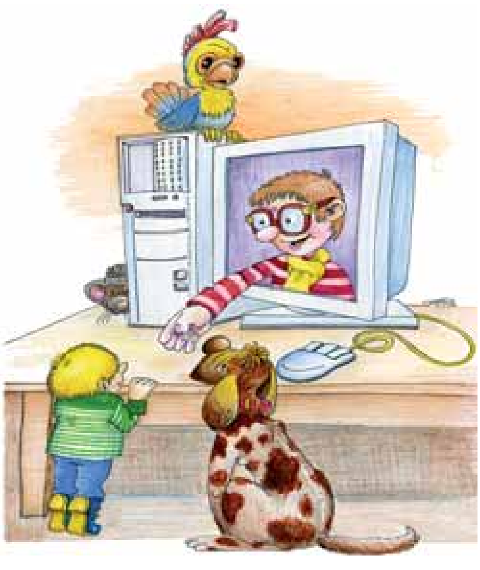
Медведев Михаил Геннадиевич
Кандидат физико-математических наук, доцент факультета кибернетики Киевского национального университета имени Тараса Шевченко.

Рекурсия и итерация

Рекурсия – одна из простейших концепций в алгоритмике. Суть рекурсии – сведение данной задачи к подобным, но более простым. Но не надо думать про рекурсию как про арифметику в математике. Несмотря на простоту идеи, иногда неясно, как практическую задачу свести к рекурсивной функции, какой у этой функции будет «физический смысл» и каким именно образом нужно осуществлять сведение. Часто альтернативным подходом рекурсии являются итерации, суть которых – повторение некоторой комбинации действий над хранимыми данными несколько раз. С первого взгляда итерации ещё примитивнее рекурсии, но, как это ни странно, на практике разрабатывать оптимальные итеративные алгоритмы сложнее, чем соответствующие им рекурсивные.

В этой статье рассмотрим ряд олимпиадных задач разной сложности, которые решаются при помощи итеративного и рекурсивного подходов.

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 операторами, 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 обычно он не сразу видит, 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 её обработки возникает её повторный вызов либо непосредственно, либо косвенно, путём 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).

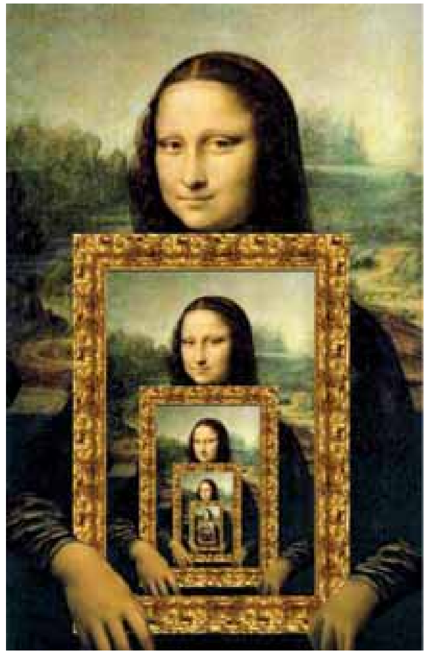
Теорема. *Произвольный алгоритм, реализованный в рекурсивной форме, может быть переписан в итерационной форме, и наоборот.*

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:

1) условие продолжения рекурсии (шаг рекурсии);

2) условие окончания рекурсии.

Рекурсию будем реализовывать посредством вызова функции самой себя. При этом в теле функции сначала следует проверять условие окончания 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:

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

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:

$$f(3)=3\cdot f(2)=3\cdot 2\cdot f(1)=$$

$= 3 \cdot 2 \cdot 1 \cdot f(0) = 3 \cdot 2 \cdot 1 \cdot 1 = 6.$

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

рекурсивная реализация	циклическая реализация
`int f(int n)`	`int f(int n)`
`{`	`{`
`  if(!n) return 1;`	`  int i, res = 1;`
`  return n * f(n - 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:

$$f(n) = 1 \cdot 2 \cdot 3 \cdot \ldots n.$$

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:

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

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

В итерационном варианте достаточно вычислить произведение $a \cdot a \cdot \ldots \cdot a$ (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:

$$\begin{cases} f(a,n) = a \cdot f(a^2, \lfloor n/2 \rfloor), \ n - \text{нечётное}, \\ f(a,n) = f(a^2, \lfloor n/2 \rfloor), \ n - \text{чётное}, \\ f(a,0) = 1. \end{cases}$$

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

```
  n / 2);
  return f(a * a, n / 2);
}
```

```
  {
    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 $f(n)$, определённой 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, делённого 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

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

циклическая реализация

```
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 $f(n)$, определённой 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:

$$n = a_1 a_2 \ldots a_{k-1} a_k 10\ldots0,$$
$$n - 1 = a_1 a_2 \ldots a_{k-1} a_k 01\ldots1,$$
$$n \,\&\, (n-1) = a_1 a_2 \ldots a_{k-1} a_k\, 000\ldots0.$$

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

рекурсивная реализация

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

циклическая реализация

```
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

$$C_n^k = \frac{n(n-1)\dots(n-k+1)}{1 \cdot 2 \cdot \dots \cdot k},$$

значение биномиального коэффициента можно вычислить при помощи цикла. При этом все операции деления будут целочисленными.

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

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

$$f(2 \cdot n) = f(n),$$

$$f(2 \cdot n + 1) = f(n) + f(n+1),$$

$$f(0) = 0, \; f(1) = 1.$$

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

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

для которой имеют место равенства:

$$g(2 \cdot n, i, j) = g(n, i + j, j),$$

$$g(2 \cdot n + 1, i, j) = g(n, i, i + j),$$

$$g(0, i, j) = i \cdot f(0) + j \cdot f(1) = j.$$

Используя приведённые соотношения, можно вычислить значение $f(n) = g(n, 1, 0)$ с временной оценкой $O(\log_2 n)$.

```
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:

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

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

$$A(m,n) = A\left(m-1, A(m,n-1)\right).$$

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:

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

$$A(1,n) = n+2,$$

$$A(2,n) = 2\cdot n+3,$$

$$A(3,n) = 2^{n+3}-3.$$

9. Отбор в разведку [ACM, 1999]. Из n солдат, выстроенных в шеренгу, требуется отобрать нескольких в разведку. Для совершения этого выполняется следующая операция: если солдат в шеренге больше чем 3, то удаляются все солдаты, стоящие на чётных позициях, или все солдаты, стоящие на нечётных позициях. Эта процедура повторяется до тех пор, пока в шеренге не останется 3 или менее солдат. Их и отсылают в разведку. Вычислить количество способов, которыми таким образом могут быть сформированы группы разведчиков ровно из трёх человек.

Вход. Количество солдат в шеренге $n\left(0<n\le 10^7\right)$.

Выход. Количество способов, которыми можно отобрать солдат в разведку описанным выше способом.

Пример входа	Пример выхода
10	2
4	0

Решение. Обозначим через $f(n)$ количество способов, которыми можно сформировать группы разведчиков из n человек в шеренге. Поскольку нас интересуют только группы по три разведчика, то $f(1)=0$, $f(2)=0$, $f(3)=1$. То есть из трёх человек можно сформировать только одну группу, из одного или двух – ни одной.

Если n – чётное, то применяя определённую в задаче операцию удаления солдат в шеренге, мы получим в качестве оставшихся либо $n/2$ солдат, стоящих на чётных позициях, либо $n/2$ солдат, стоящих на нечётных позициях. То есть $f(n) = 2 \cdot f(n/2)$ при чётном n.

Если n – нечётное, то после удаления останется либо $n/2$ солдат, стоявших на чётных позициях, либо $n/2+1$ солдат, стоявших на нечётных позициях. Общее количество способов при нечётном n равно

$$f(n)=f(n/2)+f(n/2+1).$$

Таким образом, получена рекуррентная формула для вычисления значения $f(n)$:

$$f(n)=2\cdot f(n/2),$$

если n чётное, и

$$f(n)=f(n/2)+f(n/2+1),$$

если n – нечётное.

$$f(1)=0,\ f(2)=0,\ f(3)=1.$$

Реализация функции f имеет вид:

```
int f(int n)
{
  if (n <= 2) return 0;
  if (n == 3) return 1;
  if (n % 2) return f(n / 2) + f(n / 2 + 1);
  return 2 * f(n / 2);
}
```

10. Большой модуль [Вальядолид, 374]. По заданным b, p, m вычислить значение выражения

$$b^p \bmod m.$$

Вход. Содержит несколько тестов. Числа b, p, m находятся в отдельных строках. Известно, что

$$0 \le b,\ p \le 2147483647,\ 1 \le m \le 46340.$$

Выход. Для каждого теста вывести в отдельной строке значение $b^p \bmod m$.

Пример входа	Пример выхода
3	13
18132	2
17	13195
17	
1765	
3	
2374859	
3029382	
36123	

Решение. Из ограничений на входные данные следует, что в процессе вычисления достаточно использовать тип данных int.

Возведение в степень b^p будем производить с логарифмической временной сложностью $O(\log_2 p)$, используя алгоритм, базирующийся на двоичном разложении показателя степени p:

$$b^p = \begin{cases} 1, \ p=0, \\ \left(b^{[p/2]}\right)^2, \ p - \text{чётное}, \\ b\left(b^{[p/2]}\right)^2, \ p - \text{нечётное}. \end{cases}$$

Пример. Для вычисления значения из первого теста $3^{18132}(\bmod 17)$ следует представить показатель степени в двоичной системе счисления:

$$18132_{10} = 100011011010100_2.$$

Далее

$$3^{18183}(\bmod 17) = 3^{16384} \cdot 3^{1024} \cdot 3^{512} \times$$
$$\times 3^{128} \cdot 3^{64} \cdot 3^{16} \cdot 3^4 (\bmod 17) = 13.$$

Для второго теста $17^{1765}(\bmod 3) =$

$$= (17 \bmod 3)^{1765} (\bmod 3) =$$
$$= 2^{1765} (\bmod 3) = 2.$$

Реализация. Функция **pow** вычисляет выражение $b^p \bmod m$ с временной оценкой сложности $O(\log_2 p)$.

```
#include <stdio.h>
int b,p,m,res;

int pow(int b, int p, int m)
{
 int  res = 1;
 while(p > 0)
 {
   if (p & 1) res = (res * b) % m;
   p >>= 1;
   b = (b * b) % m;
 }
 return res;
}

void main(void)
{
```

Прочитав входные значения b, p и m, следует воспользоваться формулой

$$b^p \bmod m = (b \bmod m)^p \bmod m.$$

При передаче параметров функции **pow** основание степени b должно быть не больше чем модуль m. Если этого не сделать, получим Time Limit. Отдельно следует обработать случай, когда $p = 0$:

$$b^0 \bmod m = 1.$$

```
 while (scanf("%d %d %d", &b, &p, &m) == 3)
 {
   b = b % m;
   if (!p) res = 1; else res = pow(b, p, m);
```

```
    printf("%d\n", res);
  }
}
```

11. Истина, спрятанная в рекуррентности [Вальядолид, 10547]. Функция f определена 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

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

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

$n \backslash i$	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. На вход подаётся не более 1001 тестов. Каждая строка содержит три целых числа:

k, n и t $(0<k, n<10^{19},\ 0<t<10)$.

Последний тест содержит $k=n=t=0$ и не обрабатывается.

Выход. Для каждого теста вместе с его номером в отдельной строке вывести значение x.

Пример входа	**Пример выхода**
1234 1234 4	Case #1: 736
2323 99999999999 8	Case #2: 39087387
4 99999 9	Case #3: 494777344
888 888 8	Case #4: 91255296
0 0 0	

Решение. Рассмотрим все n – цифровые числа в системе счисления 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 $(k-1)\cdot n$. Просуммировав значения $f(n,r)$ для r от 0 до $(k-1)\cdot n$, получим общее количество n-цифровых чисел в системе счисления с основанием k, то есть k^n. Таким образом $x=k^n(\bmod 10^t)$. Поскольку $t<10$, то при вычислении модулярной экспоненты достаточно использовать 64-битный целочисленный тип.

Пример. Для первого теста имеет место равенство:

$$1234^{1234}(\bmod 10^4)=736.$$

Реализация. При вычислении используем 64-битовый целый тип long long. Для простоты использования определим тип i64.

```
#include <stdio.h>

typedef long long i64;

i64 k, n, t, m, res;
int i;
```

Функция вычисления $x^y \bmod n$ с временной оценкой сложности $O(\log_2 y)$:

```
i64 powmod (i64 x, i64 y, i64 n)
{
  i64  res = 1;
  while(y > 0)
  {
    if (y & 1) res = (res * x) % n;
    y >>= 1;
    x = (x * x) % n;
  }
  return res;
}

void main(void)
{
```

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 $x \cdot x$ не будет переполнения. Выводим результат с номером теста *cs*.

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

12. Повторяющийся Иосиф [Вальядолид, 10774]. По кругу стоят n людей, занумерованных от 1 до n. Начиная отсчёт 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. Первая строка содержит количество тестов. Каждый тест в отдельной строке содержит одно число n $(0 < n \le 30000)$.

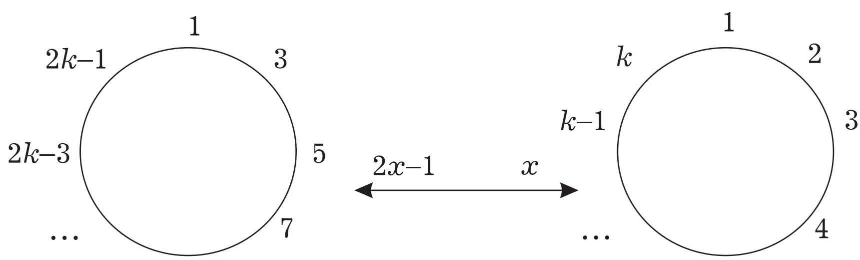
Выход. Для каждого теста вывести в отдельной строке его номер как указано в примере, количество повторений процедуры казни после первой итерации и номер выжившего в конце процедуры.

Пример входа	Пример выхода
2	`Case 1: 2  7`
13	`Case 2: 8 1023`
23403	

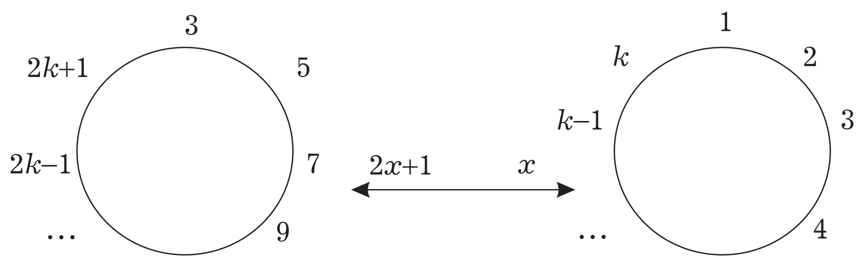
Решение. Пусть n – количество людей в круге. Обозначим через $f(n)$ номер последнего уцелевшего. Положим $f(1) = 1$.

Если $n = 2 \cdot k$ – чётное, то после прохода первого круга будут удалены люди с чётными номерами: 2, 4, ..., $2 \cdot k$. Останутся люди с нечётными номерами, а отсчёт продолжаем с номера 1. Это всё равно, что если бы у нас было k людей, а номер каждого удвоился и уменьшился на 1. То есть получим соотношение

$$f(2 \cdot k) = 2 \cdot f(k) - 1.$$



Если $n = 2 \cdot k + 1$ – нечётное, то после прохода первого круга будут удалены люди с чётными номерами 2, $4, \dots, 2 \cdot k$, а жертва с номером 1 уничтожается сразу же после жертвы с номером $2 \cdot k$. Остаётся k



людей с номерами $3, 5, 7, \ldots, 2 \cdot k + 1$. Это всё равно, что люди занумерованы от 1 до k, только номер каждого удвоился и увеличился на 1. Получаем соотношение:

$$f(2 \cdot k + 1) = 2 \cdot f(k) + 1.$$

Объединяя полученные соотношения, получим рекуррентность:

$$f(1) = 1,$$

$$f(2 \cdot k) = 2 \cdot f(k) - 1,\ k \geq 1,$$

$$f(2 \cdot k + 1) = 2 \cdot f(k) + 1,\ k \geq 1.$$

Теорема. Значение $f(n)$ получается путём циклического сдвига двоичного представления n влево на один бит. Например,

$$f(100) = f(1100100_2) = 1001001_2 = 73.$$

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, не равным n. При $n = 7$ выживет номер 7. После первой итерации проведены ещё 2 повторения процедуры казни.

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

```
#include <stdio.h>
int k, n, i, r, tests;

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;
}

void main(void)
{
  scanf("%d", &tests);
  for(i = 1; i <= tests; i++)
  {
```

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

13. Простое сложение [Вальядолид, 10994]. Определим рекурсивную функцию $f(n)$ следующим образом:

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

Определим функцию $S(p,\ q)$ следующим образом: $S(p,q)\ =\ \sum_{i=p}^{q} f(i)$.

В задаче необходимо вычислить значение $S(p,\ q)$ по заданным p и q.

Вход. Каждая строка содержит два неотрицательных 32-битовых знаковых числа p и q $(p \le q)$. Последняя строка содержит два отрицательных целых числа и не обрабатывается.

Выход. Для каждой пары p и q вывести значение $S(p,q)$.

Пример входа	Пример выхода
1 10	46
10 20	48
30 40	52
-1 -1	

Решение. Приведённая в условии функция $f(n)$ находит последнюю ненулевую цифру числа n. Обозначим $g(p)=\sum_{i=1}^{p} f(i)$. Тогда

$$S(p,q)=g(q)-q(p-1).$$

Для вычисления функции $g(p)$, суммы последних значащих цифр для чисел от 1 до p, разобьём числа от 1 до p на три множества (операция деления '/' является целочисленной).

1. Числа от $(p/10)\cdot 10+1$ до p.

2. Числа от 1 до $(p\,/\,10)\cdot 10$, не оканчивающиеся нулём.

3. Числа от 1 до $(p\,/\,10)\cdot 10$, оканчивающиеся нулём.

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 $p\,/10\cdot 45$, так как сумма всех цифр от 1 до 9 равна 45, а число полных десятков равно $p\,/10$. Требуемую сумму для третьего множества найдём рекурсивно: она равна $g(p\,/10)$.

Реализация. Поскольку выполняется обработка 32-битовых знаковых чисел, то при вычислениях используем тип **long long**.

```
#include <stdio.h>

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);
  return 0;
}
```

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);
}

void main(void)
{
```

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));
}
```

В статье рассмотрена реализация ряда олимпиадных задач, требующих написания рекурсивных функций или циклической реализации. Условия задач взяты со страницы университета Вальядолид **http://acm.uva.es//problemset,** посвящённой олимпиадному программированию.

Список литературы

1. *Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.* Алгоритмы: построение и анализ. – Москва, Санкт-Петербург, Киев, 2005, 1292 с.

2. *Винокуров Н.А., Ворожцов А.В.* Практика и теория программирования, книга 2. – М.: Физматкнига, 2008, 288 с.