Pre-Announcements

Your friend from this class is here to tell you about an event.

- Sutardja senter for Entrepreneurship is holding a hackathon called Cal Innovates.
- April 14th/15th [?]. Open to anybody, CS, econ, business, premed, etc.
- Recruiters from all sorts of companies will be there.
- Prizes will be there, not just electronics gadgets, but also ability to pitch to Skydeck which is a venture capital firm in house at Berkeley.
- Register by this Thursday for raffle eligibility.
- Innovateatcal.com

Announcements

Proj3: It’s out. If you haven’t started yet, you are behind.

CS61B

Lecture 32: Sorting I

- The Sorting Problem
- Selection Sort, Heapsort
- Mergesort
- Insertion Sort
- Shell's Sort (Extra)

Sorting - Definitions

A sort is a permutation (re-arrangement) of a sequence of elements that brings them into order according to some ***total order***. A total order $\leqslant$ is:

- Total: $x \leqslant y$ or $y \leqslant x$ for all x, y
- Reflexive: $x \leqslant x$
- Antisymmetric: $x \leqslant y$ and $y \leqslant x$ iff $x = y$ (x and y are equivalent).
- Transitive: $x \leqslant y$ and $y \leqslant z$ implies $x \leqslant z$.

In Java, total order is typically specified by compareTo or compare methods.

- May be inconsistent with equals! For example sorting an array of Strings by length has items that are equivalent, but not equal, e.g. “cat” and “dog”.

Sorting: An Alternate Viewpoint

An ***inversion*** is a pair of elements that are out of order.

Yoda

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)

Gabriel Cramer

Goal of sorting:

- Given a sequence of elements with Z inversions.
- Perform a sequence of operations that reduces inversions to 0.

Performance Definitions

Characterizations of the runtime efficiency are sometimes called the **time complexity** of an algorithm. Examples:

- DFS has time complexity $\Theta(V+E)$.

Characterizations of the "extra" memory usage of an algorithm is sometimes called the **space complexity** of an algorithm.

- DFS has space complexity $\Theta(V)$.
  - Note that the graph takes up space $\Theta(V+E)$, but we don't count this as part of the runtime of DFS, since we're only accounting for the extra space that DFS uses.

Selection Sort and Heapsort

Selection Sort

We've seen this already.

- Find smallest item.
- Swap this item to the front and 'fix' it.
- Repeat for unfixed items until all items are fixed.
- Demo: https://goo.gl/g14Cit

Sort Properties:

- $\Theta(N^2)$ time if we use an array (or similar data structure).

Seems inefficient: We look through entire remaining array every time to find the minimum.

Naive Heapsort: Leveraging a Max-Oriented Heap

Idea: Instead of rescanning entire array looking for minimum, maintain a heap so that getting the minimum is fast!

For reasons that will become clear soon, we'll use a max-oriented heap.

A min heap would work as well, but wouldn't be able to take advantage of the fancy trick in a few slides.

Naive heapsorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put largest item at the end of the unused part of the output array.

Naive Heapsort Demo: https://goo.gl/EZWwSJ

Naive Heapsort Runtime: http://yellkey.com/suffer

Naive heapsorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put largest item at the end of the unused part of the output array.

What is the TOTAL runtime of naive heapsort?

A. $\Theta(N)$
B. $\Theta(N \log N)$
C. $\Theta(N^2)$, but faster than selection sort.

Heapsort Runtime Analysis

Use the magic of the heap to sort our data.

- Getting items into the heap O(N log N) time.
- Selecting *largest* item: Θ(1) time.
- Removing *largest* item: O(log N) for each removal.

Informally?? [good eye!]

Informally, overall runtime is O(N log N) + Θ(N) + O(N log N) = **O(N log N)**

- Far better that selection sort!

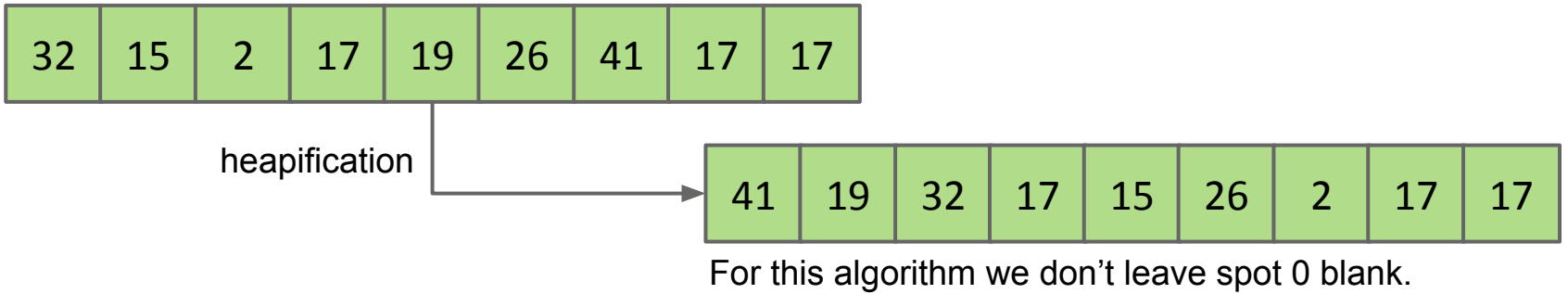
Memory usage is Θ(N) to build the additional copy of all of our data.

- Worse than selection sort, but probably no big deal (??).
- Can eliminate this extra memory cost with same fancy trickery.

In-place Heapsort

Alternate approach, treat input array as a heap!

- Rather than inserting into a new array of length N + 1, use a process known as "bottom-up heapification" to convert the array into a heap.
  - To bottom-up heapify, just sink nodes in reverse level order.
- Avoids need for extra copy of all data.
- Once heapified, algorithm is almost the same as naive heap sort.



In-place heap sort: Demo

In-place Heapsort Runtime: http://yellkey.com/fail

Use the magic of the heap to sort our data.

- Bottom-up Heapification: O(???) time.
- Selecting *largest* item: Θ(1) time.
- Removing *largest* item: O(log N) for each removal.

Give the time complexity of in-place heapsort in big O notation.

A. O(N)
B. O(N log N)
C. $O(N^2)$

In-place Heapsort Runtime

Use the magic of the heap to sort our data.

- Bottom-up Heapification: O(N log N) time.
- Selecting *largest* item: Θ(1) time.
- Removing *largest* item: O(log N) for each removal.

Give the time complexity of in-place heapsort in big O notation.

A. O(N log N)

Bottom-up heapification is N sink operations, each taking no more than O(log N) time, so overall runtime for heapification is O(N log N).

- Extra for experts, show that bottom-up heapification is Θ(N) in the worst case.
- More extra for experts, show heapsort is Θ(N log N) in the worst case.

In-place Heapsort: http://yellkey.com/once

What is the **memory complexity** of Heapsort?

- Also called “space complexity”.

A. $\Theta(1)$
B. $\Theta(\log N)$
C. $\Theta(N)$
D. $\Theta(N \log N)$
E. $\Theta(N^2)$

In-place Heapsort

What is the **memory complexity** of Heapsort?

A. Θ(1)
B. Θ(log N)
C. Θ(N)
D. Θ(N log N)
E. $\Theta(N^2)$

The only extra memory we need is a constant number instance variables, e.g. size.

- Unimportant caveat: If we employ recursion to implement various heap operations, space complexity is Θ(log N) due to need to track recursive calls. The difference between Θ(log N) and Θ(1) space is effectively nothing.

Sorts So Far

	Best Case Runtime	Worst Case Runtime	Space	Demo	Notes
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$	Link	
Heapsort (in place)	$\Theta(N)$*	$\Theta(N \log N)$	$\Theta(1)$	Link	Bad cache (61C) performance.

*: An array of all duplicates yields linear runtime for heapsort.

Mergesort

Mergesort

We've seen this one before as well.

Mergesort: Demo

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.

Time complexity, analysis from previous lecture: Θ(N log N runtime)

- Space complexity with aux array: Costs Θ(N) memory.

Also possible to do in-place merge sort, but algorithm is very complicated, and runtime performance suffers by a significant constant factor.

Sorts So Far

	Best Case Runtime	Worst Case Runtime	Space	Demo	Notes
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$	Link	
Heapsort (in place)	$\Theta(N)$*	$\Theta(N \log N)$	$\Theta(1)$	Link	Bad cache (61C) performance.
Mergesort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N)$	Link	Faster than heap sort.

*: An array of all duplicates yields linear runtime for heapsort.

Insertion Sort

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output: Demo (Link)

Input:

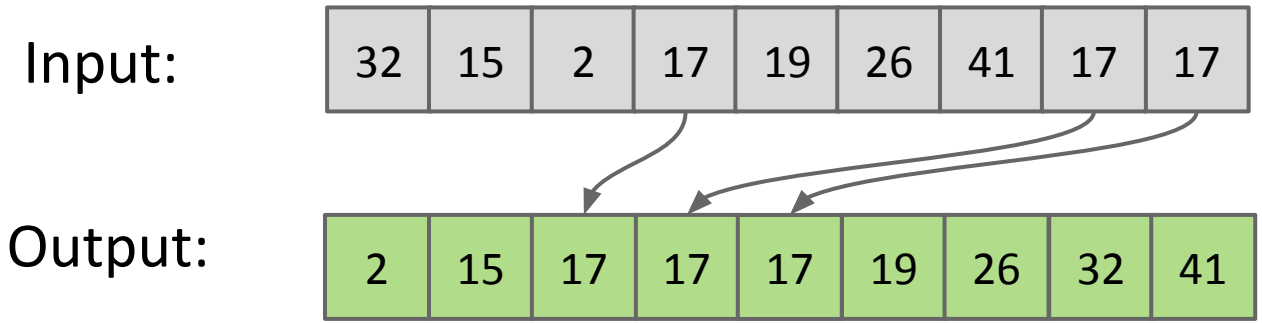
Output:

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output: Demo (Link)



Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

For naive approach, if output sequence contains k items, worst cost to insert a single item is k.

- Might need to move everything over.

More efficient method:

- Do everything in place using swapping: Demo (Link)

In-place Insertion Sort

Two more examples.

7 swaps:

```
P O T A T O
P O T A T O  (0 swaps)
O P T A T O  (1 swap )
O P T A T O  (0 swaps)
A O P T T O  (3 swaps)
A O P T T O  (0 swaps)
A O O P T T  (3 swaps)
```

36 swaps:

```
S O R T E X A M P L E
S O R T E X A M P L E  (0 swaps)
O S R T E X A M P L E  (1 swap )
O R S T E X A M P L E  (1 swap )
O R S T E X A M P L E  (0 swaps)
E O R S T X A M P L E  (4 swaps)
E O R S T X A M P L E  (0 swaps)
A E O R S T X M P L E  (6 swaps)
A E M O R S T X P L E  (5 swaps)
A E M O P R S T X L E  (4 swaps)
A E L M O P R S T X E  (7 swaps)
A E E L M O P R S T X  (8 swaps)
```

Purple: Element that we're moving left (with swaps).
Black: Elements that got swapped with purple.
Grey: Not considered this iteration.

Insertion Sort Runtime: http://yellkey.com/sing

What is the runtime of insertion sort?

A. $\Omega(1), O(N)$
B. $\Omega(N), O(N)$
C. $\Omega(1), O(N^2)$
D. $\Omega(N), O(N^2)$
E. $\Omega(N^2), O(N^2)$

36 swaps:

```
S O R T E X A M P L E
S O R T E X A M P L E  (0 swaps)
O S R T E X A M P L E  (1 swap )
O R S T E X A M P L E  (1 swap )
O R S T E X A M P L E  (0 swaps)
E O R S T X A M P L E  (4 swaps)
E O R S T X A M P L E  (0 swaps)
A E O R S T X M P L E  (6 swaps)
A E M O R S T X P L E  (5 swaps)
A E M O P R S T X L E  (4 swaps)
A E L M O P R S T X E  (7 swaps)
A E E L M O P R S T X  (8 swaps)
```

Insertion Sort Runtime: http://yellkey.com/sing

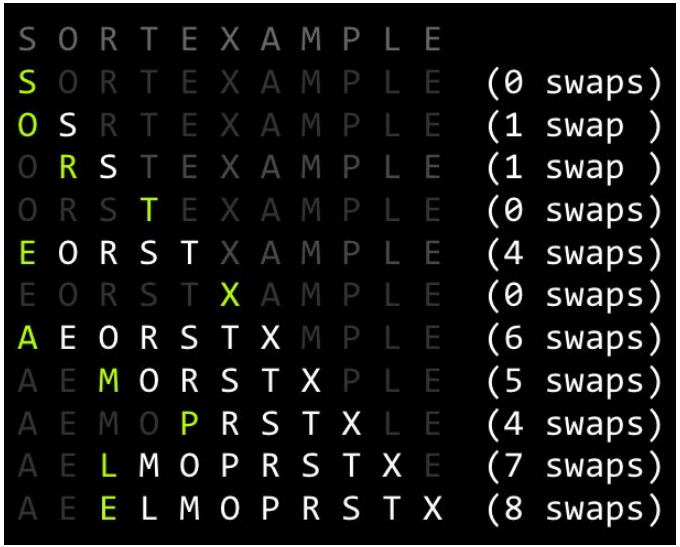
What is the runtime of insertion sort?

A. $\Omega(1)$, $O(N)$
B. $\Omega(N)$, $O(N)$
C. $\Omega(1)$, $O(N^2)$
D. $\Omega(N)$, $O(N^2)$
E. $\Omega(N^2)$, $O(N^2)$

You may recall Ω is not “best case”.

So technnnniically you could also say $\Omega(1)$

36 swaps:



Picking the Best Sort: http://yellkey.com/national

Suppose we do the following:

- Read 1,000,000 integers from a file into an array of length 1,000,000.
- Mergesort these integers.
- Select one integer randomly and change it.
- Sort using algorithm X of your choice.

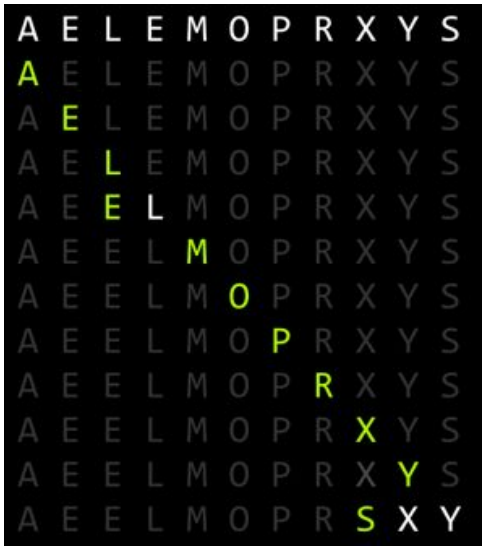
Which sorting algorithm would be the fastest choice for X?

A. Selection Sort
B. Heapsort
C. Mergesort
D. Insertion Sort

Observation: Insertion Sort on Almost Sorted Arrays

For arrays that are almost sorted, insertion sort does very little work.

- Left array: 5 inversions, so only 5 swaps.
- Right array: 3 inversion, so only 3 swaps.



Picking the Best Sort (Poll Everywhere)

Suppose we do the following:

- Read 1,000,000 integers from a file into an array of length 1,000,000.
- Mergesort these integers.
- Select one integer randomly and change it.
- Sort using algorithm X of your choice.
- In the worst case, we have 999,999 inversions: $\Theta(N)$ inversions.

Which sorting algorithm would be the fastest choice for X? Worst case run-times:

A. Selection Sort: $\Theta(N^2)$
B. Heapsort: $\Theta(N \log N)$
C. Mergesort: $\Theta(N \log N)$
D. Insertion Sort: $\Theta(N)$

Insertion Sort Sweet Spots

On arrays with a small number of inversions, insertion sort is extremely fast.

- One exchange per inversion (and number of comparisons is similar). Runtime is $\Theta(N + K)$ where K is number of inversions.
- Define an ***almost sorted*** array as one in which number of inversions $\leq cN$ for some c. Insertion sort is excellent on these arrays.

Less obvious: For small arrays ($N < 15$ or so), insertion sort is fastest.

- More of an empirical fact than a theoretical one.
- Theoretical analysis beyond scope of the course.
- Rough idea: Divide and conquer algorithms like heapsort / mergesort spend too much time dividing, but insertion sort goes straight to the conquest.
- The Java implementation of Mergesort does this (Link).

Sorts So Far

	Best Case Runtime	Worst Case Runtime	Space	Demo	Notes
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$	Link	
Heapsort (in place)	$\Theta(N)$*	$\Theta(N \log N)$	$\Theta(1)$	Link	Bad cache (61C) performance.
Mergesort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N)$	Link	Fastest of these.
Insertion Sort (in-place)	$\Theta(N)$	$\Theta(N^2)$	$\Theta(1)$	Link	Best for small N or almost sorted.

Shell's Sort (Extra)
(Not on Exam)

Optimizing Insertion Sort: Shell's Sort

Big idea: Fix multiple inversions at once.

- Instead of comparing adjacent items, compare items that are one stride length h apart.
- Start with large stride, and decrease towards 1.
- Example: h = 7, 3, 1.

Optimizing Insertion Sort: Shell's Sort

h=1 is just insertion sort.

7-sorting: 3 swaps

```
S O R T E X A M P L E
M O R T E X A S P L E   (1 swap, > 1 inversion)
M O R T E X A S P L E   (0 swaps)
M O L T E X A S P R E   (1 swap, > 1 inversion)
M O L E E X A S P R T   (1 swap, > 1 inversion)
```

3-sorting: 5 swaps

```
M O L E E X A S P R T
E O L M E X A S P R T   (1 swap,  >1 inversions)
E E L M O X A S P R T   (1 swap,  >1 inversions)
E E L M O X A S P R T   (0 swaps,  0 inversions)
A E L E O X M S P R T   (2 swaps, >1 inversions)
A E L E O X M S P R T   (0 swaps,  0 inversions)
A E L E O P M S X R T   (1 swaps, >1 inversions)
A E L E O P M S X R T   (0 swaps,  0 inversions)
A E L E O P M S X R T   (0 swaps,  0 inversions)
```

1-sorting: 6 swaps

```
A E L E O P M S X R T
A E L E O P M S X R T
A E L E O P M S X R T
A E L E O P M S X R T
A E E L O P M S X R T
A E E L O P M S X R T
A E E L M O P S X R T
A E E L M O P S X R T
A E E L M O P S X R T
A E E L M O P R S X T
A E E L M O P R S T X
```

Total swaps: 14 (vs. 31)

Example from Algorithms 4th Edition

Shell's Sort: Generalization and Performance

h=1 is just normal insertion sort.

- By using large strides first, fixes most of the inversions.

We used 7, 3, 1. Can generalize to $2^k - 1$ from some k down to 1.

- Requires $\Theta(N^{1.5})$ time in the worst case (see CS170).
- Other stride patterns can be faster.

$1, 3, 7, 15, 31, 63, \ldots$	$\Theta(N^{3/2})$
$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta(N^{3/2})$
$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta(N \log^2 N)$
$1, 4, 13, 40, 121, \ldots$	$\Theta(N^{3/2})$
$1, 3, 7, 21, 48, 112, \ldots$	$O(N^{1+\sqrt{8 \ln(5/2)/\ln N}})$
$1, 8, 23, 77, 281, \ldots$	$O(N^{4/3})$
$1, 5, 19, 41, 109, \ldots$	$O(N^{4/3})$

Sorts So Far

	Best Case Runtime	Worst Case Runtime	Space	Demo	Notes
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$	Link	
Heapsort (in place)	$\Theta(N)$*	$\Theta(N \log N)$	$\Theta(1)$	Link	Bad cache (61C) performance.
Mergesort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N)$	Link	Fastest of these.
Insertion Sort (in-place)	$\Theta(N)$	$\Theta(N^2)$	$\Theta(1)$	Link	Best for small N or almost sorted.
Shell's Sort	$\Theta(N)$	$\Omega(N \log N)$, $O(?)$	$\Theta(1)$	N/A	Rich theory!

An earlier version of this slide assumed that Heapsort was always given an array with no duplicate elements. If we omit this assumption, heapsort's best case is $\Theta(N)$.

Citations

Lego man:
http://thetechnicgear.com/wp-content/uploads/2014/02/sorting-lego.jpg

Keynote Animations of Algorithms courtesy of Kevin Wayne