Лекция 3.
Алгоритмы сортировки

Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: sillyhat34@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Весенний семестр, 2021 г.

Задача сортировки (sorting problem)

- **Дана** последовательность из *n* ключей

$$a_1, a_2, \ldots, a_n$$

- **Требуется упорядочить** ключи по неубыванию или по невозрастанию — найти ***перестановку*** $(i_1, i_2, \ldots, i_n)$ ключей
- **По неубыванию** (non-decreasing order)

$$a_{i_1} \le a_{i_2} \le \ldots \le a_{i_n}$$

- **По невозрастанию** (non-increasing order)

$$a_{i_1} \ge a_{i_2} \ge \ldots \ge a_{i_n}$$

Алгоритмы сортировки

- Алгоритму сортировки должен быть известен способ сравнения ключей

$$a < b$$

- Для этого ему сообщается внешняя функция сравнения (comparison function) — возвращает значение True, если $x \leq y$, и False в противном случае

```
int cmp(int a, int b)
{
    return a <= b ? 1 : 0;
}
```

- Алгоритм сортировки, использующий функцию сравнения ключей, называется **сортировкой сравнением** (comparison sort)

Алгоритмы сортировки

- Алгоритм сортировки, не меняющий относительный порядок следования равных ключей, называется **устойчивым** (stable)

(Сидоров, Сидор), (Есенин, Сергей), (Милосская, Венера), (Сидоров, Павел), (Милос, Рикардо)

Неустойчивая сортировка:

(Есенин, Сергей), (Милос, Рикардо), (Милосская, Венера), (Сидоров, Павел), (Сидоров, Сидор)

порядок не соблюдён

Устойчивая сортировка:

(Есенин, Сергей), (Милос, Рикардо), (Милосская, Венера), (Сидоров, Сидор), (Сидоров, Павел)

порядок соблюдён

Алгоритмы сортировки

- **Внутренние методы сортировки** (internal sort) — сортируемые элементы полностью размещены в оперативной памяти компьютера
- **Внешняя сортировка** (external sort) — элементы размещены во внешней памяти (жёсткий диск, USB-флеш-накопитель)

- Алгоритмы сортировки, не использующие дополнительной памяти (кроме сортируемого массива), называются алгоритмами **сортировки на месте** (in-place sort)

Алгоритмы сортировки

- **Алгоритмы, основанные на сравнениях** (comparison sort):
  Insertion Sort, Bubble Sort, Selection Sort, Shell Sort, Quick Sort, Merge Sort, Heap Sort и другие
- **Алгоритмы, не основанные на сравнениях:**
  Counting Sort, Radix Sort — используют структуру ключа

Утверждение. Любой алгоритм сортировки сравнением в худшем случае требует выполнения $\Omega(n \log n)$ сравнений.

[*] Donald Knuth. **The Art of Computer Programming, Volume 3: Sorting and Searching.** Second Edition. Addison-Wesley, 1997 (Section 5.3.1: Minimum-Comparison Sorting, pp. 180—197).

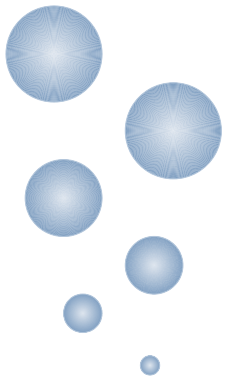
Алгоритмы сортировки

Алгоритм	Лучший случай	Средний случай	Худший случай	Память	Свойства
Сортировка вставками (Insertion Sort)	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Устойчивая, на месте, online
Сортировка выбором (Selection Sort)	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Устойчивость зависит от реализации, на месте
Быстрая сортировка (Quick Sort)	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$	Неустойчивая
Сортировка слиянием (Merge Sort)	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	Устойчивая
Пирамидальная сортировка (Heap Sort)	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	Неустойчивая, на месте

«Пузырьковая» сортировка (Bubble Sort)

```
function BubbleSort(A[1:n], n)
    swapped = True
    while swapped do
        swapped = False
        for i = 2 to n do
            if A[i - 1] > A[i] then
                swap(A[i - 1], A[i])
                swapped = True
            end if
        end for
    end while
end function
```

$T_{BubbleSort} = O(n^2)$



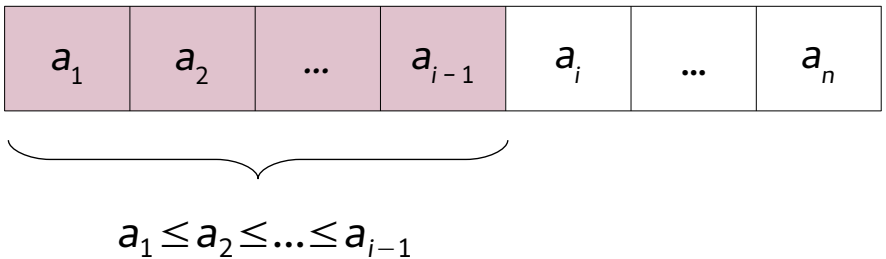
«Лёгкие» элементы перемещаются (всплывают) в начало массива

4
11
5
9
3
10
8
6

Сортировка вставками (Insertion Sort)

```
function InsertionSort(A[1:n], n)
    for i = 2 to n do
        key = A[i]
        j = i - 1
        while j > 0 and A[j] > key do
            A[j + 1] = A[j]
            j = j - 1
        end while
        A[j + 1] = key
    end for
end function
```

- Двигаемся по массиву слева направо: от 2-го до *n*-го элемента
- На шаге *i* имеем упорядоченный подмассив $A[1..i-1]$ и элемент $A[i]$, который необходимо вставить в этот подмассив



Сортировка вставками (Insertion Sort)

- **В худшем случае** цикл *while* всегда доходит до первого элемента массива — на вход поступил массив, упорядоченный по убыванию
- Для вставки элемента $A[i]$ на своё место требуется $i - 1$ итерация цикла *while*
- На каждой итерации выполняем c действий
- Учитывая, что необходимо найти позиции для $n - 1$ элемента, время $T(n)$ выполнения алгоритма в худшем случае равно

$$T(n) = \sum_{i=2}^{n} c(i-1) = c + 2c + \ldots + (i-1)c + \ldots + (n-1)c = \frac{cn(n-1)}{2} = \Theta(n^2)$$

Сортировка вставками (Insertion Sort)

- **Лучший случай для сортировки вставками?**

Сортировка вставками (Insertion Sort)

- **Лучший случай для сортировки вставками?**
- **Массив уже упорядочен**

- Алгоритм сортировки вставками является **устойчивым** — не меняет относительный порядок следования одинаковых ключей
- Используется константное число дополнительных ячеек памяти (переменные *i*, *key* и *j*), что относит его к классу алгоритмов сортировки **на месте** (in-place sort)
- Кроме того, алгоритм относится к классу **online-алгоритмов** — он обеспечивает возможность упорядочивания массивов при динамическом поступлении новых элементов

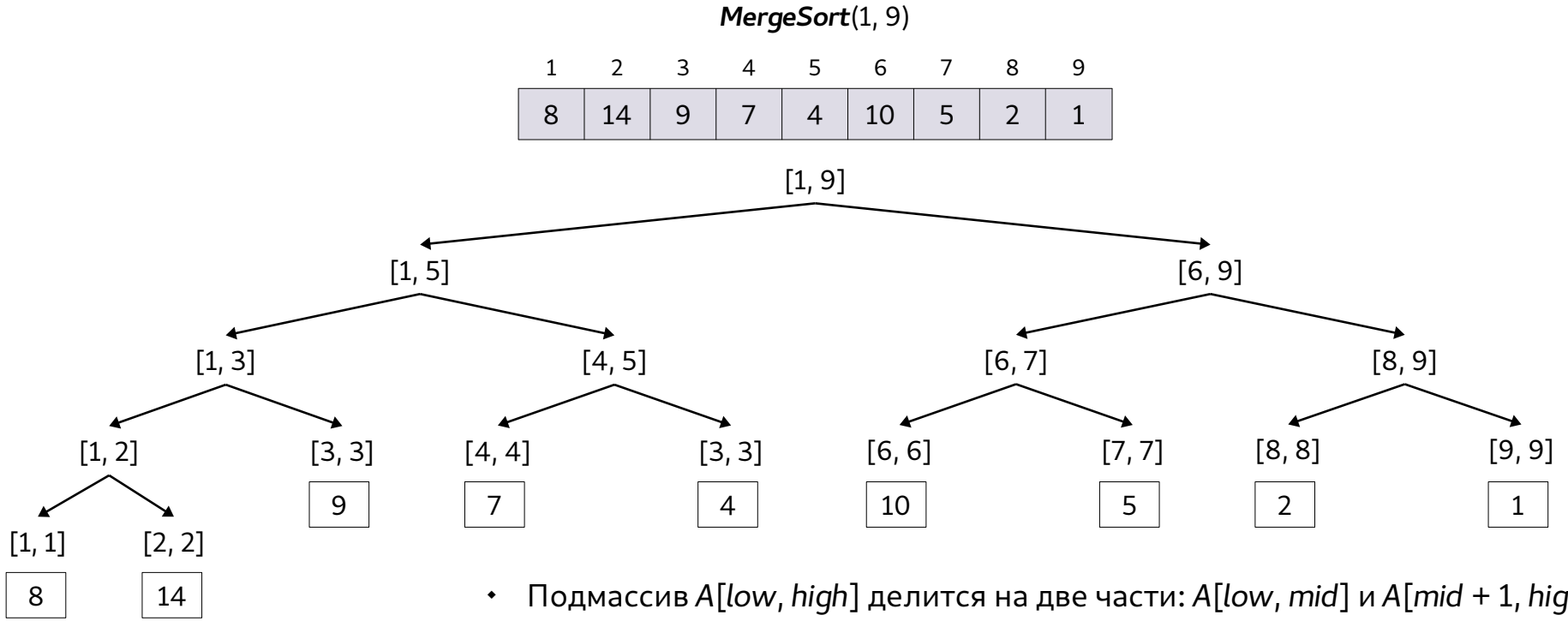
Сортировка слиянием (Merge Sort)

- **Сортировка слиянием** (merge sort) — асимптотически оптимальный алгоритм сортировки сравнением, основанный на методе *декомпозиции* («разделяй и властвуй», decomposition)
- Требуется упорядочить заданный массив $A[1..n]$ по неубыванию (non-decreasing order) так, чтобы

$$A[1] \le A[2] \le \ldots \le A[n]$$

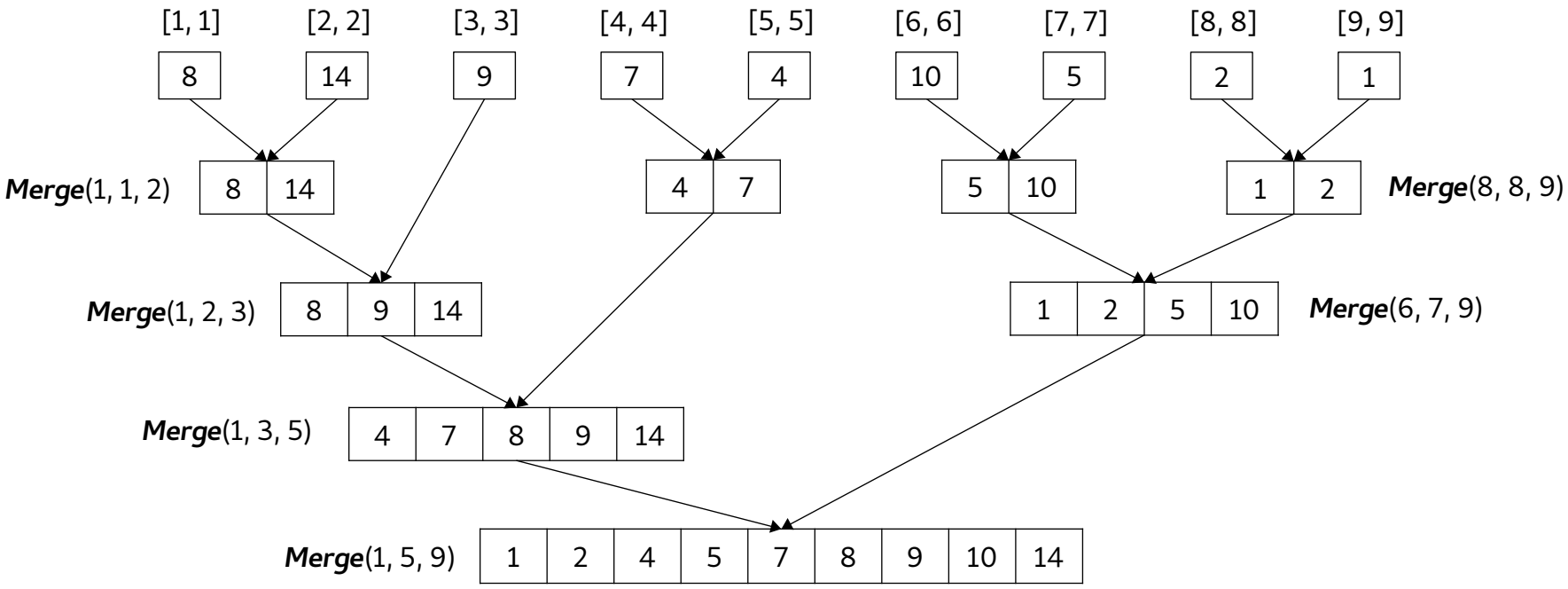
- Алгоритм включает две фазы:
  1. **Разделение** (partition) — рекурсивное разбиение массива на меньшие подмассивы, их сортировка
  2. **Слияние** (merge) — объединение упорядоченных массивов в один

Сортировка слиянием: фаза разделения



- Подмассив *A*[*low*, *high*] делится на две части: *A*[*low*, *mid*] и *A*[*mid* + 1, *high*]
- *mid* = ***floor***((*low* + *high*) / 2)

Сортировка слиянием: фаза слияния



Функция Merge сливает упорядоченные подмассивы $A[low..mid]$ и $A[mid + 1..high]$ в один отсортированный массив, элементы которого занимают позиции $A[low..high]$

Сортировка слиянием (Merge Sort)

```
function MergeSort(A[1:n], low, high)
    if low < high then
        mid = floor((low + high) / 2)
        MergeSort(A, low, mid)
        MergeSort(A, mid + 1, high)
        Merge(A, low, mid, high)
    end if
end function
```

- Сортируемый массив $A[low..high]$ *разделяется* (partition) на две максимально равные по длине части
- Левая часть содержит $\lceil n / 2 \rceil$ элементов, правая — $\lfloor n / 2 \rfloor$ элементов
- Подмассивы рекурсивно сортируются

Сортировка слиянием (Merge Sort)

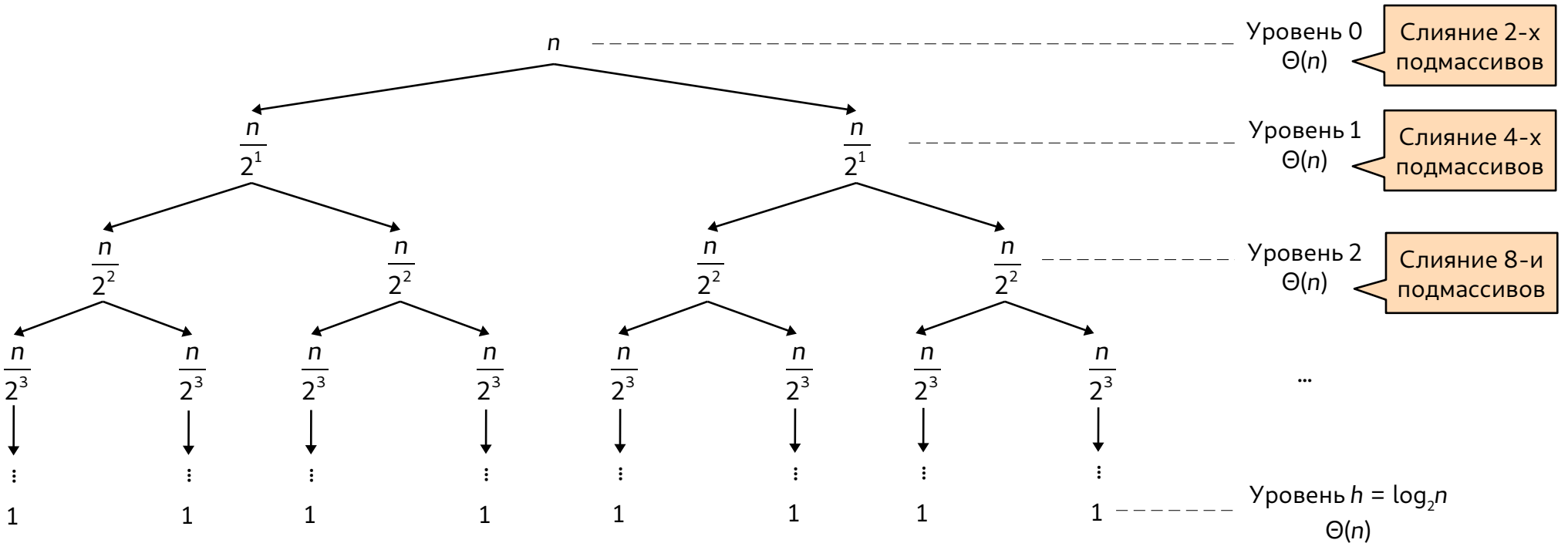
```
function Merge(A[1:n], low, mid, high)
    for i = low to high do
        B[i] = A[i]      /* Копия массива A */
    end for

    l = low       /* Начало левого подмассива */
    r = mid + 1   /* Начало правого подмассива */
    i = low
    while l <= mid and r <= high do
        if B[l] <= B[r] then
            A[i] = B[l]
            l = l + 1
        else
            A[i] = B[r]
            r = r + 1
        end if
        i = i + 1
    end while
    /* Копируем остатки подмассивов */
    while l <= mid do
        A[i] = B[l]
        l = l + 1
        i = i + 1
    end while
    while r <= high do
        A[i] = B[r]
        r = r + 1
        i = i + 1
    end while
end function
```

Сортировка слиянием (Merge Sort)

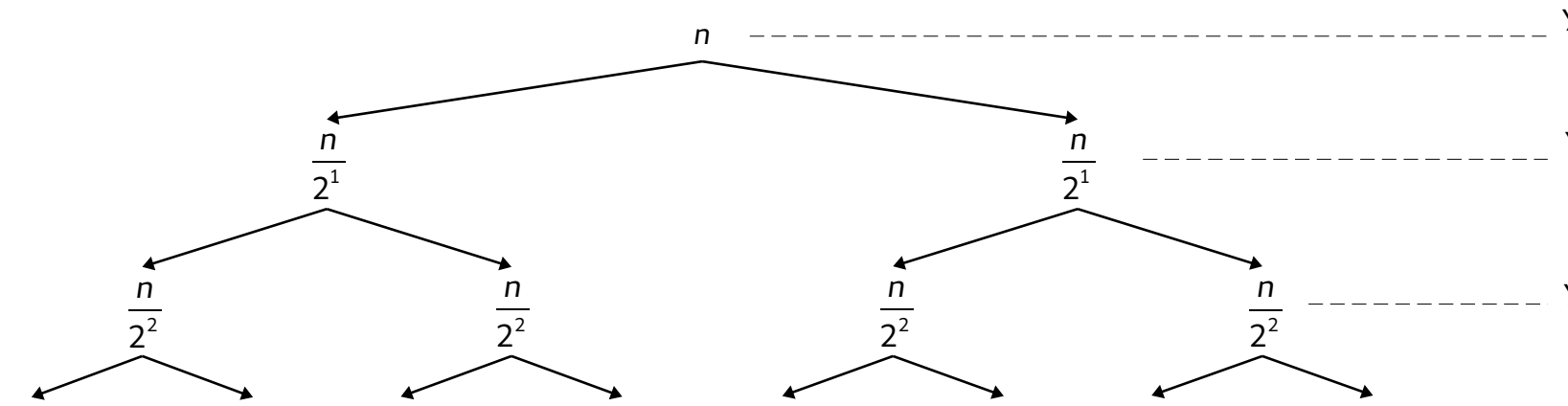
```
function Merge(A[1:n], low, mid, high)
    for i = low to high do
        B[i] = A[i]      /* Копия массива A */
    end for
        if B[l] <= B[r] then
            A[i] = B[l]
            l = l + 1
        else
            A[i] = B[r]
            r = r + 1
        end if
        i = i + 1
    end while
    /* Копируем остатки подмассивов */
    while l <= mid do
    while r <= high do
        A[i] = B[r]
        r = r + 1
        i = i + 1
    end while
end function
```

- Функция *Merge* требует порядка $\Theta(n)$ ячеек памяти для хранения копии B сортируемого массива
  - Сравнение и перенос элементов из массива B в массив A требует $\Theta(n)$

Дерево рекурсивных вызовов сортировки слиянием



Дерево рекурсивных вызовов сортировки слиянием



Уровень 0
$\Theta(n)$

Уровень 1
$\Theta(n)$

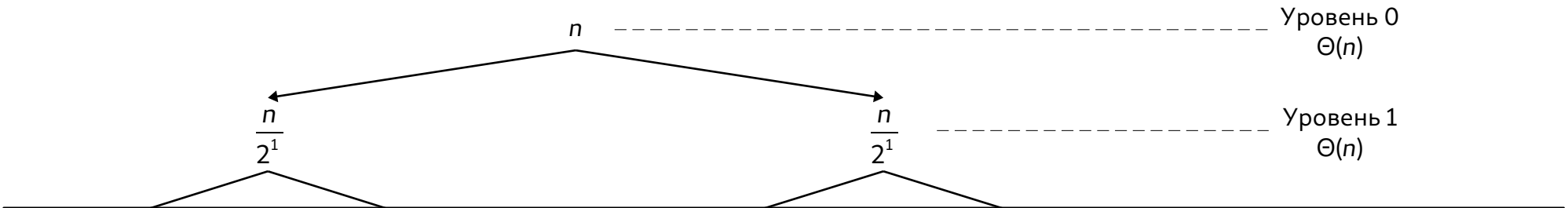
Уровень 2
$\Theta(n)$

...

- Высота дерева — $\Theta(\log n)$
- На каждом уровне i находится 2^i узлов
- Каждый узел требует выполнения $\frac{n}{2^i}$ операций

Уровень $h = \log_2 n$
$\Theta(n)$

Дерево рекурсивных вызовов сортировки слиянием



Уровень 0
Θ(*n*)

Уровень 1
Θ(*n*)

$$T(n)=\sum_{i=0}^{h}2^i\frac{n}{2^i}=\sum_{i=0}^{h}n=(h+1)n=n\log_2 n+n=\Theta(n\log n)$$

- Высота дерева — Θ(log*n*)
- На каждом уровне *i* находится 2^i узлов
- Каждый узел требует выполнения $\frac{n}{2^i}$ операций

...

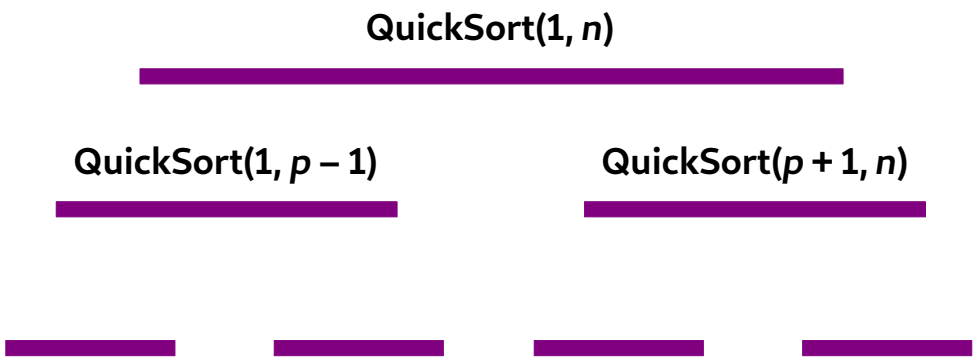
Уровень $h = \log_2 n$
Θ(*n*)

Быстрая сортировка (Quick Sort)

1. Из элементов $A[1], A[2], ..., A[n]$ выбирается **опорный элемент** (pivot element)
   - Опорный элемент желательно выбирать так, чтобы его значение было близко к среднему значению всех элементов массива
   - Вопрос о выборе опорного элемента открыт (первый, последний, средний из трёх, случайный, ...)
2. Массив разбивается на две части: элементы массива переставляются так, чтобы элементы, расположенные левее опорного, были не больше (≤), а расположенные правее — не меньше него (≥). На этом шаге определяется граница дальнейшего разбиения массива
3. Шаги 1 и 2 рекурсивно повторяются для левой и правой частей

Быстрая сортировка (Quick Sort)

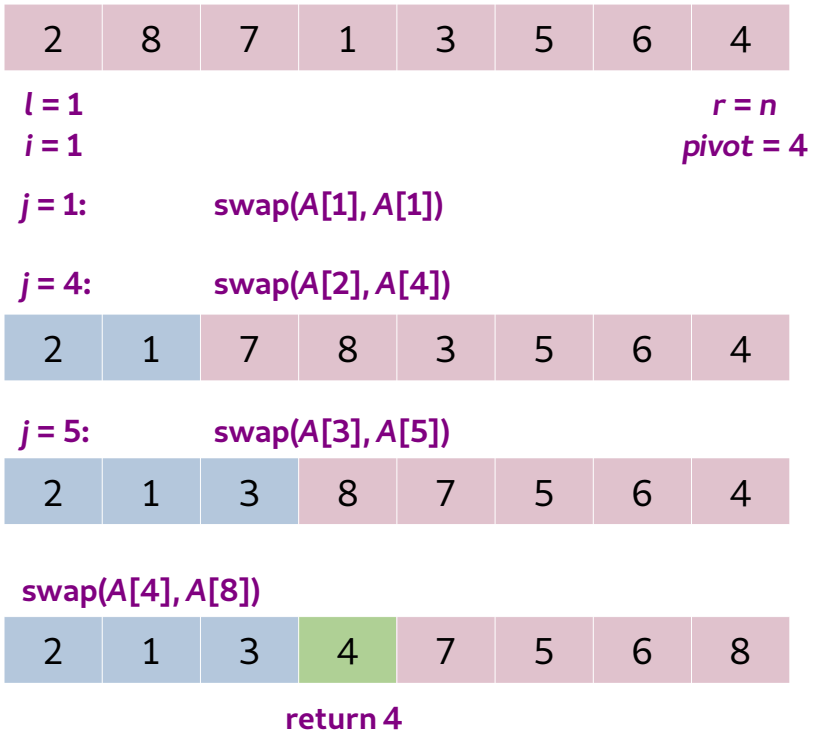
```
function QuickSort(A[1:n], low, high)
    if low < high then
        p = Partition(A, low, high)
        QuickSort(A, low, p - 1)
        QuickSort(A, p + 1, high)
    end if
end function
```



- Средний случай: $T_{QuickSort} = O(n \log n)$
- Худший случай: $T_{QuickSort} = O(n^2)$

Быстрая сортировка (Quick Sort)

```
function Partition(A[1:n], low, high)
    pivot = A[high]
    i = low
    for j = low to high - 1 do
        if A[j] < pivot then
            swap(A[i], A[j])
            i = i + 1
        end if
    end for
    swap(A[i], A[high])
    return i
end function
```



Быстрая сортировка (Quick Sort)

- **Худший случай для быстрой сортировки?**

Быстрая сортировка (Quick Sort)

- **Худший случай для быстрой сортировки?**
- **В качестве опорного элемента выбирается наименьший или наибольший ключ**

Сортировка подсчётом (Counting Sort)

```
function CountingSort(A[0:n - 1], B[0:n - 1], k)
    for i = 0 to k do
        C[i] = 0
    end for
    for i = 0 to n - 1 do
        C[A[i]] = C[A[i]] + 1
    end for
    for i = 1 to k do
        C[i] = C[i] + C[i - 1]
    end for
    for i = n - 1 to 0 do
        C[A[i]] = C[A[i]] - 1
        B[C[A[i]]] = A[i]
    end for
end function
```

$A[0..6]$	4	1	0	1	7	5	3

$k = \max(A[0..6]) = 7$

$C[0..k]$	0	1	2	3	4	5	6	7
	1	**2**	**0**	**1**	**1**	**1**	**0**	**1**

$C[0..k]$	0	1	2	3	4	5	6	7
	1	**3**	**3**	**4**	**5**	**6**	**6**	**7**

$B[0..6]$	0	1	1	3	4	5	7

- Не использует операцию сравнения
- Целочисленная сортировка (integer sort)
- Вычислительная сложность: $O(n + k)$
- Сложность по памяти: $O(n + k)$

Домашнее чтение

- **[DSABook] Глава 3. Сортировка**
- **[Aho, C. 228—247] Глава 8. Сортировка** — разделы 8.1—8.4 (простые схемы сортировки, Quick Sort, Heap Sort)
- **[Levitin, C. 169] Сортировка слиянием**

ご清聴ありがとうございました!

Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: sillyhat34@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Весенний семестр, 2021 г.