Лекция 2.
Анализ рекурсивных алгоритмов

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

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

Рекурсивные функции (recursive functions)

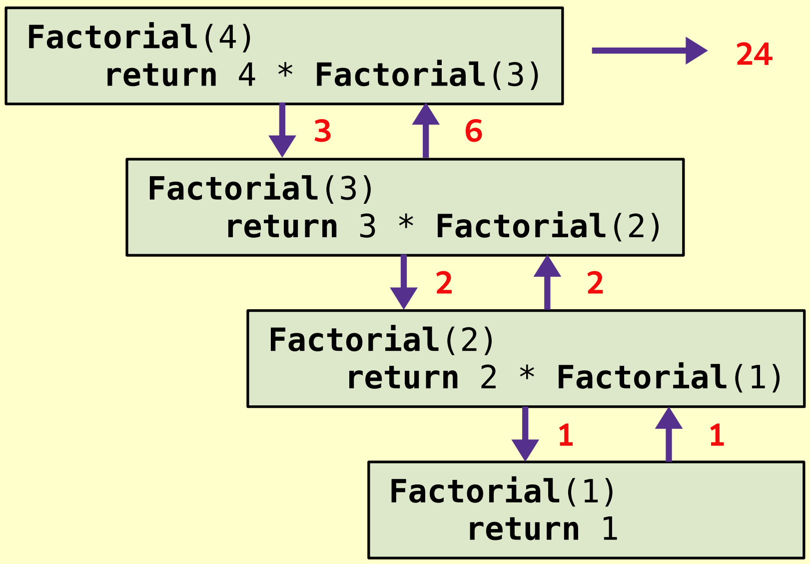
- ***Рекурсивная функция*** (recursive function) — функция, в теле которой присутствует вызов самой себя
- Алгоритм, основанный на таких функциях, называется *рекурсивным алгоритмом* (recursive algorithm)

```
function Factorial(n)
    if n = 1 then
        return 1
    end if
    return n * Factorial(n - 1)
end function

function main()
    print Factorial(4)
end function
```

Рекурсивные функции (recursive functions)

- ***Рекурсивная функция*** (recursive function) — функция, в теле которой присутствует вызов самой себя
- Алгоритм, основанный на таких функциях, называется *рекурсивным алгоритмом* (recursive algorithm)

```
function Factorial(n)
    if n = 1 then
        return 1
    end if
    return n * Factorial(n - 1)
end function

function main()
    print Factorial(4)
end function
```

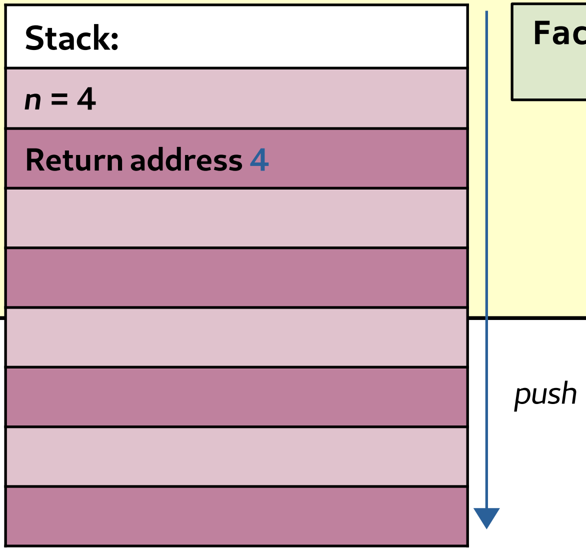


Стек вызовов функций (call stack)

- **Системный стек** (stack) — память, предназначенная для хранения адресов возврата из функций, локальных переменных и передачи аргументов в функции

```
function Factorial(n)
1:   if n = 1 then
2:      ret
3:   end if
4:   return n * Factorial(n - 1)
end function
```

Factorial(4)




```
Factorial(4)
    return 4 * Factorial(3)
```

3

- Рекурсивные функции могут занимать значительную часть стековой памяти для хранения адресов возврата!
- Стек имеет конечный размер:

```
$ ulimit -s
8192
```

Стек вызовов функций (call stack)

- **Системный стек** (stack) — память, предназначенная для хранения адресов возврата из функций, локальных переменных и передачи аргументов в функции

```
function Factorial(n)
1:   if n = 1 then
2:        ret
3:   end if
4:   return
end function
```

Factorial(4)

Factorial(3)

Stack:
n = 4
Return address 4
n = 3
Return address 4

```
Factorial(4)
    return 4 * Factorial(3)
```

3

```
Factorial(3)
    return 3 * Factorial(2)
```

2

Стек вызовов функций (call stack)

- **Системный стек** (stack) — память, предназначенная для хранения адресов возврата из функций, локальных переменных и передачи аргументов в функции

```
function Factorial(n)
1:   if n = 1 then
2:       ret
3:   end if
4:   return
end function
```

Factorial(4) →

Factorial(3) →

Factorial(2) →

Stack:
***n* = 4**
Return address 4
n = 3
Return address 4
n = 2

```
Factorial(4)
    return 4 * Factorial(3)
```

3

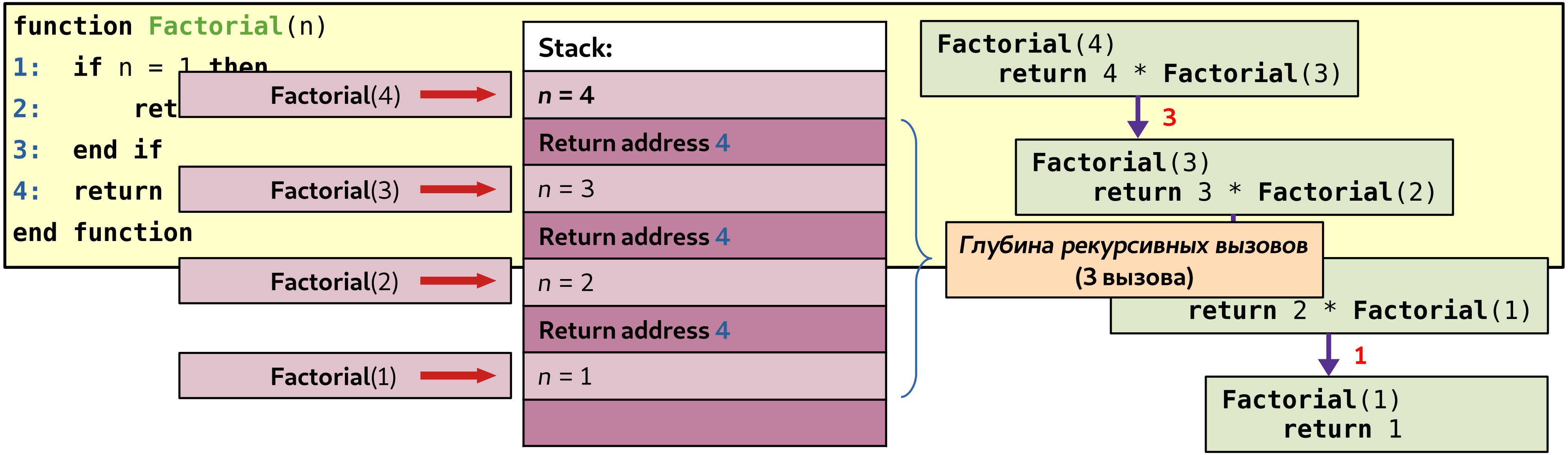
```
Factorial(3)
    return 3 * Factorial(2)
```

2

```
Factorial(2)
    return 2 * Factorial(1)
```

Стек вызовов функций (call stack)

- **Системный стек** (stack) — память, предназначенная для хранения адресов возврата из функций, локальных переменных и передачи аргументов в функции



Стек вызовов функций (call stack)

- **Системный стек** (stack) — память, предназначенная для хранения адресов возврата из функций, локальных переменных и передачи аргументов в функции

```
function Factorial(n)
1:   if n = 1 then
2:       ret
3:   end if
4:   return
end function
```

Factorial(4) →

Factorial(3) →

Factorial(2) →

Stack:
***n* = 4**
Return address 4
n = 3
Return address 4
n = 2
Return address 4

```
Factorial(4)
    return 4 * Factorial(3)
```

↓ 3

```
Factorial(3)
    return 3 * Factorial(2)
```

↓ 2

```
Factorial(2)
    return 2 * Factorial(1)
```

↓ 1

```
Factorial(1)
    return 1
```

При выходе из функции из головы стека извлекается адрес возврата

Стек вызовов функций (call stack)

- **Системный стек** (stack) — память, предназначенная для хранения адресов возврата из функций, локальных переменных и передачи аргументов в функции

```
function Factorial(n)
1:   if n = 1 then
2:         ret
3:   end if
4:   return
end function
```

Factorial(4) →

Factorial(3) →

Stack:
***n* = 4**
Return address 4
n = 3
Return address 4

```
Factorial(4)
    return 4 * Factorial(3)
```

↓ 3

```
Factorial(3)
    return 3 * Factorial(2)
```

↓ 2

```
Factorial(2)
    return 2 * Factorial(1)
```

↓ 1

```
Factorial(1)
    return 1
```

Стек вызовов функций (call stack)

- **Системный стек** (stack) — память, предназначенная для хранения адресов возврата из функций, локальных переменных и передачи аргументов в функции

```
function Factorial(n)
1:   if n = 1 then
2:       ret
3:   end if
4:   return n * Factorial(n - 1)
end function
```

Factorial(4) →

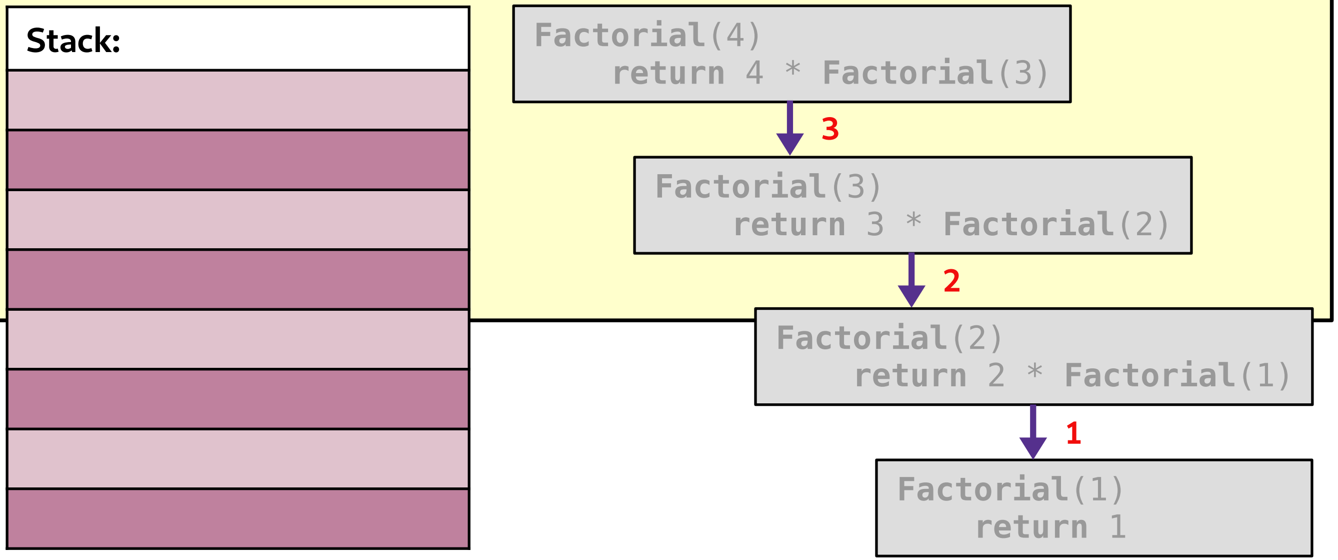
Stack:
n = 4
Return address 4

```
Factorial(4)
    return 4 * Factorial(3)
  ↓ 3
Factorial(3)
    return 3 * Factorial(2)
  ↓ 2
Factorial(2)
    return 2 * Factorial(1)
  ↓ 1
Factorial(1)
    return 1
```

Стек вызовов функций (call stack)

- **Системный стек** (stack) — память, предназначенная для хранения адресов возврата из функций, локальных переменных и передачи аргументов в функции

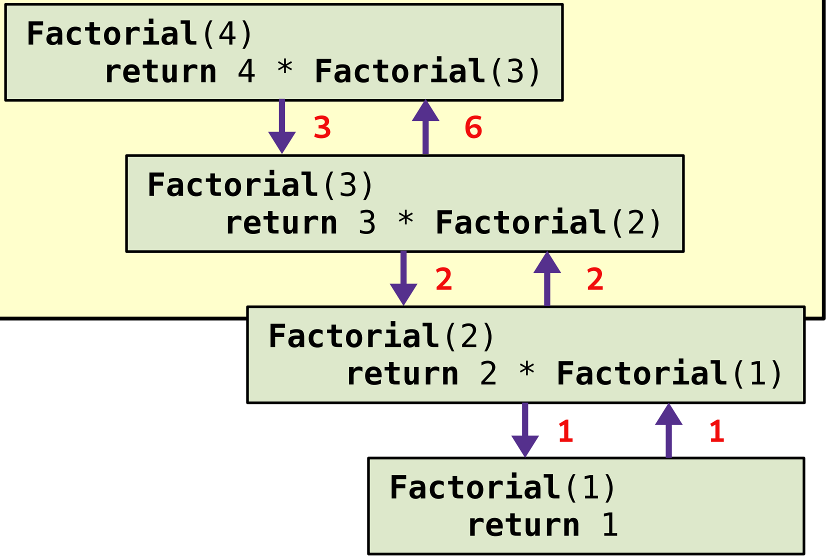
```
function Factorial(n)
    if n = 1 then
        return 1
    end if
    return n * Factorial(n - 1)
end function
```



Стек вызовов функций (call stack)

- **Системный стек** (stack) — память, предназначенная для хранения адресов возврата из функций, локальных переменных и передачи аргументов в функции

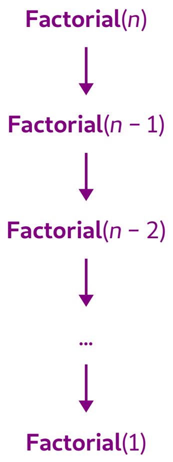
```
function Factorial(n)
    if n = 1 then
        return 1
    end if
    return n * Factorial(n - 1)
end function
```



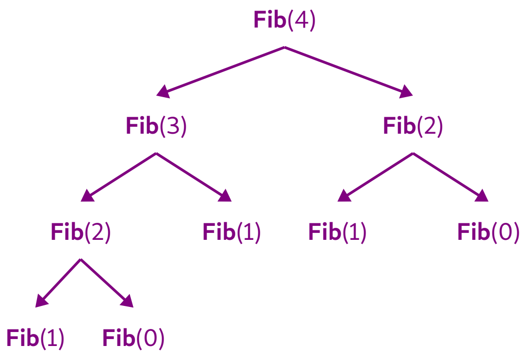
Виды рекурсии

- ***Линейная рекурсия*** (linear recursion) — в функции присутствует единственный рекурсивный вызов самой себя
- ***Древовидная рекурсия*** (нелинейная, non-linear recursion) — в функции присутствует несколько рекурсивных вызовов

```
function Factorial(n)
    if n = 1 then
        return 1
    end if
    return n * Factorial(n - 1)
end function
```




```
function Fib(n)
    if n < 2 then
        return n
    end if
    return Fib(n - 1) + Fib(n - 2)
end function
```



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)

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

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

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

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

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

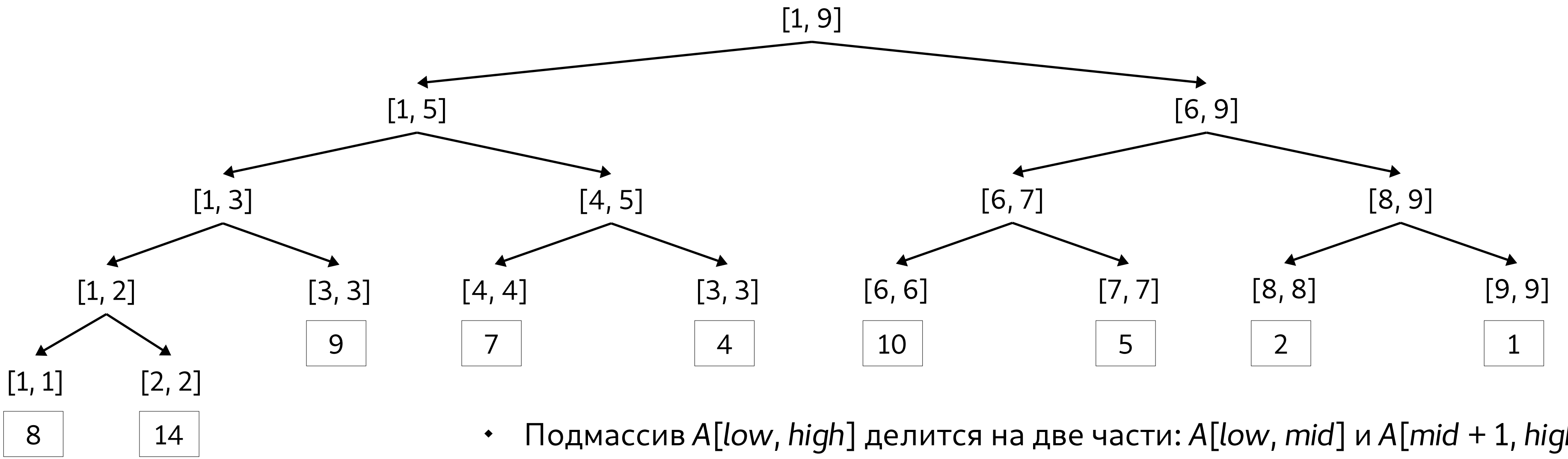
- Алгоритм включает две фазы:

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

Сортировка слиянием: фаза разделения

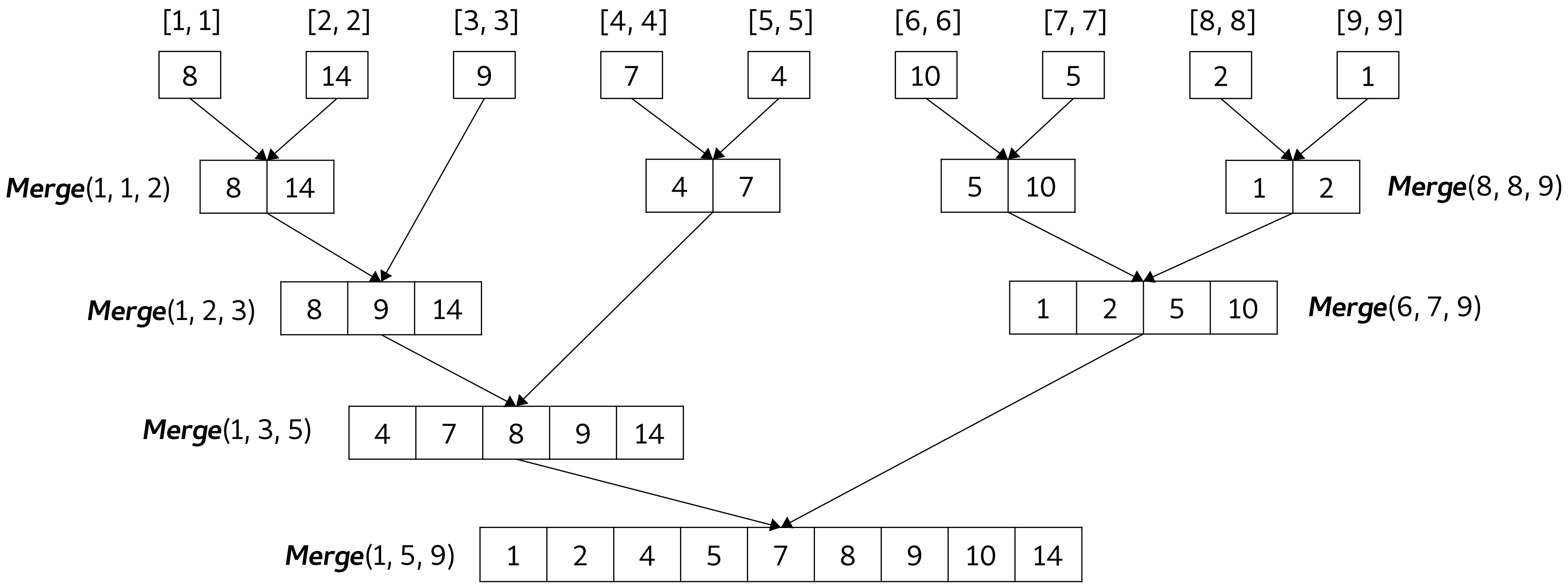
MergeSort(1, 9)

1	2	3	4	5	6	7	8	9
8	14	9	7	4	10	5	2	1



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

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
```

Анализ эффективности алгоритма сортировки слиянием

Анализ дерева рекурсивных вызовов

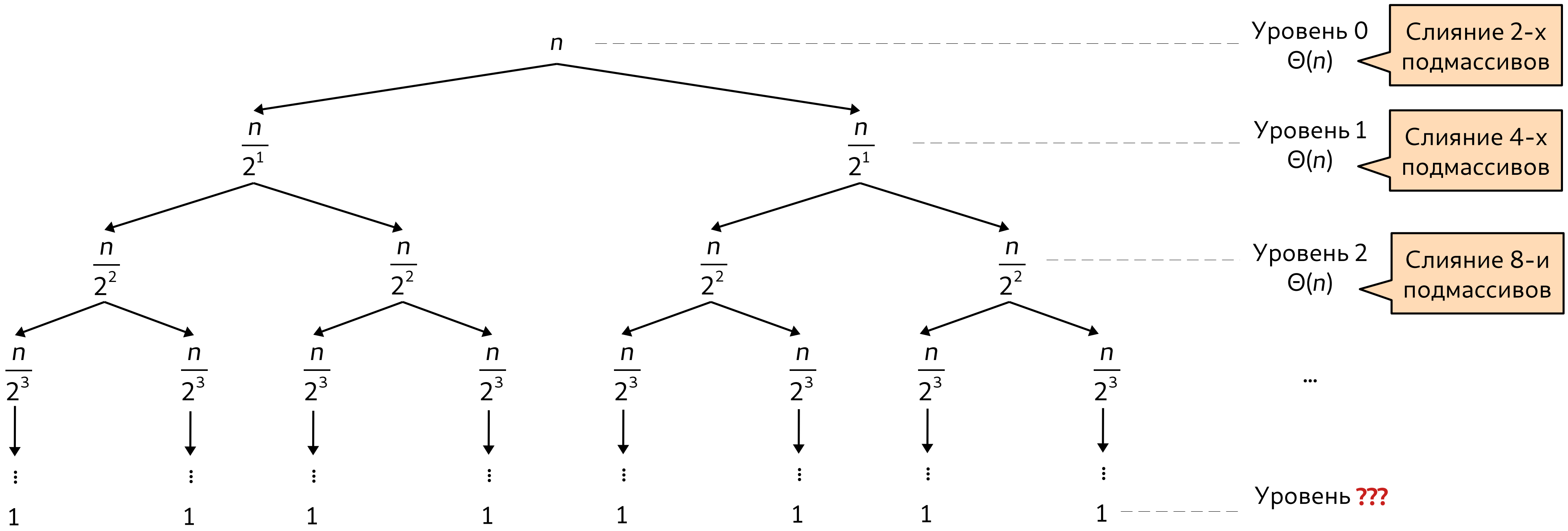
Сортировка слиянием (Merge Sort)

```
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
```

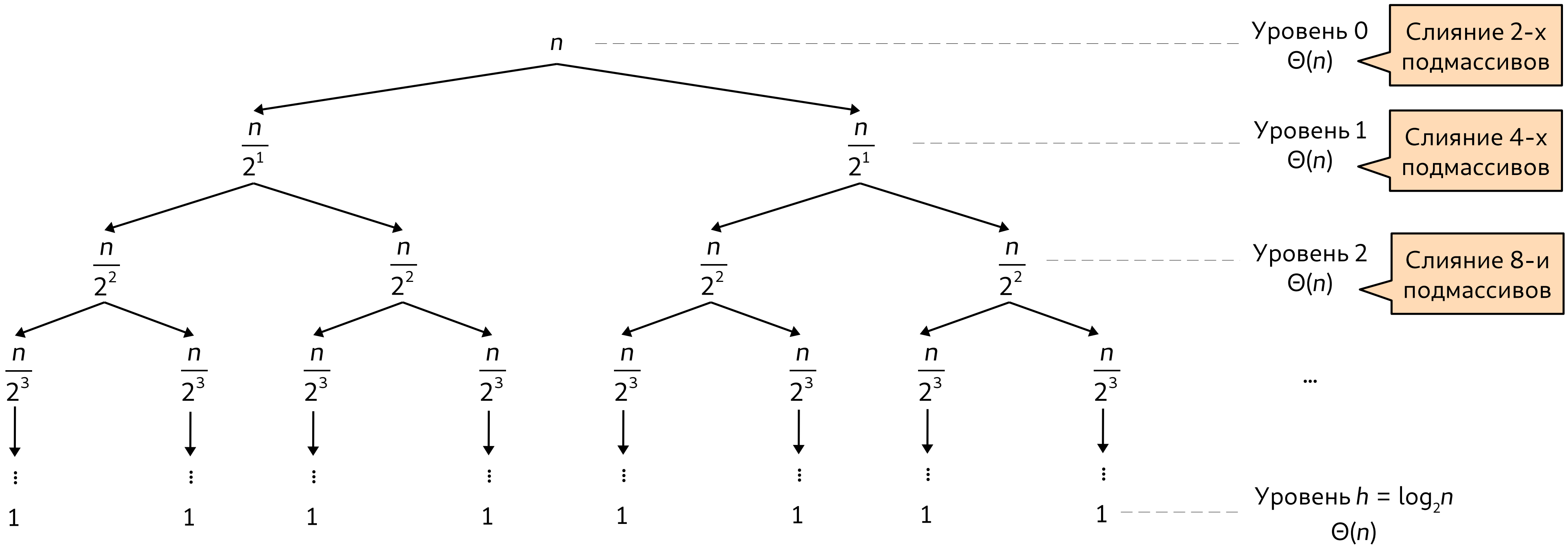
```
            end if
            i = i + 1
        end while
        /* Копируем остатки подмассивов */
        while l <= mid do

        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 $\Theta(n)$

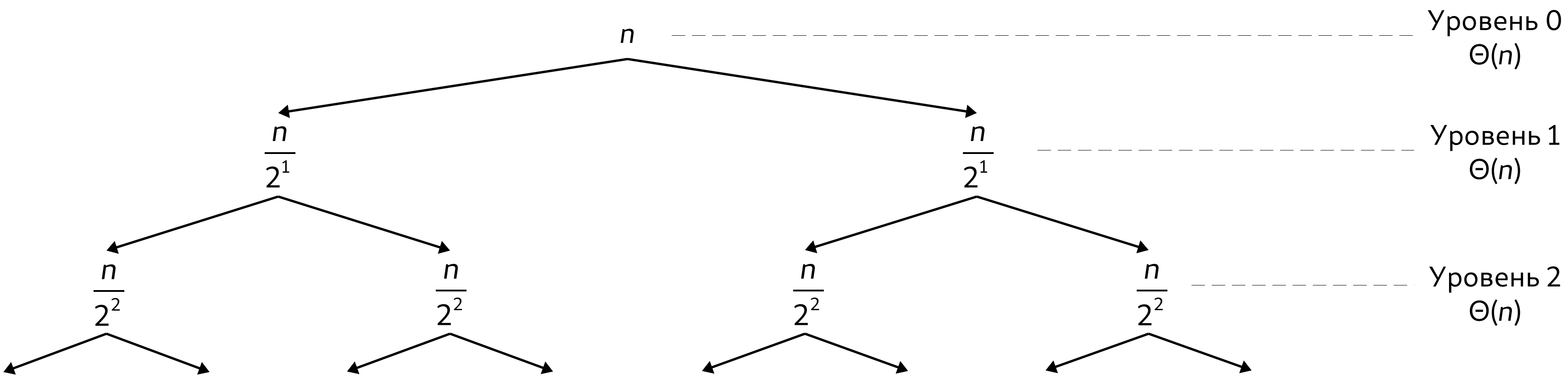
Дерево рекурсивных вызовов сортировки слиянием



Дерево рекурсивных вызовов сортировки слиянием



Дерево рекурсивных вызовов сортировки слиянием

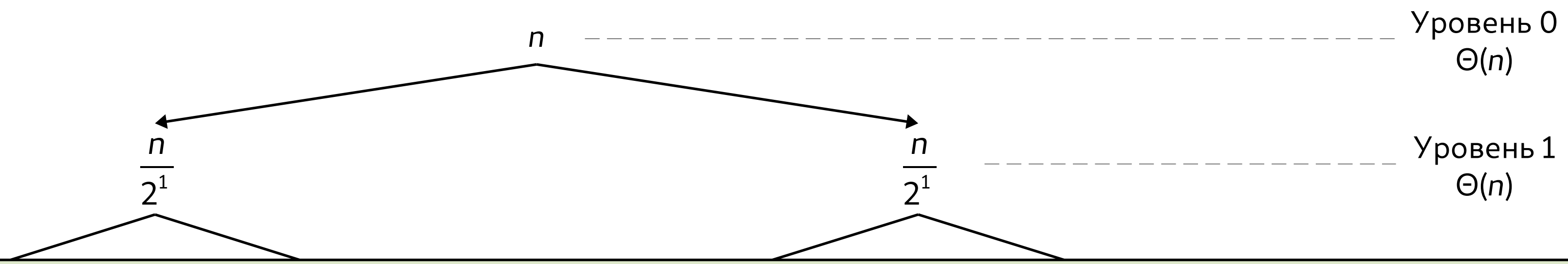


- Высота дерева — $\Theta(\log n)$
- На каждом уровне i находится 2^i узлов
- Каждый узел требует выполнения $\frac{n}{2^i}$ операций

...

Уровень $h = \log_2 n$
$\Theta(n)$

Дерево рекурсивных вызовов сортировки слиянием



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

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

...

Уровень $h = \log_2 n$

$\Theta(n)$

Анализ эффективности алгоритма сортировки слиянием

Решение рекуррентных уравнений

Решение рекуррентных уравнений

- Время $T(n)$ работы алгоритма включает время сортировки левого подмассива 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

Основной метод (master method)

- Рассмотрим решение рекуррентных уравнений, когда исходную задачу размера n можно разделить на $a \geq 1$ подзадач размера n / b
- Будем считать, что для решения задачи размера 1 требуется время $O(1)$
- Декомпозиция задачи размера n и комбинирование (слияние) решений подзадач требует $f(n)$ единиц времени
- Тогда время $T(n)$ решения задачи размера n можно записать как

$$T(n) = aT(n / b) + f(n),$$

  где $a \geq 1$, $b > 1$
- Записанное уравнение называется ***обобщённым рекуррентным уравнением декомпозиции*** (general divide-and-conquer recurrence)
- Решением этого уравнения является порядок роста функции $T(n)$, который определяется из следующей ***основной теоремы*** (master theorem)

Основная теорема (master theorem)

Теорема. *Если в обобщённом рекуррентном уравнении декомпозиции f(n) = Θ(n^d), где d ≥ 0, то*

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

- **Пример 1.** В рекуррентном уравнении алгоритма сортировки слиянием

$$T(n)=2T(n/2)+\Theta(n)$$

- $a = 2$, $b = 2$, $f(n) = \Theta(n)$ и $d = 1$
- Следовательно, имеем случай $a = b^d$
- Тогда, следуя теореме, вычислительная сложность сортировки слиянием в худшем случае равна **$T(n)$** $= \Theta(n^d \log n) =$ **$\Theta(n \log n)$**

Основная теорема (master theorem)

- **Пример 2.** Для некоторого алгоритма получено рекуррентное уравнение

$$T(n)=2T(n/2)+1$$

- Необходимо найти асимптотически точную оценку для $T(n)$
- Нетрудно заметить: $a = 2$, $b = 2$, $f(n) = \Theta(1)$ и $d = 0$
- Поскольку $a > b^d$:

$$T(n)=\Theta\left(n^{\log_b a}\right)=\Theta(n)$$

Домашнее чтение

- **[DSABook]** Глава 2. «Анализ рекурсивных алгоритмов»
- **[CLRS]** Глава 4. «Разделяй и властвуй»
- **[Levitin]** Раздел 2.4 «Математический анализ рекурсивных алгоритмов»

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.