Sorting

Exam Prep 11

Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
					4/12 Lab 10 due	
	4/15 Project 3A due					

Content Review

Insertion Sort

Insertion sort iterates through the list and swaps items backwards as necessary to maintain sortedness.

3 5 1 2 4

Runtime: $O(N^2)$

Selection Sort

Selection sort finds the smallest remaining element in the unsorted portion of the list at each time step and swaps it into the correct position.

3 5 1 2 4

Runtime: $\Theta(N^2)$

Merge Sort

Merge sort splits the list in half, applies merge sort to each half, and then merges the two halves together in a zipper fashion.

3 5 1 2 4

Runtime: Θ(NlogN)

Quicksort

Quicksort picks a pivot (ie. first element) and uses Hoare partitioning to divide the list so that everything greater than the pivot is on its right and everything less than the pivot is on its left.

3 5 1 2 4

Runtime: Average case O(NlogN), slowest case $O(N^2)$ (dependent on pivot selection)

Heap Sort

Heapsort heapifies the array into a max heap and pops the largest element off and appends it to the end until there are no elements left in the heap. You can heapify by sinking nodes in reverse level order.

3 5 1 2 4

Runtime: O(NlogN)

Summary for comparison sorts

Stability: a sort is stable if duplicate values remain in the same relative order after sorting as they were initially. In other words, is 2a guaranteed to be before 2b after sorting the list [2a, 2b, 1]?

	Worst Case	Best Case	Stable?
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	No
Insertion Sort	$\Theta(N^2)$	$\Theta(N)$	Yes
Merge Sort	$\Theta(N\log N)$	$\Theta(N\log N)$	Yes
Quicksort	$\Theta(N^2)$	$\Theta(N\log N)$	No*
Heapsort	$\Theta(N\log N)$	$\Theta(N)$	No

Try reasoning out or coming up with examples for these best and worst case runtimes!

*with hoare partitioning

Worksheet

1A Identifying Sorts

1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

1429, 3291, 7683, 192, 1337, 594, 4242, 9001, 4392, 129, 1000

1429, 3291, 192, 1337, 7683, 594, 4242, 9001, 129, 1000, 4392

192, 1337, 1429, 3291, 7683, 129, 594, 1000, 4242, 4392, 9001

1A Identifying Sorts

1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

1429, 3291, 7683, 192, 1337, 594, 4242, 9001, 4392, 129, 1000

1429, 3291, 192, 1337, 7683, 594, 4242, 9001, 129, 1000, 4392

192, 1337, 1429, 3291, 7683, 129, 594, 1000, 4242, 4392, 9001

Mergesort. One identifying feature of mergesort is that the left and right halves do not interact with each other until the very end.

1B Identifying Sorts

1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

1337, 192, 594, 129, 1000, 1429, 3291, 7683, 4242, 9001, 4392

192, 594, 129, 1000, 1337, 1429, 3291, 7683, 4242, 9001, 4392

129, 192, 594, 1000, 1337, 1429, 3291, 4242, 4392, 7683, 9001

1B Identifying Sorts

1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

1B Identifying Sorts

1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

[1337, 192, 594, 129, 1000] 1429 [3291, 7683, 4242, 9001, 4392]

1B Identifying Sorts

1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

[1337, 192, 594, 129, 1000] 1429 [3291, 7683, 4242, 9001, 4392]

[192, 594, 129, 1000] 1337 1429 3291 [7683, 4242, 9001, 4392]

1B Identifying Sorts

1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

[1337, 192, 594, 129, 1000] 1429 [3291, 7683, 4242, 9001, 4392]

[192, 594, 129, 1000] 1337 1429 3291 [7683, 4242, 9001, 4392]

[129] 192 [594, 1000] 1337 1429 3291 [4242, 4392] 7683 [9001]

1B Identifying Sorts

1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

[1337, 192, 594, 129, 1000] 1429 [3291, 7683, 4242, 9001, 4392]

[192, 594, 129, 1000] 1337 1429 3291 [7683, 4242, 9001, 4392]

[129] 192 [594, 1000] 1337 1429 3291 [4242, 4392] 7683 [9001]

Quicksort. First item was chosen as pivot, so the first pivot is 1429. Everything to the left of 1429 is less than 1429, everything to the right is greater than 1429. Chosen pivots are highlighted.

1C Identifying Sorts

1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

1337, 1429, 3291, 7683, 192, 594, 4242, 9001, 4392, 129, 1000

192, 1337, 1429, 3291, 7683, 594, 4242, 9001, 4392, 129, 1000

192, 594, 1337, 1429, 3291, 7683, 4242, 9001, 4392, 129, 1000

1C Identifying Sorts

1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

1337, 1429, 3291, 7683, 192, 594, 4242, 9001, 4392, 129, 1000

192, 1337, 1429, 3291, 7683, 594, 4242, 9001, 4392, 129, 1000

192, 594, 1337, 1429, 3291, 7683, 4242, 9001, 4392, 129, 1000

Insertion Sort. Insertion sort starts at the front, and for each item, move to the left as far as possible. This creates a sorted section (highlighted) on the left of the array. These are the first few iterations of insertion sort so the right side is left unchanged.

1D Identifying Sorts

1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

1429, 3291, 7683, 9001, 1000, 594, 4242, 1337, 4392, 129, 192

7683, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 129, 9001

129, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 7683, 9001

1D Identifying Sorts

1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

1429, 3291, 7683, 9001, 1000, 594, 4242, 1337, 4392, 129, 192

7683, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 129, 9001

129, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 7683, 9001

Heapsort. This one's a bit more tricky. Basically what's happening is that the second line is in the middle of heapifying this list into a maxheap (in blue). Then we continually remove the max and place it at the end.

2A Conceptual Sorts

We have a system running insertion sort and we find that it's completing faster than expected. What could we conclude about the input to the sorting algorithm?

2A Conceptual Sorts

We have a system running insertion sort and we find that it's completing faster than expected. What could we conclude about the input to the sorting algorithm?

The array is nearly sorted. Note that insertion sort has a best case runtime of Θ(N), which is when the array is already sorted.

Example: Sorting the array [0 2 1 3 4] with insertion sort only requires one swap

2B Conceptual Sorts

Give a 5 integer array that elicits the worst case runtime for insertion sort.

2B Conceptual Sorts

Give a 5 integer array that elicits the worst case runtime for insertion sort.

A simple example is [5, 4, 3, 2, 1]. Any 5 integer array in descending order would work.

2C Conceptual Sorts

(T/F) Heapsort is stable.

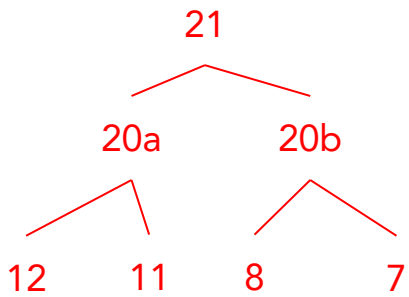
2C Conceptual Sorts

(T/F) Heapsort is stable.

False, stability for sorting algorithms mean that if two elements in the list are defined to be equal, then they will retain their relative ordering after the sort is complete. Heap operations may mess up the relative ordering of equal items and thus is not stable. As a concrete example, consider the max heap: 21 20a 20b 12 11 8 7.

2C Conceptual Sorts

(T/F) Heapsort is stable.

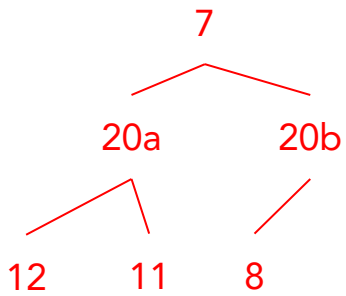


removeMax()

sorted result: [21]

2C Conceptual Sorts

(T/F) Heapsort is stable.

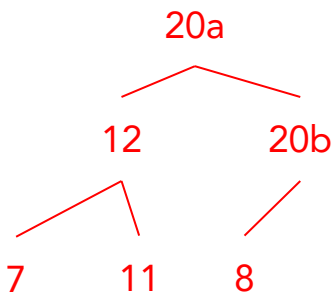


removeMax()

sorted result: [21]

2C Conceptual Sorts

(T/F) Heapsort is stable.

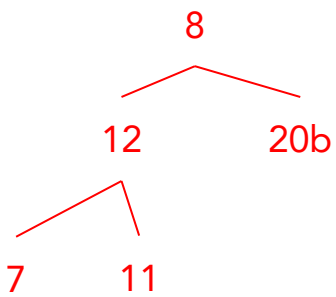


removeMax()

sorted result: [21]

2C Conceptual Sorts

(T/F) Heapsort is stable.

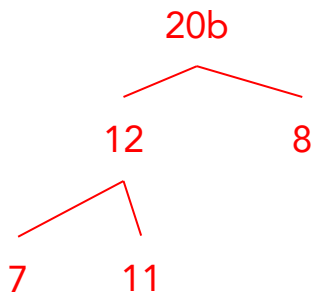


removeMax()

sorted result: [20a, 21]

2C Conceptual Sorts

(T/F) Heapsort is stable.

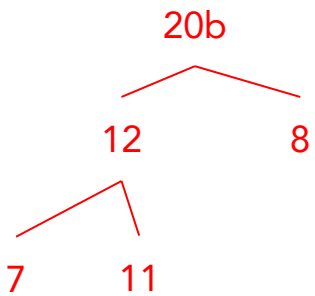


removeMax()

sorted result: [20a, 21]

2C Conceptual Sorts

(T/F) Heapsort is stable.



removeMax()

sorted result: [20b, 20a, 21]

2D Conceptual Sorts

Compare mergesort and quicksort in terms of (1) runtime, (2) stability, and (3) memory efficiency for sorting linked lists.

2D Conceptual Sorts

Compare mergesort and quicksort in terms of (1) runtime, (2) stability, and (3) memory efficiency for sorting linked lists.

- **Mergesort has a better worst case runtime: $\Theta(N \log N)$ instead of $\Theta(N^2)$**
- **Mergesort is stable, maintains relative ordering of elements**
- **Easier to sort a linked list with mergesort**

2E Conceptual Sorts

Bounded by Ω(N log N) lower bound.

Has a worst case runtime that is asymptotically better than Quicksort's worstcase runtime.

Never compares the same two elements twice.

Runs in best case Θ(log N)time for certain inputs.

2E Conceptual Sorts

Bounded by Ω(N log N) lower bound.

Quicksort, Mergesort, Selection Sort. Insertion (sorted array), heapsort (equal items) are linear in the best case.

Has a worst case runtime that is asymptotically better than Quicksort's worstcase runtime.

Never compares the same two elements twice.

Runs in best case Θ(log N)time for certain inputs.

2E Conceptual Sorts

Bounded by Ω(N log N) lower bound.

Quicksort, Mergesort, Selection Sort. Insertion (sorted array), heapsort (equal items) are linear in the best case.

Has a worst case runtime that is asymptotically better than Quicksort's worstcase runtime.

Mergesort, Heapsort are guaranteed O(N log N).

Never compares the same two elements twice.

Runs in best case Θ(log N)time for certain inputs.

2E Conceptual Sorts

Bounded by Ω(N log N) lower bound.

Quicksort, Mergesort, Selection Sort. Insertion (sorted array), heapsort (equal items) are linear in the best case.

Has a worst case runtime that is asymptotically better than Quicksort's worstcase runtime.

Mergesort, Heapsort are guaranteed O(N log N).

Never compares the same two elements twice.

Quicksort, Mergesort, Insertion Sort.

Runs in best case Θ(log N)time for certain inputs.

2E Conceptual Sorts

Bounded by Ω(NlogN) lower bound.

Has a worst case runtime that is asymptotically better than Quicksort's worstcase runtime.

Never compares the same two elements twice.
Quicksort, Mergesort, Insertion Sort.

Runs in best case Θ(logN)time for certain inputs.

2E Conceptual Sorts

Never compares the same two elements twice.

Quicksort: comparisons happen during pivot. Each item only serves as the pivot once.

2E Conceptual Sorts

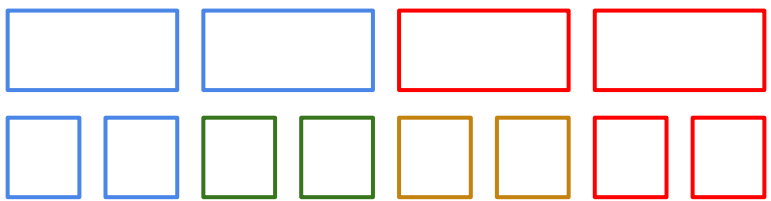
Never compares the same two elements twice.

Mergesort: comparisons happen during merging. Always compare items from different halves of the recursion.

2E Conceptual Sorts

Never compares the same two elements twice.

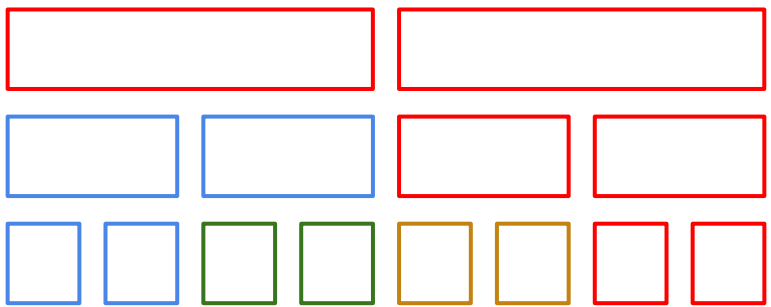
Mergesort: comparisons happen during merging. Always compare items from different halves of the recursion.



2E Conceptual Sorts

Never compares the same two elements twice.

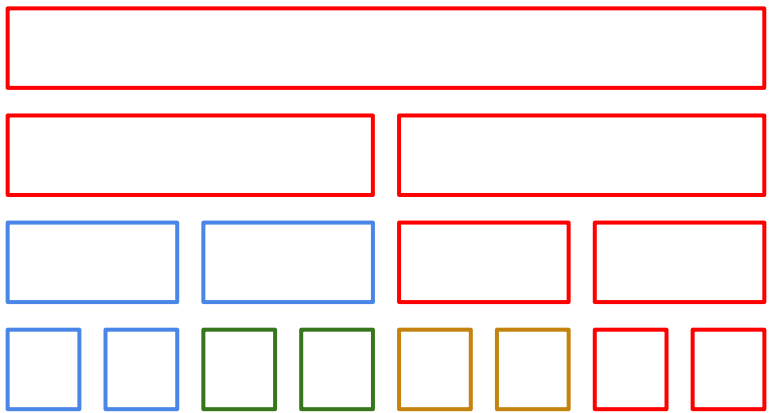
Mergesort: comparisons happen during merging. Always compare items from different halves of the recursion.



2E Conceptual Sorts

Never compares the same two elements twice.

Mergesort: comparisons happen during merging. Always compare items from different halves of the recursion.



2E Conceptual Sorts

Never compares the same two elements twice.

Selection sort: comparisons happen when finding max. May compare same elements more than once.

```
max = arr[0]
for (int i = 1; i < arr.length; i++) {
    if (arr[i] > max) {
        max = arr[i]
}
```


2E Conceptual Sorts

Never compares the same two elements twice.

Insertion sort: comparisons happen when swapping to the front. Once an item is swapped to the front, it is never swapped again.

2E Conceptual Sorts

Never compares the same two elements twice.

Heapsort: an item can be compared multiple times during heapification and bubbling down.

2E Conceptual Sorts

Bounded by Ω(NlogN)lower bound.

Has a worst case runtime that is asymptotically better than Quicksort's worstcase runtime.

Never compares the same two elements twice.

Runs in best case Θ(logN) time for certain inputs.
None - every sort looks at each element at least once

3 Bears and Beds

Inputs:

- **A list of `Bears` with unique but unknown sizes**
- **A list of `Beds` with unique but unknown sizes**
- ***Note: these two lists are not necessarily in the same order***

Output: a list of `Bears` and a list of `Beds` such that the `i`th `Bear` is the same size as the `i`th `Bed`

Example input:
Bears - [5 1 9 2 7], Beds - [2 1 5 7 9]

Example output: (multiple possible correct outputs)
Bears - [9 2 5 1 7], Beds - [9 2 5 1 7]

Constraints:

- `Bears` can only be compared to `Beds` and we can get feedback on if the Bed is too large, too small, or just right.
- `Beds` can only be compared to `Bears` and we can get feedback on if the `Bear` is too large, too small, or just right.
- Your algorithm should run in O(N log N) time on average.

3 Bears and Beds

Solution:

1. Choose a pivot bear randomly
2. Partition beds into: less than pivot bear, equal to pivot bear, greater than pivot bear
3. Only one bed will be equal to pivot bear (because beds/bears have unique sizes)
4. Choose that one bed as the pivot bed
5. Partition bears based on pivot bed
6. Repeat recursively like Quicksort

3 Bears and Beds

1. Choose a pivot bear randomly
2. Partition beds into: less than pivot bear, equal to pivot bear, greater than pivot bear
3. Only one bed will be equal to pivot bear (because beds/bears have unique sizes)
4. Choose that one bed as the pivot bed
5. Partition bears based on pivot bed
6. Repeat recursively like Quicksort

Bears - [5 1 9 2 7], Beds - [2 1 5 7 9]

Pivot = Bear 5

3 Bears and Beds

1. Choose a pivot bear randomly
2. Partition beds into: less than pivot bear, equal to pivot bear, greater than pivot bear
3. Only one bed will be equal to pivot bear (because beds/bears have unique sizes)
4. Choose that one bed as the pivot bed
5. Partition bears based on pivot bed
6. Repeat recursively like Quicksort

Bears = [5 1 9 2 7]

Beds = [2, 1] [5] [7, 9]

3 Bears and Beds

1. Choose a pivot bear randomly
2. Partition beds into: less than pivot bear, equal to pivot bear, greater than pivot bear
3. Only one bed will be equal to pivot bear (because beds/bears have unique sizes)
4. Choose that one bed as the pivot bed
5. Partition bears based on pivot bed
6. Repeat recursively like Quicksort

Bears = [5 1 9 2 7]

Beds = [2, 1] [5] [7, 9]

3 Bears and Beds

1. Choose a pivot bear randomly
2. Partition beds into: less than pivot bear, equal to pivot bear, greater than pivot bear
3. Only one bed will be equal to pivot bear (because beds/bears have unique sizes)
4. **Choose that one bed as the pivot bed**
5. Partition bears based on pivot bed
6. Repeat recursively like Quicksort

Bears = [5 1 9 2 7]

Beds = [2, 1] [5] [7, 9]

Pivot Bed = 5

3 Bears and Beds

1. Choose a pivot bear randomly
2. Partition beds into: less than pivot bear, equal to pivot bear, greater than pivot bear
3. Only one bed will be equal to pivot bear (because beds/bears have unique sizes)
4. Choose that one bed as the pivot bed
5. **Partition bears based on pivot bed**
6. Repeat recursively like Quicksort

Bears = [1, 2] [5] [9, 7]

Beds = [2, 1] [5] [7, 9]

3 Bears and Beds

1. Choose a pivot bear randomly
2. Partition beds into: less than pivot bear, equal to pivot bear, greater than pivot bear
3. Only one bed will be equal to pivot bear (because beds/bears have unique sizes)
4. Choose that one bed as the pivot bed
5. Partition bears based on pivot bed
6. Repeat recursively like Quicksort

Bears = [1, 2] [5] [9, 7]

Beds = [2, 1] [5] [7, 9]