, the first tile could only be either 1 or 2; and the second one could only be 1, 2, or 3.

If a given restriction is 100*, we know prime_factors(100) is {2,5}. Thus, any numbers whose prime factorization is not the subset of {2,5} would not fit 100*, and we should not consider it.

complex_back(letter_array,matrix,letter_dict):

3. Local search

UTILITYFUNCT(B,lettersdict,lettersB):

It receives as input a matrix, the letters of KenKen in a dictionary and the letters referred to matrix B. As output it returns the value of the Utility function

$$U = R + C + L$$

$$R \equiv Number\ of\ rows\ that\ have\ some\ equal\ numbers$$

$$C \equiv Number\ of\ columns\ that\ have\ some\ equal\ numbers$$

$$L \equiv Number\ of\ locations\ that\ don't\ meet\ constraint\ of\ operation$$

MIN_CONFLICTS(CURRENT,lettersCURRENT, lettersdic,maxsteps):

it receives as input the matrix CURRENT, the letters that correspond with each location lettersCURRENT, the letters given in a dictionary and the maximum number of steps allowed.

The main objective of this function is to minimize the cost function through assigning random numbers to each location and see if it is possible to reduce the utility function.