Динамическое программирование

Д. Кириенко

Нахождение числа сочетаний

Рассмотрим простейшую комбинаторную задачу: сколькими способами можно из данных $n \geqslant 0$ предметов выбрать некоторые k предметов ($0 \leqslant k \leqslant n$), если порядок их выбора не важен? Ответом на эту задачу является величина $C_n^k = \frac{n!}{k!(n-k)!}$, называемая *числом сочетаний из n элементов по k*. Запись $n!$ обозначает произведение $1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$, называемое *факториалом* числа n, при этом считается, что $0! = 1$.

Эту формулу несложно вывести. Расставить n предметов в ряд можно $n!$ способами. Рассмотрим все $n!$ расстановок и будем считать, что первые k предметов в ряду мы выбрали, а оставшиеся $n-k$ предметов — нет. Тогда каждый способ выбрать k предметов мы посчитали $k!(n-k)!$ раз, поскольку k выбранных предметов можно переставить $k!$ способами, а $n-k$ невыбранных — $(n-k)!$ способами. Поделив общее количество перестановок на количество повторов каждой выборки, получим число выборок: $C_n^k = \frac{n!}{k!(n-k)!}$.

Наивный метод

Как же вычислить значение C_n^k по данным n и k? Воспользуемся выведенной формулой. Функция `factorial(n)` возвращает факториал числа `n`, а функция `C(n,k)` возвращает значение C_n^k.

```
int factorial(int n)
{
    int f=1;
    for(int i=2;i<=n;++i)
        f=f*i;
    return f;
}

int C(int n, int k)
{
    return factorial(n)/factorial(k)/factorial(n-k);
}
```

Данный алгоритм понятен, очень быстр (его вычислительная сложность $O(n)$, поскольку вычисление значения $n!$ требует $n-1$ умножение), использует ограниченный размер дополнительной памяти. Но у него есть существенный недостаток: он будет корректно работать только для небольших значений n и k. Дело в том, что величина $n!$ очень быстро растет с увеличением n, например, значение $13! = 6\,227\,020\,800$ превосходит максимальное возможное значение, которое можно записать в 32-разрядном целом числе, а величина

$$21! = 51\,090\,942\,171\,709\,440\,000$$

не помещается в 64-разрядном целом числе. Поэтому пользоваться данной функцией можно только для $n \leqslant 12$ при использовании 32-битной арифметики или для $n \leqslant 20$ при использовании 64-битной арифметики. При этом само значение C_n^k может быть невелико, но при проведении промежуточных вычислений факториалов может возникнуть переполнение. Например, $C_{30}^{15} = 155117520$ можно записать с использованием 32-битной арифметики, однако, значение $30!$ при этом не поместится и в 64-разрядном целом числе и вычислить значение C_{30}^{15} таким методом не удастся.

Рекурсивный метод

Проблем с переполнением можно избежать, если воспользоваться для вычисления известной формулой: $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$ (при $0 < k < n$).

Действительно, выбрать из n предметов k можно C_n^k способами. Пометим один из данных n предметов. Тогда все выборки можно разбить на две группы: в которые входит помеченный предмет (их будет C_{n-1}^{k-1}) и не содержащие помеченного предмета (таких выборок будет C_{n-1}^k).

Добавив к этой формуле краевые значения: $C_n^n = C_n^0 = 1$ (выбрать все предметы или не выбрать ни одного предмета можно единственным способом), можно написать рекурсивную функцию вычисления числа сочетаний:

```
int C(int n, int k)
{
    if (k==0 || k==n)
        return 1;
    else
        return C(n-1,k-1)+C(n-1,k);
}
```

При таком решении задачи проблем с переполнением не возникнет: если значение конечного ответа не приводит к переполнению, то поскольку при его нахождении мы суммируем меньшие натуральные чис-

ла, все промежуточные значения, возвращаемые функцией `C(n,k)` тоже не будут приводить к переполнению, и такая программа, например, сможет вычислить значение C_{30}^{15}.

Но это решение крайне плохо тем, что работать оно будет очень долго, поскольку функция `C(n,k)` будет вызываться многократно для одних и тех же значений параметров `n` и `k`. Например, если мы вызовем функцию `C(30,15)`, то она вызовет функции `C(29,14)` и `C(29,15)`. Функция `C(29,14)` вызовет функции `C(28,13)` и `C(28,14)`, а `C(29,15)` вызовет функции `C(28,14)` и `C(28,15)`. Мы видим, что функция `C(28,14)` будет вызвана дважды. С увеличением глубины рекурсии количество повторяющихся вызовов функции быстро растет: функция `C(27,13)` будет вызвана три раза (дважды ее вызовет функция `C(28,14)` и еще один раз ее вызовет `C(28,13)` и т. д.

При этом каждая функция `C(n,k)` может либо вернуть значение 1, либо вернуть сумму результатов двух других рекурсивных вызовов, а, значит, любое значение, которое вернула функция `C(n,k)` получается сложением в различных комбинациях чисел 1, которыми заканчиваются все рекурсивные вызовы. Значит, при вычислении C_n^k инструкция `return 1` в приведенной программе будет выполнена ровно C_n^k раз, то есть сложность такого рекурсивного алгоритма для вычисления C_n^k не меньше, чем само значение C_n^k.

Метод динамического программирования

Итак, одна из проблем рекурсивных алгоритмов (и эта проблема возникает весьма часто, не только в рассмотренном примере) — длительная работа рекурсии за счет повторяющихся вызовов рекурсивной функции для одного и того же набора параметров. Чтобы не тратить машинное время на вычисление значений рекурсивной функции, которые мы уже когда-то вычислили, можно сохранить эти значения в массиве и вместо рекурсивного вызова функции мы будем брать это значение из массива. В задаче вычисления числа сочетаний создадим двумерный массив `B` и будем в элементе массива `B[n][k]` хранить величину C_n^k. В результате получим следующий массив (по строчкам — значения n, начиная с 0, по столбцам — значения k, начиная с 0, каждый элемент массива `B[n][k]` равен сумме двух элементов: стоящего непосредственно над ним `B[n-1][k]` и стоящего над ним слева `B[n-1][k-1]`).

1						
1	1					
1	2	1				
1	3	3	1			
1	4	6	4	1		
1	5	10	10	5	1	
1	6	15	20	15	6	1

Полученная числовая таблица, составленная из значений C_n^k, в которой каждый элемент равен сумме двух элементов, стоящих над ним, называется *треугольником Паскаля*.

Поскольку каждый элемент этого массива равны сумме двух элементов, стоящих в предыдущей строке, то этот массив нужно заполнять по строкам. Соответствующая функция, заполняющая массив и вычисляющая необходимое число сочетаний может быть такой: [1]

```
int C(int n, int k)
{
    int B[n+1][n+1];      // Создаем массив B из n+1 строки
    for(int i=0;i<=n;++i) // Заполняем i-ю строку массива
    {
       B[i][0]=1;         // На концах строки стоят единицы
       B[i][i]=1;
       for(int j=1;j<i;++j)
       {   // Заполняем оставшиеся элементы i-й строки
           B[i][j]=B[i-1][j-1]+B[i-1][j];
       }
    }
    return B[n][k];
}
```

Приведенный алгоритм для вычисления C_n^k требует объем памяти $O(n^2)$ и такая же его временная сложность (для заполнения одного эле-

[1] В приведенном тексте программы используется объявление `int B[n+1][k+1]` для создания двумерного массива целых чисел размера $(n+1) \times (k+1)$. Такое объявление не соответствует стандартам языков C и C++, но оно допускается в компиляторе GNU C++, поэтому, ввиду простоты такой записи, мы будем везде использовать подобное обозначение для создания динамических массивов.

Вместо такой записи можно использовать массивы фиксированного размера, например, `int B[101][101]`, если величина n не превышает 100 в решамой задаче, либо использовать динамическое распределение памяти при помощи указателей и функций `malloc` и `free` (в языке C++ используются операторы `new` и `delete`). Также в языке C++ можно использовать библиотеку STL: `vector < vector <int> > B ( n+1, vector <int> (k+1) )`.

мента массива требуется $O(1)$ операций, всего же элементов в массиве $O(n^2)$).

Подобный метод решения, когда одна задача сводится к меньшим подзадачам, вычисленные же ответы для меньших подзадач сохраняются в массиве или иной структуре данных, называется *динамическим программированием*. В рассмотренной задаче вычисления числа сочетаний использование метода динамического программирования вместо рекурсии позволяет существенно уменьшить время выполнения программы за счет некоторого увеличения объема используемой памяти.

Тем не менее, приведенный алгоритм тоже имеет небольшие недостатки. Мы вычисляем значения всех элементов последней строки, хотя нам необходим только один из них — `B[n][k]`. Аналогично, в предпоследней строке нас интересуют только два элемента — `B[n-1][k-1]` и `B[n-1][k]`, в строке, стоящей над ней — три элемента, и т. д., в то время, как мы вычисляем все элементы во всех строках. Кроме того, мы не используем почти половину созданного массива `B[n+1][n+1]`, а именно все элементы, стоящие выше главной диагонали. От этих недостатков можно избавиться, более того, можно уменьшить объем используемой памяти до $O(n)$, идея для построения такого алгоритма будет изложена в разделе «Маршруты на плоскости».

Если же в программе часто возникает необходимость вычисления числа сочетаний C_n^k для каких-то ограниченных значений n, то лучше всего создать глобальный массив для хранения всевозможных значений C_n^k для нужных нам значений n, заполнить его в начале программы, а затем сразу же брать значение числа сочетаний из этого массива, не вычисляя его каждый раз заново.

Промежуточные итоги

Составим план решения задачи методом динамического программирования.

1) Записать то, что требуется найти в задаче, как функцию от некоторого набора аргументов (числовых, строковых или еще каких-либо).

2) Свести решение задачи для данного набора параметров к решению аналогичных подзадач для других наборов параметров (как правило, с меньшими значениями). Если задача несложная, то полезно бывает выписать явное рекуррентное соотношение, задающее значение функции для данного набора параметров.

3) Задать начальные значения функции, то есть те наборы аргументов, при которых задача тривиальна и можно явно указать значение функции.

4) Создать массив (или другую структуру данных) для хранения значений функции. Как правило, если функция зависит от одного целочисленного параметра, то используется одномерный массив, для функции от двух целочисленных параметров — двумерный массив и т. д.

5) Организовать заполнение массива с начальных значений, определяя очередной элемент массива при помощи выписанного на шаге 2 рекуррентного соотношения или алгоритма.

Далее мы рассмотрим несколько типичных задач, решаемых при помощи динамического программирования.

Маршруты на плоскости

Все задачи этого раздела будут иметь общее начало.

Дана прямоугольная доска размером $n \times m$ (n строк и m столбцов). В левом верхнем углу этой доски находится шахматный король, которого необходимо переместить в правый нижний угол.

Количество маршрутов

Пусть за один ход королю разрешается передвинуться на одну клетку вниз или вправо. Необходимо определить, сколько существует различных маршрутов, ведущих из левого верхнего в правый нижний угол.

Будем считать, что положение короля задается парой чисел (a, b), где a задает номер строки, а b — номер столбца. Строки нумеруются сверху вниз от 0 до $n-1$, а столбцы — слева направо от 0 до $m-1$. Таким образом, первоначальное положение короля — клетка $(0, 0)$, а конечное — клетка $(n-1, m-1)$.

Пусть $W(a, b)$ — количество маршрутов, ведущих в клетку (a, b) из начальной клетки. Запишем рекуррентное соотношение. В клетку (a, b) можно прийти двумя способами: из клетки $(a, b-1)$, расположенной слева, и из клетки $(a-1, b)$, расположенной сверху от данной. Поэтому количество маршрутов, ведущих в клетку (a, b), равно сумме количеств маршрутов, ведущих в клетку слева и сверху от нее. Получили рекуррентное соотношение:

$$W(a, b) = W(a, b-1) + W(a-1, b).$$

Это соотношение верно при $a > 0$ и $b > 0$. Зададим начальные значения: если $a = 0$, то клетка расположена на верхнем краю доски и прийти

в нее можно единственным способом — двигаясь только влево, поэтому $W(0,b)=1$ для всех b. Аналогично, $W(a,0)=1$ для всех a.

Создадим массив `W` для хранения значений функции, заполним первую строку и первый столбец единицами, а затем заполним все остальные элементы массива. Поскольку каждый элемент равен сумме значений, стоящих слева и сверху, заполнять массив `W` будем по строкам сверху вниз, а каждую строку — слева направо.

```
int W[n][m];
int i,j;
for(j=0;j<m;++j)
    W[0][j]=1;          // Первая строка заполнена единицами
for(i=1;i<n;++i)        // i меняется от 1 до n-1
{                       // Заполняем строку с номером i
    W[i][0]=1;          // Элемент в первом столбце равен 1
    for(j=1;j<m;++j)    // Заполняем остальные элементы строки
        W[i][j]=W[i][j-1]+W[i-1][j];
}
```

В результате такого заполнения получим следующий массив (пример для $n=4$, $m=5$):

1	1	1	1	1
1	2	3	4	5
1	3	6	10	15
1	4	10	20	35

Легко видеть, что в результате получился треугольник Паскаля, записанный в немного другом виде, а именно, значение `W[n-1][m-1]` равно C_{n+m-2}^{n-1}. Ничего удивительного, ведь выписанное нами рекуррентное соотношение для количества маршрутов очень похоже на ранее выписанное соотношение для числа сочетаний.

> Действительно, чтобы попасть из клетки $(0,0)$ в клетку $(n-1,m-1)$ король должен сделать $n+m-2$ хода, из которых $n-1$ ход вниз и $m-1$ ход вправо. Поэтому количество различных маршрутов, ведущих из начальной клетки в конечную равно количеству способов выбрать из общего числа $n+m-2$ ходов $n-1$ ход вниз, то есть C_{n+m-2}^{n-1} (и эта же величина равна C_{n+m-2}^{m-1} — количеству выборок $m-1$ хода вправо).

Поскольку для вычисления очередной строки массива `W` нам необходима только предыдущая строка, более того, для вычисления одного элемента очередной строки нам необходим только один элемент

предыдущей строки, стоящий непосредственно над ним, то мы можем обойтись одномерным массивом вместо двумерного, если будем хранить только одну строку двумерного массива, а именно, ту строку, в которой находится рассматриваемый в данный момент элемент. Получим следующую программу:

```
int W[m];           // Массив элементов одной строки
int i,j;
for(j=0;j<m;++j)
  W[j]=1;           // Самая первая строка состоит из единиц
for(i=1;i<n;++i)
{ // Теперь записываем в массив W содержимое i-й строки
    for(j=1;j<m;++j)      // Заполняем все элементы строки,
      W[j]=W[j-1]+W[j];   // кроме первого
}
```

После окончания работы этого алгоритма элемент массива `W[m-1]` содержит искомое число путей.

> **Упражнение**. Напишите программу, вычисляющую значение C_n^k при помощи динамического программирования и использующую одномерный массив из $1+\min(k,n-k)$ элементов.

Как решить задачу нахождения количества маршрутов, если король может передвигаться на одну клетку вниз, вправо или по диагонали вправо-вниз? Решение будет полностью аналогичным, только рекуррентная формула для количества маршрутов изменится:

$$W(a,b)=W(a,b-1)+W(a-1,b)+W(a-1,b-1).$$

Полученный в результате заполнения по такой формуле массив `W` будет выглядеть следующим образом:

1	1	1	1	1
1	3	5	7	9
1	5	13	25	41
1	7	25	63	129

> **Упражнение.** Решите эту задачу с использованием одномерного массива.

Количество маршрутов с препятствиями

Пусть некоторые клетки на доске являются «запретными»: король не может ходить на них. Карта запретных клеток задана при помощи

массива `Map[n][m]`: нулевое значение элемента массива означает, что данная клетка запрещена, единичное значение означает, что в клетку можно ходить. Массив `Map` считывается программой после задания значений `n` и `m`. Король может ходить только вниз или вправо.

Для решения этой задачи придется изменить рекуррентное соотношение с учетом наличия запрещенных клеток. Для запрещенной клетки количество ведущих в нее маршрутов будем считать равным 0. Получим:

$$W(a,b)=\begin{cases} W(a-1,b)+W(a,b-1), & \text{если } \texttt{Map[}a\texttt{][}b\texttt{]}=1,\\ 0, & \text{если } \texttt{Map[}a\texttt{][}b\texttt{]}=0.\end{cases}$$

Также надо учесть то, что для клеток верхней строки и левого столбца эта формула некорректна, поскольку для них не существует соседней сверху или слева клетки.

Например, если $n=4$, $m=5$ и массив `Map` задан следующим образом:

1	1	1	0	1
1	1	1	1	1
1	1	0	1	1
1	1	1	1	1

то искомый массив `W` будет таким:

1	1	1	0	0
1	2	3	3	3
1	3	0	3	6
1	4	4	7	13

Упражнение. Решите эту задачу с использованием одномерного массива. Массив `Map` при этом считывается построчно и в памяти хранится только последний считанный элемент.

Путь максимальной стоимости

Пусть каждой клетке (a,b) доски приписано некоторое число $P(a,b)$ — стоимость данной клетки. Проходя через клетку, мы получаем сумму, равную ее стоимости. Требуется определить максимально возможную сумму, которую можно собрать по всему маршруту, если разрешается передвигаться только вниз или вправо.

Будем решать данную задачу методом динамического программирования. Пусть $S(a,b)$ — максимально возможная сумма, которую можно собрать на пути из начальной клетки в клетку (a,b). Поскольку в клетки верхней строки и левого столбца существует единственно возможный маршрут из начальной клетки, то для них величина $S(a,b)$ определяется как сумма стоимостей всех клеток на пути из начальной вершины в данную (а именно, для начальной клетки $S(0,0)=P(0,0)$, для клеток левого столбца $S(a,0)=S(a-1,0)+P(a,0)$, для клеток верхней строки $S(0,b)=S(0,b-1)+P(0,b)$). Мы задали граничные значения для функции $S(a,b)$.

Теперь построим рекуррентное соотношение. Пусть (a,b) — клетка, не лежащая в первой строке или первом столбце, то есть $a>0$ и $b>0$. Тогда прийти в данную клетку мы можем либо слева, из клетки $(a,b-1)$, и тогда мы сможем набрать максимальную сумму $S(a,b-1)+P(a,b)$, либо сверху, из клетки $(a-1,b)$, тогда мы сможем набрать сумму $S(a-1,b)+P(a,b)$. Естественно, из этих двух величин необходимо выбрать наибольшую:

$$S(a,b)=P(a,b)+\max(S(a,b-1),S(a-1,b)).$$

Дальнейшая реализация алгоритма не вызывает затруднений:

```
int i,j;
S[0][0]=P[0][0];
for(j=1;j<m;++j)
    S[0][j]=S[0][j-1]+P[j];     // Заполняем первую строку
for(i=1;i<n;++i)
{                               // Заполняем i-ю строку
  S[i][0]=S[i-1][0]+P[i][0];  // Заполняем 1-й столбец
  for(j=1;j<m;++j)            // Заполняем остальные столбцы
    if ( S[i-1][j] > S[i][j-1] ) // Выбираем максимум
      S[i][j]=P[i][j]+S[i-1][j]; // из S[i-1][j] и S[i][j-1]
    else                        // и записываем в S[i][j]
      S[i][j]=P[i][j]+S[i][j-1]; // с добавлением P[i][j]
}
```

Рассмотрим пример доски при $n=4$, $m=5$. Для удобства в одной таблице укажем значения обоих массивов: до дробной черты указана стоимость данной клетки, то есть значение `P[i][j]`, после черты — максимальная стоимость маршрута, ведущего в данную клетку, то есть величина `S[i][j]`:

1 / 1	3 / 4	5 / 9	0 / 9	3 / 12
0 / 1	1 / 5	2 / 11	4 / 15	1 / 16
2 / 3	4 / 9	3 / 14	3 / 18	2 / 20
3 / 6	1 / 10	2 / 16	4 / 22	3 / 25

Итак, максимальная стоимость пути из левого верхнего в правый нижний угол в этом примере равна 25.

Как и все предыдущие, данную задачу можно решить с использованием одномерного массива S.

Как изменить данный алгоритм, чтобы он находил не только максимально возможную стоимость пути, но и сам путь? Заведем массив Prev[n][m], в котором для каждой клетки будем хранить ее предшественника в маршруте максимальной стоимости, ведущем в данную клетку: если мы пришли в клетку слева, то запишем в соответствующий элемент массива Prev значение 1, а если сверху — значение 2. Чтобы не путаться, обозначим данные значения константами:

```
const L=1; // Left
const U=2; // Upper
```

В начальную клетку запишем 0, поскольку у начальной клетки нет предшественника. В остальные клетки первой строки запишем L, а в клетки первого столбца запишем U. Остальные элементы массива Prev будем заполнять одновременно с массивом S:

```
if ( S[i-1][j] > S[i][j-1] )
{   // В клетку (i,j) приходим сверху из (i-1,j)
    S[i][j]=P[i][j]+S[i-1][j];
    Prev[i][j]=U;
}
else
{   // В клетку (i,j) приходим слева из (i,j-1)
    S[i][j]=P[i][j]+S[i][j-1];
    Prev[i][j]=L;
}
```

В результате, в приведенном выше примере мы получим следующий массив (значения массива Prev мы запишем в той же таблице после второй черты дроби, то есть в каждой клетке теперь записаны значения P[i][j] / S[i][j] / Prev[i][j]). Клетки, через которые проходит маршрут максимальной стоимости, выделены жирным шрифтом:

1 / 1 / 0	**3 / 4 / L**	**5 / 9 / L**	0 / 9 / L	3 / 12 / L
0 / 1 / U	1 / 5 / U	**2 / 11 / U**	**4 / 15 / L**	1 / 16 / L
2 / 3 / U	4 / 9 / U	3 / 14 / U	**3 / 18 / U**	2 / 20 / L
3 / 6 / U	1 / 10 / U	2 / 16 / U	**4 / 22 / U**	**3 / 25 / L**

После заполнения всех массивов мы можем восстановить путь, пройдя по нему с конца при помощи цикла, начав с конечной клетки и передвигаясь влево или вверх в зависимости от значения соответствующего элемента массива Prev:


```
i=n-1;                  // (i,j) - координаты текущей точки
j=m-1;                  // начинаем с конечной клетки
while (Prev[i][j]!=0)   // проверка, не дошли ли до начальной
{
    if(Prev[i][j]==L)
        j=j-1;          // передвигаемся влево
    else
        i=i-1;          // передвигаемся вверх
}
```

При этом мы получим путь, записанный в обратном порядке. Несложную задачу обращения этого пути и вывода его на экран оставим для самостоятельного решения.

Можно обойтись и без дополнительного массива предшественников Prev. Если имеется заполненный массив S, то предшественника каждой клетки мы можем легко определить, сравнив значения элемента массива S для ее левого и правого соседа и выбрав ту клетку, для которой это значение наибольшее.

```
i=n-1;
j=m-1;
while ( i>0 && j>0 )
{
    if ( S[i][j-1] > S[i-1][j] )
       j=j-1;
    else
       i=i-1;
}
```

Этот алгоритм заканчивается, когда мы выйдем на левый или верхний край доски (цикл while продолжается, пока i>0 и j>0, то есть клетка не лежит на верхнем или левом краю доски), после чего нужно будет двигаться по краю влево или вверх, пока не дойдем до начальной клетки. Это можно реализовать двумя циклами (при этом из этих двух циклов выполняться будет только один, поскольку после выполнения предыдущего фрагмента программы значение одной из переменных i или j будет равно 0):

```
while(i>0)
{   // Движение вверх, пока возможно
    i=i-1;
}
while(j>0)
```

```
{   // Движение влево, пока возможно
    j=j-1;
}
```

Таким образом, если мы хотим восстановить путь наибольшей стоимости, необходимо либо полностью сохранять в памяти значения всего массива `S`, либо хранить в памяти предшественника для каждой клетки. Для восстановления пути потребуется память порядка $O(nm)$. Сложность алгоритма нахождения пути максимальной стоимости также имеет порядок $O(nm)$ и, очевидно, не может быть уменьшена, поскольку для решения задачи необходимо использование каждого из nm элементов данного массива `P`.

Числа Каталана

Рассмотрим произвольное арифметическое выражение. Теперь сотрем в этом выражении всё кроме скобок. Получившуюся последовательность из скобок будем называть «правильной скобочной последовательностью». Любая правильная скобочная последовательность состоит из равного числа открывающихся и закрывающихся скобок. Но этого условия недостаточно: например, скобочная последовательность «(())» является правильной, а последовательность «)()(» — нет.

Можно дать и точное определение правильной скобочной последовательности.

1) Пустая последовательность (то есть не содержащая ни одной скобки) является правильной скобочной последовательностью.
2) Если «A» — правильная скобочная последовательность, то «(A)» (последовательность A, взятая в скобки) — правильная скобочная последовательность.
3) Если «A» и «B» — правильные скобочные последовательности, то «AB» (подряд записанные последовательности A и B) — правильная скобочная последовательность.

Обозначим количество правильных скобочных последовательностей длины $2n$ (то есть содержащих n открывающихся и n закрывающихся скобок) через C_n и решим задачу нахождения C_n по заданной величине n.

Для небольших n значения C_n несложно вычислить полным перебором. $\mathrm{C}_0 = 1$ (правильная скобочная последовательность длины 0 ровно одна — пустая), $\mathrm{C}_1 = 1$ (последовательность «()»), $\mathrm{C}_2 = 2$ (последовательности «(())» и «()()»), $\mathrm{C}_3 = 5$ («((()))», «(()())», «(())()», «()(())», «()()()»).

Запишем рекуррентную формулу для C_n. Пусть X — произвольная правильная скобочная последовательность длины $2n$. Она начинается с открывающейся скобки. Найдем парную ей закрывающуюся скобку и представим последовательность X в виде:

$$X = (A)B,$$

где A и B — тоже правильные скобочные последовательности. Если длина последовательности A равна $2k$, то последовательность A можно составить C_k способами. Тогда длина последовательности B равна $2(n-k-1)$ и последовательность B можно составить C_{n-k+1} способами. Комбинация любого способа составить последовательность A с любым способом составить последовательность B даст новую последовательность X, а величина k может меняться от 0 до $n-1$. Получили рекуррентное соотношение:

$$\mathrm{C}_n = \mathrm{C}_0\mathrm{C}_{n-1} + \mathrm{C}_1\mathrm{C}_{n-2} + \mathrm{C}_2\mathrm{C}_{n-3} + \ldots + \mathrm{C}_{n-2}\mathrm{C}_1 + \mathrm{C}_{n-1}\mathrm{C}_0.$$

Напишем функцию, вычисляющую значение C_n по данному n:

```
int Catalan(int n)
{
    int C[n+1];   // Создаем массив для хранения C[m]
    C[0]=1;
    for (int m=1; m<=n; ++m) // Вычисляем C[m] для m=1..n
    {
        C[m]=0;
        for (int k=0; k<m; ++k)
            C[m]+=C[k]*C[m-1-k];
    }
    return C[n];
}
```

Мы назвали функцию `Catalan`, поскольку значения C_n называются *числами Каталана* в честь бельгийского математика XIX века Евгения Шарля Каталана. Начало ряда чисел Каталана выглядит следующим образом:

n	0	1	2	3	4	5	6	7	8	9	10
C_n	1	1	2	5	14	42	132	429	1 430	4 862	16 796

Для чисел Каталана хорошо известна и нерекуррентная формула:

$$\mathrm{C}_n = \frac{\mathrm{C}_{2n}^n}{n+1} = \frac{(2n)!}{n!(n+1)!}.$$

Более подробно про правильные скобочные последовательности а также элементарное доказательство данной формулы можно прочесть в [1, 2.6–7].

Задача «Банкомат»

Рассмотрим следующую задачу. В обороте находятся банкноты k различных номиналов: a_1, a_2, …, a_k рублей. Банкомат должен выдать сумму в N рублей при помощи минимального количества банкнот или сообщить, что запрашиваемую сумму выдать нельзя. Будем считать, что запасы банкнот каждого номинала неограничены.

Рассмотрим такой алгоритм: будем выдавать банкноты наибольшего номинала, пока это возможно, затем переходим к следующему номиналу. Например, если имеются банкноты в 10, 50, 100, 500, 1000 рублей, то при $N = 740$ рублей такой алгоритм выдаст банкноты в 500, 100, 100, 10, 10, 10, 10 рублей. Подобные алгоритмы называют «жадными», поскольку каждый раз при принятии решения выбирается тот вариант, который кажется наилучшим в данной ситуации (чтобы использовать наименьшее число банкнот каждый раз выбирается наибольшая из возможных банкнот).

Но для решения данной задачи в общем случае жадный алгоритм оказывается неприменимым. Например, если есть банкноты номиналом в 10, 60 и 100 рублей, то при $N = 120$ жадный алгоритм выдаст три банкноты: $100 + 10 + 10$, хотя есть способ, использующий две банкноты: $60 + 60$. А если номиналов банкнот только два: 60 и 100 рублей, то жадный алгоритм вообще не сможет найти решения.

Но эту задачу можно решить при помощи метода динамического программирования. Пусть $F(n)$ — минимальное количество банкнот, которым можно заплатить сумму в n рублей. Очевидно, что $F(0) = 0$, $F(a_1) = F(a_2) = \ldots = F(a_k) = 1$. Если некоторую сумму n невозможно выдать, будем считать, что $F(n) = \infty$ (бесконечность).

Выведем рекуррентную формулу для $F(n)$, считая, что значения $F(0)$, $F(1)$, …, $F(n-1)$ уже вычислены. Как можно выдать сумму n? Мы можем выдать сумму $n - a_1$, а потом добавить одну банкноту номиналом a_1. Тогда нам понадобится $F(n - a_1) + 1$ банкнота. Можем выдать сумму $n - a_2$ и добавить одну банкноту номиналом a_2, для такого способа понадобится $F(n - a_2) + 1$ банкнота и т. д. Из всевозможных способов выберем наилучший, то есть:

$$F(n) = \min(F(n - a_1), F(n - a_2), \ldots, F(n - a_k)) + 1.$$

Теперь заведем массив `F[n+1]`, который будем последовательно заполнять значениями выписанного рекуррентного соотношения. Будем

предполагать, что количество номиналов банкнот хранится в переменной `int k`, а сами номиналы хранятся в массиве `int a[k]`.

```
const int INF=1000000000;// Значение константы "бесконечность"
int F[n+1];
F[0]=0;
int m, i;
for(m=1;m<=n;++i)  // заполняем массив A
{                  // m - сумма, которую нужно выдать
  F[m]=INF;        // помечаем, что сумму i выдать нельзя
  for(i=0;i<k;++i) // перебираем все номиналы банкнот
  {
    if(m>=a[i] && F[m-a[i]]+1<F[m])
      F[m] = F[m-a[i]]+1;// изменяем значение F[m], если нашли
  }                      // лучший способ выдать сумму m
}
```

После окончания этого алгоритма в элементе `F[n]` будет храниться минимальное количество банкнот, необходимых, чтобы выдать сумму `n`. Как теперь вывести представление суммы `n` при помощи `F(n)` банкнот? Опять рассмотрим все номиналы банкнот и значения $n - a_1$, $n - a_2$, …, $n - a_k$. Если для какого-то i окажется, что $F(n - a_i) = F(n) - 1$, значит, мы можем выдать банкноту в a_i рублей и после этого свести задачу к выдаче суммы $n - a_i$, и так будем продолжать этот процесс, пока величина выдаваемой суммы не станет равна 0:

```
if (F[n]==INF)
    cout<<"Требуемую сумму выдать невозможно"<<endl;
else
{
    while(n>0)
    {
        for(i=0;i<k;++i)
        {
            if(F[n-a[i]]==F[n]-1)
            {
                cout<<a[i]<<" ";
                break;
            }
        }
    }
}
```

Задача о рюкзаке

Грабитель, проникший в банк, обнаружил в сейфе k золотых слитков, массами $w_1, w_2, \ldots, w_k$ килограмм. При этом грабитель может унести не более W килограмм. Определите набор слитков, который должен взять грабитель, чтобы унести как можно больше золота.

Эта задача является частным случаем задачи об укладке рюкзака. Сформулируем ее в общем случае.

Дано k предметов, i-й предмет имеет массу $w_i > 0$ и стоимость $p_i > 0$. Необходимо выбрать из этих предметов такой набор, чтобы суммарная масса не превосходила заданной величины W (вместимость рюкзака), а суммарная стоимость была максимальна. Другими словами, нужно определить набор бинарных величин $(b_1, b_2, \ldots, b_k)$, такой, что

$$b_1 w_1 + b_2 w_2 + \ldots + b_k w_k \leqslant W,$$

а величина $b_1 p_1 + b_2 p_2 + \ldots + b_k p_k$ — максимальная. Величина b_i равна 1, если i-й предмет включается в набор, и равна 0 в противном случае.

Задача укладки рюкзака очень сложна. Если перебирать всевозможные подмножества данного набора из k предметов, то получится решение сложности не менее чем $O(2^k)$. В настоящее время неизвестен (и, скорее всего, вообще не существует) алгоритм решения этой задачи, сложность которого является многочленом от k.

Мы рассмотрим решение данной задачи для случая, когда все входные данные — целочисленные, сложность которого будет $O(kW)$.

Рассмотрим следующую функцию. Пусть $A(s, n)$ есть максимальная стоимости предметов, которые можно уложить в рюкзак максимальной вместимости n, если можно использовать только первые s предметов из заданных k.

Зададим краевые значения функции $A(s, n)$.

Если $s = 0$, то $A(0, n) = 0$ для всех n (ни один предмет нельзя брать, поэтому максимальная стоимость равна 0).

Если $n = 0$, то $A(s, 0) = 0$ для всех s (можно брать любые из первых s предметов, но вместимость рюкзака равна 0).

Теперь составим рекуррентное соотношение в общем случае. Необходимо из предметов с номерами $1, \ldots, s$ составить рюкзак максимальной стоимости, чей вес не превышает n. При этом возможно два случая: когда в максимальный рюкзак включен предмет с номером s и когда предмет s не попал в максимальный рюкзак.

Если предмет s не попал в максимальный рюкзак массы n, то максимальный рюкзак будет составлен только из предметов с номерами $1, \ldots, s-1$, следовательно, $A(s, n) = A(s-1, n)$.

Если же в максимальный рюкзак включен предмет s, то масса оставшихся предметов не превышает $n - w_s$, а от добавления предмета s общая стоимость рюкзака увеличивается на p_s. Значит, $A(s, n) = A(s-1, n-w_s) + p_s$. Теперь из двух возможных вариантов составить рюкзак массы, не превосходящей n, из предметов $1, \ldots, s$ нужно выбрать наилучший:

$$A(s, n) = \max\left(A(s-1, n), A(s-1, n-w_s) + p_s\right).$$

Теперь составим программу. Будем считать, что веса предметов хранятся в массиве `w[1]`, ..., `w[k]`, а их стоимости в массиве `p[1]`, ..., `p[k]`. Значения функции $A(s, n)$, где $0 \leqslant s \leqslant k$, $0 \leqslant n \leqslant W$, будем хранить в массиве `A[k+1][W+1]`.

```
int A[k+1][W+1];
for(n=0;n<=W;++n)       // Заполняем нулевую строчку
    A[0][n]=0;
for(s=1;s<=k;++s)       // s - максимальный номер предмета,
{                       // который можно использовать
    for(n=0;n<=W;++n)   // n - вместимости рюкзака
    {
        A[s][n]=A[s][n-1];
        if ( n>=w[s] && ( A[s-1][n-w[s]]+p[s] > A[s][n]) )
            A[s][n] = A[s-1][n-w[s]]+p[s];
    }
}
```

В результате исполнения такого алгоритма в элементе массива `A[k][W]` будет записан ответ на поставленную задачу. Легко видеть, что сложность этого алгоритма, равно как и объем используемой им памяти, являются величиной $O(kW)$.

Рассмотрим пример работы этого алгоритма. Пусть максимальная вместимость рюкзака $W = 15$, количество предметов $k = 5$, их стоимости и массы таковы:
$w_1 = 6$, $p_1 = 5$, $w_2 = 4$, $p_2 = 3$, $w_3 = 3$, $p_3 = 1$,
$w_4 = 2$, $p_4 = 3$, $w_5 = 5$, $p_5 = 6$.

В приведенной ниже таблице указаны значения заполненного массива `A[k+1][W+1]`.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$s=0$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$s=1$	0	0	0	0	0	0	5	5	5	5	5	5	5	5	5	5
$s=2$	0	0	0	0	3	3	5	5	5	5	8	8	8	8	8	8
$s=3$	0	0	0	1	3	3	5	5	5	6	8	8	8	9	9	9
$s=4$	0	0	3	3	3	4	6	6	8	8	8	9	11	11	11	12
$s=5$	0	0	3	3	3	6	6	9	9	9	10	12	12	14	14	14

Первая строка массива соответствует значениям $A(0,n)$. Поскольку ни одного предмета брать нельзя, то строка заполнена нулями: из пустого множества предметов можно составить рюкзак нулевой массы.

Вторая строка массива соответствует значению $s=1$, то есть рюкзак можно составлять только из первого предмета. Вес этого предмета $w_1=6$, а его стоимость $p_1=5$. Поэтому при $n<6$ мы не можем включить этот предмет в рюкзак и значение $A(1,n)$ равно 0 при $n<6$. Если $n\geqslant w_1$, то мы можем включить первый предмет в рюкзак, а поскольку других предметов нет, то $A(1,n)=5$ (так как $p_1=5$).

Рассмотрим третью строку массива, соответствующую двум предметам ($s=2$). Добавляется второй предмет, более легкий и менее ценный, чем первый ($w_2=4,\ p_2=3$). Поэтому $A(2,n)=0$ при $n<4$ (ни один предмет взять нельзя), $A(2,n)=3$ при $n=4$ и $n=5$ (в рюкзак включается предмет номер 2 ценности 3), $A(2,n)=5$ при $6\leqslant n\leqslant 9$ (при данном n выгоднее в рюкзак включить предмет 1, поскольку его ценность выше) и, наконец, $A(2,n)=8$ при $n\geqslant 10$ (при данной вместимости рюкзака можно взять оба предмета).

Аналогично заполняются остальные строки массива, при заполнении элемента $A(s,n)$ рассматривается две возможности: включать или не включать предмет с номером s.

Как теперь вывести на экран тот набор предметов, который входит в максимальный рюкзак? Сравним значение `A[k][W]` со значением `A[k-1][W]`. Если они равны, то максимальный рюкзак можно составить без использования предмета с номером `k`. Если не равны, то предмет с номером `k` обязательно входит в максимальный рюкзак. В любом случае, задача печати рюкзака сводится к задаче печати рюкзака для меньшего числа предметов. Напишем это в виде рекурсивной функции `Print(int s, int n)`, которая по параметрам `s` и `n` печатает номера предметов, входящих в максимальный рюкзак массой не более `n` и составленный из предметов 1, ..., `s`:

```
void Print(int s, int n)
{
  if (A[s][n]==0) // максимальный рюкзак для параметров (s,n)
    return;       // имеет нулевую ценность,
                  // поэтому ничего не выводим
  else if (A[s-1][n] == A[s][n])
    Print(s-1,n);  // можно составить рюкзак без предмета s
  else
  {
    Print(s-1,n-w[s]); // Предмет s должен обязательно
    cout<<s<<endl;     // войти в рюкзак
  }
}
```


Для печати искомого рюкзака необходимо вызвать функцию с параметрами `(k,W)`.

В приведенном примере для печати максимального рюкзака вызовем функцию `Print(5,15)`. Поскольку $A(5,15)=14$, а $A(4,15)=12$ (с использованием только первых 4 предметов мы можем собрать рюкзак максимальной стоимости 12, а с использованием всех 5 предметов — стоимости 14), предмет номер 5 обязательно входит в рюкзак. Далее рассмотрим $A(4,10)$ (общая вместимость рюкзака уменьшилась на вес включенного предмета). Поскольку $A(4,10)=A(3,10)=A(2,10)=8$, то мы можем исключить из рассмотрения предметы номер 4 и 3 — можно собрать рюкзак вместимости 10 и стоимости 8 только из первых двух предметов. Для этого необходимо включить оба этих предмета. Таким образом, оптимальный рюкзак будет состоять из предметов 1, 2, 5, его масса будет равна $6+4+5=15$, а стоимость — $5+3+6=14$.

Можно составить рюкзак и по-другому. Поскольку вес предмета 4 равен двум, его стоимость p_4 равна трём, а $A(4,10)=A(3,8)+p_4$, то мы можем включить в наш рюкзак предмет 4. Теперь рассмотрим $A(3,8)=5$ — как составить рюкзак массы не более 8 и стоимости 5 из первых трех предметов. Поскольку $A(3,8)=A(2,8)=A(1,8)=5$, то мы исключаем из рассмотрения предметы 3 и 2, но включаем предмет 1. Получим рюкзак из предметов 1, 4, 5, его масса будет $6+2+5=13$ и стоимость также равна $5+3+6=14$. Поскольку стоимость обоих полученных рюкзаков получилась одинаковой, а масса в каждом случае не превосходит максимально допустимого значения $W=15$, то оба решения подходят.

Литература

1. *А. Шень.* Программирование: теоремы и задачи. М: МЦНМО, 2004.

2. *Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн.* Алгоритмы: построение и анализ. М: Издательский дом «Вильямс», 2005.