Internal sorting

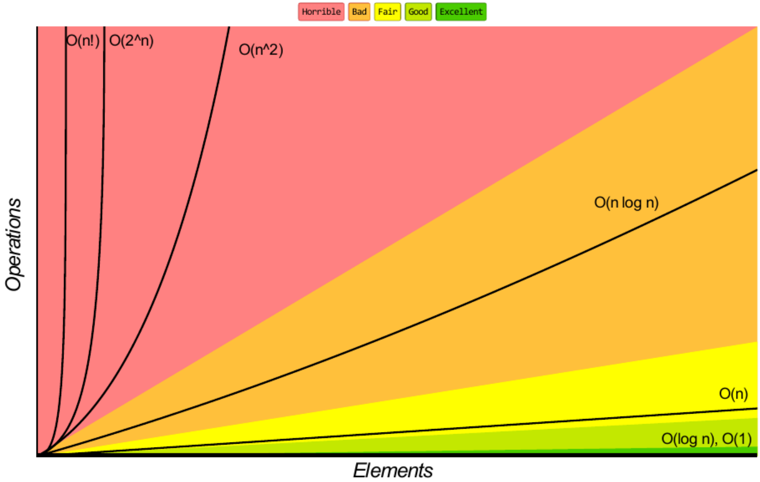
Comp Sci 1575 Data Structures

Missouri S&T | Computer Science

Debugging

Outline

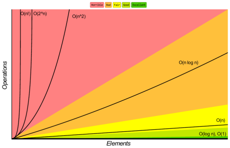
Recall: Data structures



Recall: Data structures

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Color key:



Today: Array sorting algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Which is best?

- Running time for many sorting algorithms depends on specifics of the input values
- Number of records, the size of the keys and the records, the allowable range of the key values, and the amount by which the input records are "out of order" can all greatly affect relative running time
- Number of swap operations versus number of comparisons matters
- Keys having widely varying length (such as sorting a sequence of variable length strings) will benefit from special-purpose sorting
- Small number of records be sorted, but that the sort be performed frequently, e.g., repeatedly sort groups of five numbers; constants in runtime equations that are usually ignored now become crucial.
- Some situations require that a sorting algorithm use as little memory as possible.

Outline

What is sorting?

- Given a set of records $r_1, \ r_2, \ ..., \ r_n$ with key values $k_1, \ k_2, \ ..., \ k_n$, the **sorting problem** is to arrange the records into any order s such that records $r_{s1}, \ r_{s2}, \ ..., \ r_{sn}$ have keys obeying the property $k_{s1} \leq k_{s2} \leq ... \leq k_{sn}$.
- In other words, the **sorting problem** is to arrange a set of records so that the values of their key fields are in non-decreasing order.
- Two or more records can have the same key value.
- A sorting algorithm is said to be stable if it does not change the relative ordering of records with identical key values before and after sort.

What is internal sort?

- In **internal sorting** all the data to sort is stored in memory at all times while sorting is in progress.
- In **external sorting** data is stored outside memory (e.g., on disk) and only loaded into memory in small chunks. External sorting is usually applied in cases when data can't fit into memory entirely.

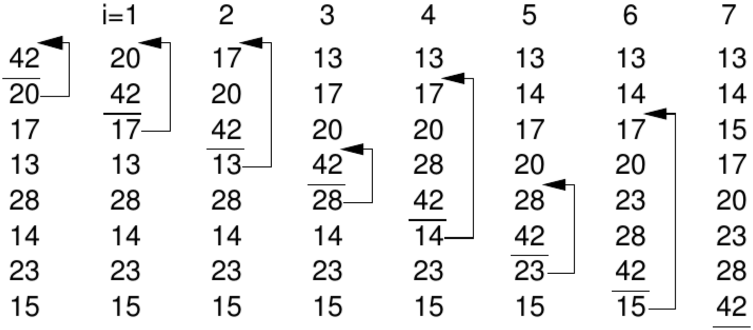
Outline

Three $\Theta(n^2)$ sorting algorithms

- Slow, but **some** of these slow sorts have benefits in other situations (context matters!)
- We're actually skipping selection and bubble, the latter of which has absolutely no redeeming qualities.

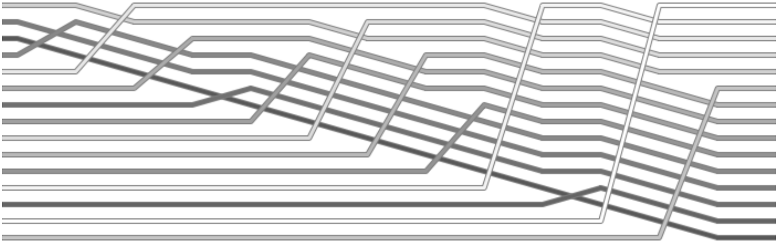
Outline

Insertion sort



- Going bottom to top
- Order 1st two, then move 3rd as far up as fits, then move 4th as far up as fits, etc

Insertion sort



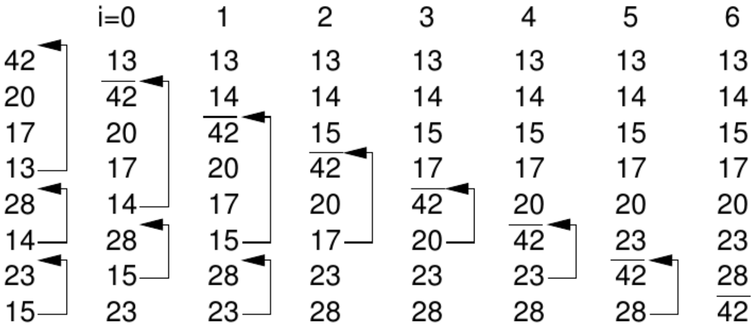
- Going top to bottom
- Order 1st two, then move 3rd as far up as fits, then move 4th as far up as fits, etc

Check out code

and GIFS

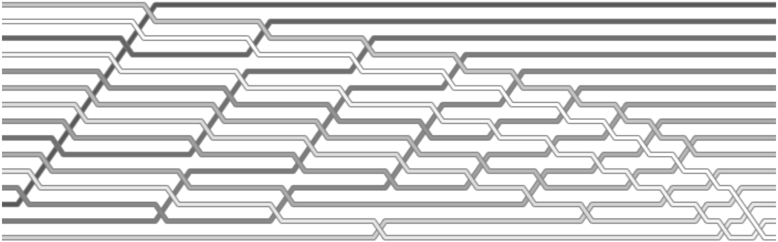
Outline

Bubble sort



- Going bottom to top
- Start at one end, comparing pairwise toward the end of the list, swapping any out-of-order pair, and stopping at incrementally earlier locations. Can start at either end.

Bubble sort



- Going bottom to top
- Start at one end, comparing pairwise toward the end of the list, swapping any given pair, and stopping at incrementally earlier locations

Check out code

and GIFS

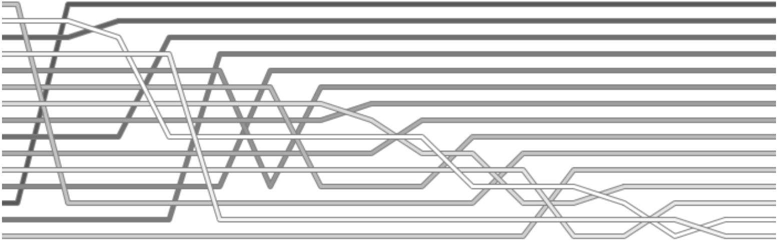
Outline

Selection sort

	i=0	1	2	3	4	5	6
42	13	13	13	13	13	13	13
20	20	14	14	14	14	14	14
17	17	17	15	15	15	15	15
13	42	42	42	17	17	17	17
28	28	28	28	28	20	20	20
14	14	20	20	20	28	23	23
23	23	23	23	23	23	28	28
15	15	15	17	42	42	42	42

- Going top to bottom
- The ith pass of Selection Sort swaps the ith smallest key in the array (with some random position) with the position i. Can start at either end.

Selection sort



- Going top to bottom
- The ith pass of Selection Sort swaps the ith smallest key in the array (with some random position) with the position i. Can start at either end.

Check out code

and GIFS

Outline

Theoretical comparison

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

- What does each do better?
- We skipped bubble and selection because they are slow, and often only included as historical notes

Empirical implementation comparison

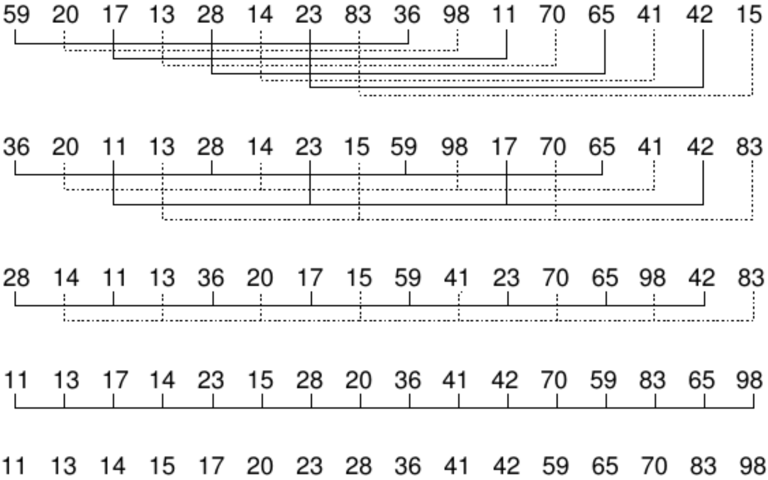
Sort	10	100	1K	10K	100K	1M	Up	Down
Insertion	.00023	.007	0.66	64.98	7381.0	674420	0.04	129.05
Bubble	.00035	.020	2.25	277.94	27691.0	2820680	70.64	108.69
Selection	.00039	.012	0.69	72.47	7356.0	780000	69.76	69.58

- What does each do better?
- We'll re-use insertion sort's best case to speed up other sort algorithms

Outline

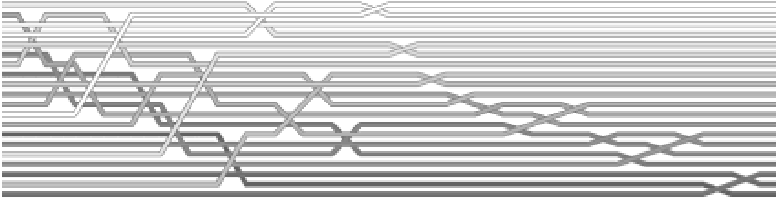
Outline

Shellsort (a.k.a. diminishing increment sort)



First pass insertion sorts 8 sublists of size 2 and increment 8. Second pass sorts 4 sublists of size 4 and increment 4. Third pass sorts 2 sublists of size 8 and increment 2. Fourth pass sorts 1 list of size 16 and increment 1 (a regular Insertion Sort).

Shellsort

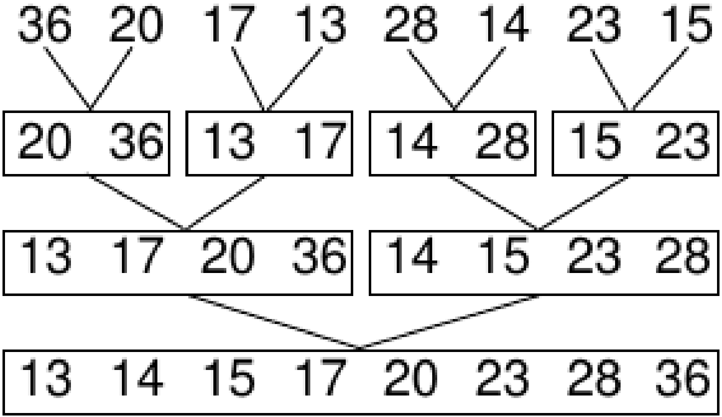


shellsort

Starts by sorting pairs of elements far apart from each other, then progressively reduces the gap between elements to be compared. Shellsort is a generalization of insertion sort that allows the exchange of items that are far apart. The idea is to arrange the list of elements so that, starting anywhere, considering every h^{th} element gives a sorted list. Such a list is said to be h-sorted. Equivalently, it can be thought of as h interleaved lists, each individually sorted. Beginning with large values of h, this rearrangement allows elements to move long distances in the original list, reducing large amounts of disorder quickly, and leaving less work for smaller h-sort steps to do. If the list is then k-sorted for some smaller integer k, then the list remains h-sorted. Following this idea for a decreasing sequence of h values ending in 1 is guaranteed to leave a sorted list in the end.

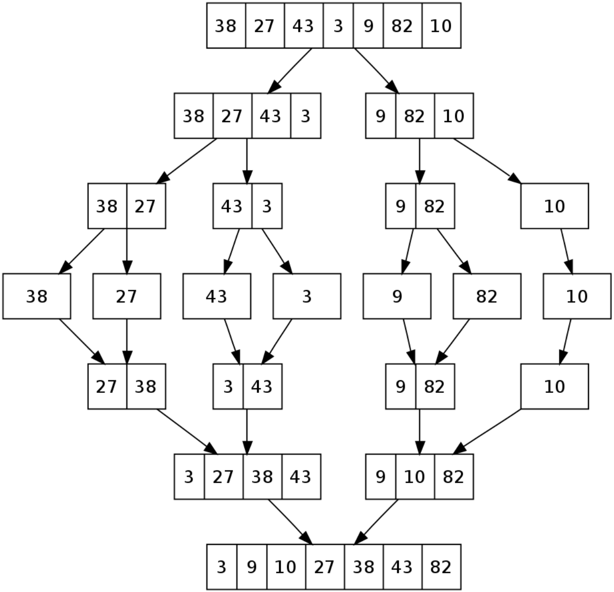
Outline

Mergesort



- Start with pairs, sort them, merge into larger pairs, then merge again.
- Merge happens by making a new list, and repeatedly appending from the front of each sublist the smallest of

Mergesort

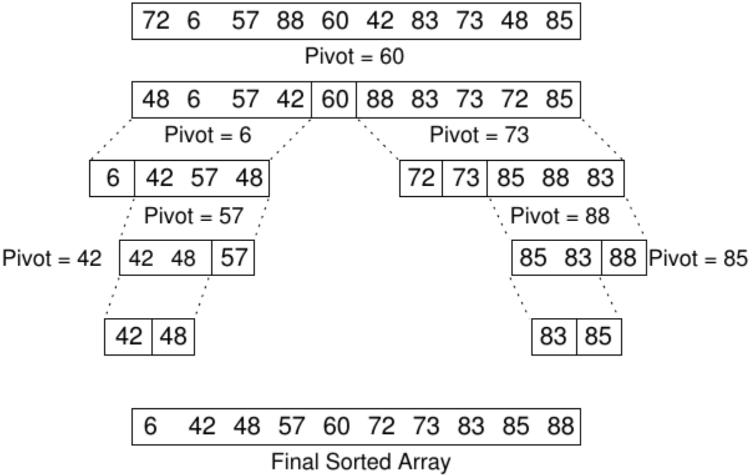

38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Mergesort

- Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
- Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

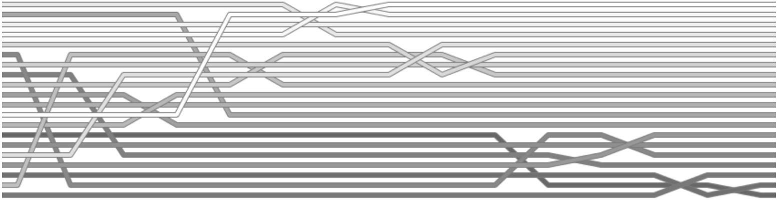
Outline

Quicksort



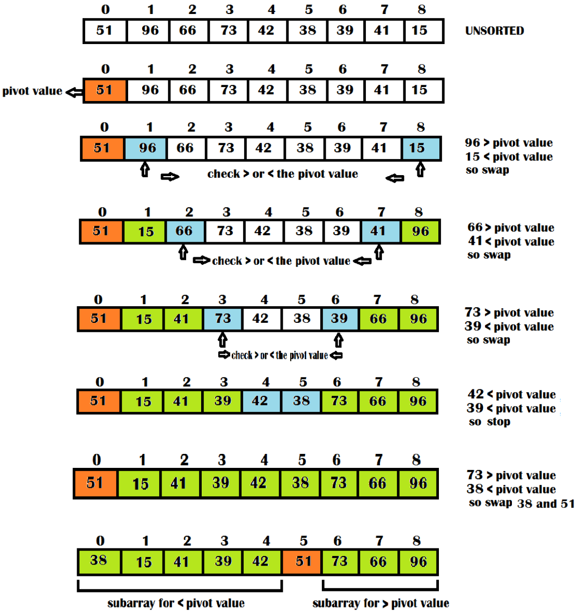
- Pick random pivot value, move smaller numbers left, larger numbers right, and recursively do the same for each left/right subdivision

Quicksort



- Pick an element, called a pivot, from the array.
- Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Quicksort: partition (multiple variations exist)

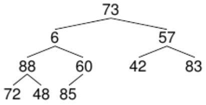


Outline

Heapsort

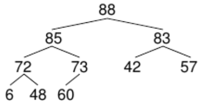
Original Numbers

73	6	57	88	60	42	83	72	48	85



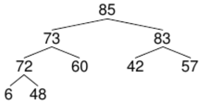
Build Heap

88	85	83	72	73	42	57	6	48	60



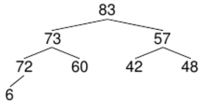
Remove 88

85	73	83	72	60	42	57	6	48	88



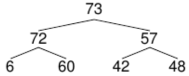
Remove 85

83	73	57	72	60	42	48	6	85	88

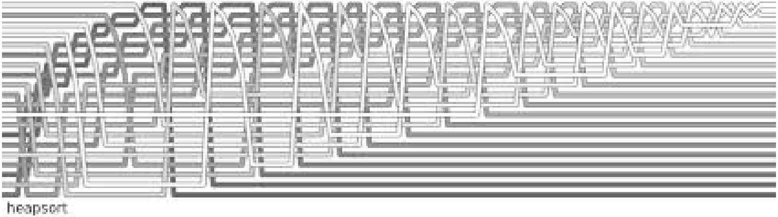


Remove 83

73	72	57	6	60	42	48	83	85	88



Heapsort



- Build heap, dequeue the max repeatedly

Outline

Comparisons

Sort	10	100	1K	10K	100K	1M	Up	Down
Insertion	.00023	.007	0.66	64.98	7381.0	674420	0.04	129.05
Bubble	.00035	.020	2.25	277.94	27691.0	2820680	70.64	108.69
Selection	.00039	.012	0.69	72.47	7356.0	780000	69.76	69.58
Shell	.00034	.008	0.14	1.99	30.2	554	0.44	0.79
Shell/O	.00034	.008	0.12	1.91	29.0	530	0.36	0.64
Merge	.00050	.010	0.12	1.61	19.3	219	0.83	0.79
Merge/O	.00024	.007	0.10	1.31	17.2	197	0.47	0.66
Quick	.00048	.008	0.11	1.37	15.7	162	0.37	0.40
Quick/O	.00031	.006	0.09	1.14	13.6	143	0.32	0.36
Heap	.00050	.011	0.16	2.08	26.7	391	1.57	1.56
Heap/O	.00033	.007	0.11	1.61	20.8	334	1.01	1.04
Radix/4	.00838	.081	0.79	7.99	79.9	808	7.97	7.97
Radix/8	.00799	.044	0.40	3.99	40.0	404	4.00	3.99

- Context matters!
- Some sorts are good only in limited contex, some are good all-round, and some are useless altogether (bubble sort)