9. Рекурсия, комплексност на алгоритмите

Лекционен курс “Програмиране на Java”
проф. д-р Станимир Стоянов

Структура на лекцията

- Обща характеристика и видове
- Примери
- Комплексност

Повторение на стъпки на обработката

Итерация: цикли `(while, for ...)`

Рекурсия: решение на проблеми чрез “самоприложение"

Един метод се нарича р е к у р
извиква (директно или ин ам себе си

Каква е рекурсията?

```
int      n) {
    ..  2 (n-2) ..
}
int f2 (int n) {
    ... f1 (n-3)
}
```

индиректна

Приложение на рекурсия

- Индуктивно дефинирани функции
  - **Фибоначи**
  - **Факториел, степен**
  - **…**
- Естествени рекурсивни решения на проблеми
  - **Методи за сортиране**
  - **Кули на Ханой**
  - **…**
- Рекурсивно изграждане на обработваемите данни
  - **Програми (EBNF)-> Компилатор**
  - **Дървета и списъци**

Индуктивно дефинирани функции (1)

Пример: Факториел

fac(n) = 1 * 2 ... * n
Начало: fac(1) = 1
Стъпка: fac(n + 1) = (n + 1) * fac(n)

Пример: Степен

pot(k , n) = k * k * ... * k
Начало: pot (k , 0) = 1
Стъпка: pot (k , n + 1) = k * pot (k , n)

Индуктивно дефинирани функции (2)

Пример: Сума

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n$$

Начало: sum(1) = 1

Стъпка: sum(n + 1) = sum(n) + n + 1

Пример: Фибоначи

Начало: fib(0) = 1 fib(1) = 1

Стъпка: fib(n + 1) = fib(n) + fib(n - 1)

Рекурсивен метод: power()

```
static int power (int k, int n) {
    if (n == 0)
        return 1;
    else
        return k * power (k , n - 1);
}
```

→ Както при итеративното решение: брой на умножения = n

→ Съществува по-ефективно решение (по-късно): брой около $\log_2 n$

Какво е логаритъм?

$$x = \log_a b$$

Логаритъм

от Уикипедия, свободната енциклопедия

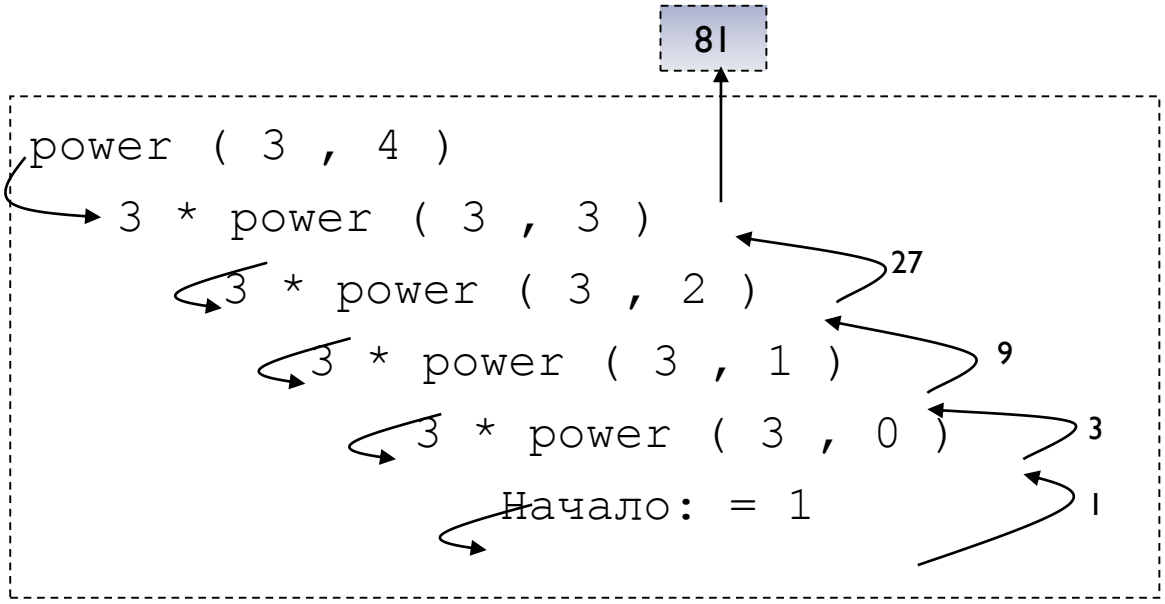
Логаритъмът е степента (*x*), на която трябва да бъде повдигната основата (*a*), за да се получи числото *b*: $x = \log_a b$ (чете се: *x* е равно на логаритъм от *b* при основа *a*). Например, логаритъм от 1000 при основа 10 е 3, защото 1000 е 10 на степен 3.

$$\log_{10} 1000 => 3$$

Извикване на power()

- 4 умножения
- 5 извиквания на методи (време!) едновременно активни

➔ Изграждане на компилатори



За размисъл: по-ефективна реализация на 'power()'

$k^n = (k^{n/2})^2$ - ако n четно

$k^n = (k^{(n-1)/2})^2 * k$ - в противен случай

```
static int power1 (int k, int n) {

    if (n == 0)
        return 1;
    else {
        int t = power1(k, n/2);
        if ((n % 2) == 0)
            return t * t;
        else
            return k * t * t;
    }
}
```

Горната формула се прилага също за нечетни числа – защо ?

Демо: рекурсивен “!”

```
pubic class Factorial {
   public static int fact(int n) {
      if (n == 0) return 1;
      else return n * fact(n-1);
   }

   public static void main(String[] args) {
      System.out.println(fact(3));
   }
}
```

n = 3

среда

fact(3)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
}
```

n = 3

среда

fact(3)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
}
```

n = 3

среда

fact(3)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
}
```

n = 3

среда

fact(3)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 2

среда

fact(2)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
}
```

n = 3

среда

fact(3)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 2

среда

fact(2)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
}
```

n = 3

среда

fact(3)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 2

среда

fact(2)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
}
```

n = 3

среда

fact(3)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 2

среда

fact(2)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 1

среда

fact(1)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
}
```

n = 3

среда

fact(3)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 2

среда

fact(2)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 1

среда

fact(1)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
}
```

n = 3

среда

fact(3)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 2

среда

fact(2)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 1

среда

fact(1)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
}
```

n = 3

среда

fact(3)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 2

среда

fact(2)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 1

среда

fact(1)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 0

среда

fact(0)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
}
```

n = 3

среда

fact(3)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 2

среда

fact(2)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 1

среда

fact(1)

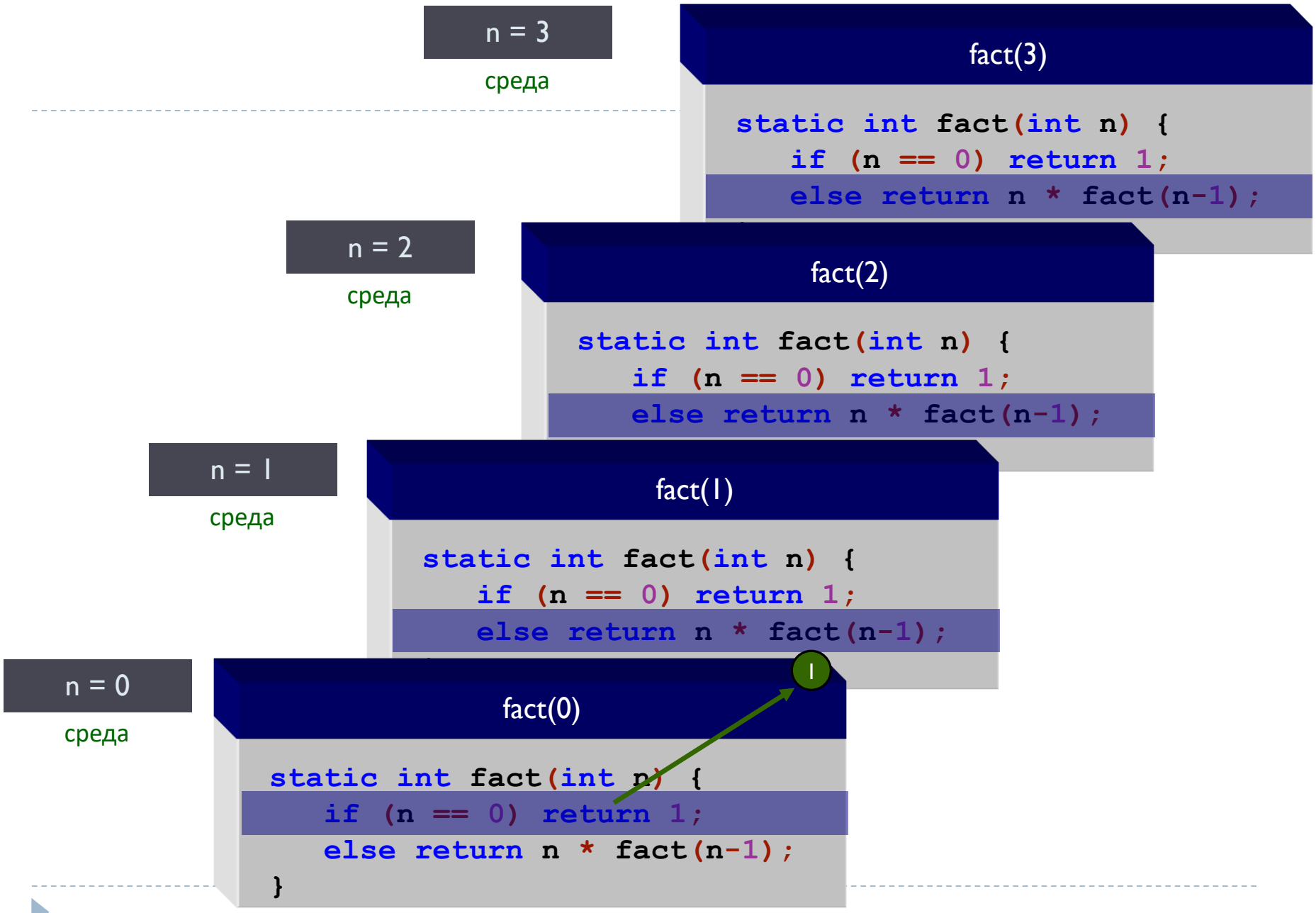
```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

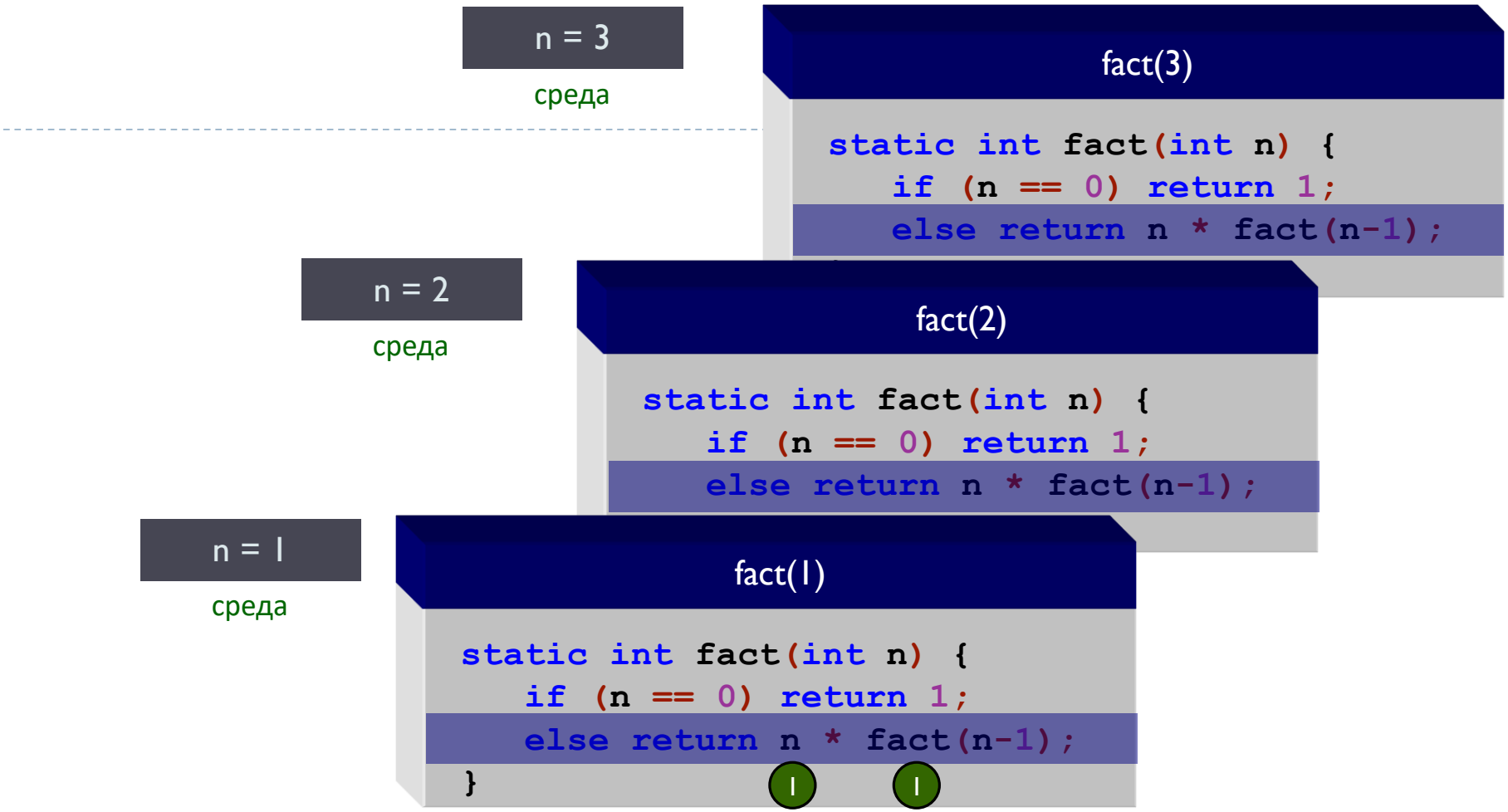
n = 0

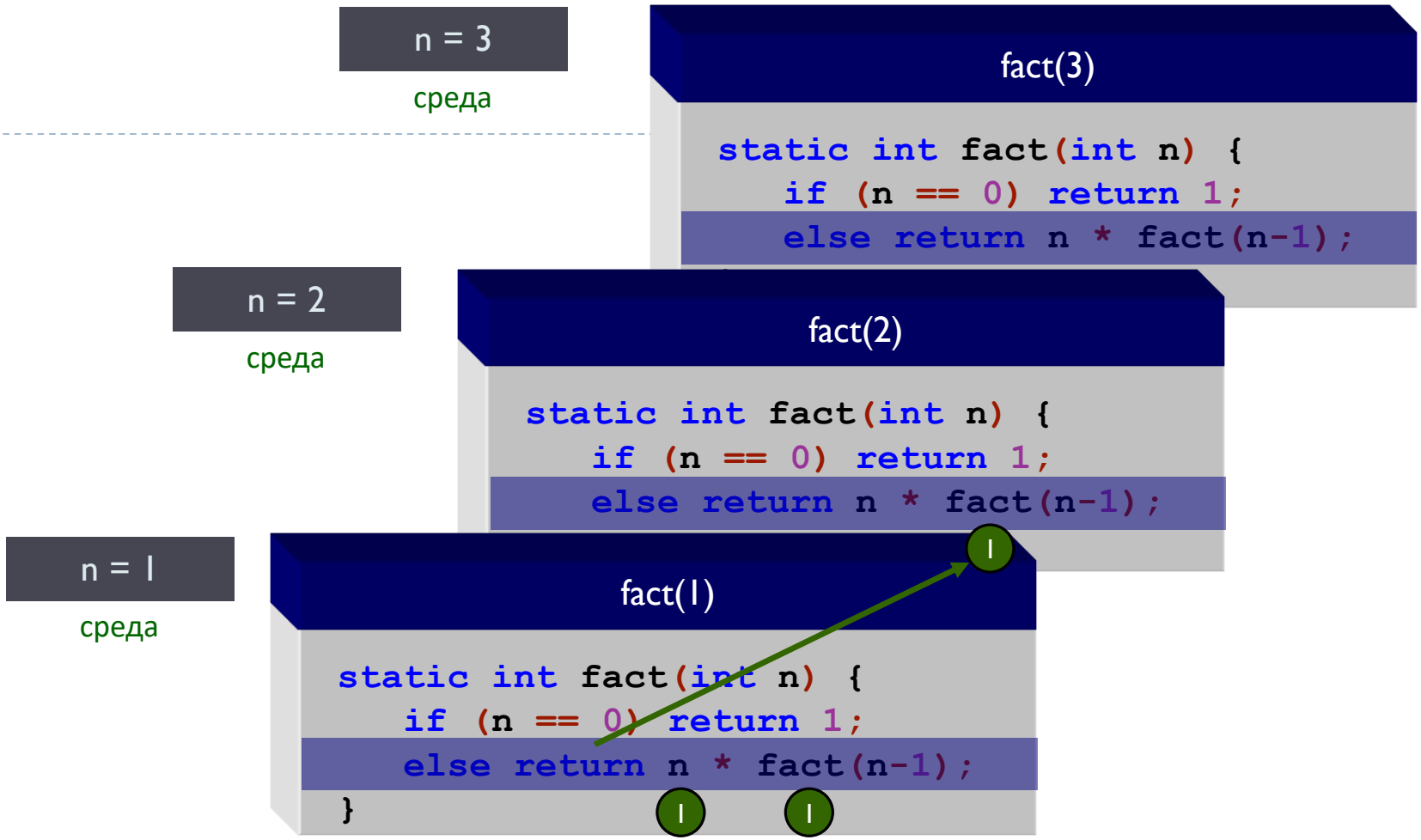
среда

fact(0)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
}
```


n = 3
среда
fact(3)
static int fact(int n) {
if (n == 0) return 1;
else return n * fact(n-1);
n = 2
среда
fact(2)
static int fact(int n) {
if (n == 0) return 1;
else return n * fact(n-1);
n = 1
среда
fact(1)
static int fact(int n) {
if (n == 0) return 1;
else return n * fact(n-1);
n = 0
среда
fact(0)
1
static int fact(int n) {
if (n == 0) return 1;
else return n * fact(n-1);
}


n = 3
среда
fact(3)
static int fact(int n) {
if (n == 0) return 1;
else return n * fact(n-1);
n = 2
среда
fact(2)
static int fact(int n) {
if (n == 0) return 1;
else return n * fact(n-1);
n = 1
среда
fact(1)
static int fact(int n) {
if (n == 0) return 1;
else return n * fact(n-1);
}
1
1


n = 3
среда
fact(3)
static int fact(int n) {
if (n == 0) return 1;
else return n * fact(n-1);
n = 2
среда
fact(2)
static int fact(int n) {
if (n == 0) return 1;
else return n * fact(n-1);
n = 1
среда
fact(1)
static int fact(int n) {
if (n == 0) return 1;
else return n * fact(n-1);
}
1
1
1

n = 3

среда

fact(3)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
```

n = 2

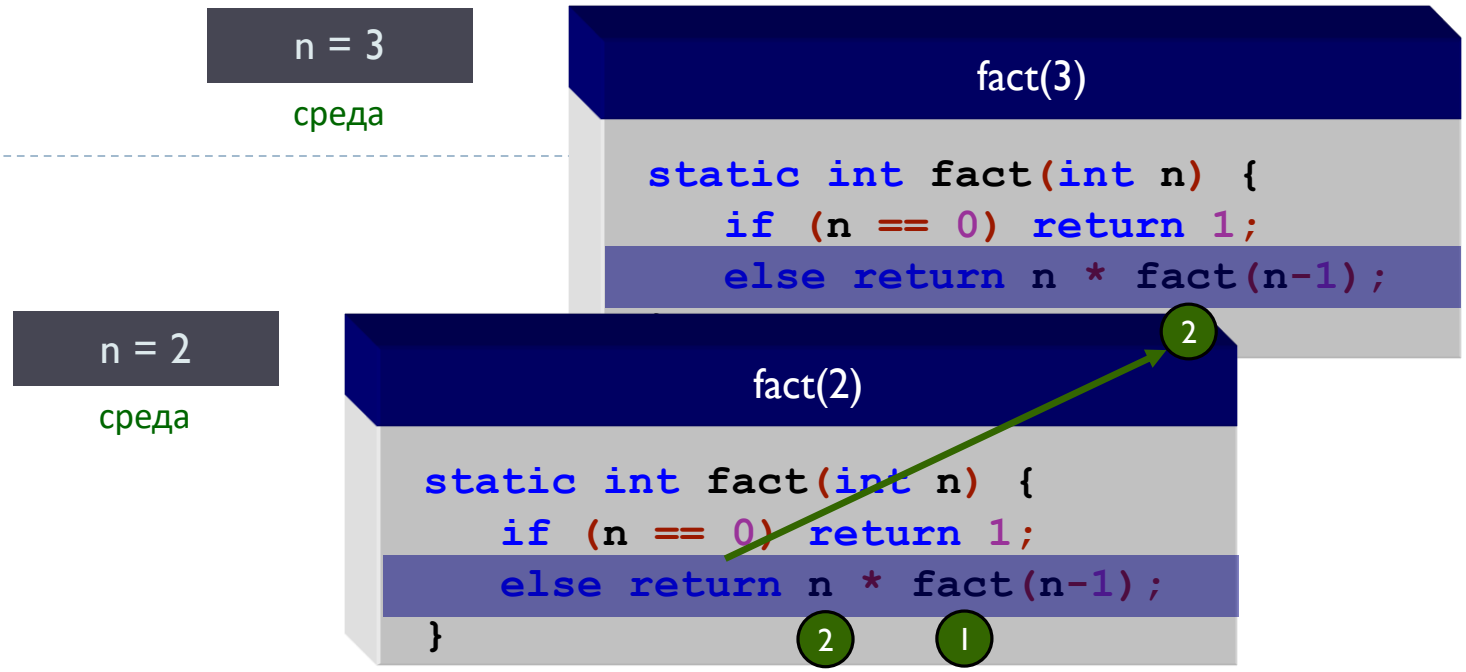
среда

fact(2)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
}
```

2

1


n = 3
среда
fact(3)
static int fact(int n) {
if (n == 0) return 1;
else return n * fact(n-1);
n = 2
среда
fact(2)
static int fact(int n) {
if (n == 0) return 1;
else return n * fact(n-1);
}
2
2
1

n = 3

среда

fact(3)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
}
```

3 2

n = 3

среда

fact(3)

```
static int fact(int n) {
   if (n == 0) return 1;
   else return n * fact(n-1);
}
```

3 2

```
public class Factorial {
   public static int fact(int n) {
      if (n == 0) return 1;
      else return n * fact(n-1);
   }

   public static void main(String[] args) {
      System.out.println(fact(3));
   }
}
```

6

```
% java Factorial
6
```

Сравняване на ефективността:брой на умноженията

- power: n
- power1: около $\log_2 n + 2$

(точно: ?)

т.е. напълно друг клас комплексност

Примери:

n	8	1024	1023	1025	999999
power	8	1024	1023	1025	999999
power1	5	12	20	13	32

напр. int k=2, n=1024 → k^n ?

Забележка: при n = 1024, k > 1 вече overrun
int: стойности до max $2^{31}-1$ → използване клас Integer

Пример: рекурсивно изграждане на данните

Компилатор: парсер (синтактичен анализ)

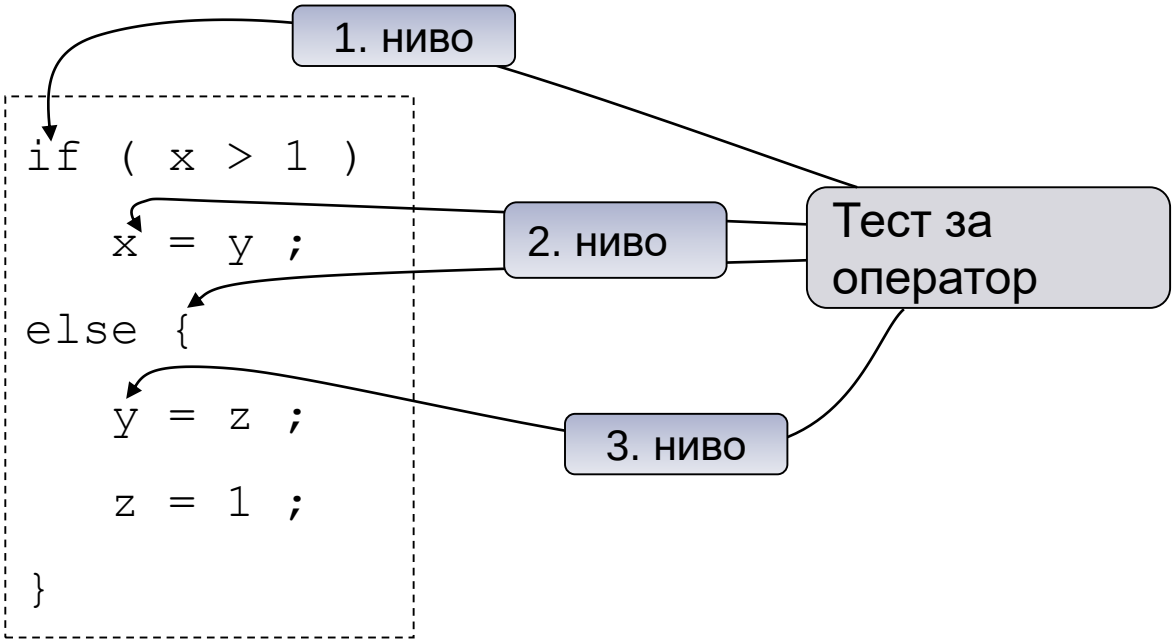
EBNF:

```
Оператор ::=          оператор за избор | ...
Оператор за избор ::=      if ( израз ) оператор
                      [ else оператор ]
```

Парсер:

1. тества, дали е наличен оператор
2. намира if-оператор
3. тества вътре в if-оператора:
   налични ли са 1 (2) вътрешни оператора
   → **рекурсивно извикване на теста за оператор**

Парсер : тества във всяко състояние дали е налична определена синтактична единица



1. Ниво:	Влизане в алгоритъма
2. Ниво:	1. рекурсивно извикване
3. Ниво:	2. рекурсивно извикване

Детайли: упражнения и лекционен курс по компилатори

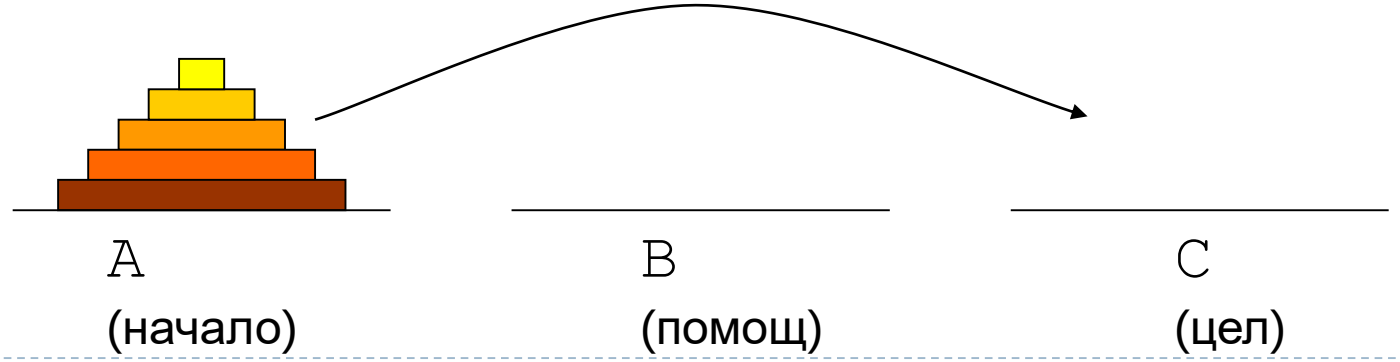
Рекурсивно решаване на проблеми:кули на Ханой

Задача :

Пулове са подредени по големина върху кулата A. Трябва да бъдат преместени върху C като се използва помощна кула B.

Условия:

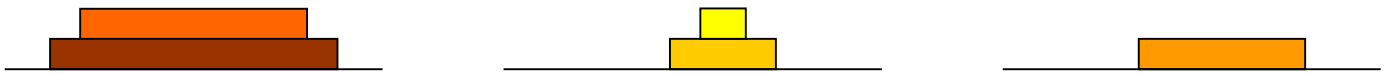
- Винаги се премества само по един пул
- Никога по-голям пул върху по-малък



Задача (продължение)

Търси се: последователност от действия, която води до целта

```
% java Hanoi
Number of pieces: 5
Pices movement:
   from A to C
   from A to B
   from C to B
   from A to C
    . . .
```



A (начало)

B (помощ)

C (цел)

Алгоритъм на решението: итеративен или рекурсивен?

Търси се: **Последователност** от отделни премествания, които водят до целта, т.е. алгоритъмът трябва да се грижи за повторните приложения на единичните стъпки.

→ Итеративен алгоритъм?

възможно - но: не съвсем естествен

→ Рекурсивен алгоритъм?

декомпозиране на задачата на по-прости **подпроблеми,** които ще бъдат рекурсивно обработвани

Рекурсивен алгоритъм

Премести n пула от ’начало’ към ’цел’ посредством ’помощ’

Начало: n = 1 (един пул)

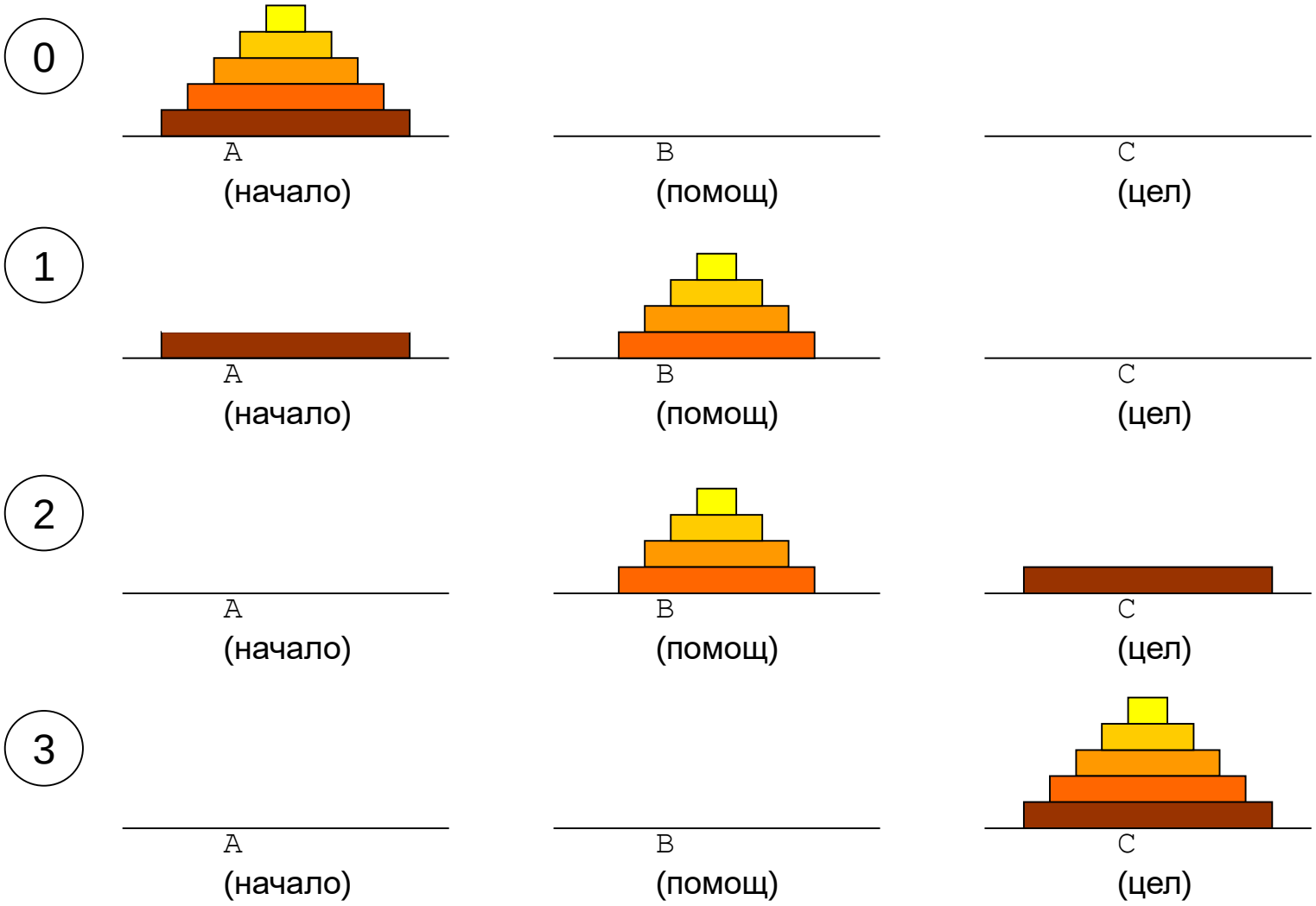
Премести пула директно от ’начало’ към ’цел’

Стъпка: n > 1

1. Премести n - 1 пула*) от ’начало’ към ’помощ’ през ’цел’
2. Премести един (по-голям) пул директно от ’начало’ към ’цел’
3. Премести n - 1 пула*) от ’помощ’ към ’цел’ през ’начало’

*) Неедновременен трансфер от n-1 > 1 пула!
Особено: прилагане на алгоритъма върху по-малко от n пула (рекурсивност)

Алгоритъм


0
A
(начало)
B
(помощ)
C
(цел)
1
A
(начало)
B
(помощ)
C
(цел)
2
A
(начало)
B
(помощ)
C
(цел)
3
A
(начало)
B
(помощ)
C
(цел)

Програма Ханой: рамка

```
public static void main (String argv[]) {
    int n;

    System.out.print("pieces number: ");
    n = Keyboard.readInt();
    if (n > 0) {
        . . .
        move(n,'A','B','C');
    }
    else
        System.out.println("number not positive");
}
```

Местата са означени със символи

Програма Ханой: рекурсивен алгоритъм

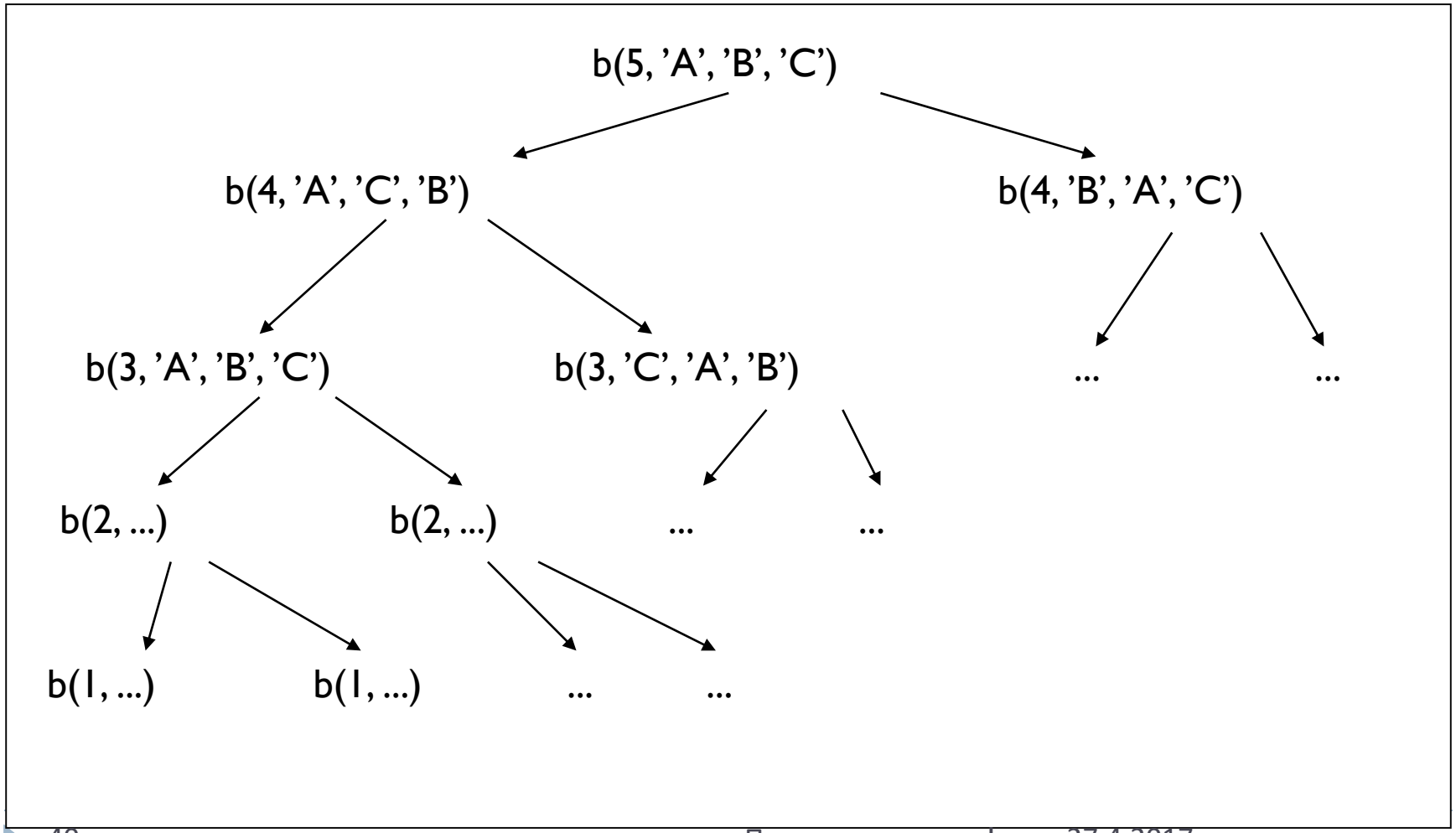
```
static void move
           (int n, char start, char help, char goal) {

  if (n == 1)
    System.out.println(" from " + start + " to " + goal);
  else {
    move(n-1, start, goal, help);
    System.out.println(" from " + start + " to " + goal);
    move(n-1, help, start, goal);
  }
}
```

Общ проблем →
3 подпроблема (с 2 рекурсивни извиквания)

Връзки между извикванията: Ханой

съкр.: move(5, ’A’, ’B’, ’C’) = b(5, ’A’, ’B’, ’C’)



Ханой: рекурсивен → итеративен (съхраняване решаваните проблеми в стек)

Основен принцип:

- В една (циклична) стъпка:
  Реши винаги **един** актуален проблем и отбележи проблемите, които трябва да бъдат решавани по-късно, в междинна памет (стек)
- Един проблем:
  Премести n пула от ‘начало' през ‘помощ' към ‘цел'
- В междинната памет: актуалният проблем е съхранен ‘отгоре'
- Алгоритъм:

```
while (!isEmpty(problemStack))    {
    // обработи най-горния проблем
}
```

n = 1 → печат и задраскване проблема

n > 1 → задраскване проблема + съхраняване три нови проблема

Пример: 'Problem-Stack'

```
move (5, 'A', 'B', 'C');
```

Развитие на стека:

Начало:

```
5  A  B  C
```

1. Циклична стъпка:

```
4  A  C  B
1  A  B  C
4  B  A  C
```

2. Циклична стъпка:

```
3  A  B  C
1  A  C  B
3  C  A  B
1  A  B  C
4  B  A  C
```

Оценка: рекурсия

- Алтернативно *итеративно* решение винаги възможно
  - понеже всичко трябва (в крайна сметка) да се обработва в машинен код – без рекурсия
- Итеративен вариант: често по-бърз (извикване на методи: времевоинтезивни)
- Рекурсивно решение: често по-читаемо
- Рекурсия: не на всяка цена

Само когато се подобрява читаемостта :

- Рекурсивни функции
- Рекурсивни данни
- Рекурсивни решения на проблеми

→ Рекурсивно решение в повечето случаи единствено смислено!

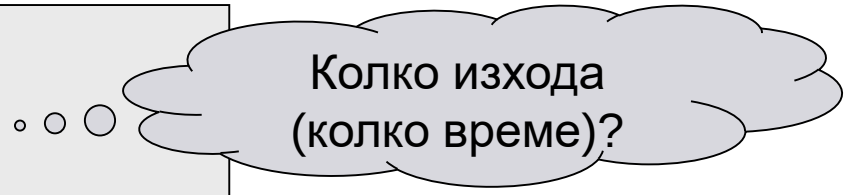
Комплексност на алгоритмите – защо ?

Важно:

- Преди разработване на програмите:
  - Въобще, заслужава ли си разработването на една програма: напр. ще изчислява 100 години
- Преди използване на програмите:
  - Какви входни данни може да "понесе" програмата съотв. run-time ?

Пример: какви входни данни може да “понесе” Ханой?

```
% java Hanoi
Pices number: 1000
Pices movements:
   from A to C
    . . .
```



Дискове	1	5	100	1000
Премествания	1	31	~ 10 Mrd.	$\sim 10^{100}$

Число със
100 нули

Комплексност: Ханой

n	1	2	6	10	1000
Брой премествания	1	3	63	1023	$\sim 10^{100}$

Общо: брой(n) = $2^n - 1$

Доказателство: пълна индукция

Начало: брой(1) = 2 - 1 = 1 (вярно според алгоритъма)

Стъпка: брой(n+1) = $2^{(n+1)} - 1$

$= 2 * 2^n - 1$

$= 2 * (2^n - 1) + 1$

$= 2 *$ брой(n) + 1 (според предпоставка)

→ с това важи:

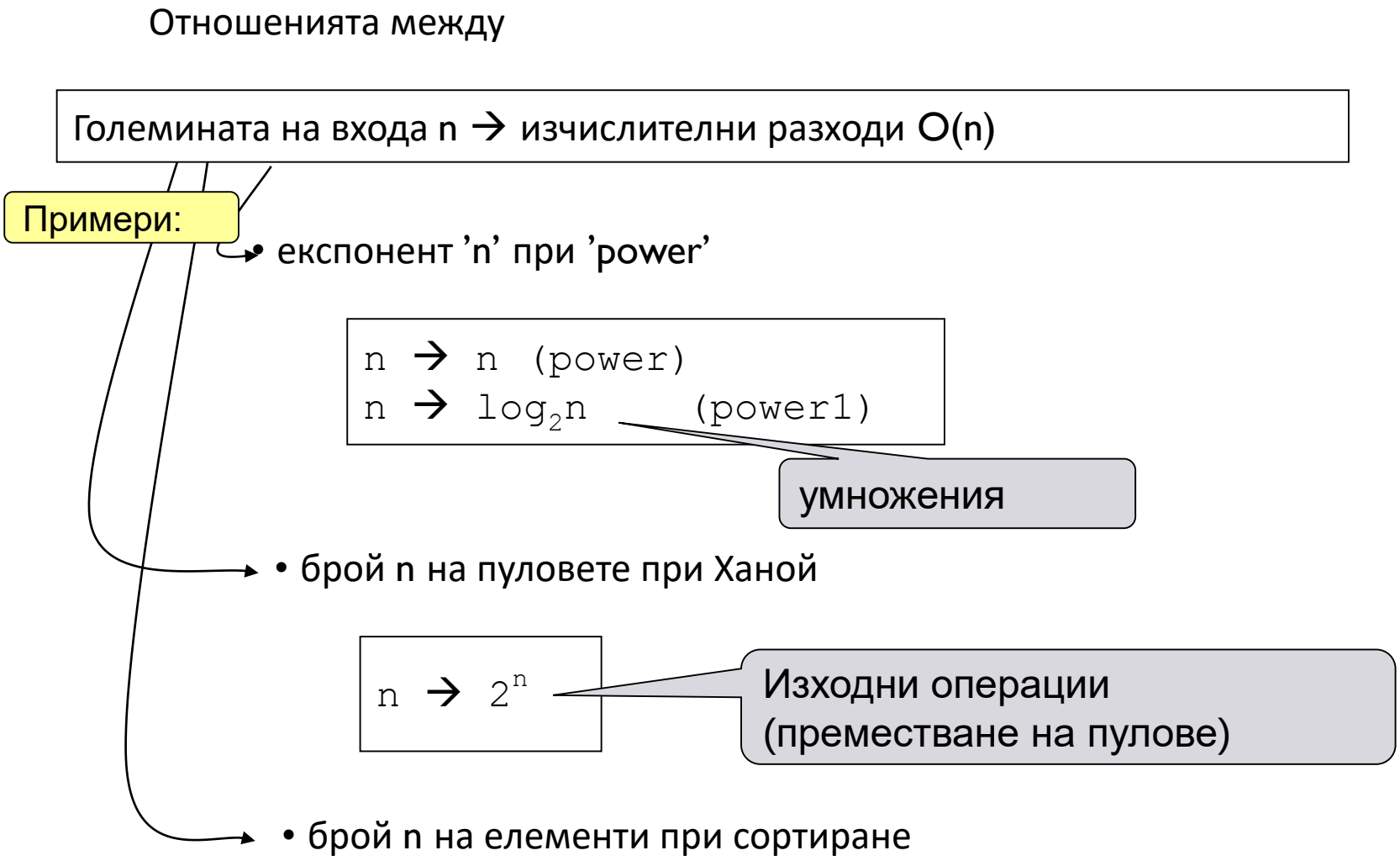
алгоритъмът удвоява досега получения брой + 1

→ Важи според: Ханой - алгоритъм

Подобласти на Информатика и техните връзки

Теория за комплексността:
Теория за изчислителните разходи на алгоритмите

Комплексност на алгоритмите



Класове комплексност на алгоритмите

Алгоритмите се наричат … , когато съотношението между n и изчислителните разходи O се изчислява по дадената формула

константни:	O(n) = константа k
логаритмични: (power1)	$O(n) = k * \log_2 n$
линейни: (power)	O(n) = k * n
n log 2 n:	$O(n) = k * n \log_2 n$
квадратични:	$O(n) = k * n^2$
полиномни:	$O(n) = k * n^m$ (m>1)
експонинциални: (Hanoi)	$O(n) = k * 2^n$

О-Голямо (Big O)

Big O notation

From Wikipedia, the free encyclopedia

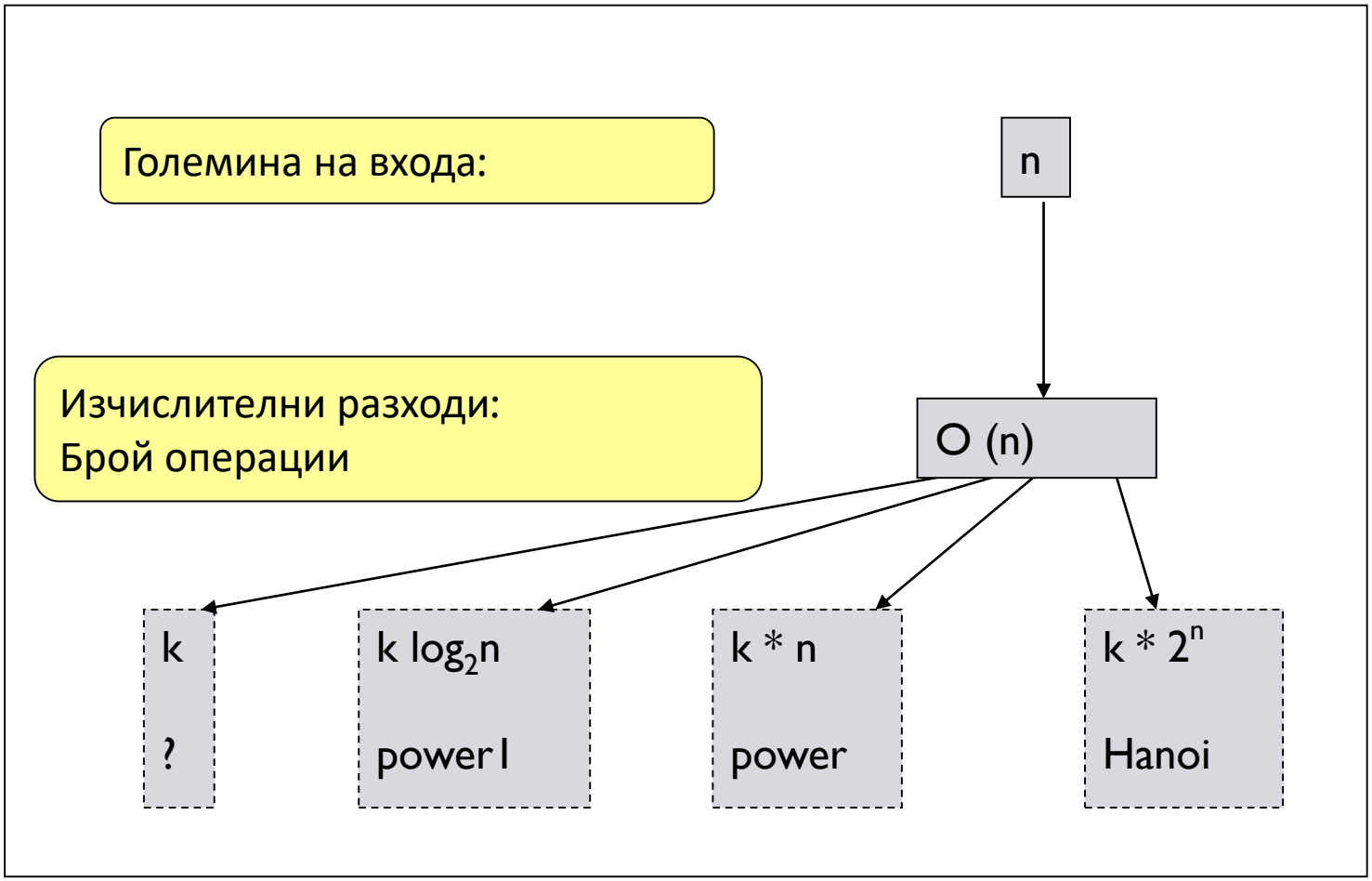
Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. It is a member of a family of notations invented by Paul Bachmann,[1] Edmund Landau,[2] and others, collectively called **Bachmann–Landau notation** or **asymptotic notation**.

In computer science, big O notation is used to classify algorithms according to how their running time or space requirements grow as the input size grows.[3] In analytic number theory, big O notation is often used to express a bound on the difference between an arithmetical function and a better understood approximation; a famous example of such a difference is the remainder term in the prime number theorem.

Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation.

- https://en.wikipedia.org/wiki/Big_O_notation
- http://www.math.bas.bg/~nkirov/2014/NETB201/slides/ch03/ch03_bg.html

Класове комплексност : съотношения



Класове комплексност: избрани стойности

	2	8	10	100	1000
константен	1	1	1	1	1
логаритмичен (power1)	1	3	4	7	10
линеен (power)	2	8	10	100	1000
квадратичен	4	64	100	10.000	1.000.000
експоненциален (Hanoi)	16	256	1024	~10 Mrd.	$\sim 10^{100}$

(някъде приблизителни стойности / костантните могат да се пренебрегнат)

Благодаря за вниманието!

Край лекция 9. “Рекурсия, комплексност на алгоритмите”