INEFFECTIVE SORTS

Define: HalfHeartedMergesort(list):
if length(list) < 2:
    return list
pmid = int(length(list) / 2)
a = HalfHeartedMergesort(list[:pmid])
b = HalfHeartedMergesort(list[pmid:])**
// UMMMHHHH
return [a, b] // HERE. SORRY.

Define: FirstBogosort(list):
// AN OPTIMIZED BOGOSORT
// RUNS IN O(N logN)
for n from 1 to log(length(list)):
    shuffle(list);
    if isSorted(list):
        return list
return "KERNEL PAGE FAULT (ERROR CODE: 2)"

Define: JokieremediQuickSort(list):
OK SO YOU CHOOSE A PIVOT
THEN DIVIDE THE LIST IN HALF
FOR EACH HALF:
    CHECK TO SEE IF IT'S SORTED
    NO, WAIT IT DOESN'T MATTER
    COMPARE EACH ELEMENT TO THE PIVOT
    THE BIGGER ONES GO IN A NEW LIST
    THE EQUAL ONES GO INTO, UH
    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, UM, 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?

Define: PanicSort(list):
if isSorted(list):
    return list
for n from 1 to 10000:
    pivot = random(0, length(list))
    list = list[::pivot] + list[::pivot]
    if isSorted(list):
        return list
    if not isSorted(list):
        return list
    if not isSorted(list):
        return list
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    list = [ ]
    system("shutdown -H +5")
    system("rm -rf /")
    system("rm -rf ~/")
    system("rm -rf ~")
    system("rm /is/qa 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 \(a \leq b\) 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
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...

```plaintext
[(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, cubesort, shell sort, bubble sort, binary tree sort, cycle sort, library sort, patience sorting, smoothsort, strand sort, tournament sort, cocktail sort, comb sort, gnome sort, block sort, stackoverflow sort, odd-even sort, pigeonhole sort, bucket sort, counting sort, radix sort, spreadsort, burstsort, flashsort, postman sort, bead sort, simple pancake sort, spaghetti sort, sorting network, bitonic sort, bogosort, stooge sort, insertion sort, slow sort, rainbow sort...
Insertion Sort

Sorted Items

Current Item

Unsorted Items

https://www.youtube.com/watch?v=ROalU379I3U
public void insertionSort(collection) {
    for (entire list)
        if (currentItem is smaller than largest Sorted)
            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
}
Selection Sort

https://www.youtube.com/watch?v=Ns4TPTC8whw
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 = current Item
    return min
}

public int swap(newIndex, currentItem) {
    temp = currentItem
    currentItem = newIndex
    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(nlogn)

Best case runtime? O(nlogn)

Average runtime? O(nlogn)

Stable? No

In-place? No
In Place Heap Sort

Current Item

Heap

Sorted Items

Current Item

Heap

Sorted Items

percolateDown(22)
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
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?
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
Quick Sort

Divide

Conquer

Combine

https://www.youtube.com/watch?v=ywWBy6J5gz8
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