Авторы:

Медведев Михаил Геннадиевич – кандидат физико-математических наук, доцент кафедры математической информатики факультета кибернетики Киевского национального университета имени Тараса Шевченко.

Присяжнюк Анатолий Васильевич – учитель-методист высшей категории специализированной школы с углубленным изучением информатики № 17 г. Бердичев Житомирской области.

Жуковский Сергей Станиславович – старший преподаватель кафедры прикладной математики и информатики Житомирского государственного университета имени Ивана Франко, учитель информатики городского лицея №25 имени Н. А. Щорса.

Задачи:

1212, 1511, 1513 – 1520.

ОГЛАВЛЕНИЕ

ТЕОРЕТИЧЕСКИЙ МАТЕРИАЛ 4
УСЛОВИЯ ЗАДАЧ 9
1212. Бесконечная последовательность 2 9
1511. Разрезание торта 9
1513. Прямой, центрированный и обратный порядок 10
1514. Истина, спрятанная в рекуррентности 11
1515. Повторяющийся Иосиф 11
1516. Создание двоичного дерева поиска 12
1517. Простое сложение 13
1518. Разбиение треугольника 14
1519. Коды Грея 14
1520. Нечетные делители 15
АНАЛИЗ ЗАДАЧ 16
1212. Бесконечная последовательность 2 16
1511. Разрезание торта 16
1513. Прямой, центрированный и обратный порядок 17
1514. Истина, спрятанная в рекуррентности 17
1515. Повторяющийся Иосиф 17
1516. Создание двоичного дерева поиска 19
1517. Простое сложение 20
1518. Разбиение треугольника 20
1519. Коды Грея 20
1520. Нечетные делители 21
РЕАЛИЗАЦИЯ ЗАДАЧ 22
1212. Бесконечная последовательность 2 22
1511. Разрезание торта 22
1513. Прямой, центрированный и обратный порядок 24
1514. Истина, спрятанная в рекуррентности 24
1515. Повторяющийся Иосиф 25
1516. Создание двоичного дерева поиска 25
1517. Простое сложение 26
1518. Разбиение треугольника 26
1519. Коды Грея 27
1520. Нечетные делители 28

ТЕОРЕТИЧЕСКИЙ МАТЕРИАЛ

Описывается два основных способа организации обработки данных: итеративный и рекурсивный. Рассматривается набор олимпиадных задач, которые решаются при помощи итеративного и рекурсивного подхода.

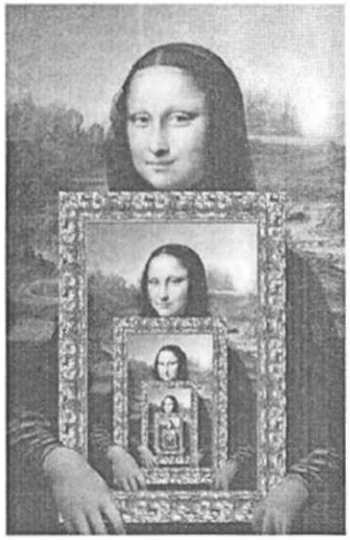
Когда мы начинаем познавать азы программирования, как правило первой написанной нами является программа, печатающая строку «Hello, world!». Потом знакомятся с переменными, операторами, функциями. И как правило, первыми, с которыми начинает знакомиться новичок, являются условный оператор и оператор цикла. Сразу же появляется желание написать какую-нибудь простую функцию: факториал числа, возведение в степень или вычисление биномиального коэффициента. При этом в большинстве случаев начинающий программист реализует итеративный вариант функций. Однако мало кто знает, что любую итеративную функцию можно реализовать и рекурсивно.

Рекурсией называется такой способ организации обработки данных, при котором программа (или функция) вызывает сама себя или непосредственно, или из других программ (функций).

Функция называется ***рекурсивной***, если во время ее обработки возникает ее повторный вызов, либо непосредственно, либо косвенно, путем цепочки вызовов других функций.

Итерацией называется такой способ организации обработки данных, при котором некоторые действия многократно повторяются, не приводя при этом к рекурсивным вызовам программ (функций).

Теорема. Произвольный алгоритм, реализованный в рекурсивной форме, может быть переписан в итерационной форме и наоборот.



Далее рассмотрим набор элементарных функций, реализованных как при помощи операторов цикла, так и при помощи рекурсивного подхода. Перед написанием рекурсивных функций на любом языке программирования, как правило, необходимо записать ***рекуррентное соотношение***, определяющее метод вычисления функций. Рекуррентное соотношение должно содержать как минимум два условия:

I) условие продолжения рекурсии (шаг рекурсии);

II) условие окончания рекурсии.

Рекурсию будем реализовывать посредством вызова функции самой себя. При этом в теле функции сначала следует проверять условие продолжения рекурсии. Если оно истинно, то выходим из функции. Иначе совершаем рекурсивный шаг.

Итеративный вариант функций будем реализовывать при помощи оператора цикла **for**.

1. Факториал числа. Факториалом целого неотрицательного числа n называется произведение всех натуральных чисел от 1 до n и обозначается $n!$. Если $f(n) = n!$, то имеет место рекуррентное соотношение:

$$\begin{cases} f(n) = n * f(n-1), \\ f(0) = 1 \end{cases}$$

Первое равенство описывает шаг рекурсии – метод вычисления $f(n)$ через $f(n-1)$. Второе равенство указывает, когда при вычислении функции следует остановиться. Если его не задать, то функция будет работать бесконечно долго.

Например, значение $f(3)$ можно вычислить следующим образом:

$$f(3) = 3 * f(2) = 3 * 2 * f(1) = 3 * 2 * 1 * f(0) = 3 * 2 * 1 * 1 = 6$$

Очевидно, что при вычислении $f(n)$ следует совершить n рекурсивных вызовов.

рекурсивная реализация	*циклическая реализация*

`int f(int n)`
`{`
`  if(!n) return 1;`
`  return n * f(n - 1);`
`}`
	`int f(int n)`
	`{`
	`  int i, res = 1;`
	`  for(i = 1; i <= n; i++) res = res * i;`
	`  return res;`
	`}`

Идея циклической реализации состоит в непосредственном вычислении факториала числа при помощи оператора цикла:

$$f(n) = 1 * 2 * 3 * \ldots * n$$

2. Степень числа за линейное время. Вычисление степени числа $f(a, n) = a^n$ с линейной ($O(n)$) временной оценкой можно определить при помощи следующего рекуррентного соотношения:

$$\begin{cases} f(a,n) = a * f(a,n-1), \\ f(a,0) = 1 \end{cases}$$

рекурсивная реализация	*циклическая реализация*

`int f(int a,int n)`
`{`
`  if (!n) return 1;`
`  return a * f(a, n - 1);`
`}`
	`int f(int a,int n)`
	`{`
	`  int i, res = 1;`
	`  for(i = 0; i < n; i++) res = res * a;`
	`  return res;`
	`}`

В итерационном варианте достаточно вычислить произведение $a * a * \ldots * a$ (n множителей a).

3. Степень числа за логарифмическое время. Вычисление степени числа $f(a, n) = a^n$ с временной оценкой $O(\log_2 n)$ определим следующим образом:

$$\begin{cases} f(a,n) = a * f(a^2, \lfloor n/2 \rfloor), n \text{ нечетное} \\ f(a,n) = f(a^2, \lfloor n/2 \rfloor), n \text{ четное} \\ f(a,0) = 1 \end{cases}$$

Например, возведение в десятую степень можно реализовать так:

$$a^{10} = \left(a^5\right)^2 = \left(a \cdot \left(a^2\right)^2\right)^2$$

Поскольку возведение в квадрат эквивалентно одному умножению, то для вычисления a^{10} достаточно совершить 4 умножения.

рекурсивная реализация	*циклическая реализация*

`int f(int a, int n)`
`{`
`  if (!n) return 1;`
`  if (n & 1) return a * f(a * a, n / 2);`
`  return f(a * a, n / 2);`
`}`
	`int f(int a, int n)`
	`{`
	`  int  res = 1;`
	`  while(n > 0)`
	`  {`
	`  if (n & 1) res *= a;`
	`  n >>= 1; a *= a;`
	`  }`
	`  return res;`
	`}`

4. Сумма цифр числа. Сумму цифр натурального числа n можно найти при помощи функции $f(n)$, определенной следующим образом:

$$\begin{cases} f(n) = n \bmod 10 + f(n/10), \\ f(0) = 0 \end{cases}$$

Условие продолжения рекурсии: сумма цифр числа равна последней цифре плюс сумма цифр числа без последней цифры (числа, деленного нацело на 10).

Условие окончания рекурсии: Если число равно 0, то сумма его цифр равна 0.

Например, сумма цифр числа 234 будет вычисляться следующим образом:

$$f(234) = 4 + f(23) = 4 + 3 + f(2) = 4 + 3 + 2 + f(0) = 4 + 3 + 2 + 0 = 9$$

рекурсивная реализация	*циклическая реализация*

`int f(int n)`
`{`
`  if (!n) return 0;`
`  return n % 10 + f(n / 10);`
`}`
	`int f(int n)`
	`{`
	`  int res = 0;`
	`  for(; n>0; n = n / 10) res = res + n % 10;`
	`  return res;`
	`}`

5. Число единиц. Количество единиц в двоичном представлении числа n можно вычислить при помощи функции $f(n)$, определенной следующим образом (& - операция побитового 'И'):

$$\begin{cases} f(n) = 1 + f(n \,\&\, (n-1)), \\ f(0) = 0 \end{cases}$$

В результате операции $n = n \,\&\, (n-1)$ уничтожается последняя единица в двоичном представлении числа n:

$$n = a_1 a_2 \ldots a_{k-1} a_k 10 \ldots 0$$
$$n - 1 = a_1 a_2 \ldots a_{k-1} a_k 01 \ldots 1$$
$$n \,\&\, (n-1) = a_1 a_2 \ldots a_{k-1} 000 \ldots 0$$

Рекурсивный вызов функции f будет совершаться столько раз, сколько единиц в двоичном представлении числа n.

рекурсивная реализация	*циклическая реализация*

`int f(int n)`
`{`
`  if (!n) return 0;`
`  return 1 + f(n & (n - 1));`
`}`
	`int f(int n)`
	`{`
	`  int res = 0;`
	`  for(; n > 0; n = n & (n - 1)) res++;`
	`  return res;`
	`}`

6. Биномиальный коэффициент. Значение биномиального коэффициента равно

$$C_n^k = \frac{n!}{k!(n-k)!}$$

и определяется рекуррентным соотношением:

$$C_n^k = \begin{cases} C_{n-1}^{k-1} + C_{n-1}^k, n > 0 \\ 1, k = n \text{ или } k = 0 \end{cases}$$

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

Учитывая, что $C_n^k = \frac{n(n-1)\ldots(n-k+1)}{1 \cdot 2 \cdot \ldots \cdot k}$, значение биномиального коэффициента можно вычислить при помощи цикла. При этом все операции деления будут целочисленными. Если $k > n - k$, то следует воспользоваться соотношением $C_n^k = C_n^{n-k}$ во избежании Time Limit при вычислении, например, значения $C_{100000000}^{100000000}$.

```
int Cnk(int k, int n)
{
  long long res = 1;
  if (k > n - k) k = n - k;
  for(int i = 1; i <= k; i++)
    res = res * (n - i + 1) / i;
  return (int)res;
}
```

7. Рекурсивная функция. Для заданного натурального n вычислим значение функции $f(n)$, заданной рекуррентными соотношениями:

$$f(2 * n) = f(n),$$
$$f(2 * n + 1) = f(n) + f(n + 1),$$
$$f(0) = 0, f(1) = 1$$

Непосредственная реализация функции $f(n)$ имеет вид:

```
int f(int n)
{
  if (n <= 1) return n;
  if (n % 2) return f(n / 2) + f(n / 2 + 1);
  return f(n / 2);
}
```

При такой реализации некоторые значения функции f могут вычисляться несколько раз. Рассмотрим другой подход к вычислению значений f. Определим функцию

$$g(n, i, j) = i * f(n) + j * f(n + 1),$$

для которой имеют место равенства:

$$g(2 * n, i, j) = g(n, i + j, j),$$
$$g(2 * n + 1, i, j) = g(n, i, i + j),$$
$$g(0, i, j) = i * f(0) + j * f(1) = j$$

Используя приведенные соотношения, можно вычислить значение $f(n) = g(n, 1, 0)$ с временной оценкой O(log n).

```
int g(int n, int i, int j)
{
  if (!n) return j;
  if (n % 2) return g(n / 2, i, i + j);
  return g(n / 2, i + j, j);
}

int f(int n)
{
  return g(n, 1, 0);
}
```

8. Функция Аккермана. Функция Аккермана $A(m, n)$ определяется рекурсивно следующим образом:

$$A(0, n) = n + 1,$$
$$A(m, 0) = A(m - 1, 1),$$
$$A(m, n) = A(m - 1, A(m, n - 1))$$

Рекурсивная реализация функции Аккермана имеет вид:

```
int a(int m, int n)
{
  if (!m) return n + 1;
  if (!n) return a(m - 1, 1);
  return a(m - 1, a(m, n - 1));
}
```

Для малых значений m функцию Аккермана можно выразить явно:

$$A(0, n) = n + 1$$
$$A(1, n) = 2 + (n + 3) - 3 = n + 2$$
$$A(2, n) = 2 * (n + 3) - 3 = 2 * n + 3$$
$$A(3, n) = 2^{n+3} - 3$$

УСЛОВИЯ ЗАДАЧ

1212. Бесконечная последовательность 2

Рассмотрим бесконечную последовательность A, определенную следующим образом:

$$A_i = 1, i \le 0,$$
$$A_i = A_{\lfloor i/p \rfloor - x} + A_{\lfloor i/q \rfloor - y}, i \ge 1$$

По заданным n, p, q, x, y вычислить A_n.

Вход. Целые значения n, p, q, x, y ($0 \le n \le 10^{13}$, $2 \le p, q \le 10^9$, $0 \le x, y \le 10^9$).

Выход. Значение A_n.

Пример входа

```
12 2 3 1 0
```

Пример выхода

```
8
```

1511. Разрезание торта

Имеется прямоугольный торт длины *length* и ширины *width*. Мы хотим разрезать его на pieces прямоугольных кусков равной площади. Каждый разрез должен совершаться параллельно сторонам торта, и должен полностью разрезать один из имеющихся кусков на две части. (Для разрезания торта на n кусков необходимо совершить $n - 1$ разрез)

Квадратные куски Вы предпочитаете тем, которые имеют большее отношение сторон. Под "отношением сторон" будем понимать отношение длины большей стороны куска к меньшей. Вам следует разрезать торт таким образом, чтобы минимизировать максимальное значение отношения сторон полученных кусков.

Например, если мы хотим разрезать торт 2x3 на шесть кусков, то это можно сделать, разрезав его на шесть кусков размера 1x1. Отношение сторон каждого куска равно 1.0, что является наименьшим возможным. Поэтому решение оптимально.

Один из возможных вариантов разрезать торт 5x5 на 5 кусков состоит в следующем: сначала разрезаем торт на две части размерами 2x5 и 3x5. Меньшую часть делим пополам (получаем две части размером 2 x 5/2), а большую часть делим на три части (каждая имеет размер 3 x 5/3). Большее отношение сторон достигается на куске 3 x 5/3 и равно 3/(5/3) = 1.8. Разделить торт на 5 частей равной площади с меньшим отношением сторон, нежели 1.8 невозможно.

Вход. Три целых числа: длина *length* и ширина *width* торта, и количество прямоугольных кусков *pieces*, на которое следует разрезать торт. Известно, что $1 \le length, width \le 1000$, $1 \le pieces \le 10$.

Выход. Разрезать торт так, чтобы минимизировать максимальное значение отношения сторон полученных кусков. Вывести полученное отношение сторон с 4 десятичными цифрами. Помните, что все полученные куски должны иметь одинаковую площадь!

Пример входа

```
2 3 6
5 5 5
950 430 9
```

Пример выхода

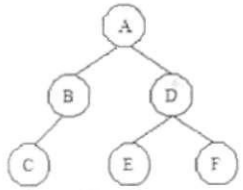
```
1.0000
1.8000
1.2573
```

1513. Прямой, центрированный и обратный порядок

Классическими методами обхода деревьев являются:

- ***прямой***: посещается корень, левое поддерево, правое поддерево;
- ***центрированный***: посещается левое поддерево, корень, правое поддерево;
- ***обратный***: посещается левое поддерево, правое поддерево, корень;

Рассмотрим рисунок:



Прямой, центрированный и обратный обходы соответственно дадут следующие последовательности вершин: ABCDEF, CBAEDF, CBEFDA. В задаче требуется найти последовательность вершин при обратном обходе, если известны прямой и центрированный обходы.

Вход. Первая строка содержит количество тестов C (C ≤ 2000). Каждая следующая строка является отдельным тестом и содержит количество вершин в бинарном дереве n ($1 \le n \le 52$) и две строки S_1 и S_2 , содержащие соответственно прямой и центрированный обход дерева. Вершины дерева пронумерованы разными символами из множеств *a..z* и *A..Z*. Значения n, S_1 и S_2 разделены пробелом.

Выход. Для каждого теста вывести последовательность вершин при обратном обходе дерева.

Пример входа

```
3
3 xYz Yxz
3 abc cba
6 ABCDEF CBAEDF
```

Пример выхода

```
Yzx
Cba
CBEFDA
```

1514. Истина, спрятанная в рекуррентности

Рекурсивная функция задана следующим образом:

$f(0, 0) = 1$,

$f(n, r) = \sum_{i=0}^{k-1} f(n-1, r-i)$, если $n > 0$ и $0 \le r \le n(k-1)+1$,

$f(n, r) = 0$ иначе.

Вычислить значение $x = \sum_{i=0}^{n(k-1)} f(n, i) \bmod m$, где $m = 10^t$.

Например, значения f(n, i) при $k = 3$ имеют вид (в пустых клетках стоят нули):

$n \setminus i$	0	1	2	3	4	5	6	7	8
0	1								
1	1	1	1						
2	1	2	3	2	1				
3	1	3	6	7	6	3	1		
4	1	4	10	16	19	16	10	4	1

Вход. Каждая строка содержит три целых числа: k ($0 < k < 10^{19}$), n ($0 < n < 10^{19}$) и t ($0 < t < 10$). Последняя строка содержит три нуля и не обрабатывается.

Выход. Для каждого теста в отдельной строке вывести номер теста и значение x. Формат вывода приведен в примере.

Пример входа

```
1234 1234 4
2323 99999999999 8
4 99999 9
888 888 8
0 0 0
```

Пример выхода

```
Case #1: 736
Case #2: 39087387
Case #3: 494777344
Case #4: 91255296
```

1515. Повторяющийся Иосиф

По кругу стоят n людей, занумерованных от 1 до n. Начиная отсчет с первого и двигаясь по кругу, будем казнить каждого второго человека, до тех пор пока не останется один. Пусть этот выживший имеет номер x. Расставим по кругу x людей и повторим процедуру, после которой выживет человек с номером y. И так далее до тех пор, пока номер выжившего не станет равным первоначальному количеству людей в текущем раунде.

Например, при $n = 5$ последовательно будут казнены 2, 4, 1, 5. Выживет номер 3. Он не равен 5 (количеству людей в раунде), поэтому следует повторить процедуру. Для $n = 3$ казнены будут 2, 1. Выживет человек с номером 3, равным n. Процедура заканчивается.

Вход. Входные данные состоят из нескольких тестов. Каждый тест в одной строке содержит одно число n ($0 < n \le 30000$)

Выход. Для каждого теста вывести в отдельной строке его номер как указано в примере, количество повторений процедуры казни после первой итерации и номер выжившего в конце процедуры.

Пример входа

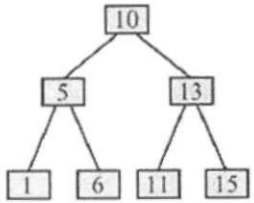
```
2
13
23403
```

Пример выхода

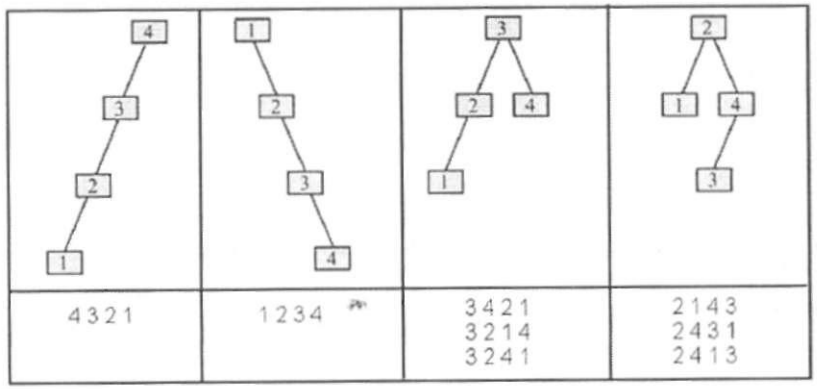
```
Case 1: 2 7
Case 2: 8 1023
```

1516. Создание двоичного дерева поиска

БДП (бинарное дерево поиска) является эффективной структурой для поиска. В БДП все элементы левого поддерева меньше, а все элементы правого поддерева больше чем значение корня. Рассмотрим пример БДП:



Обычно БДП строится в результате последовательной вставки элементов. В таком случае последовательность вставки элементов влияет на структуру результирующего дерева. Например:



В этой задаче Вам необходимо найти такой порядок вставки чисел от 1 до n, чтобы полученное БДП имело высоту не больше h. Высота БДП определяется следующим образом:

1. Высота БДП, которое не содержит ни одной вершины, равна 0.
2. Иначе высота БДП равна 1 плюс максимум высот левого и правого поддерева.

Условию задачи могут удовлетворять несколько последовательностей вставок. В таком случае следует вывести последовательность, в которой сначала идут меньшие числа. Например, для $n = 4$, $h = 3$ следует вывести последоватлельность 1 3 2 4, а не 2 1 4 3 или 3 2 1 4.

Вход. Каждый тест содержит два натуральных числа n ($1 \le n \le 10000$) и h ($1 \le h \le 30$). Последний тест содержит $n = 0$, $h = 0$ и не обрабатывается. На вход подается не более 30 тестов.

Выход. Результат каждого теста следует вывести в отдельной строке. Каждая строка начинается с "Case #: ", где "#" – номер теста. Дальше в этой же строке следует вывести последовательность из n целых чисел – порядок вершин, в котором они будут вставляться в БДП высоты не более h. В конце строки не должно быть пробелов. Если требуемое дерево построить нельзя, то вывести "Impossible." (без кавычек).

Пример входа

```
4 3
4 1
6 3
0 0
```

Пример выхода

```
Case 1: 1 3 2 4
Case 2: Impossible.
Case 3: 3 1 2 5 4 6
```

1517. Простое сложение

Определим следующую рекурсивную функцию f(n):

$$f(n) = \begin{cases} n\%10, \text{ если } n\%10 > 0 \\ 0, \text{ если } n = 0 \\ f(n/10) \text{ иначе} \end{cases}$$

Определим функцию S(p, q) следующим образом:

$$S(p, q) = \sum_{i=p}^{q} f(i)$$

По заданным p и q необходимо вычислить S(p, q).

Вход. Состоит из нескольких тестов. Каждая строка содержит два неотрицательных целых числа p и q ($p \le q$), разделенных пробелом. p и q являются 32 битовыми знаковыми целыми. Последняя строка содержит два отрицательных целых числа и не обрабатывается.

Выход. Для каждой пары p и q в отдельной строке вывести значение S(p, q).

Пример входа

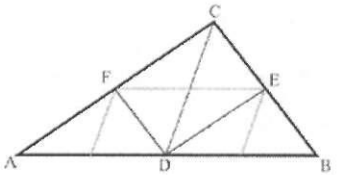
```
1 10
10 20
30 40
-1 -1
```

Пример выхода

```
46
48
52
```

1518. Разбиение треугольника

Треугольник можно разбить на два треугольника, проведя медиану к его большей стороне (на рисунке сверху такое разбиение показано красным разрезом). Затем два меньших треугольника можно подобным образом разделить на четыре треугольника (на рисунке такое разбиение показано синими разрезами). Процесс разрезания треугольников будем продолжать до бесконечности.



Математики заметили, что при описанном разрезании мы получим конечное количество "стилей" треугольников, которые отличаются друг от друга только размером. По заданным длинам сторон исходного треугольника необходимо определить количество стилей треугольников, которое можно получить. Два треугольника принадлежат одному стилю, если они подобны.

Вход. Первая строка содержит количество тестов n ($0 < n < 35$). Каждая следующая строка содержит три целых числа a, b, c ($0 < a, b, c < 100$) – стороны треугольника. Известно, что площадь каждого входного треугольника положительна.

Выход. Для каждого теста в отдельной строке вывести его номер, как показано в примере, и целое число t – количество разных стилей треугольников, которое получится в процессе указанного деления. Считать, что значение t всегда меньше 100.

Пример входа

```
2
3 4 5
12 84 90
```

Пример выхода

```
Triangle 1: 3
Triangle 2: 41
```

1519. Коды Грея

Бинарные коды Грея генерируются следующим образом. Рассмотри последовательность

0
1
-

Отобразим строки вниз относительно горизонтальной черты, припишем к первой половине строк спереди 0, а ко второй отображенной половине 1. Получим последовательность:

00
01
11
10

Продолжая процесс, на следующем шаге получим последовательность из 8 чисел. Справа от кода находится его десятичное значение.

000 0
001 1
011 3
010 2
110 6
111 7
101 5
100 4

Приведенные последовательности называются кодами Грея длины n = 1, 2, 3. Всего существует 2^n разных кодов длины n. Каждые два соседних кода отличаются одним битом.

Вход. Первая строка содержит количество тестов (не более 250000). Каждая следующая строка содержит два числа: n ($1 \le n \le 30$) и k ($0 \le k < 2^n$).

Выход. Для каждого теста вывести число, которое находится в k - ой позиции последовательности кодов Грея длины n.

Пример входа	Пример выхода
14	0
1 0	1
1 1	0
2 0	1
2 1	3
2 2	2
2 3	0
3 0	1
3 1	3
3 2	2
3 3	6
3 4	7
3 5	5
3 6	4
3 7	

1520. Нечетные делители

Пусть $f(n)$ – наибольший нечетный делитель натурального числа n. По заданному натуральному n необходимо вычислить значение суммы $f(1) + f(2) + ... + f(n)$.

Вход. Каждая строка содержит одно натуральное число n ($n \le 10^9$).

Выход. Для каждого значения n в отдельной строке вывести значение суммы $f(1) + f(2) + ... + f(n)$.

Пример входа

```
7
1
777
```

Пример выхода

```
21
1
201537
```

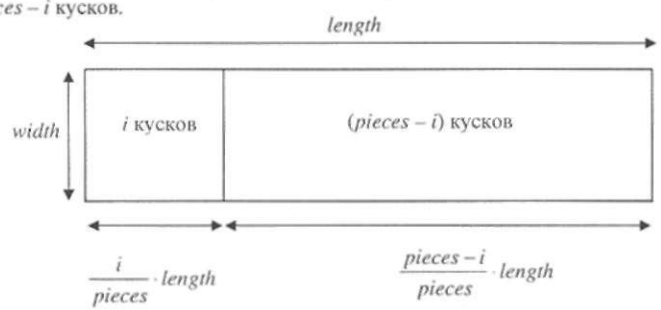
АНАЛИЗ ЗАДАЧ

1212. Бесконечная последовательность 2

Поскольку $n \le 10^{13}$, то запоминать значения A_i (i = 0, 1, …, n) последовательности невозможно ни при помощи массива, ни при помощи структуры map. Поэтому запрограммируем рекурсию как указано в рекуррентном соотношении, но при этом значения A_i, для которых $i < 5000000$, будем запоминать в массиве m.

1511. Разрезание торта

Благодаря верхнему ограничению на число кусков *pieces*, задача может быть просто решена полным перебором разрезаний торта. Если кусок размером *length* на *width* следует разбить на *pieces* кусков, то после совершения первого разреза (который по условию задачи можно совершить либо по горизонтали, либо по вертикали) площади двух полученных кусков должны быть пропорциональны числам 1 и *pieces* – 1, или 2 и *pieces* – 2 и так далее. То есть после первого разреза должны получаться два куска размером *i* * *length* / *pieces* на *width* и (*pieces* – *i*) * *length* / *pieces* на *width*, или *length* на *i* * *width* / *pieces* и *length* на (*pieces* – *i*) * *width* / *pieces*, где $1 \le i <$ *pieces*. При этом дальше первый кусок следует делить на *i* кусков, а второй на *pieces* – *i* кусков.



Совершаем разрез исходного куска на две части и далее рекурсивно запускаем разрезание каждой из полученных частей. Ищем такое разрезание, при котором максимум отношения кусков является наименьшим.

Временная оценка работы алгоритма

Пусть f(*pieces*) – функция, которая возвращает количество вызовов функции cut в зависимости от значения *pieces*.

Очевидно, что f(1) = 1, так как в этом случае сразу после вызова функции выйдем по команде return.

При *pieces* = 2 первый вызов функции cut будет со значением *pieces* = 2. Далее кусок будем стараться разделить пополам вертикальным разрезом, в результате чего дважды будет вызвана f(1). Потом попробуем совершить горизонтальный разрез, снова будет дважды вызвана f(1). Итого получим f(2) = 5 вызовов функции cut.

Пусть *pieces* = 3. Первый вертикальный разрез разобьет кусок на две части, первый из которых далее надо будет делить на 1 часть, а второй на 2 части. Второй вертикальный разрез также разобьет кусок на две части, первый из которых далее надо будет делить на 2 части, а второй на 1 часть. Аналогичное количество вызовов функции следует произвести при горизонтальных разрезах. Итого

$$f(3) = 1 + 2 * ((f(1) + f(2)) + (f(2) + f(1))) = 1 + 2 * 2 * (1 + 5) = 25$$

В общем случае $f(n) = 1 + 2\sum_{i=1}^{n-1}\left(f(i) + f(n-i)\right) = 1 + 4\sum_{i=1}^{n-1} f(i)$.

Например:

$$f(4) = 1 + 4\sum_{i=1}^{3} f(i) = 1 + 4 * (1 + 5 + 25) = 125,$$

$$f(5) = 1 + 4\sum_{i=1}^{4} f(i) = 1 + 4 * (1 + 5 + 25 + 125) = 625$$

Докажем методом математической индукции, что $f(n) = 5^{n-1}$.

База индукции. $f(1) = 5^0 = 1$.

Шаг индукции. $f(n) = 1 + 4\sum_{i=1}^{n-1} f(i) = 1 + 4\,(5^0 + 5^1 + \ldots + 5^{n-2}) = 1 + 4\frac{1-5^{n-1}}{1-5} = 5^{n-1}$.

Временная оценка работы программы составляет $O(5^{pieces})$.

1513. Прямой, центрированный и обратный порядок

Корень дерева (обозначим его через A) содержится в начале последовательности прямого обхода. Пусть последовательность прямого обхода имеет вид $Ax_2x_3\ldots x_n$, центрированного – $y_1y_2\ldots y_kAy_{k+2}\ldots y_n$. В центрированном обходе ищем корень A. Тогда левое поддерево содержит вершины $y_1y_2\ldots y_k$ (всего k вершин), а правое $y_{k+2}\ldots y_n$.

1514. Истина, спрятанная в рекуррентности

Рассмотрим все n - цифровые числа в системе исчисления с основанием k (включая числа с ведущими нулями). Общее их количество равно k^n. Пусть f(n, r) – количество таких чисел, сумма цифр которых равна r. Тогда

$$f(n, r) = f(n-1, r) + f(n-1, r-1) + \ldots + f(n-1, r-k+1) = \sum_{i=0}^{k-1} f(n-1, r-i).$$

Минимальная сумма цифр для таких чисел равна 0, максимальная (k – 1) * n. Просуммировав значения f(n, r) для r от 0 до (k – 1) * n, получим общее количество n - цифровых чисел в системе исчисления с основанием k, то есть k^n.

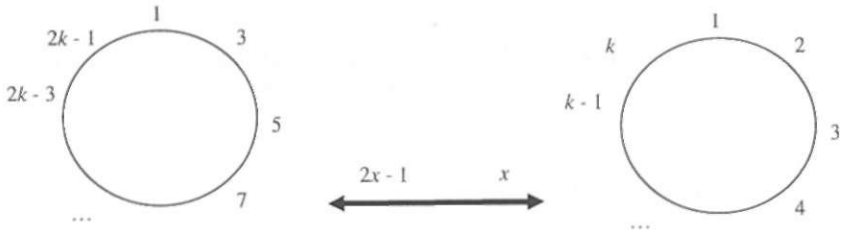
Таким образом $x = k^n \pmod{10^t}$. Поскольку $t < 10$, то при вычислении модулярной экспоненты достаточно использовать беззнаковый 64-битный целочисленный тип.

Пример. Для первого теста имеет место равенство: $1234^{1234} \pmod{10^4} = 736$.

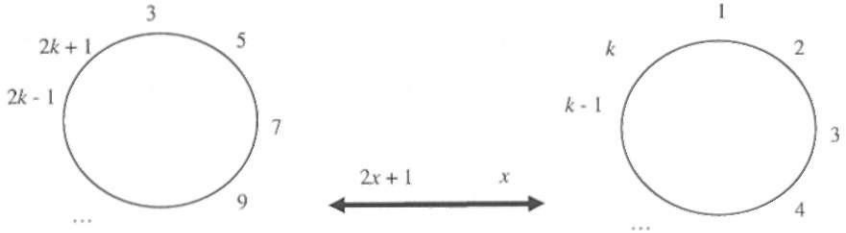
1515. Повторяющийся Иосиф

Пусть n – количество людей в круге. Обозначим через f(n) номер последнего уцелевшего. Положим f(1) = 1.

Если $n = 2k$ – четное, то после прохода первого круга будут удалены люди с четными номерами: 2, 4, ..., $2k$. Останутся люди с нечетными номерами, а отсчет продолжаем с номера 1. Это все равно, что если бы у нас было k людей, а номер каждого удвоился и уменьшился на 1. То есть получим соотношение $f(2k) = 2\,f(k) - 1$.



Если $n = 2k + 1$ – нечетное, то после прохода первого круга будут удалены люди с четными номерами 2, 4, ..., $2k$, а жертва с номером 1 уничтожается сразу же после жертвы с номером $2k$. Остается k людей с номерами 3, 5, 7, ..., $2k + 1$. Это все равно, что люди занумерованы от 1 до k, только номер каждого удвоился и увеличился на 1. Получаем соотношение: $f(2k + 1) = 2\,f(k) + 1$.



Объединяя полученные соотношения, получим рекуррентность (1):

$$f(1) = 1$$
$$f(2k) = 2\,f(k) - 1,\ k \ge 1$$
$$f(2k + 1) = 2\,f(k) + 1,\ k \ge 1$$

Составим таблицу первых значений $f(n)$.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$f(n)$	1	1	3	1	3	5	7	1	3	5	7	9	11	13	15	1

Если сгруппировать значения n по степеням 2, то в каждой группе $f(n)$ будет всегда начинаться с 1, а затем увеличиваться на 2. Если записать n в виде $n = 2^m + l$, где 2^m – наибольшая степень 2, не превосходящая n, а l – то что останется, то решение рекуррентного соотношения должно быть следующим:

$$f(2^m + l) = 2l + 1, \text{ где } m \ge 0,\ 0 \le l < 2^m \quad (2)$$

Последнее неравенство справедливо, так как $l = n - 2^m < 2^{m+1} - 2^m = 2^m$.

Доказательство последнего соотношения проведем по индукции по m.

База индукции. При $m = 0$ получим $l = 0$, откуда $f(1) = 1$.

Шаг индукции. Имеем $m > 0$. Индуктивный шаг следует провести как для четного, так и нечетного значения l. Если l четно, то

$$f(2^m + l) = 2\,f(2^{m-1} + l / 2) - 1 = 2\,(2\,l / 2 + 1) - 1 = 2l + 1$$

Если l нечетно, то

$$f(2^m + l) = 2\,f(2^{m-1} + (l - 1) / 2) + 1 = 2\,(2\,(l - 1) / 2 + 1) + 1 = 2l + 1$$

Из рекуррентности (1), например, следует соотношение $f(2k + 1) - f(2k) = 2$.

Пример. $f(100) = f(2^6 + 36) = 2 * 36 + 1 = 73$, $f(73) = f(2^6 + 9) = 2 * 9 + 1 = 19$.

Теорема. Значение $f(n)$ получается путем циклического сдвига двоичного представления числа n влево на один бит. Например, $f(100) = f(1100100_2) = 1001001_2 = 73$, $f(73) = f(1001001_2) = 10011_2 = 19$.

Доказательство. Рассмотрим бинарное представление числа

$$n = b_m b_{m-1} \ldots b_1 b_0 = 1 b_{m-1} \ldots b_1 b_0 \text{ (первый бит } b_m \text{ обязательно равен 1)}$$

Если $n = 2^m + l$, то $l = 0 b_{m-1} \ldots b_1 b_0$. Тогда $2l + 1 = b_{m-1} \ldots b_1 b_0 1$. Исходя из равенства $f(2^m + l) = 2l + 1$, получим

$$f(1 b_{m-1} \ldots b_1 b_0) = b_{m-1} \ldots b_1 b_0 1 \text{ или } f(b_m b_{m-1} \ldots b_1 b_0) = b_{m-1} \ldots b_1 b_0 b_m,$$

что и требовалось доказать.

Многократное применение функции f порождает последовательность убывающих значений, достигающих неподвижной точки n такой что $f(n) = n$. Число n будет состоять из одних единиц со значением $2^{v(n)} - 1$, где $v(n)$ – количество единиц в бинарном представлении числа n.

Пример. Рассмотрим входные данные для второго теста. При $n = 13$ последовательно будут казнены 2, 4, 6, 8, 10, 12, 1, 5, 9, 13, 7, 3. Выживет номер 11. Он не равен 13 (количеству людей в раунде), поэтому следует повторить процедуру. Для $n = 11$ казнены будут 2, 4, 6, 8, 10, 1, 5, 9, 3, 11. Выживет человек с номером 7, равным n. При $n = 7$ выживет номер 7. После первой итерации проведено еще 2 повторения процедуры казни.

$f(13) = f(1101_2) = 1011_2 = 11$, $f(11) = f(1011_2) = 111_2 = 7$, $f(7) = 7$.

1516. Создание двоичного дерева поиска

Полное бинарное дерево высоты h содержит $1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$ вершин. Если $n \ge 2^h$, то искомого дерева не существует. Иначе будем строить такое дерево, в котором будет по максимуму заполняться правое поддерево.

Пусть следует расположить в дереве поиска высоты не более h числа от a до b. Тогда в правом поддереве высоты $h - 1$ следует расположить $2^{h-1} - 1$ элементов, а число $d = b - 2^{h-1} + 1$ следует расположить в корне. Числа от a до $d - 1$ располагаем в левом поддереве, а числа от $d + 1$ до b в правом.

Если $d < a$, то в левом поддереве чисел не будет и в таком случае положим $d = a$.

Пример. Рассмотрим второй пример, где $n = 6$, $h = 3$. Дерево высоты 3 может содержать до $2^3 - 1 = 7$ вершин. В корне дерева расположим число $d = 6 - (2^2 - 1) = 3$. Рекурсивно вставляем числа от 1 до 2 в левое поддерево и числа от 4 до 6 в правое. Получим последовательность 3 1 2 5 4 6.

1517. Простое сложение

Приведенная в условии функция f(n) находит последнюю ненулевую цифру числа n. Обозначим через

$$g(p) = \sum_{i=1}^{p} f(i)$$

Тогда $S(p, q) = g(q) - q(p - 1)$. Для вычисления функции $g(p)$, суммы последних значащих цифр для чисел от 1 до p, разобьем числа от 1 до p на три множества (операция деления '/' является целочисленной):

1. Числа от (p/10)*10 + 1 до p;
2. Числа от 1 до (p/10)*10, не оканчивающиеся нулем;
3. Числа от 1 до (p/10)*10, оканчивающиеся нулем;

Например, при $p = 32$ к первому множеству отнесутся числа 31, 32, ко второму 1, …, 9, 11, …, 19, 21, …, 29, к третьему 10, 20.

Сумма последних значащих цифр в первом множестве равна $1 + 2 + \ldots + p\%10 = t(1 + t) / 2$, где $t = p \% 10$. Во втором множестве искомая сумма равна $p / 10 * 45$, так как сумма всех цифр от 1 до 9 равна 45, а число полных десятков равно $p / 10$. Требуемую сумму для третьего множества найдем рекурсивно: она равна $g(p / 10)$.

1518. Разбиение треугольника

Два треугольника считаются подобными (принадлежат одному стилю), если отношения их сторон одинаковы. Стилем треугольника со сторонами (a, b, c) будем называть пару $\left(\frac{b}{a}, \frac{c}{a}\right)$. Тогда два треугольника будут подобными, если их стили одинаковы.

Промоделируем процесс разрезания треугольников медианами, запоминая их стили. Если на очередном шаге получим треугольник уже имеющегося стиля, то дальше его разрезать не будем. Поскольку по условию задачи стилей не больше 100, то и число разрезаний будет удовлетворять этому условию.

Для каждого треугольника отсортируем длины его сторон: $a \le b \le c$. То есть самой длинной стороной бедет c. Длина медианы, проведенной к ней, равна

$$m = \frac{1}{2}\sqrt{2a^2 + 2b^2 - c^2}$$

Таким образом, треугольник со сторонами (a, b, c) будет разрезан на два треугольника со сторонами (a, m, $c/2$) и (b, m, $c/2$).

1519. Коды Грея

Запишем рекурсивную функцию find(n, k), которая будет находить число в k - ой позиции последовательности кодов Грея длины n. Если значение k лежит в первой части последовательности ($k < 2^{n-1}$, так как позиции нумеруются с нуля), то следует искать число, стоящее в k - ой позиции кодов Грея длины $n - 1$. Иначе воспользуемся симметрией при построении кодов Грея: результат будет равен 2^{n-1} плюс число, стоящее в $(2^n - k - 1)$ - ой позиции кодов Грея длины $n - 1$.

Таким образом $\text{find}(n, k) = \begin{cases} \text{find}(n-1, k), & \text{если } k < 2^{n-1} \\ 2^{n-1} + \text{find}(n-1, 2^n - k - 1), & \text{если } k \ge 2^{n-1} \end{cases}$

1520. Нечетные делители

Если число n нечетное, то $f(n) = n$. Если число n четное, то $f(n) = f(n / 2)$.

Пусть $g(n) = f(1) + f(2) + \ldots + f(n)$.

Разобьем множество натуральных чисел от 1 до n на два подмножества: нечетных ODD = {1, 3, 5, …, $2k - 1$} и четных EVEN = {2, 4, 6, …, $2l$} чисел. Предполагаем, что среди натуральных чисел от 1 до n имеется в точности $k = \left\lfloor \frac{n+1}{2} \right\rfloor$ нечетных и $l = \left\lfloor \frac{n}{2} \right\rfloor$ четных чисел.

Тогда $f(1) + f(3) + f(5) + \ldots + f(2k - 1) = 1 + 3 + 5 + \ldots + (2k - 1) = \frac{1 + 2k - 1}{2} \cdot k = k^2$.

В то же время $f(2) + f(4) + f(6) + \ldots + f(2l) = f(1) + f(2) + f(3) + \ldots + f(l) = g(l) = g\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$.

Таким образом $g(n) = k^2 + g\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$. Для окончания рекурсии положим $g(0) = 0$.

Пример. Рассмотрим первый тест: $g(7) = \left\lfloor \frac{7+1}{2} \right\rfloor^2 + g\left(\left\lfloor \frac{7}{2} \right\rfloor\right) = 16 + g(3) = 16 + \left\lfloor \frac{3+1}{2} \right\rfloor^2 + g\left(\left\lfloor \frac{3}{2} \right\rfloor\right) = 16 + 4 + g(1) = 16 + 4 + 1 = 21$.

РЕАЛИЗАЦИЯ ЗАДАЧ

1212. Бесконечная последовательность 2

Для хранения значений A_i ($i < 5000000$) объявим массив m.

```
#define MAX 5000000
long long m[MAX];
```

Функция calc вычисляет значение A_n.

```
long long calc(long long n, long long p, long long q,
               long long x, long long y)
{
  long long temp;
```

Если $n \le 0$, то $A_n = 1$.

```
  if (n <= 0) return 1;
```

Если $n < 5000000$ и значение m[n] уже вычислено (не равно нулю), то возвращаем его.

```
  if ((n < MAX) && m[n]) return m[n];
  temp = calc(n/p-x,p,q,x,y) + calc(n/q-y,p,q,x,y);
```

Если $n < 5000000$, то запоминаем A_n в массиве m чтобы избежать в дальнейшем повторных вычислений.

```
  if (n < MAX) m[n] = temp;
  return temp;
}
```

Основная часть программы. Читаем входные данные, вычисляем и выводим ответ.

```
scanf("%lld %lld %lld %lld %lld",&n,&p,&q,&x,&y);
res = calc(n,p,q,x,y);
printf("%lld\n",res);
```

1511. Разрезание торта

Функция cut возвращает наименьшее возможное максимальное значение отношения сторон полученных *pieces* кусков в результате разрезания прямоугольника длины *length* и ширины *width*.

```
double cut(double length, double width, int pieces)
{
  double temp, res = 1e100;
  int i;
```

Если *pieces* равно 1, то возвращаем отношение сторон текущего куска.

```
  if (pieces == 1) return max(length,width) / min(length,width);
```

Совершаем вертикальный разрез торта на две части, каждая из которых дальше будет делиться на i и $pieces - i$ кусков.

```
  for(i = 1; i < pieces; i++)
  {
    temp = max(cut(i * length / pieces,width,i),
               cut((pieces - i) * length / pieces,width,pieces - i));
    if (temp < res) res = temp;
  }
```

Совершаем горизонтальный разрез торта на две части, каждая из которых дальше будет делиться на i и $pieces - i$ кусков.

```
  for(i = 1; i < pieces; i++)
  {
    temp = max(cut(length,i * width / pieces,i),
               cut(length,(pieces - i) * width / pieces,pieces - i));
    if (temp < res) res = temp;
  }
  return res;
}
```

Основной цикл программы.

```
while(scanf("%d %d %d",&length, &width, &pieces) == 3)
{
  double res = cut(length,width,pieces);;
  printf("%.4lf\n",res);
}
```

Временная оценка работы алгоритма

Пусть f(*pieces*) – функция, которая возвращает количество вызовов функции cut в зависимости от значения *pieces*.

Очевидно, что f(1) = 1, так как в этом случае сразу после вызова функции выйдем по команде return.

При *pieces* = 2 первый вызов функции cut будет со значением *pieces* = 2. Далее кусок будем стараться разделить пополам вертикальным разрезом, в результате чего дважды будет вызвана f(1). Потом попробуем совершить горизонтальный разрез, снова будет дважды вызвана f(1). Итого получим f(2) = 5 вызовов функции cut.

Пусть *pieces* = 3. Первый вертикальный разрез разобьет кусок на две части, первый из которых далее надо будет делить на 1 часть, а второй на 2 части. Второй вертикальный разрез также разобьет кусок на две части, первый из которых далее надо будет делить на 2 части, а второй на 1 часть. Аналогичное количество вызовов функции следует произвести при горизонтальных разрезах. Итого

$$f(3) = 1 + 2 * ((f(1) + f(2)) + (f(2) + f(1))) = 1 + 2 * 2 * (1 + 5) = 25$$

В общем случае $f(n) = 1 + 2\sum_{i=1}^{n-1}\left(f(i) + f(n-i)\right) = 1 + 4\sum_{i=1}^{n-1} f(i)$.

Например:

$$f(4) = 1 + 4\sum_{i=1}^{3} f(i) = 1 + 4 * (1 + 5 + 25) = 125,$$

$$f(5) = 1 + 4\sum_{i=1}^{4} f(i) = 1 + 4 * (1 + 5 + 25 + 125) = 625$$

Докажем методом математической индукции, что $f(n) = 5^{n-1}$.

База индукции. $f(1) = 5^0 = 1$.

Шаг индукции. $f(n) = 1 + 4\sum_{i=1}^{n-1} f(i) = 1 + 4\,(5^0 + 5^1 + \ldots + 5^{n-2}) = 1 + 4\frac{1-5^{n-1}}{1-5} = 5^{n-1}$.

Временная оценка работы программы составляет $O(5^{preccr})$.

1513. Прямой, центрированный и обратный порядок

В символьных массивах pre_order и in_order будем хранить последовательность вершин при прямом и центрированном обходе дерева.

```
char pre_order[53],in_order[53];
```

Пусть последовательность вершин при прямом обходе некоего поддерева содержится в ячейках от pre_order[*prea*] до pre_order[*preb*], а при центрированном – в ячейках от in_order[*ina*] до in_order[*inb*]. Тогда функция post_order напечатает это поддерево в обратном порядке.

```
void post_order(int ina, int inb, int prea,int preb)
{
  int lsize,rt_in;
  char root;
  if (ina == inb) return;
  root = pre_order[prea];
  for(rt_in = ina; rt_in < inb; rt_in++)
    if (in_order[rt_in] == root) break;
  lsize = rt_in - ina;
  post_order(ina,rt_in,prea+1,prea+lsize);
  post_order(rt_in+1,inb,prea+1+lsize,preb);
  printf("%c",root);
}
```

Читаем число тестов *n*. Для каждого теста читаем количество вершин дерева *d* и последовательность вершин при прямом и центрированном обходе дерева. Запускаем процедуру post_order, которая и выводит искомый обратный порядок вершин.

```
scanf("%d",&n);
for(i = 0; i < n; i++)
{
  scanf("%d %s %s",&d,pre_order,in_order);
  post_order(0,strlen(pre_order),0,strlen(in_order));
  printf("\n");
}
```

1514. Истина, спрятанная в рекуррентности

При вычислении используем беззнаковый 64-битовый целый тип unsigned long long.

Функция вычисления $k^n \bmod m$ с оценкой сложности $O(\log_2 n)$:

```
unsigned long long PowMod(unsigned long long x, unsigned long long y,
                          unsigned long long n)
{
  if (!y) return 1;
  if (y & 1) return (x * PowMod((x * x) % n, y / 2, n)) % n;
  return PowMod((x * x) % n, y / 2, n);
}
```

Читаем входные значения *k*, *n*, *t*, вычисляем $m = 10^t$. Находим $x = k^n \pmod{10^t} = (k \bmod m)^n \pmod{10^t}$. Поскольку $k < 10^{19}$, то во избежание переполнения перед вызовом функции powmod следует найти остаток от деления *k* на *m*. Таким образом значение первого аргумента *k* функции powmod будет не более 10^9 и при вычислении $k * k$ не будет переполнения. Выводим результат с номером теста *cs*.

```
int cs = 1;
while(scanf("%llu %llu %llu",&k, &n, &t), k + n + t)
{
  m = 1; for(i = 0; i < t; i++) m *= 10;
  res = powmod(k % m,n,m);
  printf("Case #%d: %lld\n",cs++,res);
}
```

1515. Повторяющийся Иосиф

Функция last по первоначальному количеству людей *n* в круге возвращает номер уцелевшего согласно рекуррентному соотношению.

```
int last(int n)
{
  if (n == 1)   return 1;
  if (n%2 == 0) return 2 * last(n / 2) - 1;
          else return 2 * last((n - 1) / 2) + 1;
}
```

Переменная *r* содержит количество повторений процедуры казни (изначально $r = 0$). По заданному входному *n* ищем номер уцелевшего *k*. Если он не равен *n*, то повторяем в цикле процедуру казни.

```
scanf("%d",&tests);
for(i = 1; i <= tests; i++)
{
  scanf("%d",&n); r = 0;
  while ((k = last(n)) != n) r++, n = k;
  printf("Case %d: %d %d\n",i,r,n);
}
```

1516. Создание двоичного дерева поиска

Функция find располагает в дереве поиска высоты не более *h* числа от *a* до *b* и выводит их. Число $d = b - 2^{h-1} + 1$ располагаем в корне, числа от *a* до $d - 1$ располагаем в левом поддереве, числа от $d + 1$ до *b* в правом. Если правое поддерево будет заполнено не полностью, то $d < a$. В таком случае левое поддерево будет пустым, в корне расположим $d = a$, а числа от $a + 1$ до *b* вставим в правое поддерево. Обработку заканчиваем, как только правый конец интервала станет меньше левого ($a > b$).

```
void find(int a, int b, int h)
{
  int d = b - (1 << (h-1)) + 1;
  if (a > b) return;
  if (d < a) d = a;
  printf(" %d",d);
  find(a,d-1,h-1);
  find(d+1,b,h-1);
}
```

Последовательно читаем входные значения n и h, печатаем номер теста. Проверяем, существует ли искомое дерево: выводим 'Impossible.', если $n \geq 2^h$. Иначе выводим искомую последовательность вершин, вызвав функцию find(1, n, h).

```
while(scanf("%d %d",&n,&h), n+h)
{
  printf("Case %d:",cs++);
  if (n >= 1 << h) printf(" Impossible.");
  else find(1,n,h);
  printf("\n");
}
```

1517. Простое сложение

Поскольку выполняется обработка 32-битовых знаковых чисел, то для избежания переполнения при вычислениях используем тип long long.

```
long long p, q;
```

Функция g(p) вычисляет сумму значений функции f(n) для значений аргумента n от 1 до p.

```
long long g(long long p)
{
  long long t = p % 10;
  if (!p) return 0;
  return t * (1 + t) / 2 + p / 10 * 45 + g(p / 10);
}
```

Значение функции S(p, q) считаем как g(q) – q(p – 1).

```
long long s(long long p, long long q)
{
  return g(q) - g(p - 1);
}
```

Основной цикл программы. Для каждой пары чисел p и q выводим значение s(p, q).

```
while(scanf("%lld %lld",&p,&q), p + q >= 0)
  printf("%lld\n",s(p,q));
```

1518. Разбиение треугольника

Стиль i-го треугольника будем запоминать в паре (y[i], z[i]). В переменной *ptr* будем подсчитывать количество стилей.

```
double y[101],z[101];
int ptr;
```

Добавление треугольника со сторонами (x_1, y_1, z_1) в базу стилей. Если его стиль уже встречался ранее, то ничего не делаем. Иначе заносим пару $\left(\frac{y}{x_1}, \frac{z_1}{x_1}\right)$ в (y[*ptr*], z[*ptr*]).

```
int add(double x1, double y1, double z1)
{
  int i;
  y1 = y1 / x1; z1 = z1 / x1;
  for(i = 0; i < ptr;i++)
    if ((fabs(y1 - y[i]) < 0.000001) && (fabs(z1 - z[i]) < 0.000001))
      return 0;
  y[ptr] = y1; z[ptr] = z1; ptr++;
  return ptr;
}
```

Разбиваем треугольник со сторонами (a, b, c) на два со сторонами (a, m, c/2) и (b, m, c/2). Перед разбиением сортируем стороны в неубывающем порядке. Рекурсивно запускаем разбиение полученных треугольников.

```
void triangle(double a, double b, double c)
{
  double temp[3],m;
  temp[0] = a; temp[1] = b;  temp[2] = c;
  sort(temp,temp+3);
  if(!add(temp[0],temp[1],temp[2])) return;
  m = sqrt(2*temp[0]*temp[0]+2*temp[1]*temp[1]-temp[2]*temp[2])/2;
  triangle(temp[0],m,temp[2]/2);
  triangle(temp[1],m,temp[2]/2);
}
```

Основная часть программы.

```
scanf("%d",&n);
for(test = 1; test <= n; test++)
{
  scanf("%lf %lf %lf",&a,&b,&c);
  ptr = 0; triangle(a,b,c);
  printf("Triangle %d: %d\n",test,ptr);
}
```

1519. Коды Грея

Функция find(n, k) находит число, которое находится в k - ой позиции последовательности кодов Грея длины n.

```
int find(int n, int k)
{
  if (!n) return 0;
  int temp = 1 << (n-1);
  if (k < temp) return find(n-1,k);
  return temp + find(n-1, (1 << n) - 1 - k);
}
```

Основной цикл программы. Читаем количество тестов tests. Для каждого теста читаем входные данные n и k. Вычисляем и выводим значение find(n, k).

```
scanf("%d",&tests);
while(tests--)
{
  scanf("%d %d",&n,&k);
  res = find(n,k);
  printf("%d\n",res);
}
```

1520. Нечетные делители

Реализация функции g приведена ниже.

```
long long g(long long n)
{
  long long k = (n + 1) / 2;
  if (n == 0) return 0;
  return k * k + g(n / 2);
}
```

Основная часть программы. Читаем значение n и выводим $g(n)$.

```
while(scanf("%lld",&n) == 1)
  printf("%lld\n",g(n));
```