Recursive Sorting Algorithms

Mergesort and Quicksort

Recap

- Swapping adjacent elements requires $\Omega(N^2)$ swaps, worst case
- We need Omega(N) swaps, the other operation we need to estimate is comparisons

Comparisons

- We counted swaps to come up with (tight) lower bounds on the runtimes of bubble sort and insertion sort
- Selection sort performed MUCH worse than the number of swaps (O(N) swaps but O(N^2) runtime
- Selection sort spent most of it's runtime performing comparisons.
- For any algorithm, it must take at least as long to run as the number of comparisons it does
- How many comparisons do we need to do?

Alternative history

- I'm pretty sure this is NOT how history went, but we're going to do the analysis first, then use that to design an algorithm
- While that's probably not what happened from these 2 sorting algorithms it IS a useful tool for narrowing down what kind of algorithms are feasible/optimal
- The analysis is a bit complex (I first saw it in my jr level algorithms class) and I'm going to be hand-wavy, so hopefully you can just go with the “vibes” and don't worry about the details
- The key idea is that we have to know how our elements are out of order so that we can do the correct swaps. How many comparions do we need to figure that out?

Simple example

- Suppose we have an array with the value 1, 2, 3, 4, chosen just to make illustration a bit easier
- If we compare the first and second elements, how much do we learn about the elements in the array?
- If the first is smaller than the second, these are the possible orderings (we just know that we have one of them): [1,2,3,4], [1,2,4,3], [1,3,2,4], [1,3,4,2], [1,4,2,3], [1,4,3,2], [2,3,1,4], [2,3,4,1],[2,4,1,3],[2,4,3,1],[3,4,1,2],[3,4,2,1]
- That's 12 possibilities out of 24 possible orderings, so we've narrowed things by half.

Continued

- Now let's say we find that the 2nd element is less than the 3rd element, now we've narrowed the order down to: [1,2,3,4], [1,2,4,3], [1,3,4,2], [2,3,4,1]
- If we find that the first element is greater than the last, then we've used 4 comparisons to determine that the array was in the order [2,3,4,1]
- Each time we did a comparison, we eliminated some possible orderings although that's not alway guaranteed. Comparing the 1st and 3rd elements in the last step wouldn't have given us any new information

Generalizing

- Any “comparison based” sorting algorithm must perform enough comparisons, no matter what the input is to exactly determine the original order of the elements (this is done indirectly in real algorithms)
- So how many comparisons does that take to work on ALL given inputs?
- We'll draw a picture called a tree (which we'll be discussing in depth later)
- Each comparions “splits” the remaining possibilities into two pieces, one containing the orders where the comparison is true, the other where the comparison is false
- We have to keep doing comparisons as long as one of those sets has more than 1 input in it

Computing the # of comparisons

- The “worst case” number of comparions is the number of comparisons we have to go through to get from the top of the tree to any of the bottom nodes
- Each algorithm will perform different comparisons in different orders, but we can actually analyze all possible algorithms in one shot!
- How many possible orders are there of N elements? N factorial ($N * (N-1) * (N-2)$...), so that's the # of “bottom” nodes
- The way to make the tree as “short” as possible is to choose comparisons that always split the possiblities exactly in half.
- If we start with N! possibilities, how many times do we need to split them in half until we're left with 1?

The result

- We've seen this scenario before, we can split X in half lg(X) times before we get to 1, so let's do a little algebra
- $$lg(N!) = lg(N * (N-1) * (N-2)\dots) = lg(N) + lg(N-1) + lg(N$$
- We have N terms all approximately lg(N) so the total number of steps is O(NlgN)
- Here's the full statement of what we just showed:
- For any sorting algorithm that works by comparing pairs of elements, for some inputs, it must perform O(NlgN) comparisons
- Note, it can be faster than that on some inputs, but there will be some that take at least N lg N time **FOR ANY ALGORITHM**

Inspiration

- Knowing that there's a “speed limit” on sorting algorithms, let's try to think of one that would have an NlgN runtime
- The lgN term comes up in analysis when we repeatedly split something originally sized N in half
- We'll look at “divide and conquer” algorithms which splits the array, sorts the pieces, then combines the results somehow.
- We'll start with a sort of obvious recursive algorithm: sort the left and right halves of the array independently then somehow combine the results into the big sorted array.

Working Backwards

- Assume we have 2 sorted arrays and want to combine them into one
- How long does that take?
- What other resource do we require?
- How can we build a sorting algorithm out of this?

Merging:

- Conceptually simple, but some tricky bits
- Arrays of of different sizes
- Need extra storage

Merging:

```
// start→ mid is sorted, mid→ is sorted
merge(arr, start, mid, end):
    // create temp array for holding merged arr
    int[] temp = array of size end - start
    int i1 = start, i2 = mid;
    while(i1 < mid && i2 < end):
        put smaller of arr[i1], arr[i2] into temp
    copy anything left over from larger half to temp
    return temp
```

Merging → Mergesort

- If we have 2 arrays we can merge them efficiently
- How do we get 2 sorted sub arrays? Sort the two halves of our array
- To sort our array, we sort 2 smaller arrays and then merge them
- To sort those smaller arrays we need to sort 2 even smaller arrays and merge them
- When do we stop?

Merge sort

```
mergeSort(arr, start, stop):
    if stop - start < 2: return //it's sorted

    mergeSort(arr, start, (start + stop)/2)
    mergeSort(arr, (start + stop)/2, stop)
    //two halves are sorted, merge them
    merge(arr, start, (start + stop)/2, stop)
```

Tricky Bit: Space to merge

- Merging is efficient/simple when we have an extra array to store the merged result into
- Allocate a length N array in the driver method and pass it around so the recursive calls can use it

Analysis: Runtime of merge sort

- Let's call the runtime F(N)
- F(N) = O(N) to merge + 2* F(N/2) for 2 recursive calls
- F(N/2) = O(N/2) + 2*(F/N4)
- and so on
- Basically we have to merge M arrays or size N/M at each level, where M are powers of 2
- Since M* N/M = N, we're doing O(N) on each “Level”
- The number of levels is the number of times we have to split before we hit our base case: N → N/2 → N/4 → ... 1
- We've seen this pattern several times now

Divide and conquer

- MergeSort is an example of a “divide and conquer” algorithm
- We divide the problem into smaller and smaller subproblems until we can solve them trivially or at least efficiently
- Then we combine the results from the smaller problems into the final result
- D&C algorithms often implented recursively

Big O and small numbers

- Big O tells us what happens when N is really large
- If N is small, not very informative
- 10*lg 10 = 33 (expected merge sort runtime)
- $10^2/4 = 25$ (expecte insertion sort runtime)

New base case

- For merge sort, if the partial array is “small enough” it's faster to insertion-sort the small array
- Exact threshold depends on the hardware, size of the thing you're sorting, etc
- So experiment + measure

Quicksort

- Merge sort does split, sort, merge, but the split part is done naively
- Quicksort does the splitting part carefully, and so doesn't need the merge step
- Uses a new helper function called “partition”

Partition

- Partition takes an array and a “pivot value” (in quicksort this is one of the values in the array) as parameters
- It moves elements in the array so that all elements smaller than the pivot come before it in the array and all elements larger come after
- It returns the index of the pivot after partitioning. In general, we don't know where the pivot will end up until we finish partitioning, and the caller needs to know where it is

Quicksort

- Partition the array around a pivot (we'll see a few ways to pick pivots)
- Sort the part before the pivot
- Sort the part after the pivot

Implementing Partition

- Can be done “in place” (ie no need to allocate another array, like merge sort does)
- Several possible implementations, we'll look at a “two sided” version

Partition

```
Swap the pivot to the end of the array
left = 0, right = array end
while left < right:
    while arr[left] < pivot: advance left
    //left points to an element on wrong side of the pivot
    while arr[right] > pivot: decrement right
    //right points to an element on the wrong side of the pivot
    swap left/right
swap the pivot with left
```

Analyzing Complexity

- Partition is O(N)
- So cost of quicksort is O(N) + O(sort left) + O(sort right)
- A little bit tricky to analyze since we don't know how big left/right are!
- Not even obvious what best/worse cases are!

Best Case: even splits

- In the best case, our pivot is the median value and left/right sides each have N/2 elements
- In this case analysis is similar to mergesort (partition ~= merge, recursive sorts are the same in both analyses)
- Gives us O(N lg N) runtime
- If pivots are “usually” 25/75 splits or “better” we get (N lg N) complexity. Take an algorithms class for the details

Worst Case: bad splits

- In the worst case, our partition doesn't divide the array at all
- An example is when the left half is empty and the right half is N −1 elements
- In this case the cost at the “top level” is N, the next level is N - 1, then N −2, etc
- So our overall run time is N + (N −1) + (N −2) + ... 1
- Which we know is $O(N^2)$
- This is a very common input!!!

Choices of Pivot

- First element (uh oh!)
- Last element (uh oh!)
- Middle element (probably OK)
- Random element (almost certainly OK)
- Is there a choice that's guaranteed to be good enough? Stay tuned