Информатика

Построение алгоритмов

Построение алгоритмов

- **Эффективность алгоритма** (сложность)
  - время и память
- **Эффективность построения** (структура/проектирование)
  - Top-Down + структурность + подпрограммы
- **Эффективность написания** (красота/надёжность)
  - Code Conventions + Best Practices

Эффективность алгоритма

$T(n)$	Name	Problems
$O(1)$	constant	
$O(logn)$	logarithmic	
$O(n)$	linear	easy-solved
$O(nlogn)$	linear-logarithmic	
$O(n^2)$	quadratic	
$O(n^3)$	cubic	
$O(2^n)$	exponential	
$O(n!)$	factorial	hard-solved

Эффективный алгоритм

- Делает то, что нужно!
  - *Решает задачу*
- Оптимально использует **ресурсы**
  - ***Время***
    - Количество шагов (длина трассы)
  - ***Память***
    - Количество ячеек памяти

Измерение объема ресурсов

- Точное измерение
  - замеряем время работы в микро-(нано-) секундах
  - замеряем точный размер использованной памяти
- *Не годится*. Почему?

Зависимость ресурсов

- Использование ресурсов зависит от задачи
- Точнее – от входных данных
- Ещё точнее – от их размера

Размер входных данных

- Понятие относительное
  - При обработке массивов – размер массива (количество ячеек/значений в нём)
  - При обработке чисел – количество цифр
- Почему это нас не беспокоит?

Сложность вычислений

- Функция от входных данных
  - T(n) – по времени
  - S(n) – по памяти
- Можно выразить формулой
  - нам не нужно задавать точную формулу
  - ***порядок зависимости*** – важнее!

Оценка сложности

- Виды оценок сложности:
  - в лучшем случае (оценка снизу)
  - в худшем случае (оценка сверху)
  - в среднем
- Нас в первую очередь интересует **сложность в худшем случае**
  - почему?

Асимптотическая оценка

«О-большое» (не путать с «о-маленьким»)

$f = (O(g))$, если есть константа C, что

$$f(x) \leq C * g(x)$$

Оценка сверху

– «точно будет не больше $g(x)$»

Свойства O(f)

- Сумма

$$O(f) + O(g) = O(\max\{f, g\})$$

- Умножение на константу

$$C \cdot O(f) = O(f)$$

- Произведение

$$O(f) \cdot O(g) = O(f \cdot g)$$

Пример:

$$O(n^2) + O(n) = O(n^2)$$
$$n \cdot O(n) = O(n^2)$$

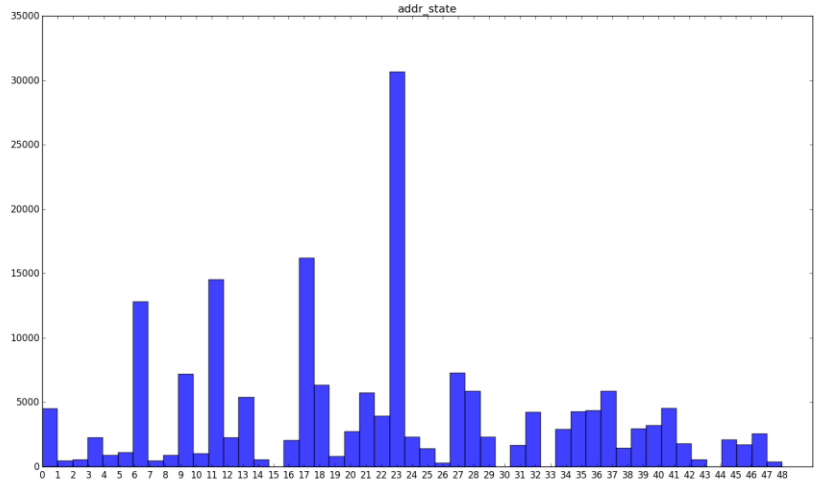
Виды сложности

- **Логарифмическая - $O(\log n)$**
  - основание не важно
- **Полиномиальная - $O(n^k)$**
  - линейная и константная – частные случаи
- **Экспоненциальная - $O(k^n)$**

Максимум массива

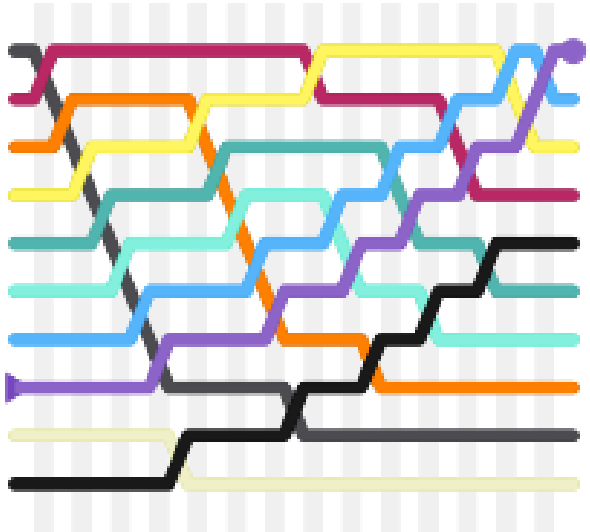
- Какова сложность?

```
Ввод числа;
int max = a[0];
foreach(int x in a)
{
    if (x > max)
    max = x;
}
```



Сортировка пузырьком

Сложность?



```
for (int i = 0; i < n - 1; i++)
{
    bool swapped = false;
    for (int j = 0; j < n - i - 1; j++)
        if (a[j] > a[j+1])
        {
            h = a[i];
            a[i] = a[j];
            a[j] = h;
            swapped = true;
        }
    if (!swapped)
        break;
}
```

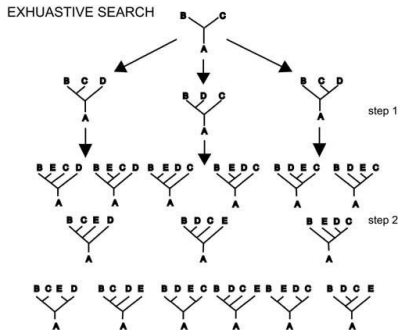
Циклы – источник полиномиальной сложности

- Вложенность – порядок полиномиальной сложности
- Какова сложность соседних циклов?

```
for (int i = 0; i < n; i++)
    ...
for (int i = 0; i < n; i++)
    ...
```

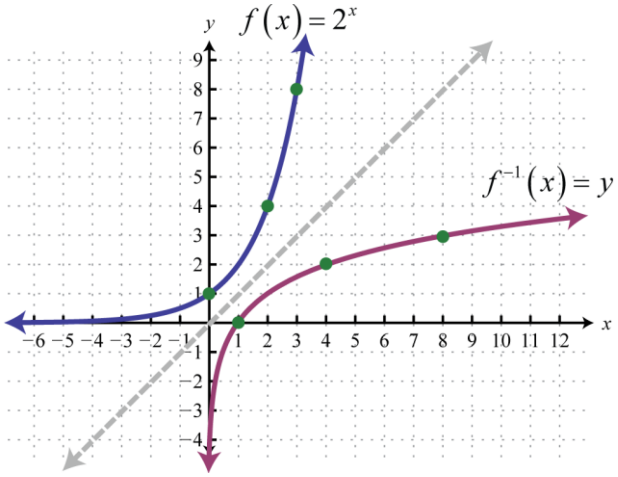
Cruel World

- Если бы все алгоритмы имели полиномиальную сложность…
  - было бы классно, но
  - стали бы бесполезными криптографические системы, обеспечивающие конфиденциальность, в том числе банковскую
- Но:
  - *Перебрать все сочетания из 4-х цифр*
    - *сколько сочетаний?*
    - *такова и сложность – экспоненциальная!*



Логарифмическая сложность

- Бинарный поиск в массиве
- Быстрое возведение в целую степень
- НОД (Алгоритм Евклида)
  – теорема Ламе



Сложность нужно оптимальную

- Типичная ошибка

Вычислить

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

с фиксированным числом слагаемых n

- *Что здесь плохого?*

```
double exp = 0;
for(int k=0; k<=n; k++)
{
    double pow = 1,fact = 1;
    for(int j=1; j<=k; j++)
    {
        pow *= x;
        fact *= j;
    }
    exp += pow / fact;
}
```

Как нужно

В $\sum_{k=0}^{n} \frac{x^k}{k!}$ каждое следующее слагаемое отличается от предыдущего на множитель x/k, поэтому:

```
double exp = 1, item = 1;
for (int k = 1; k <= n; k++)
{
    item *= x / k;
    exp += item;
}
```

Вещественные числа

- Хранятся в памяти с ограничениями
  - количество цифр ограничено мантиссой
- Вычислять необходимо с определенной точностью (до неё, дальше не нужно)
  - у калькулятора на экране 10 символов, зачем считать дальше 9го знака после запятой?

Точность вычислений

- $\forall \varepsilon > 0 \; \exists n = n(\varepsilon) \in \mathbb{N} \mid \forall \delta > n(\varepsilon) \; (|a_n - A| < \varepsilon)$
- A – предел последовательности a_n
- Раскрыв модуль: $A - \varepsilon < a_n < A + \varepsilon$
- Элемент a_n последовательности «похож» на A (с точностью ε)
  - можно брать всё меньший и меньший ε
  - начиная с какого-то номера элементы будут еще более «похожи» на A

Приближенные вычисления

- Математический анализ говорит о любом ε - какое бы малое не взяли, последовательность будет стремиться к А
  - На бесконечности будет «равна А»
- В программировании не нужна бесконечная точность.
- **Точность (ε)– фиксирована**
- **Продолжаем** вычисления только **пока не достигнем** фиксированную **точность**

Пример

- Если элементы «похожи» на предел, то они «похожи» между собой
  - Фундаментальная последовательность!
  - Любая сходящаяся последовательность – фундаментальная
    - если не знаем предел заранее
    - если знаем что предел есть
- Вычислить предел $\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$
  - чему равен предел?
  - как программно получить?

Вычисление

- Фиксируем точность 0.000001

$$a(1) = \left(1 + \frac{1}{1}\right)^1 = 2$$

$$a(2) = \left(1 + \frac{1}{n}\right)^2 = 1.5 * 1.5 = 2.25$$

- Разница между a(2) и a(1)=0.25 больше 0.000001. Продолжаем

Вычислени*е*

a(3) = 2.370370
a(3) – a(2) = 0.120370 > eps
a(4) = 2.441406
a(4) – a(3) = 0.071036 > eps
a(5) = 2.48832
a(5) – a(4) = 0.046914 > eps
…
a(100) = 2.704811
a(101) = 2.704946
a(101) – a(100) = 0.000135
…
a(200) = 2.711517
a(201) = 2.711550
a(201) – a(200) = 0.000033
…
a(300) = 2.713765
a(301) = 2.713780
a(301)-a(300) = 0.000015

- Сходство элементов друг с другом выше, чем сходство с пределом
- Какова сложность (примерно)?
  - выгоднее вычислять или хранить?
- При вызове Math.Exp, Math.Log, Math.Pow происходят такие же вычисления
  - Вызов Math.Pow(x,2) – очень плохо!

Пример

Вычисление

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

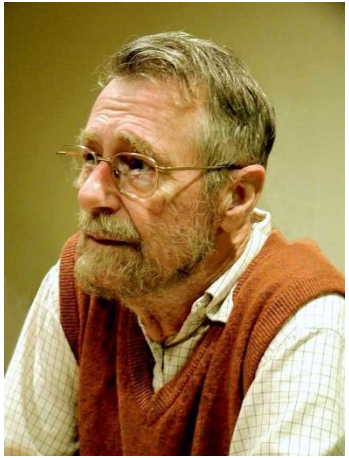
с точностью ε

```
double exp = 1, item = 1;
double eps = 0.00000000000001;
int k = 1;
while(item>eps)
{
    item *= x / k;
    exp += item;
    k++;
}
```

Эффективность построения

Идеи Э. Дейкстры

- **«Простота – залог надёжности»**
- Не только структурирование программ с помощью следования, ветвлений и циклов
- Программа – иерархическая структура блоков
  - достигается с помощью подпрограмм
- Проектирование сверху-вниз



Проектирование сверху-вниз

- Top-down проектирование
  - начинается с самого общего (абстрактного)
  - процесс проектирования – последовательные уточнения
- Позволяет найти пути преодоления больших сложностей

Метод прогрессивного jpeg-а

«В любую секунду любой проект готов на 100%, хотя проработанность может быть и на 4%»

А. Лебедев

Проектирование сверху-вниз

- Программа в каждый момент написана
  – Мы лишь конкретизируем её части
- Пример: для заданной высоты (n), вывести треугольник из единичек
  - n строк
  - в 1 строке n-1 пробелов, одна 1, перенос строки
  - в k строке n-k пробелов, 2k-1 единиц, перенос строки

1
111
11111
1111111
111111111

От наброска к реализации

```
// Первый набросок: общий вид
using System;
class Program
{
    static void Main()
    {
        //TODO ввод n;

        //TODO вывод треугольника
        //высоты n;
    }
}
```

```
// Ввод-вывод понятен
using System;
class Program
{
    static void Main()
    {
        int n =
        int.Parse(Console.ReadLine());
        //TODO вывод треугольника
        //высоты n;
    }
}
```

Анализ, детализация

```
// Раскрытие вывода строк
using System;
class Program
{
    static void Main()
    {
        int n =
        int.Parse(Console.ReadLine());
        for(int i=1; i<=n; i++)
        {
            //TODO вывод i-ой строки;
        }
    }
}
```

```
// Набросок вывода содержимого строки
using System;
class Program
{
    static void Main()
    {
        int n =
        int.Parse(Console.ReadLine());
        for(int i=1; i<=n; i++)
        {
            //TODO вывод n-i пробелов
            //TODO вывод 2n-1 единиц
            //TODO перенос строки
        }
    }
}
```

Результат проектирования

```
// Первый набросок
using System;
class Program
{
    static void Main()
    {
        //TODO ввод n;

        //TODO вывод треугольника
        //высоты n;
    }
}
```

```
// Конечный результат
using System;
class Program
{
    static void Main()
    {
        int n =
        int.Parse(Console.ReadLine());
        for(int i=1; i<=n; i++)
        {
            for (int j = 0; j < n - i; j++)
                Console.Write(' ');
            for (int j = 0; j < 2*i - 1; j++)
                Console.Write(1);
            Console.WriteLine();
        }
    }
}
```

Подпрограммы

- Структурирование
- Повторное использование
- Проектирование сверху-вниз

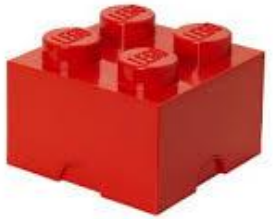
Подпрограмма

- Отдельная часть программы
  - законченный фрагмент программы
- Обладает такой же структурой, как и вся программа
- Позволяет:
  - избежать дублирования кода
  - улучшить структуру программы
  - избежать излишнего погружения в детали
  - производить независимую отладку части программы

Процедуры и функции

- Реализуют подпрограммы в процедурных языках
  - C, Fortran, Pascal, BASIC
- Входы – аргументы
  - передача значений фактических параметров
- Выходы – возвращаемые значения
- Области видимости
  - процедура и
- Позволяют совместно разрабатывать ПО
- Позволяют формировать библиотеки кода

Модули



- Переход от «программирования в малом» к «программированию в большом»
  - от структурности к модульности (1960 – 1970-е)
- Модуль – законченная программная единица, реализующая функциональность и предоставляющая интерфейс к ней
  - файл исходного кода, компилируемый отдельно
  - библиотека функций, замена компонент без пересборки программы
  - сервис
  - …
- Дальнейшее развитие идей – объектно-ориентированное программирование (1980-1990-е)

Процедуры и функции

- Статические методы в C# – аналог процедур и функций
(не нужен объект для вызова)
  - процедура не возвращает значение (void)
- `static void Main(string[] args){…}`
- тип_значения Имя (параметры){ тело }

```
static int Max(int a, int b)
{
    return a > b ? a : b;
}
```

Параметры

- Входные данные передаются в метод через параметры
- Могут передаваться
  - **по значению,**
  - **по ссылке** (ref),
  - **для выходного значения** (out)
    - например, если нужно возвратить несколько значений

Пример

- По значению

```
static int Max(int a, int b)
{
    return a > b ? a : b;
}
…
int y = Max(n, 10);
```

- По ссылке

```
static void Swap(ref int a, ref int b)
{
    int t = a;  a = b;  b = t;
}
…
Swap(ref x, ref y);
```

Возвращаемое значение

- Возвращение через return

```
static int Max(int a, int b)
{
    return a > b ? a : b;
}
…
int y = Max(n, 10);
```

- Возвращение в параметр
  - не обязательно инициализировать переменную

```
double value;
double.TryParse(str,out value);
```

Стек

- При вызове метода, память для него (фрейм) помещается в стек
- Память возвращается стеку при выходе из метода
- Время жизни локальных переменных метода ограничено временем работы метода

Эффективность написания

Нагромождение переменных

- Пример: вычислить $(a-1)*b+(b-5)*d$

```
int x = a - 1;
int y = x * b;
int z = b - 5;
int u = z * d;
int v = y + u;
```

Дублирование кода

Может влиять на сложность

```
if (x*x - 2*x > 5 - x)
    Console.WriteLine(x*x - 2*x);
else
    Console.WriteLine(5 - x);
```

Без дублирования:

```
int a = x * x - 2 * x;
int b = 5 - x;
int max = a > b ? a : b; //Math.Max
Console.WriteLine(max);
```

Дублирование кода

Может не влиять на сложность

$n!! = 2 * 4 * 6 * \cdots * n$

(если четное)

$n!! = 1 * 3 * 5 * \cdots * n$

(если n – нечетное),

```
if (n%2 == 1)
{
    int p = 1;
    for (int i = 1; i <= n; i++)
        p *= i;
}
else
{
    int p = 2;
    for (int i = 1; i <= n; i++)
        p *= i;
}
```

Упростим

```
int p = 2 - n % 2;
for (int i = 1; i <= n; i++)
    p *= i;
```

Или даже так:

```
int p = 1;
while (n >= 1)
{
    p *= n;
    n -= 2;
}
```

Сложные ветвления

Вывести max{a, b, c}

```
if (a > b)
{
    if (a > c)
        Console.WriteLine(a);
    else
        Console.WriteLine(c);
}
else
{
    if (b > c)
        Console.WriteLine(b);
    else
        Console.WriteLine(c);
}
```

Без лишнего ветвления

```
int max = a;
if (b > a)
    max = b;
if (c > a)
    max = c;
Console.WriteLine(c);
```

Code conventions

- Существуют правила и рекомендации по написанию кода
- Они помогают облегчить чтение и улучшить понимание кода
- Хороший тон
- Почерк специалиста

Комментарии

- Размещаются на отдельной строке, а не в конце строки
- Начинаются с заглавной буквы
- Завершаются точкой
- Между разделителем // и текстом ставится пробел

Комментарии

```
/// <summary>
/// Нахождение максимума двух целых чисел.
/// </summary>
static int Max(int a, int b)
{
    // Возвращается большее из чисел.
    return a > b ? a : b;
}
```

Расположение

- Один оператор в строке
- Одно объявление в строке
- Отступ – 4 пробела
- Пустая строка между определениями методов и свойств
- Использовать скобки для ясности
  - операторные и круглые

Расположение

```
/// <summary>
/// Обмен значениями двух переменных одного типа.
/// </summary>
static void Swap<T>(ref T a, ref T b)
{
    T t = a;
    a = b;
    b = t;
}

static void Main(string[] args)
{
    int dInput1;
    int.TryParse(Console.ReadLine(), out dInput1);
    int dInput2;
    int.TryParse(Console.ReadLine(), out dInput2);
    Swap(ref dInput1, ref dInput2);
    Console.WriteLine($"{dInput1} {dInput2}");
}
```

Имена

- Избегайте сокращённых вариантов
- Указывайте осмысленные имена
- Используйте глаголы для именования методов
- При наличии аббревиатур в именах
  - два символа – оба заглавные (IO)
  - более двух символов – первый (HttpUtil)

PascalCasing

- Описываются имена
  - определений типов
  - значений перечислений
  - констант и readonly полей
  - **методов**
  - свойств
  - публичный полей

PascalCasing

```
class SampleClass
{
    public int SampleIntProperty { get; set; }

    public int SampleMethod { get; set; }

    const int SampleIntConstant = 42;

    public enum SampleEnum
    {
        Value1, Value2, Value3
    }

    public int SampleIntField;
}
```

camelCasing

- Описываются имена
  - локальных переменных
  - параметров методов

```
static void Swap (ref int firstParam, ref int secondParam)
{
    int temp = firstParam;
    firstParam = secondParam;
    secondParam = temp;
}
```

Рекомендации по объёму

- Длина строки не более 120 символов
- В методе не более 5 параметров
- В методе не более 200 строк кода
- В файле не более 500 строк кода

Вопросы?

e-mail: marchenko@it.kfu.ru