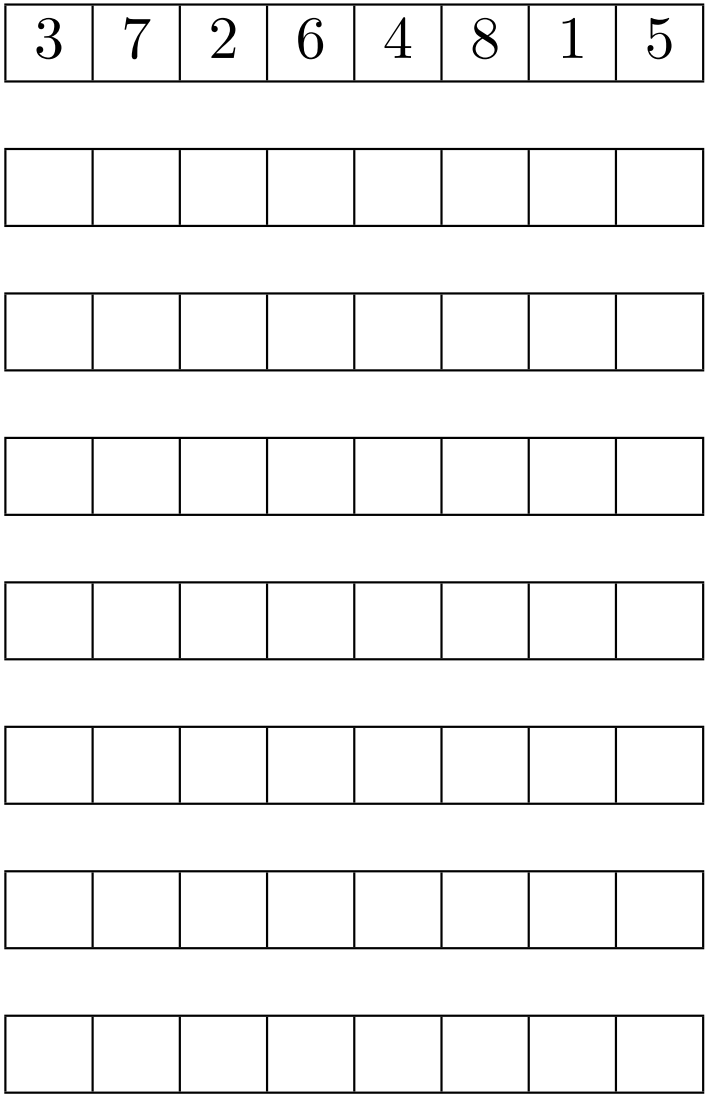
Адрианов Н.М.
Иванов А.Б.

Алгоритмы и структуры данных

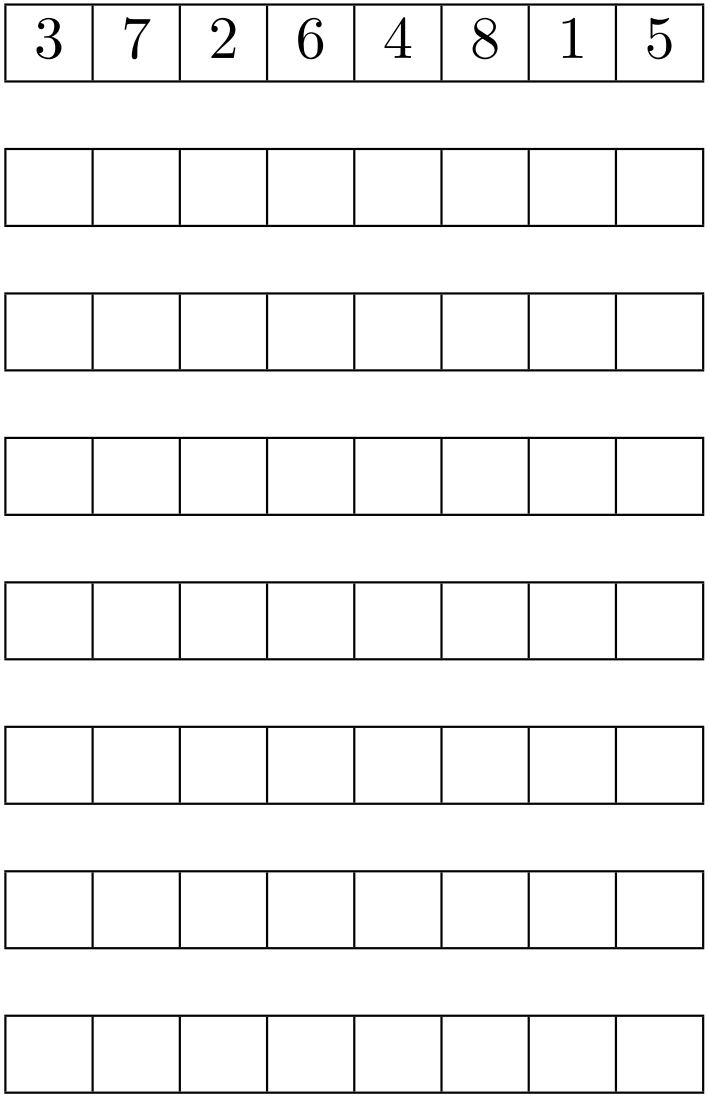
Сортировки

Сортировка пузырьком (Bubble sort)




```
procedure BubbleSort( a[] ):
  // a[] - массив от 0 до n-1
  for i in [1 .. n-1]:
    for j in [0 .. n-i- 1]:
      if (a[j] > a[j+1]:
        переставить a[j] и a[j+1]
      end
    end
  end
end
```

Сортировка пузырьком (Bubble sort)



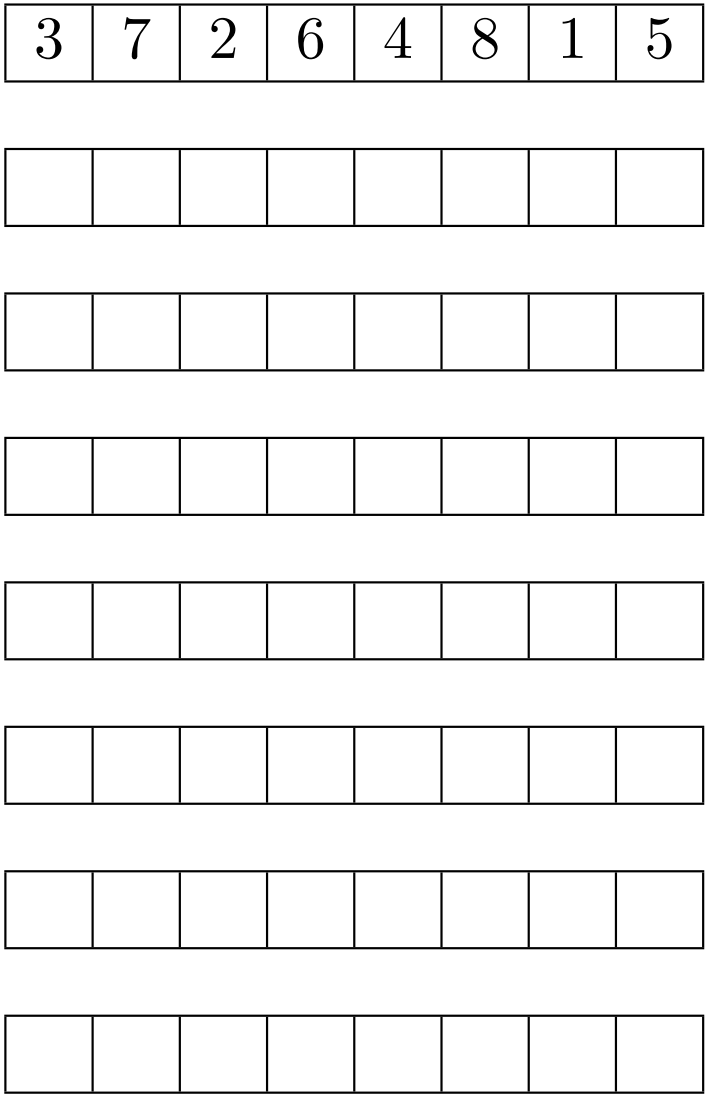

```
procedure BubbleSort( a[] ):
  // a[] - массив от 0 до n-1
  for i in [1 .. n-1]:
    for j in [0 .. n-i- 1]:
      if (a[j] > a[j+1]:
        переставить a[j] и a[j+1]
      end
    end
  end
end
```

Анализ: количество сравнений

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

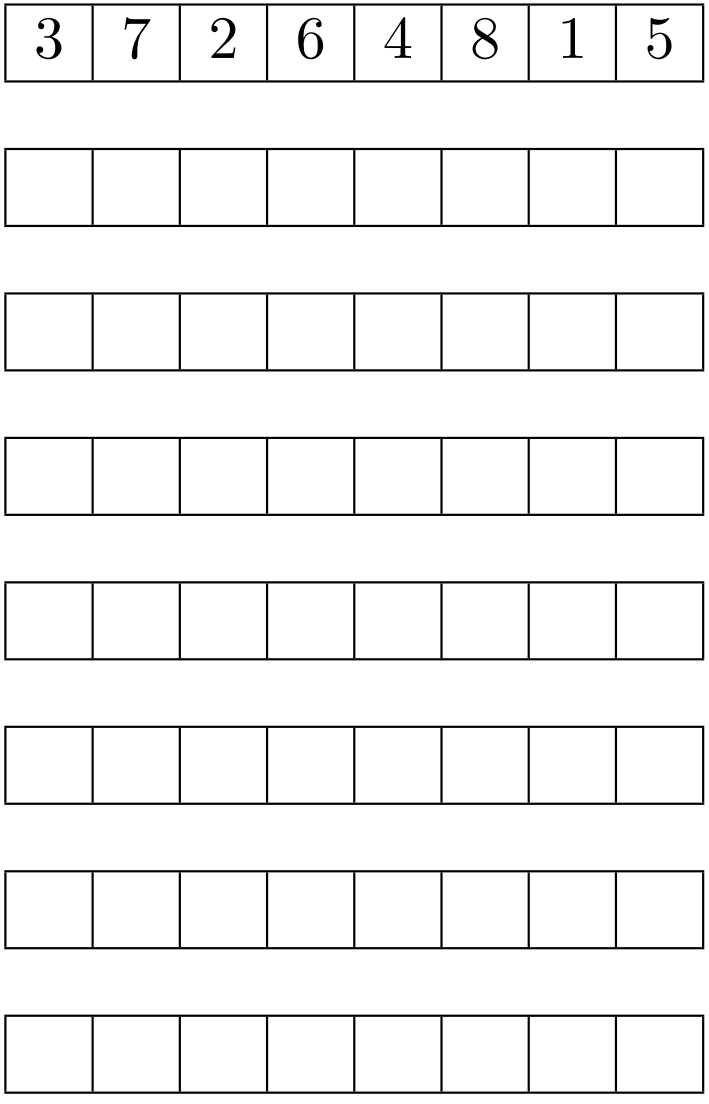
$$O(n^2)$$

Сортировка выбором (Selection sort)




```
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)



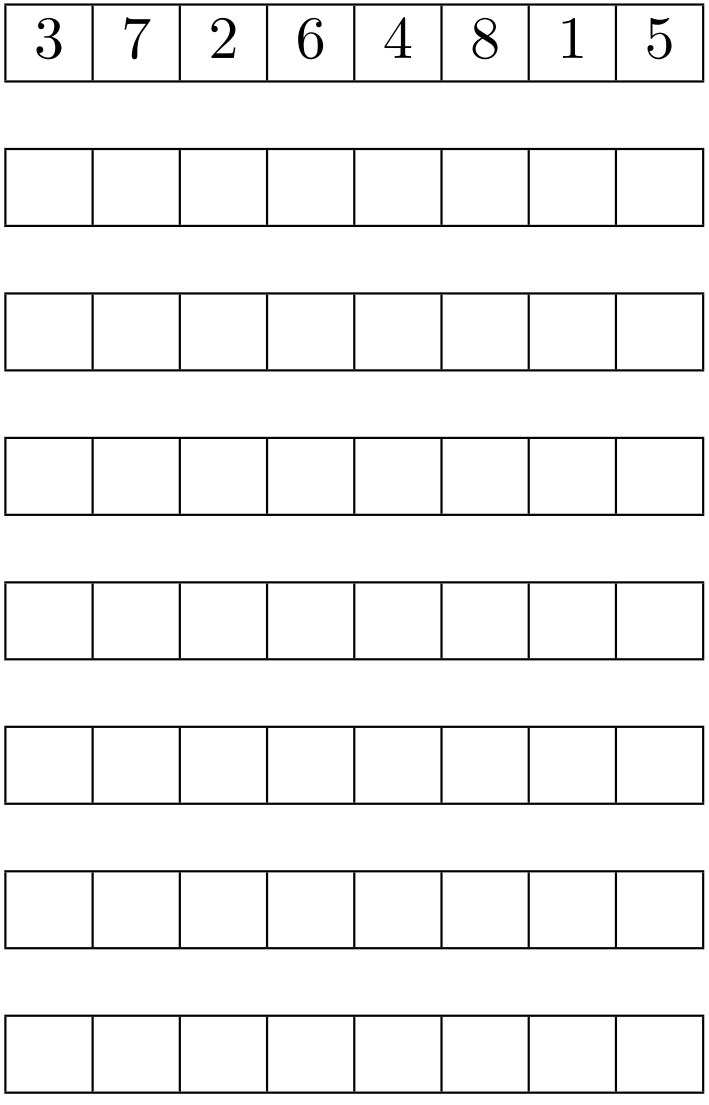

```
procedure SelectionSort( a[] ):
  // a[] - массив от 0 до n-1
  for i in [1..n-1]:
    j = индекс максимального элемента в
        a[0..n-i]
    переставить a[j] и a[n-i]
  end
end
```

Анализ: количество сравнений

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

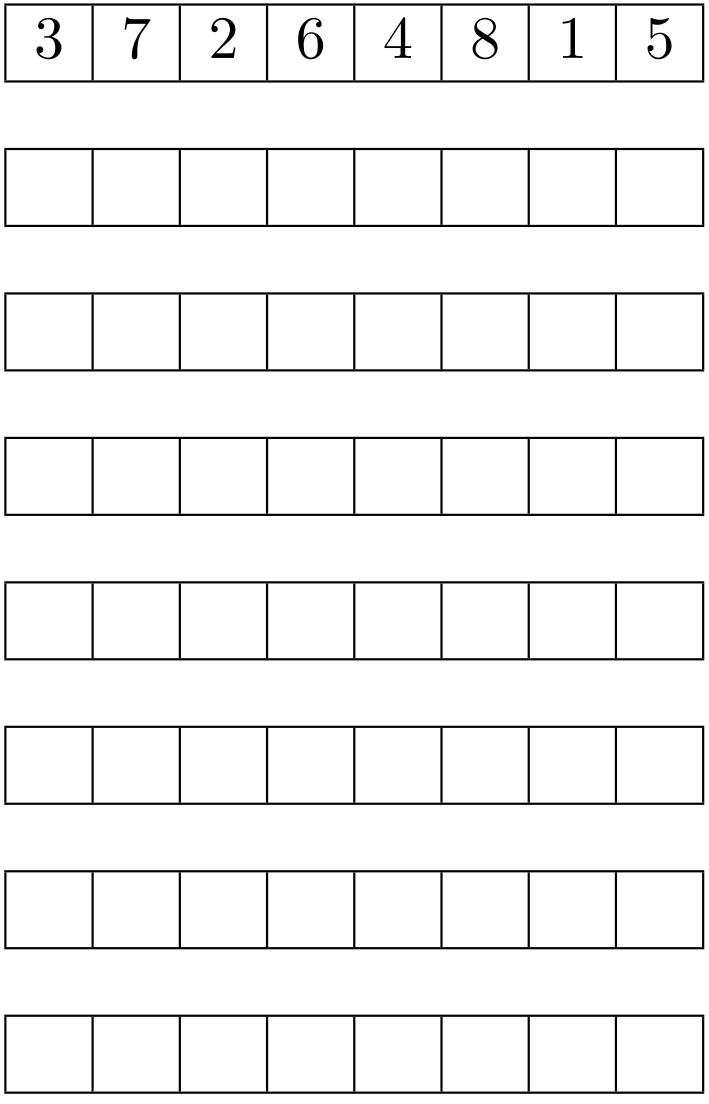
$$O(n^2)$$

Сортировка вставками (Insertion sort)




```
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)




```
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

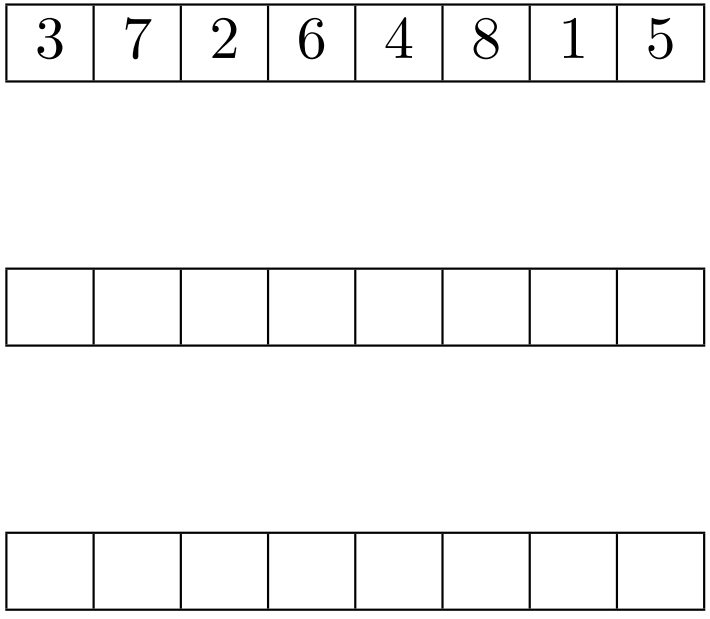
Ключевая процедура – слияние (merge)

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




```
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

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

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

Разделяй и властвуй: основной метод*

Теорема. Если сложность алгоритма определяется рекуррентным соотношением

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

то

$$T(n) = \begin{cases} O(n^d \log n), & \text{если } a = b^d \\ O(n^d), & \text{если } a < b^d \\ O(n^{\log_b a}), & \text{если } a > b^d. \end{cases}$$

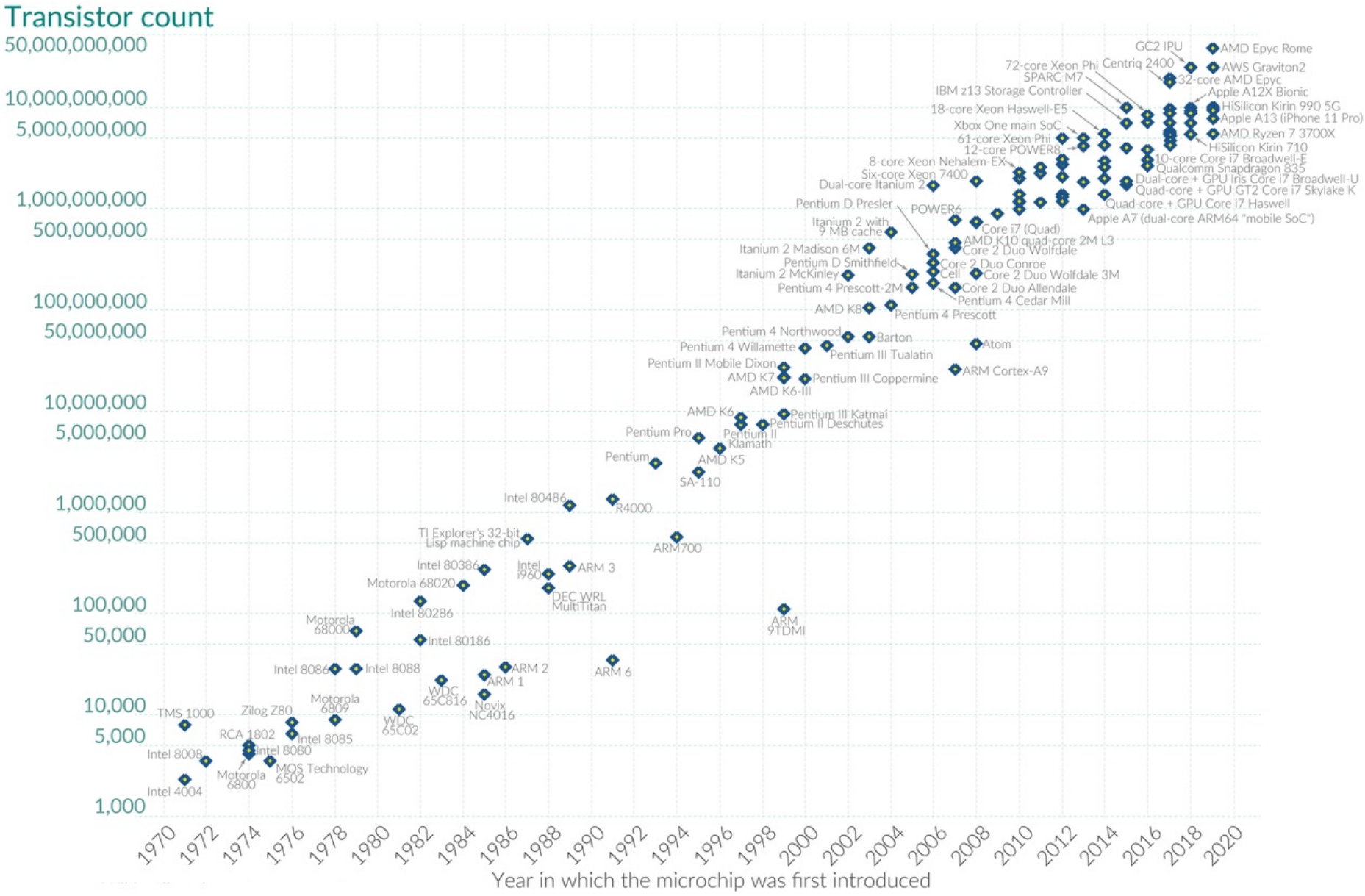
Нужны ли быстрые алгоритмы?

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	5мин
Supercomputer:		12 суток	0 сек

Закон Мура


Transistor count
50,000,000,000
10,000,000,000
5,000,000,000
1,000,000,000
500,000,000
100,000,000
50,000,000
10,000,000
5,000,000
1,000,000
500,000
100,000
50,000
10,000
5,000
1,000
GC2 IPU
AMD Epyc Rome
72-core Xeon Phi
Centriq 2400
AWS Graviton2
SPARC M7
32-core AMD Epyc
IBM z13 Storage Controller
Apple A12X Bionic
18-core Xeon Haswell-E5
HiSilicon Kirin 990 5G
Xbox One main SoC
Apple A13 (iPhone 11 Pro)
61-core Xeon Phi
AMD Ryzen 7 3700X
12-core POWER8
HiSilicon Kirin 710
8-core Xeon Nehalem-EX
10-core Core i7 Broadwell-E
Six-core Xeon 7400
Qualcomm Snapdragon 835
Dual-core Itanium 2
Dual-core + GPU Iris Core i7 Broadwell-U
Quad-core + GPU GT2 Core i7 Skylake K
Pentium D Presler
POWER6
Quad-core + GPU Core i7 Haswell
Itanium 2 with 9 MB cache
Apple A7 (dual-core ARM64 "mobile SoC")
Core i7 (Quad)
AMD K10 quad-core 2M L3
Itanium 2 Madison 6M
Core 2 Duo Wolfdale
Pentium D Smithfield
Core 2 Duo Conroe
Itanium 2 McKinley
Cell
Core 2 Duo Wolfdale 3M
Pentium 4 Prescott-2M
Core 2 Duo Allendale
Pentium 4 Cedar Mill
AMD K8
Pentium 4 Prescott
Pentium 4 Northwood
Barton
Atom
Pentium 4 Willamette
Pentium III Tualatin
Pentium II Mobile Dixon
ARM Cortex-A9
AMD K7
Pentium III Coppermine
AMD K6-III
AMD K6
Pentium III Katmai
Pentium II Deschutes
Pentium Pro
Pentium II Klamath
Pentium
AMD K5
SA-110
Intel 80486
R4000
TI Explorer's 32-bit Lisp machine chip
ARM700
Intel 80386
Intel i960
ARM 3
Motorola 68020
DEC WRL MultiTitan
Intel 80286
ARM 9TDMI
Motorola 68000
Intel 80186
Intel 8086
Intel 8088
ARM 2
ARM 6
ARM 1
WDC 65C816
Motorola 6809
Novix NC4016
TMS 1000
Zilog Z80
WDC 65C02
RCA 1802
Intel 8085
Intel 8008
Intel 8080
Motorola 6800
MOS Technology 6502
Intel 4004
1970
1972
1974
1976
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
Year in which the microchip was first introduced

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.

Нижняя оценка сложности сортировки

Теорема. Любой детерминированный алгоритм сортировки ***сравнением*** имеет сложность в ***худшем*** случае $\Omega(n \log n)$.

Количество перестановок в массиве из n элементов $= n!$

Работу алгоритма на различных входных данных можно представить в виде бинарного дерева. Каждое ветвление – сравнение элементов массива. Если при любых входных данных количество сравнений не больше S, то глубина дерева не больше S, а количество конечных узлов – не больше 2^S.

$$2^S \geq n! \qquad \Rightarrow \qquad S = \Omega(n \log n)$$

Формула Стирлинга: $n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$

или грубая оценка $n! > \left[\frac{n}{2}\right]^{\left[\frac{n}{2}\right]}$

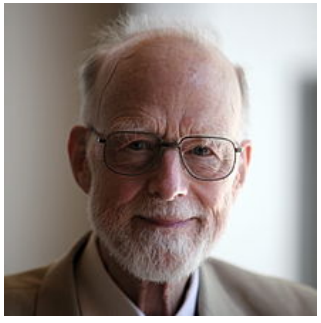
Свойства алгоритмов сортировки

- Сложность
- Необходимая память
- Устойчивость

Сортировка слиянием:

- $O(n \log n)$
- $\geq N$
- Устойчивый

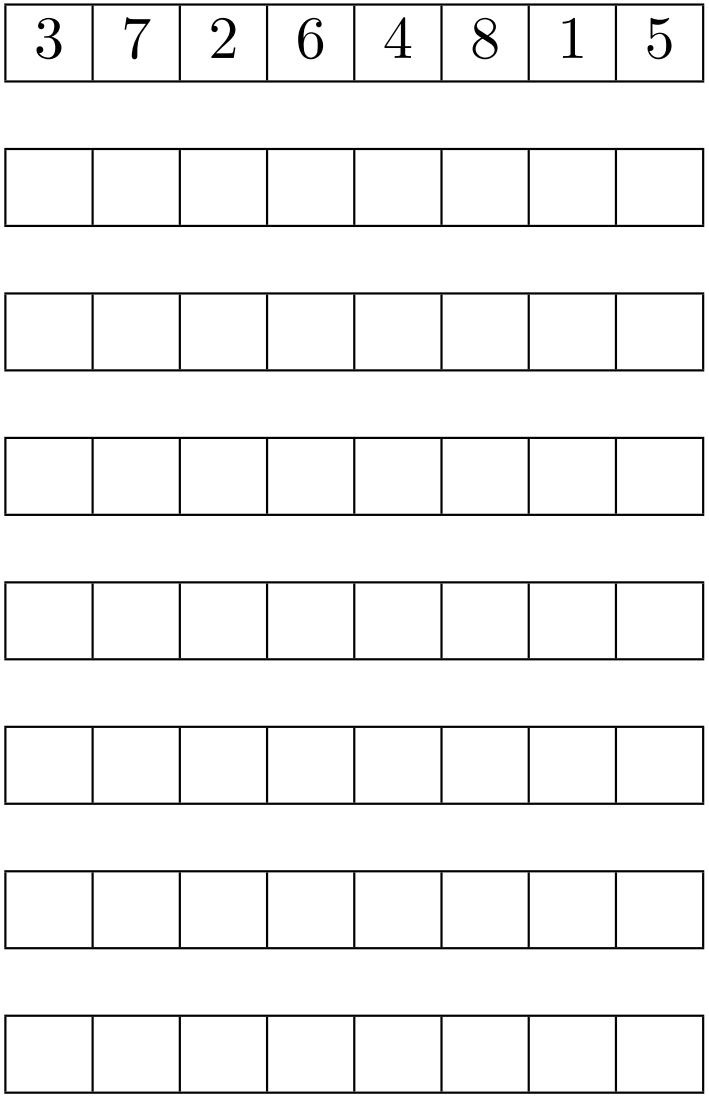
Быстрая сортировка (Quicksort)



Tony Hoare в 1959 году в СССР

Интерес Сэра Тони Хоара к компьютерным вычислениям проснулся в начале пятидесятых годов, когда он изучал философию (наряду с латинским и греческим) в Оксфордском университете, под руководством Джона Лукаса. Во время своей службы в Королевском военно-морском флоте изучал русский язык. В 1959 году, будучи аспирантом Московского государственного университета, он изучал машинный перевод языков и теорию вероятностей, в школе А.Н. Колмогорова. Для эффективного поиска слов в словаре, он разработал известный алгоритм «быстрой сортировки».
`http://theoryandpractice.ru/presenters/16008-toni-khoar`

Быстрая сортировка (Quicksort)




```
procedure QuickSort(a[], left, right):
  // a[] - массив от 0 до n-1
  pivot = a[left]
  i = left + 1
  for j in [left + 1 .. right]:
    if a[j] < pivot:
      переставить a[j] и a[i]
      i++
    end
  end
  переставить a[left] и a[i-1]

  QuickSort(a, left, i - 2)
  QuickSort(a, i, right)
end
```

Быстрая сортировка (корректность)

Быстрая сортировка (Quicksort)

Сложность? Выбор опорного элемента?

- В худшем случае: $O(n^2)$
- В лучшем случае (опорный элемент - медиана):

$$T(n) = 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n) \implies T(n) = O(n \log n)$$

- В «плохом» случае - опорный элемент делит массив в соотношении 99/100: $O(n \log n)$

Быстрая сортировка (Quicksort)

Теорема. Сложность быстрой сортировки «в среднем» – $O(n \log n)$ (при случайном выборе опорного элемента).

- Пространство элементарных событий $\Omega = \{$все возможные последовательности опорных элементов$\}$
- Случайная величина
  $X(\omega) =$"количество выполняемых сравнений при последовательности опорных элементов ω"
- Наша цель:
$$M[X] = O(n \log n)$$

Быстрая сортировка (Quicksort)

- z_i - i-ая порядковая статистика
- $X_{ij}(\omega)$ - количество сравнений z_i и z_j
- По определению

$$X(\omega) = \sum_{i<j} X_{ij}(\omega)$$

- Из линейности математического ожидания

$$M[X] = \sum_{i<j} M[X_{ij}]$$

- Или

$$M[X] = \sum_{i<j} P[X_{ij} = 1]$$

Быстрая сортировка (Quicksort)

$$P[X_{ij} = 1] = \frac{2}{j - i + 1}$$

$$M[X] = \sum_{i<j} P[X_{ij} = 1]$$

$$\Downarrow$$

$$M[X] = \sum_{i<j} P[X_{ij} = 1] = \sum_{i<j} \frac{2}{j - i + 1} =$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \leq 2n \sum_{k=2}^{n} \frac{1}{k} \leq 2n \ln n$$