Sorting and Computational Complexity

Dr. Axel Kohlmeyer

Assistant Dean for High-Performance Computing
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu

Master in High Performance Computing

Steps to Improve Performance

- Use faster Hardware (clock rate, cache, type)
- Write more efficient code (fewer instructions, better CPU utilization, faster instructions)
- Vectorize (explicitly or through compiler)
- Parallelize (MPI, OpenMP, CUDA/OpenCL)
- Use a better algorithm (has the most potential!) Question: How do we compare algorithms?
  - Computational complexity (→ Big O notation)
  - Memory use, number and cost of operations

Big O Notation

- For any algorithm the "cost" of it is the sum of the "cost" of multiple parts/steps in it: $f(x)=\sum_i f_i(x)$
- The "cost" for each part is a positive function $f_i(x)$ of the number of items x being worked on
- Thus we can write that there is a function g(x) for which we can define:

$$f(x)=O(g(x))\, for\, x \rightarrow \infty$$
$$f(x)\leq M\, g(x)\, for\, x\geq x_0$$

with M being a positive number

Typical Scaling Orders

- O(1) – constant time
- O(log n) – logarithmic time
- O(n) – linear time
- O(n log n) – quasilinear time
- O(n^2) – quadratic time
- O(n^c) – exponential time
- O(n!) - factorial time

Sorting Algorithms

- Sorting: take a sequence of items and order them according to a comparison function in either ascending or descending order
- Comparisons return: smaller, larger, or equal
- Sorting algorithms can be “stable” or “unstable”: when the comparison function returns “equal”, a “stable” algorithm will leave items in place but an “unstable” algorithm may change order
- Sorting may happen “in place” or may need an additional “holding space”

Factors that Impact Sorting Speed

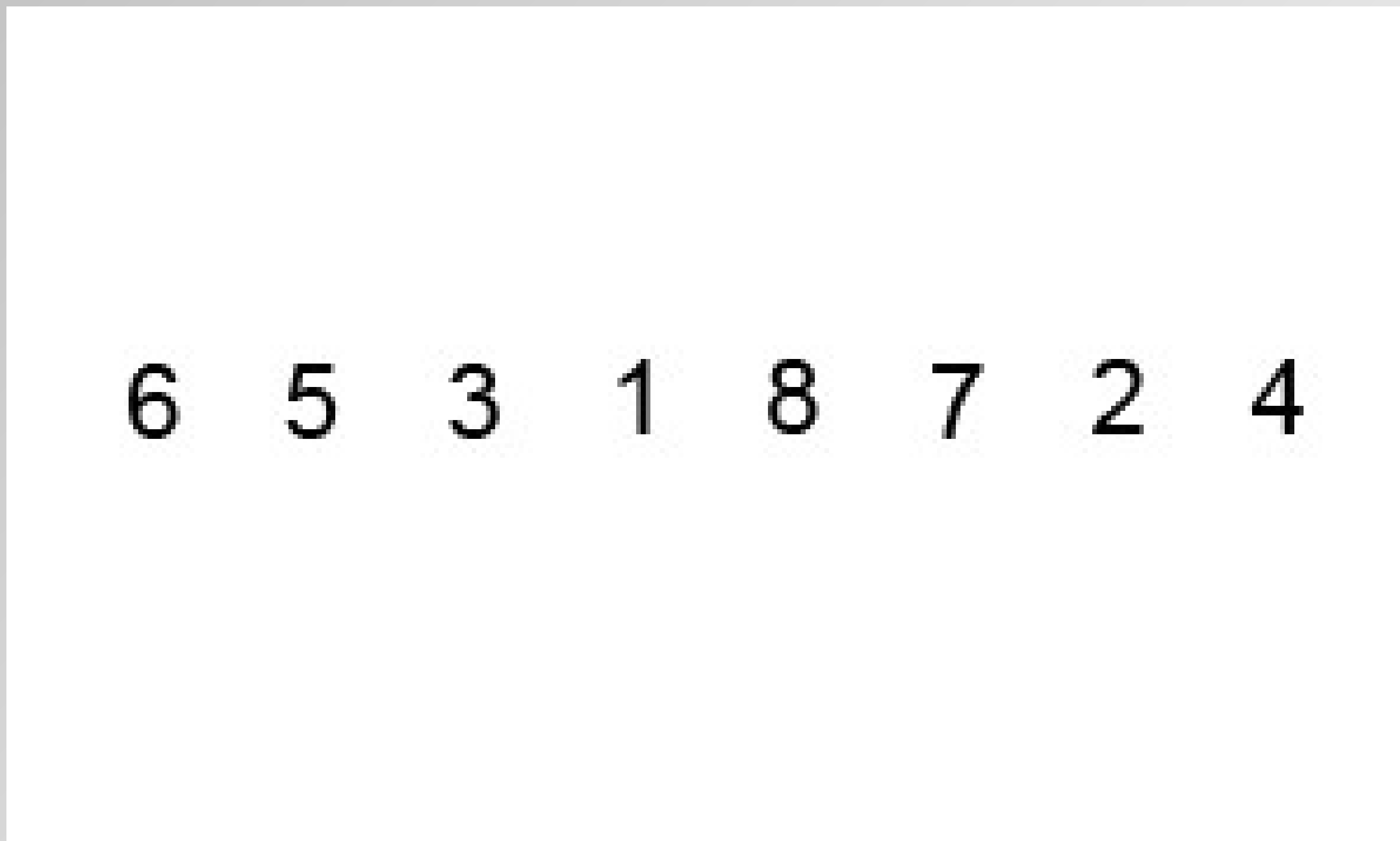
- Cost of comparison operation (e.g. comparing strings versus integers)
- Cost of swapping data (single number versus complex object)
- Number of comparisons needed
- Number of swaps needed

  => There are best case, worst case and typical case scenarios to be considered.
- All of those determine the choice of algorithm

Bubble Sort

- Start with first element and compare to next
- If next element is smaller, then swap
- Move to second element and compare to third
- Continue until last but one element
- After that comparison/swap step the last element is sorted (i.e. the largest in the list)
- Repeat from beginning, but no need to compare with last element. Next run skip 2 last elements
- Optimization: if no swaps needed, list is sorted

Bubble Sort Animation

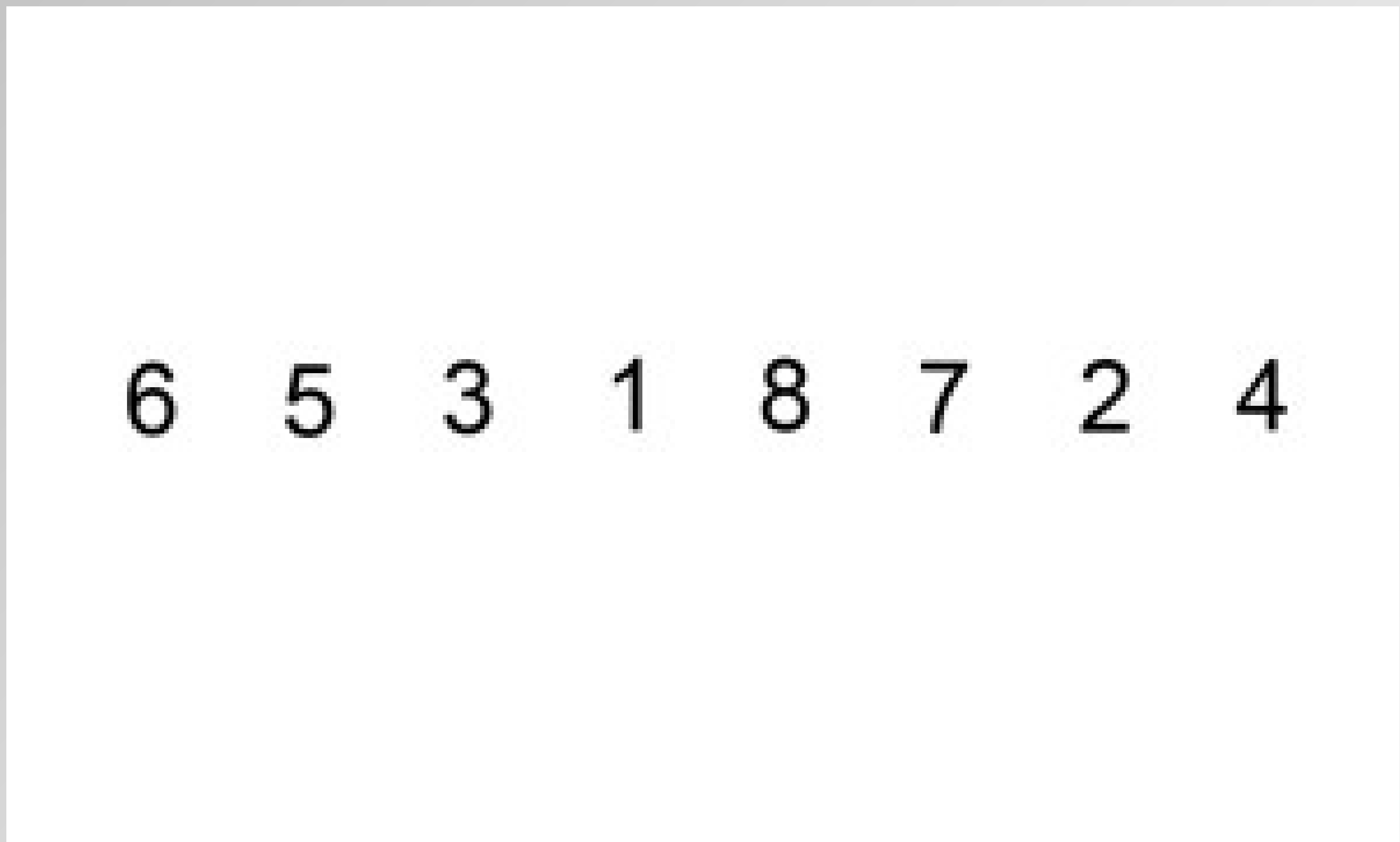


(Image/animation from Wikipedia)

Insertion Sort

- Copy second element to holding space
- Compare holding space with first element
- If 1st element is larger than hold, move 1st to 2nd and copy held value to 1st,otherwise discard
- First two elements are now sorted
- Take 3rd element and compare with 2nd. Move 2nd to 3rd position if 2nd is larger than hold, move 1st to 2nd if larger than hold. Insert hold.
- Basic idea: move if larger, insert if smaller

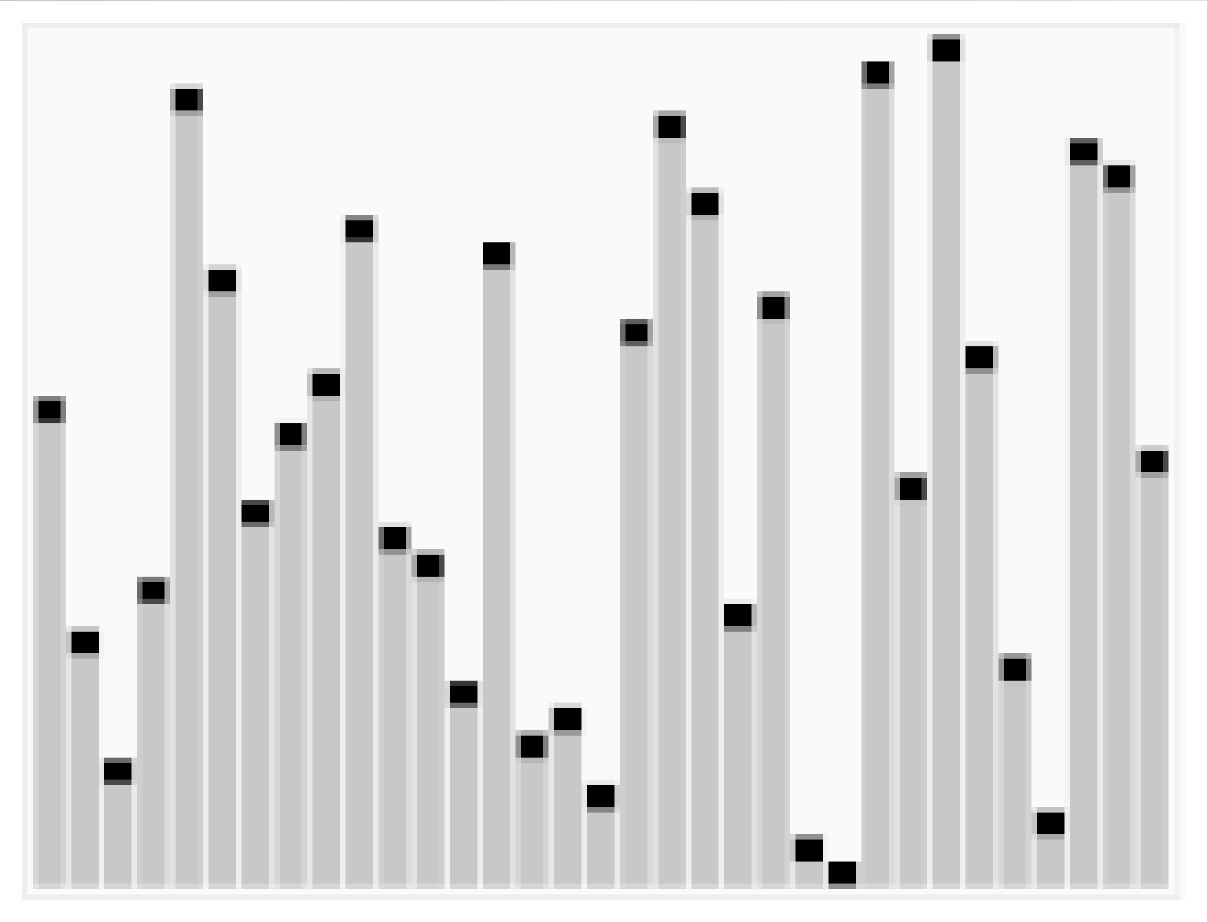
Insertion Sort Animation



(Image/animation from Wikipedia)

Quick Sort

- Pick some element from list (=pivot element)
- Now compare to remaining elements and swap them so that all elements larger than the pivot are to the right and smaller are to the left
- The pivot element is now in its final place
- Now apply the same procedure to the sub-lists to the left and the right of the pivot element
- Repeat until sub-list have 0 or 1 elements and thus are automatically sorted

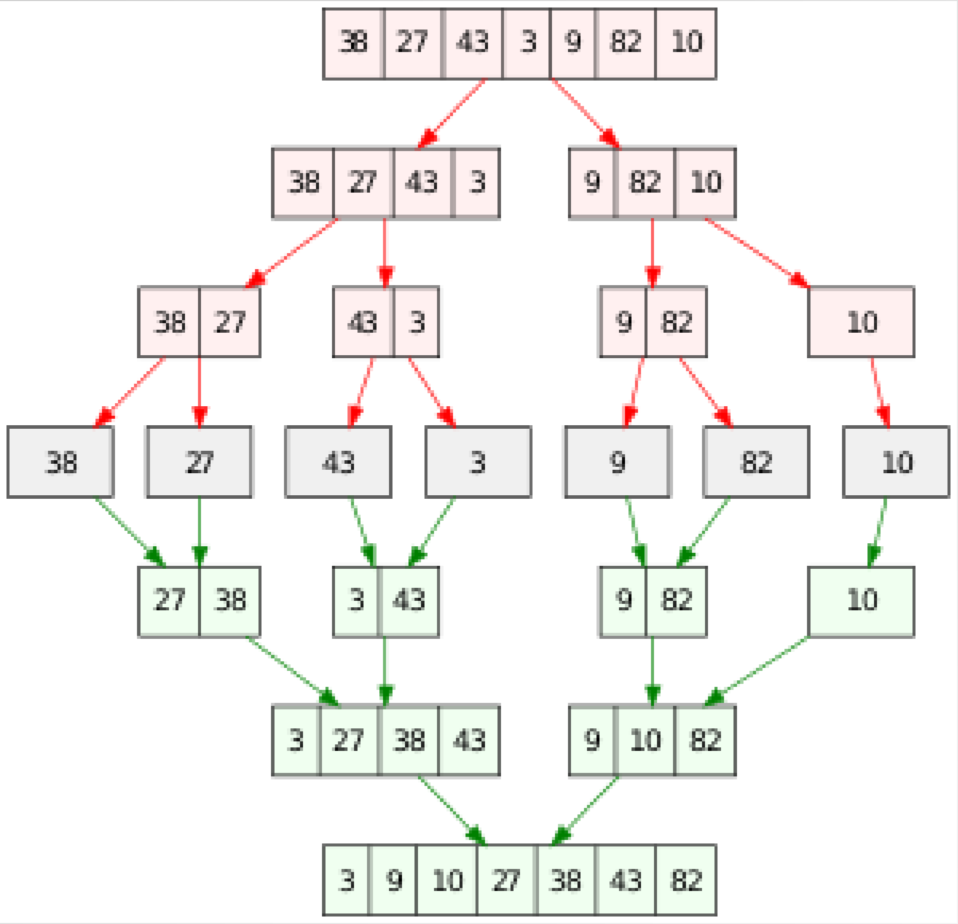


(Image/animation from Wikipedia)

Merge Sort (Top Down)

- Split list in two parts of equal size +/- 1
- Continue until sub-lists are of size 1 or 2
  size 1 is sorted, size 2 do comparison and swap
- Now take sub-lists and merge into new list:
  - Compare head of each list and select smaller
  - Copy to new list, move head to next element
  - Compare heads of sub-lists again and copy
    until one of the two sub-lists is out of elements;
    add remaining elements of other list to merged list
- Now merge larger lists until back at top

Merge Sort Schema (Top Down)

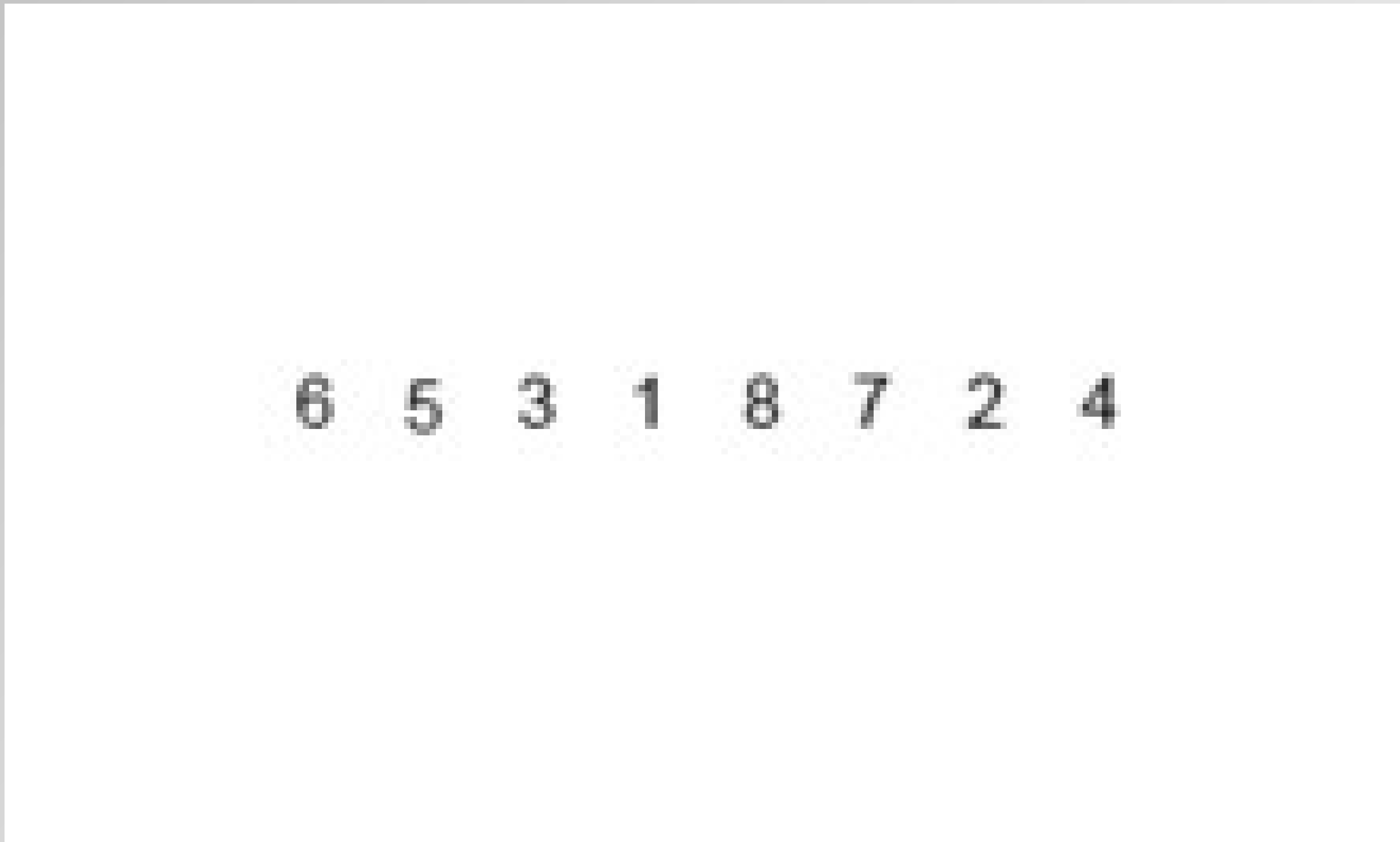


(Image/animation from Wikipedia)

Merge Sort (Bottom Up)

- Compare items in pairs: 1 and 2, 3 and 4, 5 and 6 etc. and swap if not in correct order
- Now apply merge procedure as in top down version to two neighboring sub-lists (the rightmost list may be shorter or of length 0)
- Continue doubling the size of the sub-lists and merging them until merging the full list
- Merge sort needs a holding space of the size of the entire list to merge into
- Top down version simple to implement with recursion, but then it needs 1 copy of the list per recursion level

Merge Sort Animation (Bottom Up)



(Image/animation from Wikipedia)

Properties of Sort Algorithms

Algorithm	Best Case	Worst Case	Typical	Memory Usage
Bubble Sort	O(n) comparisons O(1) swaps	$O(n^2)$ comparisons $O(n^2)$ swaps	$O(n^2)$ comparisons $O(n^2)$ swaps	O(n) + O(1)
Insertion Sort	O(n) comparisons O(1) swaps	$O(n^2)$ comparisons $O(n^2)$ swaps	$O(n^2)$ comparisons $O(n^2)$ swaps	O(n) + O(1)
Quick Sort	O(n log n)	$O(n^2)$	O(n log n)	2*O(n)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	2*O(n) / c*O(n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(n) + O(1)
Shell Sort	O(n log n)	$O(n^2)$	O(n log n) ↔ O(n2)	O(n) + O(1)

Stable Algorithms: Merge sort, Insertion sort, Bubble sort

Merge sort is easily parallelizable: Operate on n parallel sub-lists
Final merges on n, n/2, n/4, n/8 etc. parallel tasks

Further Optimizations

- If copying or moving data around would be expensive (large/complex objects):
  - Create and sort a list of pointers to the objects
  - Create and sort a list of indices instead of the data. This would also allow to have differently ordered lists in case there would be multiple properties that could be used for comparing.
- Since well scaling algorithms have more overhead, small chunks of the data could be pre-sorted with insertion/bubble sort and then further sorted with merge sort → hybrid sort

Fun with Sorts

Check out:

https://www.youtube.com/watch?v=kPRA0W1kECg

Sorting and Computational Complexity

Dr. Axel Kohlmeyer

Assistant Dean for High-Performance Computing
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu

MHPC
Master in High Performance Computing