

Лекция 3

Алгоритмы сортировки

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com

WWW: www.mkurnosov.net

Курс «Структуры и алгоритмы обработки данных»

Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)

Весенний семестр, 2016

Задача сортировки (sorting problem)

- Дана последовательность из n ключей

$$a_1, a_2, \dots, a_n$$

- Требуется упорядочить ключи по неубыванию или по невозрастанию – найти перестановку (i_1, i_2, \dots, i_n) ключей

- По неубыванию (non-decreasing order)

$$a_{i_1} \leq a_{i_2} \leq \dots \leq a_{i_n}$$

- По невозрастанию (non-increasing order)

$$a_{i_1} \geq a_{i_2} \geq \dots \geq a_{i_n}$$

Алгоритмы сортировки (Sorting)

- 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 называются **сортировкой сравнением** (comparison sort)

Алгоритмы сортировки (Sorting)

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

(Жираф, 800), (Слон, 1500), (Лиса, 40), (Волк, 90),
(Кит, 3000), (Барсук, 40)

Неустойчивая сортировка:

(Барсук, 40), (Лиса, 40), (Волк, 90), (Жираф, 800), (Слон, 1500),
(Кит, 3000) – порядок не соблюден

Устойчивая сортировки:

(Лиса, 40), (Барсук, 40), (Волк, 90), (Жираф, 800), (Слон, 1500),
(Кит, 3000) – порядок соблюден

Алгоритмы сортировки (Sorting)

- **Внутренние методы сортировки (Internal sort)** – сортируемые элементы полностью размещены в оперативной памяти компьютера
- **Внешняя сортировка (External sort)** – элементы размещены на внешней памяти (жесткий диск, USB-флеш)
- Алгоритм сортировки не использующий дополнительной памяти (кроме сортируемого массива) называется алгоритмом **сортировки на месте** (in-place sort)

Алгоритмы сортировки (Sorting)

- **Алгоритмы на основе сравнений (comparison sort):**
Insertion sort, Bubble sort, selection sort, Shell Sort, QuickSort, MergeSort, HeapSort и др.
- **Алгоритмы не основанные на сравнениях:**
Counting sort, radix sort – используют структура 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).

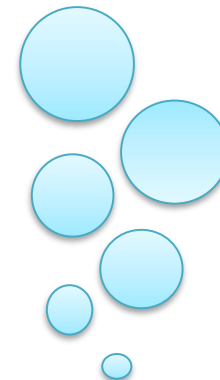
Вычислительная сложность сортировки

Алгоритм	Лучший случай	Средний случай	Худший случай	Память	Свойства
Сортировка вставками (Insertion Sort)	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Устойчивый, 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)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	На месте, неустойчивый

“Пузырьковая” сортировка (Bubble Sort)

```
function BubbleSort(v[0:n - 1], n)
  swapped = True
  while swapped do
    swapped = False
    for i = 1 to n - 1 do
      if v[i - 1] > v[i] then
        swap(v[i - 1], v[i])
        swapped = True
      end if
    end for
  end while
end function
```

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

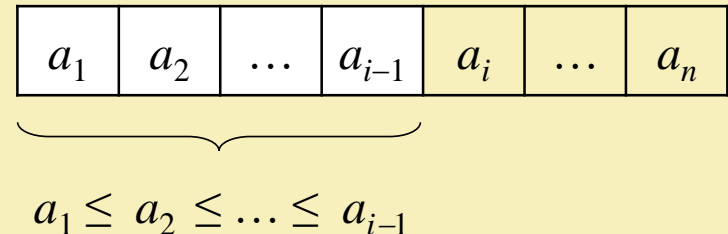


9
5
54
23
45
2
4
4
42
34

“Легкие” элемент перемещаются
(всплывают) в начало массива

Сортировка вставками (Insertion Sort)

```
function InsertionSort(A[1:n], n)
  for i = 2 to n do
    key = A[i]
    /* Вставляем A[i] в упорядоченный подмассив A[1..i-1] */
    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 нам необходимо найти позиции для $n - 1$ элемента, время $T(n)$ выполнения алгоритма в худшем случае равно

$$\begin{aligned} T(n) &= \sum_{i=2}^n c(i-1) = c + 2c + \dots + (i-1)c + \dots + (n-1)c = \\ &= \frac{cn(n-1)}{2} = \Theta(n^2). \end{aligned}$$

Сортировка вставками (Insertion Sort)

- Лучший случай для Insertion sort?

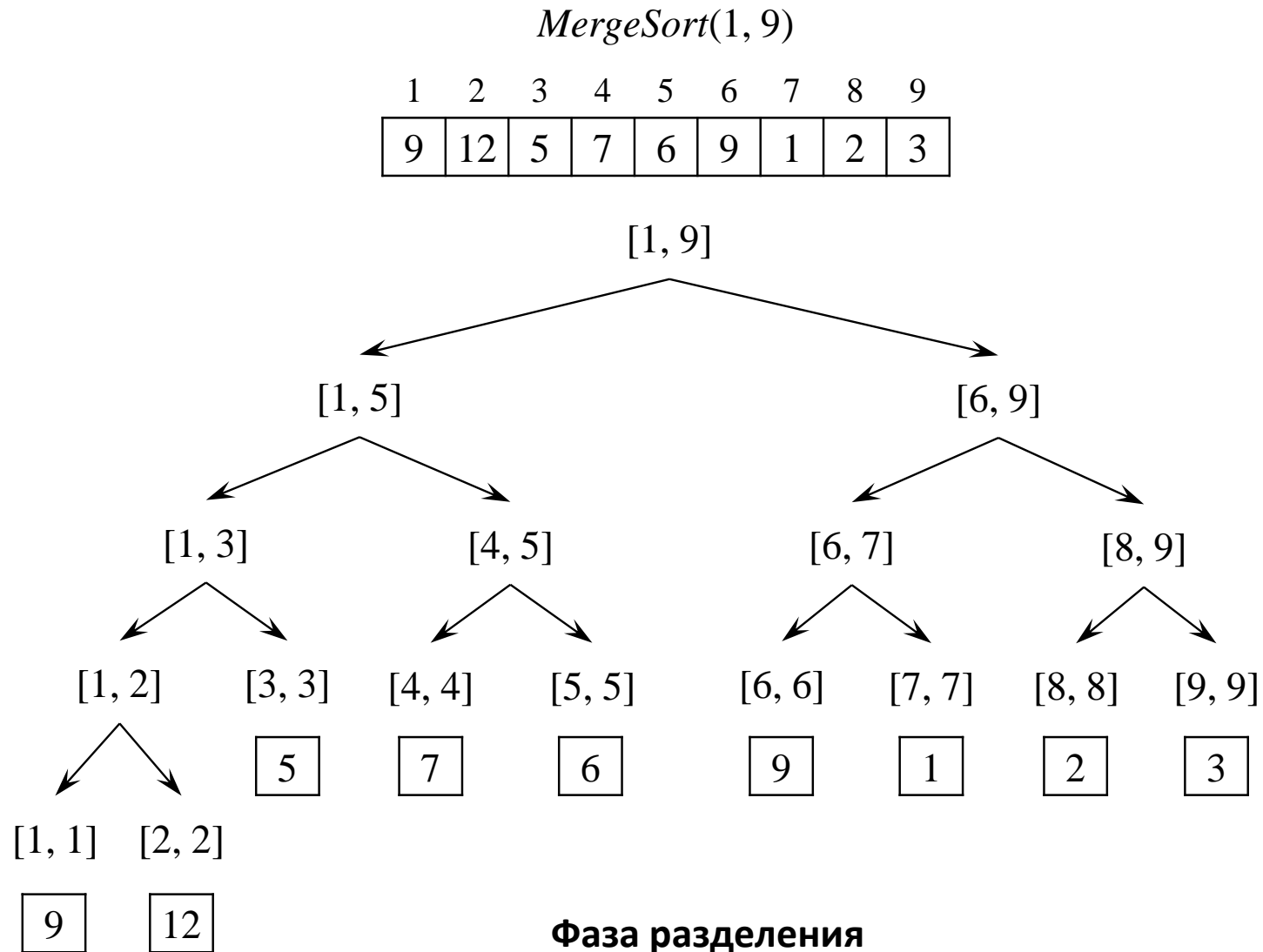
Сортировка вставками (Insertion Sort)

- Лучший случай для Insertion sort?
- Массив уже упорядочен
- Алгоритм сортировки вставкой является **устойчивым** – не меняет относительный порядок следования одинаковых ключей
- Использует 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)

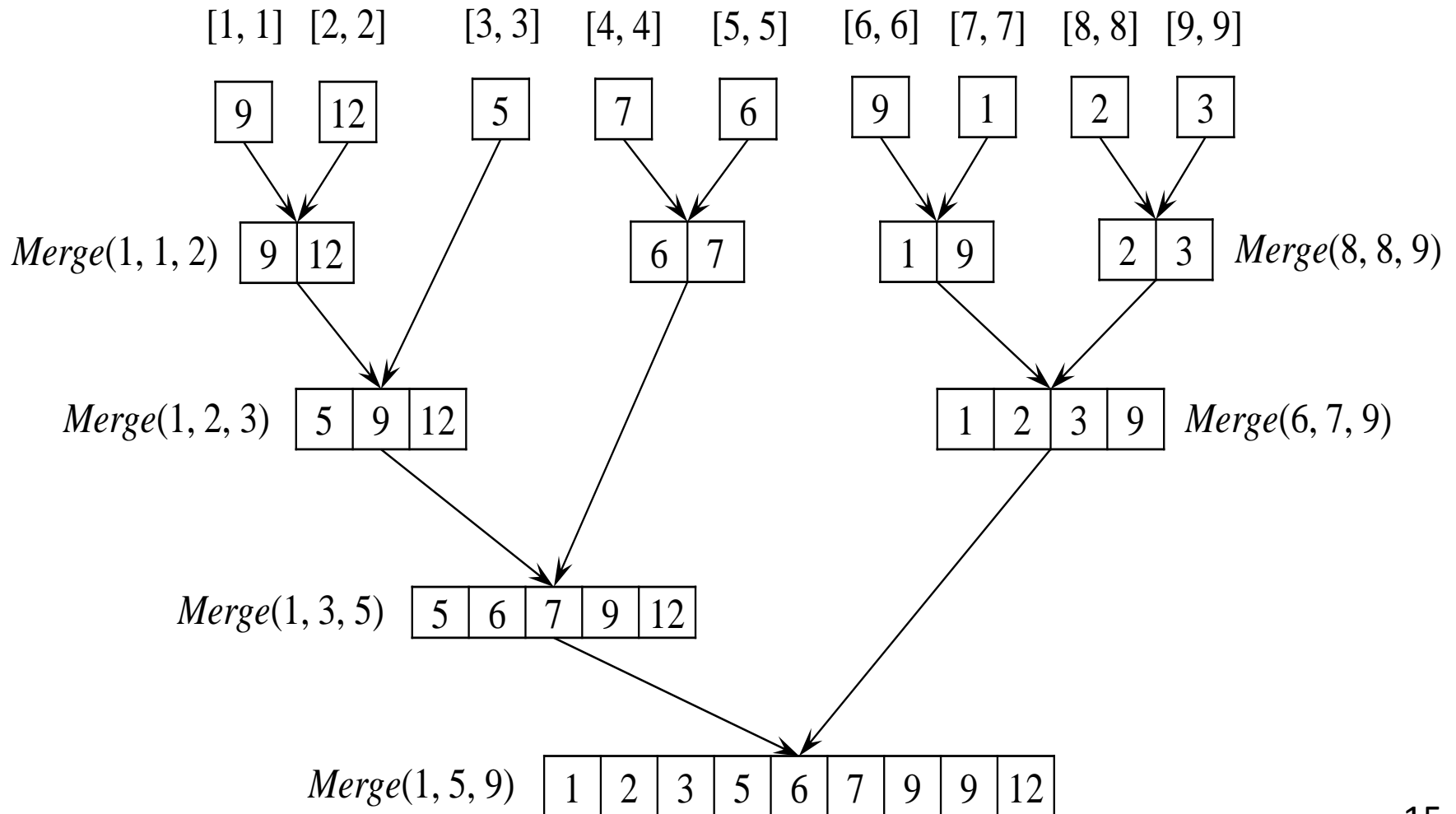
- **Сортировка слиянием** (merge sort) – рекурсивный алгоритм сортировки сравнением, основанный на методе декомпозиции (decomposition)

Сортировка слиянием (merge sort)



Сортировка слиянием (merge sort)

Фаза слияния



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
- Первая часть содержит $\lfloor n/2 \rfloor$ элементов, вторая – $\lfloor n/2 \rfloor$ 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]           /* Создаем копию массива A */
  end for

  l = low                 /* Номер первого элемента левого подмассива */
  r = mid + 1             /* Номер первого элемента 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
```

Сортировка слиянием (Merge Sort)

```
while l <= mid do
    /* Копируем оставшиеся элементы из левого подмассива */
    A[i] = B[l]
    l = l + 1
    i = i + 1
end while
while r <= high do
    /* Копируем оставшиеся элементы из правого подмассива */
    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 $\Theta(n)$

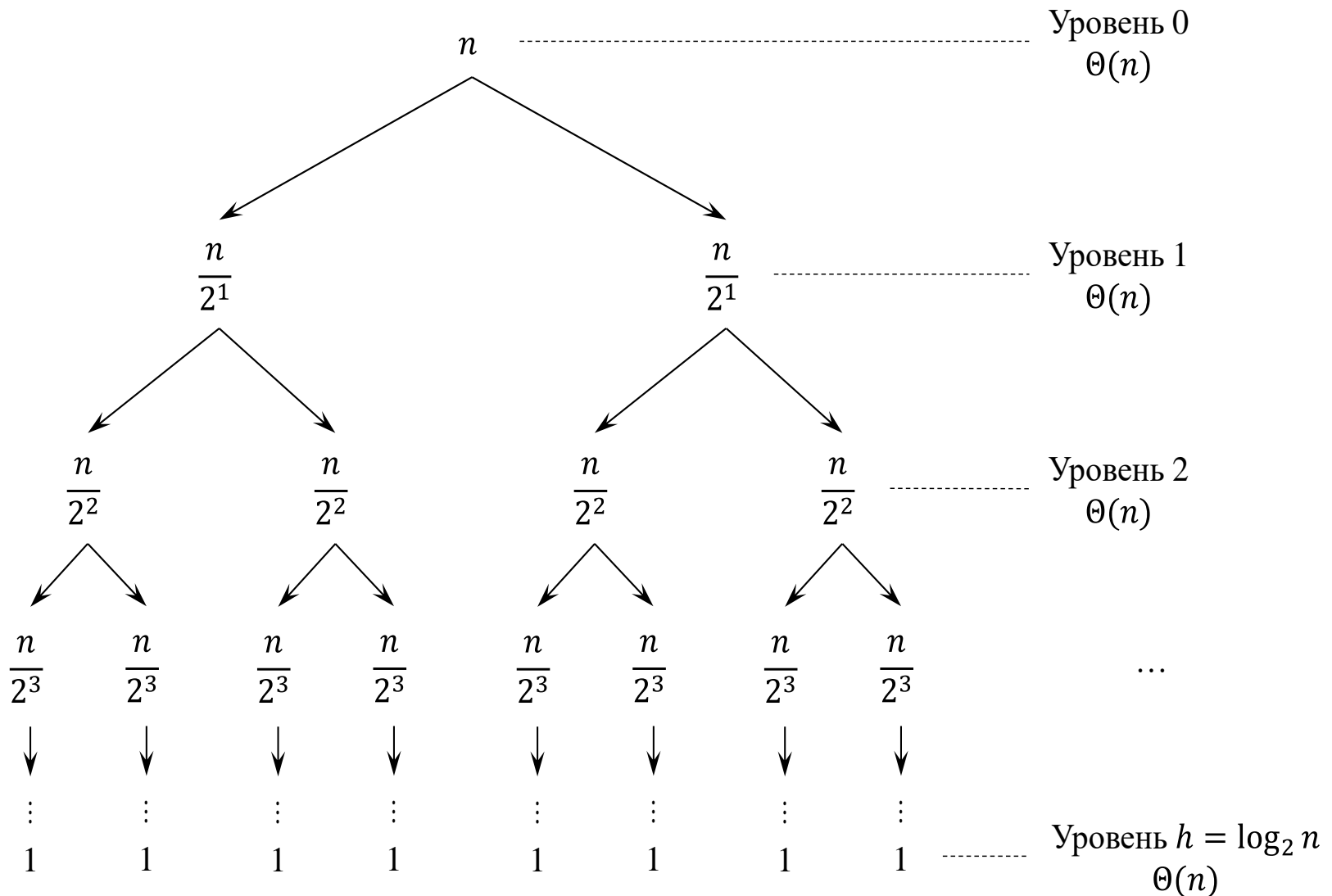
Сортировка слиянием (Merge Sort)

- Время $T(n)$ работы алгоритма включает время сортировки левого подмассивов длины $\lceil n/2 \rceil$ и правого – с числом элементов $\lfloor n/2 \rfloor$, а также время $\Theta(n)$ слияния подмассивов после их рекурсивного упорядочивания

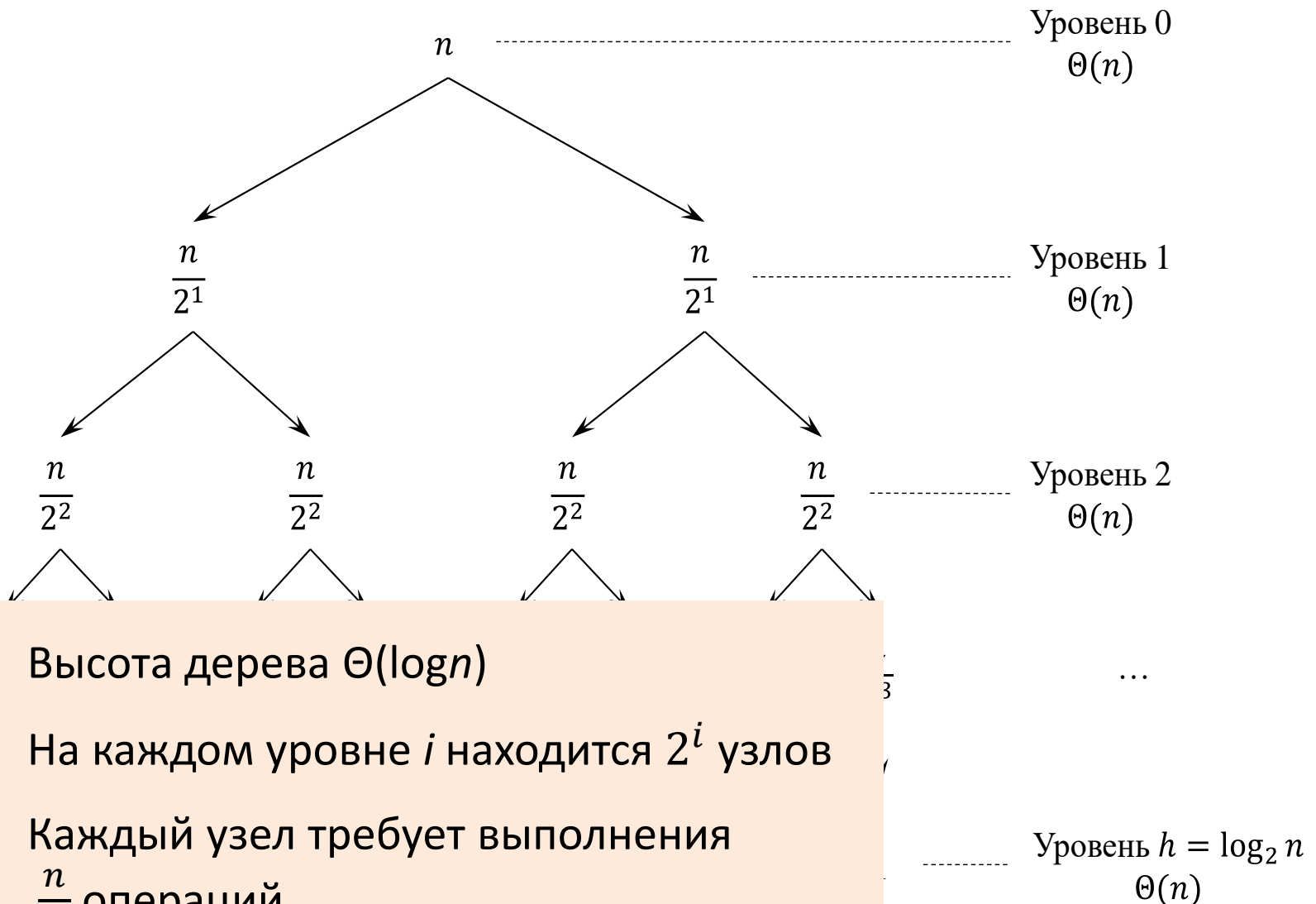
$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$$

- Необходимо решить это рекуррентное уравнение – получить выражение для $T(n)$ без рекуррентности

Дерево рекурсивных вызовов MergeSort

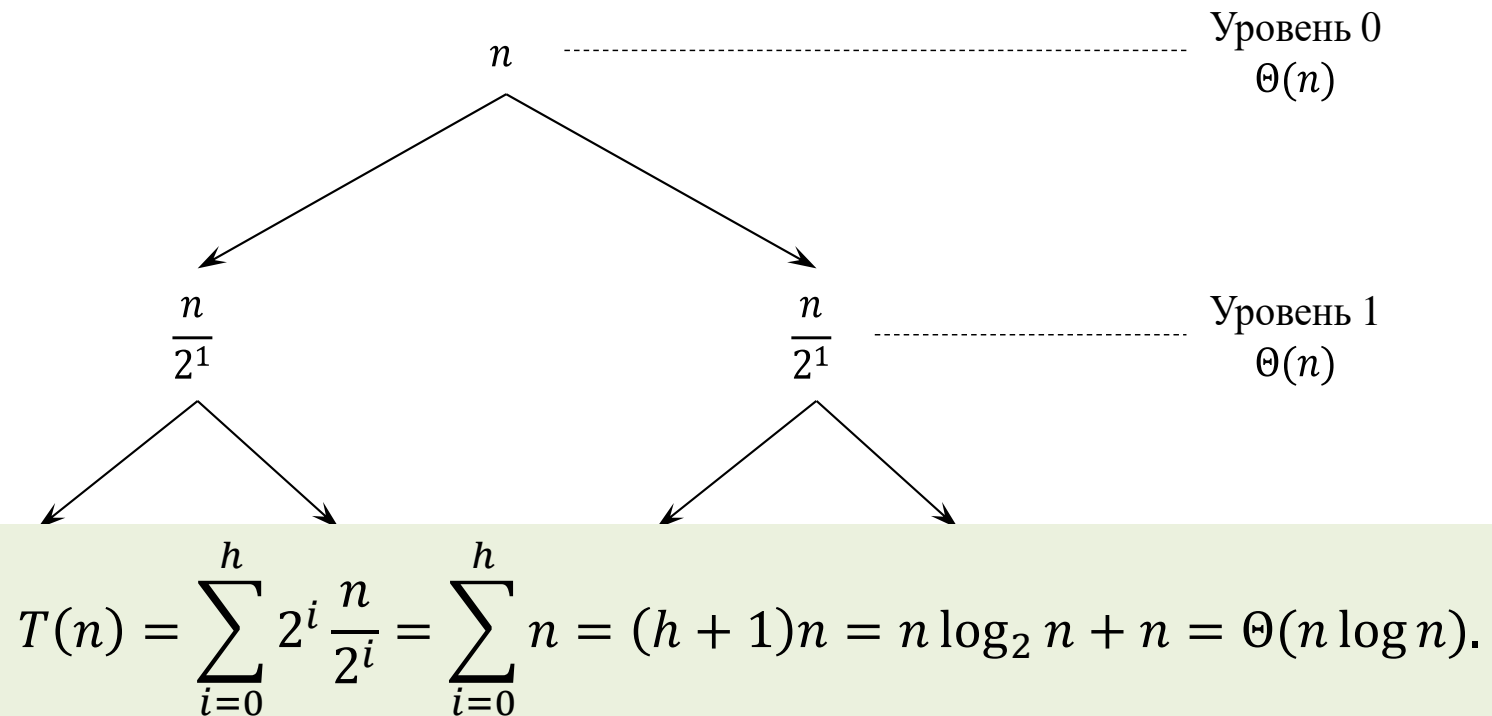


Дерево рекурсивных вызовов MergeSort



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

Дерево рекурсивных вызовов MergeSort



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

...

Уровень $h = \log_2 n$
 $\Theta(n)$

Быстрая сортировка (Quick Sort)

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

Быстрая сортировка (Quick Sort)

```
function QuickSort(v[1:n], n, l, r)
  if l < r then
    k = Partition(v, n, l, r)      // Разбиваем
    QuickSort(v, n, l, k - 1)     // Левая часть
    QuickSort(v, n, k + 1, r)     // Правая часть
  end if
end function
```

QuickSort(1, n)

QuickSort(l, k - 1)

QuickSort(k + 1, r)

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

Быстрая сортировка (Quick Sort)

```
function Partition(v[1:n], n, l, r)
```

```
    pivot_idx = r          /* Выбрали индекс опорного элемента */
```

```
    swap(v[pivot_idx], v[r])
```

```
    pivot = v[r]
```

```
    i = l - 1
```

```
    for j = l to r - 1 do
```

```
        if v[j] <= pivot then
```

```
            i = i + 1
```

```
            swap(v[i], v[j])
```

```
        end if
```

```
    end for
```

```
    swap(v[i + 1], v[r])
```

```
    return i + 1
```

```
end function
```

2	8	7	1	3	5	6	4
---	---	---	---	---	---	---	---

$l = 1$

$r = n$

$i = 0$

$\text{pivot} = 4$

$j = 1$: swap(v[1], v[1])

$j = 4$: swap(v[2], v[4])

2	1	7	8	3	5	6	4
---	---	---	---	---	---	---	---

$j = 5$: swap(v[3], v[5])

2	1	3	8	7	5	6	4
---	---	---	---	---	---	---	---

swap(v[4], v[8])

2	1	3	4	7	5	6	8
---	---	---	---	---	---	---	---

return 4

Быстрая сортировка (Quick Sort)

- Худший случай для QuickSort?

Быстрая сортировка (Quick Sort)

- Худший случай для QuickSort?
- В качестве опорного элемента выбрали наименьший или наибольший ключ

Сортировка подсчетом (Counting Sort)

Алгоритм 3.4. Сортировка подсчетом

```
1 function COUNTINGSORT( $A[0..n-1], B[0..n-1], k$ )
2   for  $i = 0$  to  $k$  do
3      $C[i] = 0$ 
4   end for
5   for  $i = 0$  to  $n-1$  do
6      $C[A[i]] = C[A[i]] + 1$ 
7   end for
8   for  $i = 1$  to  $k$  do
9      $C[i] = C[i] + C[i-1]$ 
10  end for
11  for  $i = n-1$  to  $0$  do
12     $C[A[i]] = C[A[i]] - 1$ 
13     $B[C[A[i]]] = A[i]$ 
14  end for
15 end function
```

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

$A[0..6]$	$C[0..k]$		$C[0..k]$		$B[0..6]$
	0	1	0	1	
4	1	2	1	3	1
1	2	0	2	3	1
0	3	1	3	4	3
1	4	1	4	5	4
7	5	1	5	6	5
5	6	0	6	6	7
3	7	1	7	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]
- [Aho, С. 228-247]: **Глава 8. Сортировка** – разделы 8.1 – 8.4 (простые схемы сортировки, QuickSort, HeapSort)
- [Levitin, С. 169]: **Сортировка слиянием**