Team group

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

Ahmet Hatip, Zihong Chen, and Ignacio Piera Fernández de Retana together developed the simple backtracking searching, complex backtracking search solution, and the local search. 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

First, row-checker(row, matrix):

this function checks if there are repeated numbers in rows;

Second, column-checker(column,matrix):

this function checks if there are repeated numbers in columns;

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

this function checks when a placed number works with parameters of Kenken puzzles; This function is achieved by a matrix storing the letters and a dictionary. We store parameters like 5+, 7, or 2* into dictionaries so we could recall them when iterating through the letter matrix.

Fourth, check_no_zero(location,matrix):

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

Fifth check_no_zero(location,matrix):

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

Sixth, simple_back(letter_array,matrix,letter_dict):

While (True)
{
iteration ++
if check_all, which checks if the whole matrix satisfies letter, column, and row constraints, return true, we return the matrix since we find the solution to Kenken.
if it doesn't return 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] is equal to 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.
}else{ we add 1 to matrix[row, column]}
}

}

Complex back-tracking solutions

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

get_choices_letters(solution,letter_dict):
A function that gives possible solutions of

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
RNumber of rows that have some equal numbers
CNumber of columns that have some equal numbers
LNumber 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.