COMP 200: Elements of Computer Science
Fall 2004
Lecture 21: October 18, 2004

Worst, Best, and Average Case

On the Board

Homework 5: Work problem 6.3 in the book. Due 11/1/2004.
Final Project suggestions: posted on the web site under homework.

Worst-case Complexity

The Big-O notation that we have talked about the last two classes is generally used to describe worst-case complexity—the worst behavior that the algorithm can exhibit.

Average Case versus Worst Case Complexity

Big-O notation traditionally deals with worst case behavior. One version of the bubblesort algorithm (from lecture 11) was

```
Swapped ← true

while (swapped)
swapped ← false

  for j ← 1 to N-1
      if Keys[j] > Keys[j+1]
        then
            swap Keys[j] & Keys[j+1]
            swapped ← true
```

This version of bubblesort requires $\mathbf{O}(N^2)$ time, in the worst case. In the best case, when the input is ordered, it makes a single pass over the *Keys* array and halts after $\mathbf{O}(N)$ comparisons. Thus, we see a gap between its best case behavior and its worst case behavior. Its average case behavior depends on the distribution of values in the *Keys* array.

Best-case Complexity

Just as an algorithm can have worst-case complexity and average-case complexity, it can also have best-case complexity. Worst-case analysis establishes an upper bound on the cost of solving the problem. Exhibiting a correct algorithm that solves the problem and providing an analysis of its complexity bounds the *problem's* worst-case behavior. Upper and lower bounds are a property of the problem, not of the algorithm, while average-case behavior is inherently a property of an algorithm rather than of a problem.

To establish a lower bound for a problem, we must prove that no algorithm can solve arbitrary instances of the problem faster than implied by the lower bound. This proposition is much harder than proving an upper bound — while an algorithm shows an existence proof of a method that solves the problem within some time bound, an algorithm cannot demonstrate that no faster algorithm exists. Thus, lower bounds are more difficult to establish.

Sorting is a problem for which a firm lower bound has been established. The discussion on page 146 to 148 of the book establishes a lower bound of $\mathbf{O}(N\ log_2\ N)$ for sorting. The argument is not overly complex, but the details exceed the threshold for presenting in class, so I leave it to you as a reading exercise.

Algorithms for sorting arbitrary lists in $\mathbf{O}(N\ log_2\ N)$ time exist, so the upper and lower bounds are identical. We call a problem with identical upper and lower bounds a ***closed problem*** — from the point of view of computational complexity, the problem has no interesting unsolved issues. If the upper and lower bounds are not equal, then the problem is an ***open problem*** — there is still room for developing an algorithm with lower Big-O complexity.

When we discuss NP-Complete problems, we will see that some problems have huge algorithmic gaps — the distance between their upper and lower bounds.

Sorting as the Poster Child for Average-case Complexity

In fact, sorting is a closed problem. The lower bound on sorting arbitrary numbers is $\mathbf{O}(N\ log_2\ N)$ and we have discovered algorithms that achieve this lower bound, such as HeapSort. Sorting is an interesting example because the fastest known sorting algorithms may not have $\mathbf{O}(N\ log_2\ N)$ worst case behavior. In fact, the classic example of an algorithm whose average-case behavior is better than its worst case behavior (and its average case behavior almost always occurs) is Tony Hoare's algorithm ***QuickSort***.

QuickSort has $\mathbf{O}(N^2)$ worst case behavior, but $\mathbf{O}(N\ log_2\ N)$ average case behavior, and it has an exceptionally small constant. The sorting algorithms with $\mathbf{O}(N\ log_2\ N)$ worst case behavior have larger constants than QuickSort.

In practice, QuickSort is one of the fastest known algorithms for sorting a list of numbers.

QuickSort:

1. Pick a representative list element, the pivot
2. Partition the list into two lists around the pivot
   - One list has values < pivot, other has values > pivot
3. Sort the smaller lists
   - Use recursion on non-trivial cases
4. Combine the sorted lists
   - Append smaller, pivot, and larger

Assume that each value occurs only once in the list

The key to making the algorithm work well is picking the pivot element. Ideally, you want to partition the list into equal-sized sublists. A bad implementation might pick the first element as pivot, or the last element as pivot. (*First element as pivot is a disaster for an ordered list.*) A better implementation would pick a random element — avoiding any pattern in the data. An alternative is to take the average of several elements — first & last, for example.

QuickSort gets its great average case-complexity when it partitions the list well. It wins over MergeSort because the combine step is inexpensive — no comparisons, just paste the lists back together. If it is done carefully, in a vector similar to the *Keys* vector, the combine step occurs naturally — the sublists are formed inside the *Keys* vector in a way that makes the append step implicit.

If QuickSort uses an actual list element as the pivot, it wins in another way. At each step, it removes the pivot element from the sort. This strategy gives it another slight advantage over MergeSort — at each point, it creates slightly smaller subproblems than MergeSort will.