Информатика

Горшенин Юрий Васильевич
Студент Московского физико-технического института (МФТИ), II курс, факультет инноватики высоких технологий (ФИВТ).

Динамика, рекурсия, перебор

В данной статье мы рассмотрим *динамическое программирование сверху* – простая идея, которая позволяет эффективно вычислять значения функций, задаваемых рекуррентными соотношениями. Этот метод не требует больших затрат от программиста. По сути, необходимо по рекуррентной формуле записать рекурсивную функцию и добавить к ней пару строчек – одну в начало, а другую в конец.

Часто для величины, которую необходимо вычислить, есть некоторое соотношение. Так, например, для чисел Фибоначчи это соотношение записывается следующим образом:

$$F_n = F_{n-1} + F_{n-2} -$$

каждое число Фибоначчи равно сумме двух предыдущих. Первые два числа по определению равны 1, а следующие получаются по цепочке:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Когда элемент последовательности выражен через предыдущие, то говорят, что последовательность удовлетворяет **рекуррентному соотношению**. Для чисел Фибоначчи рекуррентное соотношение дано в определении. Оно достаточно простое, по нему несложно сконструировать **алгоритм вычисления** чисел. Школьники часто уже на третьем-пятом занятии по информатике способны написать программу, которая выводит первые числа Фибоначчи.

Но на практике встречаются ситуации, когда рекуррентные формулы более сложные и изначально не известные. Мы рассмотрим одну из задач, где получение рекуррентной формулы является важной частью решения. При этом у нас возникнет необходимость введения скрытого параметра k, и вычисляемая величина будет зависеть не только от параметра n, но и от параметра k.

Формулировка задачи

Дано натуральное число n. Посчитать количество разложений данного натурального числа на неупорядоченные натуральные слагаемые (неупорядоченные – это значит, что нам не важен тот порядок, в котором они следуют). Например, для $n=5$ есть 7 различных разложений $5=4+1=3+2=3+1+1=2+2+1=2+1+1+1=1+1+1+1+1$. Разложения считаются различными, если множества слагаемых различаются. n меньше 100.

Формат ввода: натуральное число N.

Формат вывода: искомое число разложений.

Примеры:

вход	выход
5	7
15	176

Пусть $P(n)$ – искомое число разложений натурального числа n на слагаемые (то есть ответ задачи). Пусть мы нашли некоторое разложение числа n на s таких слагаемых, то есть $n=\sum_{i=1}^{s} a_i$. Так как нам не важен порядок (нас же просят найти **неупорядоченные** разложения в сумму), то пусть $a_i \geq a_{i+1}$, $i=\overline{1,s-1}$. То есть самое первое a_1 не меньше всех остальных. Другими словами, выбирая a_1, мы ставим ограничения на все остальные члены в разложении. Они будут не больше a_1.

Задача. Найдите рекуррентную формулу для $P(n)$.

Если вам не удалось решить эту задачу, не переживайте. Если вы её решили, то вас следует поздравить, потому что вы – будущий Эйлер!

Дело в том, что простой рекуррентной формулы для $P(n)$ не существует.

Оказывается, простой путь к решению данной задачи лежит через усложнение её формулировки!

Давайте вместо величины $P(n)$ будем искать набор величин $p(n,k)$ для различных k, где $p(n,k)$ равно числу разложений n на слагаемые, каждое из которых не больше k. Заметим, что $P(n)=p(n,n)$. Значит, если мы научимся вычислять $p(n,k)$, то научимся вычислять и $P(n)$.

Задача. Найдите рекуррентную формулу для $p(n,k)$.

Для $p(n,k)$ есть простая рекуррентная формула:

$$p(n,k)=p(n-k,k)+p(n,k-1).$$

Докажем её. Мы договорились, что слагаемые идут по убыванию. Рассмотрим два возможных варианта. Первое слагаемое в разложении можно взять равным k и продолжить разложение уже числа $n-k$ на слагаемые, также не превосходящие k. Либо можно взять разложение на слагаемые меньше, чем k.

Таким образом, разложения числа n на слагаемые, не превосходящие k, можно разбить на две непересекающихся части:

1) разложения, которые начинаются на k, их $p(n-k,k)$ штук;

2) разложения, которые начинаются на слагаемые меньше k, их $p(n,k-1)$ штук.

Мы получили рекуррентное соотношение для $p(n,k)$, но это лишь первый шаг решения задачи. Необходимо получить ещё **граничные** условия, то есть значения $p(n,k)$ для пар (n, k), в которых k или n равно 1.

Очевидно, что $p(n,1)=1$, так как разложить натуральное число на неупорядоченную сумму единиц мы можем только одним способом. Кроме того, $p(1,k)=1$ – тоже очевидно.

Замечу, что в коде вводятся дополнительные условия: $p(0,0)=1$, так как 0 мы раскладываем на 0 слагаемых одним способом. И проверка на отрицательность параметров n и k.

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

Эта функция правильно вычисляет значение $p(n,k)$. Но беда в том, что работает она достаточно долго. Попробуйте вызвать эту функцию так: $p(100,100)$ – и вы не сможете дождаться конца вычислений. Попробуем разобраться, в чём дело. Изменим исходный код следующим образом:

```
const size_t N = 100;
const size_t K = 100;

unsigned int call[N][K], total;

unsigned int p(int n, int k)
{
 total++;
 if(n >= 0 && k >= 0)
   call[n][k]++;
 if(n < 0 || k <= 0) return 0;
 if(n == 0 || k == 1) return 1;
 return p(n - k, k) + p(n, k - 1);
}
```

Теперь в переменной **total** будет храниться общее число вызовов функции $p(n,k)$. Величина **call[n][k]** есть число вызовов $p(n,k)$ для конкретных n и k. Если после вызова $p(n,k)$ просмотреть содержимое массива **call**, то мы обнаружим, что, помимо обилия нулевых и единичных элементов, весьма немало элементов со значением больше, чем 1. Это значит только одно – в процессе вычисления функция с одними и теми же параметрами вызывалась не один раз. А так как она рекурсивная, то

она автоматически тянула за собой заново кучу рекурсивных вызовов. А теперь посмотрим на значение переменной **total** для конкретных значений n. Будем брать в качестве n степени двойки в диапазоне от 1 до 64 включительно. Значения **total** соответственно будут: 1, 3, 13, 77, 781, 23459, 4290797. Числа растут достаточно быстро. Можно показать, что зависимость числа вызовов **total** от n *экспоненциальная*.

Когда физики или информатики говорят, что функция $f(x)$ *растёт экспоненциально*, они обычно имеют в виду, что есть некоторое положительное число $A>1$ и положительное число C такие, что $f(x)>C\cdot A^x$. То есть функцию можно ограничить снизу показательной положительной функцией с основанием больше 1.

А так как **total** – это оценочное число действий алгоритма, то рост времени работы алгоритма (как функция от n) тоже экспоненциальный. Это очень плохо. С другой стороны, после этого анализа поведения программы становится очевидной оптимизация. Действительно, зачем, если мы уже когда-то вычислили для каких-то конкретных значений n и k значение функции $p(n,k)$, считать его снова и снова? Можно просто его запомнить в ячейке двумерного массива, в которой первый индекс соответствует параметру n, а второй – параметру k. И в случае запроса на вычисление значения $p(n,k)$ мы можем либо вернуть уже вычисленное, либо, если не вычислили, вычислить, запомнить в соответствующей ячейке массива, а потом уже вернуть. В общем, рабочая программа будет такой:

```
#include <iostream>
using namespace std;

const size_t N = 100;
const size_t K = 100;
unsigned int preCalc[N][K];

unsigned int p(int n, int k) {
 if (n < 0 || k <= 0) return 0;
 if (n == 0 || k == 1) return 1;
 if (preCalc[n][k] == 0)
   preCalc[n][k] = p(n - k, k) + p(n, k - 1);
 return preCalc[n][k];
}

int main(void) {
  int n;
  cin >> n;
  cout << p(n, n) << endl;
  return 0;
}
```

Глобальные переменные по умолчанию заполняются нулями. Поэтому элементы массива **preCalc** (как, впрочем, и переменная **total** с массивом **call**) будут нулевыми. В данном случае ноль в ячейке массива – это индикатор того, что с данными параметрами функция ещё не вызывалась, то есть надо честно вычислять это значение и присваивать элементу массива. Обратите внимание, что мы предполагаем, что вычисленное значение $p(n,k)$ не равно 0. Действительно, любое натуральное число **всегда** представимо в виде неупорядоченной суммы.

Задача генерации разложений

У обычного рекурсивного решения есть свои преимущества. Например, оно пишется быстрее, чем рекурсия с запоминанием. Кроме того, из него с помощью малых изменений можно получить решение задачи генерации всех разложений.

Ниже приведён код, который не вычисляет количество разложений, а выводит все разложения данного числа в сумму на экран.

```
#include <algorithm>
#include <iostream>
#include <iterator>
using namespace std;

const size_t N = 100;
unsigned int ans[N];

void p(int n, int k, int pos) {
  if (n == 0) {
    copy(ans, ans + pos, ostream_iterator<unsigned int>(cout, " "));
    cout << endl;
  } else  {
    while (k > 0) {
      ans[pos] = k;
      p(n - k, min(k, n - k), pos + 1);
      k--;
    }
  }
}

int main(void) {
  int n0;
  cin >> n0;
  p(n0, n0, 0);
  return 0;
}
```

Примеры ввода/вывода:

вход	выход
2	2 1 1
5	5 4 1 3 2 3 1 1 2 2 1 2 1 1 1 1 1 1 1 1

В этой программе i-й элемент массива ans содержит a_{i+1} после того, как генерация очередной суммы завершена. Несоответствие индексов на 1 – это из-за того, что в C/C++ индексы с 0, а не 1. Рекурсивная функция p организована так, что n – текущее число, которое нужно разложить. Аргумент k задаёт ограничение на слагаемые: слагаемые должны быть не выше k. Аргумент pos равен номеру следующего слагаемого. Когда аргумент $n = 0$, это значит, что $n_0 = \sum_{i=0}^{pos-1} ans[i]$. То есть очередное разложение сгенерировано, надо его вывести на экран. Мы это и делаем длинной страшной строчкой. Результат её действия – все элементы массива ans от 0-го до $(pos-1)$-го. Если же $n \neq 0$, то генерация ещё не завершена. И мы просто в цикле устанавливаем конкретное значение **ans[pos]** и продолжаем генерацию. С новым значением $k = \min\{k, n-k\}$, чтобы не возникало на следующем этапе рекурсивного вызова ситуации, когда $n-k<0$. Другими словами, мы нормируем аргументы. Для функции **min** мы как раз и подключили модуль **algorithm**. В нём содержится множество часто используемых алгоритмов, в том числе и функция нахождения минимума из 2-х элементов. Можно использовать оператор **<?**, и тогда последняя строчка выглядела бы как `p(n - k, k <? n - k, pos + 1);`. Однако не все компиляторы C/C++ это поддерживают.

Строчка «**copy(**ans, ans + pos, ostream_iterator<**unsigned int**>**(**cout, " "**));**» требует объяснения. **copy(first, last, dest)** – это стандартный алгоритм копирования из одной последовательности в другую. **first** и **last** – это начало и конец копируемой последовательности. **dest** – то место, начиная с которого, будет производиться заполнение.

```
template<class inIt, class outIt>
outIt copy(inIt first, inIt last, outIt dest)
{
  while (first != last)
    *dest++ = *first++;
  return dest;
}
```

Это примерная реализация алгоритма **copy**. Как мы видим, **first** и **last** должны задавать полуинтервал и быть одного типа (логично, раз они есть некие величины от одной и той же последовательности). А **dest** – куда копируем. Обратите внимание, что мы пока не упоминали о типе **first**, **last** и **dest**. А он и не нужен – это может быть что угодно, что поддерживает операции инкремента (++) и разыменования (*). А ведь обычный указатель поддерживает обе операции. Имя массива является указателем на его первый элемент (правильнее сказать – нулевой). Так можно быстро копировать массивы. Например, команда «скопировать n элементов из массива a в массив b» будет записываться как «copy(a, a + n, b)». Выражение «*имя массива* + n» равно указателю на n-й элемент массива. Поэтому командой «copy(ans, ans + pos, ..) осуществляется копирование элементов ans[0], ans[1], ... , ans[pos – 1]. Заметьте, что элемент ans[pos] не копируется. Третий аргумент функции **copy** значительно интереснее. Это итератор, который задаёт место, куда производится копирование. В двух словах, итератор – нечто, абстрагирующее понятие указателя на элемент последовательности. Итератор должен поддерживать как минимум операции разыменования * , -> (называйте это не «звёздочка со стрелочкой», а **селекторами классов**) и инкремента. Для массива итератор – это **int***, для потока вывода **cout** для вывода целых чисел – это ostream_iterator<**unsigned int**>(cout, " "). То есть если

```
ostream_iterator<unsigned int> it = ostream_iterator<unsigned int>(cout, " "),
```

то после вызова *it = x; it++; беззнаковое целое число x будет выведено на экран, а за ним разделитель – в данном случае символ пробела. Обратите внимание, что после связки разыменование/присвоение в случае итераторов вывода обязательно должен следовать инкремент, чтобы подготовить поток к следующей порции данных (это зависит от компилятора: в некоторых реализациях инкремент можно не делать, но если сделаете, проблем с переносимостью программ точно не будет).

Замечания

Итак, было рассказано про рекурсию с запоминанием – довольно мощный приём, который также называется *динамическим программированием сверху*. Достаточно нетривиален был переход от $P(N)$ к $p(n,k)$. Он требовал определённого опыта, интуиции и смекалки. Что касается непосредственно вычислений, то мы могли бы организовать два вложенных цикла и так считать значения $p(n,k)$, но это не всегда хорошо. Хотя бы потому, что нам потребовалось бы перебрать все значения n и k, а в рекурсивной форме часть значений просто не требуется вычислять. Теперь, благодаря рекурсии с запоминанием, оценка времени работы нашего алгоритма есть $O(N^2)$, то есть время работы квадратично зависит от N. Это можно наблюдать эмпирически на массиве флажков, а можно попытаться доказать. В случае двух циклов доказательство тривиально.

Предложенное решение не является самым оптимальным. Для вели-

чины $P(N)$ существует рекуррентная формула Эйлера:

$$P(N)=P(N-1)+P(N-2)-P(N-5)-$$
$$-P(N-7)+P(N-12)+P(N-15)-$$
$$-P(N-22)-P(N-26)+$$
$$+P(N-35)+P(N-40)+\dots,$$

которая позволяет вычислять $P(N)$ за время $O(N^{3/2})$. Как формула продолжается, и как чередуются знаки – догадайтесь сами. В этой формуле предполагается, что $P(N)=0$ при всех $N<0$.

Задачи для самостоятельного решения

Задача А. Дано натуральное число N. Найти количество разложений N на неупорядоченные натуральные слагаемые **без повторений**. Для $N=5$ есть 3 различных разложения $5=5=4+1=3+2$. Разложения считаются различными, если множества слагаемых различаются. Известно, что N меньше 100.

Формат ввода: натуральное число N.

Формат вывода: искомое число разложений.

Примеры:

ВХОД	ВЫХОД
5	3
15	27

Задача Б. Сгенерировать все возможные разложения натурального числа N на неупорядоченные натуральные слагаемые без повторений. Известно, что N меньше 100.

Формат ввода: натуральное число N.

Формат вывода: искомые разложения. Каждое разложение должно быть представлено одной строкой, числа в которой должны быть записаны через пробел.

Примеры:

ВХОД	ВЫХОД
2	2
5	5 4 1 3 2

Список литературы

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штейн К. «Алгоритмы. Построение и анализ», 2-е издание – классика по алгоритмам.
2. Страуструп Б. «Язык программирования C++» – отличное руководство по языку C++ от его создателя.