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 ways 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. The initial values of column and row are zero.
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 we are at the end of the row, we go to the next beginning of the next row}

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 assign 0 matrix[row, column] and update the column index by subtracting 1. If we are at the beginning of the row, we go to the end of last row. 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 returns an array called **choices**:

[[[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, matrix[0, 1] could only be either 1 or 2; and matrix[0, 2] could only be 1, 2, or 3.

Here are constraints we use to get the list:

For +:

First, if the number is greater or equal to the total amount, it shouldn't be considered; for example, in 5+, 6 should not be considered.

Second, if the total-value/(length of the letter size -1) is greater than the maximum value possible, it can not be a solution either.

For * :

First, if a set of prime factors of a number is not a subset of the set of prime factors of the total, a number should not be considered as a choice. For example 3 could never be in tiles with 100* since the set of prime factors of 100 is {2,5}.

Second, if the total/value is greater than the maximum value possible**(length of the letter size -1), it can not be a solution.

For - :
A solution always satisfies that total+the solution < dimention of the Kenken +1

For / :
A solution always satisfies that total+the solution < dimention of the Kenken +1

It also returns a dictionary called **to_start** with locations of each tile of Kenken sorted by the level of constraints. The more constrained a tile is, the less elements are in its choice list above. **The first element of to_start gives us the column and row index of the most constrained tile, and the second element gives us the column and row index of the second most constrained tile, and so on.**

complex_back(letter_array,matrix,letter_dict):

p.s. The solution matrix is initially filled with zeros.

Index = 0
row= row index stored in to_start[index]
column= column index stored in tin to_start[index]

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 look at the choice_array; then we find the list consisting choices correspondent to the given tile, matrix[row, column], and assign the first element of the choice list to matrix[row, column]
}
	if(matrix[row, column] now meets row, letter, column constraints) {
we move to the next element of the matrix:

index ++
row= row index stored in to_start[index]
column= column index stored in tin to_start[index]
}
	else if(matrix[row, column] doesn't meet row, letter, column constraints)
{
we find the next element in the list consisting choices correspondent to the given tile, matrix[row, column]. Then we assign this element to matrix[row, column] and go back to the beginning of the while loop.
As long as there are still elements on the choice list for a given matrix[row, column], we keep updating it until it meets row, letter, column constraints. If we are running out of choices, it means that we hit a dead end and have to backtrack:

index--;
row= row index stored in to_start[index]
column= column index stored in tin to_start[index]

Then we go back to the beginning of the while loop. If we go all the way back to the tile that we begin with and also run out of choices in the choice list for matrix[0][0], we return NO SOLUTION.
}

}

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.