
# Информатика

## Базовые алгоритмы

рекуррентные вычисления, рекурсия, итерация

© Марченко Антон Александрович 2017 г.
Абрамский Михаил Михайлович

# Рекуррентные вычисления

- **Рекуррентные формулы:** текущий элемент выражается через несколько предыдущих
- Факториал

$$n! = \begin{cases} 1, & n = 0; \\ (n-1)! \cdot n, & n \geq 1. \end{cases}$$

- Числа Фибоначчи

$$F_n = \begin{cases} 0, & n = 0; \\ 1, & n = 1; \\ F_{n-1} + F_{n-2}, & n \geq 2. \end{cases}$$

# Рекуррентные вычисления

Два способа рекуррентных вычислений:

- **Итерация**
  – обход в циклах
- **Рекурсия**
  – рекурсивные вызовы метода

# Итерация

- Многократное повторение действий в цикле

```
static ulong Fact(uint n)
{
    // n=0
    ulong fact = 1;
    for (uint i = 1; i <= n; i++)
        // n!=(n-1)!*n
        fact *= i;
    return fact;
}
```

# Рекурсия

- Прямое или косвенное обращение алгоритма к самому себе с другими значениями входных параметров
- Описание рекурсивного алгоритма – рекуррентная формула

# Примеры рекурсии из жизни

- «чтобы понять рекурсию, нужно сначала понять рекурсию»
- Рекурсия, которую мы все видели – герб России
  - скипетр венчается уменьшенной копией герба

# Рекурсивные алгоритмы

- Обращаются **сами к себе**
  *рекурсивный вызов*
- Сводят задачу к такой же, но **меньшего размера**
  *сведение к подзадаче*
- Содержат условия **завершения**

# Рекурсивные методы

- ***Вызывают сами себя*** с другими значениями параметров
- ***Рекурсивные вызовы*** продолжаются до выполнения условия их прекращения
- После остановки вызовов начинаются ***рекурсивные возвраты***
- ***Метод***, начавший цепочку рекурсивных вызовов ***ждёт завершения вызванных им методов*** и т.д.

# Плюсы и минусы рекурсии

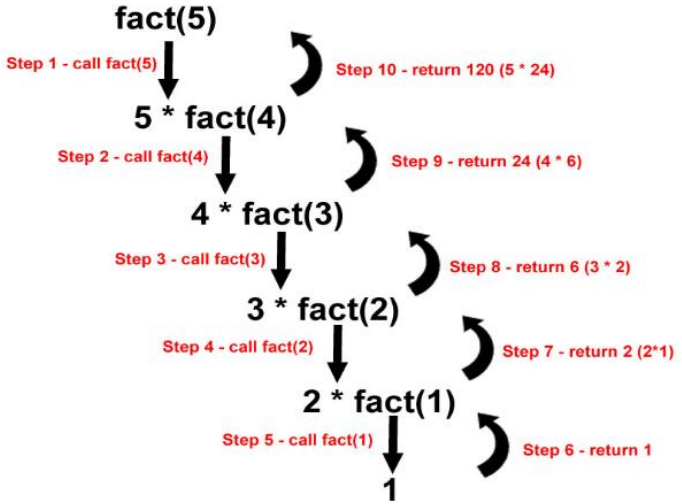
- ***Плюсы:***
  - Быстрее реализовать
  - Более простое, элегантное описание
  - Корректность можно доказать по индукции
- ***Минусы:***
  - Медленнее итерации
  - Использует больше памяти
  - Может привести к переполнению стека

# Рекурсивные методы

- Факториал

$$n! = \begin{cases} 1, & n = 0; \\ (n-1)! \cdot n, & n \geq 1. \end{cases}$$

```
static ulong Fact(uint n)
{
    // Условие завершения.
    if (n == 0) return 1;
    // Сведение к подзадаче и
    // рекурсивный вызов.
    return n*Fact(n - 1);
}
```

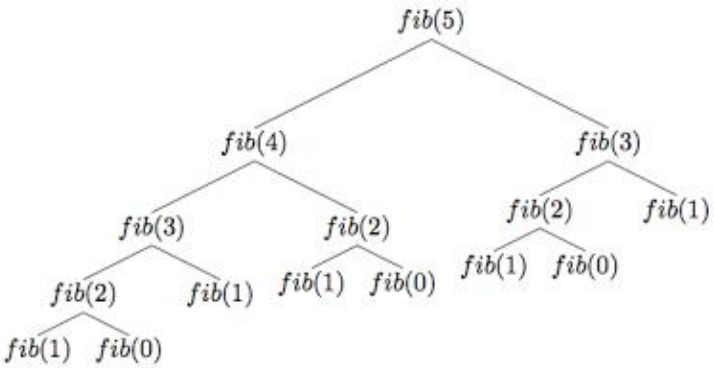


# Рекурсивные методы

- Числа Фибоначчи

$$F_n = \begin{cases} 0, & n = 0; \\ 1, & n = 1; \\ F_{n-1} + F_{n-2}, & n \geq 2. \end{cases}$$

```
static ulong Fib(uint n)
{
    // Условие завершения.
    if (n < 2) return n;
    // Сведение к подзадаче и
    // рекурсивный вызов.
    return Fib(n - 1) + Fib(n - 2);
}
```



# Рекурсия и сложность

- Можно оценивать сложность решая рекуррентные уравнения
- Сложность классического рекурсивного алгоритма вычисления чисел Фибоначчи

$$F_n \sim \frac{\varphi^n}{\sqrt{5}}, \qquad \text{где } \varphi = \frac{1+\sqrt{5}}{2} - \text{золотое сечение}$$

# Оценка сложности рекурсии

- Для некоторых рекуррентных соотношений можно применять основную теорему (Master theorem)
- Алгоритмы «разделяй и властвуй»:
  *для решения задачи* ***размера n***
  *делают* ***a рекурсивных вызовов***
  *для* ***задач размера n/b***
  *и тратят время* $\boldsymbol{O(n^d)}$ ***на подготовку*** *вызовов и сбор ответов.*

# Master theorem

- **Основная теорема о рекуррентных соотношениях**

$$T(n) = aT\left(\left\lceil\frac{n}{b}\right\rceil\right) + O\left(n^d\right)$$

где $a > 0, b > 1, d \geq 0$. Тогда

$$T(n) = \begin{cases} O\left(n^d\right), & d > \log_b a \\ O(n^d \log n), & d = \log_b a \\ O\left(n^{\log_b a}\right), & d < \log_b a \end{cases}$$

# Пример оценки сложности

- Двоичный поиск
  - Случай 2 основной теоремы:

$$c = \log_b a, \text{где } a = 1, b = 2, d = 0$$

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

**Сложность: $O(\log n)$**

# Рекурсия и итерация

- **Рекурсивную программу всегда можно преобразовать в итеративную и наоборот!**
- Итеративный алгоритм преобразовать в рекурсию – легко
  - Переменные цикла делаем параметрами метода
  - Итерации заменяем на рекурсивные вызовы
  - Условие цикла – условие завершения работы метода

# Хвостовая рекурсия

- Рекурсивный вызов – последний оператор
- Некоторые компиляторы могут оптимизировать хвостовую рекурсию, преобразуя её в итерацию
  - В .NET компилятор C# не может, но компиляторы F# и Nemerle – могут

# Хвостовая рекурсия

Рекурсивный вызов – последний оператор

```
// Нерекурсивный метод,
// вызывающий вспомогательный.
static ulong Fact(uint n)
{
    return FactTimes(n, 1);
}
```

```
// Рекурсивный вспомогательный метод
// с хвостовой рекурсией.
// Рекурсивный вызов – последний оператор.
static ulong FactTimes(uint n, ulong acc)
{
    // Условие завершения.
    if (n == 0) return acc;
    // Сведение к подзадаче и
    // рекурсивный вызов.
    return FactTimes(n - 1, acc*n);
}
```

# Оптимизация хвостовой рекурсии

```
// Метод после оптимизации хвостовой рекурсии.
// Автоматическая коррекция кода в Visual Studio.
static ulong FactTimes(uint n, ulong acc)
{
    while (true)
    {
        // Условие завершения.
        if (n == 0) return acc;
        // Сведение к подзадаче и рекурсивный вызов.
        var n1 = n;
        n = n - 1;
        acc = acc*n1;
    }
}
```

# Хвостовая рекурсия чисел Фибоначчи

```
// Нерекурсивный метод.
static ulong Fib(uint n)
{
    // Вызов рекурсивного метода.
    return FibRec(0, 1, n);
}

// Вспомогательный рекурсивный метод
// с хвостовой рекурсией.
static ulong FibRec(ulong cur, ulong prev, uint n)
{
    // Условие завершения.
    if (n == 0) return cur;
    // Сведение к подзадаче и
    // рекурсивный вызов.
    return FibRec(cur + prev, cur, n - 1);
}
```

# Оптимизация

```
// После оптимизации хвостовой рекурсии.
// Автоматическая коррекция кода ReSharper
static ulong FibRec(ulong cur, ulong prev, uint n)
{
    while (true)
    {
        // Условие завершения.
        if (n == 0) return cur;
        // Сведение к подзадаче и
        // рекурсивный вызов.
        var cur1 = cur;
        cur = cur + prev;
        prev = cur1;
        n = n - 1;
    }
}
```

# Использование стека

- Используя динамическую память можно промоделировать рекурсию
- Завести стек, складывать в него значения вместо осуществления вызовов
- Мы будем так делать при работе с графами
  - Рекурсия не подойдёт из-за размеров задачи (переполнится стек)

# Алгоритмы на массивах

**Рассмотрим следующие алгоритмы:**

- Разворот
- Поиск
  - обычный поиск
  - бинарный поиск (рекурсивный и итеративный) в упорядоченном массиве
- Сортировка
  - Быстрая (рекурсивная)

# Разворот массива

```
// Разворот с помощью класса Array.
Array.Reverse(array);

// Разворот вручную.
for(int i=0; i<array.Length/2; i++)
{
    int tmp = array[i];
    array[i] = array[array.Length - 1 - i];
    array[array.Length - 1 - i] = tmp;
}
```

# Рекурсивный разворот

```
static void Reverse(int[] array, int index=0)
{
    if (index>=array.Length/2) return;
    int tmp = array[index];
    array[index] = array[array.Length-1-index];
    array[array.Length-1-index] = tmp;
    Reverse(array,index+1);
}
```

# Поиск индекса элемента

**Идея:**

Перебираем элементы, пока не встретим искомый

- Если находим элемент, возвращаем его индекс

Если после перебора элемент не найден – возвращаем -1

# Поиск в массиве

```
// Поиск индекса элемента
// с помощью класса Array.
int idx = Array.IndexOf(array, 3);

// Поиск индекса элемента вручную.
// Начинаем с 0;
idx = 0;
for (int i = 0; i < array.Length; i++)
    if (array[i] == item)
    {
        // Если находим, фиксируем индекс.
        idx = i;
        // Прекращаем поиск.
        break;
    }
```

# Рекурсивный вариант

$$Find_{i..n} = i \;||\; Find_{(i+1)..n}$$

```
static int Find(int[] array,int item)
{
    return Find(array, item, 0);
}

static int Find(int[] array, int item, int idx)
{
    if (idx >= array.Length) return -1;
    if (array[idx] == item) return idx;
    return Find(array, item, idx + 1);
}
```

```
static int Find(int[] array,
                int item, int idx)
{
    while (idx<array.Length)
    {
        if (array[idx] == item)
            return idx;
        idx = idx + 1;
    }
    return -1;
}
```

# Бинарный поиск

**Идея:**

Сверяем элемент в середине массива с искомым

- Если нашли – возвращаем индекс середины
- Если средний элемент больше – **ищем в левой части**
- Если средний элемент меньше – **ищем в правой части**

# Бинарный поиск (рекурсия)

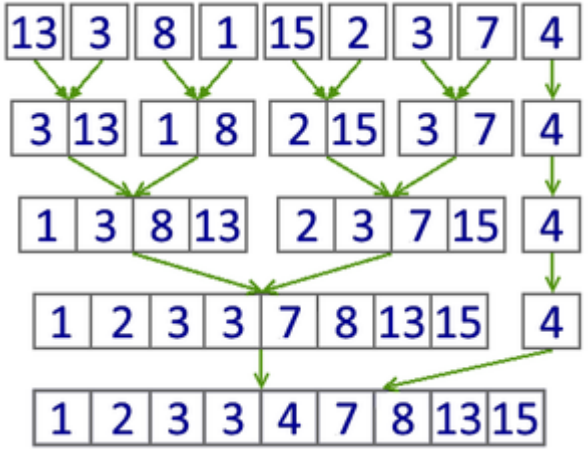
```
// Нерекурсивный метод для удобного вызова.
static int BinarySearch(int[]array, int value)
{
    return BS(array, 0, array.Length, value);
}
// Рекурсивный бинарный поиск.
static int BS(int[]array, int l, int r, int value)
{
    if (r < l) return -1;
    int mid = l+(r - l)/2;
    if (array[mid] == value) return mid;
    return array[mid] > value
        ? BS(array, l, mid-1, value)
        : BS(array, mid+1, r, value);
}
```

# Бинарный поиск (итерация)

```
static int BinarySearch(int[]array, int value)
{
    int l = 0, r = array.Length;
    while (r>l)
    {
        int mid = l + (r - l)/2;
        if (array[mid] == value) return mid;
        if (array[mid] > value)
        {
            r = mid - 1;
            continue;
        }
        l = mid + 1;
    }
    return -1;
}
```

# Сортировка слиянием

- Разделяем пополам
- Сортируем части
- Сливаем части
  - доп. массив для слияния



# Оценка сложности

- Случай 2 основной теоремы:

$$c = \log_b a, \text{где } a = 2, b = 2, d = 0$$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

**Сложность: $O(n \log n)$**

# Раскрытие рекуррентности

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$T(n) = 4T\left(\frac{n}{4}\right) + 2O(n)$$

$$\dots$$

$$T(n) = nT(1) + \log n\, O(n)$$

# Сортировка слиянием

```
static void MergeSort(int[]numbers)
 {
     // Буфер для слияния
     int[] temp = new int[numbers.Length];
     MergeSort(0,numbers.Length-1);

     // Вложенный метод
     void MergeSort(int left, int right)
     {
         if (left >= right)
             return;
         int mid = left+(right-left) / 2;
         MergeSort(left, mid);
         MergeSort(mid+1, right);
         Merge(left, mid, right);
     }
     // Вложенный метод
     // Использует numbers и temp(см. замыкания)
     void Merge(int left,int mid, int right)
     {
         int l = left, r = mid+1;
         int index = 0;
         while (l<=mid && r<=right)
             if (numbers[l] < numbers[r])
                 temp[index++] = numbers[l++];
             else
                 temp[index++] = numbers[r++];
         while (l<=mid)
             temp[index++] = numbers[l++];
         while (r<=right)
             temp[index++] = numbers[r++];
         for (int i = left; i <= right; i++)
             numbers[i] = temp[i-left];
     }
}
```

# Быстрая сортировка

Идея:

1. Выбираем опорный элемент
2. Разделяем массив на 2 части:
   со значениями больше и меньше опорного
3. Сортируем каждую из частей
4. Объединяем части

# Рекурсивная быстрая сортировка

```
static void QuickSort(int[]array)
{
    quickSort(array, 0, array.Length - 1);
}
static void quickSort(int[] a, int l, int r)
{
    // Выбираем опорный элемент.
    int pivot = a[l + (r - l) / 2];
    // Разделяем массив на > и < опорного.
    int idx = Partition(a, pivot, l, r);
    if (idx < r)
        quickSort(a, idx, r);
    if (l < idx-1)
        quickSort(a, l, idx-1);
}
```

# Разделение массива

```
private static int Partition(int[] a, int x, int i, int j)
{
    while (i <= j)
    {
        while (a[i] < x) i++;
        while (a[j] > x) j--;
        if (i <= j)
        {
            int temp = a[i];
            a[i] = a[j];
            a[j] = temp;
            i++;
            j--;
        }
    }
    return i;
}
```

# Переборные алгоритмы

- Перебор паролей
- Ханойская башня

# Перебор паролей

- Сводим к подзадачам
- Перебираем позиции
- Подставляем цифры
- Обрабатываем пароли

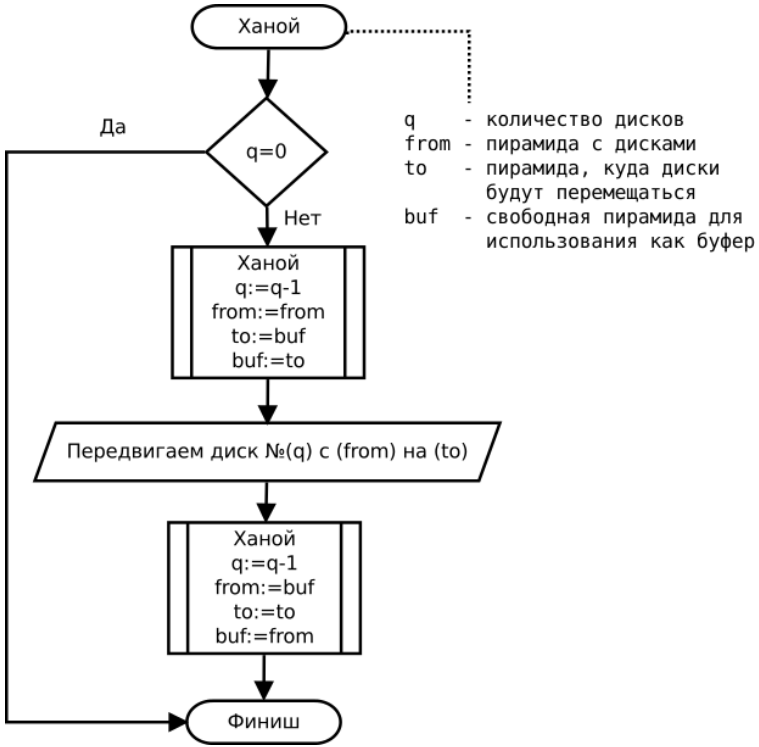
```
static void BruteForce(int[]digits, int index)
{
    if (index == digits.Length)
    {
        Process();
        return;
    }

    for (int i = 0; i < 10; i++)
    {
        digits[index] = i;
        BruteForce(digits, index + 1);
    }

    void Process()
    {
        foreach (int t in digits)
            Console.Write(t);
        Console.WriteLine();
    }
}
```

# Ханойская башня

- Три стержня
- На левом – кольца
- Перенести на правый
- Нельзя класть большие кольца на меньшие

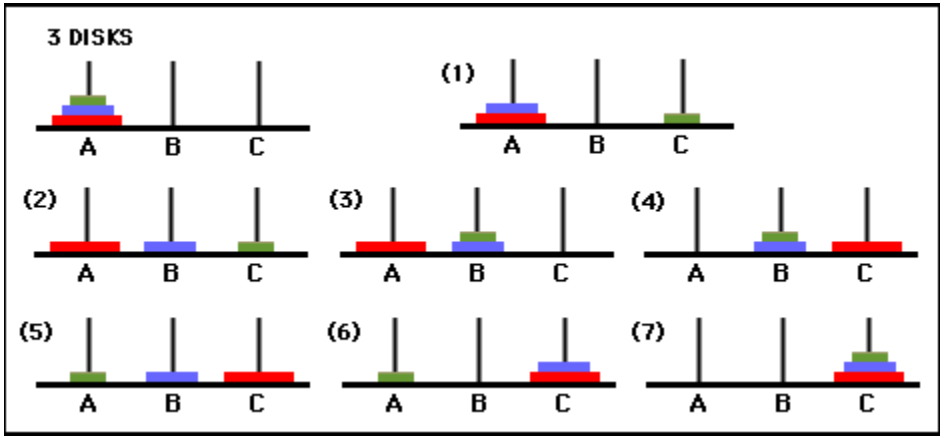
# Рекурсивный алгоритм

```
static void SolveHanoiTower
(int q, char from, char to, char buf)
{
    if (q <= 0) return;
    SolveHanoiTower(q - 1, from, buf, to);
    Console.WriteLine("Move disk from "
                      + startPeg + " to "
                      + endPeg);
    SolveHanoiTower(q - 1, buf, to, from);
}
```



# Работа программы

```
static void Main(string[] args)
 {

     char from = 'A';
     char to   = 'C';
     char buf  = 'B';
     int  q = 3;
     SolveHanoiTower(q, from, to, buf);
 }
```




Move disk from A to C
Move disk from A to B
Move disk from C to B
Move disk from A to C
Move disk from B to A
Move disk from B to C
Move disk from A to C

# Сложность

- Минимальное число шагов для решения задачи Ханойская Башня с $n$ дисками – $2^n - 1$
- Доказывается по индукции


Вопросы?

*e-mail:* marchenko@it.kfu.ru