CS 228: Introduction to Data Structures
Lecture 10
Wednesday, February 4, 2015

A Hierarchy or Running Times

We can establish a rough hierarchy of function classes based on their relative growth rates.

- ***Constant***, O(1), functions don't grow at all.
- ***Logarithmic***, O(log n), functions are slower growing that linear functions.
- ***Linear***, O(n), functions are slower growing than O(n log n) functions.
- ***Linearithmic***, O(n log n) functions are slower growing than quadratic functions.
- ***Polynomial functions*** — i.e., $O(n^k)$ functions, k constant — grow more slowly than
- ***Exponential functions*** — i.e., α^n functions, $\alpha > 1$.

Where the time complexity of an algorithm falls in this hierarchy can have a big practical impact.

Some Rules of Thumb for Algorithm Analysis

- O(1) denotes ***constant time***. Any operation *not* dependent on the input size falls in this category; e.g., assignments, comparisons, and increments.

- A polynomial is always big-O of its leading term.

- For a O(f) operation followed by an O(g) operation, you can ignore the smaller one. E.g., $O(n^2 + n)$ is $O(n^2)$.

- If a O(f) operation is repeated O(g) times, the total time is $O(f \circ g)$. E.g., if an $O(n^2)$ operation is performed O(n log n) times, the whole thing is $O(n^3 \log n)$.

- If the problem size n is decreased by a ***constant factor*** at each step, the number of steps is O(log n). Example: binary search.

Sorting

To ***sort*** an n-element array `A[0..n−1]` is to rearrange its elements so that

$$A[0] \le A[1] \le A[2] \le \ldots \le A[n-1]$$

The need to sort numbers, strings, and other records arises frequently. The entries in your phone's contact list are sorted. Spreadsheets and databases allow you to sort the records according to any field the you desire. Google sorts query results by their "relevance". Sorting is a preprocessing step in hundreds of algorithms.

We will study and compare several sorting algorithms. For now, we will assume that A consists of `ints`. Later we will see how to sort arbitrary objects using ***generics***.

Modern languages normally have good sorting routines built into their libraries, so nowadays we rarely have to implement sorting algorithms. However, the study of sorting is still useful.

- Sorting offers great examples of algorithm development and analysis. In fact, sorting is perhaps the simplest fundamental problem that offers a huge

variety of algorithms, each with its own inherent advantages and disadvantages.

- The difference between O(n log n) and O(n^2) on a million elements looks like 20 ms versus 15 minutes. This is the difference between a sophisticated and a naive sorting algorithm.

- Quicksort provides good example of using recursion

- Sorting is a good motivation for introducing generic types and methods

- To use library sorting methods you must have a good understanding of certain fundamental concepts; e.g., comparators and stability.

Selection Sort

The idea is simple:

```
for i = 0 to n−1:
    find the smallest element in A[i..n−1] and
    exchange it with A[i].
```

For example,

4 3 2 **1** --> 1 **3** **2** 4 --> 1 2 3 4

Here's the pseudocode:

```
SELECTIONSORT(A)
    n = A.length
    for i = 0 to n−1
        min = A[i]
        jmin = i
        for j = i+1 to n−1
            if A[j] < min
                min = A[j]
                jmin = j
        swap A[i] and A[jmin]
```

There are n iterations of the outer loop and the inner loop iterates ≤ n times, doing a constant amount of work per iteration. Thus, the algorithm is $O(n^2)$. More formally, at iteration i of the outer loop there are n – i – 1 iterations of the inner loop, each of which does a constant amount of work. Therefore, the number of steps is

$$(n - 1) + (n - 2) + (n - 3) + \ldots + 2 + 1 = n(n - 1) / 2.$$

(from a formula learned in calculus I). So the running time is indeed $O(n^2)$, verifying our initial rough estimate.

Insertion Sort

The idea is again simple:

```
for i = 1 to n - 1:
    insert A[i] in its proper place amongst A[0..i].
```

For example,

```
4 3 2 1 --> 3 4 2 1 --> 2 3 4 1 --> 1 2 3 4
```

Note that inserting `A[i]` in the correct place implies shifting up some, maybe all, elements of `A[0 .. i-1]` to make room for `A[i]`. All this is handled by the pseudocode below.

```
INSERTIONSORT(A)
    n = A.length
    for i = 1 to n-1
        temp = A[i]
        j = i - 1
        while j > -1 && A[j] > temp
            A[j+1] = A[j]
            --j
        A[j+1] = temp
```

The analysis is very similar to that of selection sort: There are n iterations of the outer loop and the inner loop iterates no more than n times, doing a constant amount of work per iteration. Thus, the algorithm is also $O(n^2)$.

Insertion Sort versus Selection Sort

Both algorithms take $O(n^2)$ time, so which should you use?

Here is why insertion sort might be preferable:

- In selection sort, iteration i of the outer loop always scans elements `A[i]` through `A[n−1]`. This takes time proportional to `i`, which means that the total time is at least cn^2 for some c.

- The running time of insertion sort is proportional to n plus the number of ***inversions***. An inversion is a pair of keys `j < k` such that `j` appears after k in A. There are anywhere between zero and n·(n – 1) / 2 inversions in A. If A has few inversions (i.e., it is "almost" sorted), insertion sort can be as fast as O(n).

Moral of the Story: The O-bound tells only part of the story.

Loop Invariants

Loop invariants are statements that remain true during the execution of a loop; they provide a way to prove the correctness of algorithms. For example, insertion sort maintains the following:

> **Invariant.** At the start of iteration `i` of the outer loop, subarray `A[0..i−1]` consists of the elements originally in `A[0..i−1]`, but in sorted order.

Now:

- The invariant is true at the outset.
- The loop maintains the invariant through shifting and insertion.
- At termination, `i = n`, so the invariant implies that subarray `A[0..n-1]` — *the whole array* — consists of the elements originally in `A[0..n-1]`, but in sorted order.

The last statement proves the correctness of insertion sort.

Exercise. What loop invariant does selection sort maintain?