Computer Science 385
Analysis of Algorithms
Siena College
Spring 2011

Topic Notes: Decrease and Conquer

Our next class of algorithms are the *decrease-and-conquer* group.

The idea here:

1. Reduce the problem to a smaller instance
2. Solve the smaller instance
3. Modify the smaller instance solution to be a solution to the original

The main variations:

- Decrease by a constant (often by 1)
- Decrease by a constant factor (often by half)
- Variable size decrease

For example, consider variations on how to compute the value a^n.

The brute-force approach would involve applying the definition and multiplying a by itself, n times.

A decrease-by-one approach would reduce the problem to computing the result for the problem of size $n-1$ (computing a^{n-1}) then modifying that to be a solution to the original (by multiplying that result by a).

A decrease-by-constant-factor (2, in this case) approach would involve computing $a^{\frac{n}{2}}$ and multiplying that result by itself to compute the answer. The only complication here is that we have to treat odd exponents specially, leading to a rule:

$$a^n = \begin{cases} (a^{\frac{n}{2}})^2 & \text{if } n \text{ is even} \\ (a^{\frac{n-1}{2}})^2 \cdot a & \text{if } n > 1 \text{ and odd} \\ a & \text{if } n = 1 \end{cases}$$

This approach will lead to $O(\log n)$ multiplications.

Insertion Sort

Our decrease-by-one approach to sorting is the *insertion sort*.

The insertion sort sorts an array of n elements by first sorting the first $n-1$ elements, then inserting the last element into its correct position in the array.

```
insertion_sort(A[0..n-1]) {
  for (i=0 to n-1) {
    v = A[i]
    j = i-1
    while (j >= 0 and A[j] > v) {
      A[j+1] = A[j]
      j--
    }
    A[j+1] = v
```

This is an in-place sort and is stable.

Our basic operation for this algorithm is the comparison of keys in the while loop.

We do have differences in worst, average, and best case behavior. In the worst case, the while loop always executes as many times as possible. This occurs when each element needs to go all the way at the start of the sorted portion of the array – exactly when the starting array is in reverse sorted order.

The worst case number of comparisons:

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$$

In the best case, the inner loop needs to do just one comparison, determining that the element is already in its correct position. This happens when the algorithm is presented with already-sorted input. Here, the number of comparisons:

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

This behavior is unusual – after all, how often do we attempt to sort an already-sorted array? However, we come close in some very important cases. If we have nearly-sorted data, we have nearly this same performance.

A careful analysis of the average case would result in:

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$$

Of the simple sorting algorithms (bubble, selection, insertion), insertion sort is considered the best option in general.

Graph Traversals

We return to graph structures for our next group of decrease-and-conquer algorithms. In particular, we consider the problem of visiting all vertices in a graph.

The two main approaches are the *depth-first search (DFS)* and the *breadth-first search (BFS)*.

See the text for details...

Topological Sort

Again, see the text for details...