ASSORTED MINUTIAE

• HW5 Due Tonight – Code + Writeup
ASSORTED MINUTIAE

- HW5 Due Tonight – Code + Writeup
- HW6 Out Monday – Covers Sorting
ASSORTED MINUTIAE

• HW5 Due Tonight – Code + Writeup
• HW6 Out Monday – Covers Sorting
• Extra assignments are out
ASSORTED MINUTIAE

• HW5 Due Tonight – Code + Writeup
• HW6 Out Monday – Covers Sorting
• Extra assignments are out
  • Small change, instead of throwing an ObjectNotFound exception, throw a NoSuchElementException exception.
    (which is in java.util)
ASSORTED MINUTIAE

• Eclipse run configurations
ASSORTED MINUTIAE

• Eclipse run configurations
  • It is possible to pass command line arguments in Eclipse under run configurations
ASSORTED MINUTIAE

• Eclipse run configurations
  • It is possible to pass command line arguments in Eclipse under run configurations
  • If you have edited your main function in FindPahts so that it does not use the String[] args commands, please return it to it’s old state. This is part of the testing script.
SORTING

• Problem statement:
  • Collection of Comparable data
SORTING

• Problem statement:
  • Collection of Comparable data
  • Result should be a sorted collection of the data
SORTING

• Problem statement:
  • Collection of Comparable data
  • Result should be a sorted collection of the data
• Motivation?
SORTING

• Problem statement:
  • Collection of Comparable data
  • Result should be a sorted collection of the data

• Motivation?
  • Pre-processing v. find times
SORTING

• Problem statement:
  • Collection of Comparable data
  • Result should be a sorted collection of the data

• Motivation?
  • Pre-processing v. find times
  • Sorting v. Maintaining sortedness
SORTING

• Important definitions
SORTING

• Important definitions
  • In-place:
SORTING

• Important definitions
  • In-place: Requires only $O(1)$ extra memory
SORTING

• Important definitions
  • In-place: Requires only $O(1)$ extra memory
    • usually means the array is mutated
SORTING

• Important definitions
  • In-place: Requires only $O(1)$ extra memory
    • usually means the array is mutated
  • Stable: For any two elements have the same comparative value, then after the sort, which ever came first will stay first
SORTING

• Important definitions
  • In-place: Requires only $O(1)$ extra memory
    • usually means the array is mutated
  • Stable: For any two elements have the same comparative value, then after the sort, which ever came first will stay first
    • Sorting by first name and then last name will give you last then first with a stable sort.
Important definitions

- In-place: Requires only $O(1)$ extra memory
  - usually means the array is mutated
- Stable: For any two elements have the same comparative value, then after the sort, which ever came first will stay first
  - Sorting by first name and then last name will give you \textit{last then first} with a stable sort.
  - The most recent sort will always be the primary
SORTING

• Important definitions
  • Interruptable:
SORTING

• Important definitions
  • Interruptable: the algorithm can run only until the first $k$ elements are in sorted order
SORTING

• Important definitions
  • Interruptable: the algorithm can run only until the first $k$ elements are in sorted order
  • Comparison sort: utilizes comparisons between elements to produce the final sorted order.
SORTING

• Important definitions
  • Interruptable: the algorithm can run only until the first $k$ elements are in sorted order
  • Comparison sort: utilizes comparisons between elements to produce the final sorted order.
    • Bogo sort is not a comparison sort
SORTING

• Important definitions
  • Interruptable: the algorithm can run only until the first $k$ elements are in sorted order.
  • Comparison sort: utilizes comparisons between elements to produce the final sorted order.
    • Bogo sort is not a comparison sort
    • Comparison sorts are $\Omega(n \log n)$, they cannot do better than this
SORTING

• What are the sorts we’ve seen so far?
• What are the sorts we’ve seen so far?
  • Selection sort:
SORTING

• What are the sorts we’ve seen so far?
  • Selection sort
    • Algorithm?
SORTING

• What are the sorts we’ve seen so far?
  • Selection sort
    • Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
What are the sorts we’ve seen so far?

Selection sort

Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.

Runtime:
SORTING

• What are the sorts we’ve seen so far?
  • Selection sort
    • Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
    • Runtime:
      • First run, you must select from $n$ elements, the second, from $n-1$, and the $k$th from $n-(k-1)$. 
• **What are the sorts we’ve seen so far?**
  • Selection sort
    • Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
    • Runtime:
      • First run, you must select from \( n \) elements, the second, from \( n-1 \), and the \( k \)th from \( n-(k-1) \).
      • **What is this summation?** \( n(n-1)/2 \)
    • Stable?
SORTING

• What are the sorts we’ve seen so far?
  • Selection sort
    • Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
    • Runtime:
      • First run, you must select from $n$ elements, the second, from $n-1$, and the $k$th from $n-(k-1)$.
      • What is this summation? $n(n-1)/2$
    • Stable? How?
• **What are the sorts we’ve seen so far?**
  • Selection sort
    • Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
    • Runtime:
      • First run, you must select from \( n \) elements, the second, from \( n-1 \), and the \( k \)th from \( n-(k-1) \).
      • **What is this summation?** \( n(n-1)/2 \)
    • Stable? How?
      • When you have your lowest candidate, do not replace with an element that ties.
• What are the sorts we’ve seen so far?
  • Selection sort
    • Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
    • Runtime:
      • First run, you must select from \( n \) elements, the second, from \( n-1 \), and the \( k \)th from \( n-(k-1) \).
      • What is this summation? \( n(n-1)/2 \)
    • Stable? How?
      • When you have your lowest candidate, do not replace with an element that ties.
    • In place?
• What are the sorts we’ve seen so far?
  • Selection sort
    • Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
    • Runtime:
      • First run, you must select from $n$ elements, the second, from $n-1$, and the $k$th from $n-(k-1)$.
      • What is this summation? $\frac{n(n-1)}{2}$
    • Stable? How?
      • When you have your lowest candidate, do not replace with an element that ties.
    • In place? Can be, but can also create a separate collection (if we only want the top 5, for example)
SORTING

• What are the sorts we’ve seen so far?
  • Insertion Sort:
    • Algorithm?
SORTING

• What are the sorts we’ve seen so far?
  • Insertion Sort:
    • Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches its correct location
SORTING

• What are the sorts we’ve seen so far?
  • Insertion Sort:
    • Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches its correct location
    • Runtime?
SORTING

• What are the sorts we’ve seen so far?
  • Insertion Sort:
    • Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches its correct location
    • Runtime?
      • Worst-case: $O(n^2)$ – what case is this?
SORTING

• What are the sorts we’ve seen so far?
  • Insertion Sort:
    • Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches its correct location
    • Runtime?
      • Worst-case: O(n^2) – reverse sorted order
• What are the sorts we’ve seen so far?
  • Insertion Sort:
    • Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches its correct location
    • Runtime?
      • Worst-case: $O(n^2)$ – reverse sorted order
      • Best-case:
SORTING

• What are the sorts we’ve seen so far?
  • Insertion Sort:
    • Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches its correct location
    • Runtime?
      • Worst-case: $O(n^2)$ – reverse sorted order
      • Best-case: $O(n)$
SORTING

• What are the sorts we’ve seen so far?
  • Insertion Sort:
    • Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches its correct location
    • Runtime?
      • Worst-case: $O(n^2)$ – reverse sorted order
      • Best-case: $O(n)$ – sorted order
SORTING

• What are the sorts we’ve seen so far?
  • Insertion Sort:
    • Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches its correct location
    • Runtime?
      • Worst-case: $O(n^2)$ – reverse sorted order
      • Best-case: $O(n)$ – sorted order
      • Where does this difference come from?
SORTING

• What are the sorts we’ve seen so far?
  • Insertion Sort:
    • Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches its correct location
    • Runtime?
      • Worst-case: $O(n^2)$ – reverse sorted order
      • Best-case: $O(n)$ – sorted order
      • Where does this difference come from?
        • When “swapping” into the sorted array, it can stop when it reaches the correct position, possibly terminating early. Selection sort must check all $k$ elements to be sure it has the correct one
SORTING

• What are the sorts we’ve seen so far?
  • Insertion Sort:
    • Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches its correct location
    • Runtime?
      • Worst-case: $O(n^2)$ – reverse sorted order
      • Best-case: $O(n)$ – sorted order
      • Where does this difference come from?
        • When “swapping” into the sorted array, it can stop when it reaches the correct position, possibly terminating early. Selection sort must check all $k$ elements to be sure it has the correct one
  • Stable?
SORTING

• What are the sorts we’ve seen so far?
  • Insertion Sort:
    • Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches its correct location
    • Runtime?
      • Worst-case: $O(n^2)$ – reverse sorted order
      • Best-case: $O(n)$ – sorted order
      • Where does this difference come from?
        • When “swapping” into the sorted array, it can stop when it reaches the correct position, possibly terminating early. Selection sort must check all $k$ elements to be sure it has the correct one
    • Stable? Same as before, if we maintain sorted order in case of ties.
SORTING

• What are the sorts we’ve seen so far?
  • Insertion Sort:
    • Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches its correct location
    • Runtime?
      • Worst-case: $O(n^2)$ – reverse sorted order
      • Best-case: $O(n)$ – sorted order
      • Where does this difference come from?
        • When “swapping” into the sorted array, it can stop when it reaches the correct position, possibly terminating early. Selection sort must check all $k$ elements to be sure it has the correct one
    • Stable? Same as before, if we maintain sorted order in case of ties.
    • In-place?
SORTING

• What are the sorts we’ve seen so far?
  • Insertion Sort:
    • Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it’s correct location
    • Runtime?
      • Worst-case: $O(n^2)$ – reverse sorted order
      • Best-case: $O(n)$ – sorted order
    • Where does this difference come from?
      • When “swapping” into the sorted array, it can stop when it reaches the correct position, possibly terminating early. Selection sort must check all $k$ elements to be sure it has the correct one
    • Stable? Same as before, if we maintain sorted order in case of ties.
    • In-place? Can be easily. Since not interruptable, having a duplicate array is only necessary if you don’t want the original array to be mutated
SORTING

• What other sorting techniques can we consider?
• What other sorting techniques can we consider?
  • We know $O(n \log n)$ is possible. How do we do it?
SORTING

• What other sorting techniques can we consider?
  • We know $O(n \log n)$ is possible. How do we do it?
  • Heap sort works on principles we already know.
SORTING

• What other sorting techniques can we consider?
  • We know $O(n \log n)$ is possible. How do we do it?
  • Heap sort works on principles we already know.
    • Building a heap from an array takes $O(n)$ time
SORTING

• What other sorting techniques can we consider?
  • We know $O(n \log n)$ is possible. How do we do it?
  • Heap sort works on principles we already know.
    • Building a heap from an array takes $O(n)$ time
    • Removing the smallest element from the array takes $O(\log n)$
SORTING

• What other sorting techniques can we consider?
  • We know $O(n \log n)$ is possible. How do we do it?
  • Heap sort works on principles we already know.
    • Building a heap from an array takes $O(n)$ time
    • Removing the smallest element from the array takes $O(\log n)$
    • There are $n$ elements.
SORTING

• What other sorting techniques can we consider?
  • We know $O(n \log n)$ is possible. How do we do it?
  • Heap sort works on principles we already know.
    • Building a heap from an array takes $O(n)$ time
    • Removing the smallest element from the array takes $O(\log n)$
  • There are $n$ elements.
  • $N + N*\log N = O(N \log N)$
SORTING

• What other sorting techniques can we consider?
  • We know O(n log n) is possible. How do we do it?
  • Heap sort works on principles we already know.
    • Building a heap from an array takes O(n) time
    • Removing the smallest element from the array takes O(log n)
  • There are n elements.
  • N + N*log N = O(N log N)
  • Using Floyd’s method does not improve the asymptotic runtime for heap sort, but it is an improvement.
HEAP SORT

• How do we actually implement this sort?
• Can we do it in place?
HEAP SORT

• How do we actually implement this sort?
• Can we do it in place?
IN-PLACE HEAP SORT

- Treat the initial array as a heap (via `buildHeap`)
- When you delete the $i^{th}$ element, put it at $arr[n-i]$
  - That array location isn’t needed for the heap anymore!

```
4 7 5 9 8 6 10 3 2 1
```

The array is divided into two parts:
- **Sorted part**
- **Heap part**

Put the minimum element at the end of the heap.

```
5 7 6 9 8 10 4 3 2 1
```

```
arr[n-i] = deleteMin()
```

The diagram shows the process of in-place heap sort.
HEAP SORT

• How do we actually implement this sort?
• Can we do it in place?
• Is this sort stable?
HEAP SORT

• How do we actually implement this sort?
• Can we do it in place?
• Is this sort stable?
  • No. Recall that heaps do not preserve FIFO property
HEAP SORT

• How do we actually implement this sort?
• Can we do it in place?
• Is this sort stable?
  • No. Recall that heaps do not preserve FIFO property
  • If it needed to be stable, we would have to modify the priority to indicate its place in the array, so that each element has a unique priority.
IN-PLACE HEAP SORT

What is undesirable about this method?

arr[n-i] = deleteMin()

put the min at the end of the heap data
IN-PLACE HEAP SORT

What is undesirable about this method?

You must reverse the array at the end.

arr[n-i] = deleteMin()
HEAP SORT

- Can implement with a max-heap, then the sorted portion of the array fills in from the back and doesn’t need to be reversed at the end.
“AVL SORT”? “HASH SORT”?

AVL Tree: sure, we can also use an AVL tree to:
“AVL SORT”? “HASH SORT”?

AVL Tree: sure, we can also use an AVL tree to:

- **insert** each element: total time $O(n \log n)$
- Repeatedly **deleteMin**: total time $O(n \log n)$
  - Better: in-order traversal $O(n)$, but still $O(n \log n)$ overall
- But this cannot be done in-place and has worse constant factors than heap sort
“AVL SORT”? “HASH SORT”?

AVL Tree: sure, we can also use an AVL tree to:

- **insert** each element: total time $O(n \log n)$
- Repeatedly **deleteMin**: total time $O(n \log n)$
  - Better: in-order traversal $O(n)$, but still $O(n \log n)$ overall
- But this cannot be done in-place and has worse constant factors than heap sort

Hash Structure: don’t even think about trying to sort with a hash table!
“AVL SORT”? “HASH SORT”?

AVL Tree: sure, we can also use an AVL tree to:

- **insert** each element: total time $O(n \log n)$
- Repeatedly **deleteMin**: total time $O(n \log n)$
  - Better: in-order traversal $O(n)$, but still $O(n \log n)$ overall
- But this cannot be done in-place and has worse constant factors than heap sort

Hash Structure: don’t even think about trying to sort with a hash table!

- Finding min item in a hashtable is $O(n)$, so this would be a slower, more complicated selection sort
SORTING: THE BIG PICTURE

Simple algorithms: \( O(n^2) \)
- Insertion sort
- Selection sort
- Shell sort
...

Fancier algorithms: \( O(n \log n) \)
- Heap sort
- Merge sort
- Quick sort (avg)
...

Comparison lower bound: \( \Omega(n \log n) \)

Specialized algorithms: \( O(n) \)
- Bucket sort
- Radix sort

Handling huge data sets
- External sorting
Divide-and-conquer is a useful technique for solving many kinds of problems (not just sorting). It consists of the following steps:

1. Divide your work up into smaller pieces (recursively)
2. Conquer the individual pieces (as base cases)
3. Combine the results together (recursively)

```
algorithm(input) {
    if (small enough) {
        CONQUER, solve, and return input
    } else {
        DIVIDE input into multiple pieces
        RECURSE on each piece
        COMBINE and return results
    }
}
```
Two great sorting methods are fundamentally divide-and-conquer.

**Mergesort:**
- Sort the left half of the elements (recursively)
- Sort the right half of the elements (recursively)
- Merge the two sorted halves into a sorted whole

**Quicksort:**
- Pick a “pivot” element
- Divide elements into less-than pivot and greater-than pivot
- Sort the two divisions (recursively on each)
- Answer is: sorted-less-than....pivot....sorted-greater-than