Lecture 22: Introduction to Sorting II
Warm Up

What sorting algorithm do the following steps represent. The steps are not necessarily consecutive but they are in the correct sequence.

Sort 1

[23, 37, 48, 34, 11, 34, 37, 34, 23, 39, 41, 47]
[11, 37, 48, 34, 23, 34, 37, 34, 23, 39, 41, 47]
[11, 23, 24, 34, 48, 34, 37, 34, 44, 39, 41, 47]
[11, 23, 24, 34, 34, 37, 48, 44, 37, 39, 41, 47]

Sort 2

[2, 27, 18, 12, 14, 43, 8, 5, 41, 32, 48, 10, 37]
[2, 27, 12, 18, 14, 43, 8, 5, 41, 32, 48, 10, 37]
[2, 8, 12, 14, 18, 27, 43, 5, 41, 32, 48, 10, 37]
[2, 8, 12, 14, 18, 27, 43, 5, 10, 32, 37, 41, 48]

https://visualgo.net/en/sorting
Announcements

Final Exam Friday May 26th in class

Topics

Classes Week 5 - Week 8

● Heaps
● Graphs
  ○ Graph Modeling
  ○ BFS/DFS
  ○ Topological Sort
  ○ Dijkstra’s
  ○ MSTs
● Disjoint Sets
● Sorting Algorithms
Intro to Sorting
Selection Sort
Insertion Sort
Merge Sort
Quick Sort
Divide and Conquer

There’s more than one way to divide!

**Mergesort**
- Split into two arrays.
- Elements that just happened to be on the left and that happened to be on the right.

**Quicksort**
- Split into two arrays.
- Roughly, elements that are “small” and elements that are “large”
- How to define “small” and “large”? Choose a “pivot” value in the array that will partition the two arrays!
Quick Sort (v1)

Choose a “pivot” element, partition array relative to it!

Again, no extra conquer step needed!

Simply concatenate the now-sorted arrays!

https://www.youtube.com/watch?v=ywWBvJ5gz8
Quick Sort (v1): Divide Step

Recursive Case:
- Choose a “pivot” element
- Partition: linear scan through array, add smaller elements to one array and larger elements to another
- Recursively partition

Base Case:
- When array hits size 1, stop dividing
Quick Sort (v1): Combine Step

Simply concatenate the arrays that were created earlier!
Partition step already left them in order 😊
Quick Sort (v1)

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

Worst case runtime? \( T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
T(n - 1) + n & \text{otherwise}
\end{cases} = \Theta(n^2) \)

Best case runtime? \( T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
2T\left(\frac{n}{2}\right) + n & \text{otherwise}
\end{cases} = \Theta(n \log n) \)

In-practice runtime? Just trust me: \( \Theta(n \log n) \) (absurd amount of math to get here)

Stable? No

In-place? Can be done!

Useful for: Fast sorting of primitives! (This is what Java uses for Primitives)
Can we do better?

How to avoid hitting the worst case?
- It all comes down to the pivot. If the pivot divides each array in half, we get better behavior

Here are four options for finding a pivot. What are the tradeoffs?
- Just take the first element
- Take the median of the full array
- Take the median of the first, last, and middle element
- Pick a random element
Strategies for Choosing a Pivot

Just take the first element
- Very fast!
- But has worst case: for example, sorted lists have $\Omega(n^2)$ behavior

Take the median of the full array
- Can actually find the median in $O(n)$ time (google QuickSelect). It’s complicated
- $O(n \log n)$ even in the worst case... but the constant factors are awful. No one does quicksort this way.

Take the median of the first, last, and middle element
- Makes pivot slightly more content-aware, at least won’t select very smallest/largest
- Worst case is still $\Omega(n^2)$, but on real-world data tends to perform well!

Pick a random element
- Get $O(n \log n)$ runtime with probability at least $1-1/n^2$
- No simple worst-case input (e.g. sorted, reverse sorted)
Quick Sort (v2: In-Place)

Select a pivot

Move pivot out of the way

Bring low and high pointers together, swapping elements if needed

Meeting point is where pivot belongs; swap in. Now recurse on smaller portions of same array!
Heap Sort
Bucket Sort
Radix Sort
Sorting Summary

INEFFECTIVE SORTS

**INDEFINITE MERGESORT**

```
DEFINE HIERARCHICAL-MERGE-SORT(List):
    IF LENGTH(List) < 2:
        RETURN List
    Pivot = INT(LENGTH(List) / 2)
    A = HIERARCHICAL-MERGE-SORT(List[:Pivot])
    B = HIERARCHICAL-MERGE-SORT(List[Pivot:])
    RETURN [A, B] // HERE, SORRY.
```

**INDEFINITE Bishopsort**

```
DEFINE FIRST-BISHOPSORT(List):
    // AN OPTIMIZED BISHOPSORT
    // RAND IN 0(NXN)
    FOR N FROM 1 TO LOG2(LENGTH(List)):
        SHUFFLE(List)
        IF ISORTED(List):
            RETURN List
        RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

**INDEFINITE JOHNSON-SORT**

```
DEFINE JOHNSON-SORT(List):
    OK, SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
        NO MATTER WHAT IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
        THE BIGGER ONES GO IN A NEW LIST
        THE SMALLER ONES GO INTO LH, UN
        THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
        THIS IS LH
        THE NEW ONE IS LIST RH
        PUT THE BIG ONES INTO LIST RH
        NOW TAKE THE SECOND LIST
        CALL IT List Rh, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
        RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
        AM I ALLOWED TO USE THE STANDARD LIBRARIES
```

**INDEFINITE FRANK-SORT**

```
DEFINE FRANK-SORT(List):
    IF ISORTED(List):
        RETURN List
    FOR N FROM 1 TO 10000:
        Pivot = RANDOM([1, LENGTH(List)])
        List = List[:Pivot] + List[:Pivot]
    IF ISORTED(List):
        RETURN List
    IF ISORTED(List):
        RETURN LIST
    IF ISORTED(List): // THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISORTED(List): // COME ON, COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    List = []
    SYSTEM("SHUTDOWN -- H.S")
    SYSTEM("RM -RF /")
    SYSTEM("RM -RF /")
    SYSTEM("RD AS 10 00+V") // PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```
Heap Sort

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

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

Worst case runtime? \( \Theta(n \log n) \)

Best case runtime? \( \Theta(n) \)

In-practice runtime? \( \Theta(n \log n) \)

Stable? No

In-place? If we get clever…

https://www.youtube.com/watch?v=Xw2D9aJRBY4
Selection sort:
After \( k \) iterations of the loop, the \( k \) smallest elements of the array are (sorted) in indices 0, \ldots, k-1
Runs in \( \Theta(n^2) \) time no matter what

Using data structures
● Speed up our existing ideas

If only we had a data structure that was good at getting the smallest item remaining in our dataset...
● We do!
In Place Heap Sort

Current Item:

percolateDown(22)

Heap

Sorted Items

Current Item:

Heap

Sorted Items

Current Item:

Heap

Sorted Items
In Place Heap Sort

```java
public void inPlaceHeapSort(input) {
    buildHeap(input) // alters original array
    for (n : input)
        input[n - i - 1] = removeMin(heap)
}
```

Worst case runtime? \(\Theta(n \log n)\)
Best case runtime? \(\Theta(n)\)
In-practice runtime? \(\Theta(n \log n)\)
Stable? No
In-place? Yes

Complication: final array is reversed! Lots of fixes:
- Run reverse afterwards \(O(n)\)
- Use a max heap
- Reverse compare function to emulate max heap
Heap Sort
Bucket Sort
Radix Sort
Sorting Summary
Bucket Sort (aka Bin Sort)

- If all values are ints known to be in the range of 1 - K
- Create array of size K and put each element in its proper bucket (“scatter”)
  - If elements are only ints simply store count of ints in each bucket
- Output results via linear pass through array of buckets (“gather”)

```
[5, 1, 3, 4, 3, 2, 1, 1, 5, 4, 5]
```

```
1 3
2 1
3 2
4 4
5 3
```

```
[1, 1, 1, 2, 3, 3, 4, 4, 5, 5, 5]
```

**Total Runtime: O(K + n)**
Bucket Sort with Data

- Instead of using int counts, make buckets of array of lists
- put items into bucket, use **insertion sort** to sort individual buckets

\[0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68\]

\[
\begin{array}{c|c|c|c|c}
0 & 0.17 & 0.12 & 0.12 & 0.17 \\
1 & 0.26 & 0.21 & 0.23 & 0.26 \\
2 & 0.39 & 0.39 & 0.39 & 0.39 \\
3 & 0.68 & 0.68 & 0.68 & 0.68 \\
4 & 0.78 & 0.72 & 0.72 & 0.72 \\
5 & 0.89 & 0.94 & 0.94 & 0.94 \\
\end{array}
\]

- best: \(O(K)\)
- worst: \(O(K + n^2)\)
- \(O(n)\)
## Bucket Sort

**Function**

```plaintext
function bucketSort(array, k) is
    buckets ← new array of k empty lists
    M ← 1 + the maximum key value in the array
    for i = 0 to length(array) do
        insert array[i] into buckets[floor(k × array[i] / M)]
    for i = 0 to k do
        nextSort(buckets[i])
    return the concatenation of buckets[0], ...., buckets[k]
```

<table>
<thead>
<tr>
<th>Worst case runtime?</th>
<th>O(K + n) for ints</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>O(K + n^2) for data if insertion sort is used</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Best case runtime?</th>
<th>O(n)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>In-practice runtime?</th>
<th>O(n) if K ≈ n, always for ints, and if values are evenly distributed for data</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Stable?</th>
<th>Can be because insertion sort</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>In-place?</th>
<th>No</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Useful for:</th>
<th>When range, K, is smaller or not much larger than n (not many duplicates)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Not good when K &gt;&gt; N, wasted space</td>
</tr>
</tbody>
</table>
Heap Sort
Bucket Sort
Radix Sort
Sorting Summary
Moving away from comparison sorts

So far we’ve learned about comparison sorts

- work on any comparable object
- have a best case lower bound of $\Omega(n\log n)$

This is because to sort using comparisons requires all elements to be compared against one another

- $n$ runtime to process all values into some ordered structure (tree)
- $\log n$ runtime to remove items from structure in sorted order

What if we didn’t need to compare each element, what if we built a sort based on inherent knowledge about the ordering of specific data types ie numbers
Specialized Sorts ("Niche Sorts")

Sorting algorithms that only work on data types with ordering already known to computer logic: numbers

- Bucket Sort for ints
- Radix Sort
Radix Sort

- **Radix** = “the base of a number system”
  - We will use “10” as we are comfortable with 10 based systems
  - Could use any value, such as 128 for ASCII strings
- **Idea**
  - Bucket sort on one digit at a time
    - Only works on sequences of countable data: ints, doubles, strings
  - Number of buckets = radix
  - Start with least significant digit, do one pass of bucket sort per digit
- **Fun fact:** invented in 1890 as part of US census

Input: [170, 45, 75, 90, 802, 24, 2, 66]

- **ones:** [170, 90, 802, 2, 24, 45, 75, 66]
- **tens:** [802, 2, 24, 45, 66, 170, 75, 90]
- **hundreds:** [2, 24, 45, 66, 75, 90, 170, 802]
# Radix Sort

We consider the problem of sorting a list of integers. Radix sort is a non-comparative sorting algorithm that sorts the numbers based on the digits of the numbers. The basic idea is to sort elements according to the individual digits, starting from the least significant digit (LSD) to the most significant digit (MSD).

**Example:**

Let's consider the list of numbers: [478, 537, 9, 721, 3, 38, 143, 67].

1. **Initial List:** [478, 537, 9, 721, 3, 38, 143, 67]
2. **First Pass:** Sort on the least significant digit (9s place). The list becomes: [721, 3, 143, 537, 67, 478, 38, 9]
3. **Second Pass:** Sort on the next significant digit (8s place). The list becomes: [3, 9, 38, 67, 143, 478, 537, 721]

**Time Complexity:**

- First Pass: \(O(n)\)
- Second Pass: \(O(n)\)
- Third Pass: \(O(n)\)
- Total: \(O(n)\)
Radix Sort

Worst case runtime? \( O(n) \)
Best case runtime? \( O(n) \)
In-practice runtime? \( O(n) \)
Stable? Yes
In-place? No
Useful for: Sorting ints
Heap Sort
Bucket Sort
Radix Sort

Sorting Summary
Sorting: Summary

<table>
<thead>
<tr>
<th></th>
<th>Best-Case</th>
<th>Worst-Case</th>
<th>Space</th>
<th>Stable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>No</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
<td>No</td>
</tr>
<tr>
<td>In-Place Heap Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(1)$</td>
<td>No</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>No</td>
</tr>
<tr>
<td>In-place Quick Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td>No</td>
</tr>
<tr>
<td>Bucket Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(K+n)$</td>
<td>Yes</td>
</tr>
<tr>
<td>Radix</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>Yes</td>
</tr>
</tbody>
</table>

What does Java do?

- Actually uses a combination of 3 different sorts:
  - If objects: use Merge Sort* (stable!)
  - If primitives: use Dual Pivot Quick Sort
  - If “reasonably short” array of primitives: use Insertion Sort
    - Researchers say 48 elements

Key Takeaway: No single sorting algorithm is “the best”!

- Different sorts have different properties in different situations
- The “best sort” is one that is well-suited to your data

* They actually use Tim Sort, which is very similar to Merge Sort in theory, but has some minor details different
What Else is There?

Can we do better than \( n \log n \)?

- For comparison sorts, **NO**. A proven lower bound!
  - Intuition: \( n \) elements to sort, no faster way to find “right place” than \( \log n \)
- However, niche sorts can do better in specific situations!

Many cool niche sorts beyond the scope of 373!
- Counting Sort ([Wikipedia](https://en.wikipedia.org/wiki/Counting_sort))
Questions?
That’s all!