Warmup

buildHeap([8, 2, 4, 0, 1, 12, 3, 5 6])
Comparison Sorts

Compare two elements at a time

General sort, works for most types of elements

Element must form a “consistent, total ordering”

For every element a, b and c in the list the following must be true:

- If a \leq b and b \leq a then a = b
- If a \leq b and b \leq c then a \leq c
- Either a \leq b is true or \leq a

What does this mean? `compareTo()` works for your elements

Comparison sorts run at fastest \( O(n\log(n)) \) time

Niche Sorts aka “linear sorts”

Leverages specific properties about the items in the list to achieve faster runtimes

niche sorts typically run \( O(n) \) time

In this class we’ll focus on comparison sorts
In Place sort
A sorting algorithm is in-place if it requires only $O(1)$ extra space to sort the array
Typically modifies the input collection
Useful to minimize memory usage

Stable sort
A sorting algorithm is stable if any equal items remain in the same relative order before and after the sort

Why do we care?
- Sometimes we want to sort based on some, but not all attributes of an item
- Items that "compareTo" the same might not be exact duplicates
- Enables us to sort on one attribute first then another etc...

[(8, "fox"), (9, "dog"), (4, "wolf"), (8, "cow")]

[(4, "wolf"), (8, "fox"), (8, "cow"), (9, "dog")]

Stable

[(4, "wolf"), (8, "cow"), (8, "fox"), (9, "dog")]

Unstable

Other Considerations
Worst Case, Average Case
External Sorts (can’t fit in memory)
Good for small data sets
Ease of implementation
LinkedList vs Array
Selection Sort

Basic Idea:

Repeatedly scan through the list, moving the next smallest element to the front.

This sounds a lot like heap sort, but worse...
Selection Sort

Sorted Items

Unsorted Items

Current Item

https://www.youtube.com/watch?v=Ns4TPTC8whw
Selection Sort

```java
public void selectionSort(collection) {
    for (entire list)
        int newIndex = findNextMin(currentItem);
        swap(newIndex, currentItem);
}

public int findNextMin(currentItem) {
    min = currentItem
    for (unsorted list)
        if (item < min)
            min = currentItem
    return min
}

public int swap(newIndex, currentItem) {
    temp = currentItem
    currentItem = newIndex
    newIndex = currentItem
}
```

Worst case runtime?   O(n^2)
Best case runtime?     O(n^2)
Average runtime?       O(n^2)
Stable?                Yes
In-place?              Yes
Insertion Sort

Basic Idea:

Like you would sort a hand of cards – pull out the next card, then insert it into it where it belongs

https://www.youtube.com/watch?v=ROalU379I3U
Insertion Sort

https://www.youtube.com/watch?v=ROalU379l3U
Insertion Sort

public void insertionSort(collection) {
    for (entire list)
        if (currentItem is bigger than nextItem)
            int newIndex = findSpot(currentItem);
            shift(newIndex, currentItem);
}
public int findSpot(currentItem) {
    for (sorted list)
        if (spot found) return
}
public void shift(newIndex, currentItem) {
    for (i = currentItem > newIndex)
        item[i+1] = item[i]
        item[newIndex] = currentItem
}
Heap Sort

1. run Floyd’s buildHeap on your data
2. call removeMin n times

```java
public void heapSort(collection) {
    E[] heap = buildHeap(collection)
    E[] output = new E[n]
    for (n)
        output[i] = removeMin(heap)
}
```

- Worst case runtime? $O(n\log n)$
- Best case runtime? $O(n\log n)$
- Average runtime? $O(n\log n)$
- Stable? No
- In-place? No

https://www.youtube.com/watch?v=Xw2D9aJRBY4
In Place Heap Sort

Heap

Sorted Items

Current Item

Heap

Sorted Items

Current Item

Heap

Sorted Items

Current Item
public void inPlaceHeapSort(collection) {
    E[] heap = buildHeap(collection)
    for (n)
        output[n - i - 1] = removeMin(heap)
}

Complication: final array is reversed!
- Run reverse afterwards (O(n))
- Use a max heap
- Reverse compare function to emulate max heap

Worst case runtime? O(nlogn)
Best case runtime? O(nlogn)
Average runtime? O(nlogn)
Stable? No
In-place? Yes
Divide and Conquer Technique

1. Divide your work into smaller pieces recursively
   - Pieces should be smaller versions of the larger problem

2. Conquer the individual pieces
   - Base case!

3. Combine the results back up recursively

```java
divideAndConquer(input) {
    if (small enough to solve)
        conquer, solve, return results
    else
        divide input into a smaller pieces
        recurse on smaller piece
        combine results and return
}
```
Merge Sort

https://www.youtube.com/watch?v=XaqR3G_NVoo

Divide

Conquer

Combine
Merge Sort

mergeSort(input) {
    if (input.length == 1)
        return
    else
        smallerHalf = mergeSort(new [0, ..., mid])
        largerHalf = mergeSort(new [mid + 1, ...])
    return merge(smallerHalf, largerHalf)
}

Worst case runtime? \( O(n \log n) \)
Best case runtime? \( T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases} \)
Average runtime?
Stable? Yes
In-place? No

Steak = \( \frac{n}{2^k} \) work per node

work per layer = \( 2^l \cdot \frac{n}{2^k} \) work per node

CSE 373 SP 18 - KASEY CHAMPION
Merge Sort Optimization

Use just two arrays – swap between them
Quick Sort

Divide

Conquer

Combine

https://www.youtube.com/watch?v=ywWB6J5gz8
Quick Sort

```java
quickSort(input) {
    if (input.length == 1)
        return
    else
        pivot = getPivot(input)
        smallerHalf = quickSort(getSmaller(pivot, input))
        largerHalf = quickSort(getBigger(pivot, input))
    return smallerHalf + pivot + largerHalf
}
```

Worst case runtime? $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n + T(n - 1) & \text{otherwise} \end{cases}$

Best case runtime? $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n + 2T(n/2) & \text{otherwise} \end{cases}$

Average runtime? $T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n + 2T(n/2) & \text{otherwise} \end{cases}$

Stable? No

In-place? No
Can we do better?

Pick a better pivot
- Pick a random number
- Pick the median of the first, middle and last element

Sort elements by swapping around pivot in place
Better Quick Sort

Low
X < 6

High
X >= 6

Low
X < 6

High
X >= 6
Announcements

Project 1 (Calculator) is Due Tonight! Use “SUBMIT” as tag.

HW3 (Individual Assignment) will be assigned this weekend
- Due Sunday 7/22
- Make sure you know how to do it before the midterm! It’s the best midterm review

Come to Class Next Week:
- Monday: Midterm Review – Going from Diagrams to Code
- Wednesday: Software Engineering – Deep Dive into Git, Pair Programming, and Testing
- Friday: Midterm Exam! – Review materials and practice midterms on website (this evening).
More Announcements

If you are applying to the CSE major, send me an e-mail reminding me of our interactions.

Office hours immediately after class – follow me!