Задача E. Взвешивания

Условие задачи:

Даны двухчашечные весы и набор гирек. На левую чашу весов положили взвешиваемый предмет весом *K* граммов. Можно ли привести весы в состояние равновесия, и если можно, то определите для каждой чаши весов, какие гирьки на нее для этого нужно положить. Имеющиеся гирьки разрешается класть на любую из чаш весов (каждая гирька имеется только в одном экземпляре, некоторые гирьки можно не использовать).

Входные данные

Вводится сначала *K* — вес предмета, который положили на левую чашу ($1 \le K \le 50$). Далее записано общее количество гирек *N* ($1 \le N \le 10$). Далее записано *N* различных натуральных чисел, не превышающих 50, — веса гирек.

Выходные данные

В первой строке выведите веса гирек, которые нужно поместить на левую чашу весов, во второй строке — гирьки, которые нужно поместить на правую чашу. Если на какую-то чашу ни одной гирьки помещать не нужно — выведите в этой строке число 0. Если с помощью данных гирек привести весы в равновесие нельзя, выведите одно число –1. Если вариантов несколько, выведите любой из них.

Примеры

Входные данные	Выходные данные
5 2 3 5	0 5
5 3 6 3 4	4 3 6
5 1 2	-1

Разбор решения:

Поскольку гирек, по условию задачи, немного, задачу можно решать прямым перебором. Каждая из гирек может находиться в одном из трёх состояний: на левой чаше весов, на правой чаше весов и вообще не на весах. Вычислительная сложность решения составит 3^N, где N – это количество гирек. Для самого сложного случая, когда N = 10, это даст 59049 вариантов. Очевидно, что это довольно немного, и на современных компьютерах решение, основанное на прямом переборе, будет выполняться достаточно быстро.

Ниже приводится решение задачи на языке программирования C++.

```
typedef enum
{
    No,
    Left,
    Right
} State;

bool run(int difference, // Разность весов на разных чашах
         int* weights,   // Веса гирек
         State* states   // Статусы гирек
         int n,          // Количество гирек
         int index)      // Номер гирьки
{
    if (index == n)      // Если перебрали все гирьки
        return difference == 0;  // Уравновешены ли весы
```

```
    // Не кладём index-ую гирьку на весы
    states[index] = No;
    if (run(difference, weights, states, n, index + 1))
        return true;

    // Кладём index-ую гирьку на левую чашу
    states[index] = Left;
    if (run(difference + weights[index],
        weights, states, n, index + 1))
        return true;

    // Кладём index-ую гирьку на правую чашу
    states[index] = Right;
    if (run(difference - weights[index],
        weights, states, n, index + 1))
        return true;

    return false;
}
```

Для получения решения эту функцию нужно вызвать, в качестве первого параметра передав ей вес груза, второй параметр должен быть указателем на веса гирек, третий – на статусы гирек, четвёртый – количество гирек, совпадающее с количеством их статусов, пятым параметром нужно передать значение 0. Например:

```
int k, n;
cin >> k;
cin >> n;

int weights = new int[n];
State states = new State[n];

for (int i = 0; i < n; i++)
    cin >> weights[i];

if (run(k, weights, states, n, 0))
{
    // Выводим результат
}
```

Если функция вернула true, то в массиве статусов гирек останется информация о том, где какая гирька лежала в момент, когда весы были уравновешены. Функция сделает столько рекурсивных вызовов, сколько у нас гирек, и на каждом уровне рекурсии содержит три варианта, куда положить гирьку. Как только весы уравновесятся, произойдёт выход из всех рекурсивных вызовов, и гирьки при этом перекладываться не будут. Если ни один из вариантов не позволил уравновесить весы, то функция вернёт значение false.

Задача D. Маршрут

Условие задачи:

Дана матрица NxN, заполненная положительными числами. Путь по матрице начинается в левом верхнем углу. За один ход можно пройти в соседнюю по вертикали или горизонтали клетку (если она существует). Нельзя ходить по диагонали, нельзя оставаться на месте. Требуется найти

максимальную сумму чисел, стоящих в клетках по пути длиной K (клетку можно посещать несколько раз).

Входные данные

В первой строке находятся разделенные пробелом числа N и K. Затем идут N строк по N чисел в каждой. 2 <= N <= 100, элементы матрицы имеют значения от 1 до 9999, 1 <= K <= 2000, все числа целые.

Выходные данные

Вывести одно число - максимальную сумму.

Примеры

Входные данные	Выходные данные
5 7 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 100	7

Разбор решения:

Наиболее просто решается эта задача методом прямого перебора: можно перебрать все возможные пути прохождения матрицы, посчитать для них сумму и выбрать наилучшую. От верхней левой клетки можно перемещаться либо вниз, либо вправо. От клетки в верхнем ряду можно перемещаться вниз, вправо и влево, от клетки в левом ряду можно перемещаться вверх, вниз и вправо, и так далее. Таким образом, можно сказать, что из клетки, находящейся в углу, можно сделать 2 возможных хода, из клетки, находящейся на краю, но не в углу, можно сделать 3 возможных хода, а из клетки, находящейся где-то в середине матрицы, можно сделать 4 возможных хода. Очевидно, что матрица имеет 4 угла, количество краевых клеток равно (N – 2) * 4, а количество центральных клеток составляет N^2 - (N – 1) * 4. Это можно наглядно показать, расположив краевые ячейки матрицы по диагонали – тогда каждый из четырёх краёв матрицы будет иметь длину N – 1. Схематически это можно изобразить с использованием таблицы 1.

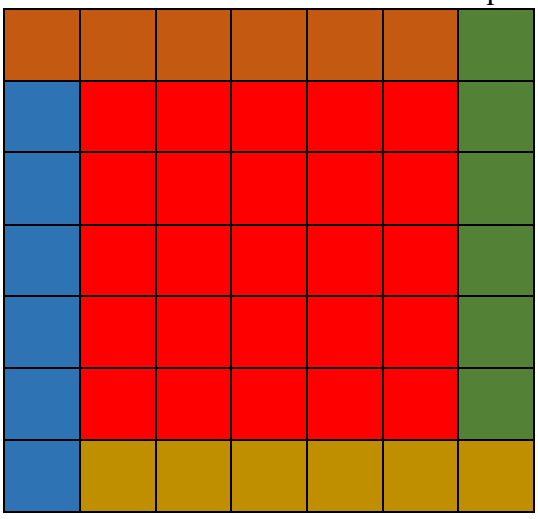

Таблица 1. Схема разделения матрицы размером 7*7.

Как известно из теории алгоритмов, при оценке вычислительной сложности в многочлене учитывается тот член, который быстрее всего растёт с увеличением объёма входных данных. Поскольку квадрат растёт быстрее, чем линейная функция, последней можно пренебречь. При максимальном по условию задачи N = 100, количество центральных клеток будет равно 9604, а количество краевых клеток 392. Таким образом, допустимо считать, что, если на i-м шаге мы находимся в определённой клетке, у нас есть 4 возможных варианта i+1-го шага. Таким образом, если рассматривать все возможные пути прохождения матрицы, мы получим 4^K шагов: с каждым следующим шагом количество возможных вариантов увеличивается в 4 раза. Поскольку

максимальное значение K = 2000, общее количество шагов будет слишком большим, чтобы алгоритм смог отработать за приемлемое время.

Тем не менее, при небольших K этот алгоритм может послужить инструментов отладки. Для его реализации необходима рекурсивная функция, которая будет прекращать рекурсивные вызовы по достижению K-го шага. В данной статье фрагменты кода приводятся на языке C++.

```
int last_weight = 0;

void run(int** matrix,      // Матрица, которую обходим
    int n,                  // Размер матрицы
    int k,                  // Сколько шагов осталось
    int x, int y,           // Текущие координаты
    int weight)             // Набираемый вес пути
{
    if (k == 0)  // Дошли до конца маршрута
    {
        if (weight + matrix[x][y] > last_weight)
            last_weight = weight + matrix[x][y];
    }

    if (x > 0)
    {
        run(matrix, n, k - 1, x - 1, y,
                weight + matrix[x][y]);
    }
    if (x < n - 1)
    {
        run(matrix, n, k - 1, x + 1, y,
                weight + matrix[x][y]);
    }
    if (y > 0)
    {
        run(matrix, n, k - 1, x, y - 1,
                weight + matrix[x][y]);
    }
    if (y < n - 1)
    {
        run(matrix, n, k - 1, x, y + 1,
            weight + matrix[x][y]);
    }
}
```

С помощью данного, достаточно несложного для отладки кода, можно получить наборы тестовых данных, с которыми можно сверять результаты более продвинутого алгоритма. Однако недостатком данного алгоритма является не только высокая вычислительная сложность, но и то, что при больших значениях K он может переполнить стек, поскольку глубина рекурсивных вызовов зависит от K линейно.

Для оптимизации алгоритма поиска следует задуматься о том, что количество маршрутов, которыми можно обойти множество смежных ячеек матрицы, является заведомо избыточным. Например, начиная с верхнего левого края, матрицу размером 2 * 2 можно полностью обойти двумя способами: 1,1; 1,2; 2,2; 2,1 и 1,1; 2,1; 2,2; 1,2. Оба этих маршрута посещают одни и те же

ячейки матрицы, но в разном порядке. Поскольку сложение коммутативно, очевидно, что сумма всех значений ячеек в обоих случаях получится одна и та же. Для решения задачи важно то, какие ячейки посещаются, а не их порядок. Поэтому следует не посещать маршруты, а сразу же оценивать возможный максимум для каждого шага. Для каждой ячейки этот максимум будет свой. Его можно вычислить через рекуррентную процедуру: максимум, достижимый в ячейке с координатами (x, y) будет равен сумме значения, хранящегося в соответствующей ячейке исходной матрицы, и самого большого из максимумов в соседних ячейках. Для сохранения максимумов, достижимых на предыдущем шаге, во время расчёта максимумов, достижимых на текущем шаге, потребуется две дополнительные матрицы размером N * N, в которых хранятся максимумы. Таким образом, алгоритм решения будет следующим.

Первой напишем функцию, которая формирует начальные матрицы:

```
int** getMatrix(int n)
{
    int ** ret = new int*[n];
    for (int i = 0; i < n; i++)
    {
        ret[i] = new int[n];
        for (int j = 0; j < n; j++)
            ret[i][j] = 0;
    }
    return ret;
}

int ** prev = getMatrix(n); // Два поля для хранения
int ** curr = getMatrix(n); // предыдущего и текущего шага
```

После этого нам потребуется переходить от одного шага к другому. Для этого необходимо заменять матрицу максимумов, полученную на предыдущем шаге, матрицей, сформированной на текущем шаге. При этом для следующего шага матрицу максимумов следует обнулить.

```
void step(int n)
{
    int ** tmp = prev;
    prev = curr;
    curr = tmp;

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            curr[i][j] = 0;
    }
}
```

Следующая функция, которая потребуется нам для решения задачи – это функция, которая ищет самый большой максимум рядом с интересующей нас ячейкой.

```
int findMax(int x, int y, int n)
```

```
{
    int ret = 0;

    if (x > 0 && prev[x - 1][y] > ret)
        ret = prev[x - 1][y];
    if (x < n - 1 && prev[x + 1][y] > ret)
        ret = prev[x + 1][y];
    if (y > 0 && prev[x][y - 1] > ret)
        ret = prev[x][y - 1];
    if (y < n - 1 && prev[x][y + 1] > ret)
        ret = prev[x][y + 1];

    return ret;
}
```

Следующая функция формирует матрицу максимумов для следующего шага по матрице для предыдущего шага.

```
void transform(int** matrix, int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            curr[i][j] = matrix[i][j] + findMax(x, y, n);
    }

    step(n);
}
```

Следующая функция показывает окончательное решение задачи.

```
int run(int** matrix, int n, int k)
{
    prev[0][0] = matrix[0][0];  // Максимум для 0 шагов

    for (int i = 0; i < k; i++) // Пересчитываем максимумы
        transform(matrix, n);   // k раз

    int ret = 0;

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            if (prev[i][j] > ret)
                ret = prev[i][j];
    }

    return ret;
}
```

Оценим вычислительную сложность приведённого решения. Отметим, что оно не содержит рекурсии, поэтому можно не опасаться переполнения стека. Сначала дважды вызывается функция getMatrix(), имеющая два вложенных цикла, которые выполняются N раз. В принципе, вместо

заполнения матрицы нулями можно использовать стандартную функцию memset, которая позволяет задействовать аппаратное ускорение при работе с большими блоками оперативной памяти. В любом случае, данный фрагмент кода даст либо N, либо N^2 операций.

После этого алгоритм N раз вызывает функцию transform, которая содержит, в свою очередь, два вложенных цикла, которые выполняются N раз, и один раз вызывает метод step, который так же содержит два вложенных цикла. Кроме того, в самом внутреннем цикле функция transform вызывает функцию findMax, но эта функция не содержит ни рекурсии, ни циклов, поэтому не влияет на вычислительную сложность. Таким образом, вычислительная сложность функции transform составляет $O(N^2)$. Поскольку она выполняется K раз, вычислительная сложность всего этого фрагмента составляет $O(K * N^2)$.

После этого происходит выполнение ещё одной пары вложенных циклов, которые дают N^2 операций. Поскольку при вычислительной сложности оценивается наиболее быстро растущий член многочлена, общая вычислительная сложность данного алгоритма по времени составляет $O(K * N^2)$. В самом тяжёлом случае, при N = 100 и K = 2000, алгоритм будет делать порядка 20 000 000 шагов, что на современных компьютерах будет выполняться менее секунды.

Данный алгоритм так же потребует выделения памяти для двух дополнительных матриц такого же размера, как исходная. Поскольку объём памяти, занимаемой матрицей, зависит от её размера квадратично, вычислительная сложность данного алгоритма по памяти составит $O(N^2)$.

Автор разборов Медведев Сергей Алексеевич, преподаватель дополнительного образования, предмет Алгоритмы и структуры данных.