Game of 24

XueJin Cui (S5876095) & Jiayi Yang (S6456774)

October 2025

1 Problem description

A program is required that determines whether 4 input numbers can form a value of 24, using only addition, subtraction, division and multiplication. If can, determine the process to get the number 24. The 4 input numbers are integers between 1 and 10(inclusive)

2 Problem analysis

We interpret the problem as follows:

Given 4 numbers from 1 to 10(inclusive), determine whether there exists a sequence of mathematical calculation (addition, subtraction, multiplication, division only) and eventually form the number of 24. This interpretation is motivated by the following statements:

- Recursion decomposition: Combines 2 numbers by one single mathematical calculation to form a new number in order to reduce the scale of the problem. In the next recursion, these two numbers are deleted from the set, and the new number is added to the set. The process(recursion) repeats until only 1 number remains.
- If only 1 number exists, check if it equals to 24. If it is equal to 24, store the corresponding expression and terminate the further operation. Else, go for other new combinations.

The termination condition is follow:

```
if (there is only 1 remaining number ) {
    if (the remaining number is 24)) ...
}
```

Among the process, we need to consider about the sequence as well. For multiplication and addition, we only need to consider 1 sequence : x+y=y+x ; x*y=y*x

However, in terms of division and subtraction, there are 2 sequences as x-y not equal to y-x; and x/y not equal to y/x;

The advantage of the algorithm is : is able to track the complete corresponding expression and once obtain the value of 24, return the expression directly and skips the remaining steps.

3 Program code

24gamesolver.c

```c
/* Xuejin Cui & Jiayi Yang written in october 2025*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void input(double numbers[],char expr[][100]) {

   // Input function to read 4 numbers
   double a;
   printf("Input 4 numbers between 1 and 10 (inclusive):\n");
   for (int i=0;i<4;i++) {
      scanf ("%lf",&a);
      // check validity
      if ((a>=1) && (a<=10)) {
         numbers[i] = a;
      } else {
         printf("Invalid input.\n");
         i--;
      }
   }

   // Initialize expressions as strings of the numbers
   for (int j=0;j<4;j++) {
      snprintf(expr[j],100,"%.0f",numbers[j]);
   }
}

int isEqual(double a,double b) {
   // Check if two doubles are equal
   return (fabs(a-b)<1e-9);
}

int isPossible(
   int length, double numbers[4],
   char expressions[4][100], char result[100]) {

      double newNumbers[4]; // Temporary array for new numbers
      char newExpressions[4][100]; // Temporary array for new expressions

      // Base case: if only one number left, check if it equals 24
      if (length == 1) {
         if (isEqual(numbers[0],24)) {
            snprintf(result,100,"%s",expressions[0]); //set result
            return 1;
         } else {
            return 0;
         }
      }
```

```
48
49      // Use double loops to consider all pairs of numbers
50      for (int i=0;i<length;i++) {
51         for (int j=i+1;j<length;j++) {
52
53            int index = 0;
54            // Fill newNumbers and newExpressions with remaining numbers
55            for (int k=0;k<length;k++) {
56               if ((k!=i)&&(k!=j)) {
57                  newNumbers[index] = numbers[k];
58                  snprintf(newExpressions[index],100,"%s",expressions[k]);
59                  index++;
60               }
61            }
62
63            //Then try all operations between numbers[i] and numbers[j]
64
65            //For addition and multiplication, we only need to do one order
66            //addition:
67            newNumbers[index] = numbers[i] + numbers[j]; // store sum
68            snprintf (
69               newExpressions[index],100,
70               "(%s + %s)",expressions[i],expressions[j]);
71               // store expression
72
73            //Check if this leads to a solution, if so return 1 and skip rest
74            if (isPossible(length-1,newNumbers,newExpressions,result)) {
75               return 1;
76            }
77
78            // The other operation following the same pattern:
79            //multiplication:
80            newNumbers[index] = numbers[i] * numbers[j];
81            snprintf (
82               newExpressions[index],100,
83               "(%s * %s)",expressions[i],expressions[j]);
84
85            if (isPossible(length-1,newNumbers,newExpressions,result)) {
86               return 1;
87            }
88
89            //For subtraction and division, we need to consider both orders
90            //subtraction (order i - j):
91            newNumbers[index] = numbers[i] - numbers[j];
92            snprintf (
93               newExpressions[index],100,
94               "(%s - %s)",expressions[i],expressions[j]);
95
96            if (isPossible(length-1,newNumbers,newExpressions,result)) {
97               return 1;
98            }
```

```
99
100            //subtraction (order j - i):
101            newNumbers[index] = numbers[j] - numbers[i];
102            snprintf (
103               newExpressions[index],100,
104               "(%s - %s)",expressions[j],expressions[i]);
105
106            if (isPossible(length-1,newNumbers,newExpressions,result)) {
107               return 1;
108            }
109
110            //For division, we also need to ensure the denominator is not zero
111            //division (order j / i):
112            if (!isEqual(numbers[i],0)) {
113               newNumbers[index] = numbers[j] / numbers[i];
114               snprintf (
115                  newExpressions[index],100,
116                  "(%s / %s)",expressions[j],expressions[i]);
117
118               if (isPossible(length-1,newNumbers,newExpressions,result)) {
119                  return 1;
120               }
121            }
122
123            //division (order i / j):
124            if (!isEqual(numbers[j],0)) {
125               newNumbers[index] = numbers[i] / numbers[j];
126               snprintf (
127                  newExpressions[index],100,
128                  "(%s / %s)",expressions[i],expressions[j]);
129
130               if (isPossible(length-1,newNumbers,newExpressions,result)) {
131                  return 1;
132               }
133            }
134         }
135      }
136
137      return 0;
138 }
139
140 int main(int argc, char *argv[]) {
141    double numbers[4];
142    char expressions[4][100];
143    char result[100];
144    input(numbers,expressions);
145    if (isPossible(4,numbers,expressions,result)) {
146       printf("Solution found: %s = 24\n",result);
147    } else {
148       printf ("No solution found.\n");
149    }
```

```
150     return 0;
151 }
```

4 Test results

4.1 Test Cases for 24-Point Game

Input: (normal simple case)
8 9 3 4
Output:
Solution found: ((8 + 9) + (3 + 4)) = 24

Input: (case that cannot form the value of 24)
1 1 1 1
Output:
No solution founded.

Input: (the case that input contains the boundary)
1 10 5 6
Output:
Solution found: (6 * ((10 - 1) - 5)) = 24

Input: (the case that input is out of boundary)
11 8 9 1
Output:
Invalid input.

Input: (the complex case that contains decimal)
3 8 3 8
Output:
Solution found: (8 / (3 - (8 / 3))) = 24

5 Evaluation

As the algorithm uses a recursive structure, the design and implementation of the program is relatively straightforward. The definition of the problem is using the four given numbers to obtain a value of 24 by using the basic mathematical calculation. The core idea of our program is to reduce the number of elements in the set step by step until the number 24 is achieved. This concept is simple and clear. Under the current rules

of 24 game, the program is able to print out the correct result. It considers all possible combinations of 4 numbers. For subtraction and division, specifically, will test 2 possible sequences. In our originally program, we used 'float' which may produce tiny errors after multiple calculations such as 23.999999 or 24.000001 and we replaced it with 'double' in the current program. The precision increases from 7 significant digits to 15–16 significant digits. Floating-point errors are smaller, making calculations more stable. And by using isEqual(a, b) function with a tolerance of 1e-9, this precision problem can be resolved. Alternatively, approaches of using iteration with all permutations and dynamic programming would be possible as well. However, iteration approach has a extremely high complexity and need to manage the combinations of calculations manually, While the dynamic programming approach is complicate to implement and need to design data structure. In conclusion, We believe that the recursion approach is much more efficient and easy to achieve, and guarantees correct results for all valid inputs.