

Лекция 6b

Рекурсия и приложения

Основни теми

- Концепция за рекурсия.
- Компютърна реализация на рекурсия в изпълнимия стек
- Примери за приложения на рекурсия.
 - Сортиране на масив с рекурсивни алгоритми
 - Merge sort
 - Quick sort

- 6.1 Въведение**
- 6.2 Представяне на рекурсия**
- 6.3 Примери: Факториели и ред на Fibonacci**
- 6.4 Рекурсия и изпълним стек**
- 6.5 Рекурсивно и итеративно реализиране на метод**
- 6.6 Примери за приложения на рекурсия.**
 - Сортиране на масив с рекурсивни алгоритми
 - Merge sort
 - Quick sort

6.1 Въведение

Обикновено програмите се изпълняват, чрез извикване на методи с различно ниво на вложеност, които могат да се представят с дървовидна структура, в която няма затворени цикли

Рекурсивни методи

- **Тялото на метода съдържа извикване на метода**
 $A \rightarrow A \rightarrow A$
- **Рекурсивното извикване може да е индиректно изпълнено** **$A \rightarrow B \rightarrow C \rightarrow A$**
- **Подходящ при програмиране на проблеми, изискващи извикването един и същ метод**

Примери за рекурсия

Метод за пресмятане на членовете на реда на Фибоначи

Задачата за ханойските кули

Повдигане на цяло число на цяла степен

Отпечатване на елементите на масив или свързан списък в обратен ред

Намиране на най- малката стойност в масив или свързан списък

Намиране на дължината на свързан списък

Обхождане на бинарно дърво

Методи за сортиране и търсене

Фиг. 6.1 Примери за проблеми, позволяващи рекурсия.

6.2 Основен модел на рекурсия

Основни елементи на рекурсията

- Базов(и) (граничен/ни) случай/и
 - Рекурсивен метод, който дава решение само за най- простия случай— the base case
 - Ако рекурсивният метод се извика с базовия случай, методът връща резултат (не се извиква рекурсивно)

- Пример:

При сумиране на множество от N числа, граничният случай е сумиране на множество с един или нула елементи.

6.2 Основен модел на рекурсия

Рекурсивна стъпка

- При извикване на рекурсивния метод за решаване на случай, различен от базовия, задачата се разделя на две части— част, която методът дефинира как да изпълни и част, която методът не дефинира как да изпълни (наричана рекурсивна стъпка, извикване)
- Рекурсивна стъпка, извикване- изисквания
 - Трябва да решава зададения първоначален проблем, но в по- прост вид или умален размер
 - Методът извиква себе си за решаване на същия проблем, но с по- малка размерност
 - Обикновено, включва `return statement`

Пример за рекурсивна стъпка

- Сумирането на N елемента се свежда до сумиране на първия елемент и сумата от останалите $N - 1$ елемента

6.2 Основен модел на рекурсия

Индиректна рекурсия

- Възможно е при изпълнението на рекурсивната стъпка да се извика друг метод, който от своя страна да извика рекурсивния метод и така да се затвори цикъла на рекурсията

6.3 Прост пример за рекурсия:

N факториел е произведението

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

като дефинираме

$$1! = 1$$

$$0! = 1.$$

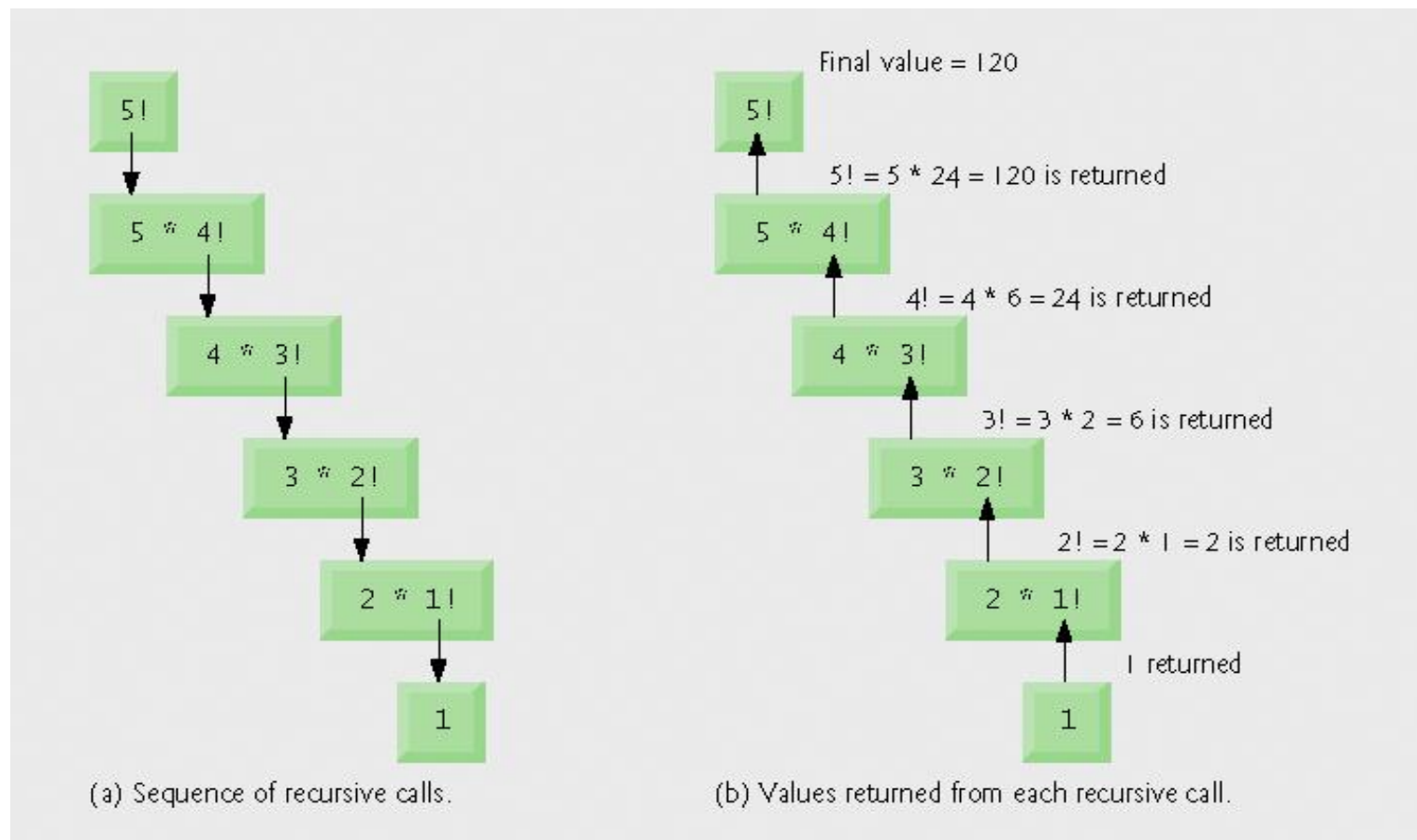
Може да се реализира рекурсивно или итеративно

За рекурсивното решение забелязваме, че:

$$n! = n \cdot (n - 1)!$$

Тук

- *граничният случай е $n \leq 1$*
- *Рекурсивната стъпка е $\cdot (n - 1)!$*



Фиг. 6.2 Рекурсивно пресмятане на $5!$.

6.3 Прост пример за рекурсия:

Безкрайна рекурсия– неограничена последователност от рекурсивни извиквания на метода

- Програмата прекратява поради изчерпване на отделената памет за изпълнение
- Предизвиква се от **пропуснато или неправилно дефинирано гранично условие** , при което рекурсивните извиквания не се сходят до базовия (граничен) случай

```

1 // Fig. 15.3: FactorialCalculator.java
2 // Recursive factorial method.
3
4 public class FactorialCalculator
5 {
6     // recursive method factorial
7     public long factorial( long number )
8     {
9         if ( number <= 1 ) // test for base case
10             return 1; // base cases: 0! = 1 and 1! = 1
11         else // recursion step
12             return number * factorial( number - 1 );
13     } // end method factorial
14
15     // output factorials for values 0-10
16     public void displayFactorials()
17     {
18         // calculate the factorials of 0 through 10
19         for ( int counter = 0; counter <= 10; counter++ )
20             System.out.printf( "%d! = %d\n", counter, factorial( counter ) );
21     } // end method displayFactorials
22 } // end class FactorialCalculator

```

Граничният случай връща 1

Рекурсивната стъпка разделя проблема на две части: **едната** методът дефинира как да изпълни, **другата е аналог** на зададения проблем, но с по-малка размерност

Рекурсивно извикване

Първото извикване на рекурсивния метод

Тази част от проблема за пресмятане на $n!$ е дефинирана в рекурсивния метод



Обичайна грешка при програмиране

Пропускането или неправилното дефиниране на граничния случай води до безкрайна рекурсия докато се изчерпи заделената памет за изпълнение на програмата. Това е **аналогична грешка** на безкрайния цикъл при итеративните методи.

```
1 // Fig. 15.4: FactorialTest.java
2 // Testing the recursive factorial method.
3
4 public class FactorialTest
5 {
6     // calculate factorials of 0-10
7     public static void main( String args[] )
8     {
9         FactorialCalculator factorialCalculator = new FactorialCalculator();
10        factorialCalculator.displayFactorials();
11    } // end main
12 } // end class FactorialTest
```

Пресмятане и извеждане на
факториелите

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```



6.3 Особености на изпълнението

Факториелите нарастват много бързо и излизат от областта на `int` и `long` целочислените типове (например **12!**).

Може да се използват **float** и **double**, но и те не са достатъчни в повечето случаи

За такива цели използваме класове:

[BigInteger](#)

[BigDecimal](#)

6.3 Друг прост пример за рекурсия : Ред на Fibonacci

Редът на Fibonacci **започват** с 0 и 1, а останалите членове са **суми на двата предходни** Fibonacci члена в редицата .

Редицата на Fibonacci numbers се схожда до “златната пропорция” на два съседни члена на редицата-съотношение дължина към височина на картини, прозорци, сгради и пр.

Дефиниция за редицата на Fibonacci :

$$\text{fibonacci}(0) = 0$$

$$\text{fibonacci}(1) = 1$$

$$\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$$

След първото рекурсивно извикване следва много бързо нарастване на рекурсивните извиквания

6.3 Друг прост пример за рекурсия : Ред на Fibonacci

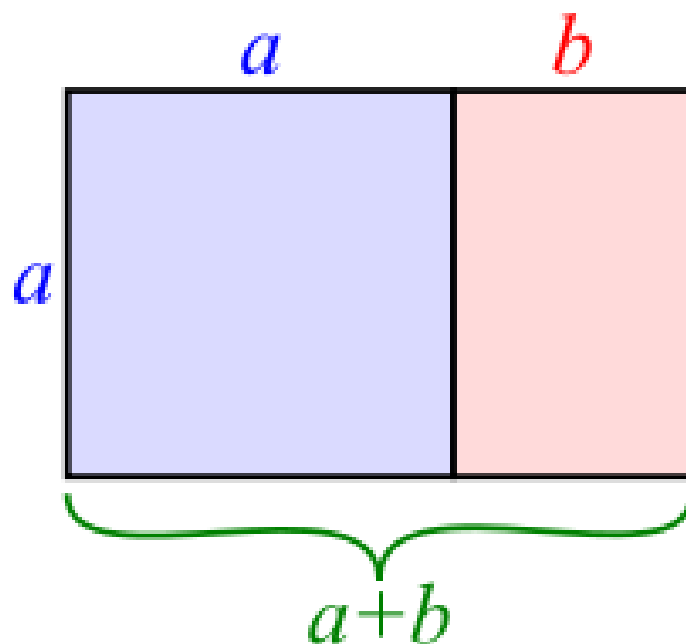
Две величини a и b са в „златна пропорция“, когато **тяхното отношение е равно на отношението на сумата им към по-голямото от тях т.е.**

$$\frac{a+b}{a} = \frac{a}{b} \stackrel{\text{def}}{=} \varphi,$$

откъдето „златна пропорция“ е,

$$\varphi = \frac{1 + \sqrt{5}}{2} = 1.6180339887 \dots$$

$$F(n) = \frac{\varphi^n - (1 - \varphi)^n}{\sqrt{5}} = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}, \quad \lim_{n \rightarrow \infty} \frac{F(n+1)}{F(n)} = \varphi.$$



```
1 // Fig. 15.5: FibonacciCalculator.java
2 // Recursive fibonacci method.
3
4 public class FibonacciCalculator
5 {
6     // recursive declaration of method fibonacci
7     public long fibonacci( long number )
8     {
9         if ( ( number == 0 ) || ( number == 1 ) ) // base cases
10             return number;
11         else // recursion step
12             return fibonacci( number - 1 ) + fibonacci( number - 2 );
13     } // end method fibonacci
14
15     public void displayFibonacci()
16     {
17         for ( int counter = 0; counter <= 10; counter++ )
18             System.out.printf( "Fibonacci of %d is: %d\n", counter,
19                               fibonacci( counter ) );
20     } // end method displayFibonacci
21 } // end class FibonacciCalculator
```

Две гранични условия

Две рекурсивни извиквания

Първото извикване на рекурсивния
метод

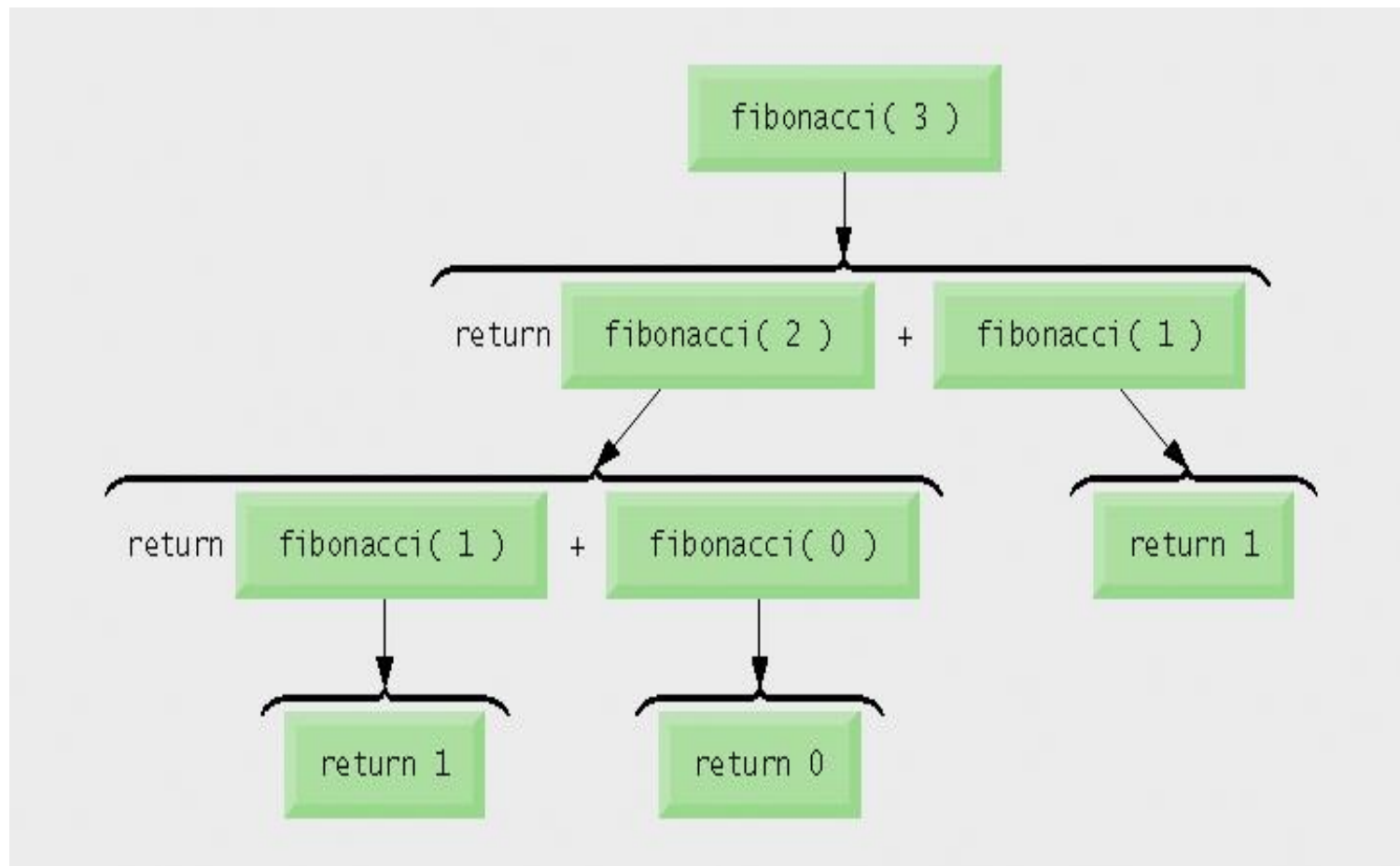


```
1 // Fig. 15.6: FibonacciTest.java
2 // Testing the recursive fibonacci method.
3
4 public class FibonacciTest
5 {
6     public static void main( String args[] )
7     {
8         FibonacciCalculator fibonacciCalculator = new FibonacciCalculator();
9         fibonacciCalculator.displayFibonacci();
10    } // end main
11 } // end class FibonacciTest
```

```
Fibonacci of 0 is: 0
Fibonacci of 1 is: 1
Fibonacci of 2 is: 1
Fibonacci of 3 is: 2
Fibonacci of 4 is: 3
Fibonacci of 5 is: 5
Fibonacci of 6 is: 8
Fibonacci of 7 is: 13
Fibonacci of 8 is: 21
Fibonacci of 9 is: 34
Fibonacci of 10 is: 55
```

Пресмята и извежда членовете на
редицата на Фибоначи





Фиг. 6.7 Множество от рекурсивни извиквания ($n = 3$).

Съвет за качество на програмиране

Избягвайте рекурсивни методи, включващи две и повече рекурсивни извиквания в рекурсивната стъпка по примера на програмата за пресмятане на членовете на Fibonacci, защото те водят експоненциално нарастване на рекурсивните извиквания.

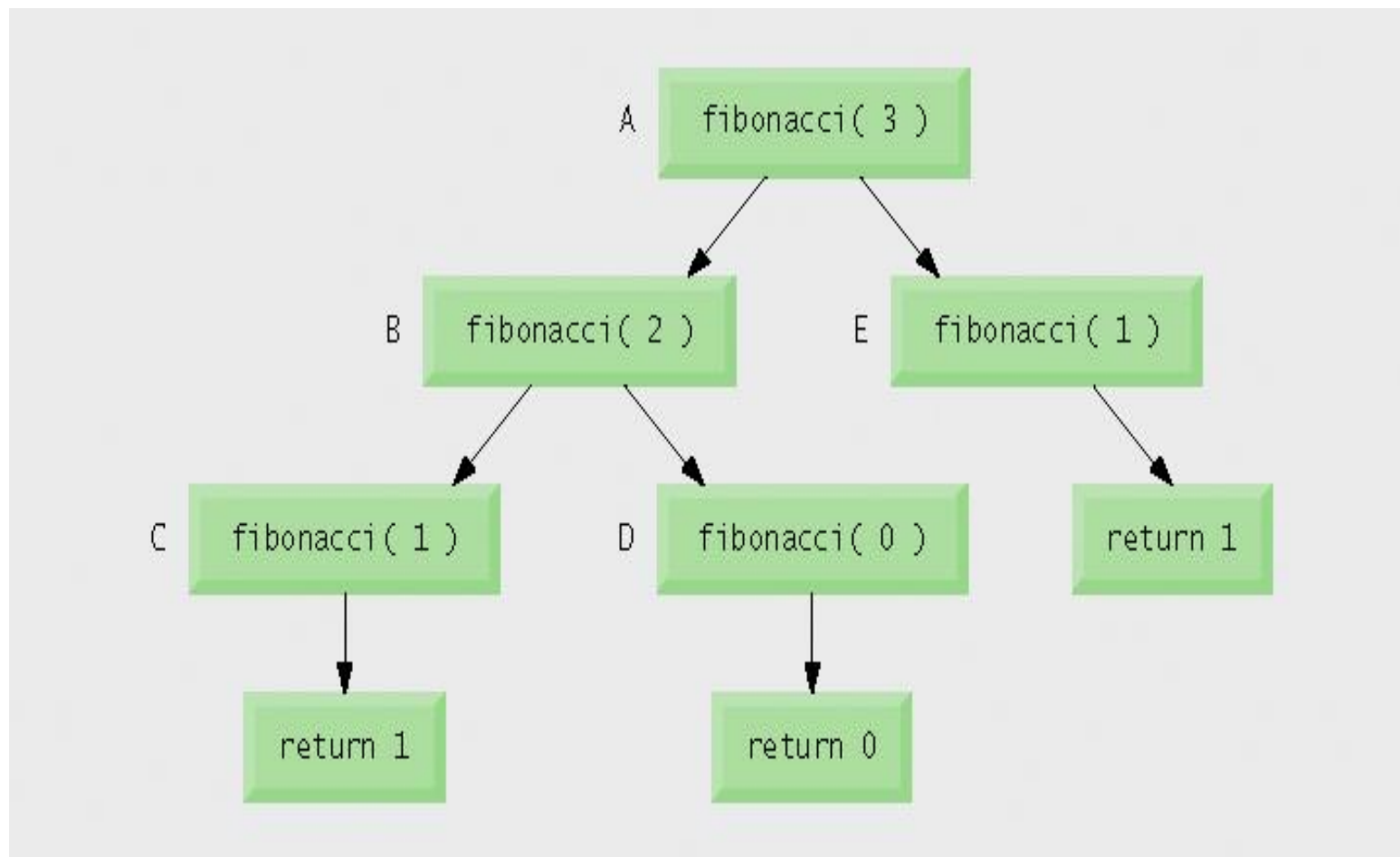
6.4 Рекурсия и изпълним стек

stack структурирана памет **съхранява поредните извиквания на методи и локалните данни**, зададени като аргументи на метода

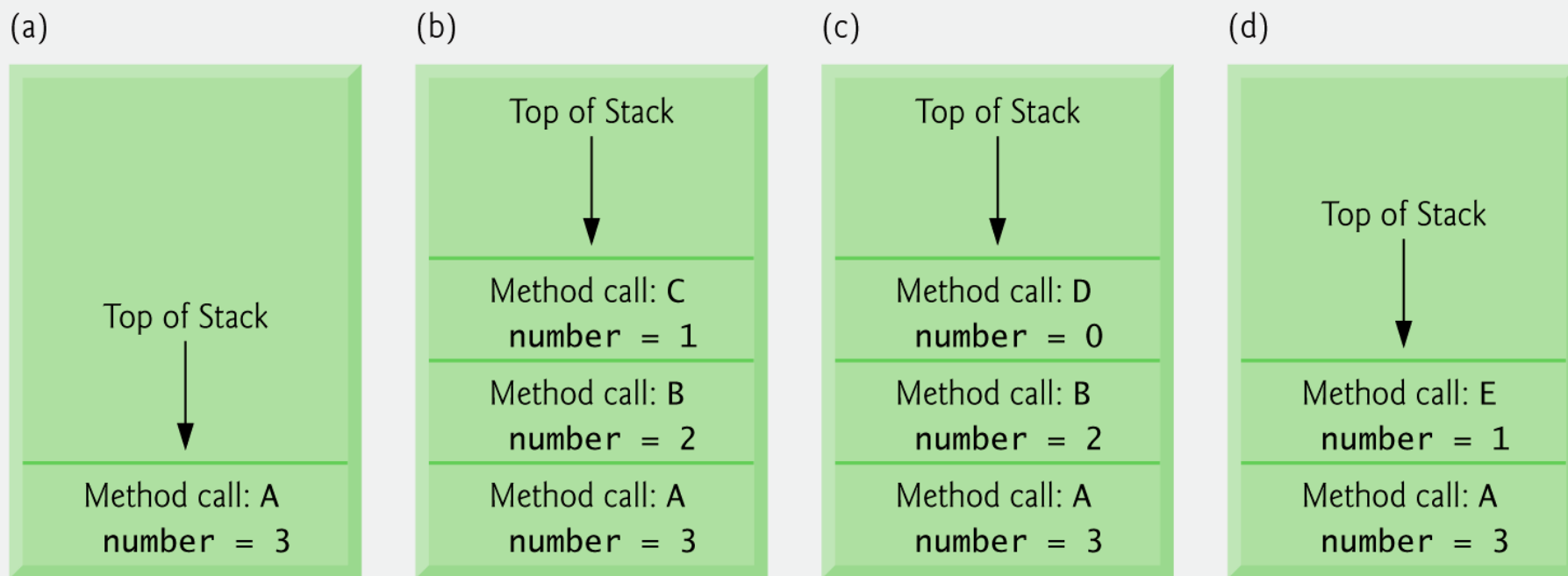
Също както и нерекурсивните методи, **адресите на рекурсивните методи се записват отгоре** на стек-а с извиквания на методи

Когато рекурсивен метод изпълни **return**, **техните записи за действие (activation record) и тези на предхождащите ги рекурсивни извиквания се “изхвърлят”** от стек-а

Текущо изпълняваният метод е този метод, чиито запис за действие е на върха на стек-а



Фиг. 6.8 Извиквания на методи за пресмятане на първите 3 члена от редицата на Фибоначи



Фиг. 6.9 Извикванията на методи в изпълнимия стек на програмата.

/* Problem Binary Search

Input: an array arr sorted in increasing order, an integer key

Output: index of key in arr, -1 if absent

*/

```
public int binarySearch(int[] arr, int key, int low, int high){
    if (high <= low) {
        if (arr[high] == key)
            return low;
        else
            return -1;
    } else {
        int middle = ( high + low + 1 ) / 2;
        if (arr[middle] > key) {
            return binarySearch(arr, key, low, middle - 1 );
        } else {
            if (arr[middle] < key)
                return binarySearch(arr, key, middle + 1, high);
            else
                return middle;
        }
    }
}
```

Три гранични условия

Едно рекурсивно извикване,
но при различни условия

Намерено е съвпадение в средата на
масива



6.6 Рекурсивно и итеративно реализиране на метод

Всеки проблем, решен рекурсивно, може да се реши също и итеративно

Рекурсията, както и итеративният метод използват управляваща структура

- Итерациите използват **команди за цикъл**
- Рекурсията използва **команди за условен преход**

Рекурсията, както и итеративният метод имат условие за край

- Итерацията приключва при **нарушаване на условието за брояча на цикъла**
- Рекурсията приключва при **достигане на граничния случай**

6.5 Рекурсивно и итеративно реализиране на метод

Рекурсията е **скъпа операция** по отношение на ресурси на ОС (процесорно време и памет)

Предлага **много по-интуитивно решение** за проблеми, които по същество са рекурсивно дефинирани

```
1 // Fig. 15.10: FactorialCalculator.java
2 // Iterative factorial method.
3
4 public class FactorialCalculator
5 {
6     // recursive declaration of method factorial
7     public long factorial( long number )
8     {
9         long result = 1;
10
11         // iterative declaration of method factorial
12         for ( long i = number; i >= 1; i-- )
13             result *= i;
14
15         return result;
16     } // end method factorial
17
18     // output factorials for values 0-10
19     public void displayFactorials()
20     {
21         // calculate the factorials of 0 through 10
22         for ( int counter = 0; counter <= 10; counter++ )
23             System.out.printf( "%d! = %d\n", counter, factorial( counter ) );
24     } // end method displayFactorials
25 } // end class FactorialCalculator
```

Итеративно решение на задачата за пресмятане на факториел



```
1 // Fig. 15.11: FactorialTest.java
2 // Testing the iterative factorial method.
3
4 public class FactorialTest
5 {
6     // calculate factorials of 0-10
7     public static void main( String args[] )
8     {
9         FactorialCalculator factorialCalculator = new FactorialCalculator();
10        factorialCalculator.displayFactorials();
11    } // end main
12 } // end class FactorialTest
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```



Software Engineering факт

Рекурсивният подход е за предпочитане когато предлага по-интуитивно решение и задачата е по-лесно разбираема при това решение. Всяко рекурсивно решение се реализира с по-малко код от съответното му итеративно. Има и случай, когато итеративното решение е доста по-трудно реализируемо от рекурсивното.

Съвет за добро програмиране

Избягвайте рекурсия при задачи с изискване за бързодействие и малко памет.

Обичайна грешка при програмиране

Случайното извикване на нерекурсивно моделиран метод (директно или индиректно) чрез рекурсия води до безкрайна рекурсия.

6.6 Примери за приложения на рекурсия

Merge sort на масиви

- По- ефективен метод за сортиране, но и по сложен
- Разделя дадения масив на два по- малки масива с приблизително равен брой елементи, сортира всеки от тези по- малки масива, накрая смесва двата сортирани подмасива в един масив
- Рекурсивен модел на Merge sort
 - Граничният случай е едно елементен масив, който е сортиран
 - Рекурсивната стъпка включва (1)разделяне на масива на две части, (2)сортиране на всяка част и (3) смесването на двете части

```
1 // Figure 16.10: MergeSort.java
2 // Class that creates an array filled with random integers.
3 // Provides a method to sort the array with merge sort.
4 import java.util.Random;
5
6 public class MergeSort
7 {
8     private int[] data; // array of values
9     private static Random generator = new Random();
10
11     // create array of given size and fill with random integers
12     public MergeSort( int size )
13     {
14         data = new int[ size ]; // create space for array
15
16         // fill array with random ints in range 10-99
17         for ( int i = 0; i < size; i++ )
18             data[ i ] = 10 + generator.nextInt( 90 );
19     } // end MergeSort constructor
20
21     // calls recursive split method to begin merge sort
22     public void sort()
23     {
24         sortArray( 0, data.length - 1 ); // split entire array
25     } // end method sort
26
```

Извиква рекурсивният метод



// splits array, sorts subarrays and merges subarrays into sorted array

private void sortArray(int low, int high)

Тества за гранични случаи

{

// test base case; size of array equals 1

if ((high - low) >= 1) // if not base case

Пресмята средата на масива

{

int middle1 = (low + high) / 2; // calculate middle of array

int middle2 = middle1 + 1; // calculate next element over

Пресмята индекса на елемента отдясно на средния

// output split step

System.out.println("split: " + subarray(low, high));

System.out.println(" " + subarray(low, middle1);

System.out.println(" " + subarray(middle2, high));

System.out.println();

Рекурсивно сортиране на първата половина масив

// split array in half; sort each half (recursive calls)

sortArray(low, middle1); // first half of array

sortArray(middle2, high); // second half of array

...сортира и втората половина

// merge two sorted arrays after split calls return

merge (low, middle1, middle2, high);

} // end if

Смесва сортирано двете
половинки

} // end method split



```

51 // merge two sorted subarrays into one sorted subarray
52 private void merge( int left, int middle1, int middle2, int right )
53 {
54     int leftIndex = left; // index into left subarray
55     int rightIndex = middle2; // index into right subarray
56     int combinedIndex = left; // index into temporary working array
57     int[] combined = new int[ data.length ]; // working array
58
59     // output two subarrays before merging
60     System.out.println( "merge:  " + subarray( left, middle1 ) );
61     System.out.println( "        " + subarray( middle2, right ) );
62
63     // merge arrays until reaching end of either
64     while ( leftIndex <= middle1 && rightIndex <= right )
65     {
66         // place smaller of two current elements into result
67         // and move to next space in arrays
68         if ( data[ leftIndex ] <= data[ rightIndex ] )
69             combined[ combinedIndex++ ] = data[ leftIndex++ ];
70         else
71             combined[ combinedIndex++ ] = data[ rightIndex++ ];
72     } // end while
73

```

Индекс за левия масив

Индекс за дясния масив

Индекс за общия масив

Общият масив

Цикъл докато не се изчерпи един
от двата масива- ляв и десен

Определя по- малкия от двата

Записва по- малкия елемент в
общия масив



Ако лявият масив е празен

```

74 // if left array is empty
75 if ( leftIndex == middle2 )
76     // copy in rest of right array
77     while ( rightIndex <= right )
78         combined[ combinedIndex++ ] = data[ rightIndex++ ];
79 else // right array is empty
80     // copy in rest of left array
81     while ( leftIndex <= middle1 )
82         combined[ combinedIndex++ ] = data[ leftIndex++ ];
83
84 // copy values back into original array
85 for ( int i = left; i <= right; i++ )
86     data[ i ] = combined[ i ];
87
88 // output merged array
89 system.out.println( "
90     " + subarray( left, right ) );
91 } // end method merge
92

```

Допълваме общия с останалите елементи от дясния

Ако дясният масив е празен

Допълваме общия с останалите
елементи от лявия

Копираме елементите в
зададения масив



```
93 // method to output certain values in array
94 public String subarray( int low, int high )
95 {
96     StringBuffer temporary = new StringBuffer();
97
98     // output spaces for alignment
99     for ( int i = 0; i < low; i++ )
100         temporary.append( "  " );
101
102     // output elements left in array
103     for ( int i = low; i <= high; i++ )
104         temporary.append( " " + data[ i ] );
105
106     return temporary.toString();
107 } // end method subarray
108
109 // method to output values in array
110 public String toString()
111 {
112     return subarray( 0, data.length - 1 );
113 } // end method toString
114} // end class MergeSort
```



```
1 // Figure 16.11: MergeSortTest.java
2 // Test the merge sort class.
3
4 public class MergeSortTest
5 {
6     public static void main( String[] args )
7     {
8         // create object to perform merge sort
9         MergeSort sortArray = new MergeSort( 10 );
10
11         // print unsorted array
12         System.out.println( "Unsorted:" + sortArray + "\n" );
13
14         sortArray.sort(); // sort array
15
16         // print sorted array
17         System.out.println( "Sorted:  " + sortArray );
18     } // end main
19 } // end class MergeSortTest
```



Unsorted: 75 56 85 90 49 26 12 48 40 47

split: 75 56 85 90 49 26 12 48 40 47
75 56 85 90 49
26 12 48 40 47

split: 75 56 85 90 49
75 56 85
90 49

split: 75 56 85
75 56
85

split: 75 56
75
56



merge: 75
 56
 56 75

merge: 56 75
 85
 56 75 85

split: 90 49
 90
 49

merge: 90
 49
 49 90

merge: 56 75 85
 49 90
 49 56 75 85 90

split: 26 12 48 40 47
 26 12 48
 40 47

split: 26 12 48
 26 12
 48

split: 26 12
 26
 12



```

merge:          26
                12
            12 26

merge:          12 26
                48
            12 26 48

split:          40 47
                40
                47

merge:          40
                47
            40 47

merge:          12 26 48
                40 47
            12 26 40 47 48

merge:  49 56 75 85 90
        12 26 40 47 48 49 56 75 85 90

Sorted:  12 26 40 47 48 49 56 75 85 90

```

Ефективност на Merge Sort

Merge sort

- Много по- ефективен от сортиране с избор и вмъкване
- Последният merge изисква $n - 1$ сравнения за смесване на сортираните части от масива
- На всяко по- долно ниво има два пъти повече извиквания на merge, и всяко извикване работи с половината от елементите на предишното ниво- $O(n)$ общо сравнения
- Общо нивата на разделяне на половинки е $O(\log n)$
- Общата ефективност е $O(n \log n)$

QuickSort

Quick Sort

Рекурсивно сортиране на едномерен масив:

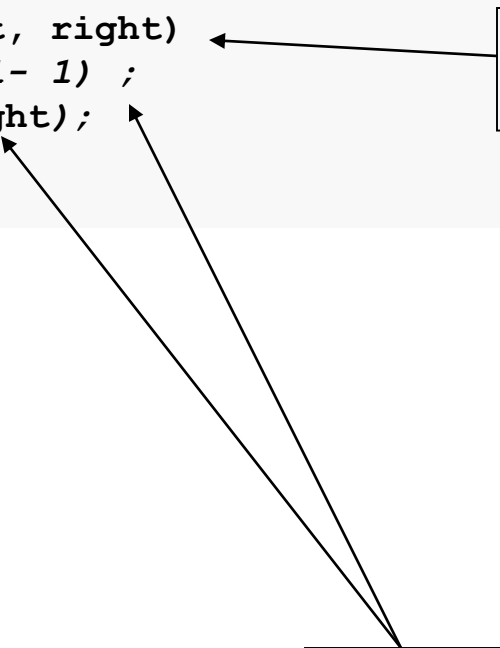
- а) Стъпка 1 (разделяне на елементите):* Взима се първият елемент от несортирания масив и се определя крайното му положение в сортирания масив (т.е., масивът се разделя на две части спрямо този елемент- всички елементи наляво от него са по- малки от него, а всички елементи надясно от него са по- големи от него). Така получаваме един елемент на неговото си място в сортирания масив и два несортирани масива отляво и отдясно на него.**
- б) Стъпка 2 (рекурсия):* Изпълнява стъпка 1 за всеки от двата несортирани масива.**

Псевдокод на алгоритъма

QuickSort

```
procedure quicksort(arr: integer[], left, right: integer);  
  var i;  
  begin  
    if right > left then  
      begin  
        i:=partition(arr, left, right)  
        quicksort (arr, left, i- 1) ;  
        quicksort(arr, i+1, right);  
      end  
    end  
  end
```

Разделяне на масива на две
несортирани части



Рекурсивна стъпка



QuickSort

Quick Sort- пример

Несортиран на едномерен масив:

37 2 6 4 89 8 10 12 68 45

Стъпка 1

- Започваме от най- десния елемент на масива и го сравняваме с първия елемент **37** , докато **намерим елемент по- малък** от **37** при което разменяме **37** с намерения елемент. Първият елемент по- малък от **37** е **12**, затова разменяме **37** и **12**. Получаваме

12 2 6 4 89 8 10 **37** 68 45

.

QuickSort

Quick Sort- пример

Стъпка 1 *продължение*

12 2 6 4 89 8 10 **37** 68 45

Продължаваме отляво надясно, започвайки от втория елемент да сравняваме с **37** **докато намерим елемент по-голям** от **37**, при което разменяме тези елементи.

Първият по- голям от **37** е **89**, затова разменяме **37** и **89**.

Получаваме

12 2 6 4 **37** 8 10 **89** 68 45

QuickSort

Quick Sort- пример

Стъпка 1 *продължение*

12 2 6 4 **37** 8 10 **89** 68 45

Продължаваме от дясно наляво, но от елемента преди 89, да сравняваме всеки елемент с 37 докато намерим елемент по- малък от 37 , при което разменяме 37 с този елемент. Първият такъв елемент е 10, затова разменяме 37 и 10. Получаваме

12 2 6 4 **10** 8 **37** 89 68 45

QuickSort

Quick Sort- пример

Стъпка 1 край

12 2 6 4 **10** 8 **37** 89 68 45

Продължаваме отляво надясно, започвайки от елемента след 10 да сравняваме с 37 **докато намерим елемент по-голям** от 37 , при което разменяме този елемент с 37.

Понеже в случая няма повече елементи по- големи от 37, и затова при сравняване на 37 със себе си, разбираме че 37 е на мястото си в сортирания масив

12 2 6 4 10 8 **37** 89 68 45

QuickSort

Quick Sort- пример

Стъпка 2

12 2 6 4 10 8 37 89 68 45

След разделянето на масива получаваме две части-
отляво са само елементи по- малки от 37 , отдясно са
само елементи по- голями от 37. Това са несортирани
масиви, за всеки от които прилагаме по- отделно

Стъпка 1

QuickSort

```
public void quickSort(int[] b, int first, int last) {  
    int currentLocation;  
    if(first >=last)  
        return;  
    // place an element  
    currentLocation = partition(b, first, last);  
    // sort the left side  
    quickSort(b, first, currentLocation - 1);  
    // sort the right side  
    quickSort(b, currentLocation + 1, last);  
}
```

Разделяне на масива на две
несортирани части

Рекурсивна стъпка



QuickSort

```
private int partition(int[] c, int left, int right) {  
    int position = left;  
  
    while(true){  
        while ((c[position] <= c[right]) && (position != right))  
            right--;  
        if(position == right)  
            return position;  
  
        if (c[position] > c[right]) {  
            swap(c, position, right);  
            position = right;  
        }  
  
        while (c[left] <= c[position] && left != position)  
            ++left;  
        if(position == left)  
            return position;  
  
        if (c[left] > c[position]) {  
            swap(c, position, left);  
            position = left;  
        }  
    }  
}
```

По- малките елементи остават
преди зададения ляв краен
елемент, а по- голямите остават
след него

Масивът е разделен на две части



QuickSort

```
private void swap(int[] d, int position, int right) {  
    int temp = d[position];  
    d[position] = d[right];  
    d[right] = temp;  
}
```

Метод за размяна на елементи



QuickSort

Quick Sort- ефективност

- В най-лошия случай, при всяко разделяне получаваме два масива единият от които има размерност 1. Ако това става на всяка стъпка, то прилагаме стъпката за разделяне към масиви с дължина $n, n - 1, n - 2, \dots, 1$. тогава за ефективността намираме $O(n^2)$
- Общата ефективност в средния случай, когато на всяка стъпка се обособяват масиви с равни дължини, е $O(n \log n)$

Задачи

Задача 1.

Напишете рекурсивен метод за пресмятане на квадратите на първите n числа

Задачи

Задача 2.

При сортиране на масив с вмъкване се намира най- малкият елемент, после този елемент се разменя с първият елемент. Този процес се повтаря за подмасива образуван от елементите след първия до края на масива и така докато не остане подмасив от един елемент. При всеки пас в масива се поставя по един елемент на неговото място в сортирания масив. За масив от n - елемента, са необходими $n-1$ паса, а за подмасивите са необходими $n-1$ сравнения за намиране на най- малката стойност. При достигане на подмасив от един елемент, сортирането се прекратява.

Напишете рекурсивен метод и клас за тестване на този алгоритъм. Използвайте графичен интерфейс