

10. Търсене и сортиране с масиви

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

Структура на лекцията

- ▶ Обща характеристика
- ▶ Комплексност на алгоритмите
- ▶ Линейно търсене
- ▶ Двоично търсене
- ▶ Методи за сортиране
- ▶ Смесване

Задача: търсене и сортиране

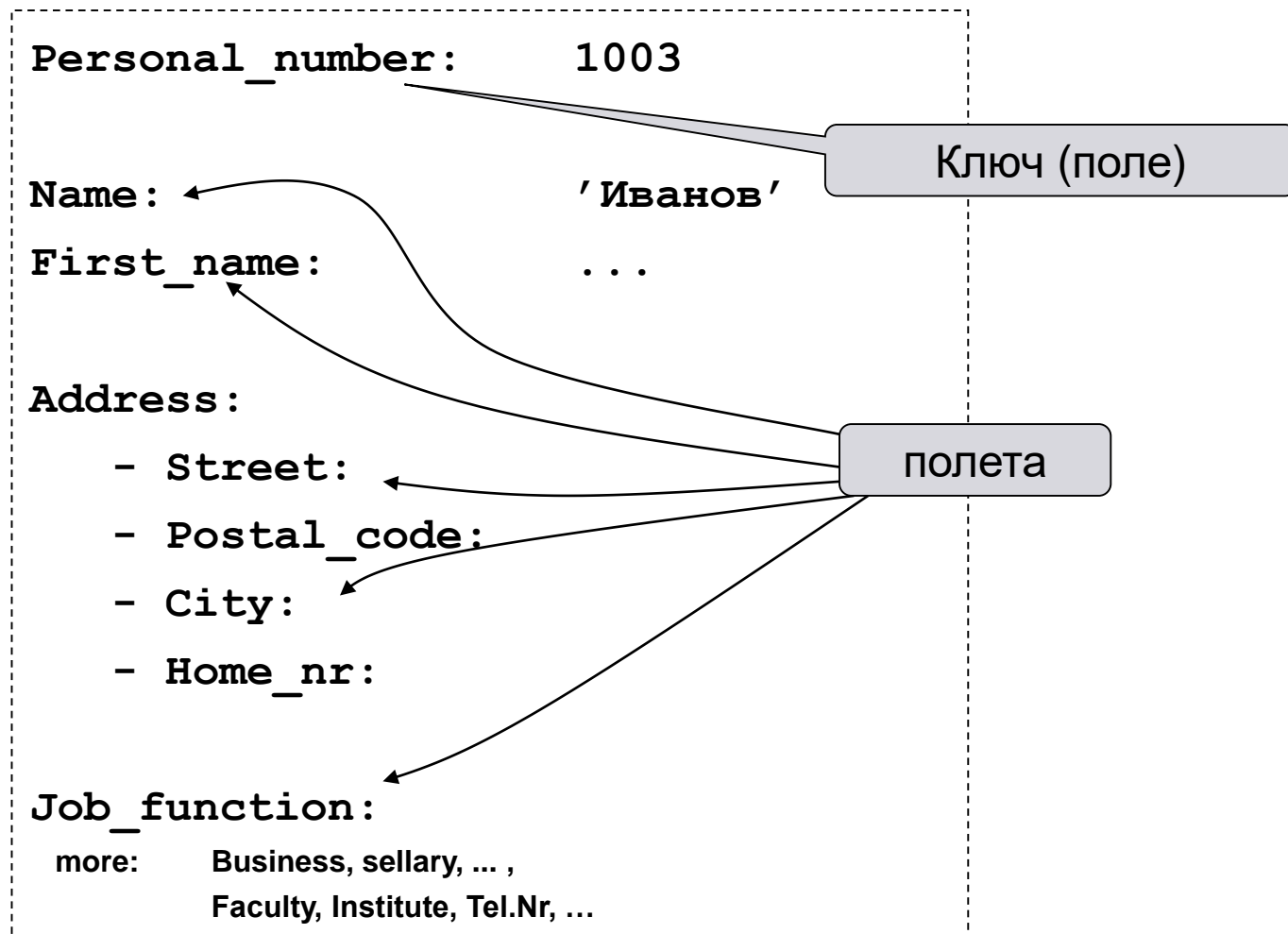
- Основна задача на много софтуерни системи:

Търсене на съхранени данни

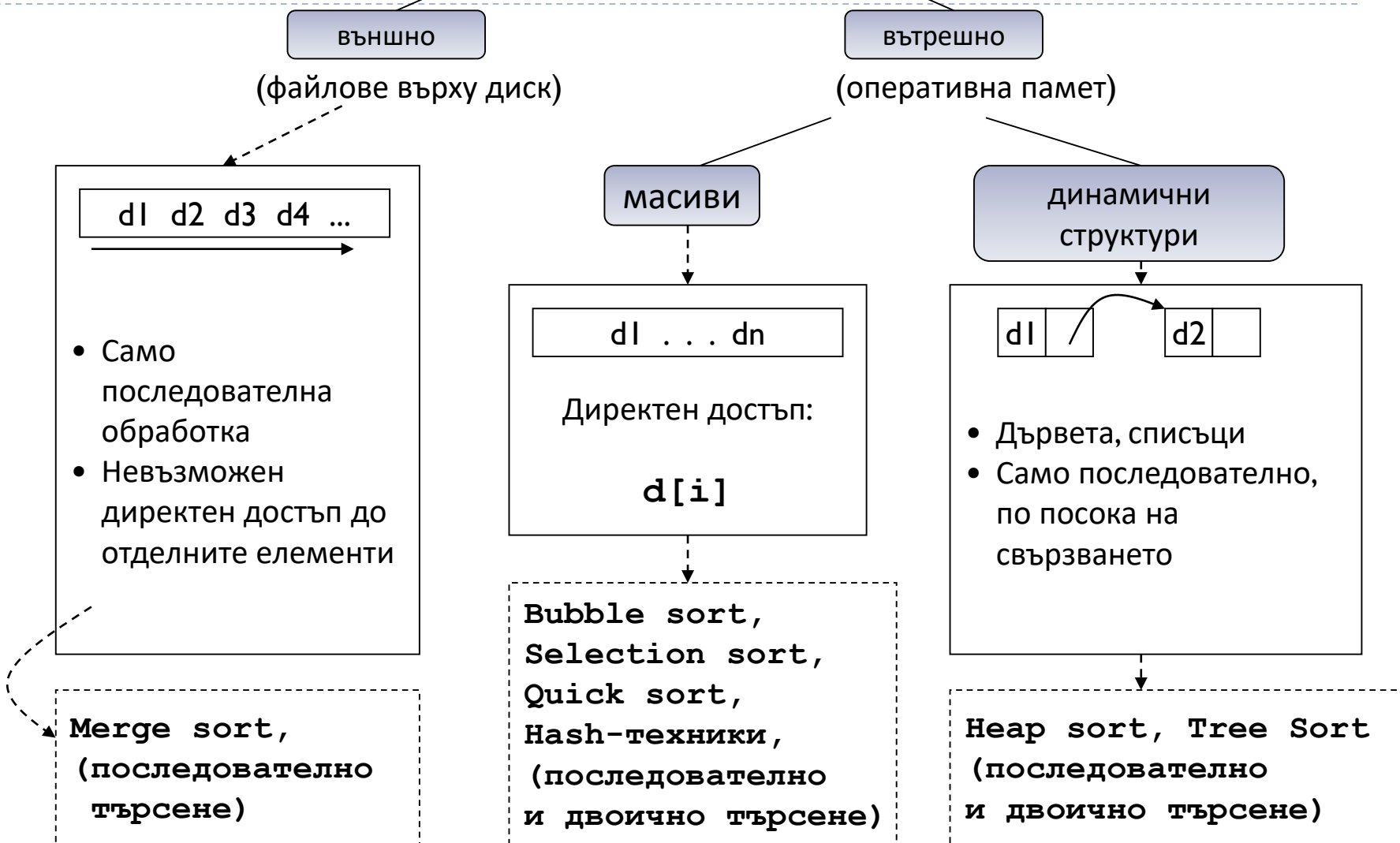
- Базы данни
- Компилатори, Операционни системи
- Управление (жители)
- Банки ... (клиенти, сметки)

- **Сортиране** : опростява последващо търсене
- Съществено: **Ефективност** (комплексност на алгоритмите)
 - Комплексни структури данни
(жители на София: около 2. ... милиона)
 - Отделните данни(записи) също така комплексни
(движение / обмен с високи разходи)

Записи данни: комплексни структури



Къде/как ще се съхраняват записите?



Класове комплексност на алгоритмите

логаритмични: (power1, bin search)	$O(n) = k * \log_2 n$
линейни: (power, lin search)	$O(n) = k * n$
$n \log_2 n$: (Quick sort, Merge sort, Heap sort)	$O(n) = k * n \log_2 n$
квадратичен: (Selection sort, Bubble sort)	$O(n) = k * n^2$
полиномен:	$O(n) = k * n^m \quad (m > 1)$
експоненциален: (Hanoi)	$O(n) = k * 2^n$


Кратка нотация: $O(f(n))$ за $O(n) = k * f(n)$

напр. power1 е $O(\log_2 n)$, Selection sort е $O(n^2)$

Линейно търсене в масиви: основен принцип

Несортирано поле:

100	6	33	77	39	20	20	206	200
-----	---	----	----	----	----	----	-----	-----



По посока на търсене: претърсване всички елементи за търсения (напр. 39)

Среден разход за търсене: $O(n) = \frac{1}{2}n$

т.е. последователното търсене има линейна комплексност

но: 2 милиона жители ...

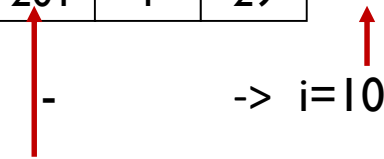
Линейно търсене в масиви (Java)

```
public static void linearSearch(int[] a, int x) {  
  
    int i;  
  
    for (i = 0; (i < a.length) && (x != a[i]); i++);  
  
    if (i == a.length)  
        System.out.println("not found");  
    else  
        System.out.println("found in position " + i);  
}
```

33	73	69	0	22	15	983	201	1	29
----	----	----	---	----	----	-----	-----	---	----

търси 1000: - - - - - - -> i=10

търси 201: - - - -> i=7



Проблем: конюнкцията комутативна?

```
for (i = 0; (i < a.length) && (x != a[i]); i++);
```

същото ?

```
for (i = 0; (x != a[i]) && (i < a.length); i++);
```

Възможна run-time грешка!

търси 1000:

33	73	69	0	22	15	983	201	1	29
----	----	----	---	----	----	-----	-----	---	----

↑
i=10

Съкратено оценяване чрез **&&** съществено

Проблем: for-оператор адекватен?

```
for (i = 0; (i < a.length) && (x != a[i]); i++);
```

- Основна идея за 'for':
 - константен брой повторения
 - условието за прекъсване дава горната граница за брояча
- Тук не е оправдано!


Естествено решение:

```
while ((i < a.length) && (x != a[i]))  
    i++;
```

Двоично търсене: метод на разделянето

Сортиран масив **a**:

2	5	7	10	20	55	77	78	80	100	101
---	---	---	----	----	----	----	----	----	-----	-----



Алгоритъм: търсен елемент x (напр. 80)

- Сравни x със средния елемент $a[m]$ на масива (напр. $m=5$)
 - 1. случай: $x = a[m] \rightarrow$ край: елементът е намерен
 - 2. случай: $x > a[m] \rightarrow$ търсене в десния подмасив
 - 3. случай: $x < a[m] \rightarrow$ търсене в левия подмасив
- Заключение: подмасивът е празен
 \rightarrow край: елементът не е намерен

Демо: Двоично търсене

- ▶ Двоично търсене. Даден е key и сортиран масив $a[]$, намери индекса i , за който $a[i] = key$,
- ▶ Или съобщи, че такъв индекс не съществува.
- ▶ Подвариант: $a[lo] \leq key \leq a[hi]$.
- ▶ Пример: двоично търсене за 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑														↑
lo														hi



Демо: Двоично търсене

- ▶ Двоично търсене. Даден е key и сортиран масив $a[]$, намери индекса i , за който $a[i] = key$,
- ▶ Или съобщи, че такъв индекс не съществува.
- ▶ Подвариант: $a[lo] \leq key \leq a[hi]$.
- ▶ Пример: двоично търсене за 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑							↑							↑
lo							mid							hi



Демо: Двоично търсене

- ▶ Двоично търсене. Даден е key и сортиран масив $a[]$, намери индекса i , за който $a[i] = key$,
- ▶ Или съобщи, че такъв индекс не съществува.
- ▶ Подвариант: $a[lo] \leq key \leq a[hi]$.
- ▶ Пример: двоично търсене за 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑						↑								
lo						hi								



Демо: Двоично търсене

- ▶ Двоично търсене. Даден е key и сортиран масив $a[]$, намери индекса i , за който $a[i] = key$,
- ▶ Или съобщи, че такъв индекс не съществува.
- ▶ Подвариант: $a[lo] \leq key \leq a[hi]$.
- ▶ Пример: двоично търсене за 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑			↑			↑								
lo			mid			hi								



Демо: Двоично търсене

- ▶ Двоично търсене. Даден е key и сортиран масив $a[]$, намери индекса i , за който $a[i] = key$,
- ▶ Или съобщи, че такъв индекс не съществува.
- ▶ Подвариант: $a[lo] \leq key \leq a[hi]$.
- ▶ Пример: двоично търсене за 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				↑		↑								
				lo		hi								



Демо: Двоично търсене

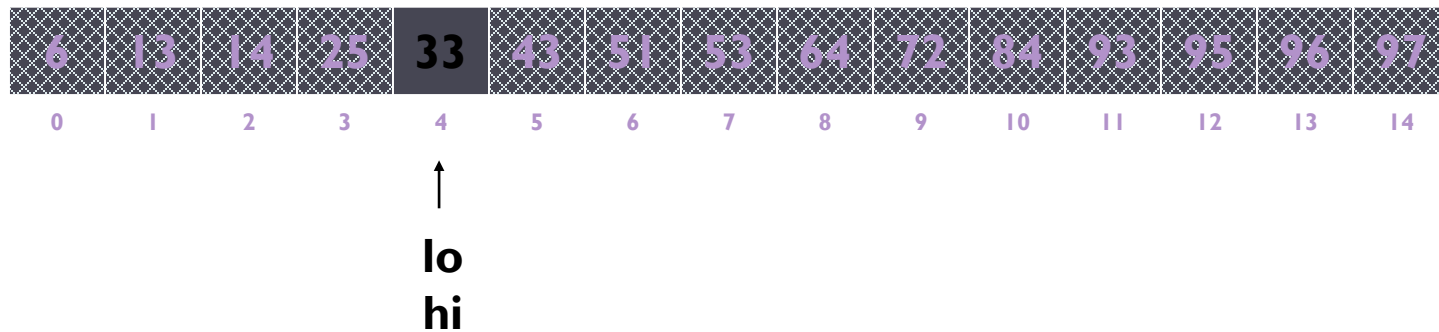
- ▶ Двоично търсене. Даден е key и сортиран масив $a[]$, намери индекса i , за който $a[i] = key$,
- ▶ Или съобщи, че такъв индекс не съществува.
- ▶ Подвариант: $a[lo] \leq key \leq a[hi]$.
- ▶ Пример: двоично търсене за 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				↑	↑	↑								
				lo	mid	hi								



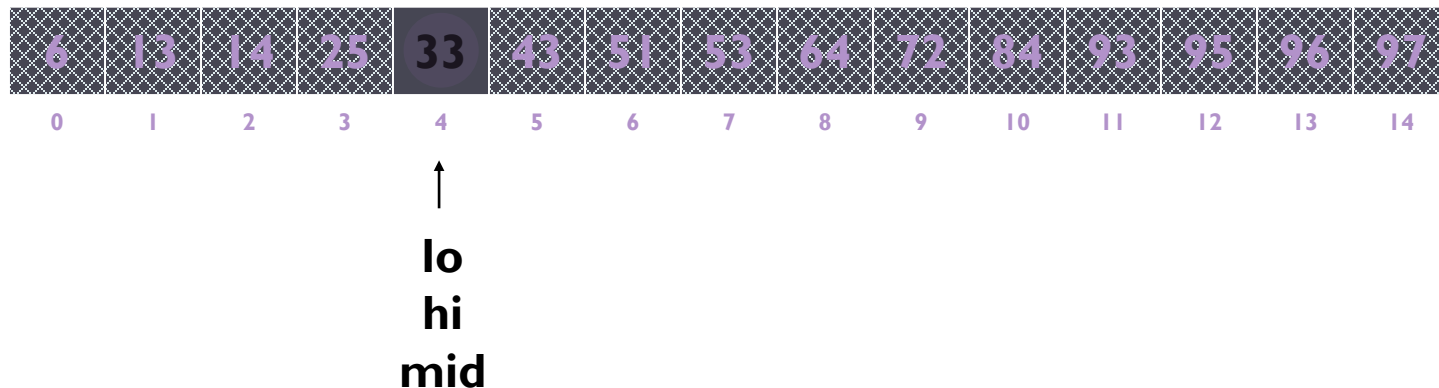
Демо: Двоично търсене

- ▶ Двоично търсене. Даден е key и сортиран масив $a[]$, намери индекса i , за който $a[i] = key$,
- ▶ Или съобщи, че такъв индекс не съществува.
- ▶ Подвариант: $a[lo] \leq key \leq a[hi]$.
- ▶ Пример: двоично търсене за 33.



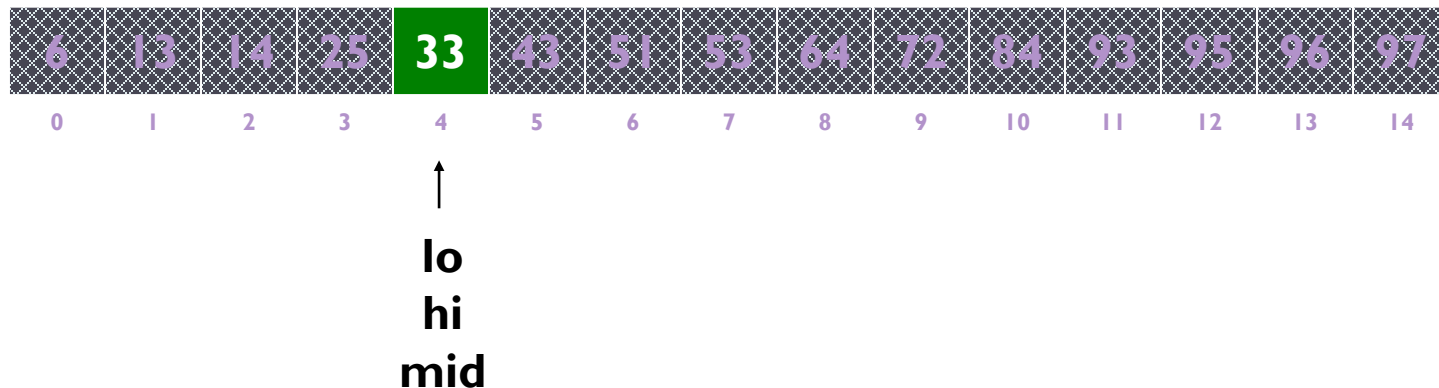
Демо: Двоично търсене

- ▶ Двоично търсене. Даден е key и сортиран масив $a[]$, намери индекса i , за който $a[i] = key$,
- ▶ Или съобщи, че такъв индекс не съществува.
- ▶ Подвариант: $a[lo] \leq key \leq a[hi]$.
- ▶ Пример: двоично търсене за 33.



Демо: Двоично търсене

- ▶ Двоично търсене. Даден е key и сортиран масив $a[]$, намери индекса i , за който $a[i] = key$,
- ▶ Или съобщи, че такъв индекс не съществува.
- ▶ Подвариант: $a[lo] \leq key \leq a[hi]$.
- ▶ Пример: двоично търсене за 33.



Комплексност: двоично търсене

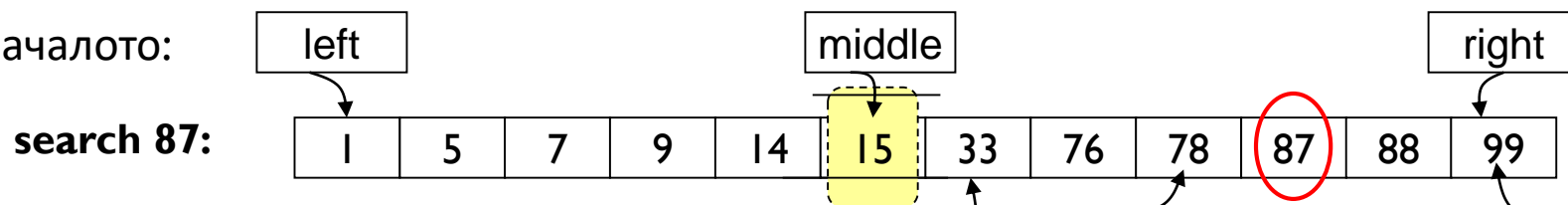
- Брой на разделянията:
 - Най-лошият случай: делим полето, докато само един елемент остане наличен
 - Максимално: $\log_2 n$ стъпки
- Сравнение: линейно и двоично търсене

Брой	100	1024	1 Mio
Линейно (средно)	50	512	500.000
Двоично (максимално)	7	10	20

Двоично търсене: Java-програма

```
public static void binarySearch (int [] a, int x) {  
    int left, right, middle;  
  
    left = 0;  
    right = a.length-1;  
    while (left <= right) {  
        middle = (left + right) / 2;  
        if (a[middle] == x) {  
            System.out.println("found in position " + middle);  
            return;  
        }  
        if (a[middle] < x)  
            left = middle + 1; //search right  
        else  
            right = middle - 1; //search left  
    }  
    System.out.println("not found");  
}
```

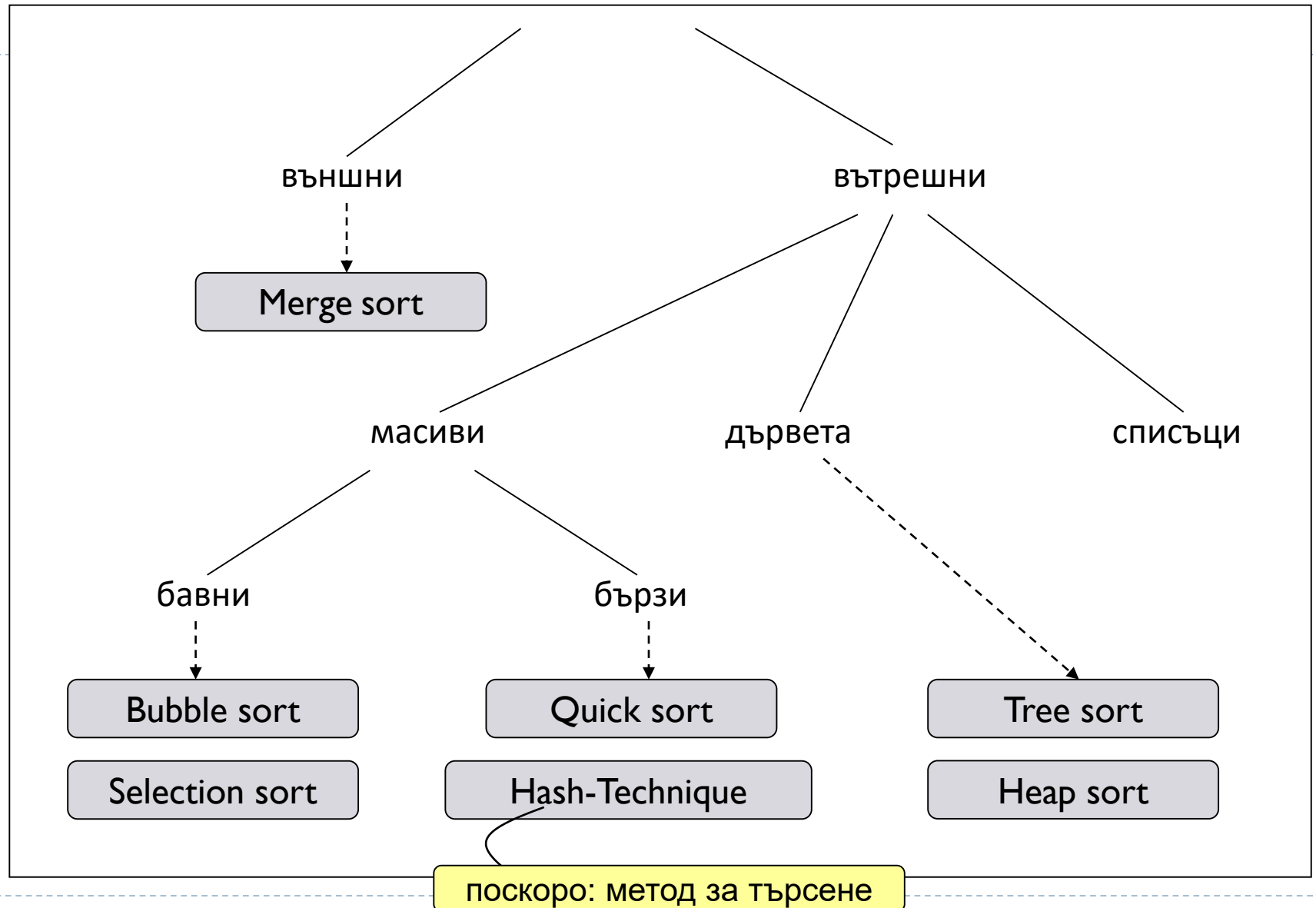
В началото:



След една стъпка:



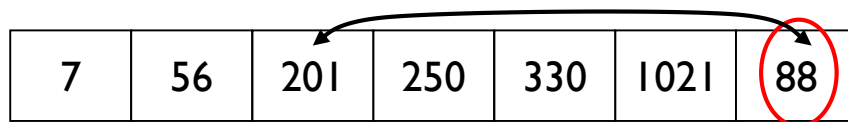
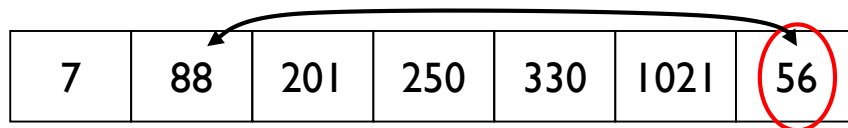
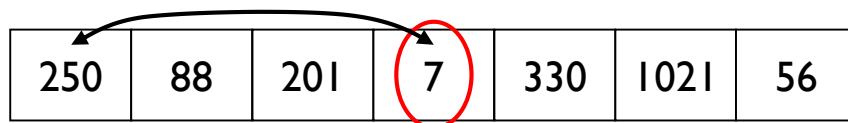
Методи за сортиране



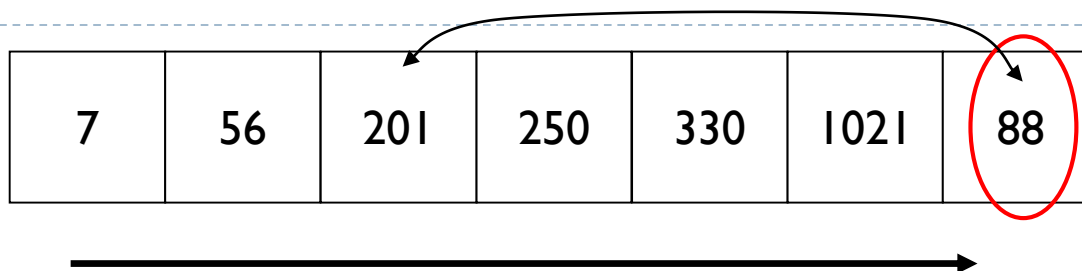
Selection sort: сортиране чрез пряка селекция

Алгоритъм:

1. Търсим най-малкия елемент и го разменяме с първия елемент
2. Търсим втория най-малък елемент и го разменяме с втория елемент
- ... и т.н.



Комплексност: Selection sort



- **Размени: $O(n)$**

(точно: $n - 1$)

- **Сравнения: $O(n^2)$**

(точно:

$$(n-1) + (n-2) + \dots + 1$$

$$= n * (n - 1) / 2$$

$$= (n^2 - n) / 2$$

Класът комплексност се определя чрез най-високата размерност

Клас на комплексност при Selection sort: $O(n^2)$

въпреки точната стойност за броя на сравнения:

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

Клас на комплексност: $O(n^2)$ или $O(n^2 - n)$?

Клас на комплексност: $O(n^2)$

- понеже по-малкият клас при голям n без влияние
- по-високият клас определя размерността
- по-малкият клас се пренебрегва.
- $O(n^2 - n)$ не съществува (не се раглежда)

Класове комплексност: избрани стойности

	2	8	10	100	1000
костантен	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.	~10 ¹⁰⁰

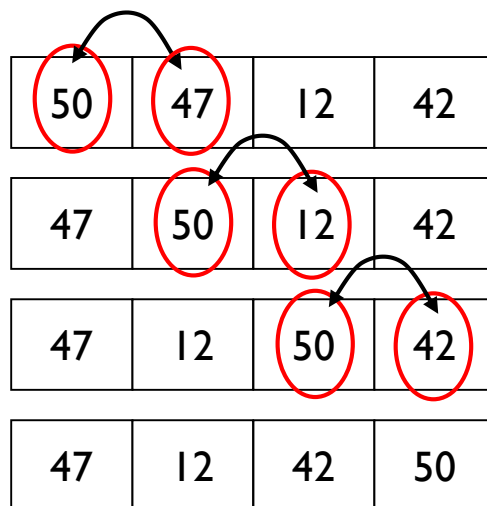
$$n^2 - n$$

(частично приблизителни стойности / константен може да бъде пренебрегнат)

Bubble sort: метод на мехурчето

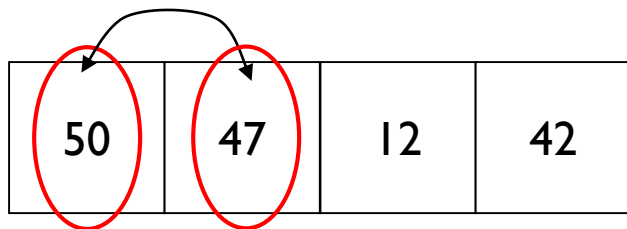
Алгоритъм:

1. Претърсваме масива и разменяме съседните елементи, които не са в правилната последователност.
→ Резултат: най-големият елемент вдясно.
2. Както при 1. – само до предпоследния елемент.
... и т.н.



1. Стъпка

Комплексност: Bubble sort



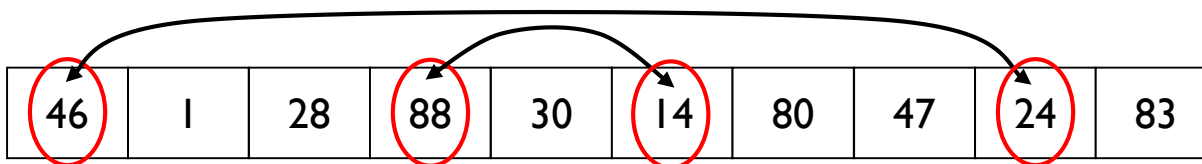
- **Размени: $O(n^2)$**

- min: 0
- max: $(n-1) + (n-2) + \dots + 1$
 $= \frac{1}{2}(n^2 - n)$
- avr.: $\frac{1}{4}(n^2 - n)$

- **Сравнения: $O(n^2)$**

точно: $\frac{1}{2}(n^2 - n)$

Quick sort: основна идея



Алгоритъм (идея):

- Разделяме целия масив посредством размяна на елементи в 2 части:
всички елементи от лявата част са \leq от всички елементи от дясната част
- Сортираме двете части независимо една от друга (рекурсивно).

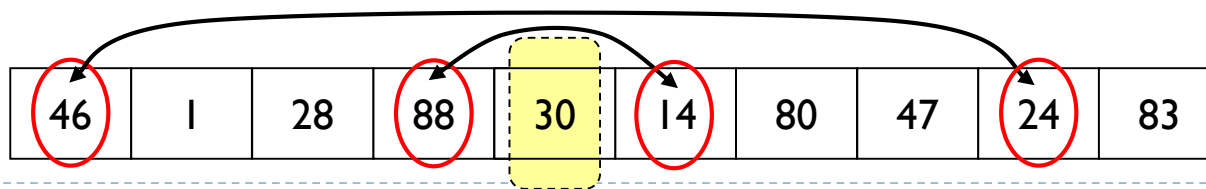


Quick sort: разделяне на две части

Разделяме целия масив чрез размяна на елементите в 2 части:
Всички елементи на лявата част \leq всички елементи на дясната част

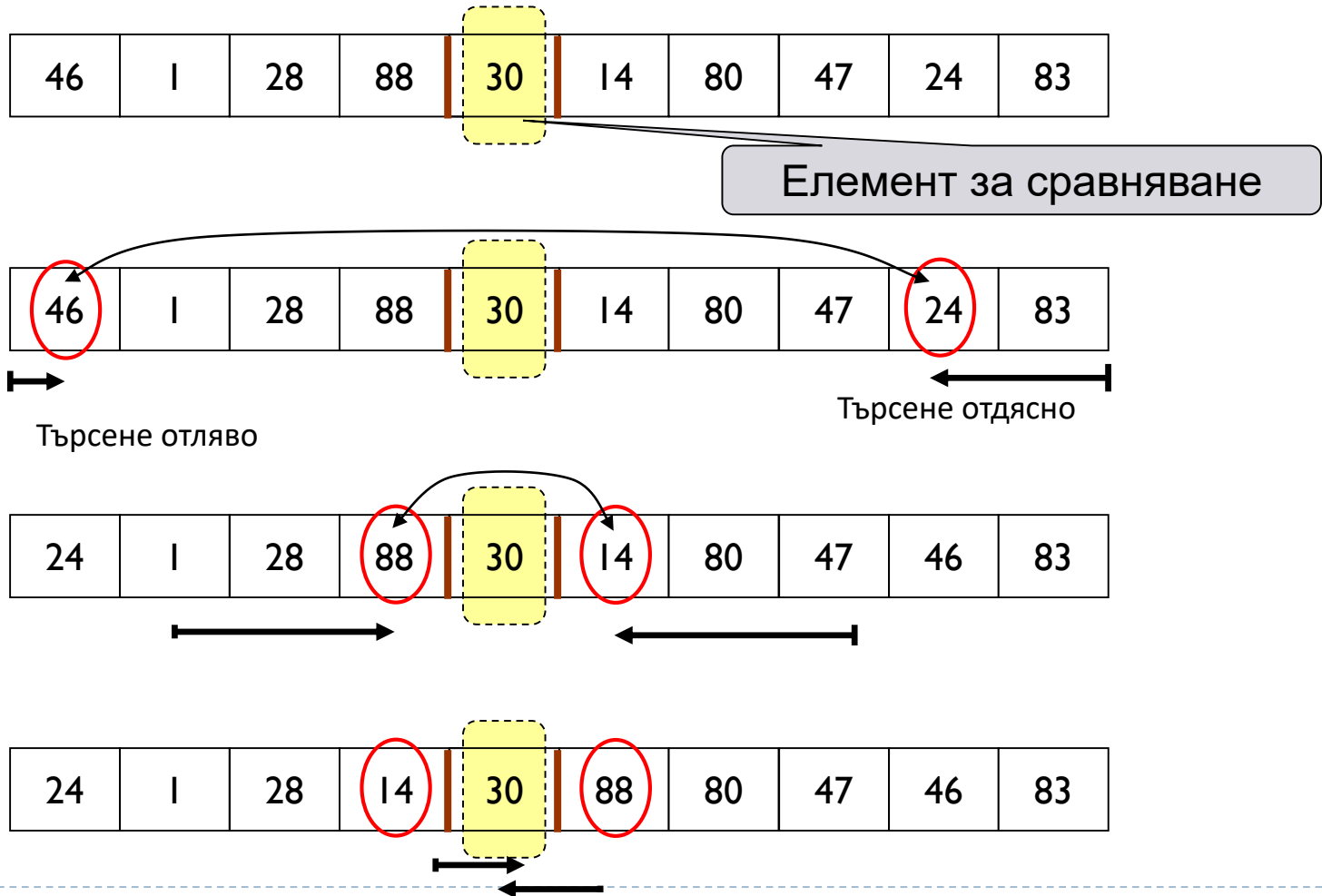
Техника:

- Избираме произволен елемент от масива
(обикновено: средния елемент, наречен **пивот** - елемент за сравняване)
- Претърсваме масива отляво докато: (елемент \geq pivot) намерен
- Претърсваме масива отдясно докато: (елемент \leq pivot) намерен
- Раменяме двата елемента



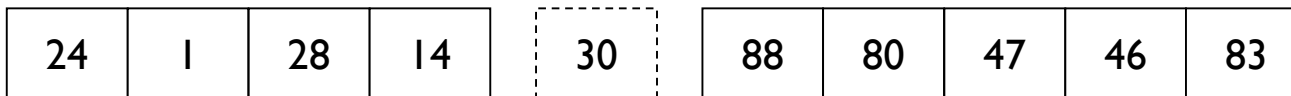
Пример: Quick sort (1)

I. Стъпка на разделяне

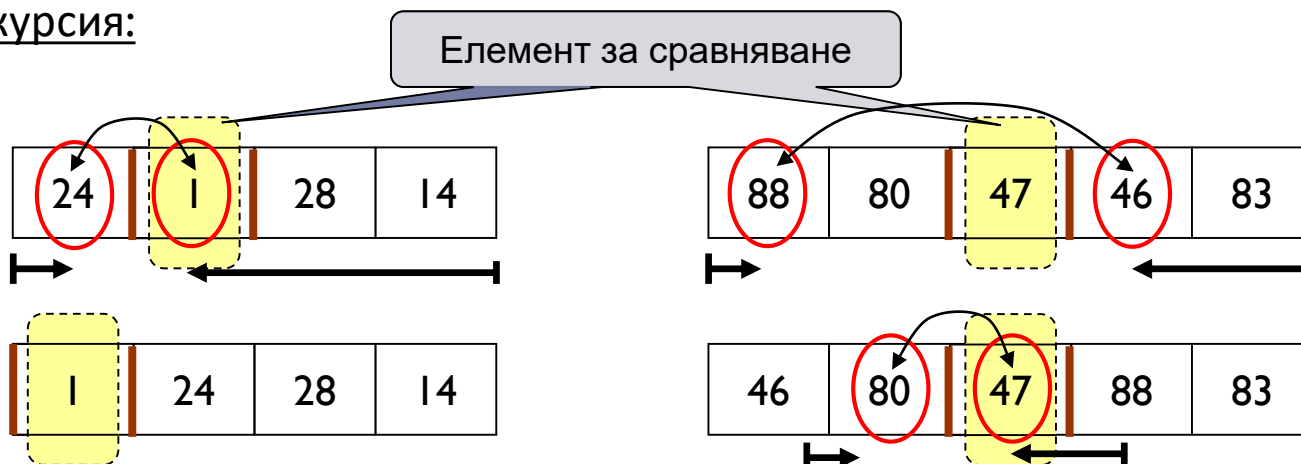


Пример: Quick sort (2)

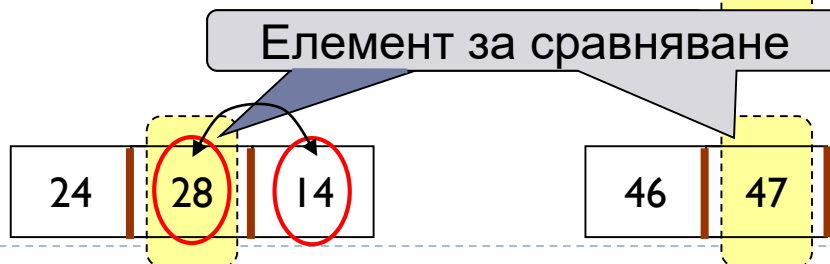
След 1. стъпка на разделяне:



1. Рекурсия:

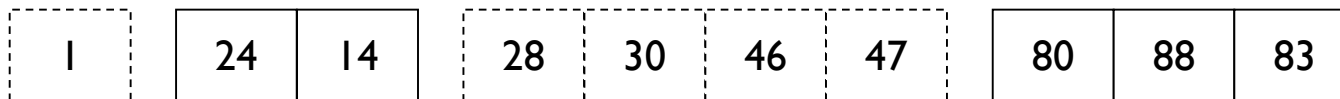


2. Рекурсия:

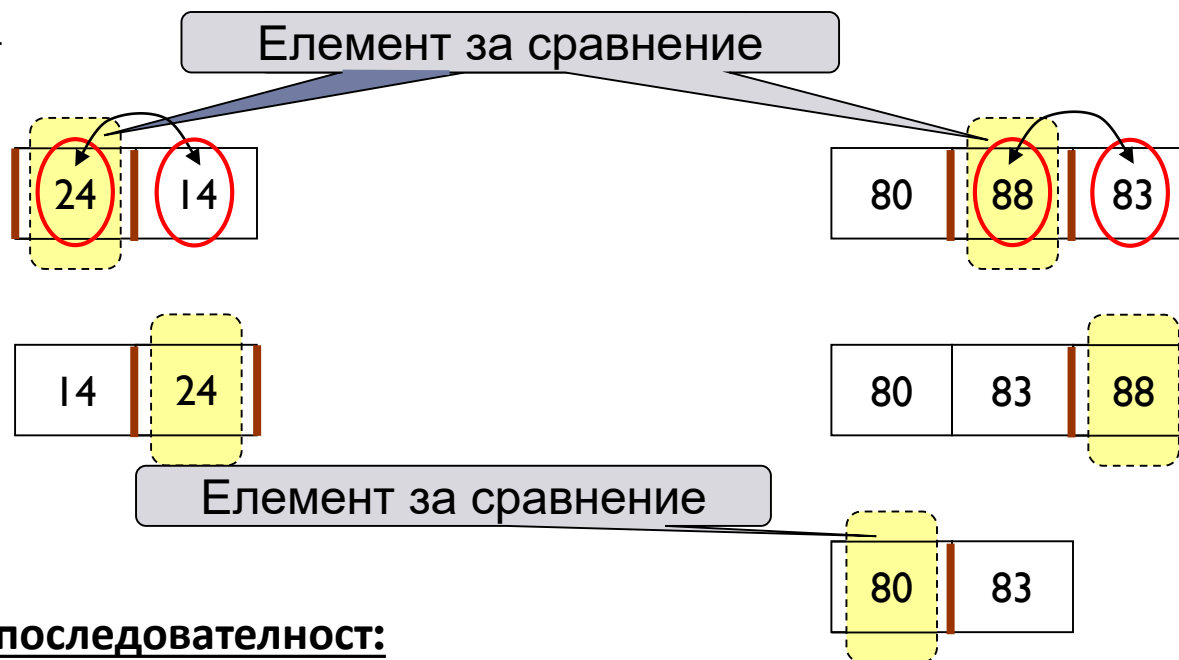


Пример: Quick sort (3)

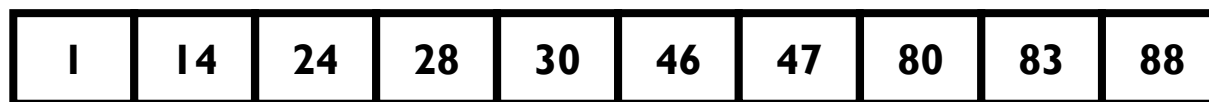
След 5 стъпки на разделяне:



3. Рекурсия:



Сортирана последователност:



→ 8 стъпки на разделяне са необходими от (под) масиви.

Разделяне на две части: Java-програма

```
public static void quicksort (int[] a, int left, int right) {  
    int help ;  
    int i = left;  
    int j = right;  
    // pivot element  
    int x = a[(left+right) / 2];  
    do {  
        while (a[i] < x) i++;  
        while (a[j] > x) j--;  
        if ( i<=j ) {  
            help = a[i];  
            a[i] = a[j];  
            a[j] = help;  
            i++; j--;  
        }  
    } while (i <= j);  
    // сега: елементите в лявата част са по-малки  
    // отколкото елементите в дясната част  
    // -> след това: сортиране по отделно на лявата и дясната част  
}
```

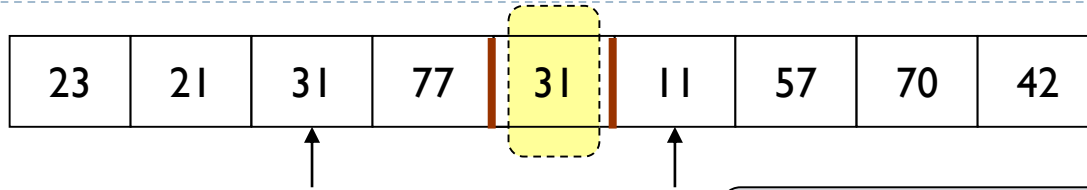
пропуска: леви малки елементи

пропуска: десни по-големи елементи

Размяна, ако елементите стоят от грешната страна

продължава, докато разделянето завършено

Детайли: сравнения



**\leq : еднакво големи елементи
I остават ?**

Алгоритъмът
при определени
условия не
завършва

3 $i < j$: не разменя
при $i = j$?

При $i < j$ условието
за прекъсване
не се получава

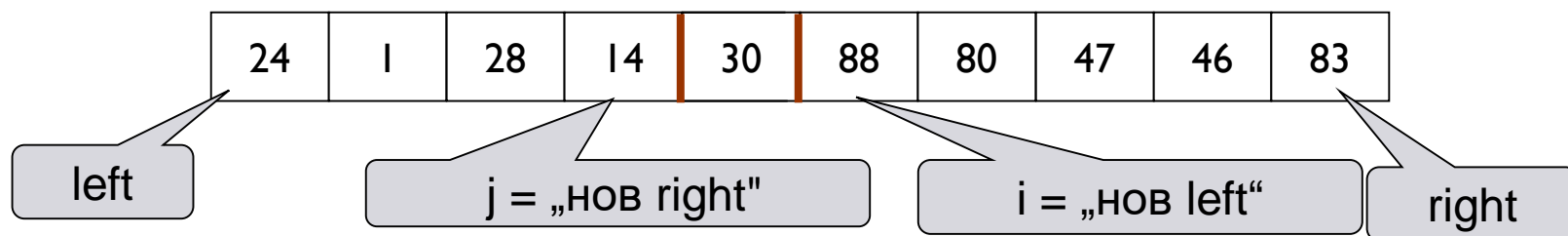
```
do {
    while (a[i] < x) i++;
    while (a[j] > x) j--;
    if ( i<=j ) {
        help = a[i];
        a[i] = a[j];
        a[j] = help;
        i++; j--;
    }
} while (i <= j);
```

② $i < j$: спира при $i = j$?

Средният елемент
рекурсивно
също сортиран
(ефективност)

Отделно сортиране: лява и дясна част

```
public static void quicksort (int[] a, int left, int right) {  
    ...  
  
    // сортиране на лявата и дясната част по отделно,  
    // ако съществува повече от един елемент  
  
    if (left < j)  
        quicksort(a, left, j);  
    if (i < right )  
        quicksort(a, i, right );  
}
```



Quick sort – възможна също итеративно ?

```
public static void quicksort
    (int[] a, int left, int right)  {

    // разделяне на две части
    do {...} while(...);

    if (left < j)
        quicksort(a, left, j)
    if (i < right)
        quicksort(a,i, right);
}
```

Quick sort: 'Problem-Stack' ?

Проблем за съхраняване: индексна област, която ще се сортира

Извикване в Quicksort.java:

```
quicksort(a, 0, n-1);
```

напр. 100

Развитие на стека:

Начало:

0	100
---	-----

1. Циклична стъпка:

0	68
70	100

2. Циклична стъпка:

0	21
23	68
70	100

Необходим 'Problem-stack' ?

Не става ли въпрос
за последователността
на решаваните проблеми?

Комплексност : Quick sort

- **Най-добрият случай :**

- Array винаги на две равни части разделен.

→ Сравнения:

$$O(n) = n \log_2 n$$

→ Операции за размяна:

$$O(n) = n \log_2 n / 6$$

- **Средна комплексност:**

- Само с фактор $2 * \ln 2 = 1.39...$ по-лоша

- **Най-лошият случай :**

$$O(n) = n^2$$

- Винаги отделяме само един елемент

n = 1.000.000 → Сравнения	Selection sort	1.000.000.000.000
	Quick sort	20.000.000

50.000 пъти
повече
сравнения!

Hash–техника: свойства

- ▶ Най-бързият метод за търсене
- ▶ Цел : търсене с един достъп:
 - ▶ т.е. Константна комплексност $O(n) = k$
 - ▶ \rightarrow Евентуално дори с $k = 1 \dots$
- ▶ Времето за изпълнение за сметка на паметта

Hash-техника: основна идея

Hash-сортиран масив :

-	-	3	50	-	40	-	30	-	20	2	10	5
0	1	2	3	4	5	6	7	8	9	10	11	12

масив с дължина 13 :

- Частично неаът
- Изключително несортиран (разпръснато съхраняване)

идея: не търсим елемент,
а изчисляваме позиция в масива

Hash-функция H : асоциативно адресиране

H : ключ \rightarrow адреси

Пример: Hash-функция

-	-	3	50	-	40	-	30	-	20	2	10	5
0	1	2	3	4	5	6	7	8	9	10	11	12

Цел:

$H : \text{Integer} \rightarrow [0 \dots 12]$

Възможност:

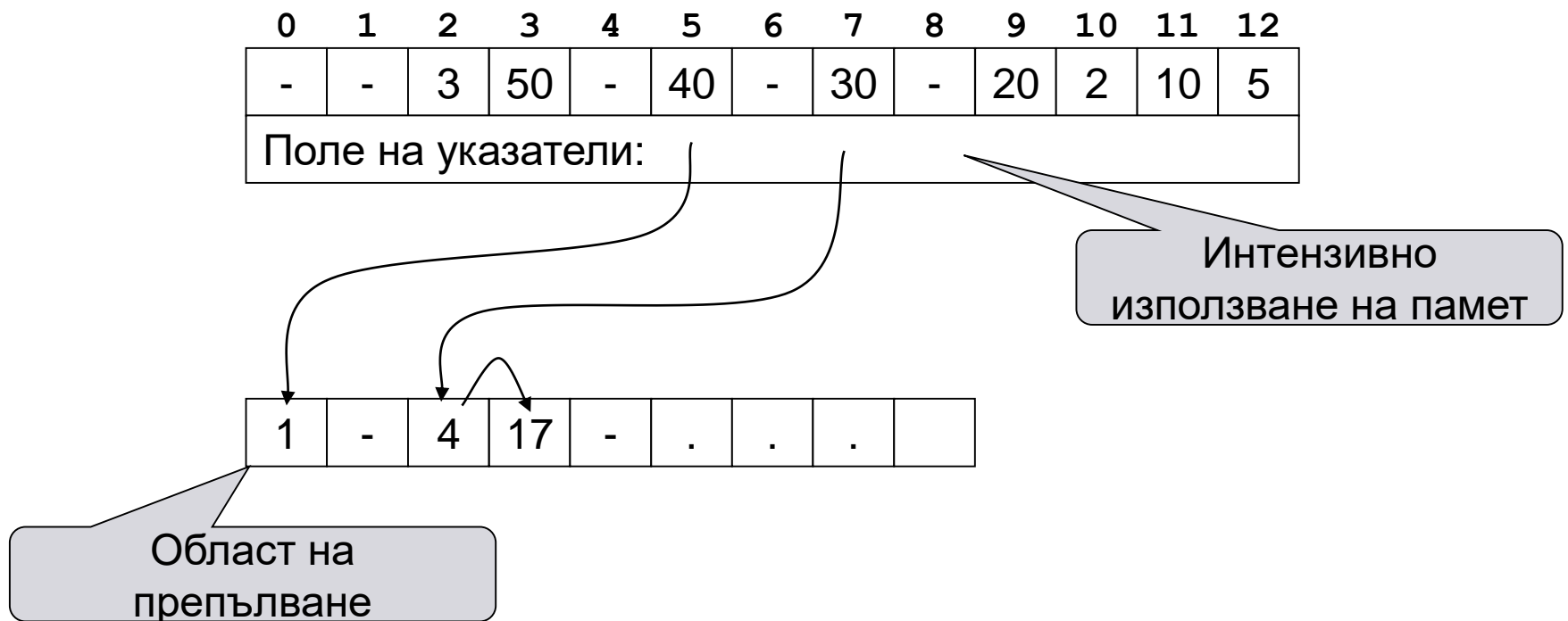
$$H(n) = 5 * n \bmod 13$$

$$H(30) = 150 \% 13 = 7$$

$$H(4) = 20 \% 13 = 7$$

Колизия: $H(n_1) = H(n_2)$

Обработка на колизии



Обработка на колизии: отворено свързване

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	3	50	-	40	-	30	-	-	2	10	5

Записваме нов : 4 , 17

- $H(4) = H(17) = H(30) = 7$

→ Ако мястото е заето :
Записваме в следващото свободно място
(накрая: започваме отново от начало)

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	3	50	-	40	-	30	4	17	2	10	5

Търсене при отворено свързване

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	3	50	-	40	-	30	4	17	2	10	5

Търсене на елемент s :

алгоритъм: как да намерим s ?

$H(s) = i$ (на място i би трябвало да стои s)

1. Случай : $a[i] = s$

→ елемент намерен

2. Случай : $a[i] = -$ (дупка)

→ елемент неналичен

3. Случай : $a[i] \neq s$ (няма дупка)

→ търсим елемент s , започвайки от позиция $i + 1$ на Array докато:

→ стигнем дупка '-' : излизане или

→ елемент намерен

Примери: търсене при отворено свързване

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	3	50	-	40	-	30	4	17	2	10	5

1. Неналичен: $s = 1$, $H(1) = 5$
 - $a[5] < 1$
 - Търсим нататък при $a[6] : '-'$ готови
2. Наличен: $s = 17$, $H(17) = 7$
 - $a[7] < 17$
 - Търсим нататък от позиция $i = 8$:
 $a[9] = 17$, т.е. готови: намерен
3. Неналичен: $s = 51$, $H(51) = 7$
 - $a[7] < 51$
 - Търсим нататък от позиция $i = 8$:
 $i = 8, 9, 10, 11, 12, 0 : '-'$ готови

Реализация на Hash-техника

1. Определяме размера на Hash-таблицата
(Array с твърда индексна област)

0	1	2	3	4	5	6	7	8	9	10	11	12
-	-	3	50	-	40	-	30	4	17	2	10	5

2. Определяме Hash-функцията (+реализираме)

$H : \text{Integer} \rightarrow [0 \dots 12]$

int hash (int key, int tableSize)

3. Реализация на операциите:

write (с колизии),

search

→ в Hash.java: write и search още липсват

→ задача: самостоятелна реализация

Hash-функции: дефиниционни области

- Числови ключови полета
(номера на лични карти, ЕГН)

hash: int	→	адресна област
-----------	---	----------------

- Имена, Идентификатори (Компилятор) ...

hash: String	→	адресна област
--------------	---	----------------

Hash-функция за Strings

```
static int hash (String key, int tableSize) {  
  
    int hashVal = 0;  
  
    for (int i = 0; i < key.length(); i++)  
        hashVal = 37 * hashVal + key.charAt(i);  
  
    hashVal %= tableSize;  
    if (hashVal < 0)  
        hashVal += tableSize;  
  
    return hashVal;  
}
```

String:
API-клас
(не Array)

Тип с
операции

i: 0 1 2 3 4 5
key: p e r s N r

hashVal: 750

`key.length() - 1`

не: `key[4]`

`key.charAt(4)`

Hash-функция за Strings: детайли

напр.: "aa"

1000

```
static int hash (String key, int tableSize) {
```

```
    int hashVal = 0;
```

int + char ?

```
    for (int i = 0; i < key.length(); i++)
```

```
        hashVal = 37 * hashVal + key.charAt(i);
```

```
    hashVal %= tableSize;
```

```
    if (hashVal < 0)
```

```
        hashVal += tableSize;
```

```
    return hashVal;
```

```
}
```

37 * 97 + 'a'

за i = 1

37 * 97 + 97

Коректура на отрицателните Hash-стойности?

автоматично конвертиране
Unicode / ASCII

Приложение на Hash-функции

```
public static void main (String[] args) {  
    String str;  
    int length;  
  
    System.out.print("Enter table length: ");  
    length = Keyboard.readInt();  
  
    while (true) {  
        System.out.print("Enter a string: ");  
        str = Keyboard.readString();  
        System.out.println("String: " + str  
            + " Hash value: "  
            + hash(str, length));  
        if (str.equals("0")) return;  
    }  
}
```

String-сравнение

```
% java Hash  
Enter table length: 1000  
Enter a string: abcdef  
String: abcdef Hash value: 401
```

Ефективност

L : фактор на натоварване на една Hash-таблица
(част на запълнените места на таблицата)

L = 0.5

- Въвеждане: средно 2.5 разгледани места
- Търсене: средно 1.5 разгледани места
(успешно търсене)

L = 0.9

- Въвеждане: средно 50 разгледани места
- Търсене: средно 5.5 разгледани места
(успешно търсене)

→ Клас на комплексност: константен

Java : API-клас 'Hashtable'

- **Самостоятелна реализация:**

- Метод 'hash' за изчисляване на Hash-функция
- equals() – определя еквивалентност на обекти

- **Параметри:**

- capacity: размер на таблицата
- loadFactor: фактор на натоварване L
 - Ако таблицата е напълнена до процент L: автоматично разширяване на таблицата
 - При по-пълни таблици: много колизии, т.е. дълги времена за търсене

Website: API-Class Hashtable

java.util

Class Hashtable<K,V>

[java.lang.Object](#)

└ [java.util.Dictionary<K,V>](#)

└ [java.util.Hashtable<K,V>](#)

All Implemented Interfaces:

[Serializable](#), [Cloneable](#), [Map<K,V>](#)

Direct Known Subclasses:

[Properties](#), [UIDefaults](#)

```
public class Hashtable<K,V>
    extends Dictionary<K,V>
    implements Map<K,V>, Cloneable, Serializable
```

This class implements a hashtable, which maps keys to values. Any non-null object can be used as a key or as a value.

To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

An instance of Hashtable has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of *buckets* in the hash table, and the *initial capacity* is simply the capacity at the time the hash table is created. Note that the hash table is *open*: in the case of a "hash collision", a single bucket stores multiple entries, which must be searched sequentially. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. The initial capacity and load factor parameters are merely hints to the implementation. The exact details as to when and whether the rehash method is invoked are implementation-dependent.

If many entries are to be made into a Hashtable, creating it with a sufficiently large capacity may allow the entries to be inserted more efficiently than letting it perform automatic rehashing as needed to grow the table.

This example creates a hashtable of numbers. It uses the names of the numbers as keys:

```
Hashtable numbers = new Hashtable();
numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));
```

To retrieve a number, use the following code:

```
Integer n = (Integer)numbers.get("two");
if (n != null) {
    System.out.println("two = " + n);
}
```



Merge sort: сортиране на външни файлове посредством смесване

Данните не могат да бъдат цялостно заредени в оперативната памет

- последователности (файлове: външна памет)
- Винаги само един елемент достъпен

399	423	505	201	1001	8023	232	...
-----	-----	-----	-----	------	------	-----	-----

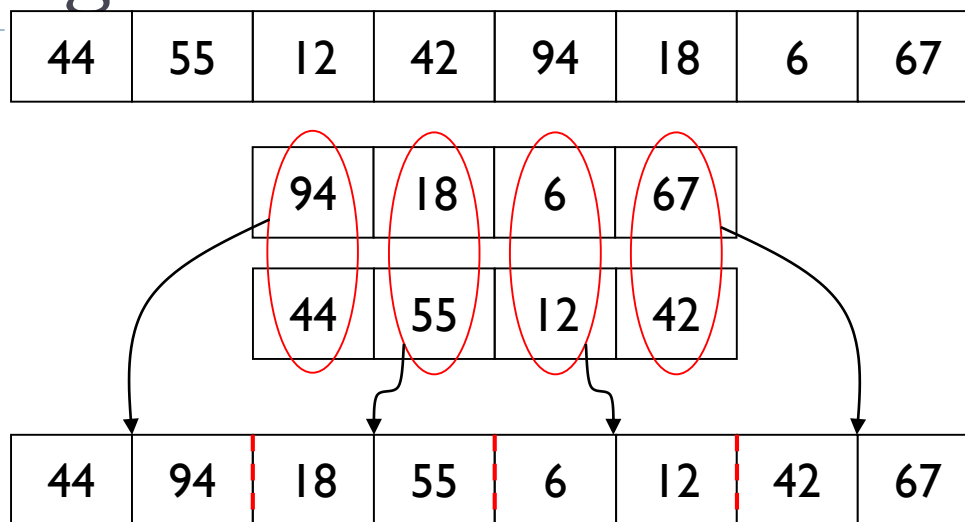


Алгоритъм:

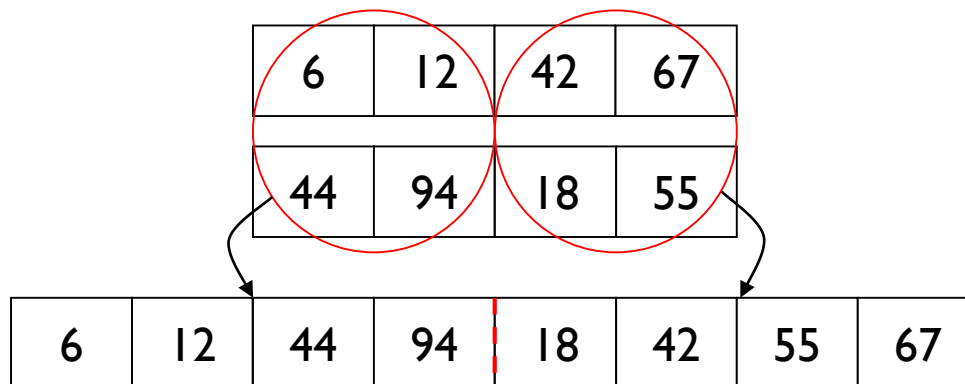
1. Декомпозираме последователността на 2 подпоследователности **a** и **b**
2. Смесваме **a** и **b** : последователност **c** от наредени двойки
3. Декомпозираме последователност **c** на 2 последователности **a1** и **b1**
4. Смесваме **a1** и **b1**: последователност **c1** от наредени четворки
- ...

Пример: Merge sort

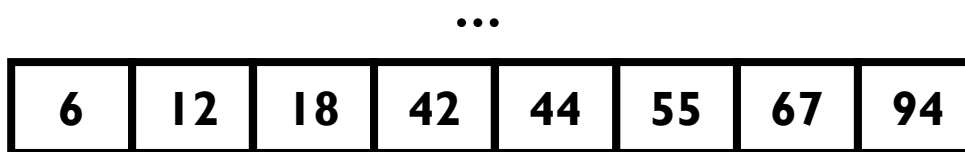
1. Стъпка:



2. Стъпка:



3. Стъпка:



Смесване: сортирани последователности*)

```
public static int[] merge (int[] a, int[] b) {
    int i=0, j=0, k=0;
    int[] c = new int[a.length + b.length];
    // смесване, докато масивът стане празен
    while ((i < a.length) && (j < b.length)) {
        if (a[i] < b[j])
            c[k++] = a[i++];
        else
            c[k++] = b[j++];
    }
    // останалите не празни последователности:
    if (i == a.length)
        while (j < b.length)    c[k++] = b[j++];
    else
        while (i < a.length)    c[k++] = a[i++];
    return c;
}
```

Тип на резултата: Array

Странични ефекти !

- Разглеждаме последователности като масиви (вътрешно)
→ Файлове (външно)
- но: обработваме масивите като файлове (последователно)

Благодаря за вниманието!

Край лекция 10. “Търсене и сортиране с масиви”