24 Game

You have 4 cards each containing a number from 1 to 9. You need to judge whether they could operated through \*, /, +, -, (, ) to get the value of 24.

Example 1:

Input: [4, 1, 8, 7]

Output: **True**, because (8-4) \* (7-1) = 24

Example 2:

Input: [1, 2, 1, 2]

Output: **False** because it is impossible to get 24 from digits above.

First, we need to mention that we do not need to consider parentheses, it just a to calculate two elements first. This problem just like you pick two elements from these four numbers and then you do some calculation to make three numbers from four. Then, you need to pick next two items do some arithmetic calculation to make two numbers from three and so far. In the last we will only have one element and we need to check is it 24 or not.

Let us see how this algorithm should work for Example 1:

[4, 1, 8, 7]

1. First of all, we need to turn our vector of ints into vector of doubles because we will use ‘/’ operator.
2. Then by using helper function we will create one more vector for all calculation (\*, /, +, -, (, ) ) for any two digits.
3. For example, for first two digits (4.0 , 1.0) we will get [5.0, 3.0, -3.0, 15.0, 4.0, 0.0]
4. Then we change fourth and second element of original array [4.0, 1.0, 8.0, 7.0] and get [4.0, 7.0, 8.0, 7.0] because in one of the next steps we will start substituting first element with all results of calculations from [5.0, 3.0, -3.0, 15.0, 4.0, 0.0] and decreasing size of original array by one.
5. Substituting results of calculations with first index and using recursion by decreasing size of original array by one.
6. If solved return true, if not put initial values for first and second index.
7. Do the same for all possible combinations by using two for loops.

Thus, we have only 4 cards and 4 operations, let’s see how many possibilities we have to express those. We choose two numbers in 12 ways and perform one of 4 operations (12\*4), then 3 remaining and we do the same thing, but we already have 6 ways – (6 \* 4), then 2 remaining and 2 ways to express – (2 \* 4).

So, 12 \* 6 \* 2 \* 4 \* 4 \* 4 = 9216 ways to calculate 4 cards with 4 operation by using parentheses.

For solving this problem, I used two helper functions.

First helper function I would explain is getAllExpressions(double a, double b) which let us get all possible results by using two cards as an arguments:

vector<double> getAllExpressions(

    double a, double b) // method to get all possible solution by using two cards

{

  return {a + b, a - b, b - a, a \* b, a / b, b / a};

}

Second helper function is solve(vector<double> &numbers, int s) which takes a vector of doubles and try to solve the problem by substituting all possible cards and expressions.:

bool solve(vector<double> &numbers,

           int s) // helper function with vector of doubles and size of vector

{

  if (s == 1)                              // if one item left

    return abs(numbers[0] - 24) < checker; // return true or false, checker is 0.0001

  for (int i = 0; i < s; i++)              // from 0 to size of Array

  {

    for (int j = i + 1; j < s; j++) // from 1 to size of Array

    {

      double index1 = numbers[i];  // taking 1 element

      double index2 = numbers[j];  // taking 2 element

      numbers[j] = numbers[s - 1]; // second number becomes last one

      for (double x : getAllExpressions(index1, index2)) // for 6 results

      {

        numbers[i] = x;            // we put each result on first place

        if (solve(numbers, s - 1)) // and try solve it again

          return true;

      }

      numbers[i] = index1; // if not solved we put index1 to its place

      numbers[j] = index2; // if not solved we put index2 to its place

    }

  }

  return false;

}

And judgePoint24(vector<int>& nums) which uses solve function and get us the right output:

bool judgePoint24(vector<int> &nums) {

  vector<double> numsDouble(nums.begin(), nums.end()); // changing int to double

  int sizeofArray = numsDouble.size();

  return solve(numsDouble, sizeofArray);

}

When my code worked, I submitted it couple times and end up with this time and space complexity:

Graphical user interface, text

Description automatically generated

I started discovering other solutions and all of them use the same strategy for solving particular problem, but I decided to mark pieces of the code where we can economy some space and time.

Best time complexity was 0 ms and it used this solution, it used only this helper function and got calculations manually without second helper, what let it use only two for loops for solving this problem:

void dfs(vector<double> &v, int n) {

  if (n == 1) {

    if (abs(v[0] - 24) <= 0.01) {

      flag = true;

    }

    return;

  }

  for (int i = 0; i < n; ++i) {

    for (int j = i + 1; j < n; ++j) {

      double a = v[i];

      double b = v[j];

      v[i] = a + b;

      v[j] = v[n - 1];

      dfs(v, n - 1);

      v[i] = a - b;

      v[j] = v[n - 1];

      dfs(v, n - 1);

      v[i] = b - a;

      v[j] = v[n - 1];

      dfs(v, n - 1);

      v[i] = a \* b;

      v[j] = v[n - 1];

      dfs(v, n - 1);

      if (b != 0) {

        v[i] = a / b;

        v[j] = v[n - 1];

        dfs(v, n - 1);

      }

      if (a != 0) {

        v[i] = b / a;

        v[j] = v[n - 1];

        dfs(v, n - 1);

      }

      v[i] = a;

      v[j] = b;

    }

  }

}

Best space complexity was 7.2 MB or 7200 KB and main difference between mine and this solution was that we could avoid creating a new vector for double type of variable and also we can avoid using getAllExpression() method and do everything manually in solve() function.

A picture containing text

Description automatically generated

I hope my explanation was clear to you and now you can come up with your own solution which will show the most efficient result.