Информатика

Базовые алгоритмы

Побитовые операции

Операции с целочисленными типами данных

Число – набор битов

$$240_{10} = 11110000_2$$

Оператор	Значение
&	Поразрядное И
|	Поразрядное ИЛИ
^	Поразрядное исключающее ИЛИ
<<	Сдвиг влево
>>	Сдвиг вправо
~	Поразрядное дополнение

Побитовые логические операции

& (И) , **|** (ИЛИ), **~** (НЕ), **^**(XOR)

Пример:

```
byte a = 29;
byte b = 11;
byte c = (byte)(a & b);
byte d = (byte)(a | b);
byte e = (byte)(a ^ b);
byte f = (byte)~a;
sbyte g = (sbyte)~a;
```

Чему равны c, d, e, f, g?

Побитовые логические операции

29 = 0001 1101

```
0001 1101
&&&& &&&&
0000 1011
==== ====
0000 1001 = 9
```

```
0001 1101
^^^^ ^^^^
0000 1011
==== ====
0001 0110 = 22
```

11 = 0000 1011

```
0001 1101
|||| ||||
0000 1011
==== ====
0001 1111 = 31
```

```
~~~~ ~~~~
0001 1101
==== ====
1110 0010 = …
```

В byte 226
А что в sbyte?

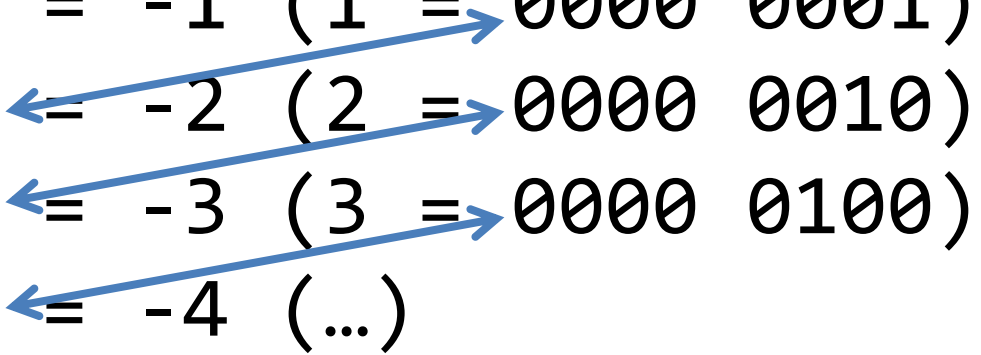
Обратный и дополнительный код

- Как хранятся отрицательные числа?

Десятичный	Прямой	Обратный	Дополнительный
127	0111 1111	0111 1111	0111 1111
2	0000 0010	0000 0010	0000 0010
1	0000 0001	0000 0000	0000 0001
0	0000 0000	0000 0000	0000 0000
-0	1000 0000	1111 1111	---- ----
-1	1000 0001	1111 1110	1111 1111
-2	1000 0010	1111 1101	1111 1110
-127	1111 1111	1000 0000	1111 1101
-128	---- ----	---- ----	1000 0000

Дополнительный код

```
0000 0000 –
        1
1111 1111 = -1 (1 = 0000 0001)
1111 1110 = -2 (2 = 0000 0010)
1111 1101 = -3 (3 = 0000 0100)
1111 1100 = -4 (…)
```



Дополнительный код

- Инверсия знака – инвертируем биты и прибавляем 1
- Для нашего примера:
  - `0001 1101`
  - инвертируем: `1110 0010` = -30
  - прибавляем 1: `1110 0011` = -29
- Таким образом, `~n=-n+1`

Сдвиговые операции

`>>` (сдвиг вправо) `<<` (сдвиг влево)

- Операции низкого уровня
- Сдвигают набор битов влево или вправо

```
0001 0011 << 1 = 0010 0110
```

- Вышедшие «за границы» биты теряются
- На освободившиеся места ставятся нули

Сдвиговая арифметика

- **Умножение на 2:** `x <<= 1;`

```
0001 0011 = 19 |<<1| 0010 0110 = 38
```

- **Целочисленное деление на 2:** `x >>= 1;`

```
0001 0011 = 19 |>>1| 0000 1001 = 9
```

- **Вычисление степеней 2:** `int x = 1 << p;`

Особенности сдвигов

- *Сдвиг вправо* >> не изменяет старший бит отрицательное число *сохраняет знак*!

```
1001 0011  |>>1|  10001001
```

- Сдвиг влево << сдвигает старший бит!
- ***Чтобы не зависеть от знака при сдвигах – используйте беззнаковые целые типы***

Сравнение знаков чисел

Если у чисел разные знаки, то после XOR старший бит будет 1 и результат будет <0

```
// Проверка, имеют ли два числа разные знаки.
static bool DifferentSigns(int x, int y)
{
    return (x ^ y) < 0;
}
```

Проверка на степень 2

Если число x – степень двойки, то у него не будет общих битов с x-1

```
// Проверка, является ли число степенью 2.
static bool isPower2(int x)
{
    return x != 0 && (x & (x - 1)) == 0;
}
```

Определение знака

Если число положительное – сдвиг вправо на 31 даёт 0, иначе -1

```
// Определение знака.
static int GetSign(int x)
{
    if (x == 0) return 0;
    int mask = 1;
    return mask | x >> 31;
}
```

Модуль числа

Если x>=0, то mask=0 и результат равен x
Иначе, mask=-1, работаем с x в дополнительном коде

```
// Модуль числа.
static int XorAbs(int x)
{
    int mask = x >> 31;
    return (x + mask) ^ mask;
}
```

Циклические сдвиги

```
// Циклический сдвиг битов влево
static byte RotateLeft(byte x, int shift)
{
    shift &= 7;
    return (byte)((x >> (8 - shift)) | (x << shift));
}

// Циклический сдвиг битов вправо
static byte RotateRight(byte x, int shift)
{
    shift &= 7;
    return (byte)((x << (8 - shift)) | (x >> shift));
}
```

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 *= n;
    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

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

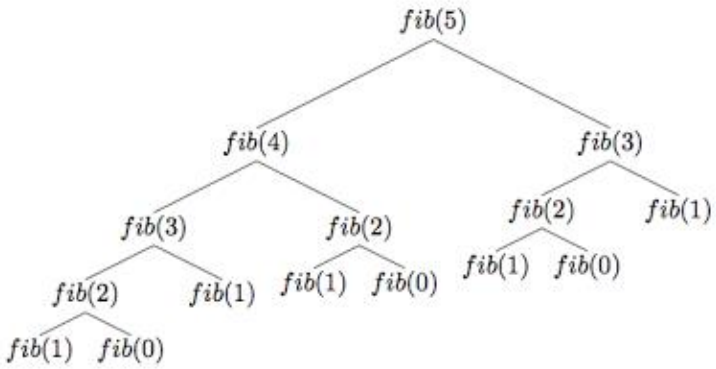
```
static ulong Fact(uint n)
{
    // Условие завершения.
    if (n == 0) return 1;
    // Сведение к подзадаче и
    // рекурсивный вызов.
    return n*Fact(n - 1);
}
```

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

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

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**
- Частный вид рекурсии – *хвостовую*
  компиляторы .NET могут оптимизировать
  (C# не может, но 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 в Visual Studio
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;
    }
}
```

Алгоритмы на массивах

Рассмотрим следующие алгоритмы:

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

Ввод/вывод (повторение)

```
using (StreamReader sr = new StreamReader("input.txt")
    string str = sr.ReadLine();

// Разбиваем по пробелам, убираем пустые строки.
string[] data = str?.Split(new[]{' '},
        StringSplitOptions.RemoveEmptyEntries);

// null заменяем на пустой массив.
data = data??new string[0];

// Преобразуем все элементы к int.
int[] array = Array.ConvertAll(data, Convert.ToInt32);

using (StreamWriter sw = new StreamWriter("output.txt"))
    sw.WriteLine(string.Join(" ", array));
```

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;
}
```

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] == 3)
    {
        // Если находим, фиксируем индекс.
        idx = i;
        // Прекращаем поиск.
        break;
    }
```

Рекурсивный вариант

Сверяем элемент.
Нашли – возвращаем индекс
Нет – ищем в оставшейся части

```
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);
}
```

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;
}
```

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;
}
```

Вопросы?

e-mail: marchenko@it.kfu.ru