CMPS 101
Algorithms and Abstract Data Types

Introduction to Algorithm Analysis

Summary of the Theory Side of this Course

- Mathematical Preliminaries
    - Asymptotic growth rates of functions
    - Some common functions and their properties
    - Induction Proofs
    - Recurrences
- Standard ADTs
    - Elementary Data Structures like Stacks, Queues, and Lists
    - Graphs and Directed Graphs
    - Priority Queues
    - Disjoint Sets
    - Binary Search Trees and Red-Black Trees
    - Dictionaries
    - Hash Tables
- Algorithms Associated with these ADTs
    - Sorting and Searching
    - Breadth First Search, Depth First Search, Shortest Paths, Minimum Weight Spanning Trees
    - Insertion and Deletion in Balanced Trees
- Run Time Analysis of these Algorithms

Some Sorting Algorithms

A classic problem in Computer Science is that of sorting a collection of objects in increasing order. We assume for the moment that the objects to be sorted are stored in an array A of fixed length n.

$$A = (A_1, A_2, A_3, \ldots, A_n)$$

Our convention is that array indices range from 1 to $n = \text{length}[A]$ (not 0 to $n-1$, as in many computer languages). We denote the subarray with indices ranging from i to j by $A[i \cdots j] = (A_i, \ldots, A_j)$. If $i > j$ this is understood to denote the empty array, i.e. an array of length 0. We begin by analyzing two algorithms that solve this problem: *Insertion Sort* and *Merge Sort*.

InsertionSort(A)
1.) for $j \leftarrow 2$ to n
2.)   temp $\leftarrow A_j$
3.) $i \leftarrow j - 1$
4.)   while $i > 0$ and temp $< A_i$
5.) $A_{i+1} \leftarrow A_i$
6.) $i \leftarrow i - 1$
7.) $A_{i+1} \leftarrow$ temp

If this notation is not absolutely clear, read the pseudo-code conventions outlined on pages 19-20 of the text. In particular, notice that the left arrow "←" stands for the assignment operator, and that loop bodies and conditional branches are indicated solely by indentation, not by braces or other punctuation characters. Also there are no semicolons, parenthesis, or brackets, and local variables need not be declared before use.

Notice that on the jth iteration of loop 2-7, the subarray $A[1\cdots(j-1)]$ is already sorted, while $A[j\cdots n]$ is unsorted. Steps 3-7 have the effect of inserting A_j into its correct position in $A[1\cdots(j-1)]$, thus expanding the sorted section by exactly 1.

$$A_1 \cdots\cdots\cdots A_{j-1} \; A_j \; A_{j+1} \cdots\cdots\cdots A_n$$
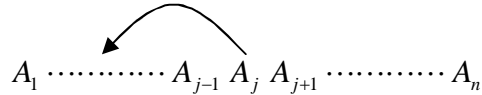
Exercise: Trace InsertionSort(A) on $A = (8, 5, 3, 1, 4, 7, 2, 6)$

We wish to determine the *run time* $T(n)$ of this algorithm as a function of the input size n. This analysis should be, as far as possible, independent of the computing machine that is executing the algorithm. Let c_k denote the cost of step k. We remain uncommitted as to just what units or even what quantity c_k measures. This could be processor time in seconds, or power consumed in watts, or even some appropriate monetary unit. As we shall see, details of this kind are not critical to the analysis. Thus for instance, c_2 and c_3 represent the costs of assignment operations, while c_1 and c_4 are the costs of performing tests of loop repetition conditions. Notice that c_2 and c_3 may be unequal since step 3 is an integer assignment, while step 2 is an assignment of array elements, which need not be integers.

Let t_j denote the number of executions of the while loop test (line 4) on the jth iteration of the outer for loop 1-7. Observe that the body of the while loop 4-6 executes $t_j - 1$ times on the jth iteration of loop 1-7. The total cost is then

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 (n-1)$$

$$= (c_4 + c_5 + c_6)\left(\sum_{j=2}^{n} t_j\right) + (c_1 + c_2 + c_3 - c_5 - c_6 + c_7)n + (-c_2 - c_3 + c_5 + c_6 - c_7)$$

We see that $T(n)$ depends on the numbers t_j, which themselves depend on the particular permutation of the input array A. We consider three measures of run time that take into account all possible arrangements of A, namely *best case*, *worst case*, and *average case*.

In the best case, the array A is already sorted in increasing order, so that $t_j = 1$ for $2 \le j \le n$, and $\sum_{j=2}^{n} t_j = n-1$, whence

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n + (-c_2 - c_3 - c_4 - c_7).$$

Best case is unlikely to be of much practical interest however. A more useful measure is worst case, which occurs when the array is initially sorted in decreasing order. In this case $t_j = j$ for $2 \le j \le n$, and $\sum_{j-2}^{n} t_j = \frac{n(n+1)}{2} - 1$, and therefore

$$T(n) = \left(\frac{1}{2}c_4 + \frac{1}{2}c_5 + \frac{1}{2}c_6 \right)n^2 + \left(c_1 + c_2 + c_3 + \frac{1}{2}c_4 - \frac{1}{2}c_5 - \frac{1}{2}c_6 + c_7 \right)n + (-c_2 - c_3 - c_4 - c_7).$$

To determine the average case, we must make some assumption about the likelihood of each of the $n!$ distinct permutations of the input array. In the absence of any other information, we assume that each arrangement of array A is equally likely as input. This suggests that, on average, half the elements in $A[1\cdots(j-1)]$ are less than A_j, and half are greater. On average then, $t_j = j/2$, from which it follows that $\sum_{j=2}^{n} t_j = \frac{1}{2}\left(\frac{n(n+1)}{2} - 1 \right)$, and hence

$$T(n) = (\frac{1}{4}c_4 + \frac{1}{4}c_5 + \frac{1}{4}c_6)n^2 + (c_1 + c_2 + c_3 - \frac{3}{4}c_5 - \frac{3}{4}c_6 + c_7)n + (-c_2 - c_3 - \frac{1}{2}c_4 + \frac{1}{2}c_5 + \frac{1}{2}c_6 - c_7).$$

We leave it as an exercise for the reader to verify all of the preceding algebraic calculations. Summarizing these results we have

	$T(n)$	Asymptotic Growth Rate of $T(n)$
Best Case	$an + b$	$\Theta(n)$
Worst Case	$cn^2 + dn + e$	$\Theta(n^2)$
Average Case	$fn^2 + gn + h$	$\Theta(n^2)$

where the a-h depend on the constants c_1-c_7, which in turn depend on the particular computing device used. Our goal is to define a measure of run time that is machine independent. We will call this measure the *asymptotic growth rate* of $T(n)$. Informally speaking, the asymptotic growth rate is a measure of how fast $T(n)$ increases or "scales up" with n.

To make these ideas more concrete, consider four algorithms A, B, C, and D whose run times on inputs of size n are respectively

	Run Time	Asymptotic Growth Rate
A	n^2	$\Theta(n^2)$
B	$10n^2$	$\Theta(n^2)$
C	$10n^2 + 2n + 100$	$\Theta(n^2)$
D	$1000n + 10000$	$\Theta(n)$

3

We can see that D is superior for large n, and worst for small n. Why can A, B, and C be classified as equivalent? The lower order terms in C are negligible for large n, so there is no effective difference between B and C. Algorithms A and B can be equalized by running B on a machine that is 10 times faster than the one running A, so we should not distinguish between them if we seek a machine independent measure of run time. We will give precise definitions of these notions in due course, but in the mean time, observe that the asymptotic growth rate of $T(n)$ is obtained by dropping the low order terms in $T(n)$, and replacing the coefficient of the highest order term by 1.
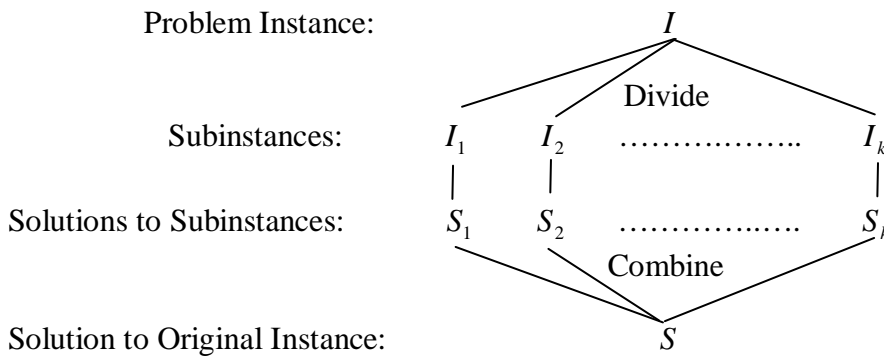
Returning to our analysis of Insertion Sort, since the constants *a-h* are not critical to the asymptotic growth rate, and likewise for c_1-c_7, we need make no effort to calculate them explicitly. Instead we pick a representative *basic operation* (also called a *barometer operation*), and count the number of times that operation is executed on inputs of a fixed size n (in best, worst, and average cases.) In sorting algorithms it is customary to take the comparison of array elements as basic operation. This takes place on line 4 of Insertion Sort.

Exercise: Show that the numbers of array comparisons performed by Insertion sort on arrays of length n are, respectively:

|	Number of Comparisons	Asymptotic Growth Rate
Best Case	$n-1$	$\Theta(n)$
Worst Case	$\dfrac{n(n-1)}{2}$	$\Theta(n^2)$
Average Case	$\dfrac{n(n-1)}{4}$	$\Theta(n^2)$

Thus, as far as the asymptotic growth rate is concerned, we obtain the same results as in our earlier detailed analysis.

Merge Sort is a sorting algorithm whose strategy is quite different from that of Insertion Sort. It utilizes a technique called *divide and conquer*, which we describe here in general terms. Given an instance of the problem to be solved, divide that instance into some number of subinstances of smaller size, solve the subinstances recursively, then combine the subproblem solutions so as to construct a solution to the original instance.

Problem Instance: I

Divide

Subinstances: I_1 I_2 ……….…….. I_k

Solutions to Subinstances: S_1 S_2 ………..…… S_k

Combine

Solution to Original Instance: S

To say that the subinstances are solved recursively means that the same divide and conquer technique is applied to each of them, obtaining yet smaller instances of the problem. This process eventually reaches a point where the size of the problem instance is so small, that the solution is trivial.

MergeSort(A, p, r) begins with an unsorted subarray $A[p \cdots\cdots r]$, divides it into two subarrays of approximately equal length $A[p \cdots q]$ and $A[q+1 \cdots r]$ where $p \le q < r$, sorts the two subarrays recursively, then combines the two sorted subarrays into a single sorted array $A[p \cdots\cdots r]$. The combine stage is performed by calling another algorithm called Merge(A, p, q, r).

$A[p \cdots\cdots\cdots\cdots r]$ Unsorted

$A[p \cdots\cdots q]$ $A[q+1 \cdots\cdots r]$

$A[p \cdots\cdots\cdots\cdots r]$ Sorted
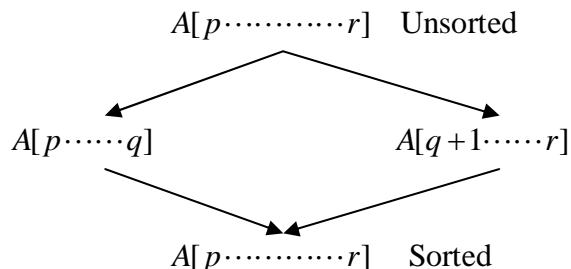
Merge(A, p, q, r) requires as precondition that the subarrays $A[p \cdots q]$ and $A[q+1 \cdots r]$ are already sorted, and that $p \le q < r$. It copies each subarray into temporary spaces $T[p \cdots q]$ and $T[q+1 \cdots r]$, then steps through the temporaries from left to right, comparing an element in $T[p \cdots q]$ to one in $T[q+1 \cdots r]$, then placing the smaller in its final position in $A[p \cdots\cdots r]$.

Exercise: Write pseudo-code for the algorithm Merge(A, p, q, r), or just read it on page 29 of the text. Show that in worst case, Merge(A, p, q, r) performs $r - p$ array comparisons.

<u>MergeSort(A, p, r)</u>
1. if $p < r$
2. $q \leftarrow \lfloor (p+r)/2 \rfloor$
3.   MergeSort(A, p, q)
4.   MergeSort(A, $q+1$, r)
5.   Merge(A, p, q, r)

If $p \ge r$ then $A[p \cdots\cdots r]$ contains at most one element, and so is already sorted. MergeSort() does nothing in this case. If $p < r$ we calculate the index q approximately half way between p and r, then call MergeSort() on the two subarrays $A[p \cdots q]$ and $A[q+1 \cdots r]$, then call Merge() to combine these into a single sorted subarray $A[p \cdots\cdots r]$. The top level call to MergeSort(A, 1, n) sorts the full array $A[1 \cdots\cdots n]$ of length n.

Exercise: Trace MergeSort(A, 1, 8) on $A = (8, 5, 3, 1, 4, 7, 2, 6)$.
Exercise: Show that if $p = 1$, $r = n$, and $q = \lfloor (1+n)/2 \rfloor$, then $A[p \cdots q]$ has length $\lceil n/2 \rceil$, and $A[q+1 \cdots r]$ has length $\lfloor n/2 \rfloor$.

We wish to determine the run time of MergeSort(A, 1, n) as a function of $n = \text{length}[A]$. Let $T(n)$ denote the worst case number of comparisons performed by MergeSort() on arrays of length n. Observe that all array comparisons take place within the various calls to Merge(), and recall that Merge(A, p, q, r) performs $r - p$ array comparisons in worst case. When $p = 1$ and $r = n$ this is $n - 1$ comparisons. Therefore $T(n)$ must satisfy the following recurrence relation.

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + (n-1) & n \geq 2 \end{cases}$$

If n happens to be an integer power of 2, this reduces to

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T(n/2) + (n-1) & n \geq 2 \end{cases}$$

The solution to this last recurrence is $T(n) = n \lg(n) - n + 1$, where $\lg(n)$ denotes the base 2 logarithm of n. We emphasis that this solution is only valid in the special case that $n = 2^k$ for some $k \in \mathbf{Z}$.

Proof: Right Hand Side $= 2T(n/2) + (n-1)$
$$= 2\big((n/2)\lg(n/2) - (n/2) + 1\big) + (n-1)$$
$$= n\lg(n/2) - n + 2 + n - 1$$
$$= n(\lg(n) - 1) + 1$$
$$= n\lg(n) - n + 1$$
$$= T(n)$$
$$= \text{Left Hand Side} \qquad\qquad ///$$

The highest order term in $T(n)$ is obviously $n\lg(n)$, and hence the asymptotic growth rate of $T(n)$ is $\Theta(n\lg(n))$. As we will soon see, the asymptotic solution $T(n) = \Theta(n\lg(n))$ is valid for all n, not just when $n = 2^k$. This analysis must remain somewhat vague until we define precisely what is meant by $\Theta(n\lg(n))$, $\Theta(n^2)$, and more generally $\Theta(g(n))$ for any function $g(n)$. In the mean time, we observe that n^2 grows much faster than $n\lg(n)$. We conclude that for large n, Merge Sort is a better algorithm than Insertion Sort, regardless of the machine implementation.

Our treatment of Merge Sort has been typical of the way we will analyze recursive algorithms. We let $T(n)$ denote the (best, worst, or average) case run time, or the number of barometer operations, on inputs of size n. If n is sufficiently small, say $n < n_0$ for some positive constant n_0, then the problem is trivial, and no subdivisions are necessary. In this case the solution runs in constant time: $T(n) = c$. This is the point at which the recursion "bottoms out". If $n \geq n_0$, we divide the problem into a subproblems, each of size n/b, where $a \geq 1$ and $b > 1$. Suppose our algorithm takes time $D(n)$ to divide the original problem instance into subinstances, and time $C(n)$ to combine the solutions to these instances into a solutions to the original problem instance. Then we obtain a recurrence relation for $T(n)$:

$$T(n) = \begin{cases} c & n = 1 \\ aT(n/b) + D(n) + C(n) & n \geq 2 \end{cases}$$

We will learn in Chapter 4 how to solve recurrence of this form, both explicitly, and in the asymptotic sense.