Лекция 4. Алгоритмы сортировки

E-mail: lazycoyote11@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Весенний семестр, 2026 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 $a \leq b$, 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*)

(76, Лис), (34, Тигр), (29, Единорог), (76, Кот), (95, Дракон)

Неустойчивая сортировка:

(95, Дракон), (76, Кот), (76, Лис) (34, Тигр), (29, Единорог)

порядок *не соблюден*

Устойчивая сортировка:

(95, Дракон), (76, Лис), (76, Кот), (34, Тигр), (29, Единорог)

порядок *соблюден*

Алгоритмы сортировки

- **Внутренняя сортировка** (*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

- **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
Сортировка подсчетом (Counting Sort)	$O(k + n)$	$O(k + n)$	$O(k + n)$	$O(k + n)$	Устойчивая, целочисленная

«Пузырьковая» сортировка (Bubble Sort)

```
1  function BubbleSort(A[1:n], n)
2      swapped = True
3      while swapped do
4          swapped = False
5          for i = 2 to n do
6              if A[i - 1] > A[i] then
7                  swap(A[i - 1], A[i])
8                  swapped = True
9              end if
10         end for
11     end while
12 end function
```

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

«Легкие» элементы перемещаются (всплывают) в начало массива

«Пузырьковая» сортировка (Bubble Sort)

```
1   function BubbleSort(A[1:n], n)
2       swapped = True
3       while swapped do
4           swapped = False
5           for i = 2 to n do
6               if A[i - 1] > A[i] then
7                   swap(A[i - 1], A[i])
8                   swapped = True
9               end if
10          end for
11      end while
12  end function
```

Сортировка массива из 5 элементов:

32 65 21 19 28

```
32 65 21 19 28
i = 2 | swap: 0
32 65 21 19 28
i = 3 | swap: 1
32 21 65 19 28
i = 4 | swap: 1
32 21 19 65 28
i = 5 | swap: 1
32 21 19 28 65
WHILE COUNT: 1
i = 2 | swap: 1
21 32 19 28 65
i = 3 | swap: 1
21 19 32 28 65
i = 4 | swap: 1
21 19 28 32 65
i = 5 | swap: 1
21 19 28 32 65
WHILE COUNT: 2
```

```
i = 2 | swap: 1
19 21 28 32 65
i = 3 | swap: 1
19 21 28 32 65
i = 4 | swap: 1
19 21 28 32 65
i = 5 | swap: 1
19 21 28 32 65
WHILE COUNT: 3
i = 2 | swap: 0
19 21 28 32 65
i = 3 | swap: 0
19 21 28 32 65
i = 4 | swap: 0
19 21 28 32 65
i = 5 | swap: 0
19 21 28 32 65
WHILE COUNT: 4
19 21 28 32 65
```

Сортировка вставками (Insertion Sort)

```
1  function InsertionSort(A[1:n], n)
2      for i = 2 to n do
3          key = A[i]
4          j = i - 1
5          while j > 0 and A[j] > key do
6              A[j + 1] = A[j]
7              j = j - 1
8          end while
9          A[j + 1] = key
10     end for
11 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)

```
1  function InsertionSort(A[1:n], n)
2      for i = 2 to n do
3          key = A[i]
4          j = i - 1
5          while j > 0 and A[j] > key do
6              A[j + 1] = A[j]
7              j = j - 1
8          end while
9          A[j + 1] = key
10     end for
11 end function
```

Сортировка массива из 5 элементов:

32 65 21 19 28

```
32 65 21 19 28
STEP: 1, KEY: 65
32 65 21 19 28
STEP: 2, KEY: 21
j = 2
32 65 65 19 28
j = 1
32 32 65 19 28
21 32 65 19 28
STEP: 3, KEY: 19
j = 3
21 32 65 65 28
j = 2
21 32 32 65 28
j = 1
21 21 32 65 28
19 21 32 65 28
STEP: 4, KEY: 28
j = 4
19 21 32 65 65
j = 3
19 21 32 32 65
```

19 21 28 32 65

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 свое 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 выбором (Selection Sort)

```
1  function SelectionSort(A[1:n], n)
2      for i = 1 to n – 1 do
3          minindex = i
4          for j = i + 1 to n do
5              if A[j] < A[minindex] then
6                  minindex = j
7              end if
8          end for
9          if minindex != i then
10             temp = A[i]
11             A[i] = A[minindex]
12             A[minindex] = temp
13         end if
14     end for
15 end function
```

- Ищем индекс минимального ключа (*minindex*)
- Ставим найденный ключ на место
- Сколько памяти требуется?

Сортировка выбором (Selection Sort)

```
1  function SelectionSort(A[1:n], n)
2      for i = 1 to n – 1 do
3          minindex = i
4          for j = i + 1 to n do
5              if A[j] < A[minindex] then
6                  minindex = j
7              end if
8          end for
9          if minindex != i then
10             temp = A[i]
11             A[i] = A[minindex]
12             A[minindex] = temp
13         end if
14     end for
15 end function
```

- Ищем индекс минимального ключа (*minindex*)
- Ставим найденный ключ на место
- Сколько памяти требуется?

– Используется константное количество памяти (*i*, *j*, *minindex*), следовательно, это сортировка «на месте»

Сортировка выбором (Selection Sort)

```
1  function SelectionSort(A[1:n], n)
2      for i = 1 to n – 1 do
3          minindex = i
4          for j = i + 1 to n do
5              if A[j] < A[minindex] then
6                  minindex = j
7              end if
8          end for
9          if minindex != i then
10             temp = A[i]
11             A[i] = A[minindex]
12             A[minindex] = temp
13         end if
14     end for
15 end function
```

Сортировка массива из 5 элементов:

32 65 21 19 28

32 65 21 19 28

minindex: 4

19 65 21 32 28

minindex: 3

19 21 65 32 28

minindex: 5

19 21 28 32 65

minindex: 4

19 21 28 32 65

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

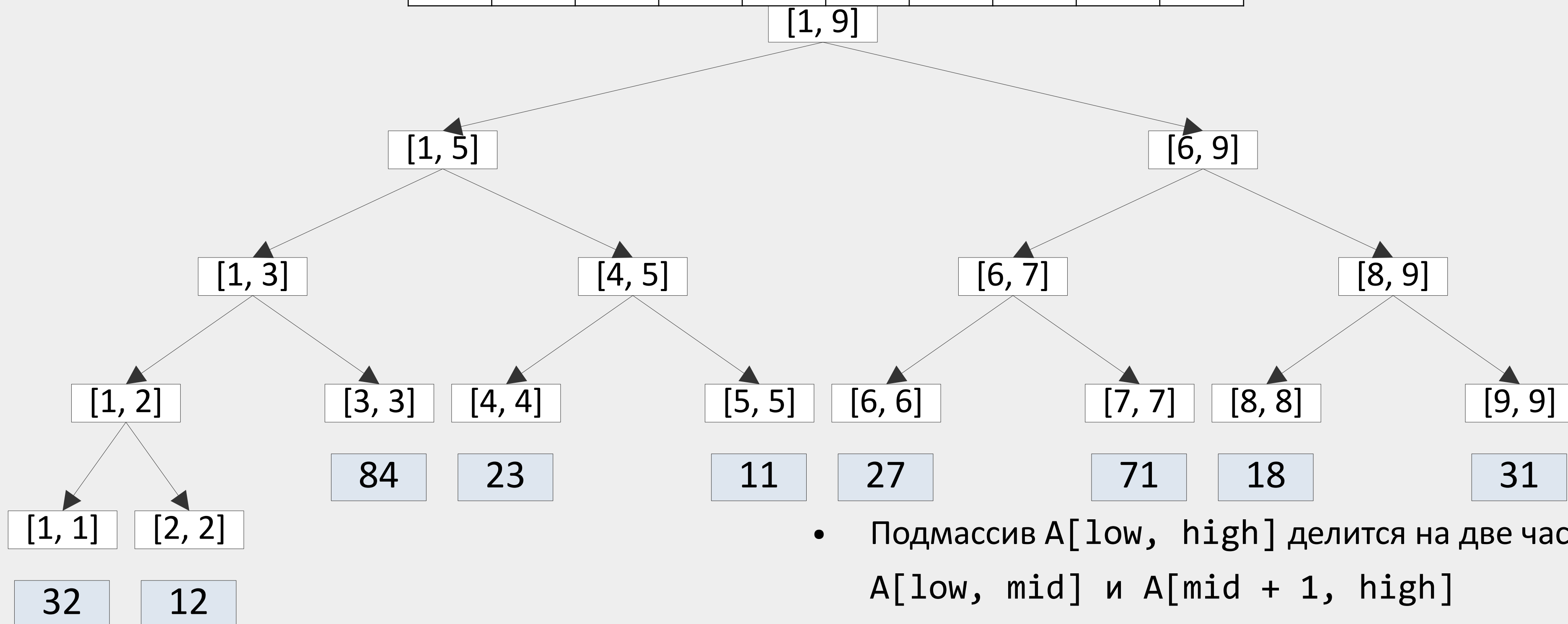
$$A[1] \leq A[2] \leq \ldots \leq A[n]$$

- 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

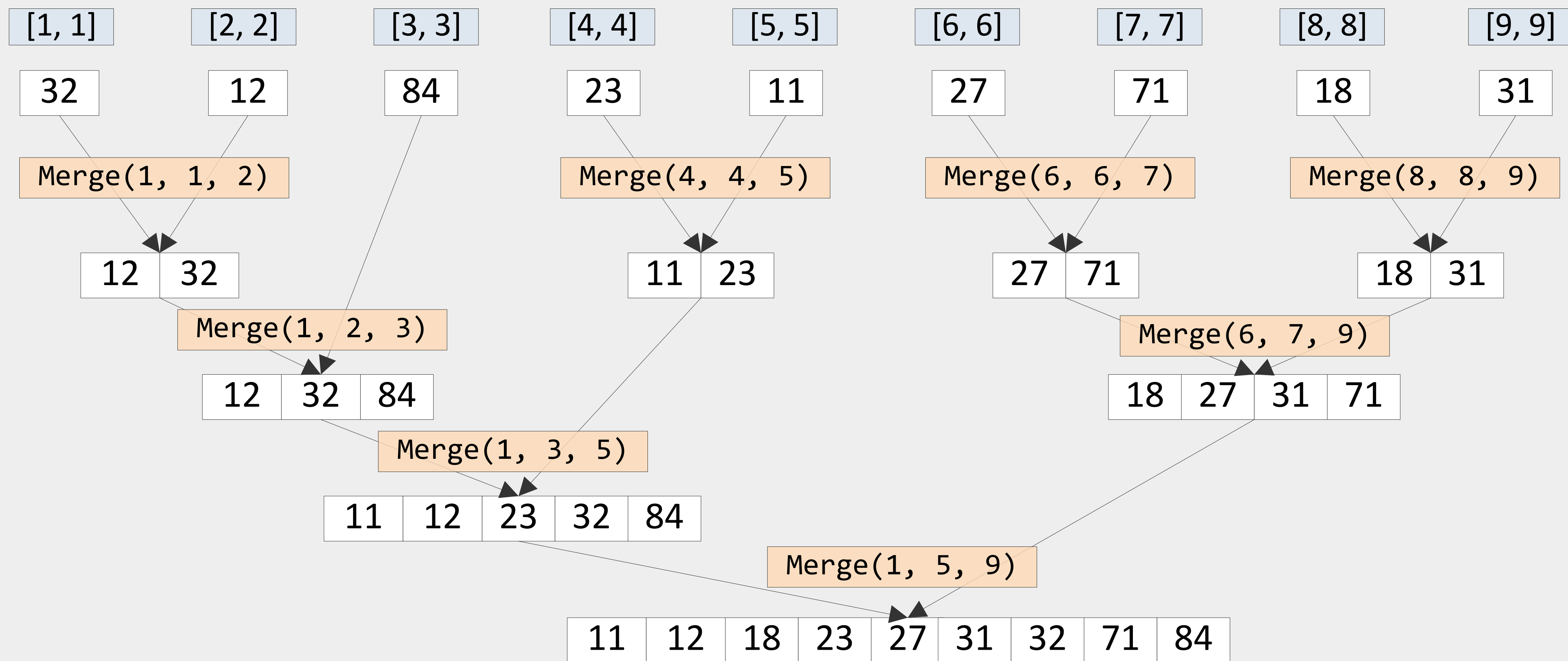
MergeSort(1, 9):

$a[i]$	32	12	84	23	11	27	71	18	31
i	1	2	3	4	5	6	7	8	9



- 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)

```
1 function MergeSort(A[1:n], low, high)
2     if low < high then
3         mid = floor((low + high) / 2)
4         MergeSort(A, low, mid)
5         MergeSort(A, mid + 1, high)
6         Merge(A, low, mid, high)
7     end if
8 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 — $\lfloor n / 2$ элементов$\rfloor$
- Подмассивы рекурсивно сортируются

Сортировка слиянием (Merge Sort)

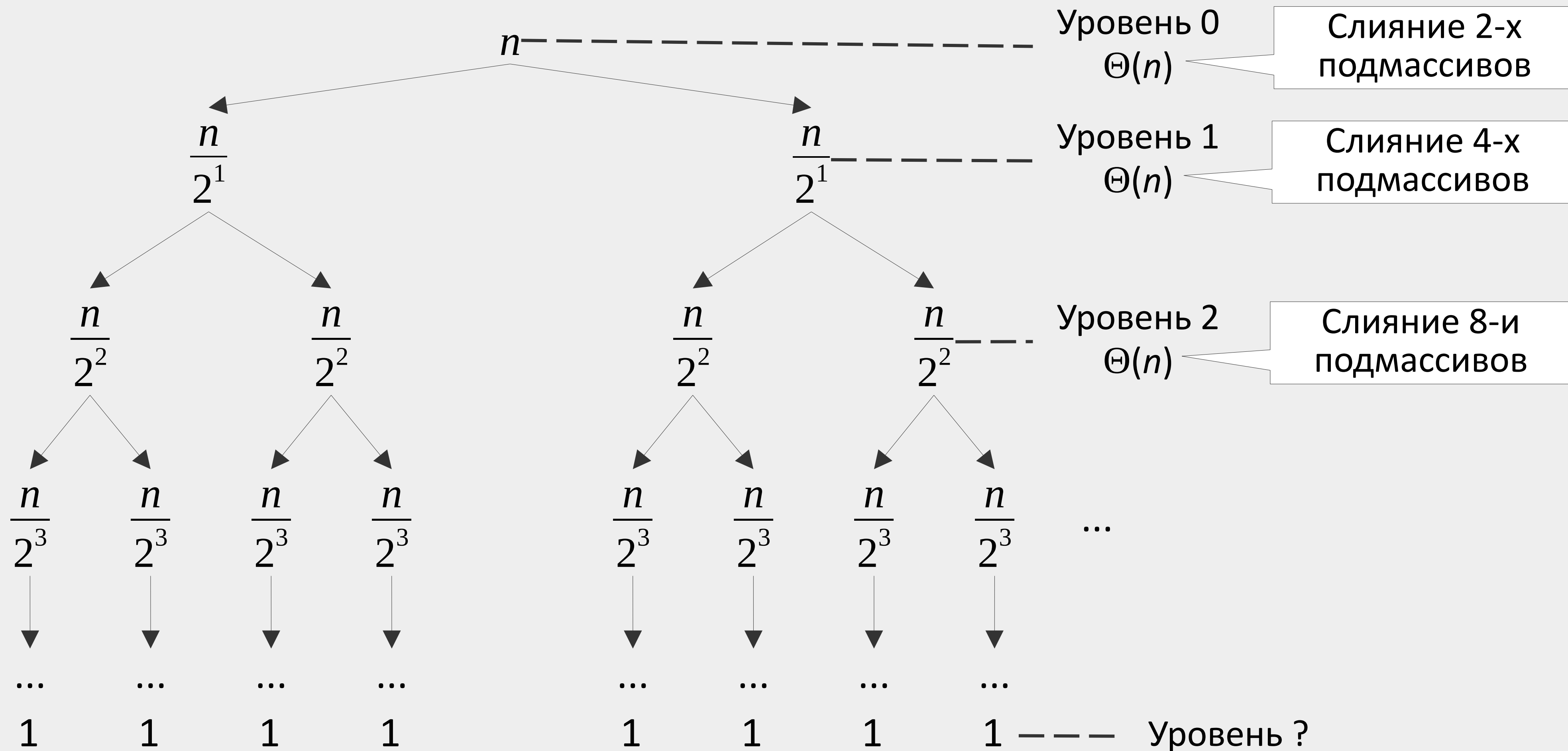
```
1   function Merge(A[1:n], low, mid, high)
2     for i = low to high do
3       B[i] = A[i] /* Копия массива A */
4     end for
5     l = low /* Начало левого подмассива */
6     r = mid + 1 /* Начало правого подмассива */
7     i = low
8     while l <= mid and r <= high do
9       if B[l] <= B[r] then
10        A[i] = B[l]
11        l = l + 1
12      else
13        A[i] = B[r]
14        r = r + 1
15      end if
16      i = i + 1
17    end while
18  /* Копируем остатки подмассивов */
19    while l <= mid do
20      A[i] = B[l]
21      l = l + 1
22      i = i + 1
23    end while
24    while r <= high do
25      A[i] = B[r]
26      r = r + 1
27      i = i + 1
28    end while
29  end function
```

Сортировка слиянием (Merge Sort)

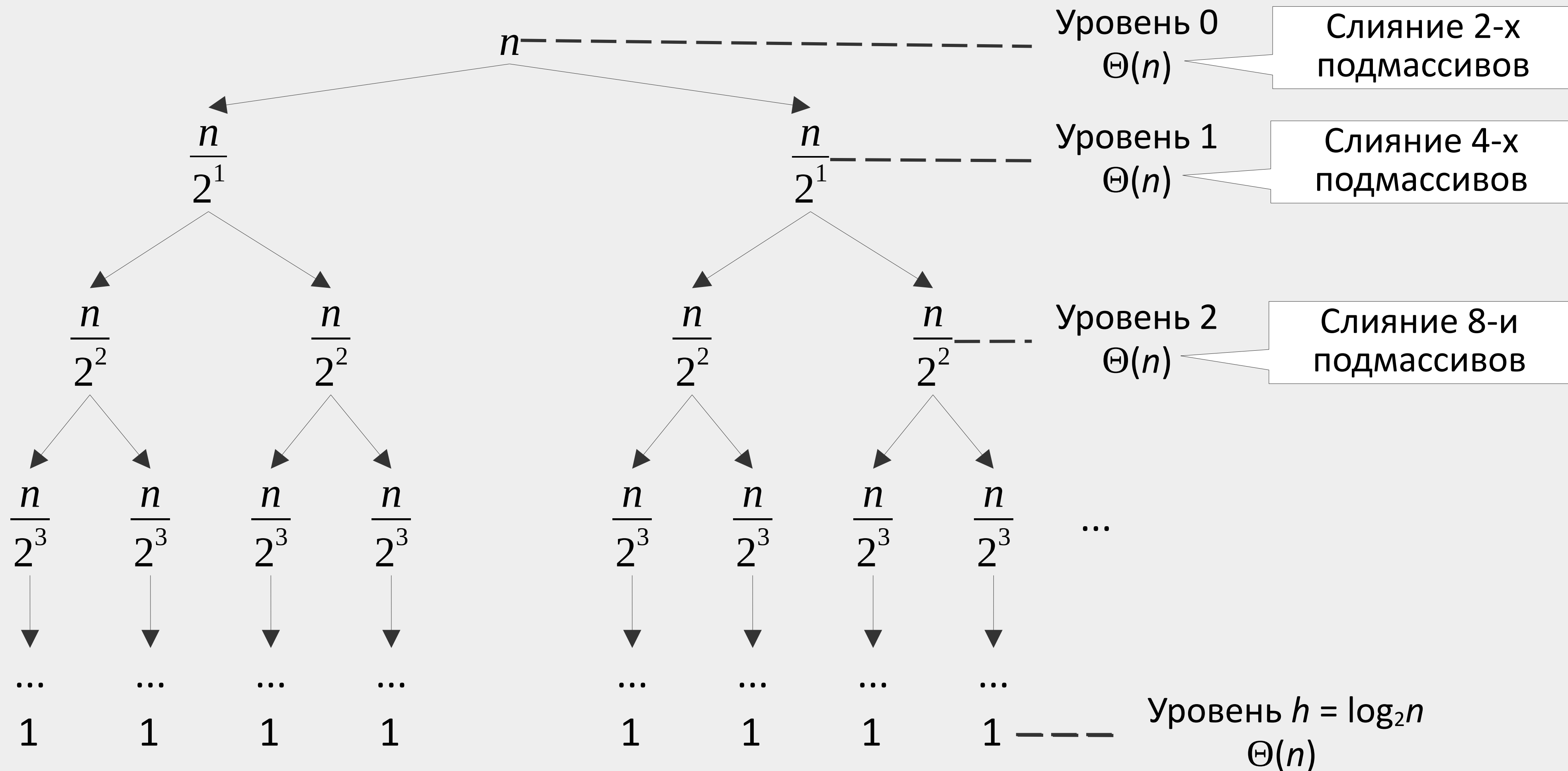
```
1  function Merge(A[1:n], low, mid, high)
2    for i = low to high do
3      B[i] = A[i] /* Копия массива A */
4    end for
5    l = low /* Начало левого подмассива */
6    r = mid + 1 /* Начало правого подмассива */
7    i = low
8    while l <= mid and r <= high do
9      if B[l] <= B[r] then
10       A[i] = B[l]
11       l = l + 1
12     else
13       A[i] = B[r]
14       r = r + 1
15     end if
16     i = i + 1
17   end while
18 /* Копируем остатки подмассивов */
19   while l <= mid do
20     A[i] = B[l]
21     l = l + 1
22     i = i + 1
23   end while
24   while r <= high do
25     A[i] = B[r]
26     r = r + 1
27     i = i + 1
28   end while
29 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 $\Theta(n)$

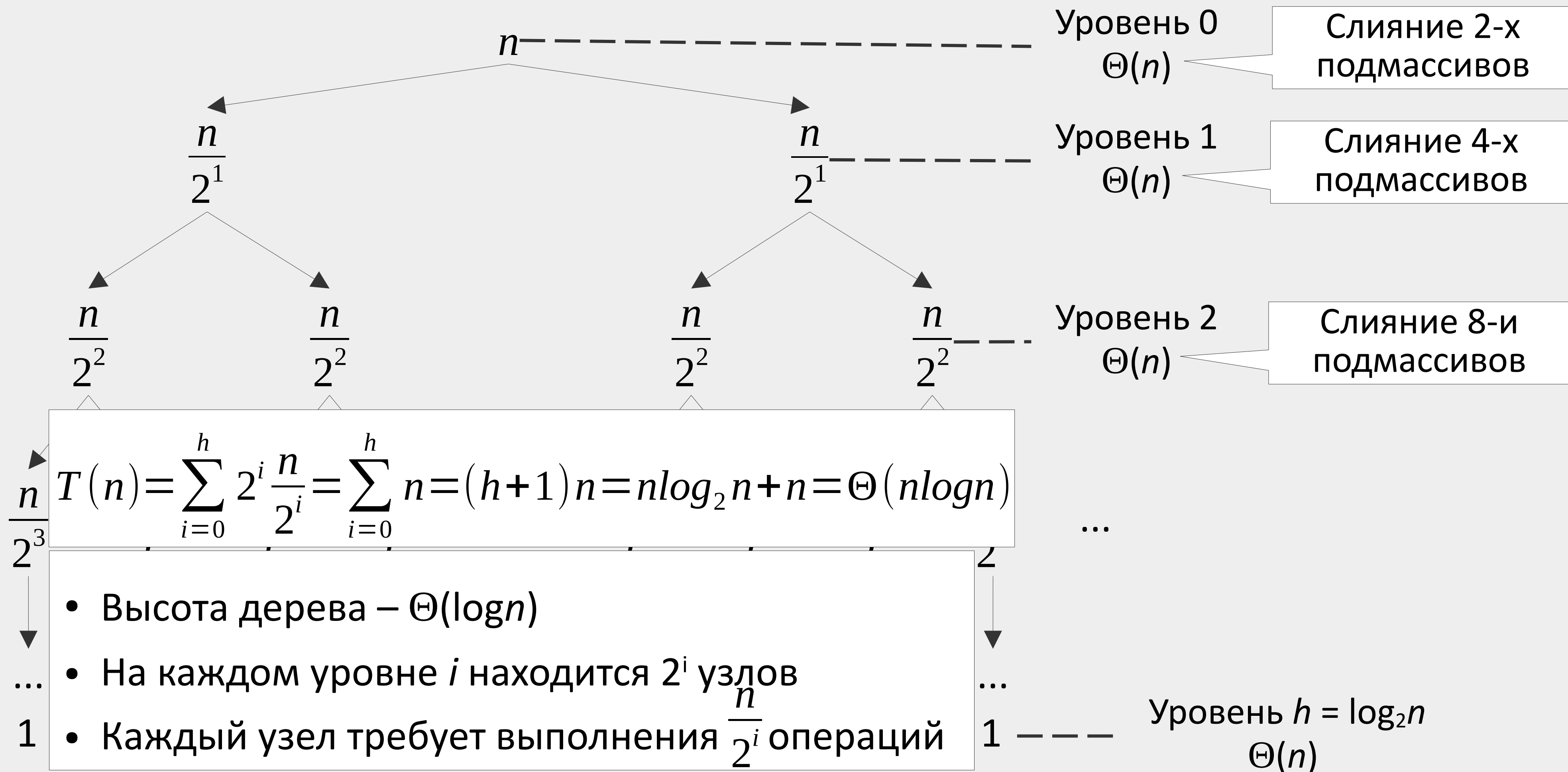
Дерево рекурсивных вызовов сортировки слиянием



Дерево рекурсивных вызовов сортировки слиянием



Дерево рекурсивных вызовов сортировки слиянием



Быстрая сортировка (Quick Sort)

1. Из элементов A[1], A[2], …, A[*n*] выбирается **опорный элемент** (*pivot element*)
   A. Опорный элемент желательно выбирать так, чтобы его значение было близко к среднему значению всех элементов массива
   B. Вопрос о выборе опорного элемента открыт (первый, последний, средний из трех, случайный, …)

- 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
- Шаги 1 и 2 рекурсивно повторяются для левой и правой частей

Быстрая сортировка (Quick Sort)

```
void QuickSort(int* A, int low, int high)
{
    if (low < high) {
        int p = Partition(A, low, high);
        QuickSort(A, low, p - 1);
        QuickSort(A, p + 1, high);
    }
}
```

```
n = 6, array:
83 45 10 22 17 36
pivot: 36
83 45 10 22 17 36
83 45 10 22 17 36
10 45 83 22 17 36
10 22 83 45 17 36
10 22 17 45 83 36
10 22 17 36 83 45
[0, 5], 3
```

```
pivot: 17
10 22 17
10 22 17
10 17 22
[0, 2], 1
pivot: 45
83 45
45 83
[4, 5], 4
10 17 22 36 45 83
```

```
int Partition(int* A, int low, int high)
{
    int pivot = A[high];
    int i = low;
    for (int j = low; j < high; j++) {
        if (A[j] < pivot) {
            swap(A[i], A[j]);
            i = i + 1;
        }
    }
    swap(A[i], A[high]);
    return i;
}
```

- Код на С
- Массив начинается с 0

Сортировка подсчетом (Counting Sort)

```
function CountingSort(A[n], B[n], n, k)
    count[k];
    for (i = 0; i < k; i++)
        count[i] = 0;
    for (i = 0; i < n; i++)
        count[A[i]]++;
    for (i = 1; i < k; i++)
        count[i] += count[i - 1];
    for (i = n - 1; i >= 0; i--)
        count[A[i]]--;
        B[count[A[i]]] = A[i];
end function
```

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

```
n = 12, k = 15, array:
14 0 9 8 3 6 9 12 8 3 4 10
count:
1 0 0 2 1 0 1 0 2 2 1  0  1  0  1
count:
1 1 1 3 4 4 5 5 7 9 10 10 11 11 12
i:  count[A[i]] A[i]
11: 9           10
10: 3           4
9:  2           3
8:  6           8
7:  10          12
6:  8           9
5:  4           6
4:  1           3
3:  5           8
2:  7           9
1:  0           0
0:  11          14
result:
i:    0 1 2 3 4 5 6 7 8 9  10 11
B[i]: 0 3 3 4 6 8 8 9 9 10 12 14
```

Поразрядная сортировка (Radix Sort)

```
function RadixSortLSD(A[n], n)
    maxnum = A[0];
    for (i = 1; i < n; i++)
        if (A[i] > maxnum)
            maxnum = A[i]
        end if
    end for
    r = 1
    while maxnum / r > 0 do
        CountingSort(A, n, r)
        r = r * 10
    end while
end function
```

CountingSort index:

`(A[i] / r) % 10`

- LSD (*least significant digit*, начиная с последней цифры) может быть использована для стандартной сортировки чисел
- MSD (*most significant digit*, начиная с первой цифры) может быть использована для сортировки строк
- Можно сортировать по основанию 2, 8, 10, 16…
- Не использует операцию сравнения
- В данном случае для сортировки разрядов используется CountingSort

– Чему равна вычислительная сложность алгоритма?

Поразрядная сортировка (Radix Sort)

```
function RadixSortLSD(A[n], n)
    maxnum = A[0];
    for (i = 1; i < n; i++)
        if (A[i] > maxnum)
            maxnum = A[i]
        end if
    end for
    r = 1
    while maxnum / r > 0 do
        CountingSort(A, n, r)
        r = r * 10
    end while
end function
```

CountingSort index:

```
(A[i] / r) % 10
```

- LSD (*least significant digit*, начиная с последней цифры) может быть использована для стандартной сортировки чисел
- MSD (*most significant digit*, начиная с первой цифры) может быть использована для сортировки строк
- Можно сортировать по основанию 2, 8, 10, 16…
- Не использует операцию сравнения
- В данном случае для сортировки разрядов используется CountingSort

– Чему равна вычислительная сложность алгоритма?

$O(d(n + k))$

Поразрядная сортировка (Radix Sort)

```
function RadixSortLSD(A[n], n)
    maxnum = A[0];
    for (i = 1; i < n; i++)
        if (A[i] > maxnum)
            maxnum = A[i]
        end if
    end for
    r = 1
    while maxnum / r > 0 do
        CountingSort(A, n, r)
        r = r * 10
    end while
end function
```

```
n = 7, maxnum = 564, array:
12 564 43 2 64 536 263

564
CURRENT: 4
COUNTING:
12 2 43 263 564 64 536
CURRENT:6
COUNTING:
2 12 536 43 263 564 64
CURRENT:5
COUNTING:
2 12 43 64 263 536 564
```

CountingSort index:

```
(A[i] / r) % 10
```

Дальнейшее чтение

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