Sorting Algorithms

Review

- We'll start by reviewing + analyizing some sorts we've already seen
- Then we'll see why they are necessarily bad
- Then we'll come up with a better sort
- On monday, we'll see 2 MUCH better sorts

Sort 1: Bubble sort

```
do
    done = true
    for i = 0 to N -1
        if arr[i], arr[i+1] are in the wrong order
            swap them
            done = false
while not done
```

Sort 2: Selection Sort

```
for i = 0 to N
    min = i
    for j = i to N:
        if arr[j] < arr[min]:
            min = j
    swap arr[i], arr[min]
```

Sort 3: Insertion Sort

```
for i = 1 to N:
    for j = i; j > 0 && arr[j] < arr[j -1]; j--
        swap arr[j], arr[j-1]
```

Overall analysis

- Bubble sort and insertion sort work by swapping adjacent elements
- One way of thinking about complexity is considering how many swaps are necessary
- Our algorithm might be slower than that (we'll see that), but it's a lower bound on how well we can do

Measuring “Unsortedness”

- For each pair of elements in the array (how many pairs of elements are there?), if they're in the wrong order relative to each other, we count that as an “inversion”
- How many inversions can there be?
- Upper bound?
- Lower bound?
- Expected Average?

Algorithm analysis

- Insertion + Bubble sort “undo” one inversion per swap, so their runtime is Omega(numInversions) (big Omega, because it takes at LEAST that many operations)
- For insertion sort this is a tight bound since we don't do any significant work when there are no inversions
- For bubble sort, the true cost can be much higher!
- For selection sort the true cost is ALWAYS much higher!

Good idea in selection sort

- Swapping non adjacent elements means our runtime isn't necessarily tied to the number of inversions!
- Swapping far elements can eliminate many inversions in one shot
- Selection sort is bad because we need to find a good pair to swap, and that takes a long time
- Can we apply this good idea to insertion sort?

Shell sort (Named after Don Shell, not a shape/sea creature)

- Basically insertion sort, but we try to swap elements over large distances

```
for gapSize = N/2, N/4,N/8,... 1
    for i = gapSize; i < N; i++
        for j = i; j > gapSize && arr[j] < arr[j - gapSize]; j -= gapSize
            swap arr[j], arr[j - gapSize]
```

Analysis of Shell sort

- The gapSize and i loops run log N, and O(N) times, so the runtime is basically O(N lg N * cost of the j loop)
- This is tricky to prove, but in worst case, the overall cost is ~ $O(N^2)$ as written
- The average case as written is $O(N^3/2)$ = O(N*Sqrt(N)) which is worse than O(N Lg N) but better than quadratic
- You can reduce the avg case exponent by changing how you pick the gap sizes

Foreshadowing

- All the sorts we've seen are “comparison” sort algorithms which compare 2 elements and then perform some swaps
- The lower bound on any comparison sort based algorithm is Omega(N lg N)
- Can we find an algorithm that gets down to this lower bound?
- What should we look for in an algorithm to get a lg N term in our complexity?

Reminder: Recursion

Recursion

- In general recursion means specifying the answer in terms of the answer of an easier version of the same problem
- At some point the problem gets so “easy” we know the answer
- Example: fibonacci sequence. $F(n) = F(n-1) + F(n-2)$. $F(0) = F(1) = 1$
- We will have to evaluate F a BUNCH of times, but each time will be easier than the last because the parameter keeps getting smaller
- Eventually we'll try to compute F(1) or F(0) and we know the answer is 1

Recursion in Java

- In Java a recursive method is one that calls itself
- The “super easy” versions of the thing we're computing are called the “base cases” where we can just return the answer (like `if(n == 0) return 1` for fibonacci)
- We our recursive calls must “get easier” so that we progress towards the base case, otherwise we'll get infinite recursion!

How recursion works

- How do we keep track of all the calls to a recursive funcion that are “in flight?”
- Each time we call a method, we create a “call stack frame” that stores parameters and local variables for that method call (and where to go after the function returns)
- Each stack frame takes up some space, and the stack is usually fixed size. If we make too many recursive calls, or our stack frames are really we'll run out of stack space
- This is called a stack overflow
- It happens most often when we have infinite recursion, usually because we forgot our base case(s)

When/why to use recursion

- Some problems are naturally defined/explained recursively, so in those cases a recursive algorithm might be much simpler to implement than interative one (ie one with loops)
- You can usually write functionally equivalent or recursive implementations of the same algorithm
- Iterative versions don't have to allocate stack frames so may use less memory (when recursive versions are written naively), the CPU is also usually faster at looping than making function calls/returning

Driver methods

- Recursive methods (particularly if we optimize them) often require some extra parameters that we don't want to require users of our public methods to pass
- Typically we provide a public “driver” method that passes initial required parameters to our recursive method
- These parameters are often things like “0 and the length of our array” etc.

Example: Binary search

```java
public static int binarySearch(int[] sortedArr, int val){
    return binarySearchImpl(sortedArr, 0, sortedArr.length);
}

private static int binarySearchImpl(int[] arr, int val, int beg, int end){
    if( (end - beg) <= 0){ return -1; }//not found
    var middle = (beg + end)/2;
    if(arr[middle] == val){ return middle; }
    if(arr[middle] < val){
        return binarySearchImpl(arr, val, middle + 1, end);
    } else {
        return binarySearchImpl(arr, val, beg, middle);
    }
}
```