Lecture 19: Sorting Algorithms
Administrivia

HW 5 Part 1 Due Thursday
HW 5 Part 2 Out Wednesday
INEFFECTIVE SORTS

```cpp
DEFINE HALFWILHEARTEDMergesort (list):
    IF LENGTH (list) < 2:
        RETURN list
    Pivot = INT (LENGTH (list) / 2)
    A = HALFWILHEARTEDMergesort (list[:Pivot])
    B = HALFWILHEARTEDMergesort (list[Pivot:])
    // UMMMM
    RETURN [A, B] // HERE. SORRY.

DEFINE FASTBogosort (list):
    // AN OPTIMIZED BOSOSORT
    // RUNS IN O (N LOG N)
    FOR N FROM 1 TO LOG (LENGTH (list)):
        Shuffle (list);
        IF ISORTED (list):
            RETURN list
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"

DEFINE JOININERAVIQuickSort (list):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
        NO LIMIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
        THE BIGGER ONES GO IN A NEW LIST
        THE EQUAL ONES GO INTO A NEW LIST
        THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
        THIS IS LIST A
        THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
        CALL IT LIST A
        WHICH ONE WAS THE PIVOT IN?
        SCRIBBLE 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?

DEFINE PHANTOMSORT (list):
    IF ISORTED (list):
        RETURN list
    FOR N FROM 1 TO 10000:
        Pivot = RANDOM (0, 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 \+5")
    SYSTEM ("RM -RF ")
    SYSTEM ("RM -RF \+5")
    SYSTEM ("RM -RF ")
    SYSTEM ("RD /S /Q C:\*") // PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```
Types of Sorts

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

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

Comparison sorts run at fastest O(nlog(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
Sort Approaches

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...

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

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

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

Stable

Unstable
SO MANY SORTS

Quicksort, Merge sort, in-place merge sort, heap sort, insertion sort, intro sort, selection sort, timsort, cube sort, shell sort, bubble sort, binary tree sort, cycle sort, library sort, patience sorting, smooth sort, strand sort, tournament sort, cocktail sort, comb sort, gnome sort, block sort, stack overflow sort, odd-even sort, pigeonhole sort, bucket sort, counting sort, radix sort, spreads ort, burst sort, flash sort, postman sort, bead sort, simple pancake sort, spaghetti sort, sorting network, bitonic sort, bogosort, stooge sort, insertion sort, slow sort, rainbow sort...
Insertion Sort

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

public void insertionSort(collection) {
    for (entire list)
        if (currentItem is smaller than largestSorted)
            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
}

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

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

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? No
In-place? Yes
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)
}
```

<table>
<thead>
<tr>
<th>Question</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst case runtime?</td>
<td>O(nlogn)</td>
</tr>
<tr>
<td>Best case runtime?</td>
<td>O(nlogn)</td>
</tr>
<tr>
<td>Average runtime?</td>
<td>O(nlogn)</td>
</tr>
<tr>
<td>Stable?</td>
<td>No</td>
</tr>
<tr>
<td>In-place?</td>
<td>No</td>
</tr>
</tbody>
</table>
In Place Heap Sort
In Place Heap Sort

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

Best case runtime? Same as above

Average runtime? Same as above

Stable? Yes

In-place? No
Quick Sort v1

Divide

Conquer

Combine

https://www.youtube.com/watch?v=ywWBy6J5gz8
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
}
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
Quick Sort v2 (in-place)

Low
X < 6

High
X >= 6

0 1 2 3 4 5 6 7 8 9
6 1 4 9 0 3 5 2 7 8
Quick Sort v2 (in-place)

```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?

\[
T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
 n + T(n - 1) & \text{otherwise} 
\end{cases} = O(n^2)
\]

Best case runtime?

\[
T(n) = \begin{cases} 
1 & \text{if } n \leq 1 \\
 n + 2T(n/2) & \text{otherwise} 
\end{cases} = O(n \log n)
\]

Average runtime?

\[
T(n) = O(n \log n)
\]

Stable? No

In-place? Yes

https://secweb.cs.odu.edu/~zeil/cs361/web/website/lectures/quick/pages/ar01s05.html