Data Structures and Algorithms – W05

Effective sorting algorithms

Contents

Effective sorting – O(n log n) and better

- Mergesort
  - Recursive version
  - Iterative version
- Quicksort
  - Worst case - O(n^2)
  - Average - O(n log n)
- Heapsort
- Other sorting
  - Countsort - O(n)
  - Radixsort - O(n)
  - Bucketsort - O(n)
- Sorting in the standard Java library
  - with `RandomAccess`
  - without `RandomAccess`

Mergesort

Idea:

- Merging two sorted sequences into one sorted can be done in time O(*n*)
- One-element arrays are sorted
- Using the "divide and conquer" technique, we divide the array into two subarrays of equal (or almost equal) size, sort each (using the same sorting method recursively), merging the results into a sorted sequence.

---

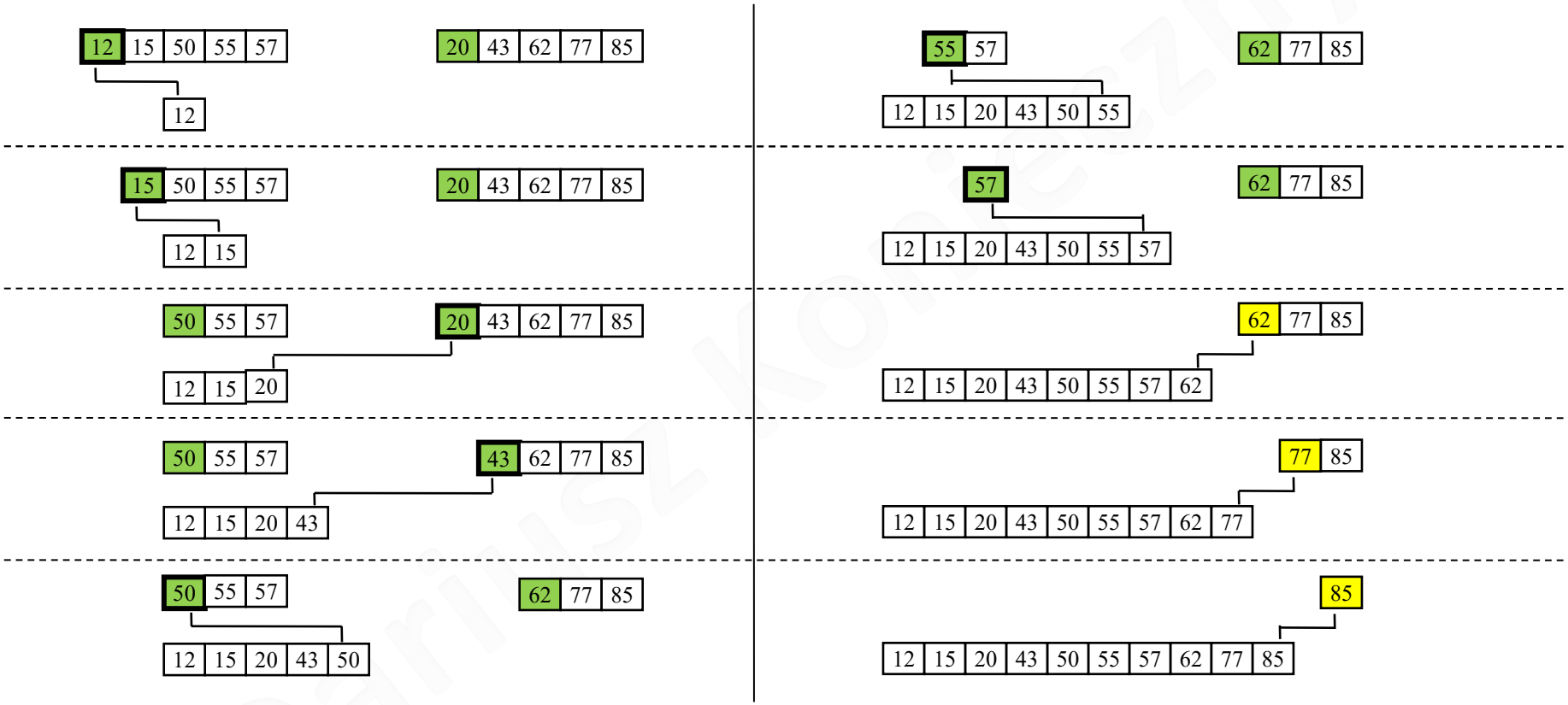
12 The item being compared, **selected** for the output

62 The item being compared, not selected for the output

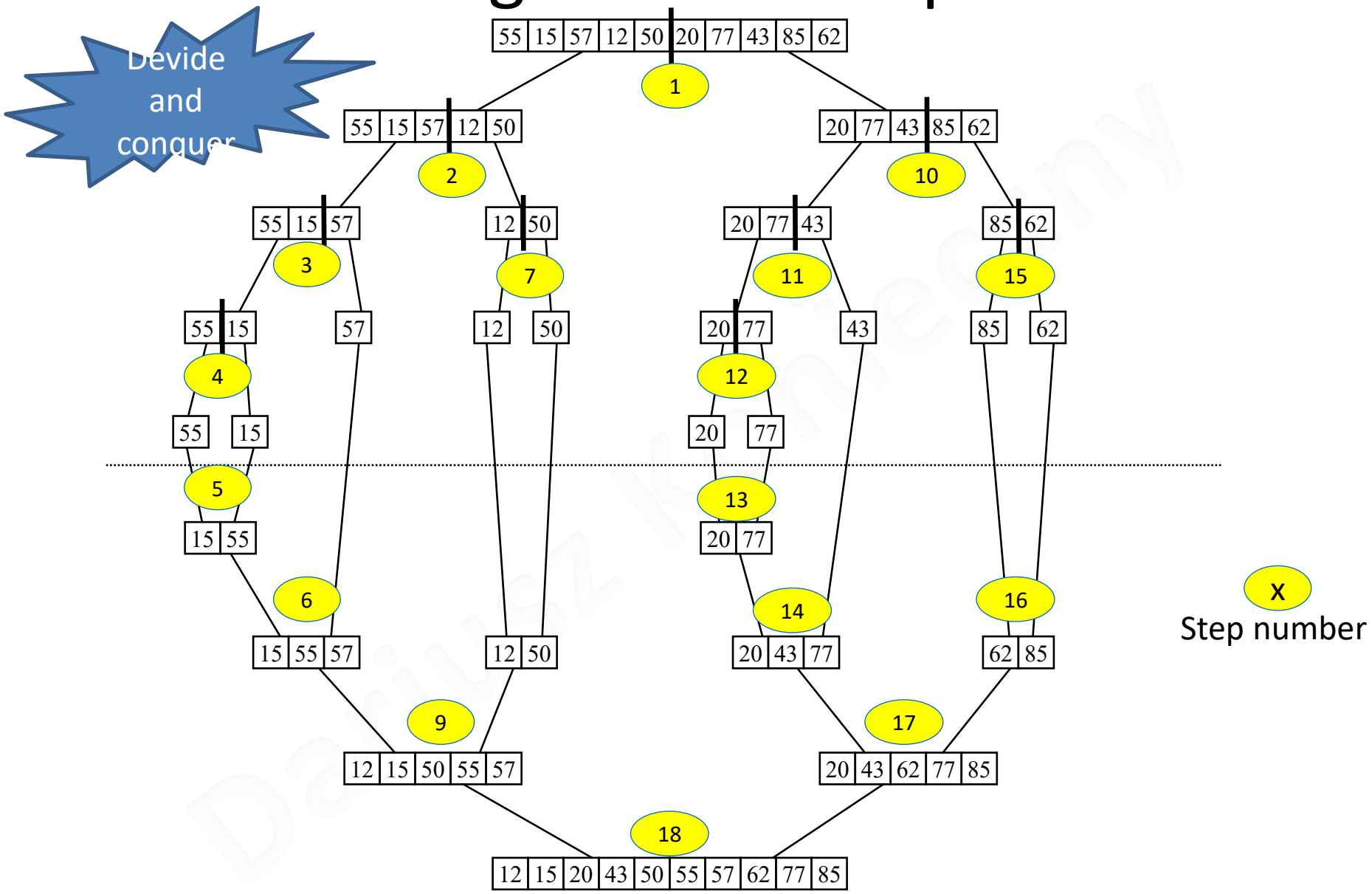
62 The item is rewritten to the output

Merging in time O(*n*)

- In a greedy way to the result, another element is added from the two input lists, always choosing a smaller value (when they are equal - left).



Mergesort - example



Mergesort – code 1/2

```
// the result is a new list
public IList<T> sort(IList<T> list)
{
  return mergesort(list, 0, list.size() - 1);
}

@SuppressWarnings("unchecked")
private IList<T> mergesort(IList<T> list, int startIndex, int endIndex) {
  if (startIndex == endIndex) {
    IList<T> result = (IList<T>)(new ArrayList<Object>());
    result.add(list.get(startIndex));
    return result;}
  int splitIndex = startIndex + (endIndex - startIndex) / 2;
  return merge(mergesort(list, startIndex, splitIndex),
               mergesort(list, splitIndex + 1, endIndex));
}
```

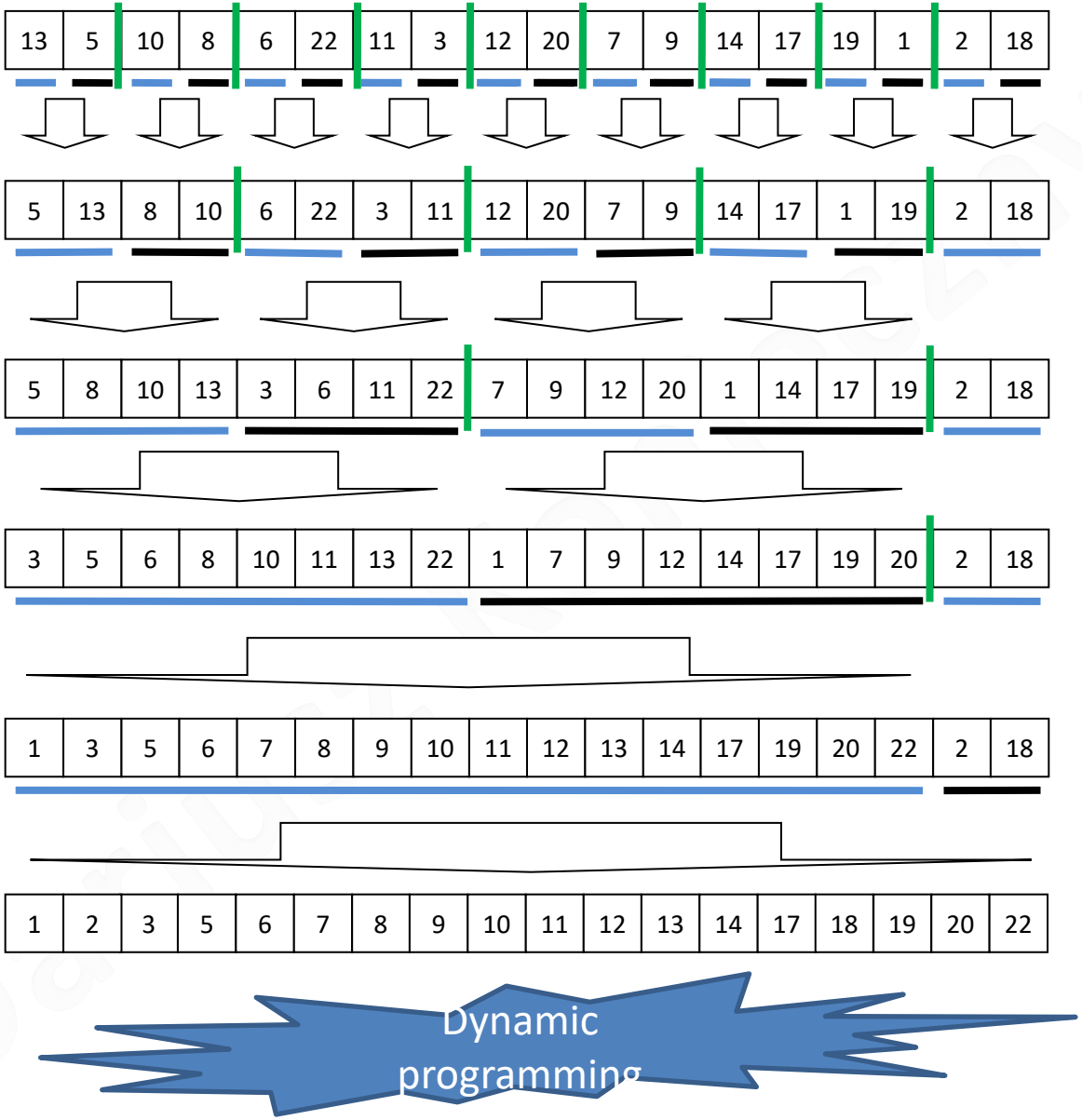
Mergesort – code 2/2

```
@SuppressWarnings("unchecked")
private IList<T> merge(IList<T> left, IList<T> right) {
// after all, we have to decide on the specific implementation of the list.
  IList<T> result = (IList<T>)new ArrayList<Object>();
  Iterator<T> l = left.iterator();
  Iterator<T> r = right.iterator();
  T elemL=null,elemR=null;
// we have to delay leaving the loop until the last element of one of the
// arrays is added to the result
  boolean contL,contR;
  if(contL=l.hasNext())    elemL=l.next();
  if(contR=r.hasNext())    elemR=r.next();
  while (contL && contR) {
    if (_comparator.compare(elemL, elemR) <= 0){
      result.add(elemL);
      if(contL=l.hasNext())   elemL=l.next();
      else result.add(elemR);} //already read element of the second list to the result
    else {
      result.add(elemR);
      if(contR=r.hasNext())    elemR=r.next();
      else result.add(elemL);} // already read element of the first list to the result
  }
  while(l.hasNext()) result.add(l.next());
  while(r.hasNext()) result.add(r.next());
  return result;}
```

Mergesort - analysis

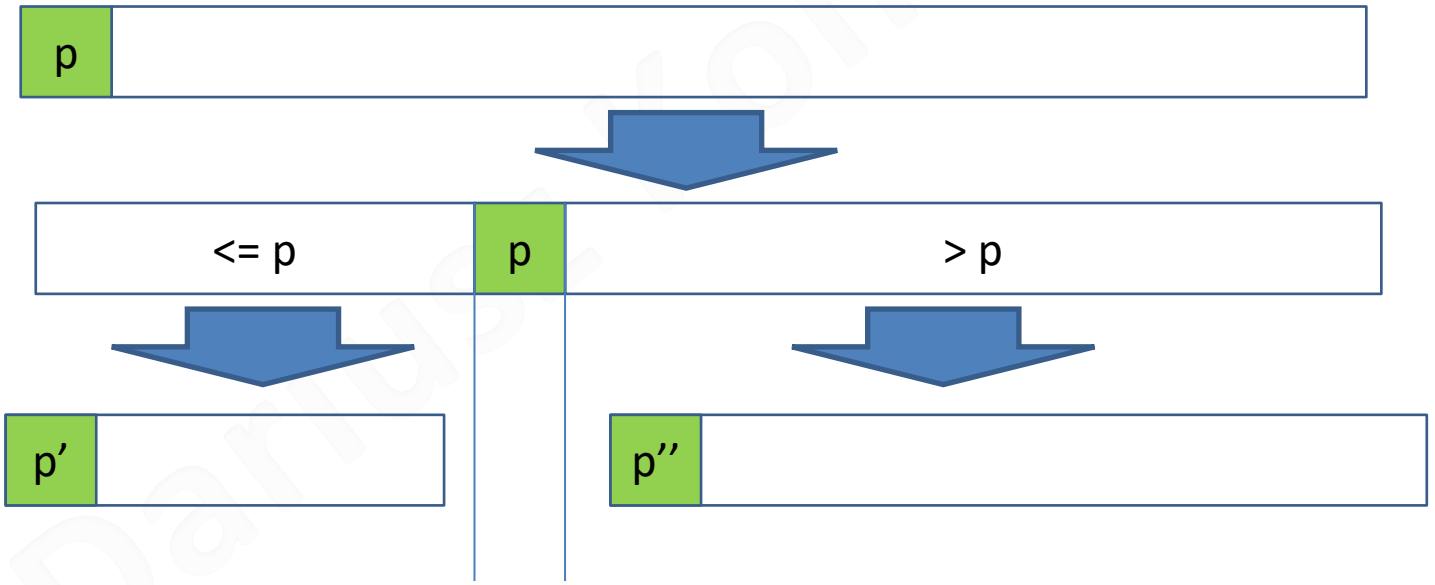
- Computational complexity
  - Pessimistic, average: O(n log n)
- The complexity of additional memory: $\Omega(n)$
- Stable.
- Easy to parallelize
- You can implement this code in the iteration version:
  - With a stack of lists / tables:
    - Refactoring a recursive version
  - With the queue of lists/tables:
    - Firstly, single-element arrays from each element of the list input are sent to the queue.
    - We take two lists from the queue, we merge and the result enqueue at the end of the queue.
  - No stacks/queues for random access lists:
    - Without dividing, we treat fragments of the list as subarrays of length 1, then 2, 4, 8, 16 etc. in the same way as for a solution with a queue.

Iterative mergesort - example



Quicksort

- Idea: a pivot element is selected from the array. On the left of the array we collect all elements which are smaller ot equal to the pivot, and on the right part we put elements bigger than the pivot. We place the pivot between this two parts and this is its final place. We do the same with each subarrays recursively. Of course, single-element arrays are sorted and recursive returns.
- The division into these two tables can be done "in place" during O(*n*).



Quicksort - pseudocode

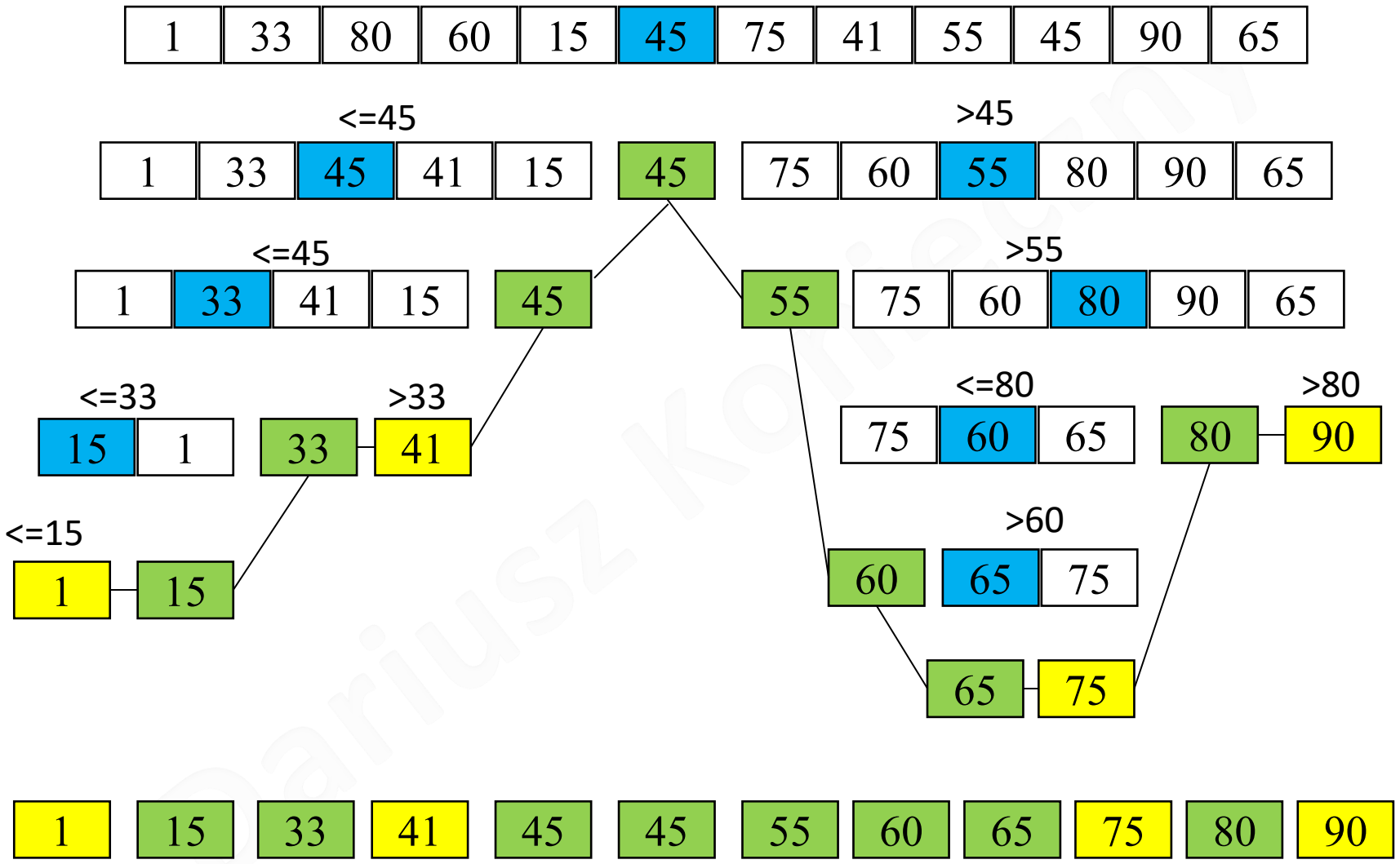
```
// for array from position p to position r
quicksort(A,p,r)
{
  // if the array is longer then one
  if(p<r-1)
  {
    // using selected pivot, divide the array into two part:
    // left one with elements less ot equal pivot,
    // and right one with elements bigger than pivot
    // q ist a posiotion of partitioning
    q=partitioning(A,p,r);
    // sort the left part in place
    quicksort(A,p,q);
    // sort the right part in place
    quicksort(A,q+1,r);
    // now the part from p to r is sorted
  }
}
```

45 Pivot in current step

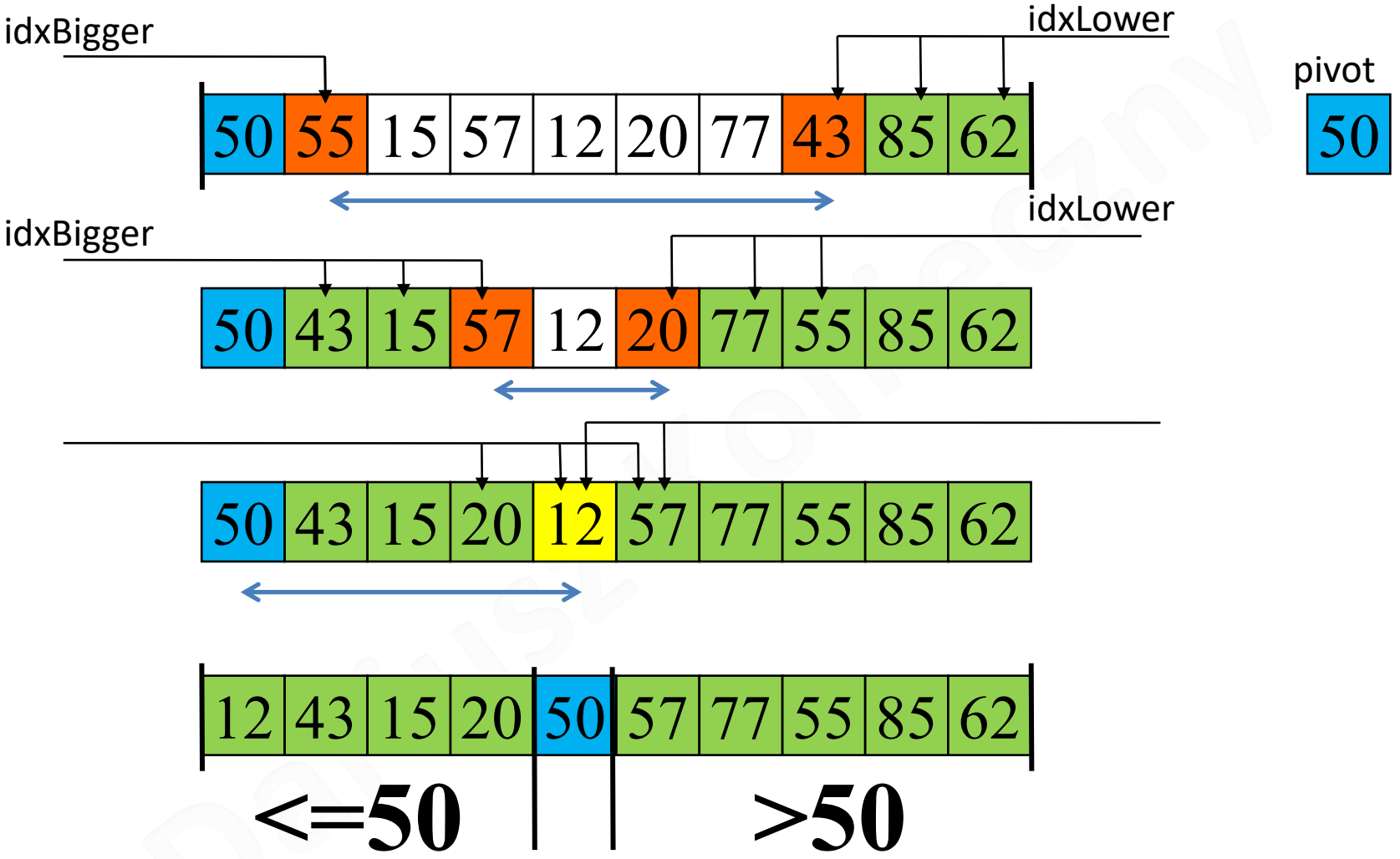
41 One-element array

33 Pivot from previous step on its final position

Steps in quicksort



Division in quicksort



Quicksort – code 1

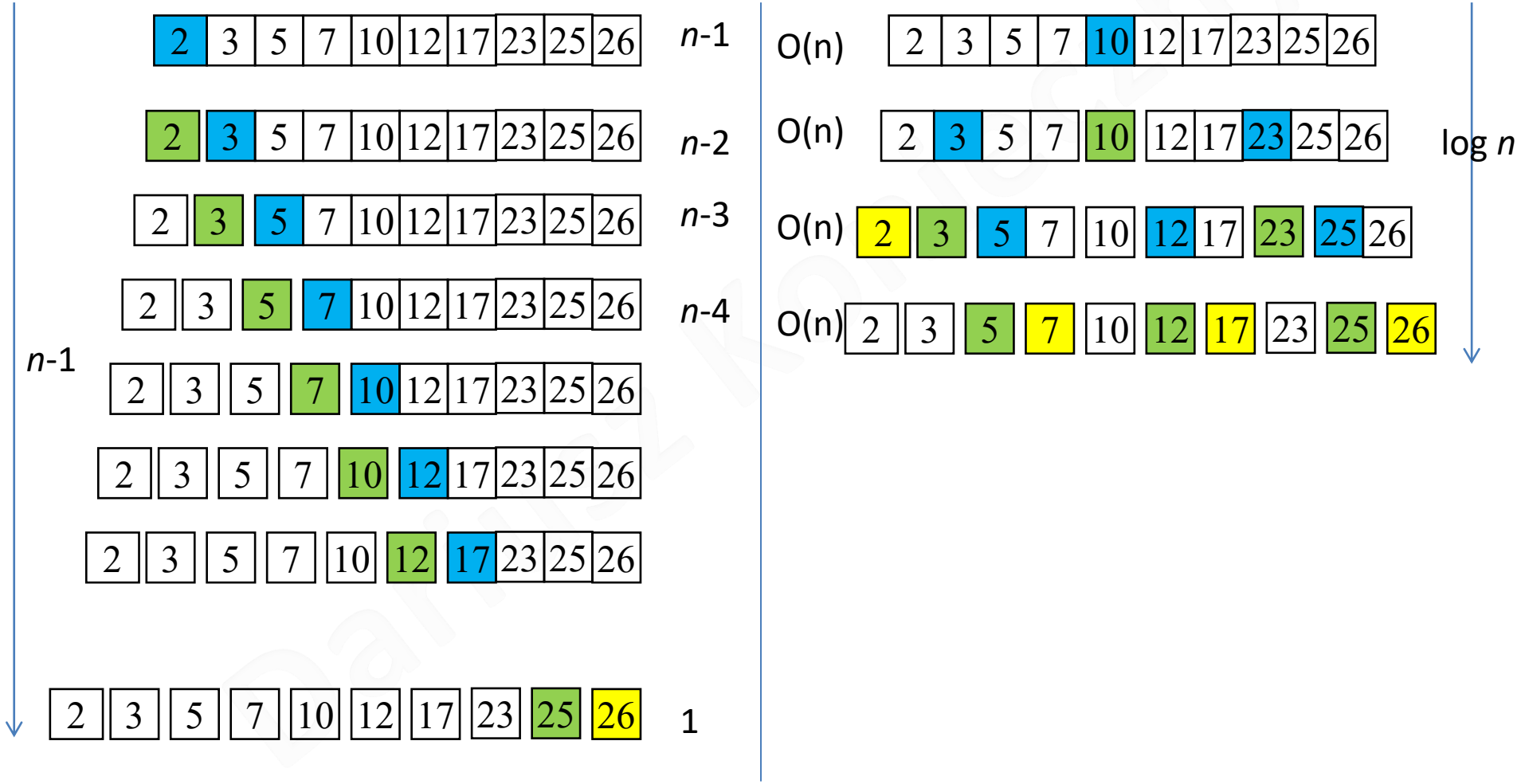
```
public IList<T> sort(IList<T> list) {
  quicksort(list, 0, list.size());
  return list;}
private void quicksort(IList<T> list, int startIndex, int endIndex) {
if (endIndex - startIndex > 1) {
  int partition = partition(list, startIndex, endIndex);
  quicksort(list, startIndex, partition );
  quicksort(list, partition + 1, endIndex);}}
private int partition(IList<T> list, int nFrom, int nTo) {
  //as a pivot we take a random one
  int rnd=nFrom+random.nextInt(nTo-nFrom);
  swap(list,nFrom,rnd);
  T value=list.get(nFrom);
  int idxBigger=nFrom+1, idxLower=nTo-1;
  do{
    while(idxBigger<=idxLower &&  _comparator.compare(list.get(idxBigger),value)<=0)
      idxBigger++;
    while(_comparator.compare(list.get(idxLower),value)>0)
      idxLower--;
    if(idxBigger<idxLower)
      swap(list,idxBigger,idxLower);
   }while(idxBigger<idxLower);
   swap(list,idxLower,nFrom);
   return idxLower;}
private void swap(IList<T> list, int left, int right) {
  if (left != right) {
    T temp = list.get(left);
    list.set(left, list.get(right));
    list.set(right, temp);}
}
```

Quicksort – code 2(Lomuto)

```
// the result is in the same list
public IList<T> sort(IList<T> list) {
  quicksort(list, 0, list.size() - 1);
  return list;
}
private void quicksort(IList<T> list, int startIndex, int endIndex) {
  if (endIndex > startIndex) {
    int partition = partition(list, startIndex, endIndex);
    quicksort(list, startIndex, partition );
    quicksort(list, partition + 1, endIndex);}
}
// division according to Lomuto
private int partition(IList<T> list, int left, int right) {
  // as a pivot we take the last one
  T value=list.get(right);
  int i=left-1;
  while (left <= right){
    if( _comparator.compare(list.get(left), value) <= 0)
      swap(list, ++i,left);
    ++left;}
  return i<right ? i :i-1;
}
private void swap(IList<T> list, int left, int right) {
  if (left != right) {
    T temp = list.get(left);
    list.set(left, list.get(right));
    list.set(right, temp);}
}
```

Quicksort – analysis 1/2

- The division depends on the pivot and the distribution of numbers
- The best and worst case

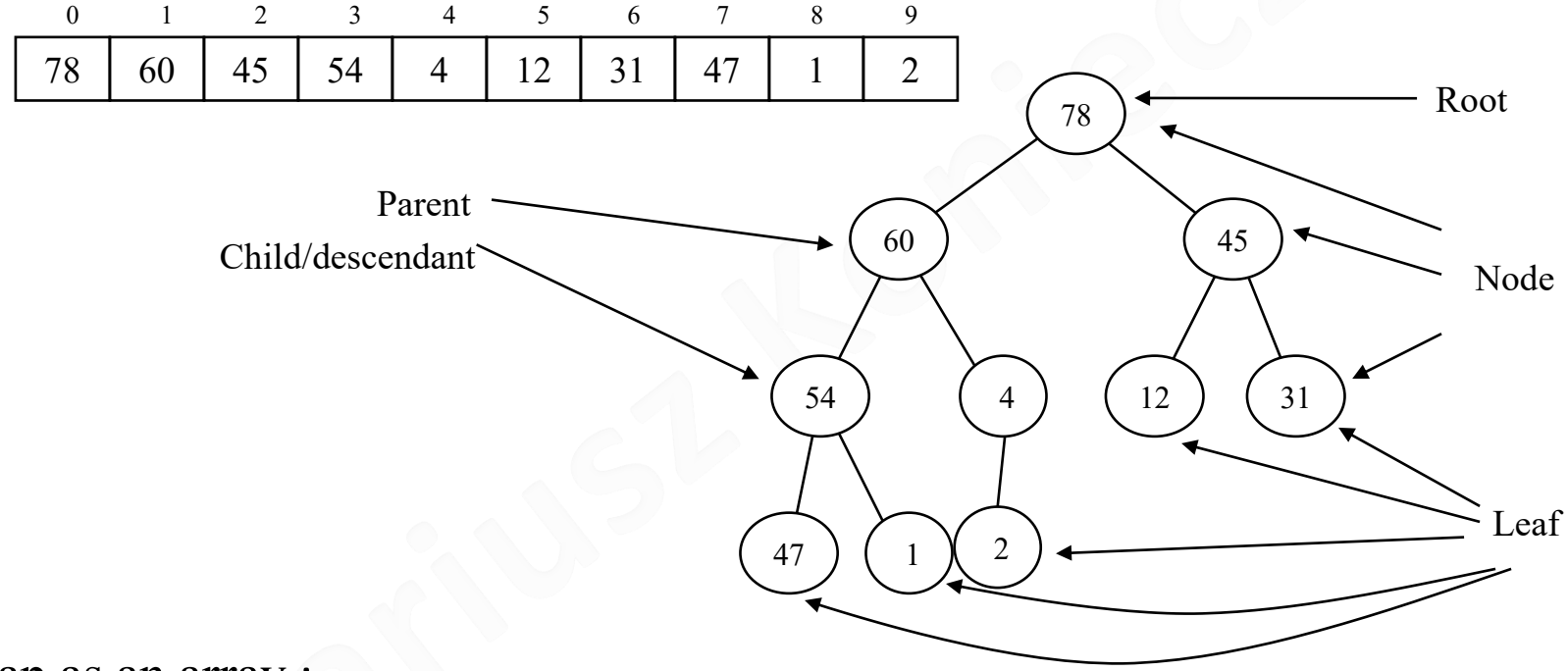


Quicksort – analysis 2/2

- Worse-case complexity - **O(n^2)**
- In the literature, one can find proof that the average expected time complexity is **O(n log n)**.
- Theoretically in-place, but you need space on the stack of function calls of order **Ω(log n)** .
- Unstable.
- Selecting a pivot:
  - Incorrect: The first element, the middle element, the last element .
  - Good: a random element
  - For large n:
    - choose 3 pieces randomly and "medium" is pivot
    - choose k ($k << n$) random elements, sort with a simple algorithm, select the median of the sorted sequence
- For adequately small n, stop the recursion in favor of the simple algorithm.
- Improvements for many equal values

Heap - definition

- The binary heap is an array object that can be viewed as a **nearly complete binary tree**:
  - The tree is **completely filled on all levels except** possibly the **lowest**
  - The **lowest level** is **filled** from the left **up to a point**.
- The value of a node **is at most the value** of its parent



Heap as an array :

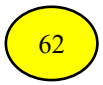
1) a node with index i has two children with indexes $2*i+1$ and $2*i+2$

2) a node with index i has a parent with index $\left\lfloor \frac{i-1}{2} \right\rfloor$

Creation of a heap

- Make the correct heap from any array.
- You can use the original array.
- The idea is to repair the heap from the bottom (first level after the leaves).
- The node and its descendants are analyzed. When it turns out that the parent does not have the highest value - it is exchanged with a larger value of the descendants. This swap may break the rule at a lower level, so recursively, you need to repair this part of the heap often up to the leaf.
- Since the heap has the height log *n*, then for each of *n* nodes it is necessary to do a maximum of log *n* steps, so in creating the heap requires O(*n* log *n*) time.
  - A more detailed analysis of the algorithm allows to state that the method of building a heap is carried out in **O(*n*)**. time.

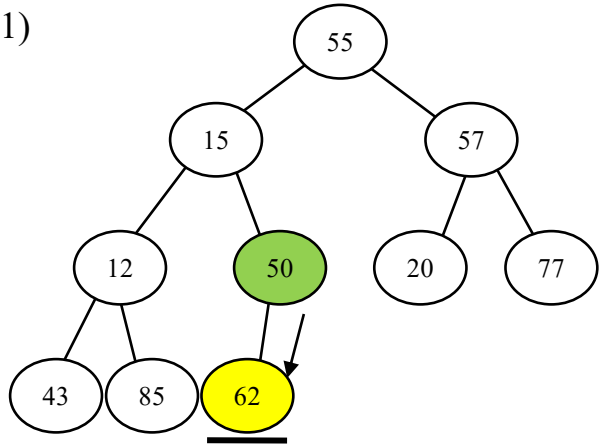
Node tested



Descendant of the tested node

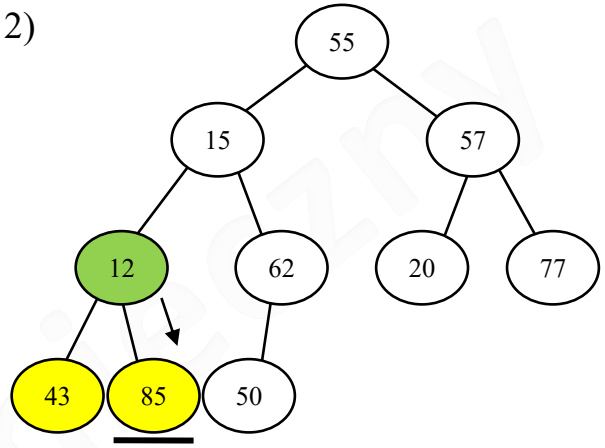
heapAdjustment, sink 1/2

1)



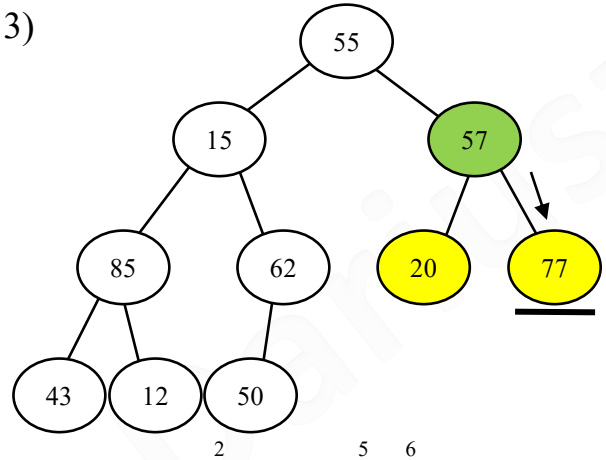
55	15	57	12	50	20	77	43	85	62
				4					9

2)



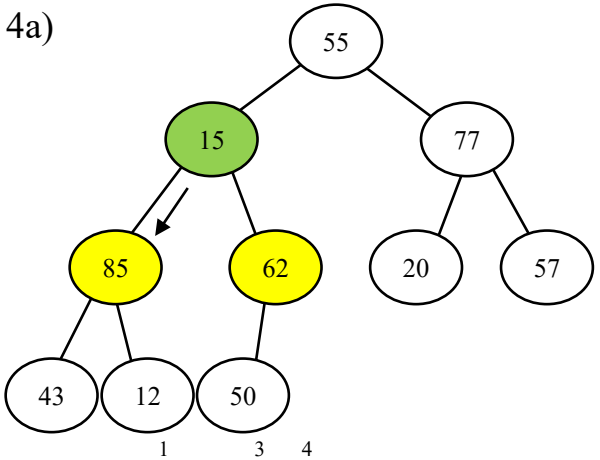
55	15	57	12	62	20	77	43	85	50
			3				7	8	

3)



55	15	57	85	62	20	77	43	12	50
		2			5	6			

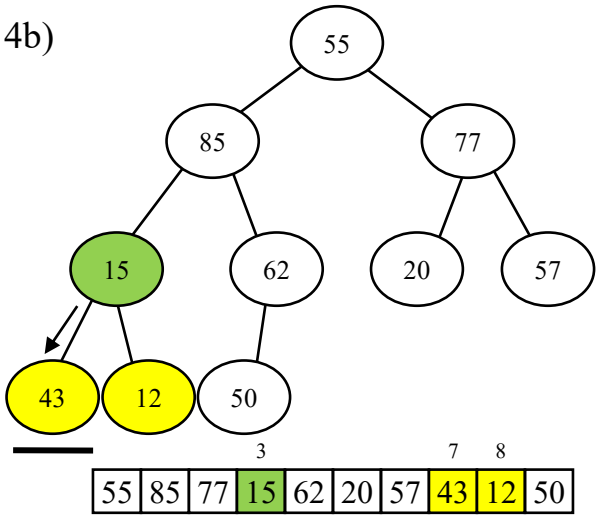
4a)



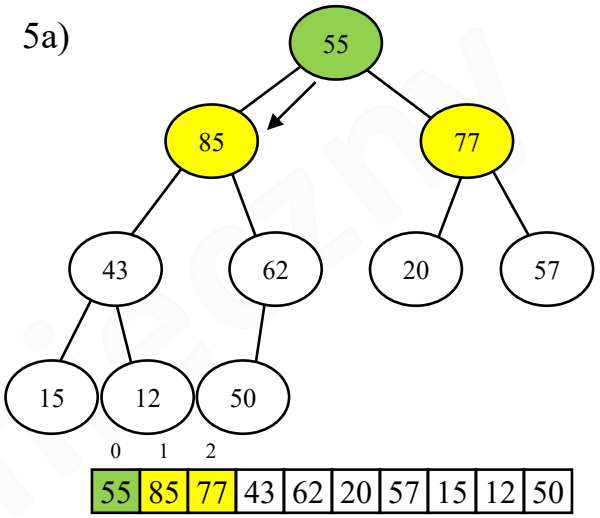
55	15	77	85	62	20	57	43	12	50
	1		3	4					

heapAdjustment, sink 2/2

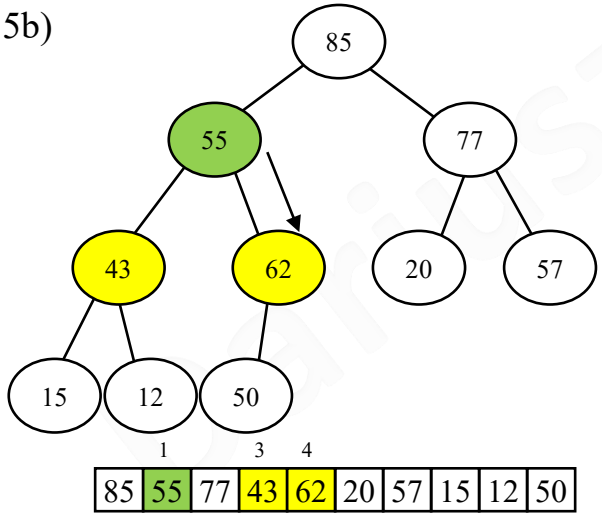
4b)



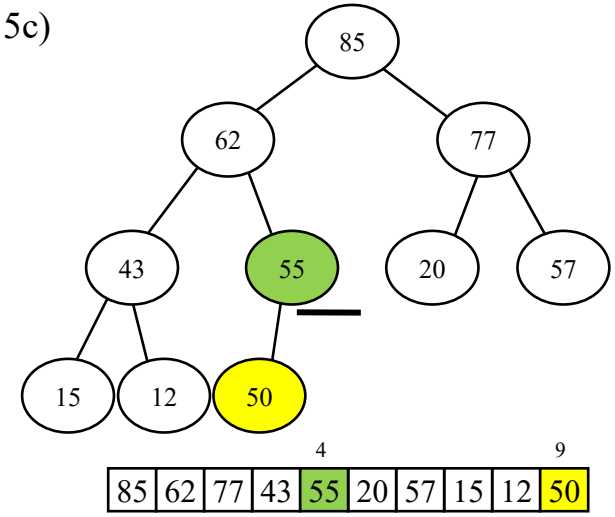
5a)



5b)



5c)



Heap creation

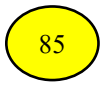
```
/** assumption: left != right */
private void swap(IList<T> list, int left, int right) {
  T temp = list.get(left);
  list.set(left, list.get(right));
  list.set(right, temp);
}

public void sink(IList<T> heap,int idx, int n){
  int idxOfBigger=2*idx+1;
  if(idxOfBigger<n){
    if(idxOfBigger+1<n &&
     _comparator.compare(heap.get(idxOfBigger), heap.get(idxOfBigger+1))<0)
        idxOfBigger++;
    if(_comparator.compare(heap.get(idx), heap.get(idxOfBigger))<0){
      swap(heap,idx,idxOfBigger);
      sink(heap,idxOfBigger,n);
    }
  }
}

void heapAdjustment(IList<T> heap,int n)
{
  for(int i=(n-1)/2;i>=0;i--)
    sink(heap, i, n);
}
```

Heapsort

- 1) Creation of a heap
- 2) heap -> sorted array: replacement of the root with the last element. After swapping, the last element of the array is no longer part of the heap. However, you need to repair the heap from the root. The next steps look similar.
- For each element, O(log *n*) steps should be made, that is, a total of O(*n* log *n*) steps.

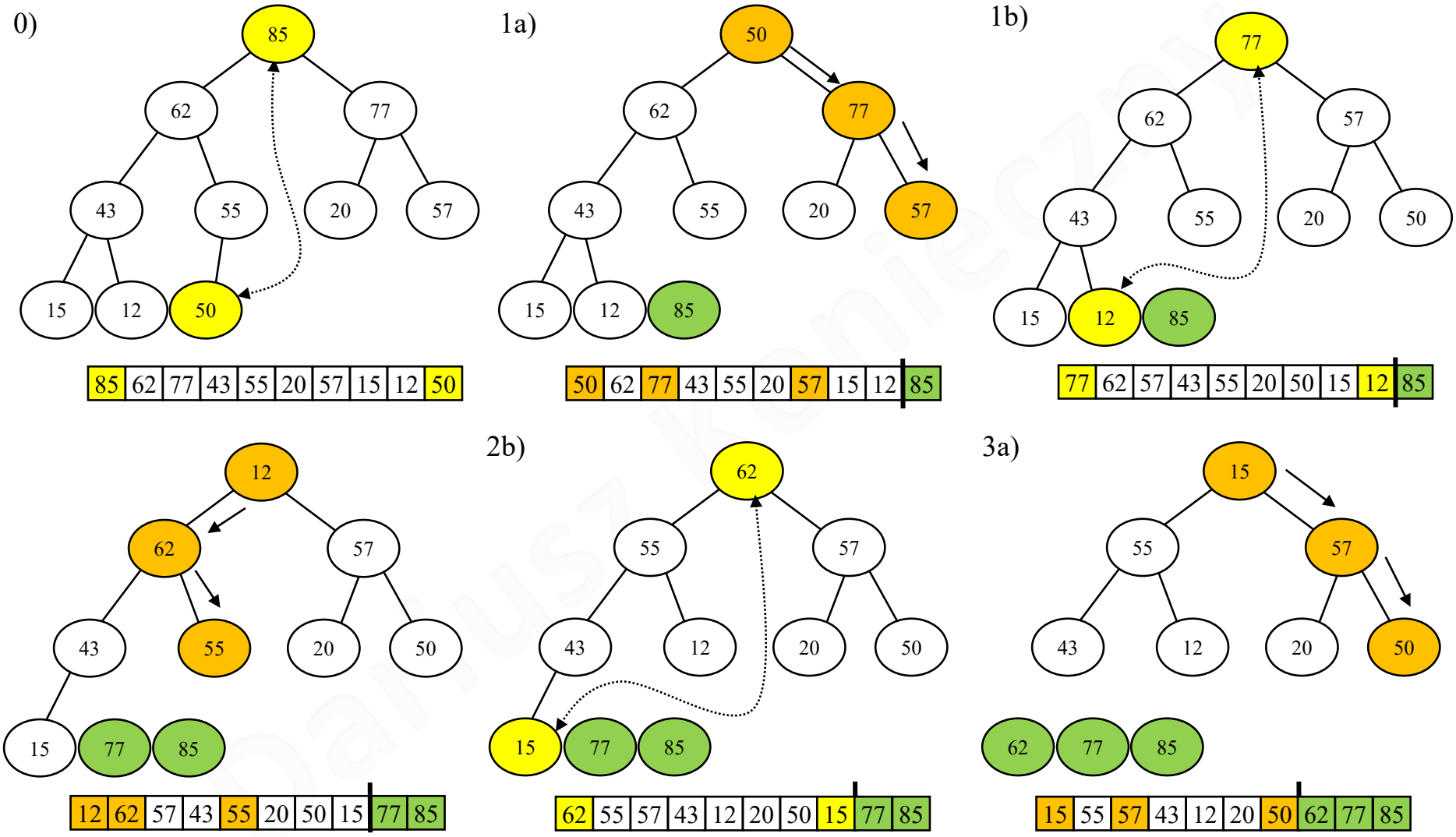


The element that can be replaced

Element sorted,
no longer belongs to the heap

Elements moved when repairing the heap

Heapsort - example



Heapsort-code, property

```
// result is in the original array
@Override
public IList<T> sort(IList<T> list){
  heapsort(list, list.size());
  return list;}

private void heapsort(IList<T> heap, int n) {
  heapAdjustment(heap, n);
  for(int i=n-1;i>0;i--){
    swap(heap,i,0);
    sink(heap,0,i);
  }
}
```

- Time complexity:
  - Average, worse-case : O(*n*log*n*)
  - However, slower than quicksort and mergesort
  - For equal data, it works in time O (n)
- Memory complexity: O (1) - it is not difficult to write an iterative version of the `sink ()` function.
- Features:
  - In place
  - Unstable
  - The heap can be used for priority queues (see next lectures)

Examples of performance times

- *n*=10.000.000 random elements of type Integer
- Execution time:

```
MergeSort          : 18826 milliseconds
IterMergeSort      : 11482 milliseconds
QuickSort          : 11841 milliseconds
QuickSortBetter    : 11939 milliseconds
HeapSort           : 40564 milliseconds
```

```
MergeSort          : 18995 milliseconds
IterMergeSort      : 15083 milliseconds
QuickSort          : 12817 milliseconds
QuickSortBetter    : 10354 milliseconds
HeapSort           : 31106 milliseconds
```

- *n*=10.000 random elements of type Integer
- Execution time:

```
BubbleSort         : 640 milliseconds
InsertSort         : 453 milliseconds
SelectSort         : 312 milliseconds
MergeSort          : 32 milliseconds
Iter.MergeSort     : 16 milliseconds
QuickSort          : 32 milliseconds
QuickSortBetter    : 31 milliseconds
HeapSort           : 31 milliseconds
```

- *n*=50.000 random elements of type Integer
- Execution time:

```
BubbleSort         : 20059 milliseconds
InsertSort         : 9970 milliseconds
SelectSort         : 8860 milliseconds
MergeSort          : 156 milliseconds
Iter.MergeSort     : 109 milliseconds
QuickSort          : 156 milliseconds
QuickSortBetter    : 78 milliseconds
HeapSort           : 110 milliseconds
```

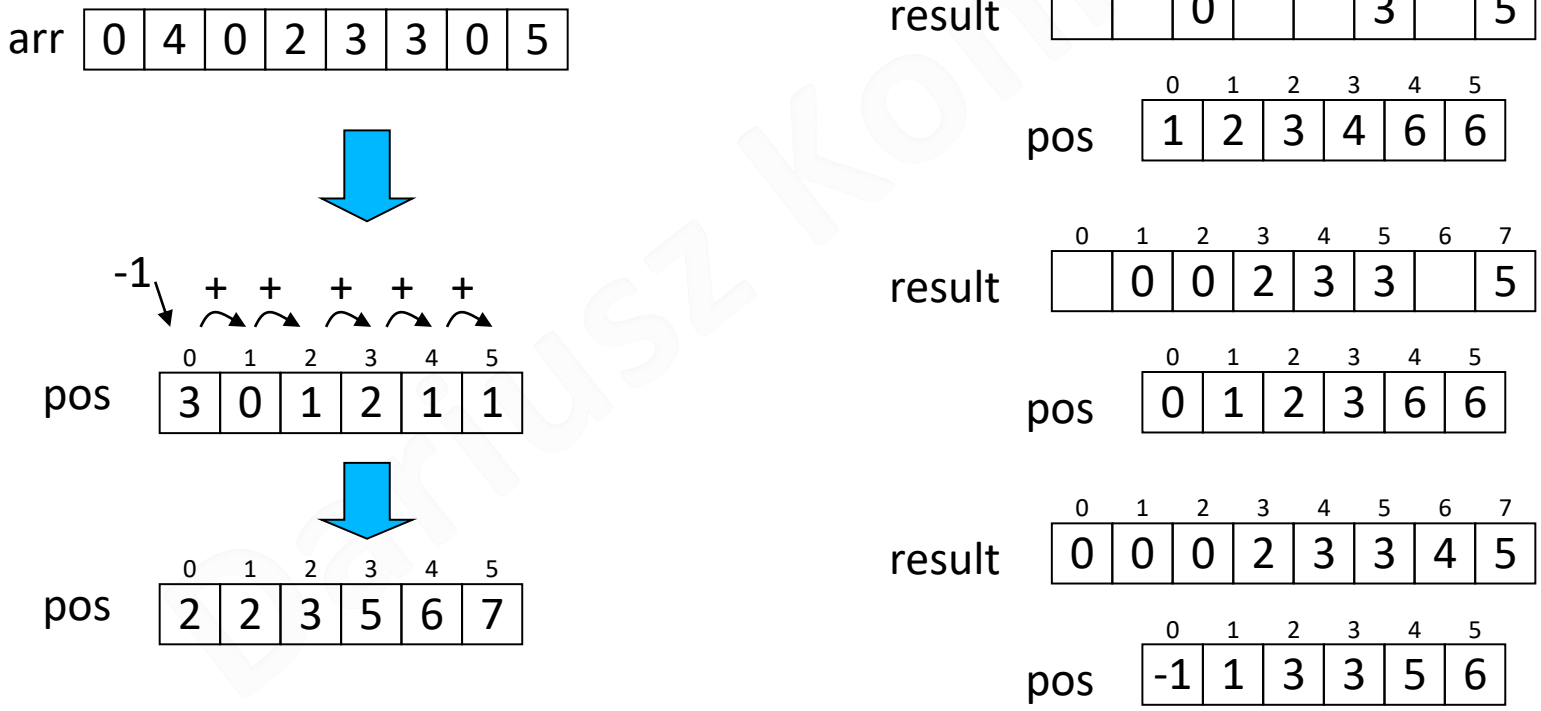
Other sorting

- Sorting without comparing items with each other:
  - most often for keys that are numbers
  - They can be treated as natural numbers from certain ranges
  - Or even they must be numbers in the positional system
  - Or we know the distribution / range of key numbers
- Computational complexity (with appropriate assumptions): O(*n*)

Countingsort

- Let each of the n input data be a number in the range from 0 to k, for some fixed k,
- If k = O (n) then sorting works in time O (n).
- Idea: count, for each input element x, the number of elements smaller than x. This information can be used to insert an element x directly to the correct position of the result array
- Time complexity: $\Theta(n+k)$, or $O(n)$
- The complexity of additional memory: $\Theta(n+k)$

n= 8, k=5



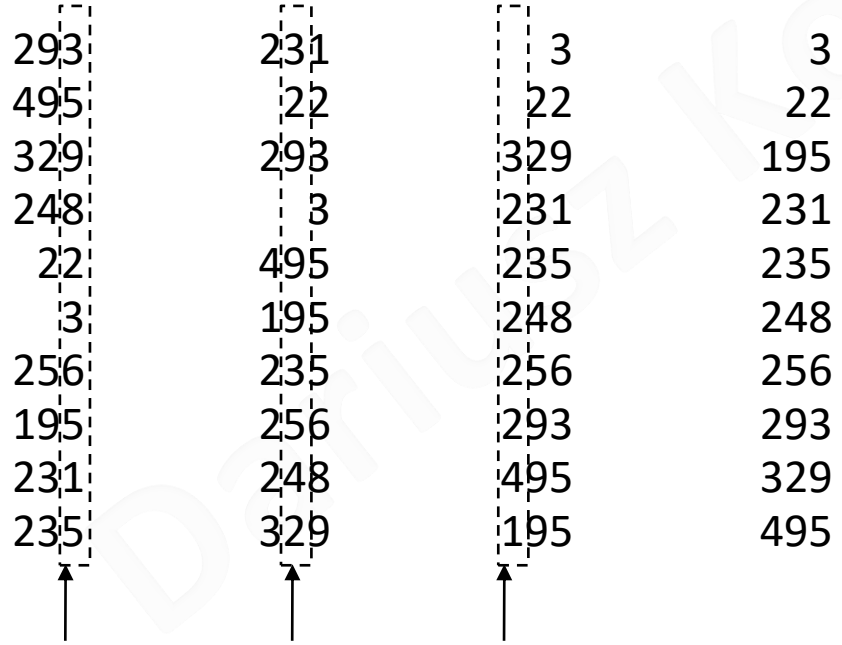
Countingsort - code

```
public static void countingSort(int arr[], int k)
{
  k++;
  int n=att.length;
  int pos=new int[k];
  int result=new int[n];
  int i,j;
  for(i=0;i<k;i++)
    pos[i]=0;
  for(j=0;j<n;j++)
    pos[arr[j]]++;
  pos[0]--;
  for(i=1;i<k;i++)
    pos[i]+=pos[i-1];
  for(j=n-1;j>=0;j--)
  {
    result[pos[arr[j]]]=arr[j];
    pos[arr[j]]--;
  }
  for(j=0;j<n;j++)
    arr[j]=result[j];
}
```

Radix sort

- For non-negative integer numbers
- pseudocode:

```
RadixSort(A, d) //d – number of digits
  for i from 0 to d-1
    use stable sorting for A with respect to a digit i
```



We can use countingsort!

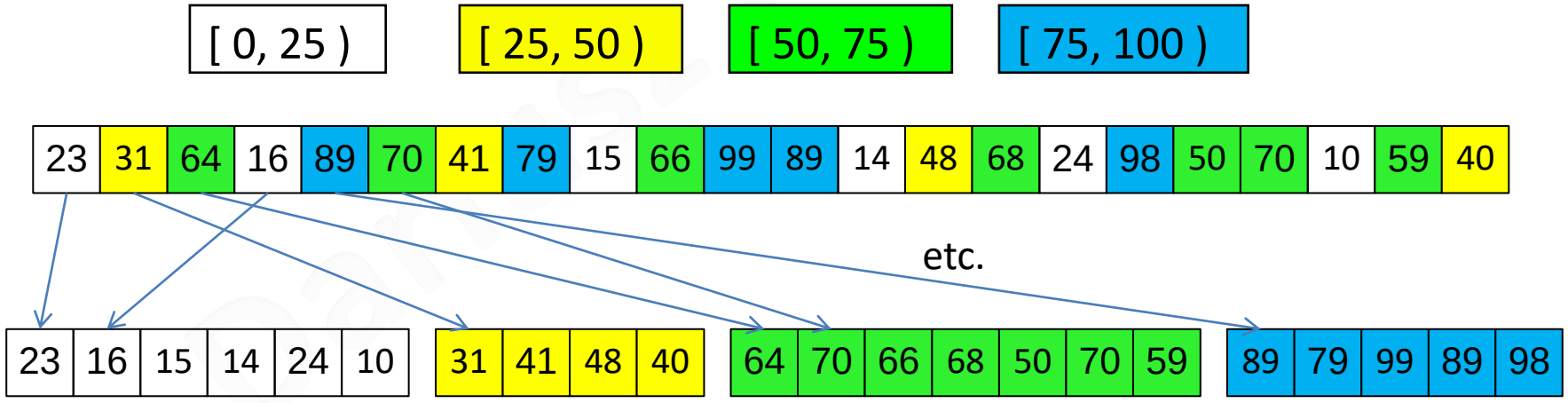
Complexity: $\Omega(d*(n+k))$

If d is fixed: $O(n)$

Bucket sort

- Assumptions:
  - We know the range of the numeric key value (it can be floating point) – X
  - The key values are evenly distributed
- Idea:
  - Range division into m equal parts (buckets)
  - Allocation of data according to the key to the appropriate buckets in O(1)
  - Sort buckets
- Time complexity (under certain assumptions): O(*n*)
- Pseudocode:

```
for (i=0; i<n; i++){
  pr=floor (el[i]/X*M)
  enqueue(Q[pr],el[i])} // e.a. to a priority queue
```



Comparision simple algorithms

Algorithm	Stability	Compare	Assigments/ Move	Time Complexity
insertSort - array	YES	O(n log n)	$O(n^2)$	$O(n^2)$
insertSort – linked list	YES	O(n log n)	$O(n^2)$	$O(n^2)$
selectSort - array	YES	$O(n^2)$	O(n)	$O(n^2)$
selectSort – linked list	YES	$O(n^2)$	O(n)	$O(n^2)$
bubbleSort- array	YES	$O(n^2)$	$O(n^2)$	$O(n^2)$
bubbleSort – linked list	YES	$O(n^2)$	$O(n^2)$	$O(n^2)$

Comparision effective algorithms

Algorithm	Stability	Time Complexity	Memory Complexity
mergeSort - array	YES	O(n log n)	O(n) Merge operation
mergeSort – linked list	YES	O(n log n)	O(n) queue
quickSort - array	NO	O(n log n)	O(log n) recurency
quickSort – linked list	NO	O(n log n)	O(log n) recurency
heapSort-array	NO	O(n log n)	O(1)
heapSort – linked list	-	-	-

Comparision special algorithms

Algorithm	Stability	Time Complexity	Memory Complexity
Counting sort - array	YES	O(n+k)	O(n+k)
Counting sort – linked list	YES	O(n)	O(k)
RadixSort - array	YES	O(d*n)	O(k)
RadixSort – linked list	YES	O(d*n)	O(k)
BuckedSort - array	YES	O(n)	O(n)
BuckedSort – linked list	YES	O(n)	O(k)

Tech - Sorting in Java library

- Class `Arrays` with the `sort (...)` method for arrays:
  - Call with only array - sorts the entire array
  - Call with an array and index range - sorts a part of the array
  - Uses quicksort
- The `Collections` class with the `sort (...)` method for lists:
  - Uses mergesort
- The above methods use either a built-in comparison or a natural comparator, you can call them with an additional parameter - a comparator.
- There are classes and methods that transform any collections into arrays or lists.
- For other collections (eg maps, hash tables) there are special sorting methods.
- Since Java 8.0, you can use lambda expressions for a short code of the comparator
- From Java 8.0 in streaming (lazy), instead of a complex comparator, you can use the methods of the class `Comparator:comparing (...)` and `thenComparing (...)`