CPSC-320 Intermediate Algorithm Design and Analysis Winter 2000

Homework # 2

Due in class on Friday, September 29.

1. Binary search is an efficient way to search a sorted array for a given key K. Here is a version of binary search that can be carried out in parallel with p processors.

   First, divide the sorted array into $p+1$ segments of approximately equal size, so that there are p keys at the internal boundaries of the segments. Each processor $i, 1 \leq i \leq p$ **compares** the given key K to the ith internal boundary key and writes, in a variable, c_i, a 0 if K is greater and a 1 if K is smaller than the boundary key (in case of equality, the search ends). All processors do this in parallel.

   Then, in parallel, each processor i checks to see if $c_i = 0$ and $c_{i+1} = 1$. We assume that $c_0 = 0$ and $c_{p+1} = 1$, so for exactly one value $j, 0 \leq j \leq p$, both $c_j = 0$ and $c_{j+1} = 1$. Note that the key K will be in the jth segment, if the key happens to be in the array.

   If $j \geq 1$ then the jth processor broadcasts "j" to the other processors, and if no broadcast is done, all processors determine that $j = 0$. Once all processors have j, the search then continues recursively within the jth segment, again with the p processors working in parallel.

   When the segment is of size p or less, each processor compares one element and the search ends.

   Give a recurrence relation for the **parallel** number of key comparisons, say $P(n)$, done by this algorithm in the worst case. ($P(n)$ can also be viewed as the number of comparisons done by a single processor during the algorithm.) Use the iteration method to solve your recurrence, so that you get an expression for $P(n)$ as a function of both n and p. You should find an exact solution for your recurrence rather than expressing your result using big-O notation.

2. An arithmetic operation that we will see later this semester is that of exponentiation. Given three (binary) non-negative numbers x, e, and N, the exponentiation problem is to compute $x^e \bmod N$. For example, if $x = 3, e = 4$, and $N = 15$ then $x^e = 3^4 = 81$ and $81 \bmod 15 = 6$, so the answer in this case is 6.

   The following equations explain what is $x^e \bmod N$ in terms of either $x^{e/2}$ or $x^{(e-1)/2}$, for $e > 1$, depending on whether e is even or odd:

$$x^e \bmod N = \begin{cases} (x^{e/2} \bmod N)^2 \bmod N, & \text{if } e \text{ is even} \\ (x^{(e-1)/2} \bmod N)^2 x \bmod N, & \text{if } e \text{ is odd.} \end{cases}$$

   (a) Using the above "inductive" definition of $x^e \bmod N$, describe a recursive algorithm that computes $x^e \bmod N$. (You can assume that multiplication and mod operations on n-bit numbers are available, and that you can use the usual programming constructs such as "if" statements, testing if a number is even, and so on.)

   (b) Give recurrence relations that describe (i) the number of multiplications and (ii) the number of mod operations done by your algorithm, and solve these recurrences.

(c) Using the fact that the running time for each multiplication and mod operation on numbers with at most n bits is $O(n^2)$ (where running time is measured, say, in terms of "bit operations"), give a bound on the running time of the exponentiation algorithm.

(d) A simpler algorithm for exponentiation is as follows. Let *result* be a variable initialized to 1. Repeat e times: *result* $\leftarrow$ *result* $\times\ x \bmod N$, and finally output *result*. What is the worst-case running time of this algorithm if the inputs are n-bit numbers? (Again, you can assume that multiplication and mod operations take $O(n^2)$ time on n-bit numbers.) Which algorithm do you think is faster on large inputs?

3. In the case of algorithms that take more than one input, recurrence relations sometimes have two parameters. Consider an algorithm that takes two inputs of sizes m and n, say with $m \leq n$, with a recurrence for the running time given by

$$T(m,n) = T(m,n-1) + 1; \quad T(m,0) = T(0,n) = 0.$$

For example,

$$\begin{aligned} T(3,4) &= T(3,3) + 1 \\ &= T(3,2) + 1 + 1 \\ &= T(2,2) + 1 + 1 + 1 \text{ because it is always the } \textit{larger} \text{ input that reduces in size} \\ &= T(2,1) + 1 + 1 + 1 + 1 \\ &= T(1,1) + 1 + 1 + 1 + 1 + 1 \\ &= T(1,0) + 1 + 1 + 1 + 1 + 1 + 1 \\ &= 6. \end{aligned}$$

Solve this recurrence to get a bound on the total running time of the algorithm, as a function of m and n.

4. Suppose a country's currency consists of n coins, worth $c_1, \ldots, c_n$ units. You want to compute the minimum number of coins necessary to make change for an amount L. Assume that there is an unlimited supply of each coin. You can also assume that c_1 is 1 unit, so it always possible to make change.

(a) Let $P(L)$ denote the minimum number of coins needed to make change for amount L. Find a recurrence relation for $P(L)$.

(b) Use your recurrence to design an $O(nL)$ time algorithm for this problem.