Адрианов Н.М.
Иванов А.Б.

Алгоритмы и структуры данных

Введение. Сложность алгоритма

Т. Кормен, Ч. Лейзерсон,
Р. Ривест, К. Штайн

Алгоритмы. Построение и анализ

2-е издание, 2006

3-е издание, 2013

С. Дасгупта, Х. Пападимитриу,
У. Вазирани

Алгоритмы

2014

R. Sedgewick, K. Wayne

Algorithms

4th edition, 2011

Алгоритмы на Java

4-е издание, 2013

http://www.coursera.org/

Robert Sedgewick, Kevin Wayne
Algorithms, Part I
Algorithms, Part II

Tim Roughgarden
Algorithms: Design and Analysis, Part 1
Algorithms: Design and Analysis, Part 2

Что будет:

- «Разделяй и властвуй»
- Жадные алгоритмы
- Динамическое программирование
- Линейное программирование

- Умножение (числа, многочлены, матрицы)
- Сортировки
- Графы

- Теоретические задачи
- Практические задания: Java или C#

Числа Фибоначчи

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ 233,\ 377,\ \ldots$$

Числа Фибоначчи

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ 233,\ 377,\ \ldots$$

```
procedure Fib(n):
  if n = 0:
    return 0
  if n = 1:
    return 1

  return Fib(n-1) + Fib(n-2)
end
```

Числа Фибоначчи

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$$

```
procedure Fib(n):
  if n = 0:
    return 0
  if n = 1:
    return 1

  return Fib(n-1) + Fib(n-2)
end
```

```
procedure Fib(n):
  if n = 0:
    return 0
  if n = 1:
    return 1

  (a,b) = (0,1)
  for i in [2..n]:
    (a,b) = (b,a+b)
  return b
end
```

Числа Фибоначчи

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$$

```
procedure Fib(n):
  if n = 0:
    return 0
  if n = 1:
    return 1

  return Fib(n-1) + Fib(n-2)
end
```

```
procedure Fib(n):
  if n = 0:
    return 0
  if n = 1:
    return 1

  (a,b) = (0,1)
  for i in [2..n]:
    (a,b) = (b,a+b)
  return b
end
```

$\sim F_n \approx 2^{0.694n}$

$\sim n$

Анализ алгоритма

1. Правильно ли работает алгоритм?

2. Какова сложность (время работы в зависимости от n)?

3. Существует ли более быстрый алгоритм?

Сложность в худшем случае и в среднем

Ω_n – множество всех допустимых наборов входных данных размера n.

$T(\omega)$ – сложность (количество операций) алгоритма A на входных данных $\omega \in \Omega_n$.

Сложность **в худшем случае**:

$$T(n) = \max_{\omega \in \Omega_n} T(\omega)$$

Сложность **в среднем** (average):

$$T_{avg}(n) = \frac{1}{|\Omega_n|} \sum_{\omega \in \Omega_n} T(\omega)$$

O-нотация

$$f,\, g : \mathbb{N} \to \mathbb{R}_+ = \{x \in \mathbb{R} \,|\, x > 0\}$$

$f = O(g)$ (f растет не быстрее g),
если существует $c \in \mathbb{R}_+$, что $f(n) \leq c \cdot g(n)$.

O-нотация

$$f,\, g : \mathbb{N} \to \mathbb{R}_+ = \{x \in \mathbb{R} \,|\, x > 0\}$$

$f = O(g)$ (f растет не быстрее g),
если существует $c \in \mathbb{R}_+$, что $f(n) \leq c \cdot g(n)$.

$f = \Omega(g)$ (f растет не медленнее g),
если существует $c \in \mathbb{R}_+$, что $f(n) \geq c \cdot g(n)$.

$f = \Theta(g)$ (f и g имееют одинаковый порядок роста),
если $f = O(g)$ и $g = O(f)$.

Сортировка массива

Различная сложность алгоритмов решения одной и той же задачи хорошо иллюстрируется алгоритмами сортировки. Начнем с них и мы.

А зачем вообще нужно сортировать массивы?

Поиск элемента в массиве

Дано: массив `a[]`, элемент `x`
Требуется: определить, содержится ли `x` в `a[]`

Поиск элемента в массиве

Дано: массив `a[]`, элемент `x`
Требуется: определить, содержится ли `x` в `a[]`

Решение: перебрать все элементы `a[]`, сравнивая их с `x`
Сложность: $O(n)$

Бинарный поиск

Дано: **отсортированный** массив `a[]`, элемент `x`
Требуется: определить, содержится ли `x` в `a[]`

Бинарный поиск

Дано: **отсортированный** массив `a[]`, элемент `x`
Требуется: определить, содержится ли `x` в `a[]`

- Делим массив пополам
- Сравниваем `x` с центральным элементом `a[mid]`
  - `x` = `a[mid]`? – нашли
  - `x` < `a[mid]`? – ищем в нижней части массива
  - `x` > `a[mid]`? – ищем в верхней части массива

Бинарный поиск

Пусть $n \leq 2^k$ ($k = \lceil \log n \rceil$)

Итераций: $k + 1$

На каждой итерации: $O(1)$

Итого: $O(\log n)$

```
procedure BinarySearch( a[], x ):
  // a[] - массив от 0 до n-1
  low = 0
  high = n-1
  while low <= high:
    mid = low + (high - low) / 2

    if a[mid] < x:
      low = mid + 1
    else if a[mid] > x:
      high = mid - 1
    else
      return mid
  end
  return -1
end
```

Бинарный поиск – простой алгоритм?..

Donald Knuth (*Искусство программирования*):
бинарный поиск был впервые опубликован в 1946 году, но только в 1962 году была опубликована первая версия без багов.

Jon Bentley (*Жемчужины программирования*):
только 10% профессиональных программистов смогли реализовать бинарный поиск без ошибок.

Joshua Bloch (2006):
образцовый код в книге Бентли также содержит ошибку (также ошибка была в реализации для языка Java – и оставалась незамеченной почти 10 лет)
http://googleresearch.blogspot.ru/2006/06/extra-extra-read-all-about-it-nearly.html

Сортировка выбором (Selection sort)

3	7	2	6	4	8	1	5

```
procedure SelectionSort( a[] ):
  // a[] - массив от 0 до n-1
  for i in [1..n-1]:
    j = индекс максимального
         элемента в a[0..n-i]
    переставить a[j] и a[n-i]
  end
end
```

Сортировка выбором (Selection sort)

3	7	2	6	4	8	1	5

```
procedure SelectionSort( a[] ):
  // a[] - массив от 0 до n-1
  for i in [1..n-1]:
    j = индекс максимального
         элемента в a[0..n-i]
    переставить a[j] и a[n-i]
  end
end
```

Анализ: количество сравнений

$$(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$$

$$O(n^2)$$

Сортировка вставками (Insertion sort)

3	7	2	6	4	8	1	5

```
procedure InsertionSort( a[] ):
  // a[] - массив от 0 до n-1
  for i in [1..n-1]:
    j = i, t = a[j]
    while j > 0 && t < a[j-1]:
      a[j] = a[j-1]
      j = j-1
    end
    a[j] = t
  end
end
```

Сортировка вставками (Insertion sort)

3	7	2	6	4	8	1	5

```
procedure InsertionSort( a[] ):
  // a[] - массив от 0 до n-1
  for i in [1..n-1]:
    j = i, t = a[j]
    while j > 0 && t < a[j-1]:
      a[j] = a[j-1]
      j = j-1
    end
    a[j] = t
  end
end
```

В лучшем случае сравнений

$$1 + 1 + \ldots + 1 = n - 1$$

в худшем случае

$$1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2}$$

Сортировка слиянием (Merge sort)

«Разделяй и властвуй» (Divide and conquer)

- Разделим массив на 2 части размера $n/2$
- Отсортируем обе части (2 рекурсивных вызова)
- Выполним процедуру слияния: объединяем отсортированные части таким образом, чтобы получить полностью отсортированный массив

Сортировка слиянием

Ключевая процедура – слияние (merge)

Особенность: слияние требует дополнительный массив `aux` (auxiliary - вспомогательный)

Выделяем его сразу, чтобы не делать этого при каждом вызове

```
procedure MergeSort( a[], aux[],
                     low, high ):
  if (high <= low):
   return

  mid = low + (high - low) / 2

  MergeSort(a, aux, low, mid)
  MergeSort(a, aux, mid + 1, high)

  Merge(a, aux, low, mid, high)
end

procedure MergeSort( a[] ):
  // a[] - массив от 0 до n-1
  aux = создать дополнительный
        массив [0..n-1]
  MergeSort(a, aux, 0, n-1)
end
```

Сортировка слиянием

3	7	2	6	4	8	1	5


```
procedure Merge( a[], aux[],
                 low, mid, high ):
  for k in [low..high]:
    aux[k] = a[k]

  i = low, j = mid + 1
  for k in [low..high]:
    if i > mid:
      a[k] = aux[j++]
    else if j > high:
      a[k] = aux[i++]
    else if aux[j] < aux[i]:
      a[k] = aux[j++]
    else:
      a[k] = aux[i++]
  end
end
```

Сортировка слиянием: сложность

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

Сортировка слиянием: сложность

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

$$T(n) = O(n \log n)$$

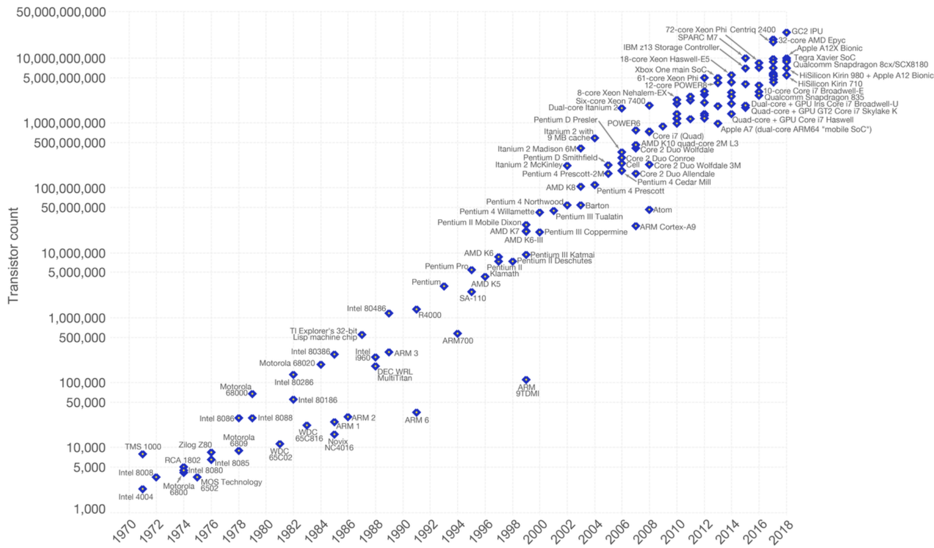
Нужны ли быстрые алгоритмы?

Так ли важно – $O(n^2)$ или $O(n \log n)$?

Предположим: PC выполняет 10^8 операций в секунду, гипотетический суперкомпьютер – 10^{12} операций.

	n	n^2	$n \log n$
	10^3	10^6	10^4
PC:		0 сек	0 сек
Supercomputer:		0 сек	0 сек
	10^6	10^{12}	$2 \cdot 10^7$
PC:		2.8 часа	0.2 сек
Supercomputer:		1 сек	0 сек
	10^9	10^{18}	$3 \cdot 10^{10}$
PC:		317 лет	5 мин
Supercomputer:		12 суток	0 сек

Закон Мура



Закон Мура

Если бы авиапромышленность в последние 25 лет развивалась столь же стремительно, как компьютерная техника, то сейчас самолёт Boeing 767 стоил бы 500$ и совершал облёт земного шара за 20 минут, затрачивая при этом пять галлонов топлива.

Scientific American, 1983

Закон Мура - следствие

Можно ли можно пренебречь эффективностью алгоритмов, так как производительность компьютеров растет, в соответствии с законом Мура, экспоненциально?

Наоборот, закон Мура увеличивает важность эффективных алгоритмов!

Вместе с ростом производительности растут объемы памяти и размеры входных данных, которые хочется обрабатывать.