Illinois Institute of Technology
Department of Computer Science

Lecture 3: September 3, 2014

CS 430 Introduction to Algorithms
Fall Semester, 2014

How to Sort A Sequence of Numbers

Defining the problem

Input: A sequence of n numbers $a_1, a_2, \cdots, a_n$
Output: A permutation (reordering) of the input sequence such that $a_1 \leq a_2 \leq \cdots \leq a_n$

Insertion Sort

Referring to the pseudocode for insertion sort on page 18 of CLRS, if we want to analyze the time complexity of insertion sort, we need to answer the following question: *How often will lines 6–7 be executed?* Let $n = A.length$.

Obviously, insertion sort will be very fast if file already is nearly sorted; if it is actually sorted, insertion sort uses $\Theta(n)$ comparisons; this is the best case.

Proposition: Insertion sort uses $O(n^2)$ comparisons in the worst case as well as on the average (assuming all permutations are equally likely).

Proof: We need to compute be number of elements to the left of a_i and larger than it. Call this d_i; this is the number of *inversions* relative to a_i; $M = \sum_{i=1}^{n} d_i$ is the number of inversions of the input permutation.

The number of comparisons made by insertion sort is $M + n - 1$ (the $n - 1$ occur once for each iteration of the while loop, as it ends). $0 \leq d_i < i$, $i = 1, \ldots, n$. In the worst case, $d_i = i - 1$, occurs when the elements to be sorted are in decreasing order, giving $M = n(n-1)/2$.

To analyze the average case, we need to compute

$$\mathbf{E}(M) = \frac{1}{n!} \sum_{\text{all permutations } \pi} \sum_{i=1}^{n} [d_i \text{ for } \pi]$$

How many permutations of n elements have $d_i = m$? Call this $\#[d_i = m]$. Then by regrouping, the above sum becomes

$$\mathbf{E}(M) = \frac{1}{n!} \sum_{i=1}^{n} \sum_{m=0}^{i-1} m \times \#[d_i = m]$$

The rule of product tells us that the first i elements of the input permutation can be chosen in $\binom{n}{i}$ ways; the ith element must be the $m+1$st largest element of those i for d_i to be m (that is, for there to be m elements to the left of a_i and larger than it). The $i-1$ elements to its left can be arranged any of the $(i-1)!$ possible ways and the $n-i$ elements to its right can be arranged in any of the possible $(n-i)!$ ways. Thus

$$\#[d_i = m] = \binom{n}{i} \times (i-1)! \times (n-i)!$$

and the sum we want becomes

$$\mathbf{E}(M) = \frac{1}{n!}\sum_{i=1}^{n}\sum_{m=0}^{i-1} m\binom{n}{i} \times (i-1)! \times (n-i)! = \sum_{i=1}^{n}\frac{1}{i}\sum_{m=0}^{i-1} m = \sum_{i=1}^{n}\frac{i-1}{2} = \frac{1}{2}\sum_{i=0}^{n-1} i = \frac{n(n-1)}{4}.$$

Thus for a random permutation,

$$\mathbf{E}(M) = \frac{n(n-1)}{4}.$$

Heap Sort (Chapter 6 of CLRS)

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called its key. A max-priority queue supports the following operations: **MAXIMUM(S)** returns the element of S with the largest key. **EXTRACT-MAX(S)** removes the element of S with the largest key. **INCREASE-KEY(S, x, k)** increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current value. **INSERT(S, x)** inserts the element x in the set S.

The (binary) heap data structure is an array object that can be viewed as a nearly completely binary tree. Each node of the tree corresponds to an element of the array that stores the value in the node. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. An array A that represents a heap is an object with two attributes: $length[A]$, which is the number of elements in the array, and $heap-size[A]$, the number of elements in the heap stored within array A.

The root of the tree is $A[1]$, and given the index i of a node, the indices of its parent PARENT(i), left child LEFT(i), and the right child RIGHT(i) can be computed simply:

$$PARENT(i) = \lfloor i/2 \rfloor$$
$$LEFT(i) = 2i$$
$$RIGHT(i) = 2i + 1$$

In a max-heap, the max-heap property is that for every node i

$$A[PARENT(i)] \geq A[i]$$

Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.

The heapsort algorithm starts by BUILD-MAX-HEAP to build a max-heap on the input array $A[1..n]$, where $n = length[A]$. Since the maximum element of the array is stored at the root $A[1]$, it can be put into its correct final position by exchanging it with $A[n]$. If we now "discard" node n from the heap(by decrementing heap-size[A]), we observe that $A[1..(n-1)]$ can easily be made into a max-heap. The children of the root remain max-heaps, but the new root element may violate the max-heap property. All that is needed is to restore the max-heap property, however, is one call to MAX-HEAPIFY(A,1), which leaves a max-heap in $A[1..(n-1)]$. The heapsort algorithm then repeats this process for the heap if size $n-1$ down to a heap of size 2.

The HEAPSORT algorithm takes time $O(n \log n)$, since the call to BUILD-MAX-HEAP takes time $O(n)$ and each of the $n-1$ calls to MAX-HEAPIFY takes time $O(\log n)$.