

ТЪРСЕНЕ И СОРТИРАНЕ

ЛЕКЦИОНЕН КУРС “ПРОГРАМИРАНЕ НА JAVA”



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
- Компилятори, 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

Ключ (поле)

Personal_number: 1003

Name: 'Иванов'

First_name: ...

Address:

- **Street:**

- **Postal_code:**

- **City:**

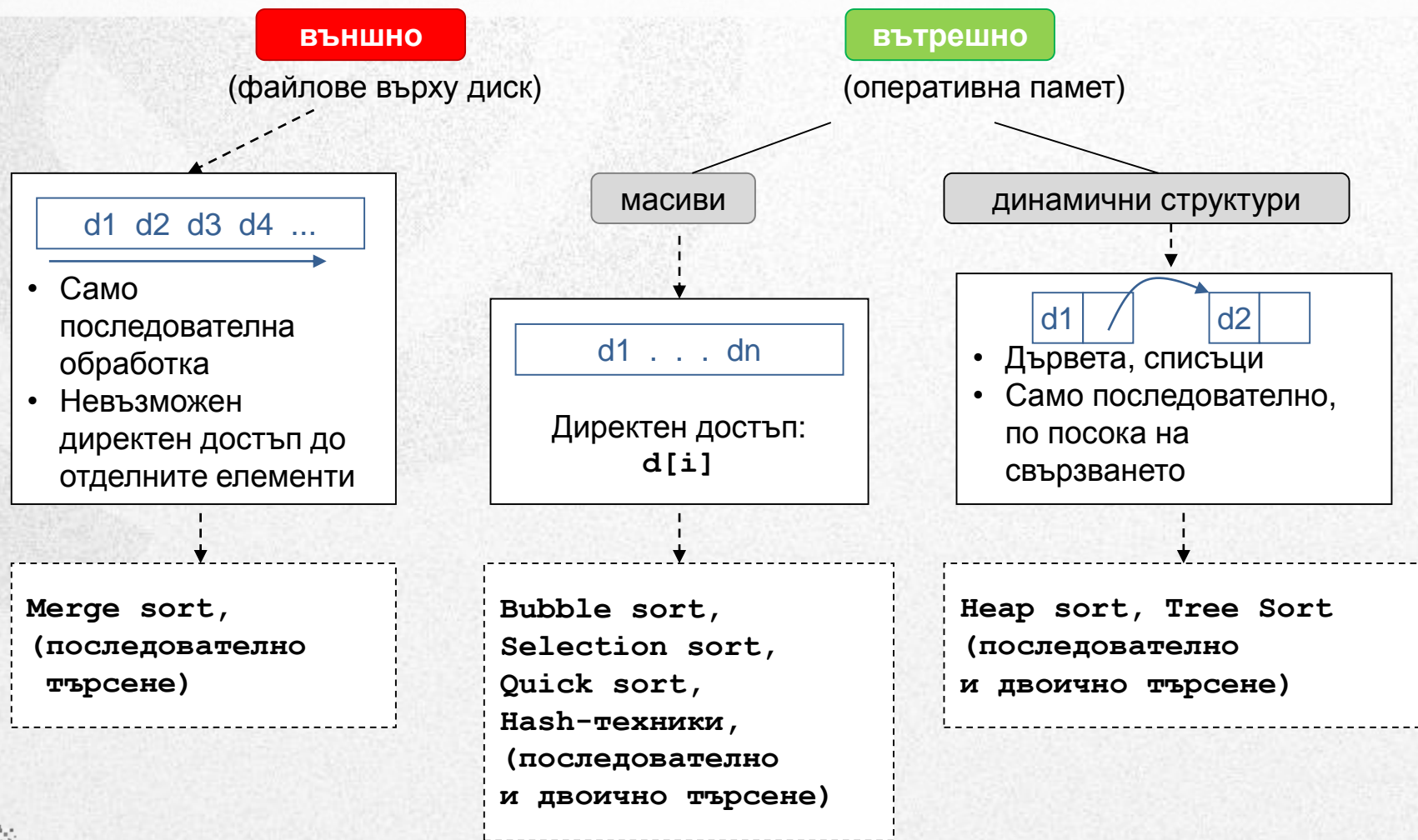
- **Home_nr:**

Job_function) :

more: Business, sellary, ... ,
Faculty, Institute, Tel.Nr, ...

полета

КЪДЕ/КАК ЩЕ СЕ СЪХРАНЯВАТ ЗАПИСИТЕ



КЛАСОВЕ КОМПЛЕКСНОСТ НА АЛГОРИМИТЕ

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: за милиони записи . . .

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?

1

Разлика между двете версии?

Възможна run-time грешка!

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

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

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 ПОДЕЛЯНЕТО

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 полето (напр. $m=5$)
 - 1. случай: $x = a[m] \rightarrow$ край: елементът е намерен
 - 2. случай: $x > a[m] \rightarrow$ търсене в дясното подполе
 - 3. случай: $x < a[m] \rightarrow$ търсене в лявото подполе
- Заключение: подполе празно \rightarrow край: елементът не се появява

ДЕМО: ДВОИЧНО ТЪРСЕНЕ

- Дадени: един ключ **key** и сортиран масив **a[]**
- Намери: индекс **i** такъв че **a[i] = key**, или информирай, че не съществува такъв индекс
- Инвариант: Алгоритъмът поддържа **a[lo] ≤ key ≤ 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

- Дадени: един ключ **key** и сортиран масив **a[]**
- Намери: индекс **i** такъв че **a[i] = key**, или информирай, че не съществува такъв индекс
- Инвариант: Алгоритъмът поддържа **a[lo] ≤ key ≤ 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

- Дадени: един ключ key и сортиран масив $a[]$
- Намери: индекс i такъв че $a[i] = key$, или информирай, че не съществува такъв индекс
- Инвариант: Алгоритъмът поддържа $a[lo] \leq key \leq a[hi]$
- 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

- Дадени: един ключ key и сортиран масив $a[]$
- Намери: индекс i такъв че $a[i] = key$, или информирай, че не съществува такъв индекс
- Инвариант: Алгоритъмът поддържа $a[lo] \leq key \leq a[hi]$
- 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

- Дадени: един ключ key и сортиран масив $a[]$
- Намери: индекс i такъв че $a[i] = key$, или информирай, че не съществува такъв индекс
- Инвариант: Алгоритъмът поддържа $a[lo] \leq key \leq a[hi]$
- 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

- Дадени: един ключ key и сортиран масив $a[]$
- Намери: индекс i такъв че $a[i] = key$, или информирай, че не съществува такъв индекс
- Инвариант: Алгоритъмът поддържа $a[lo] \leq key \leq a[hi]$
- 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

- Дадени: един ключ key и сортиран масив $a[]$
- Намери: индекс i такъв че $a[i] = key$, или информирай, че не съществува такъв индекс
- Инвариант: Алгоритъмът поддържа $a[lo] \leq key \leq a[hi]$
- 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

- Дадени: един ключ key и сортиран масив $a[]$
- Намери: индекс i такъв че $a[i] = key$, или информирай, че не съществува такъв индекс
- Инвариант: Алгоритъмът поддържа $a[lo] \leq key \leq a[hi]$
- 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
mid

ДЕМО: ДВОИЧНО ТЪРСЕНЕ

- Дадени: един ключ key и сортиран масив $a[]$
- Намери: индекс i такъв че $a[i] = key$, или информирай, че не съществува такъв индекс
- Инвариант: Алгоритъмът поддържа $a[lo] \leq key \leq a[hi]$
- 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
mid

КОМПЛЕКСНОСТ: ДВОИЧНО ТЪРСЕНЕ

- Брой на разделянията:
 - Най-лошият случай: делим полето, докато само един елемент 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

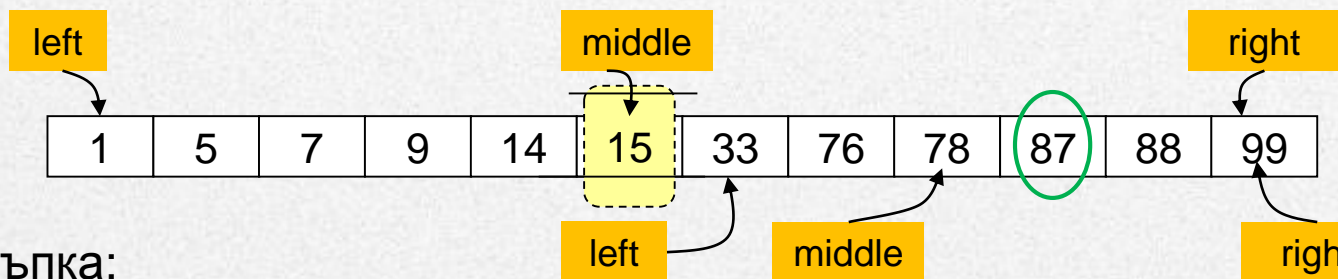
ДВОИЧНО ТЪРСЕНЕ: JAVA-ПРОГРАМА

```
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:

търси **87**:



Методи за сортиране

ВЪНШНИ

Merge sort

вътрешни

масиви

дървета

списъци

бавни

Bubble sort

Selection sort

бързи

Quick sort

Hash-Technique

поскоро: метод за търсене


Tree sort

Heap sort

СОРТИРАНЕ ЧРЕЗ ИЗБОР

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.



250	88	201	7	330	1021	56
-----	----	-----	---	-----	------	----

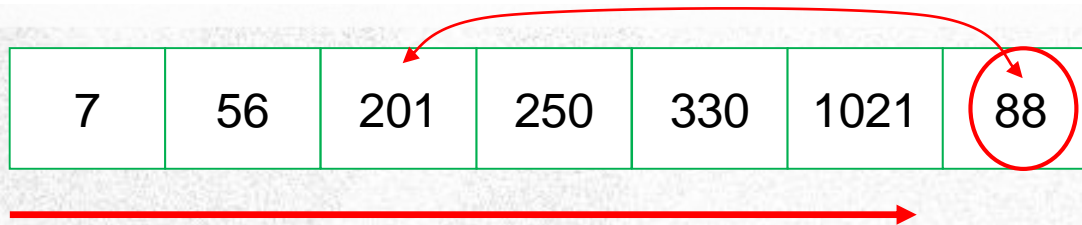


7	88	201	250	330	1021	56
---	----	-----	-----	-----	------	----



7	56	201	250	330	1021	88
---	----	-----	-----	-----	------	----

КОМПЛЕКСНОСТ: SELECTION SORT



- **Размени: $O(n)$**

(точно: $n - 1$)

- **Сравнения: $O(n^2)$**

(точно:

$$\begin{aligned} & (n-1) + (n-2) + \dots + 1 \\ &= n * (n - 1) / 2 \\ &= (n^2 - n) / 2 \end{aligned}$$

КЛАСЪТ КОМПЛЕКСТНОСТ СЕ ОПРЕДЕЛЯ ЧРЕЗ НАЙ-ВИСОКАТА РАЗМЕРНОСТ

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

въпреки точната стойност за броя на сравнения:

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

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

Клас на комплекстност: $O(n^2)$

- 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 разглежда)

КЛАСОВЕ КОМПЛЕКСНОСТ: ИЗБРАНИ СТОЙНОСТИ

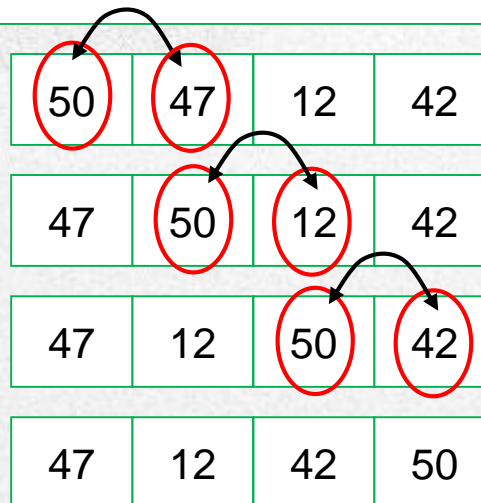
	2	8	10	100	1000
костантен	1	1	1	1	1
логаритмичен (power1)	1	3	4	7	10
линеен (power)	2	8	10	100	1000
квадратичен	4	64	100	10.000	1.000.000
експоненциален (Hanoi)	16	256	1024	~10 Mrd.	~10 ¹⁰⁰

$$n^2 - n$$

BUBBLE SORT: СМЯНА СЪС СЪСЕДА

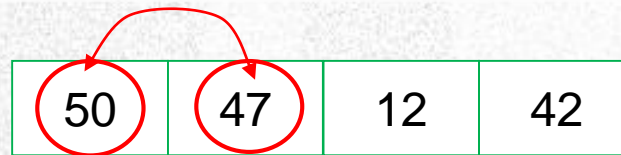
Алгоритъм:

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.



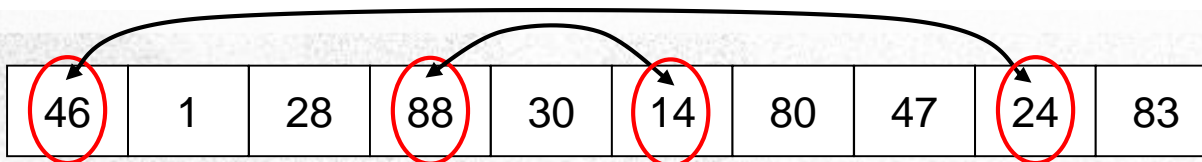
1. Стъпка

КОМПЛЕКСНОСТЬ: 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):

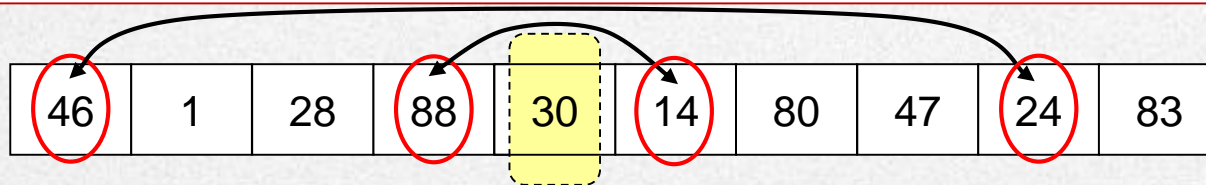
- Разделяме целия масив посредством размяна на елементи в 2 части:
всички елементи от лявата част са \leq от всички елементи от дясната част
- Сортираме двете части независимо една от друга (рекурсивно)

24	1	28	14	30	88	80	47	46	83
----	---	----	----	----	----	----	----	----	----

QUICK SORT: РАЗДЕЛЯНЕ НА ДВЕ ЧАСТИ

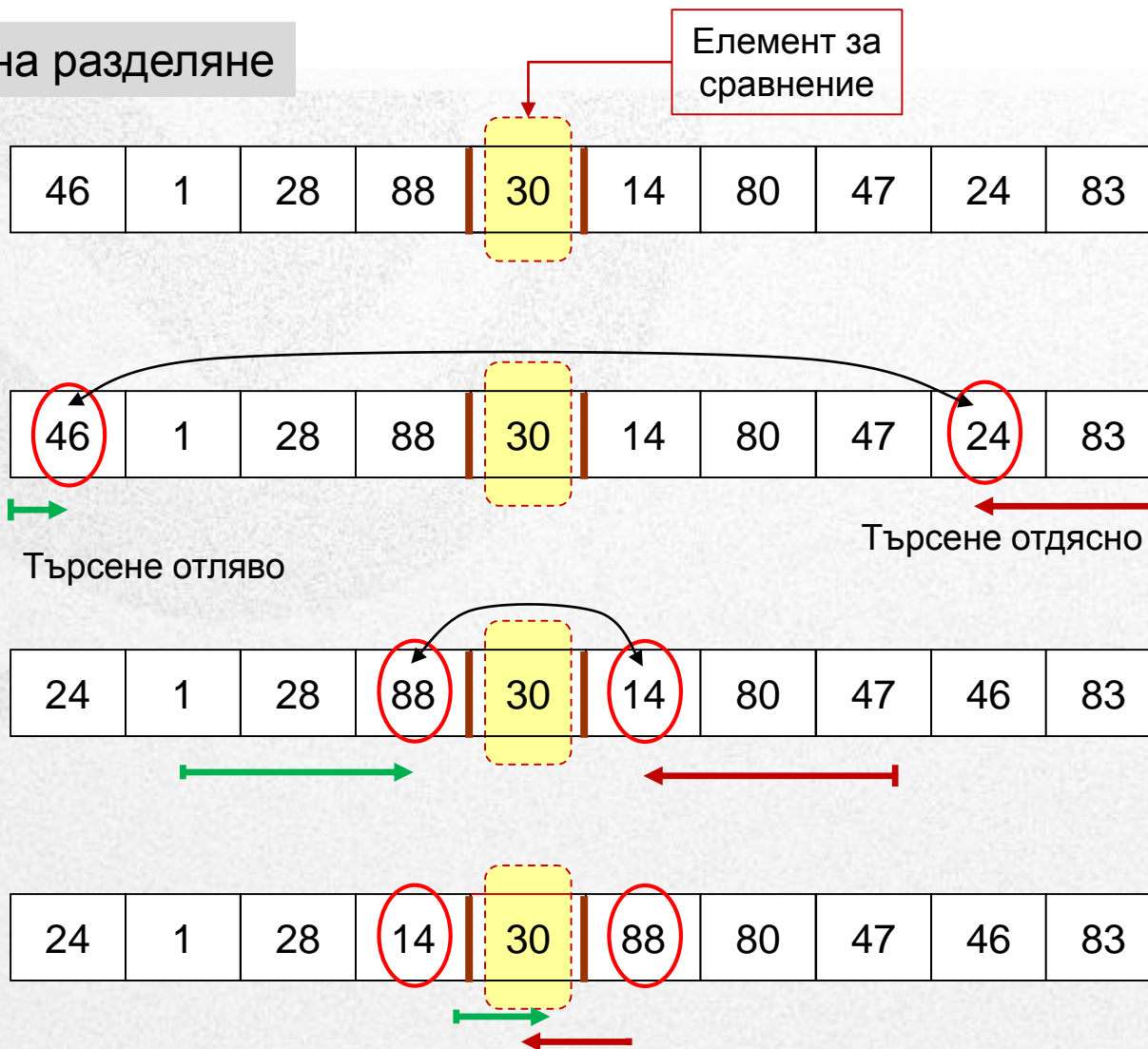
Техника:

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



ПРИМЕР: QUICK SORT

1. Стъпка на разделяне

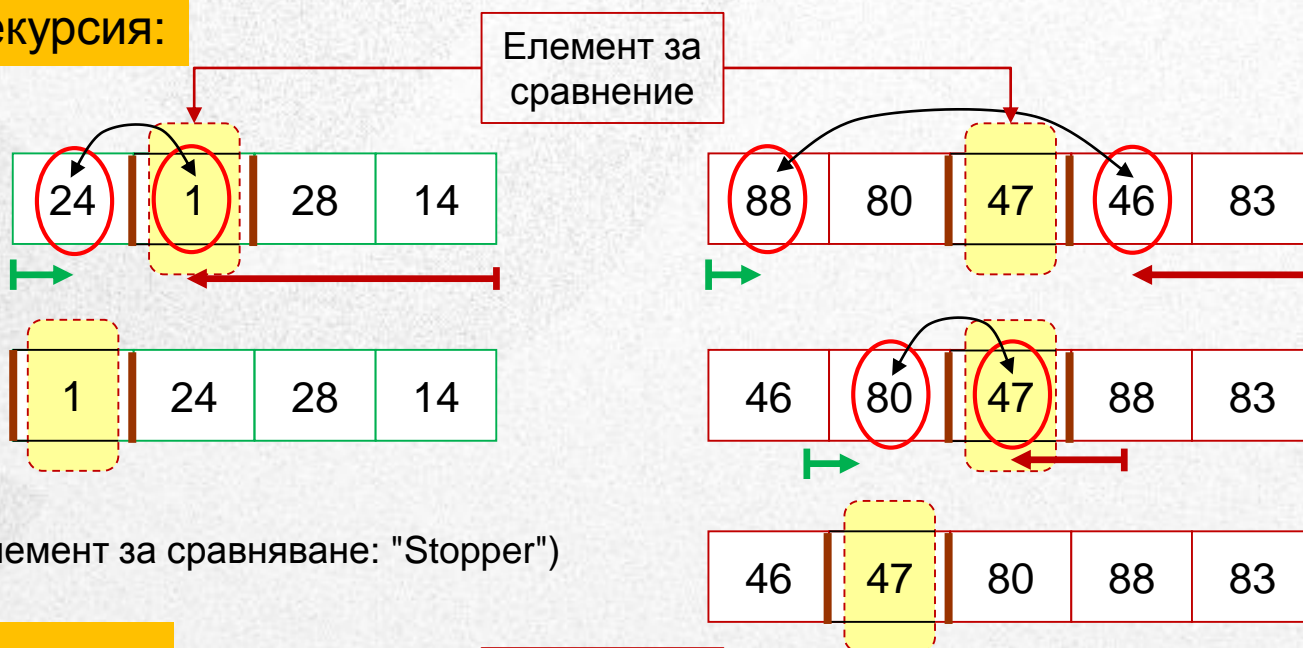


ПРИМЕР: QUICK SORT

След 1. стъпка на разделяне:



1. Рекурсия:



2. Рекурсия:

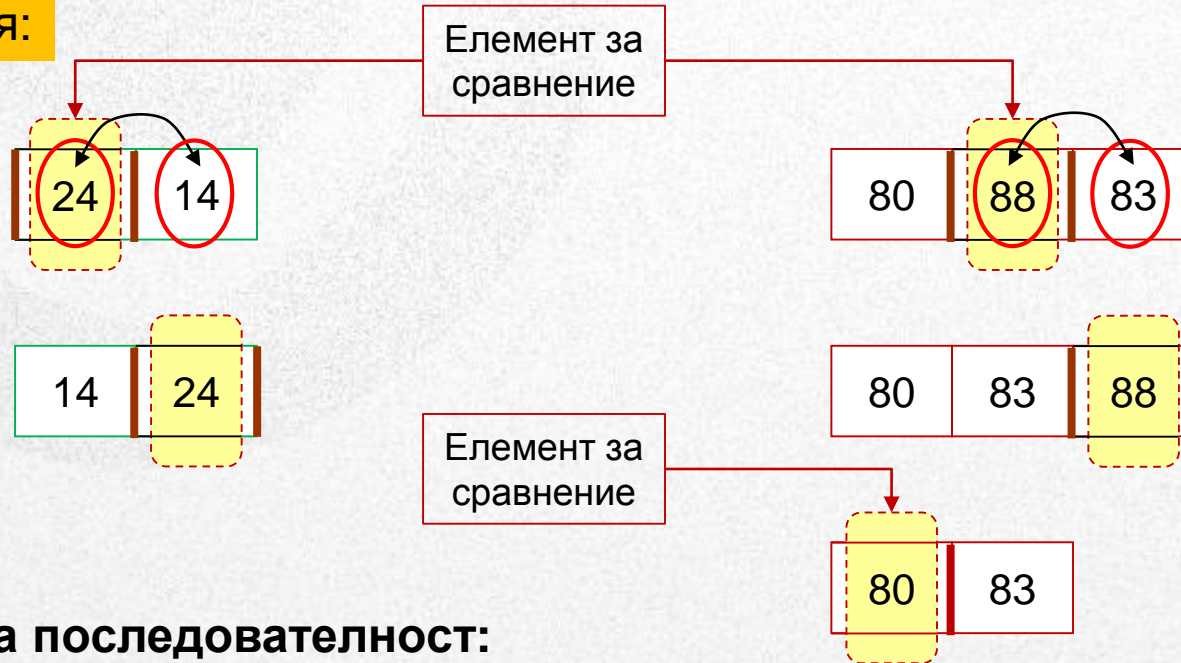


ПРИМЕР: QUICK SORT

След 5 стъпки на разделяне:



3. Рекурсия:



Сортирана последователност:



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

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++; //пропуска: леви по-малки елементи
        while (a[j] > x) j--; //пропуска: десни по-големи елементи
        if ( i<=j ) {
            help = a[i]; // размяна: ако елементите стоят от
            a[i] = a[j]; // грешната страна
            a[j] = help;
            i++; j--;
        }
    } while (i <= j); // продължава: докато разделянето завършено
    // now: elements in the left part smaller
    // as elements in the right part
    // -> after: sort left and right part separately
}
```

ДЕТАЙЛИ: СРАВНЕНИЯ

Diagram illustrating a B-tree node structure. The node contains 9 slots. The 5th slot contains the value 31, which is highlighted with a yellow background and a red dashed border. Arrows point to the 3rd slot (31) and the 6th slot (11).

1 - \leq : еднакво големи елементи остават ?

```
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 $i < j$: не разменя при $i = j$?

При определени условия
алгоритъмът не завършва

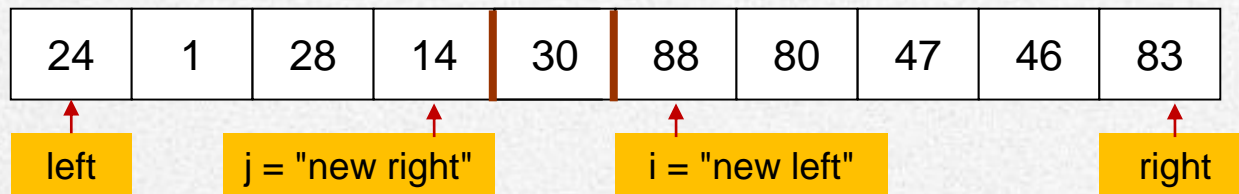
Средният елемент
рекурсивно също сортиран
(ефективност)

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

3 $i < j$: спира при $i = j$?

ОТДЕЛНО СОРТИРАНЕ: ЛЯВА И ДЯСНА ЧАСТ

```
public static void quicksort (int[] a, int left, int right) {  
    ...  
  
    // sort left and right part separately,  
    // if more as an element exists  
  
    if (left < j)  
        quicksort(a, left, j);  
    if (i < right )  
        quicksort(a, i, right );  
}
```



QUICK SORT

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

    //divide into two parts
    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:

Напр. 100

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

Развитие на стека:

Начало:

0	100
---	-----

Необходим 'Problem-stack' ?

1. Циклична стъпка:

0	68
70	100

2. Циклична стъпка:

0	21
23	68
70	100

Не става ли въпрос
за последователността
на решаваните проблеми ?

КОМПЛЕКСНОСТ : QUICK SORT

- **Най-добрият случай :**

- Array винаги разделен на две равни части

→ 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 $O(n) = k$
 - → 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 незает
- Изключително несортиран (разпръснато съхраняване)

Идея: не търсим елемент, а изчисляваме позиция в масива

Hash-функция **H** : асоциативно адресиране

H : **ключ** → **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

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

Интезивно на памет

1	-	4	17	-	.	.	.	
---	---	---	----	---	---	---	---	--

Област на препълване

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

$$\bullet H(4) = H(17) = H(30) = 7$$

→ 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** : $a[i] = s \rightarrow$ елемент намерен
2. **Случай** : $a[i] = -$ (дупка) \rightarrow елемент неналичен
3. **Случай** : $a[i] < > s$ (няма дупка) \rightarrow търсим елемент s , започвайки от позиция $i + 1$ на Array докато:
 - \rightarrow стигнем дупка '-' : излизане или
 - \rightarrow 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:** $s = 51, H(51) = 7$
 - $a[7] < > 17$
 - 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

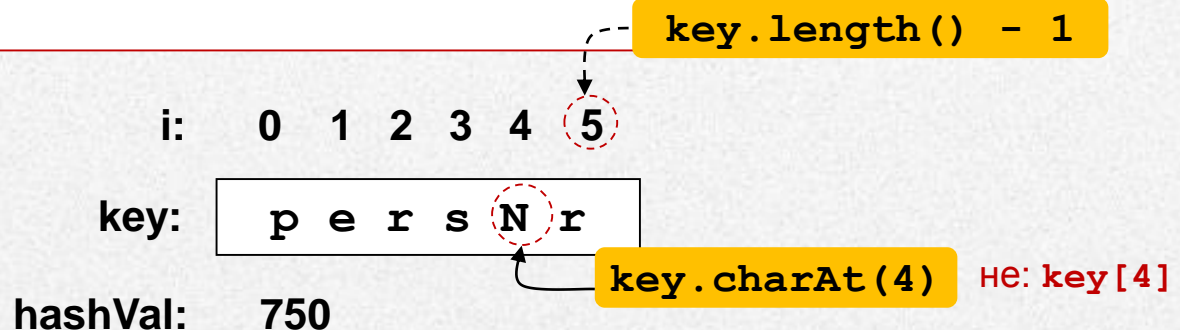
Имена, Идентификатори (Компилятор) ...

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



HASH-ФУНКЦИЯ ЗА STRINGS: ДЕТАЙЛИ

Пример:

"aa"

1000

```
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;  
}
```

int + char?

$37 * 97 + 'a'$

за $i = 1$

$37 * 97 + 97$

автоматично конвертиране
Unicode / ASCII

Коректура на отрицателните Hash-стойности?

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



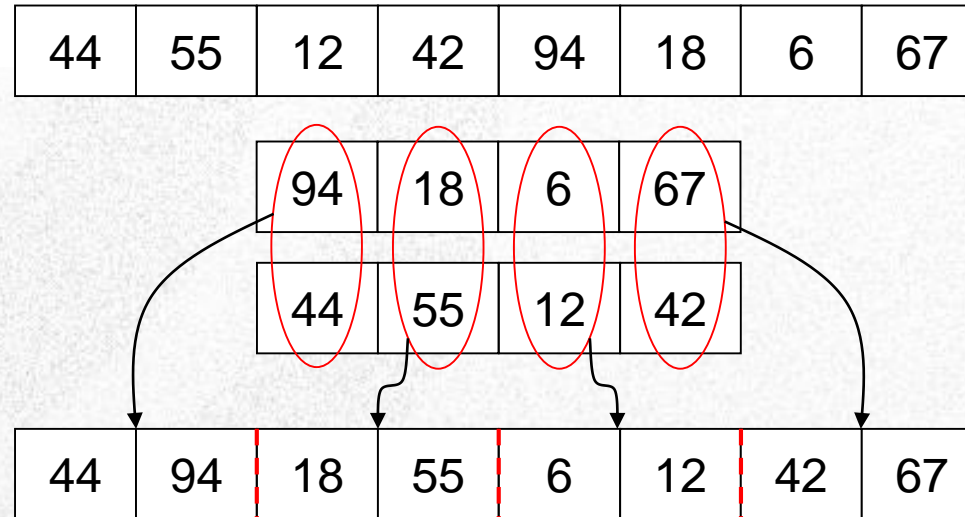
Алгоритъм:

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**

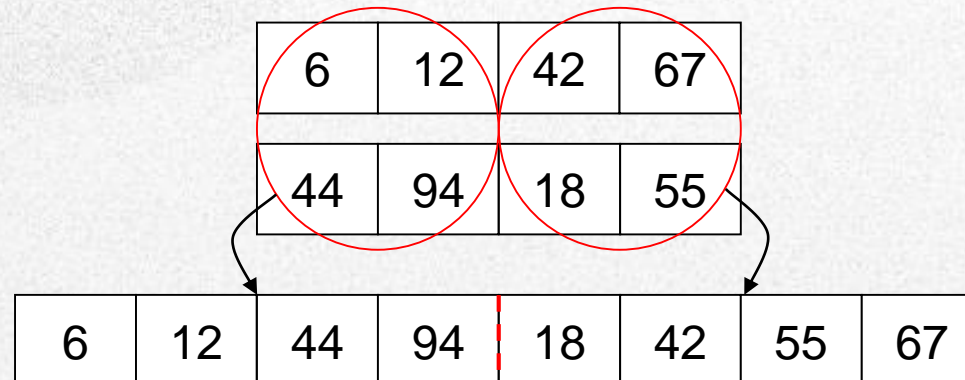
...

ПРИМЕР: MERGE SORT

1. Стъпка:



2. Стъпка:



3. Стъпка:



СМЕСВАНЕ: СОРТИРАНИ ПОСЛЕДОВАТЕЛНОСТИ*)

Тип на резултата: array

```
public static int[] merge (int[] a, int[] b)    {
    int i=0, j=0, k=0;
    int[] c = new int[a.length + b.length];
    // merge, until an Array empty
    while ((i < a.length) && (j < b.length))    {
        if (a[i] < b[j]) ←
            c[k++] = a[i++];
        else
            c[k++] = b[j++];
    }
    // Rest of not empty sequence:
    if (i == a.length)
        while (j < b.length)    c[k++] = b[j++];
    else
        while (i < a.length)    c[k++] = a[i++];
    return c;
}
```

Странични ефекти!

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

*)

БЛАГОДАРЯ ЗА ВНИМАНИЕТО!

КРАЙ “ТЪРСЕНЕ И СОРТИРАНЕ”

