Comparison Sorts

CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Aaron Johnston	Ethan Knutson	Nathan Lipiarski
Amanda Park	Farrell Fileas	Sam Long
Anish Velagapudi	Howard Xiao	Yifan Bai
Brian Chan	Jade Watkins	Yuma Tou
Elena Spasova	Lea Quan	

Announcements

- HW8 (Seam Carving) has been released!
  - 🚨 NOTE 🚨: We are NOT offering late days for this homework. If you think you'll need extra time, pretend it's due on Tuesday instead of Friday

Lecture Outline

- **Intro to Sorting**
- SelectionSort and Naïve HeapSort
- In-place HeapSort
- MergeSort
- InsertionSort

Our Major Focus for Several Lectures: Sorting

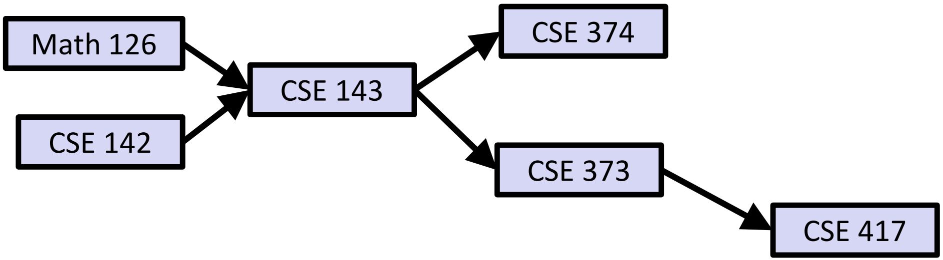
- For the next 3 lectures we'll discuss the sorting problem
  - Informally: Given items, put them in order

- Sorting is a useful task in its own right! Examples:
  - Equivalent items are adjacent, allowing rapid duplicate finding
  - Items are in increasing order, allowing binary search
  - Can be converted into *balanced* data structures (e.g. BSTs, k-d Trees)

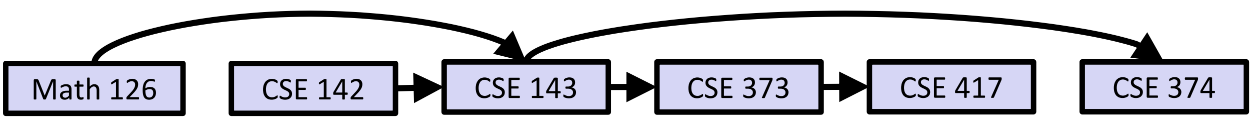
- But it's also an interesting case study for how to approach computational problems
  - We'll use data structures and algorithms we've already studied

Not Everything Can Be Sorted

- Our course prerequisite chart:



- Possible ordering:



Sorting Definitions: Knuth's TAOCP

- An **ordering relation** < for keys a, b, and c has the following properties:
  - Law of Trichotomy: Exactly one of a < b, a = b, b < a is true
  - Law of Transitivity: If a < b, and b < c, then a < c

- An ordering relation with these properties is also known as a **total order**

- A **sort** is a permutation (re-arrangement) of a sequence of elements that puts the keys into non-decreasing order, relative to the ordering relation
  - $x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_N$

Sorting Definition: An Alternate Viewpoint

Gabriel Cramer

- An ***inversion*** is a pair of elements that are out of order with respect to <

  0 1 1 2 3 4 8 6 9 5 7

  8-6, 8-5, 8-7, 6-5, 9-5, 9-7

  (6 inversions out of 55 max)

- Another way to state the goal of sorting:
  - Given a sequence of elements with Z inversions, perform a sequence of operations that reduces inversions to 0

- Max number of inversions is N*(N -1)/2
  - A “partially sorted” array has O(N) inversions

Sorting: Performance Definitions

- Runtime performance is sometimes called the **time complexity**
  - Example: Dijkstra's has time complexity O(E log V).

- *Extra* memory usage is sometimes called the **space complexity**
  - Dijkstra's has space complexity Θ(V)
  - The input graph takes up space Θ(V+E), but we don't count this as part of the space complexity since the graph itself already exists and is an input to Dijkstra's

Sorting: Stability

- A sort is **stable** if the relative order of *equivalent* keys is maintained after sorting
  - Examples:
    - Email is originally sorted by date, but you re-sort by sender
    - T-shirts originally sorted by size, but you re-sort by color

sorting by name even though we also have date info

Anita	Basia	Anita	Duska	Esteban	Duska	Caris
2010	*2018*	*2016*	*2020*	*2014*	*2015*	*2019*

Anita	Anita	Basia	Caris	Duska	Duska	Esteban
2010	*2016*	*2018*	*2019*	*2020*	*2015*	*2014*

  - Stability and Equivalency only matter for complex types

sorting by name, and there are no other fields

Anita	Basia	Anita	Duska	Esteban	Duska	Caris

Anita	Anita	Basia	Caris	Duska	Duska	Esteban

?

Lecture Outline

- Intro to Sorting
- **SelectionSort and Naïve HeapSort**
- In-place HeapSort
- MergeSort
- InsertionSort

Selection Sort Review

- We've seen this already
  - Find smallest item in the unsorted region
  - Swap this item to the end of the sorted region
  - Repeat until the unsorted region is empty / sorted region is full
  - Demo: https://goo.gl/g14Cit
- Performance Characteristics:
  - Time: $\Theta(N^2)$
  - Space: $\Theta(1)$ (we can reuse the input array)
  - Stable: No 2a, 2b, 1 $\Longrightarrow$ 1, 2b, 2a
- Inefficient! Finding the minimum element requires N work

If only we had an algorithm or data structure that found the minimum quickly …

Naïve HeapSort

- Instead of rescanning entire array looking for minimum, maintain a heap so that getting the minimum is fast!
- Demo: https://goo.gl/EZWwSJ

```
Naïve HeapSorting N items:
  Insert all items into
    a min heap
  Discard input array
  Create output array

  Repeat N times:
    Delete smallest item from
      the min heap
    Put that item at the end
      of the sorted region
```

Poll Everywhere

pollev.com/uwcse373

Time=red Space=blue

- What is the time and space complexity for naïve HeapSort?

A. Θ(log N) / Θ(1)
B. Θ(N) / Θ(N)
C. Θ(N) / Θ(N^2)
D. Θ(N log N) / Θ(N)
E. Θ(N log N) / Θ(N^2)
F. Θ(N^2) / Θ(N)
G. I'm not sure …

```
NaïveHeapSort:
  Insert all items into
    a min heap
  Discard input array
  Create output array

  Repeat N times:
    Delete smallest item from
      the min heap
    Put that item at the end
      of the sorted region
```

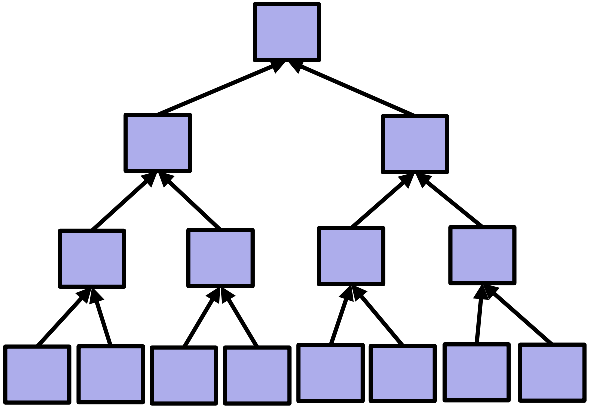
N logN
1
1 ← N
N
logN
1

Lecture Outline

- Intro to Sorting
- SelectionSort and Naïve HeapSort
- **In-place HeapSort**
- MergeSort
- InsertionSort

Remember Floyd!

- buildHeap:
  - Start with full array (representing a binary heap with lots of violations)
  - Call percolateDown() N/2 times
  - Runtime: Θ(N)



- No need to copy input into a heap; just modify the input

- (review lecture 11 if you still have questions)

This "clever implementation" is called Floyd's Algorithm

In-Place HeapSort

- Instead of copying the input array into a heap, reuse and modify the input array as a **MAX** Heap
  - Note: max-heap instead of min-heap lets us reuse input array!
  - When we removeMax(), the heap shrinks by one element; we then reuse the emptied back of array to store our sorted items
- Demo: https://docs.google.com/presentation/d/1SzcQC48OB9agStD0dFRgccU-tyjD6m3esrSC-GLxmNc/present?ueb=true&slide=id.g12a2a1b52f_0_0
- Performance Characteristics:
  - Time: ?? $N + N \log N$ (naive heapsort $N \log N + N \log N$)
  - Space: Θ(1) (we can reuse the input array)
  - Stability: ?? No $1, 2a, 2b \Longrightarrow 1, 2b, 2a$

Lecture Outline

- Intro to Sorting
- SelectionSort and Naïve HeapSort
- In-place HeapSort
- **MergeSort**
- InsertionSort

MergeSort Review

- We've seen this one before as well
  - Split array in half
  - MergeSort each half (steps not shown; this is a recursive algorithm!)
  - Merge the two sorted halves to form the final result

- Demo: https://docs.google.com/presentation/d/1h-gS13kKWSKd_5gt2FPXLYigFY4jf5rBkNFl3qZzRRw/present?ueb=true&slide=id.g463de7561_042

MergeSort Characteristics

- Performance Characteristics:
  - Time: $\Theta(N \log N)$ (see lecture 6 for analysis)
  - Space: $\Theta(N)$ (auxiliary array used for merges)
  - Stable: Yes!

- Note: in-place MergeSort is possible. However, the algorithm is very complicated and runtime performance suffers by a significant constant factor

Lecture Outline

- Intro to Sorting
- SelectionSort and Naïve HeapSort
- In-place HeapSort
- MergeSort
- **InsertionSort**

Naïve InsertionSort

- General strategy:
  - Start with an empty output sequence
  - Add each item from input, inserting into output at correct position
- Demo: http://goo.gl/bVyVCS

Input:	32	15	2	17	19	26	41	17	17
Output:	2	15	17	17	17	19	26	32	41

- Performance Characteristics:
  - Time: $O(N^2)$
  - Space: $\Theta(N)$
  - Stable: Yes!

In-Place InsertionSort

- General strategy for in-place variant:
  - Similar to HeapSort: grow the "output region" as the "input region" shrinks
  - Instead of shift-then-copy into the output array, shift into the output region using pairwise swaps

- Demo: https://docs.google.com/presentation/d/10b9aRqpGJu8pUk8OpfqUIEEm8ou-zmmC7b_BE5wgNg0/present?ueb=true&slide=id.g463de7561_042

- Performance Characteristics:
  - Stable: Yes!

In-Place InsertionSort: Runtime

- How many swaps did we do for:

32	15	2	17	19	26	41	17	17

- Versus:

41	32	26	19	17	17	17	15	2

- Or even: (partially sorted: O(N) inversions)

0	1	1	2	3	4	8	6	9	5	7

Poll Everywhere

pollev.com/uwcse373

- What is the runtime of In-Place InsertionSort?

A. $\Omega(1)$ / $O(N)$

B. $\Omega(N)$ / $O(N)$

C. $\Omega(1)$ / $O(N^2)$

D. $\Omega(N)$ / $O(N^2)$

E. $\Omega(N^2)$ / $O(N^2)$

F. I'm not sure …

In-Place InsertionSort: Runtime

- InsertionSort's lower bound is linear!
  - InsertionSort does one swap per inversion
  - Its runtime is $\Theta(N + K)$, where K is the number of inversions
  - When the number of inversions is small – say, $O(N)$ – InsertionSort has the fastest *asymptotic* runtime

- InsertionSort also has the fastest *empirical* runtime for small arrays (~N < 15)
  - Theoretical analysis beyond scope of the course, but rough idea is that divide-and-conquer algorithms like HeapSort and MergeSort spend too much time dividing
  - The Java implementation of Mergesort does this!

tl;dr (1 of 2)

- Techniques/ideas we've seen today:
  - HeapSort:
    - Use a data structure to help us pick the smallest element
    - Reuse the input array to minimize space complexity
  - MergeSort:
    - Process our input sequentially to maintain stability
    - (Non-sequential swaps are sometimes known as "long-distance jumps")
  - InsertionSort:
    - Reuse the input array to minimize space complexity
    - Process our input sequentially to maintain stability

tl;dr (2 of 2)

	Best-Case Time	Worst-Case Time	Space	Stable?	Notes
SelectionSort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$	No	
In-Place HeapSort	$\Theta(N)$	$\Theta(N \log N)$	$\Theta(1)$	No	Slow in practice
MergeSort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N)$	Yes	Fastest stable sort
In-Place InsertionSort	$\Theta(N)$	$\Theta(N^2)$	$\Theta(1)$	Yes	Best for small or partially-sorted input