Week 4 - Sorting Algorithms

Brenden West - 2023

Contents

Reading & Videos

- LaFore, Chapters 3 & 7
- https://www.coursera.org/learn/algorithms-part1/home/week/2 - Elementary Sorts
- https://www.coursera.org/learn/algorithms-part1/home/week/3
- https://algs4.cs.princeton.edu/20sorting/ (review)
- https://www.geeksforgeeks.org/sorting-algorithms/ (review)

Learning Outcomes

- Elementary sorting algorithms
- Comparable interfaces
- Advanced sorting algorithms
- Performance considerations for sorting

Key Concepts

- **comparable** - data to be sorted must be comparable - e.g. implement Java's *Comparable* interface
- **recursion** - The process in which a function calls itself directly or indirectly. Can achieve results similar to loop, but with different memory usage.
- **cost model** - number of compares and exchanges (or array accesses).
- **best, worst, average cases** - a sorting algorithm's performance can vary in different cases - e.g. whether input data is already sorted.
- **memory usage** - in-place sorting algorithms use constant extra space by modifying elements in the existing array or list. Others use extra memory to hold another copy of the array to be sorted.

Key Concepts, cont.

- **stability** - output from a stable sorting algorithm has objects with equal values in the same order as in the original input. https://www.geeksforgeeks.org/stability-in-sorting-algorithms/
- **divide-and-conquer** - is an algorithmic paradigm that solves a problem using following three steps:
  - Divide: Break the given problem into subproblems of same type.
  - Conquer: Recursively solve these subproblems
  - Combine: Appropriately combine the answers

Comparable Data

- Data items being sorted must be comparable
- In Java, this means Objects to sort must support the **Comparable** interface and implement a **compareTo()** method
- Some built-in comparable types - Integer, Double, String, Date
- Comparable methods must:
  - Support **total order** (antisymmetry, transivity, totality)
  - Return a negative integer, zero, or a positive integer
  - Throw an exception for incompatible types

Key Sorting Algorithms

Elementary

- Selection sort
- Insertion sort
- Shell sort

Advanced

- Mergesort
- Quicksort

Selection Sort

Select the smallest item in the array, and exchange it with the first entry. Then, find the next smallest item and exchange it with the second entry. Continue in this way until the entire array is sorted.

- average case: uses ~$N^2/2$ compares and N exchanges
- Running time insensitive to input. Quadratic time, even if input is sorted

Insertion Sort

Consider items one at a time, swapping the current item with larger items to the left.

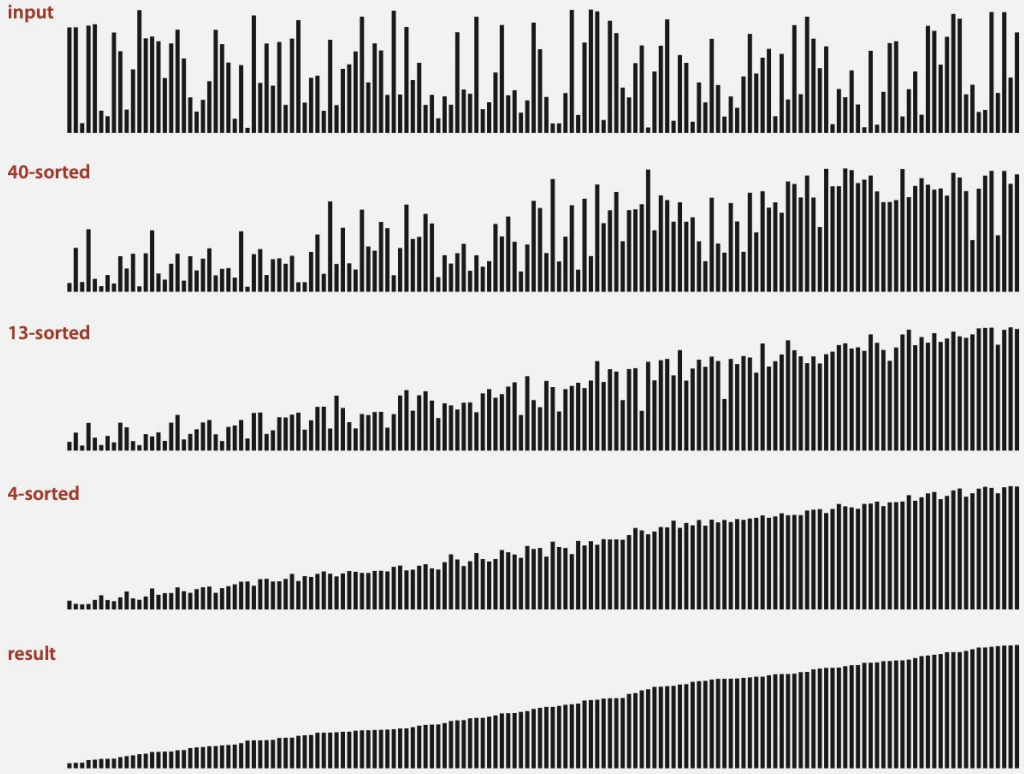
- **average case**: for randomly ordered arrays, uses ~$N^2/4$ compares and ~$N^2/4$ exchanges.
- **worst case**: for array sorted in descending order. ~ $N^2/2$ compares and ~ $N^2/2$ exchanges
- **best case**: for array sorted in ascending order. N-1 compares and 0 exchanges
- works well for certain types of non-random arrays. E.g. for partially sorted arrays insertion sort runs in linear time

Shell Sort

A simple extension of insertion sort that gains speed by exchanging entries that are far apart (*h-sorting*).

- produces partially sorted arrays that can be efficiently sorted, eventually by insertion sort.
- h-sorting is repeated with smaller values of *h* until the array is fully sorted.
- Optimal increment sequence is 3x + 1
- **Worst case**: number of compares used with 3x+1 increments is ~ $O(N^{3/2})$

Shellsort: visual trace



Mergesort

- A simple recursive method to sort an array by:
  - dividing it into two halves,
  - sorting each half (recursively),
  - merging the results
- **Worst case**: sorts an array of N items in time proportional to N lg N
- uses extra space proportional to N
- Mergesort improvements:
  - Use Insertion Sort for sub-arrays with < 7 items
  - Stop if already sorted

Mergesort: trace

	a[]														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 6, 6, 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 5, 7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, aux, 0, 3, 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, aux, 8, 8, 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, aux, 10, 10, 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, aux, 8, 9, 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux, 12, 12, 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux, 14, 14, 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, aux, 12, 13, 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, aux, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, aux, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

lo

hi

result after recursive call

Quicksort

- Works well for a variety of different kinds of input data, and is substantially faster than any other sorting method in **typical** applications.
- Quicksort is a divide-and-conquer method for sorting. It works by partitioning an array into two parts, then sorting the parts independently.
- It uses in-place swaps and requires time proportional to NlogN on the average to sort N items
- Quicksort uses ~2NlnN compares (and one-sixth that many exchanges) on the average to sort an array of length N with distinct keys.
- Quicksort uses ~$N^2/2$ compares in the worst case, but random shuffling protects against this case.

Comparison of Sorting Algorithms

	Order of growth					
Algorithm	**Average**	**Best**	**Worst**	**Space**	**Stable ?**	**Notes**
Selection Sort	O(n^2)	O(n^2)	O(n^2)	Constant	No	Even a perfectly sorted input requires scanning the entire array
Insertion Sort	O(n^2)	O(n)	O(n^2)	O(n)	Yes	In the best case (already sorted), every insert requires constant time
Heap Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Constant	No	By using input array as storage for the heap, it is possible to achieve constant space
Merge Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Depends	Yes	On arrays, merge sort requires O(n) space; on linked lists, merge sort requires constant space
Quicksort	O(n*log(n))	O(n*log(n))	O(n^2)	O(log(n)) (optimized)	No	Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array.