CS109A Assignment #4

Due 3:15 PM, 2/14/96

Note: Please remember to separate the written homework from the programming problems and hand in each separately.

Written Homework

H1. (20 points) The following ML function is similar to the C function `find0` on p. 87 of FCS. You can analyze ML programs much as we do C programs, understanding that pattern matching takes $O(1)$ time. The pattern match itself causes a branch like an if-statement.

```
fun find0(nil) = false
|   find0(0::xs) = true
|   find0(x::xs) = find0(xs);
```

a) Draw a structure tree for this function.

b) What is a suitable size measure for the input argument?

c) Give smooth and tight big-oh upper bounds on each of your tree nodes, in terms of the unknown running time $T(n)$ for calls to `find0` with arguments of size n.

d) Set up and solve the recurrence equation for $T(n)$.

H2. (15 points) Suppose we have a recurrence of the form

$$T(1) = a$$
$$T(n) = T(n-1) + g(n), \text{ for } n > 1$$

Give tight big-oh upper bounds on the solution if $g(n)$ is

a) n^2
b) $n \log n$
c) 2^n

(Part of Exercise 3.11.2, FCS)

H3. (15 points) Guess each of the following as the solution to the recurrence

$$T(1) = a$$
$$T(n) = 2T(n/2) + bn, \text{ for } n \text{ a power of 2 and } n > 1$$

a) $cn \log_2 n + dn + e$
b) $cn + d$
c) cn^2

What constraints on the unknown constants c, d, and e are implied? For which of these forms does there exist an upper bound on $T(n)$? (Exercise 3.11.4, FCS)

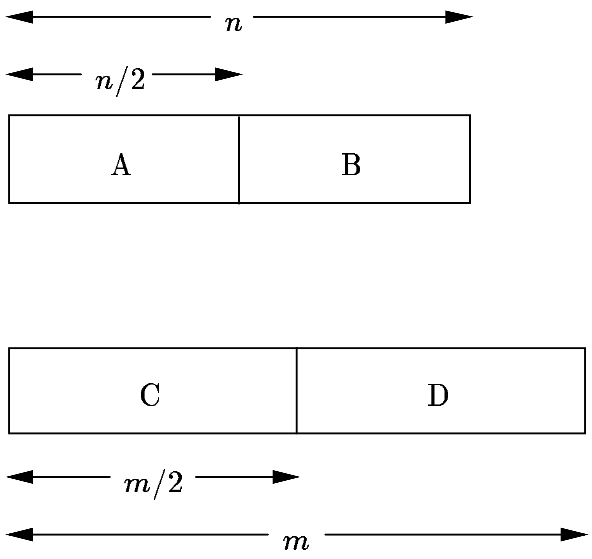
Programming Problems

This week we are going to write a fairly difficult ML program and evaluate its running time experimentally, to verify that the analysis matches reality. The problem is multiplication of Binary Integers.

There is a standard way to multiply n-bit numbers; it is analogous to the way decimal numbers are multiplied and involves writing down n "partial products" and adding them. Since the partial products are up to $2n$ bits long (including 0's introduced by shifting), and adding two $2n$-bit numbers takes $O(n)$ time, this algorithm takes $O(n^2)$ time.

This week, you will write a variation of the standard algorithm, designed to be conveniently recursive. It too is an $O(n^2)$ algorithm. Surprisingly, the best algorithm for multiplying n-bit integers takes only time $O(n \log n \log\log n)$. We're not going to be able to implement that algorithm this quarter; it is too complicated. However, next week we will implement a multiplication algorithm that is better than the standard. It is called the "Karatsuba-Ofman" algorithm, and it takes only $O(n^{1.59})$ time. The constant 1.59 is actually $\log_2 3$.

Now, let us consider this week's recursive $O(n^2)$ algorithm. The idea is that we can think of an n-bit number as if it were a two-digit number in base $2^{n/2}$. If we want to multiply an n-bit number by an m-bit number, we can break them each in half, giving us four half-sized numbers, as suggested by the following diagram.



The first number to be multiplied is $A + 2^{n/2}B$ and the second is $C + 2^{m/2}D$. The product is $AC + 2^{m/2}AD + 2^{n/2}BC + 2^{m/2+n/2}BD$. All divisions by 2 should be thought of as integer divisions, throwing away the remainders. We can compute this formula using four half-sized multiplications, some shifts, and additions. Note that you already wrote shift and add functions for Binary Integers, and you should use these functions here.

P1. (20 points) Write a function `mult(x,y)` that multiplies two Binary Integers by the approach described above. Show a listing of your entire program, including functions from previous weeks' assignments that you use now. In a script, show its response to

a) Multiplying `(5,[1,0,1,0,1])` by `(7,[1,1,1,0,1,1,1])`.

b) Multiplying `(0,nil)` by `(1,[0])`.

P2. (15 points) Analyze your program from (P1). No programming is required here. Rather,

a) Draw the expression tree for your function. See Problem (H1) for some comments on how ML functions can have their running time evaluated. Draw a sequence of pattern matches as if-statements, where the pattern-match is performed. In one branch, where the match is successful, the expression associated with that pattern is evaluated. In the other branch, the match is unsuccessful, and the next pattern is evaluated if there is one.

b) Evaluate the running time for each node of your tree for the case that both arguments are Binary Integers of length n, and n is a power of 2. Assume $T(n)$ is the running time of `mult` on two arguments of length n. We will accept something less formal here than in class or book examples. You may state without proof the running time of any functions that `mult` calls. Your result should be an expression at the root node of your tree representing $T(n)$ as an expression involving big-oh functions of n and also involving $T(n/2)$ (without a big-oh).

c) Solve your recurrence from (b). Again, you may be more informal here, perhaps citing FCS for a solution to your recurrence.

P3. (15 points) Measure the running time of your function `mult` on two binary integers of n-bits, for various values of n. Your function `genBinInt` from Assignment 2 is handy for generating large Binary Integers. Hand in a script showing your measurement of the running time of `mult` for n = 100, 200, 300, 400, and 500. How close to a quadratic does your measured running time come? Give the approximate value of a for which an^2 best approximates your measured running times (i.e., a minimizes the largest deviation of your data from an^2). Don't forget to remove the compile time from your measurements before fitting your running times to an^2.

To do the measurement, you need to realize that SML spends some time, perhaps several seconds, compiling your program before it runs it. You should not count the compile time when measuring the time it takes your program to run. How do you know how long SML takes to compile your program? Run it without calling `mult` on any arguments. The time that run takes will be the compile time, which you then subtract from each run in which you follow your code by a call to `mult` with some arguments.

To time a call to SML, you use the UNIX `time` command, followed by the command you want to time, e.g.:

```
time sml <foo
```

if your program is in file `foo`.

The response to `time` is somewhat mysterious. You will get a line like

```
0.6u 0.7s 0:04 31% 1128+1839k 30+1io 127pf+0w
```

The first element says you took 0.6 seconds of "user time," i.e., your program kept the processor busy for that long. The second element says that in addition, the processor spent 0.7 seconds handling the system calls (for disk operations, e.g.) required by your program. The third says that the total elapsed time was 4 seconds, including time the machine was handling somebody else's program. The fourth says that you got 31% of the processor while you were running. The remaining elements tell you about the space used by the program, and how many disk operations of various types were performed on your behalf. If you are curious, a complete explanation can be had by consulting the on-line UNIX manual; type

```
man time
```

In a sense, the appropriate measure of the running time of your program is the sum of user and system time, or 1.3 sec. in the above example. However, system time is often a function of the load on the machine, and can vary for different runs. (User time can vary too, but generally not by much.) In this experiment, let us take only the user time as the "running time."