

# Лекция 9

Рекурсия и итерация. Поиск.

Сергей Леонидович Бабичев

# Алгоритмика. Итерация и рекурсия. Поиск.

# Задача 1

**Задача 1.** На вход алгоритма подаётся натуральное число  $N$ . На выходе должно быть число  $M$  такое, что  $2^M \leq N < 2^{M+1}$ .

- Математически, задача заключается в вычислении целочисленного логарифма по основанию 2 от  $N$ .
- Здесь и далее будем реализовывать алгоритмы в виде функций.
- Имя функции должно отражать сущность алгоритма.
- Неплохим именем будет, например, `ilog2`.
- *Прототипом* этой функции будет `int ilog2(int n);`

## Задача 1: способ 1: прямолинейный

- Заведём переменные `down` и `up`, которые будут нам показывать нижнюю и верхнюю границы поиска при соответствующем `m`.
- Будем увеличивать переменную `m` каждый раз на 1, увеличивая вместе с ней переменные `down` и `up` в два раза.
- Как только окажется, что наше число `n` лежит в границах между `up` и `down`, алгоритм завершится.
- Начальные значения переменных обязательны.
- Минимальным значением `m` является 0, поэтому значением переменной `down` должно быть  $1 = 2^0$ , а переменной `up` — значение  $2 = 2^{0+1}$ .
- Алгоритм закончится тогда, когда `n` попадёт в нужные границы между `down` и `up` и выходным результатом станет `m`.
- Условие продолжения итераций противоположно условию завершения итераций.

## Задача 1: способ 1: код

```
int ilog2(int n) {
    int m = 0;
    int down = 1;
    int up = 2;
    while (!(down <= n && n < up)) {
        m++;
        down *= 2;
        up *= 2;
    }
    return m;
}
```

Три вечных вопроса:

1. Корректен ли алгоритм?
2. Какова его сложность?
3. Можно ли его улучшить?

## Задача 1: способ 1: доказательство корректности

- Мы использовали две *граничные переменные* `down` и `up`.
- Они *синхронно* изменяются вместе с переменной `m`.
- На каждой итерации цикла сохраняется истинность *высказываний, предикатов*  $down = 2^m$  и  $up = 2^{m+1}$ .
- Эти высказывания сохраняют истинность на всё время исполнения алгоритма и являются *инвариантами*.
- Наличие таких инвариантов и позволяет нам подтвердить корректность алгоритма.

## Задача 1: способ 1: считаем сложность алгоритма

- Нужно оценить, пропорционально какой функции изменяется количество операций и от какого аргумента.
- *Главным параметром* алгоритма является  $n$ .
- Общее число требуемых итераций совпадает с вычисленным значением  $m$ , которое есть  $\lceil \log_2 n \rceil$ .
- Сложность всего алгоритма будет составлять  $O(\log n)$ .
- Верхней границей количества операций будет нечто, пропорциональное  $\log n$ .

## Задача 1: способ 2: уменьшаем коэффициент

- Давайте совершенствовать наш алгоритм, чтобы уменьшить коэффициент амортизации  $C$ .
- Не производим ли мы лишние операции?
- Если  $n=10$ , то сначала вычисляются значения для операций сравнения  $n \geq 1$  и  $n < 2$ , на второй итерации —  $n \geq 2$  и  $n < 4$ , на третьей —  $n \geq 4$  и  $n < 8$ , и так далее.
- Заметим, что если оказалось, что истинно высказывание  $n < 4$ , то уже не окажется истинным высказывание  $n \geq 4$ .
- Используем это для сокращения количества операций в алгоритме.
- Переменная `down` нам не нужна совсем и от неё можно избавиться.



## Задача 1: способ 2: код

```
int ilog2(int n) {
    int m = 0;
    int up = 1;
    while (n >= up) {
        m++;
        up *= 2;
    }
    return m;
}
```

Корректность алгоритма доказать чуть сложнее, но мы уже проделали необходимые рассуждения.

## Задача 1: способ 2: сложность

- А что произошло со сложностью?
- С точки зрения пределов  $O$ -сложность не изменилась.
- Раньше она была  $O(\log N)$ .
- Она осталась  $O(\log N)$ .
- Очевидно, что алгоритм стал тратить меньше операций на проведение одной и той же работы.
- Количество операций уменьшилось пропорционально для любых значений  $N$ .
- Мы, не изменив сложность, уменьшили *коэффициент амортизации*.
- Для алгоритмов, одинаковых по сложности, меньшее значение этого коэффициента будет означать более быстрый алгоритм.

## Задача 1: способ 3: рекурсия

- Представим, что мы знаем значение функции  $\text{ilog2}(N)$  для какого-то  $N$ .
- Можем ли мы, не производя новых вычислений, быстро определить  $\text{ilog2}(2*N)$ ?
- Да, легко.  $\text{ilog2}(2*N) = \text{ilog2}(N)+1$ ;
- Можно увидеть, что имеет место следующее рекурсивное соотношение:

$$\text{ilog2}(N) = \begin{cases} 0, & \text{если } N \leq 1 \\ 1 + \text{ilog2}(N/2), & \text{если } N > 1 \end{cases}$$

Любые выражения такого рода можно легко запрограммировать:

```
int ilog2(int n) {
    if (n <= 1) return 0;
    else return 1 + ilog2(n / 2);
}
```

## Задача 1: способ 3: корректность и сложность

- Корректность этого алгоритма определяется математически: рекуррентой.
- Доказательств проводится по индукции.
- Оценим сложность этого алгоритма.
- Количество вызовов функции равно возвращаемому значению, то есть  $O(\log N)$ .
- Что происходит с памятью?
- Пусть  $i\log_2(100)$  вызывает  $i\log_2(50)$ , чтобы, получив результат, добавить единицу и вернуть сумму.
- В момент вызова  $i\log_2(50)$  в памяти находятся обе функции.
- Пока рекурсия не закончится (при условии  $n \leq 1$ ), в памяти окажется вся цепочка вызовов, и  $i\log_2(100)$ , и  $i\log_2(50)$ ,  $i\log_2(25)$ ,  $i\log_2(12)$ ,  $i\log_2(6)$ ,  $i\log_2(3)$ ,  $i\log_2(1)$ .
- Алгоритм стал хуже по памяти, её требуется тоже  $O(\log N)$ , когда до этого требовалось  $O(1)$ .
- Уменьшилась *описательная сложность*, так как программа стала проще.
- Рекурсивные программы часто имеют меньшую описательную сложность, чем эквивалентные нерекурсивные.

## Задача 2

**Задача 2.** На вход алгоритма подаётся число  $n > 0$ . Требуется вывести его десятичное представление. Разрешено пользоваться только функцией вывода одного символа `putchar`. Не разрешается пользоваться массивами и структурами.

- Назовём её `outnum`.
- Надо определить параметры и тип возвращаемого значения.
- Параметр — один, число число целого типа `n`.
- Так как цель функции — только в выводе цифр на экран, возвращаемое значение не требуется, тип функции — `void`.
- Прототип функции, следовательно, будет `void outnum(int n);`

## Задача 2: решение

- Чтобы определить последнюю цифру числа, достаточно взять остаток от деления этого числа на 10.
- Что с этим делать?
- При выдаче числа 123, нам нужно вывести вначале символ '1', а мы можем найти цифру '3'?
- Мы уже запоминали все символы получившегося числа в массиве, который обращали.
- Воспользуемся рекурсией, которая задержит вывод последней цифры, пока остальные экземпляры функции не выведут то, что перед ней.

```
void outnum(int n) {  
    if (n > 0) {  
        outnum(n/10);  
        putchar('0' + n%10);  
    }  
}
```

## Задача 2: ограничители

- В любой рекурсивной функции, вызовы функции самой себя не должны продолжаться бесконечно.
- Здесь мы поставили ограничитель — аргумент должен быть строго больше нуля.
- Как только аргумент оказывается равным нулю, рекурсия прерывается.
- Эта функция не может напечатать правильный результат, если мы вызовем её как `outnum(0)` ;.
- Не заменить ли  $n > 0$  на  $n \geq 0$ ?
- Попробуйте. Не обижайтесь на программу — она делает то, что приказано.

## Задача 3

**Задача 3.** Для чисел  $a, b > 0$  найти их НОД.

- Вспомним, что если два числа делятся на одно и то же число, то их разность делится на то же самое число.
- Отсюда если  $a > b$ , то  $\text{gcd}(a, b) = \text{gcd}(a, a-b)$ .
- Можно поступить, как Евклид, вычитая каждый раз из большего числа меньшее.
- Можно ускорить действия, заменив постоянные вычитания нахождение остатка.
- $\text{gcd}(a, b) = \text{gcd}(b, a \% b)$ .

$$\text{gcd}(a, b) = \begin{cases} a, & \text{если } b = 0 \\ b, & \text{если } a = 0 \\ \text{gcd}(b, a \bmod b), & \text{если } a > b \\ \text{gcd}(b \bmod a, a) & \text{иначе} \end{cases}$$



## Задача 3: код

```
unsigned gcd(unsigned a, unsigned b) {  
    if (b == 0) return a;  
    if (a == 0) return b;  
    if (a > b)  return gcd(b,  a%b);  
    else      return gcd(b%a, a);  
}
```

- Можно упростить этот код, разрешив одну лишнюю итерацию:

```
unsigned gcd(unsigned a, unsigned b) {  
    if (a == 0 || b == 0) return a+b;  
    return gcd(b, a % b);  
}
```

## Задача 4.

**Задача 4.** Для заданных положительных чисел  $x, y, p$  найти  $x^y \pmod{p}$ .

- Экая невидаль. Возведение в степень же просто многократное умножение.
- Умножим  $y-1$  раз и получим результат.
- Не будем забывать после каждого умножения брать модуль!
- А если  $y$  очень большое?
- Где такая задача может понадобиться?
- Поищем более короткое решение.

## Задача 4: размышления о коротком решении

- Нельзя ли разбить задачу на более мелкие задачи?
- Нельзя ли сократить количество умножений?

## Задача 4: размышления о коротком решении

- Нельзя ли разбить задачу на более мелкие задачи?
- Нельзя ли сократить количество умножений?
- Можно, если понять, что умножать можно не только на  $x$ .
- Можно число возводить в квадрат.
- Например,  $(((((x^2)^2)^2)^2)^2)^2 = x^{32}$ .
- Вместо 31 умножения получилось 5.
- Как это использовать?

## Задача 4: продолжение размышлений

- Позовём на помощь рекурсию.
- Если степень — чётная, попросим нашу функцию решить задачу в два раза проще, затем возведём в квадрат результат.

```
if (y % 2 == 0) {  
    unsigned long long temp = mypow(x, y/2, mod);  
    return temp * temp % mod;  
}
```

- Но с нечётным так не получится?

## Задача 4: продолжение размышлений

- Позовём на помощь рекурсию.
- Если степень — чётная, попросим нашу функцию решить задачу в два раза проще, затем возведём в квадрат результат.

```
if (y % 2 == 0) {  
    unsigned long long temp = mypow(x, y/2, mod);  
    return temp * temp % mod;  
}
```

- Но с нечётным так не получится?
- Давайте превратим его в чётное похожим приёмом.

```
if (y % 2 != 0) {  
    return mypow(x, y-1, mod) * x % mod;  
}
```

## Задача 4: решение

- Рекурсия должна как-то закончиться.
- Степень постоянно уменьшается и в конце концов достигнет нуля.
- Пусть это и будет точкой останова.
- В рекурсивных функциях обязательно случаи нерекурсивного решения рассматривать сначала — иначе рекурсия никогда не завершится.

```
typedef unsigned long long T;
```

```
T mypow(T x, T y, T mod) {  
    if (y == 0) return 1;  
    if (y % 2 == 0) {  
        T temp = mypow(x, y/2, mod);  
        return temp * temp % mod;  
    } else {  
        return mypow(x, y-1, mod) * x % mod;  
    }  
}
```

## Задача 4: сложность

- Очевидно, что показатель степени — дискретно убывающая величина, алгоритм завершится.
- Как оценить сложность алгоритма?
- Давайте попробуем возвести число, например, в 23-ю степень и проследим, как меняется степень в рекурсивных вызовах.
- $23 \rightarrow 22 \rightarrow 11 \rightarrow 10 \rightarrow 5 \rightarrow 4 \rightarrow 2 \rightarrow 1 \rightarrow 0$



## Задача 4: сложность

- Очевидно, что показатель степени — дискретно убывающая величина, алгоритм завершится.
- Как оценить сложность алгоритма?
- Давайте попробуем возвести число, например, в 23-ю степень и проследим, как меняется степень в рекурсивных вызовах.
- $23 \rightarrow 22 \rightarrow 11 \rightarrow 10 \rightarrow 5 \rightarrow 4 \rightarrow 2 \rightarrow 1 \rightarrow 0$
- А что, если эти числа записать в двоичном виде?
- $10111 \rightarrow 10110 \rightarrow 1011 \rightarrow 1010 \rightarrow 101 \rightarrow 100 \rightarrow 10 \rightarrow 1 \rightarrow 0$
- При нечётном показателе степени мы уходим в рекурсию, обнулив последнюю единицу.
- При чётном показателе степени последняя цифра равна нулю и мы её уничтожаем.
- Худший случай — все единицы в двоичной записи числа, число вида  $N = 2^n - 1$ .
- Тогда потребуется  $2n = O(\log N)$  операций.

## Задача 5

**Задача 5.** Имеется массив  $n$  целых чисел  $arr$ . Требуется по заданному числу  $x$  определить, существует ли число с таким значением в массиве, и если существует, то под каким индексом.

- Это — задача поиска значения в неупорядоченном массиве.
- Назовём искомую функцию `int dummysearch(int *arr, int n, int x);`

Индекс	0	1	2	3	4	5	6	7	8	9
Элемент	132	612	232	890	161	222	123	861	120	330

`dummysearch(arr, 10, 222) = 5`

`dummysearch(arr, 10, 999) = 10` (элемент за границей поиска).

## Задача 5: последовательный поиск

```
int dummysearch(int *arr, int N, int key) {
    for (int i = 0; i < N; i++) {
        if (arr[i] == key) {
            return i;
        }
    }
    return N;
}
```

- Число операций сравнения в худшем случае  $2N$ .

## Задача 5: последовательный поиск

- Небольшая подготовка: добавим искомый элемент к концу массива;
- Внимание! Так можно сделать не всегда!

Индекс	0	1	2	3	4	5	6	7	8	9	10
Ключ	132	612	232	890	161	222	123	861	120	330	999

Результаты не изменились.

`cleversearch(arr, 10, 222) = 5`

`cleversearch(arr, 10, 999) = 10` (элемент за границей поиска).

## Задача 5: последовательный поиск

```
int cleversearch(int *arr, int N, int key) {
    arr[n] = key;
    int i;
    for (i = 0; arr[i] != key; i++)
        ;
    return i;
}
```

- Число операций сравнения  $N$  в худшем случае.
- Поиск ускорен в два раза!
- Для получения хороших результатов надо подготовиться!

## Задача 6

**Задача 6.** Имеется массив  $n$  целых чисел `arr`, которые расположены в порядке неубывания. Требуется по заданному числу  $x$  определить, существует ли число с таким значением в массиве, и если существует, то под каким индексом.

- Это — задача поиска значения в *упорядоченном* массиве.
- Назовём искомую функцию `int search(int *arr, int n, int x);`

## Задача 6: размышления о решении

- Лёгкое решение у нас уже есть: `dummysearch`.
- Сложность такого алгоритма —  $O(n)$ .
- Чтобы понять, что такого числа нет, нужно просмотреть весь массив.

## Задача 6: размышления о решении

- Решение `dummysearch` годится и для упорядоченного массива, и для произвольного.
- Как использовать факт упорядоченности?
- Упорядоченность означает, что если, скажем,  $x > arr[20]$ , то искать  $x$  для элементов с индексами, меньшими 20, не нужно!
- А если  $x < arr[20]$ , то не нужно искать  $x$  среди элементов в индексами, большими 20.
- Так что если мы разобьём массив на две половинки и посмотрим на средний элемент, то поиск сузится в два раза.



## Задача 6: размышления о решении

- Сегодня у нас день рекурсии.
- Чтобы сделать алгоритм рекурсивным, немного изменим функцию.
- Нам нужны теперь границы участка, где мы ищем.
- Тогда делаем попытку сравнить  $x$  со средним элементом.
- При равенстве — возвращаем индекс.
- Иначе уменьшаем задачу, передавая оставшуюся часть массива рекурсивно.

## Задача 6: код

```
int binsearch(int const *arr, int l, int r, int x) {
    if (l >= r) return arr[l] == x? l : -1;
    int mid = (l+r)/2;
    if (arr[mid] == x) return mid;
    if (arr[mid] < x) {
        return binsearch(arr, l, mid-1, x);
    } else {
        return binsearch(arr, mid+1, r, x);
    }
}
```

## Задача 6: решение итерациями

- Рекурсивные алгоритмы часто писать проще, чем итеративные.
- В данном случае рекурсия помогла нам найти идею.
- Для каждого рекурсивного алгоритма существует эквивалентный итеративный.
- Иногда итеративные алгоритмы проще, чем рекурсивные, иногда — сложнее.

```
int binsearch(int const *arr, int l, int r, int x) {
    while (l < r) {
        int mid = (l + r)/2;
        if (a[mid] == x) return mid;
        if (a[mid] < x) {
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return a[l] == val? l : -1;
}
```