LECTURE 5, JAN 27, 2011, SORTING ALGORITHMS

In this class, we discuss several sorting algorithms.

Bubble sort. 1. Its worst complexity is $n(n-1)/2$.
During each pass the bubble sort successively compares adjacent elements, interchanging them if necessary. When the ith pass begins, the $i-1$ largest elements are guaranteed to be in the correct positions. During this pass, $n-i$ comparisons are used. Consequently, the total number of comparisons used by the bubble sort to order a list of n elements is

$$(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}$$

2. Minor improvement to the bubble sort: keep a record of the position where last exchange happened.

Note: the smaller elements "bubble" to the top as they are interchanged with larger elements. The larger elements "sink" to the bottom.

Insertion sort. 1. Its worst complexity is $\Theta(n^2)$.
The insertion sort inserts the jth element into the correct position among the first $j-1$ elements that have already been put into the correct order. It does this by using a linear search technique. Consequently, in the worst case, j comparisons are required to insert the jth element into the correct position. Therefore, the total number of comparisons used by the insertion sort to sort a list of n elements is

$$2+3+\cdots+n=\frac{n(n+1)}{2}-1.$$

2. Minor improvement to the insertion sort, *binary insertion sort*, based on the fact the first $j-1$ elements have been put into the correct order at the beginning of the jth insertion.

Merge sort. 1. It is a *recursive algorithm*, which solves a problem by reducing it to an instance of the same problem with smaller input.

2. To do a merge sort, we split a list into two sublists of equal, or approximately equal, size, sorting each sublist using the merge sort algorithm, and then merging the two lists.

3. Do an example about how to merge two sorted lists.

4. How many comparisons are required of merging two ordered lists L_1 and L_2 into an ordered list L?
Two sorted lists with m elements and n elements can be merged into a sorted list using no more than $m+n-1$ comparisons.

5. Worst-case complexity analysis?

Theorem 1 (The master theorem)**.** *Let $a \geq 1$ and $b \geq 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence*

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\text{floor}(n/b)$ *or* $\text{ceil}(n/b)$*. Then*

(1) *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.*
(2) *If $f(n) = O(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.*
(3) *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c \leq 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

Example: let $T(n)$ denotes the complexity of the merge sort when the input size is n. For simplicity, let's assume $n = 2^m$. We then have

$$T(n) = 2T(n/2) + n - 1.$$

For this recurrence, we have $a = 2$, $b = 2$, $f(n) = n - 1$, and thus $n^{\log_b a} = n$. Since $f(n) = \Theta(n)$, we conclude that $T(n) = \Theta(n \log n)$.

6. Prove the algorithm work using mathematical induction.

Notes on recursive algorithm. An algorithm is called recursive if it solves a problem by reducing it to an instance of the same problem with smaller input.

Recursive algorithm depends on the fact that the solution to the problem is easy to establish when the input becomes sufficiently small. Additionally, the solution to the smaller problem can lead to that of a larger problem.

Example: compute a^n using a recursive algorithm, where n is a nonnegative integer and $a \neq 1$.

We know that $a^0 = 1$ by definition, meaning we know the result of the problem when n is small (actually the smallest input in this case). Knowing the result a^0 gives result to a^1, which in turn gives result to a^2, and so on and so forth.

To find a^n, successively use the recursive step to reduce the exponent until it becomes zero.

The correctness of a recursive algorithm can be established using mathematical induction.

The recursive algorithm can solves the problem with a very small input directly. This corresponds to the basis step in a mathematical induction proof.

Then, the recursive algorithm itself is similar to the inductive step of the proof. The algorithm itself provides a way to handle a larger problem by assuming all smaller cases can be solved. (More like a strong mathematical induction.)

For example, let $P(n)$ denote the proposition that the merge sort algorithm gives correct result when the input size is n.

Proof. Basis step: we know $P(1)$ is true due to the fact that a list with a single element is in the correct order.

Inductive step: assuming $P(j)$ is true for $j \leq k$. When the input size is $k+1$, the algorithm will separate them into two parts of length $k_1, k_2 \leq k$. By inductive assumption, those two parts are sorted correctly. Since the merge routine works correctly, we know that an input of size $k+1$ can be sorted correctly by the algorithm.

□