Лекция 2. Анализ рекурсивных алгоритмов

E-mail: lazycoyote11@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Весенний семестр, 2026 г.

Рекурсивные функции (recursive functions)

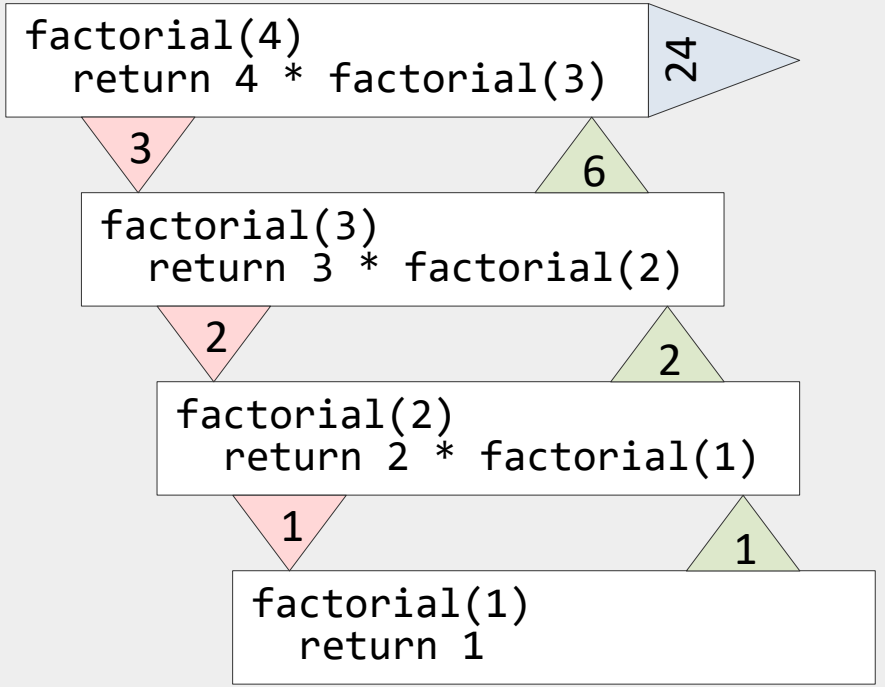
- **Рекурсивная функция** (*recursive function*) – функция, в теле которой присутствует вызов самой себя
- Алгоритм, основанный на таких функциях, называется **рекурсивным алгоритмом** (*recursive algorithm*)

```
int factorial(int n)
{
    if (n == 1)
        return 1;
    return n * factorial(n - 1);
}

int main()
{
    printf("%d\n", factorial(4));
    return 0;
}
```

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*)

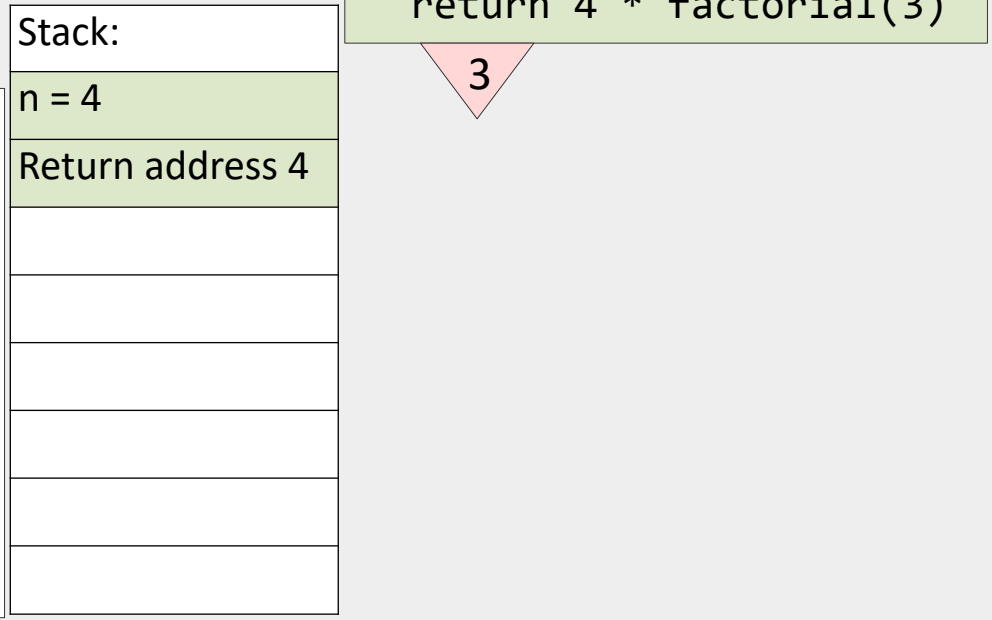
```
int factorial(int n)
{
    if (n == 1)
        return 1;
    return n * factorial(n - 1);
}

int main()
{
    printf("%d\n", factorial(4));
    return 0;
}
```



Стек вызовов функций (call stack)

- **Системный стек** (*stack*) – память, предназначенная для хранения адресов возврата из функций, локальных переменных и передачи аргументов в функции
- Рекурсивные функции могут занимать значительную часть стековой памяти для хранения адресов возврата
- Стек имеет конечный размер:
  - `$ ulimit -s`
  - `8192`

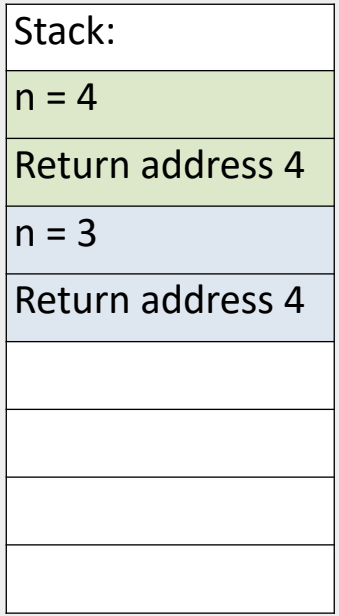
```c
int factorial(int n)
{
  if (n == 1)
    return 1;
  return n * factorial(n - 1);
}

int main()
{
  printf("%d\n", factorial(4));
  return 0;
}
```

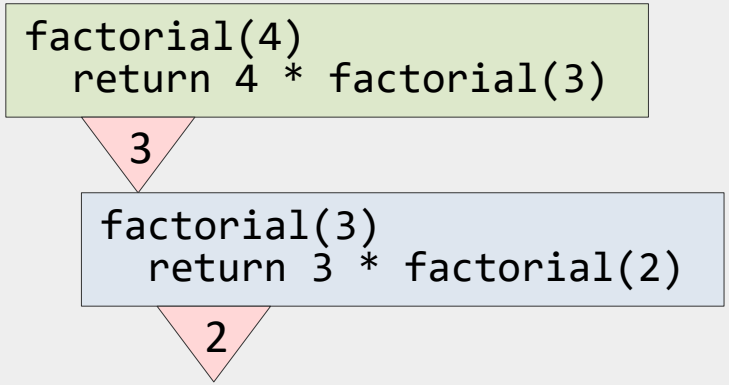


Стек вызовов функций (call stack)

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

```
int factorial(int n)
{
  if (n == 1)
    return 1;
  return n * factorial(n - 1);
}

int main()
{
  printf("%d\n",factorial(4));
  return 0;
}
```

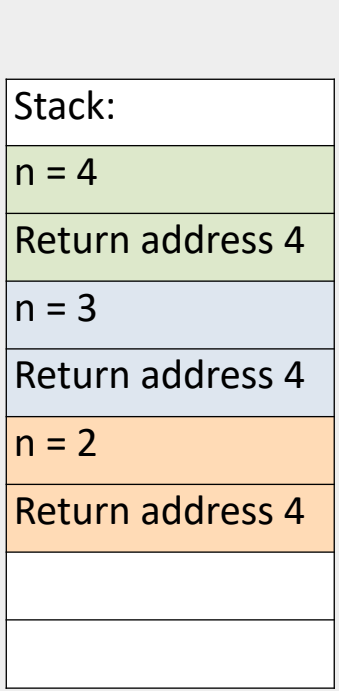


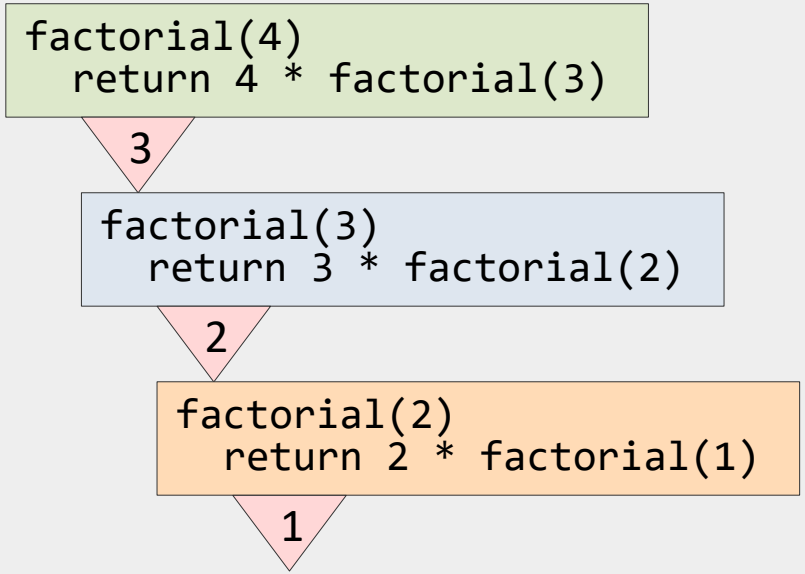


Стек вызовов функций (call stack)

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

```
int factorial(int n)
{
  if (n == 1)
    return 1;
  return n * factorial(n - 1);
}

int main()
{
  printf("%d\n",factorial(4));
  return 0;
}
```

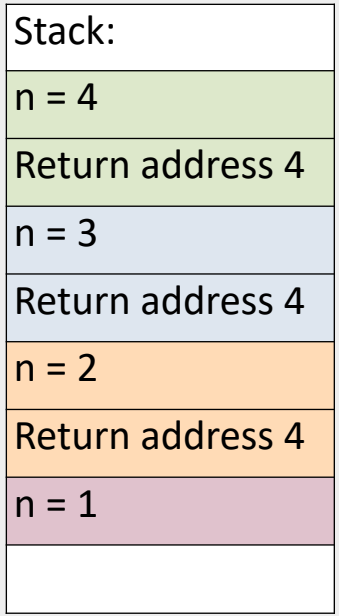


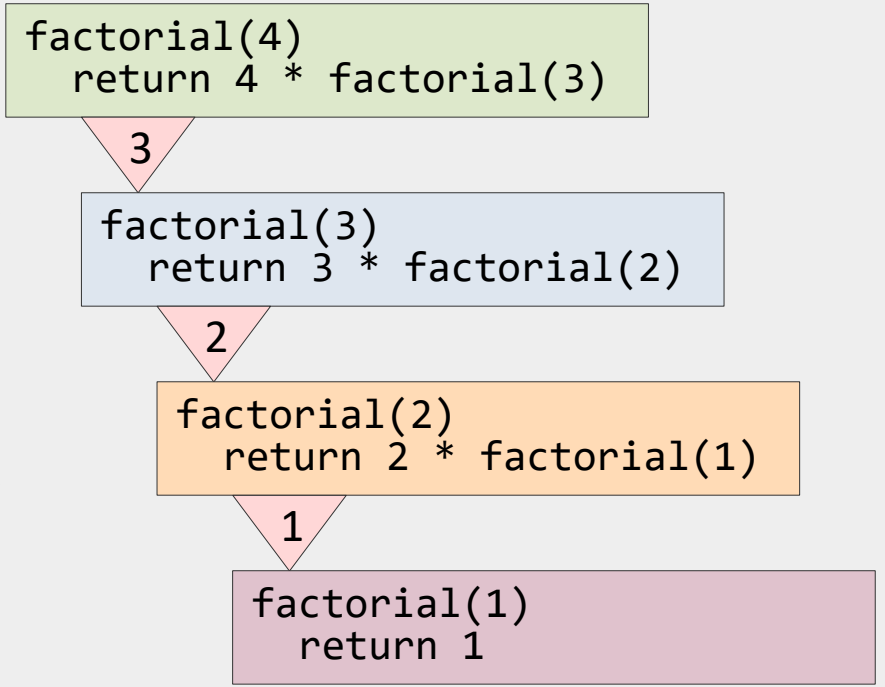


Стек вызовов функций (call stack)

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

```
int factorial(int n)
{
  if (n == 1)
    return 1;
  return n * factorial(n - 1);
}

int main()
{
  printf("%d\n",factorial(4));
  return 0;
}
```



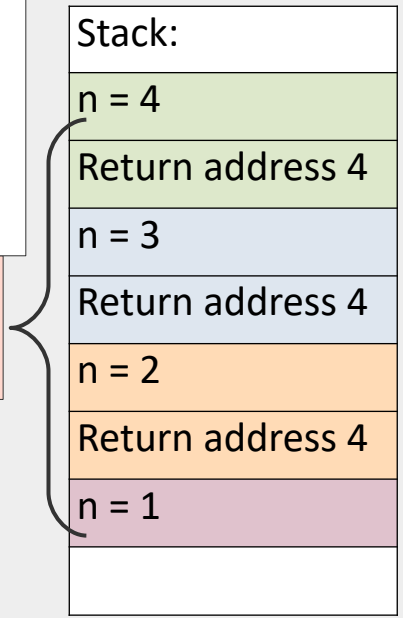


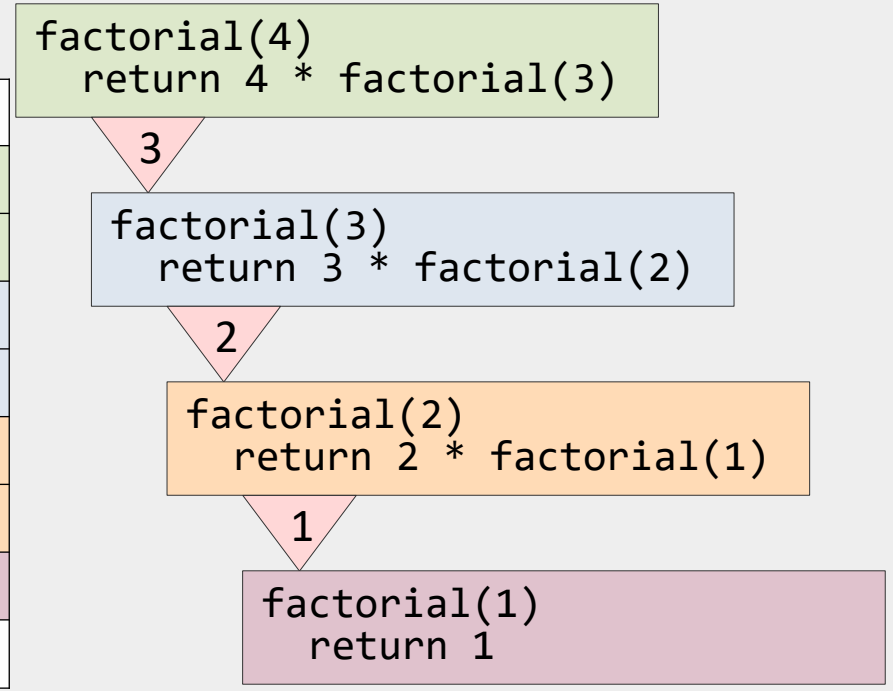
Стек вызовов функций (call stack)

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

```c
int factorial(int n)
{
    if (n == 1)
        return 1;
    return n * factorial(n - 1);
}

int main()
{
    printf("%d\n",factorial(4));
    return 0;
}
```

Глубина рекурсивных вызовов (3 вызова)

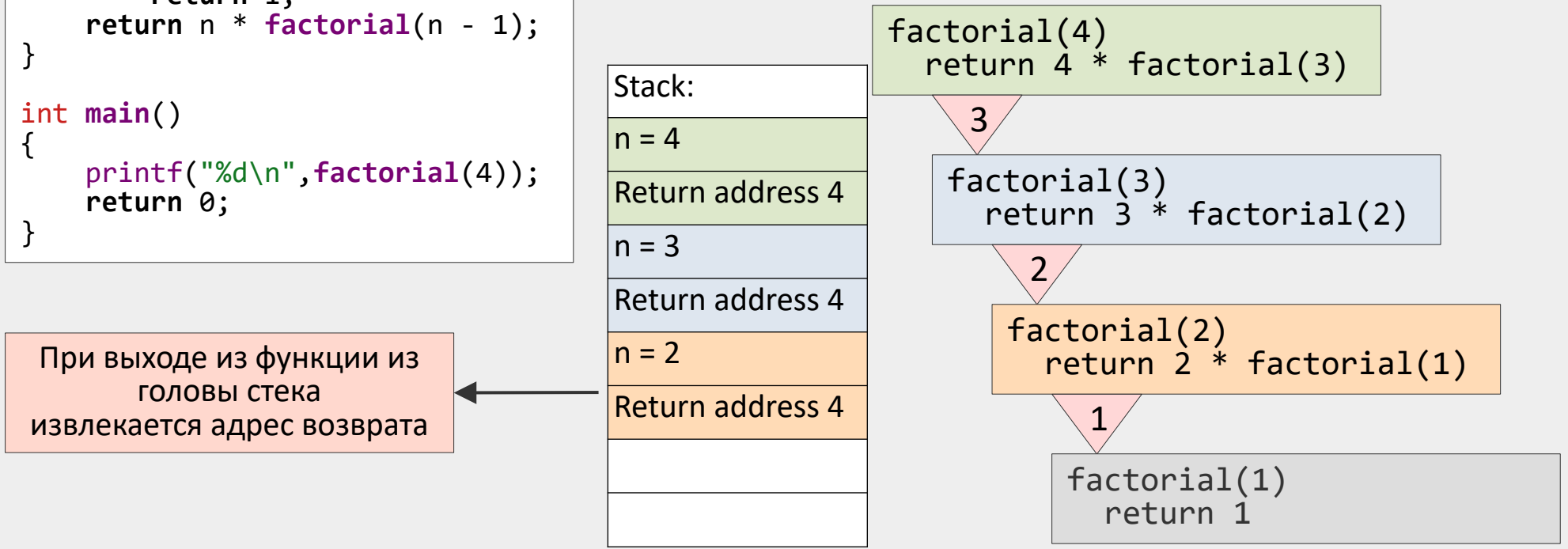




Стек вызовов функций (call stack)

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

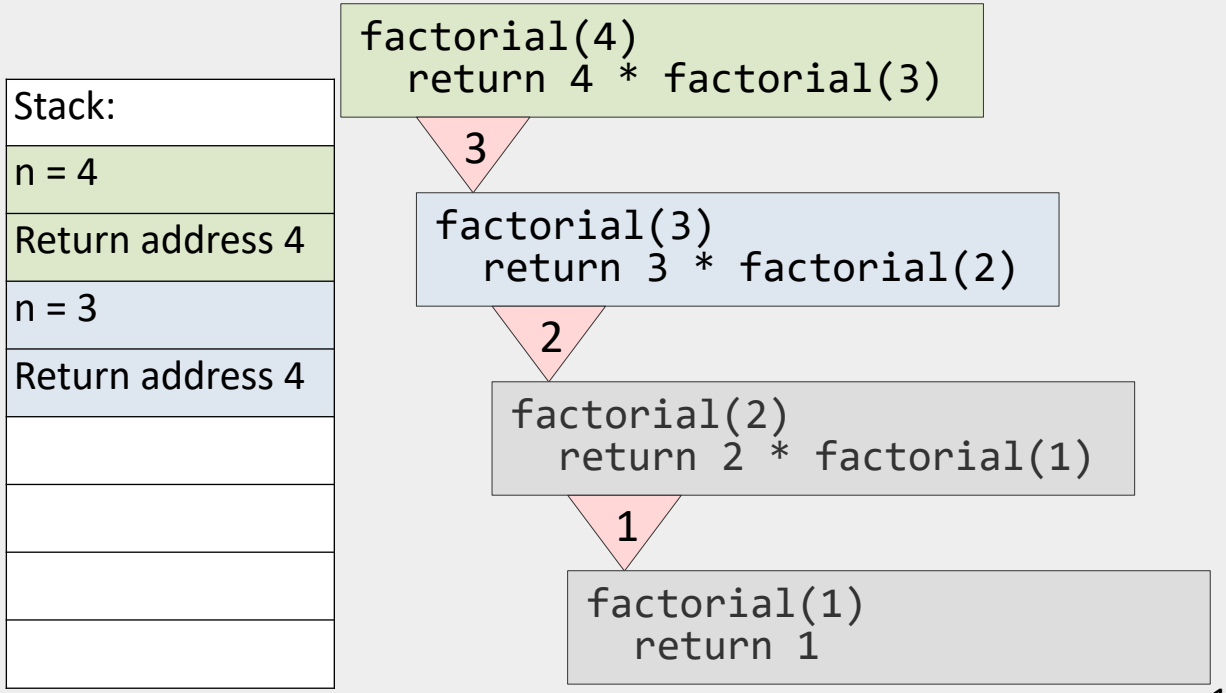
```c
int factorial(int n)
{
    if (n == 1)
        return 1;
    return n * factorial(n - 1);
}

int main()
{
    printf("%d\n",factorial(4));
    return 0;
}
```



Стек вызовов функций (call stack)

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

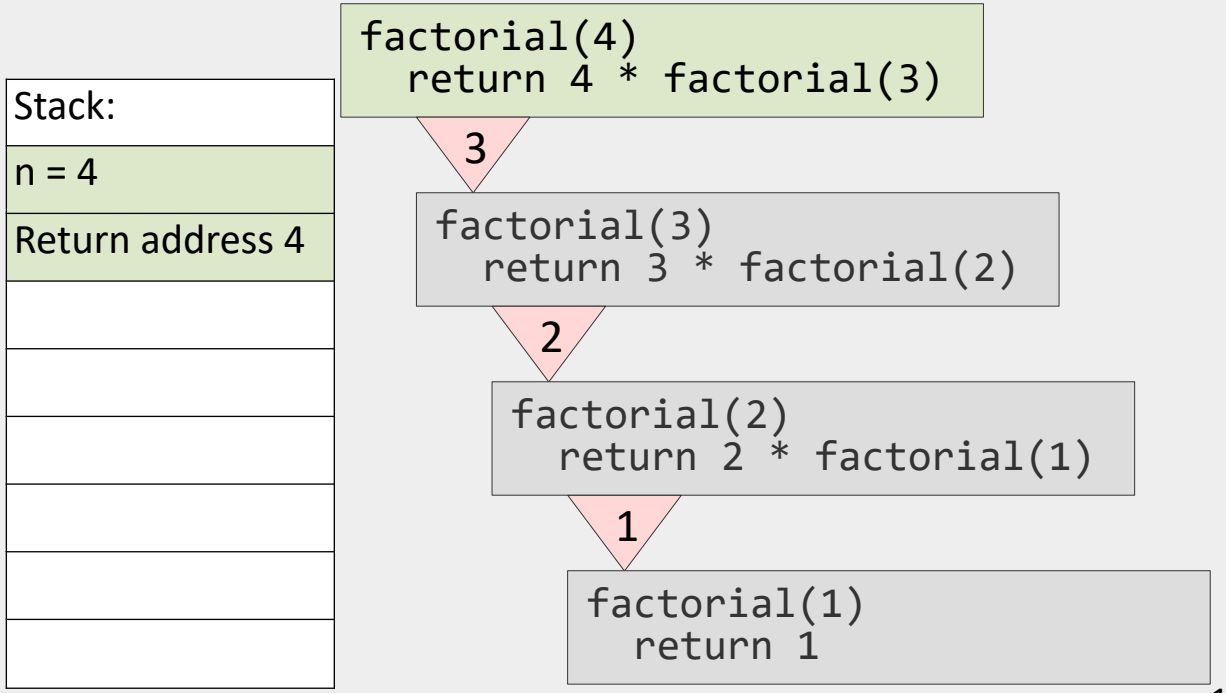
```c
int factorial(int n)
{
    if (n == 1)
        return 1;
    return n * factorial(n - 1);
}

int main()
{
    printf("%d\n",factorial(4));
    return 0;
}
```



Стек вызовов функций (call stack)

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

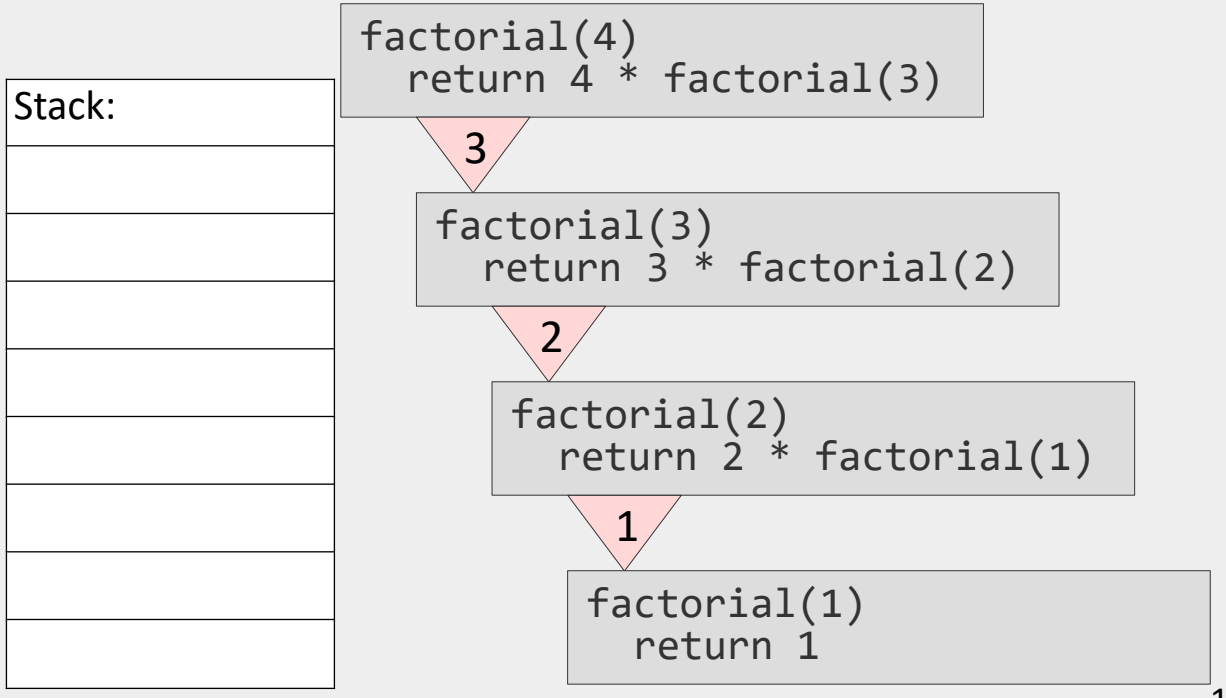
```c
int factorial(int n)
{
    if (n == 1)
        return 1;
    return n * factorial(n - 1);
}

int main()
{
    printf("%d\n",factorial(4));
    return 0;
}
```



Стек вызовов функций (call stack)

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

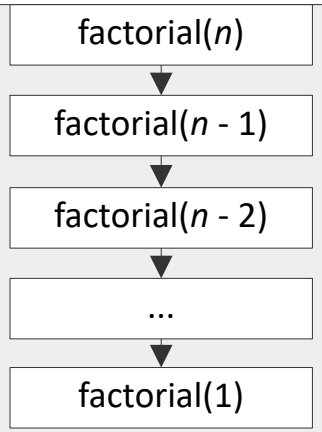
```c
int factorial(int n)
{
    if (n == 1)
        return 1;
    return n * factorial(n - 1);
}

int main()
{
    printf("%d\n",factorial(4));
    return 0;
}
```



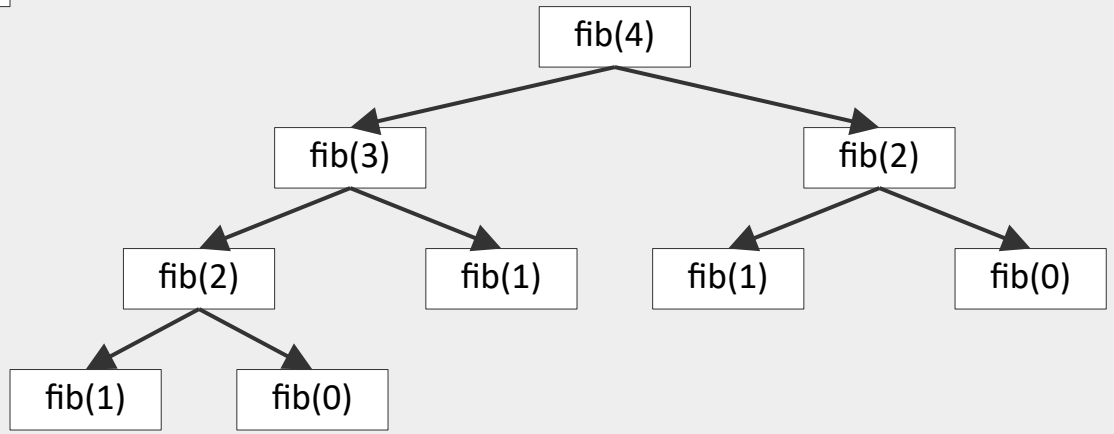
Виды рекурсии

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

```
int factorial(int n)
{
    if (n == 1)
        return 1;
    return n * factorial(n - 1);
}
```




```
int fib(int n)
{
    if (n < 2)
        return n;
    return fib(n - 1) + fib(n - 2);
}
```



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*)

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

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

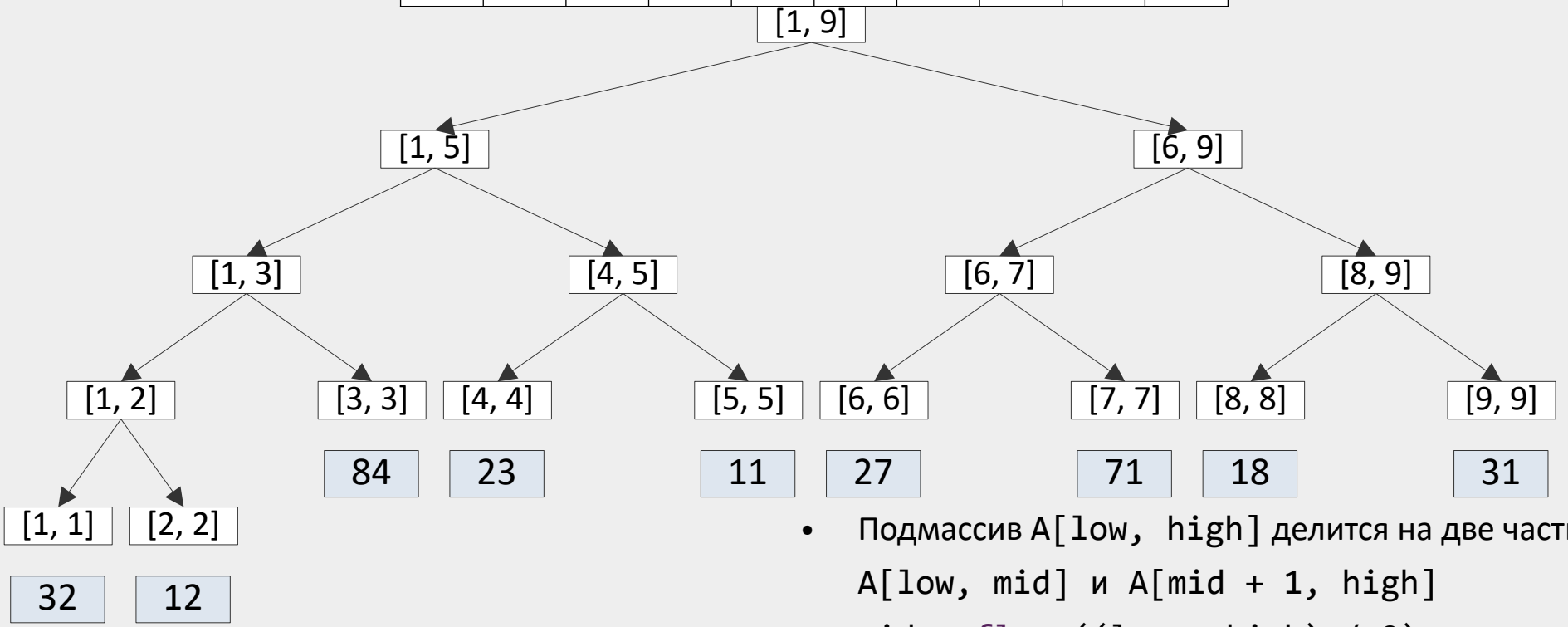
$$A[1] \le A[2] \le \ldots \le A[n]$$

- 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

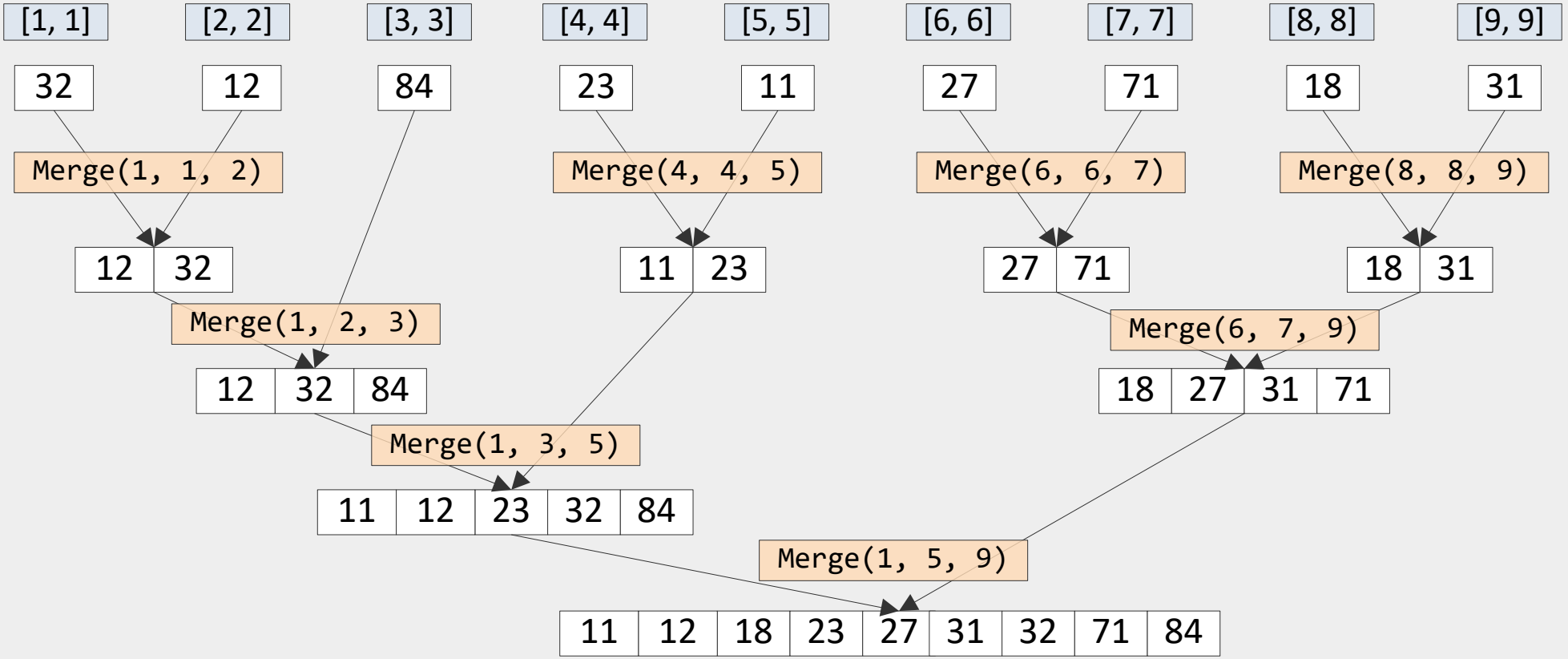
MergeSort(1, 9):

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)

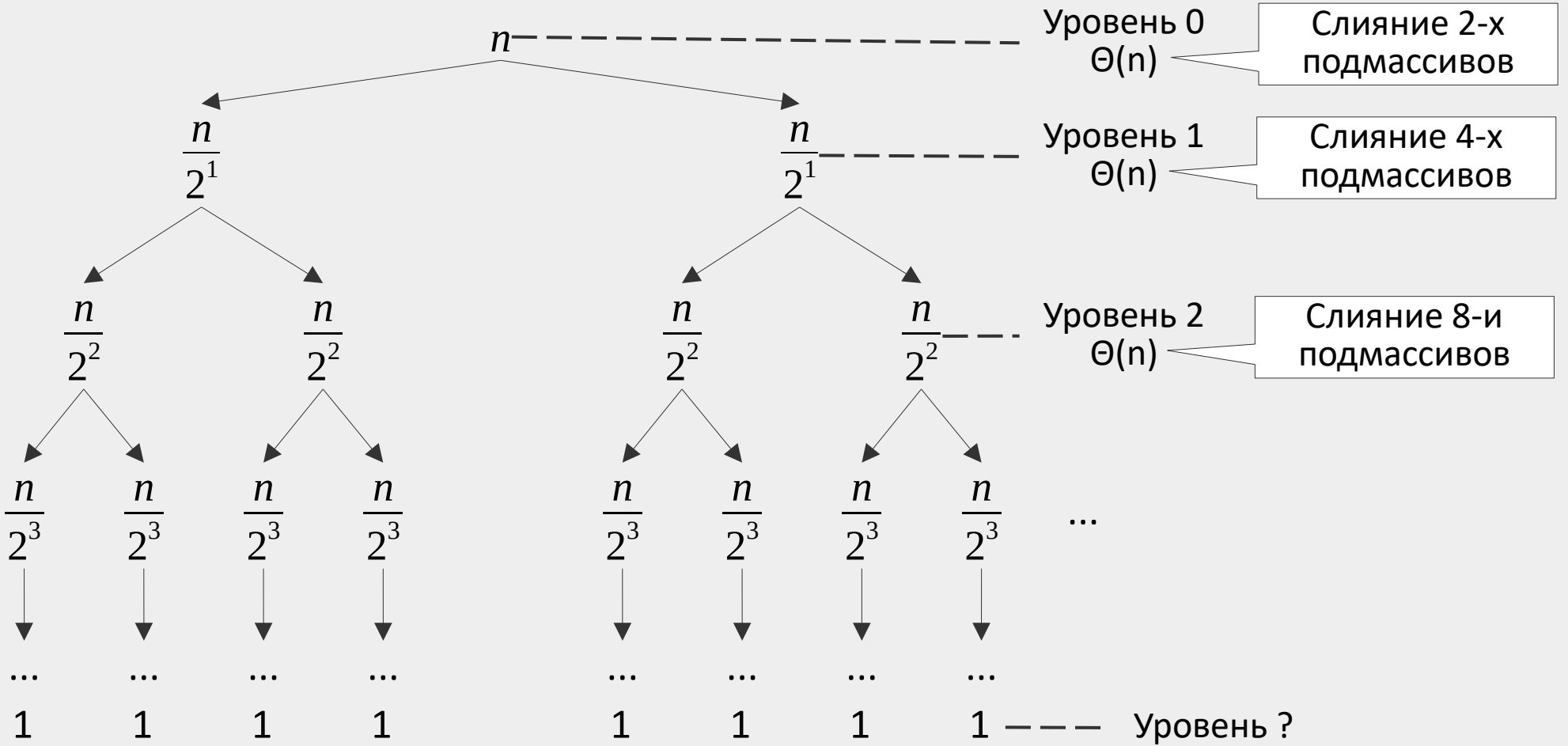
```
1  function Merge(A[1:n], low, mid, high)
2    for i = low to high do
3      B[i] = A[i] /* Копия массива A */
4    end for
5    l = low /* Начало левого подмассива */
6    r = mid + 1 /* Начало правого подмассива */
7    i = low
8    while l <= mid and r <= high do
9      if B[l] <= B[r] then
10       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 $\Theta(n)$

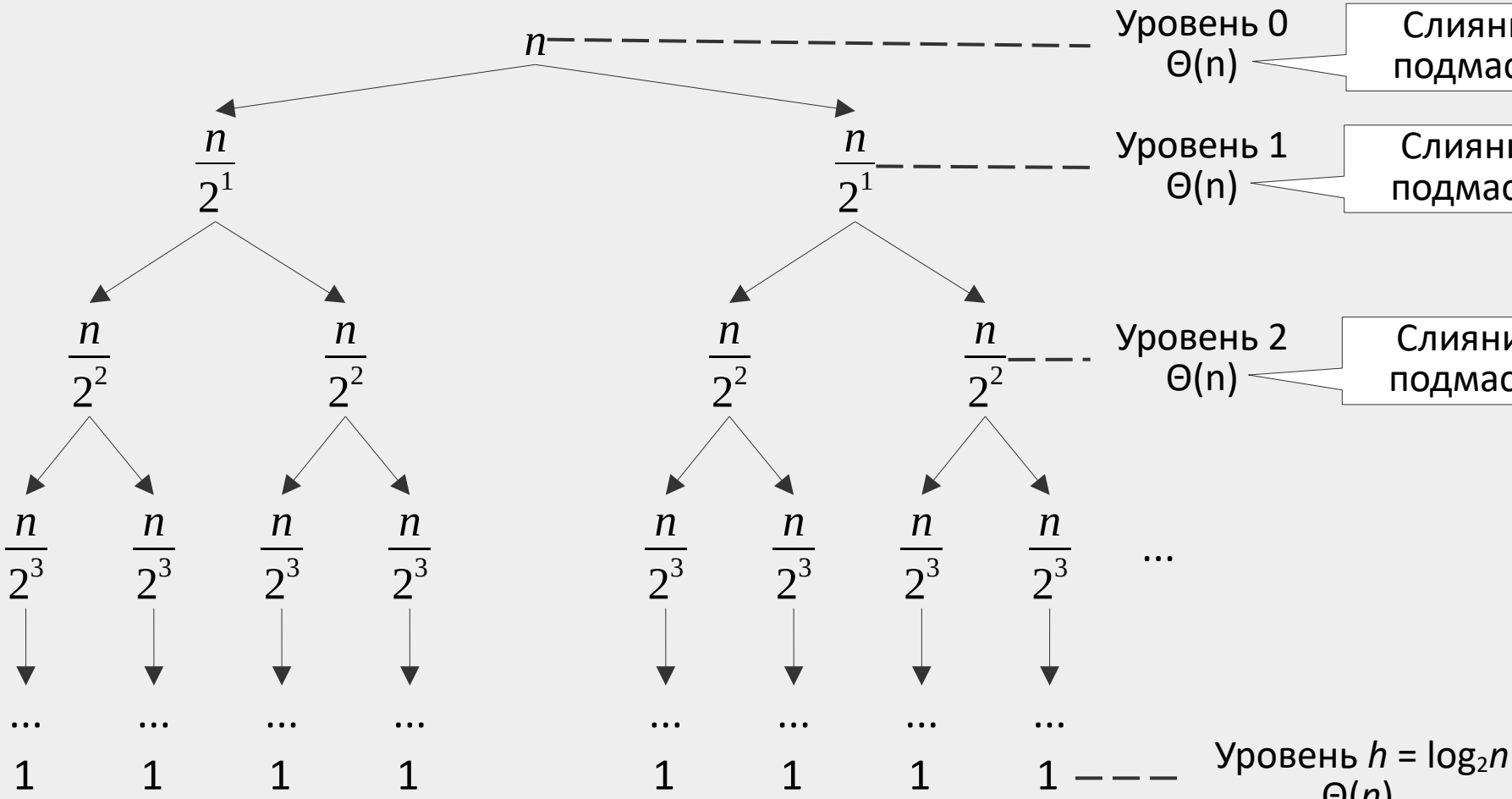
Анализ эффективности алгоритма сортировки слиянием

Анализ дерева рекурсивных вызовов

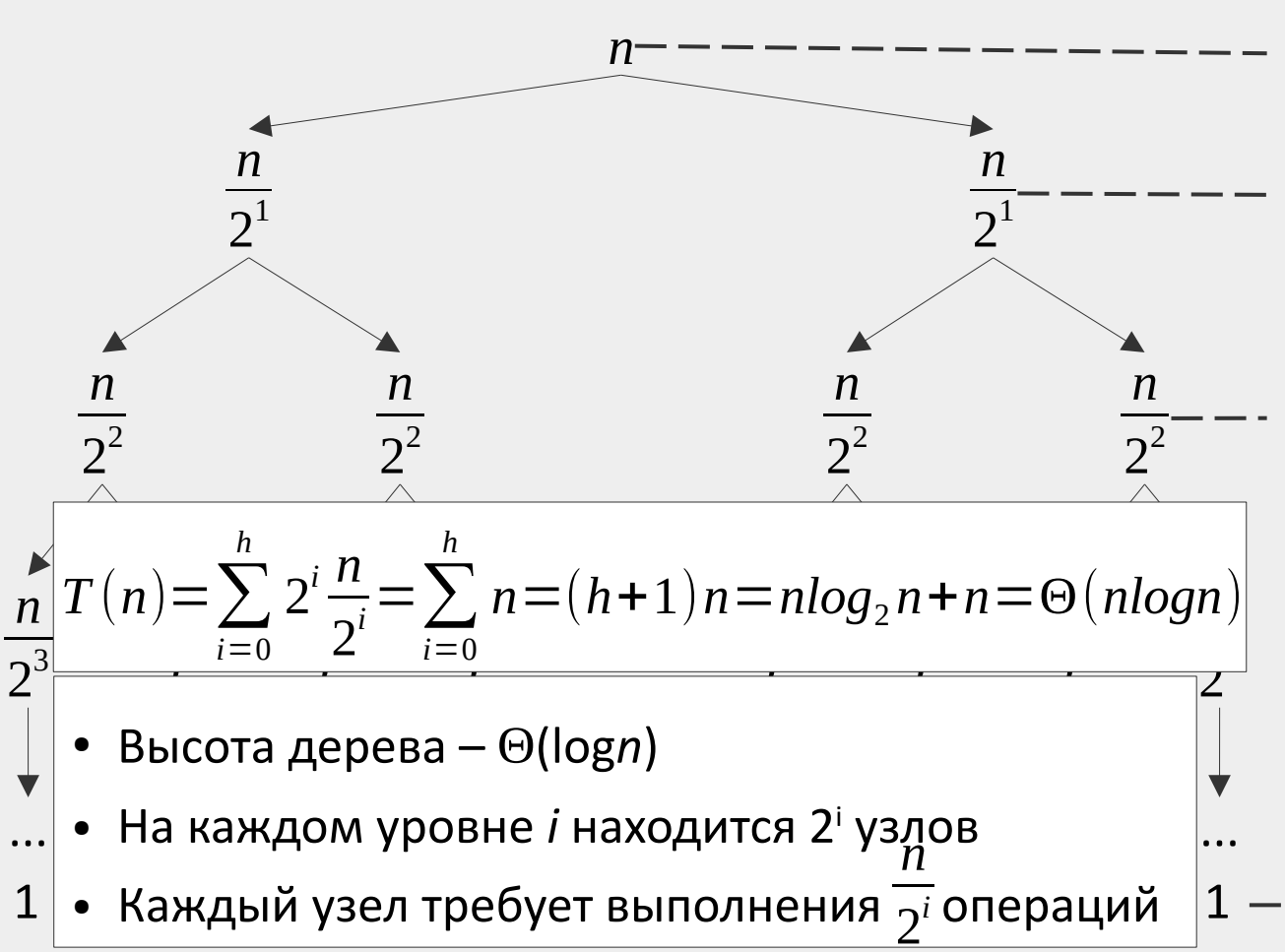
Дерево рекурсивных вызовов сортировки слиянием



Дерево рекурсивных вызовов сортировки слиянием



Дерево рекурсивных вызовов сортировки слиянием



Уровень 0
Θ(n)

Слияние 2-х подмассивов

Уровень 1
Θ(n)

Слияние 4-х подмассивов

Уровень 2
Θ(n)

Слияние 8-и подмассивов

...

Уровень $h = \log_2 n$
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

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 **обобщенным 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 обобщенном рекуррентном уравнении декомпозиции

$$f(n) = \Theta(n^d), \text{ где } d \geq 0, \text{ то}$$

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

- **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 $a > b^d$:

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

Дальнейшее чтение

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