Algorithms: Lecture 6

Chalmers University of Technology

Recap

- Greedy & Dynamic Programming
  - ➢extend solutions from smaller sub-instances incrementally to larger sub-instances, up to the full instance.
- Divide & Conquer
  - ➢follows the pattern of reducing a given problem to smaller instances of itself

  BUT

  - ➢it makes jumps rather than incremental steps.

Recap

- **Divide-and-conquer**
  - Split problem instance into a few significantly smaller sub-instances.
  - Sub-instances are solved, independently, in the same way (recursion).
  - Combine partial solutions to sub-instances into an overall solution.

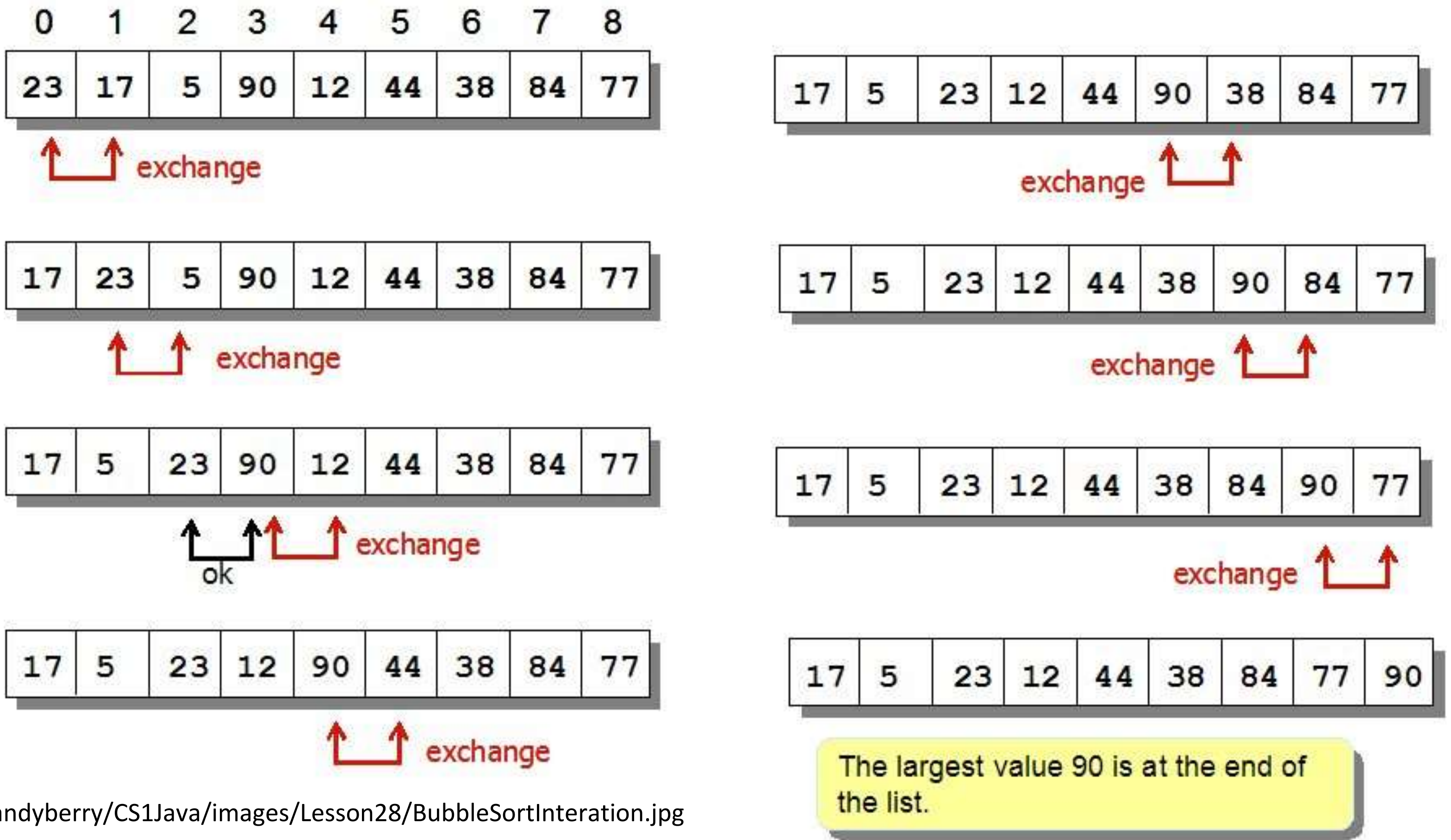
- **Most common usage**
  - Break up problem instance of size n into two equal parts of size ½n.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in linear time.

Today’s Lecture

- **Important technique for Searching & Sorting**
  - Binary Search O(log n) (last lecture)
  - Brute force Sorting, e.g., Bubble sort : $O(n^2)$.
  - Divide-and-conquer: O(n log n).

Bubble Sort

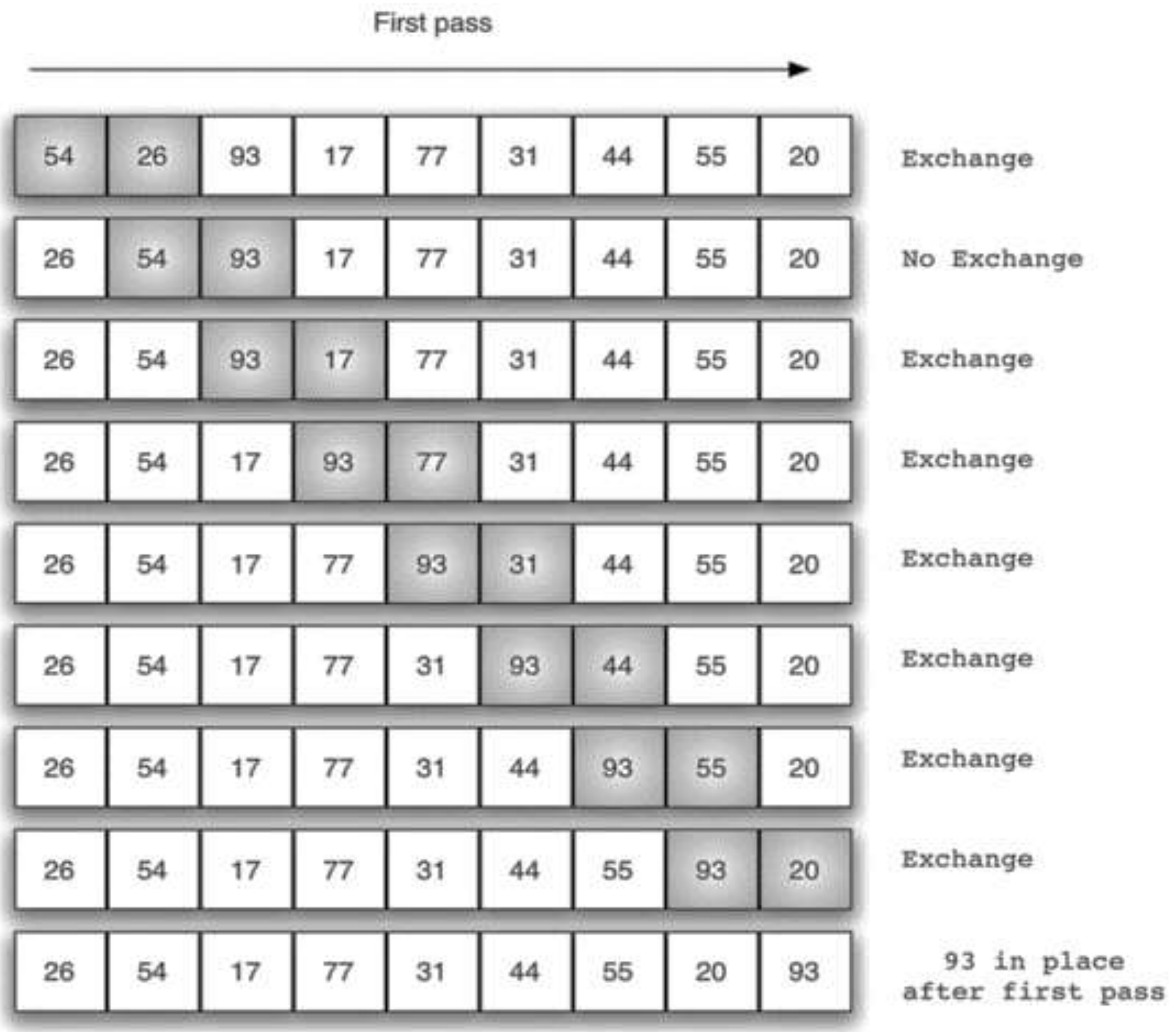
- Scan the list of elements from left to right
  - ➢whenever two neighbored elements are in the wrong order, swap them.



http://www.csit.parkland.edu/~mbrandyberry/CS1Java/images/Lesson28/BubbleSortInteration.jpg

Bubble Sort

- Every pass puts one elements to its proper place & reduces the instance size by 1
  - **n(n-1)/2**
    - **$O(n^2)$**



Bubble Sort

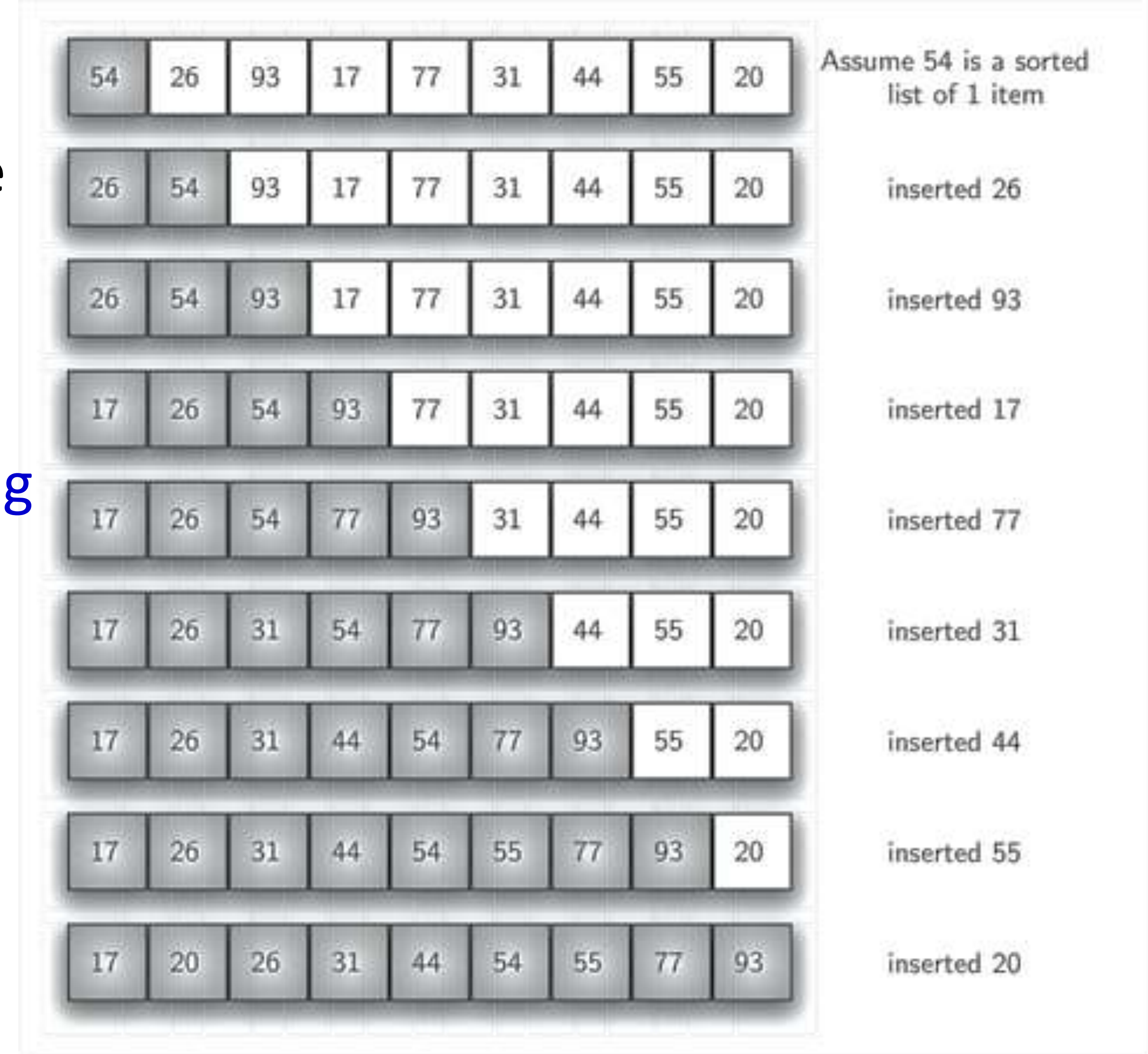
- **In place:** Needs only one array of size n for everything, except, possibly a few memory units.

- **Best:** In the first pass, if we don't have to make any swaps, that means that the array is sorted already.

- **Worst:** if many elements are far from their proper places(reverse order), because the algorithm moves them only step by step.
  - **Insertion Sort to overcome**

Insertion Sort

- After k rounds of Insertion Sort, the first k elements (k = 1,…,n) are sorted.
- To insert the (k + 1)st element we search for the correct position, using binary search.
  - ➢ **O(n log n)?**
  - ➢ we may be forced to move O(k) elements in the k-th round, giving again an overall time complexity of **$O(n^2)$.**



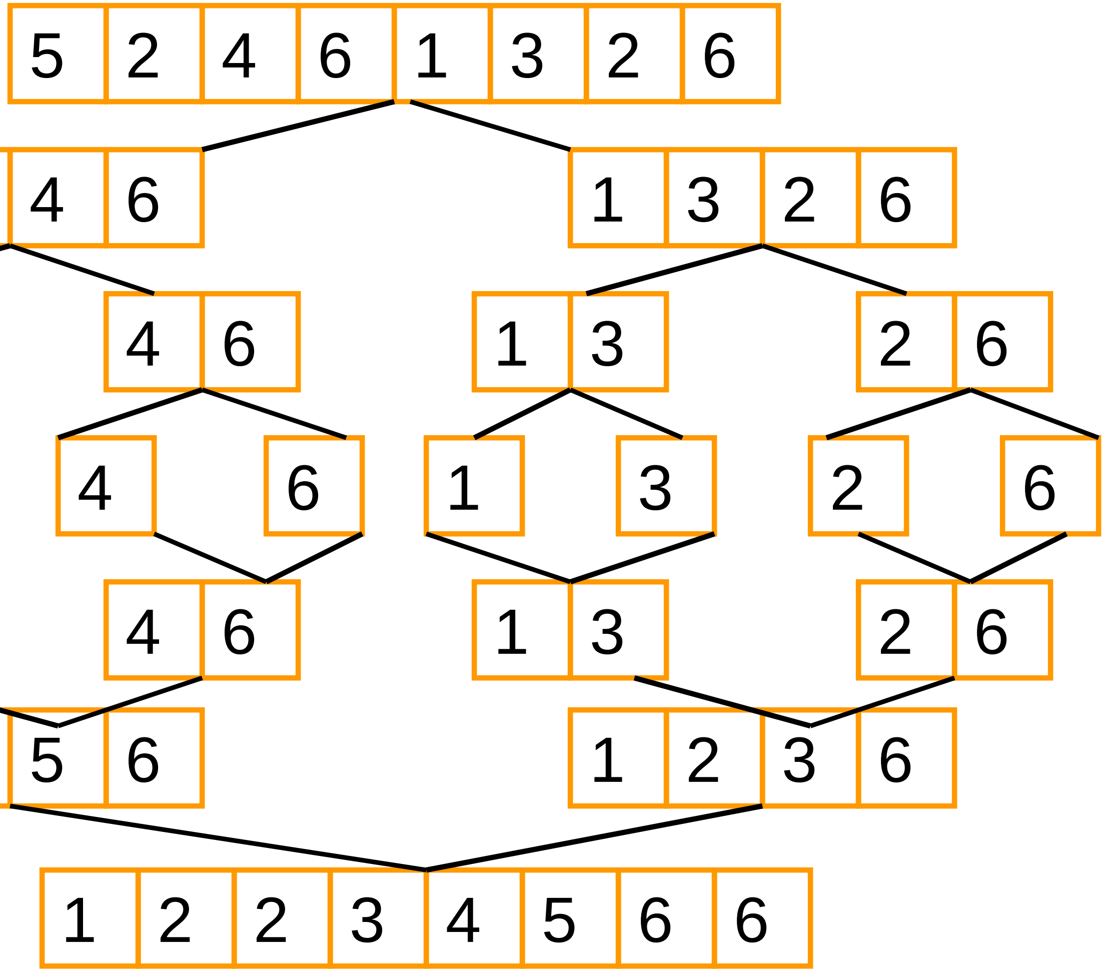
Insertion Sort

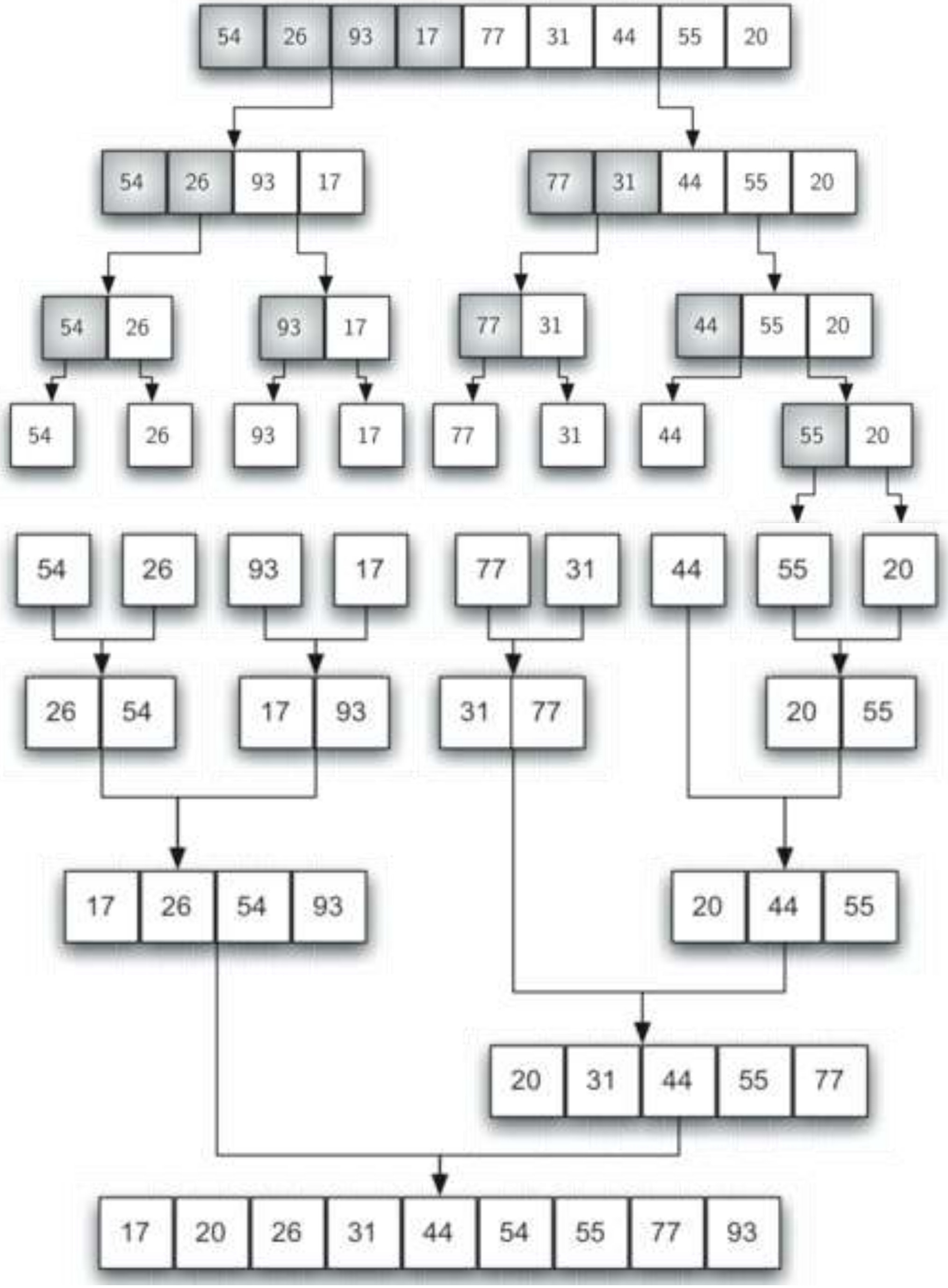
- **Idea:** We can avoid moving the elements
  - ➢ Insert an element in $O(1)$ time at a desired position using doubly linked list.
- But, how do we apply Binary Search without indices?
  - ➢ We have to apply linear search, and once again: **$O(n^2)$** for all n rounds.
- However, $O(n \log n)$ sorting algorithms are known, as we already know
  - ➢ **Divide-and-conquer**

Mergesort

- **How it works:**
  - arbitrary split the set into two halves
  - recursively sort the two halves separately
  - merge the two sorted halves
- Merging the two sorted halves involves *comparing* the elements to each other
  - ➢ scan both ordered sequences simultaneously and always move the currently smallest element to the next position in the result sequence, **implies O(n)**

Mergesort Example




54 26 93 17 77 31 44 55 20
54 26 93 17
77 31 44 55 20
54 26
93 17
77 31
44 55 20
54
26
93
17
77
31
44
55 20
55
20
54
26
93
17
77
31
44
55
20
26 54
17 93
31 77
20 55
17 26 54 93
20 44 55
20 31 44 55 77
17 20 26 31 44 54 55 77 93

Time Complexity for Mergesort

Recurrence Relation:

- Let $T(n)$ be worst case time on a sequence of n elements
- If $n = 1$, then $T(n) = O(1)$ (constant)
- If $n > 1$, then $T(n) = 2\ T(n/2) + O(n)$
  - two sub-problems of size $n/2$ each that are solved recursively
  - $O(n)$ time to do the merge
- Solving the recurrence gives $T(n) = O(n \log n)$
- Remember general result from the Master Theorem
  - ➢ **$T(n) = aT(n/b) + cn^k$, and for $a = b^k$ it gives $O(n^k \log n)$**
  - ➢ For Mergesort, we have **$a=2$, $b=2$ and $k= 1$.**

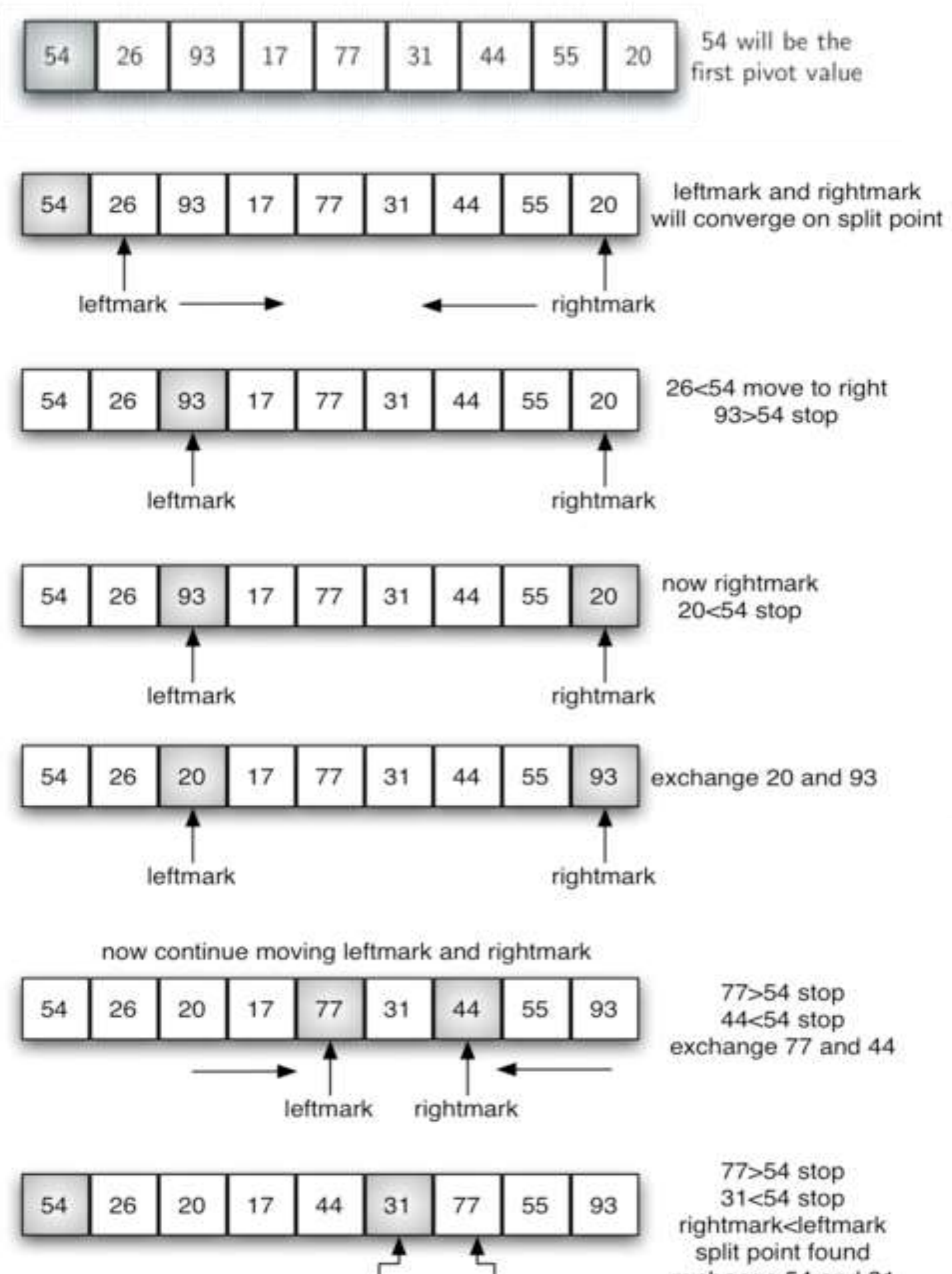
Caveat

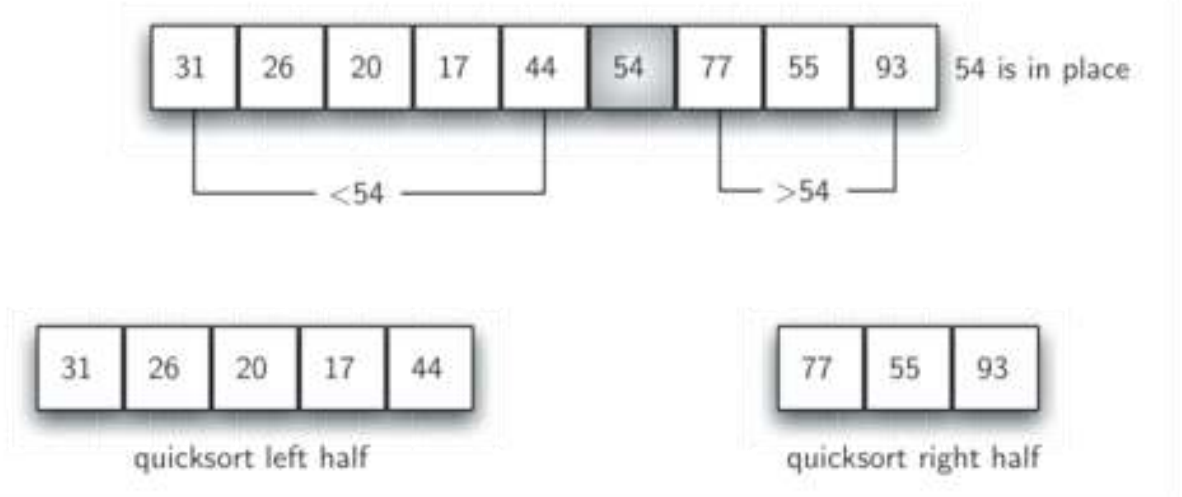
- **Simple Structure but** not the fastest sorting algorithm in practice
  - Too many copy operations (In every merging phase on every recursion level we have to move all elements of the merged subsets into a new array.)
- NOT *in place*
  - Additional memory required, while n could be very large in practice.
- Other alternatives with O(n log n) time
  - Different hidden constants factors
  - Hard to analyze theoretically
  - runtime experiments can figure out what is really faster.

- Remark: our Skyline algorithm from the previous lecture implicitly uses Mergesort to sort all endpoints of the rectangles.

Quicksort

- How it works:
  - choose one element to be the *pivot/ splitter,* called *p*
  - put all the elements < *p*, and those > *p* in two different subsets
  - recursively sort the two subsets and concatenate putting *p* in between
- *In place*
- Conquer phase trivial
- Implementation of Divide makes quick sort - quick


54 26 93 17 77 31 44 55 20
54 will be the first pivot value
leftmark and rightmark will converge on split point
leftmark
rightmark
26<54 move to right
93>54 stop
now rightmark
20<54 stop
54 26 20 17 77 31 44 55 93
exchange 20 and 93
now continue moving leftmark and rightmark
77>54 stop
44<54 stop
exchange 77 and 44
54 26 20 17 44 31 77 55 93
77>54 stop
31<54 stop
rightmark<leftmark
split point found


31 26 20 17 44 54 77 55 93
54 is in place
<54
>54
31 26 20 17 44
quicksort left half
77 55 93
quicksort right half

Time Complexity for Quicksort

- **Worst case:** the splitter is always the minimum or maximum element of the set, $O(n^2)$ is needed.

- Only **careful selection** of the splitter can guarantee the better bound.
- If the splitters would exactly halve the sets on every recursion level, we have our standard recursion:
  - $T(n) = 2\,T(n/2) + O(n)$
  - With solution: $T(n) = O(n \log n)$

Ideal Splitter for Quicksort

- **Rank of an element:** the position of this element if the set were already sorted.
- **Median:** Element with rank **n/2**

- **Computing Median?**
  - Sort and read off the element of rank n/2
  - **Stupid idea…** sorting is the actual problem for which we need to find out Median.
- **A splitter is selected at random!**
  - the worst case (rank nearly 1 or n) is very unlikely.
  - The splitters will mostly have ranks in the middle.
    - ➢ reasonably balanced partitions in two sets.
    - ➢ **O(n log n) time is needed on expectation.**
- In practice, chose three random elements and take their median as the splitter.

Center of a Point Set on the Line

Given: n points $x_1, \ldots, x_n$ on the real line.

Goal: Compute a point x so that the sum of distances to all given points $\sum_{i=1}^{n} |x - x_i|$ is minimized.

Distance: Walking or driving distance along the street, not the Euclidean distance.

Median of the given coordinates, not the **average**.

Selection and Median Finding

- **Given:** A set of n elements, where an order relation is defined, and an integer k.
- **Goal:** Output the element of rank k, that is, the kth smallest element.
- **Median:** Special case in Selection problem, **k := n/2**
  - often better suited as a **“typical"** value than the average, because it is robust against outliers.
- Wealth in a population
  - **Mean vs. Median**

Algorithm for Selection and Median Finding

- Choose: a random splitter s and compare all elements to *s* in *O(n)* time to get *rank r* of *s*.
- Decide:
  - If $r > k$ then throw out *s* and all elements *larger than s*. REPEAT
  - If $r < k$ then throw out s and all elements *smaller than s*, and set k := k-r REPEAT
  - **If r = k then return s.** STOP
- Time Complexity
  - Given the splitters are always in the middle: $T(n) = T(n/2) + O(n)$
  - ➢ ***$T(n) = aT(n/b) + cn^k$* , and for a < b^k it gives *$O(n^k)$***
  - ➢ We have **a=1, b=2** and **k= 1,** therefore we get: ***O(n)***
  - ➢ ***O(n) is expected time, worst case could still be $O(n^2)$***
  - ➢ **Fast Algorithm:** Intuition is that Selection needs much less information than Sorting.

Algorithm for Selection and Median Finding

- A deterministic divide-and-conquer, with ***O(n)*** time exists
  - Complicated
  - More importantly, the hidden constant in ***O(n)*** is large
  - Practically, random splitter algorithm is better

Information Flow and Optimal Time Bounds

- One of our primary goals is to make algorithms as fast as possible.
  - ➢How good are our time bounds for sorting and searching algorithms?
- **Searching:**
  - Find a specific element in an ordered set of size n
  - Comparisons counted as the elementary operations
- **Binary Search:** $\log_2 n$ comparisons of elements
- **Claim:** No other algorithm with comparisons as elementary operations can have a better worst-case bound.
  - ➢Claim holds due to the **information-theoretic** argument

Information Flow and Optimal Time Bounds

- **Binary Search:** $\log_2 n$ comparisons of elements
- **Claim:** No other algorithm with comparisons as elementary operations can have a better worst-case bound.
  - ➢ Claim holds due to the **information-theoretic** argument
  - How much information do we gain from our elementary operation?
    - ➢ Binary Answer ("smaller" or "larger"), splitting the set of possible results in two subsets for which either of the answers is true.
    - ➢ worst case: the answer is true for the larger subset, always
    - ➢ candidate solutions are reduced by a factor at most 2
    - ➢ n possible solutions in the beginning, any algorithm needs at least $\log_2 n$ comparisons in the worst case.
  - such arguments are used to define the lower bound on the execution of a computation based on the rate at which information can be accumulated.

Information Flow and Optimal Time Bounds

- **Sorting:** We have $O(n \log_2 n)$ algorithms.
- **Claim:** No other algorithm with comparisons as elementary operations can have a better worst-case bound.
  - The n elements can be ordered in n! possible ways, and only one of them is the correct order
  - Claim holds due to a similar reasoning as for Searching
    - ➢ Any sorting algorithm can be forced to use $\log_2 n!$ comparisons
    - ➢ Calculation shows that $\log_2 n!$ is $n \log_2 n$ subject to a constant factor

$$\log_2 n! = \sum_{k=1}^{n} \log_2 k \geq (n/2) \log_2(n/2).$$

Information Flow and Optimal Time Bounds

- **Selection Problem:** O(n)
- **Reasoning for Searching does not apply here**
  - O ($\log_2$ n) would be a very poor lower bound
- **O(n) is optimal**
  - No order known before hand, ALL the n elements needs to be read
  - Every change in the instance can change the result

Information Flow and Optimal Time Bounds

- **Faster Algorithms for special cases:**
- **Bucket Sort: O(m + n)**
  - ➢ n elements come from a fixed range of m different numbers.
- **O(n) sorting in lexicographic order**
  - Words defined over a fixed alphabet
  - Total length of the given words: n
- Do these two results contradict?
  - **NO!**
  - **?**
  - **Because…**