Game of 24

A. Croitoru-Rusan & A. R. Smaranda
a.croitoru-rusan@student.rug.nl & a.r.smaranda@student.rug.nl

November 4, 2021

1 Problem description

A program is required to find a solution to the problem called Game of 24. The problem consists of finding the number 24 from 4 numbers between 1-10, only by using the $+$, $-$, $*$, and the $/$ operations. You may also use additional () to change the order of operations.

2 Problem analysis

The problem seems easy at first but as you try to think of a solution it gets harder and harder. The first thing we can observe is that this problem can easily be solved by generating permutations in two steps as it follows:

- Generating all permutations of the 4 numbers, this will result in 4! permutations.
- For each permutation of numbers we generate another permutation, but this time including 3 signs.
- For each of that we also need to add every combination of parenthesis.

This results in a huge amount of data and because of this, the problem becomes over complicated. Besides that, it's difficult to add operators and parenthesis in C, because we don't have a direct way to translate a string into a mathematical operation, and solving that goes beyond the scope of what this problem requires us to do.

3 Design

First, we start the program by reading the user's input. The data will be stored in an array called `numbers[4]`. We only accept numbers between **1-10**. To do that in a simple way, we store the value we read in a new variable called `input`. Then we check if the variable is between 1-10, if it is we go on and store it in the array, we also give the user the option to input **-1** to stop the program and start again.

To make the program more readable we create a `boolean` data type. We do that instead of using `boolean.h` because our program is not big enough to import a whole new library just for that. We implement that by creating a `typedef int bool` and an `enum` with values `true, false`.

To solve the problem we implement 2 functions, one that only calculates an operation between 2 numbers and one which computes the problem.

The first function is called `float calculate(int number1, int number2, char operator)`. It returns the calculation of an operation between `number1` and `number2`, operation given by the parameter `operator` which can be either $+$, $-$, $*$ or $/$.

The second function is called `bool computeGameOf24(int *numbers, int numbersSize)`. It returns either `true` if a solution was found or `false` if a solution was not found. We need to know if a solution was found to let the user know if there is a way to write the 4 numbers to get to the number 24.

In the function we make use of 2 important arrays.

- `char operators[4] = {'+', '-', '*', '/'}`
- `int signs[3]` - Store either 0 for +, - or 1 for *, /

The function starts by implementing 3 `for` loops, each for a permutation of the numbers array. After that it does another 3 `for` loops, but for the operators now.

We go on and check for each case if the value is 24. We only have a few cases to check for:

- `((perm1 OP perm2) OP perm3) OP perm4`
- `(perm1 OP perm2) OP (perm3 OP perm4)`
- `(perm1 OP (perm2 OP perm3)) OP perm4`
- `perm1 OP ((perm2 OP perm3) OP perm4 )`
- `perm1 OP (perm2 OP (perm3 OP perm4))`
- `perm1 OP perm2 OP perm3 OP perm4`

Where `OP` means operator and `permn` means the n number in a permutation of `numbers`.

If any of those values equals 24, we return `true` and also `print the values` inside the function.

By `default` the function will `return false` when it reaches the end.

4 Program code

Start of main.c

main.c

```c
#include <stdio.h>

#include "boolean.h"
#include "helpers.h"

// Function declarations
float calculate(int number1, int number2, char operator);
bool computeGameOf24(int *numbers, int numbersSize);

int main() {
    int numbers[4];
    for (int idx = 0; idx < 4; idx++) {
        int scan;

        scanf("%d", &scan);

        // we only accept inpus between 1 and 10
        while (scan != -1) {
            if (scan > 0 && scan < 11) {
                break;
            }

            printf("Please input a number between 1-10. \n");
            printf("Input -1 to stop \n");

            scanf("%d", &scan);
        }

        if (scan == -1) {
            return 0;
        }

        numbers[idx] = scan;
    }

    // if no way to reach 24 was found we let the user know that
    if (!computeGameOf24(numbers, 4)) {
        printf("No solution exists for numbers ");
        printArray(numbers, 4);
        printf(".");
    }

    printf("\n");
    return 0;
}
```

Calculate function

main.c

```c
/**
 * Function to calculate an operation between 2 number
 * @param number1
 * @param number2
 * @param operator Operator to use for the calculation, one of the following: +,-,*,/
 * @return The value it calculated
 */
float calculate(int number1, int number2, char operator) {
    switch (operator) {
        case '+':
            return number1 + number2;

        case '-':
            return number1 - number2;

        case '*':
            return number1 * number2;

        case '/':
            // we can't divide by 0
            if (number2 == 0) {
                return -1000;
            }

            // we do this to make sure we don't false trigger values like 24.5 as true
            if ((float) number1 / number2 != number1 / number2) {
                return -1000;
            }

            return (float) number1 / number2;

        default:
            return 0;
    }
}
```

Compute Game of 24 Function

main.c

```
bool computeGameOf24(int *numbers, int numbersSize) {
  int operatorsSize = 4;
  char operators[4] = {'+', '-', '*', '/'};

  float cal[3];
  int signs[3];// store either 0 for +, - or 1 for *, /

  for (int permutation1 = 0; permutation1 < numbersSize; permutation1++) {

    // start permutation 2
    for (int permutation2 = 0; permutation2 < numbersSize; permutation2++) {
      // skip if we try to use same number
      if (permutation2 == permutation1) {
        continue;
      }

      // start permutation 3
      for (int permutation3 = 0; permutation3 < numbersSize; permutation3++) {
        // skip if we use same number
        if (permutation3 == permutation1 || permutation3 == permutation2) {
          continue;
        }

        // 3 + 2 + 1 = 6 biggest value
        int permutation4 = 6 - (permutation1 + permutation2 + permutation3);

        // choose operators location
        for (int operator1 = 0; operator1 < operatorsSize; operator1++) {
          signs[0] = 0;
          if (operator1 <= 1) {
            signs[0] = 1;
          }

          for (int operator2 = 0; operator2 < operatorsSize; operator2++) {
            signs[1] = 0;
            if (operator2 <= 1) {
              signs[1] = 1;
            }

            for (int operator3 = 0; operator3 < operatorsSize; operator3++) {
              signs[2] = 0;
              if (operator3 <= 1) {
                signs[2] = 1;
              }

              CODE IN NEXT BLOCK
            }
          }
        }
      }
    }
  }

  return false;
}
```

main.c

```
if ((signs[0] ^ signs[1]) || (signs[0] ^ signs[2]) || (signs[1] ^ signs[2])) {

// ( (permutation1 + permutation2) * permutation3 ) / permutation4
cal[0] = calculate(numbers[permutation1], numbers[permutation2], operators[operator1]);
cal[1] = calculate(cal[0], numbers[permutation3], operators[operator2]);
cal[2] = calculate(cal[1], numbers[permutation4], operators[operator3]);
if (cal[2] == 24) {
    PRINT OPERATION
    return true;
}

// ( permutation1 + permutation2) * (permutation3 - permutation4)
cal[0] = calculate(numbers[permutation1], numbers[permutation2], operators[operator1]);
cal[2] = calculate(numbers[permutation3], numbers[permutation4], operators[operator3]);
cal[1] = calculate(cal[0], cal[2], operators[operator2]);
if (cal[1] == 24) {
    PRINT OPERATION
    return true;
}

// ( permutation1 * (permutation2 + permutation3) ) / permutation4
cal[1] = calculate(numbers[permutation2], numbers[permutation3], operators[operator2]);
cal[0] = calculate(numbers[permutation1], cal[1], operators[operator1]);
cal[2] = calculate(cal[0], numbers[permutation4], operators[operator3]);
if (cal[2] == 24) {
    PRINT OPERTAION
    return true;
}

// permutation1 / ( (permutation2 + permutation3) * permutation4 )
cal[1] = calculate(numbers[permutation2], numbers[permutation3], operators[operator2]);
cal[2] = calculate(cal[1], numbers[permutation4], operators[operator3]);
cal[0] = calculate(numbers[permutation1], cal[2], operators[operator1]);
if (cal[0] == 24) {
    PRINT OPERATION
    return true;
}

// permutation1 / ( permutation2 * (permutation3 + permutation4) )
cal[2] = calculate(numbers[permutation3], numbers[permutation4], operators[operator3]);
cal[1] = calculate(numbers[permutation2], cal[2], operators[operator2]);
cal[0] = calculate(numbers[permutation1], cal[1], operators[operator1]);
if (cal[0] == 24) {
    PRINT OPERATION
    return true;
}

} else {
    cal[0] = calculate(numbers[permutation1], numbers[permutation2], operators[operator1])
    cal[1] = calculate(cal[0], numbers[permutation3], operators[operator2]);
    cal[2] = calculate(cal[1], numbers[permutation4], operators[operator3]);
    if (cal[2] == 24) {
        PRINT OPERATION
        return true;
    }
}
```

Where, **PRINT OPERATION** is (parenthesis are the only change for each case):

PRINT OPERATION

```
printf("%d %c %d %c %d %c %d", numbers[permutation1], operators[operator1], numbers[
    permutation2], operators[operator2], numbers[permutation3], operators[operator3],
    numbers[permutation4]);
```

Other Files

boolean.h

```
#ifndef GAME24_BOOLEAN_H
#define GAME24_BOOLEAN_H

typedef int bool;
enum {
    false,
    true
};

#endif//GAME24_BOOLEAN_H
```

helpers.h

```
#ifndef GAME24_HELPERS_H
#define GAME24_HELPERS_H

void printArray(int *array, int size);

#endif //GAME24_HELPERS_H
```

helpers.c

```
#include "helpers.h"

#include <printf.h>

/**
 * Print array on screen
 * @param array
 * @param size
 */
void printArray(int *array, int size) {
    for (int idx = 0; idx < size; idx++) {
        printf("%d", array[idx]);

        if (idx != (size - 1)) {
            printf(" ");
        }
    }
}
```

5 Test results

- Input: (Should work case)

```
1 1 8 8
```

Output:

```
((1 + 1) * 8) + 8
```

- Input: (Should work case)

```
1 2 7 7
```

Output:

```
((7 * 7) - 1) / 2
```

- Input: (Should work case)

```
2 2 3 9
```

Output:

```
(2 + 2) * (9 - 3)
```

- Input: (Wrong input case)

```
11
```

Output:

```
Please input a number between 1-10.
Input -1 to stop
```

- Input: (Should fail case)

```
8 8 8 8
```

Output:

```
No solution exists for numbers 8 8 8 8.
```

- Input: (Should fail case)

```
6 7 7 8
```

Output:

```
No solution exists for numbers 6 7 7 8.
```

6 Evaluation

The problem seems easy to solve at first but it becomes more complicated, the deeper you try to implement it. It was a challenge to keep the code simple and easy to understand for another person.

Reaching this solving method took a long time. The first idea was to make use of a sort-of Depth First Search algorithm, but after some more thinking, we realised that we cannot display the values we used to get to 24 so it's basically useless.

The second approach was generating all possible combinations, but as we described in the design part, changing strings to actual math operations in C is very complicated, and we want to avoid over complication with this problem.

The last and final approach was the one described in this report, writing more explicit ifs but keeping the code simple.