CS109A Assignment #5
Due 3:15 PM, 2/23/96

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

Written Homework

Note: Many of these problems call only for an answer. If you want partial credit, you should include a formula or other description of how you got the answer.

Also note: The UNIX command `bc` is an infinite-precision desk calculator that responds with the value of any expression when you type it and hit return.

H1. (7 points) Solve the following linear recurrence:

$$T(n) = 5T(n-1) - 6T(n-2)$$
$$T(0) = 2$$
$$T(1) = 5$$

H2. (7 points) In the game of "Hollywood squares," X's and O's may be placed in any of the nine squares of a tic-tac-toe board (a 3×3 matrix) in any combination (i.e., unlike ordinary tic-tac-toe, it is not necessary that X's and O's be placed alternately, so, for example, all the squares could wind up with X's). Squares may also be blank, i.e., not containing either an X or and O. How many different boards are there? (Exercise 4.2.4, FCS)

H3. (7 points) In a class of 200 students, we wish to elect a President, Vice President, Secretary, and Treasurer. In how many ways can these four officers be selected? (Exercise 4.4.2, FCS)

H4. (7 points) One hundred people participate in a lottery. One will win the grand prize of $1000, and five more will win consolation prizes of a $50 savings bond. How many different possible outcomes of the lottery are there? (Exercise 4.6.4, FCS)

H5. (7 points) How many anagrams of the word `little` are there in which the two `t`'s are not adjacent? (Exercise 4.8.6, FCS)

H6. (7 points) How many distinct ways are there to seat 10 people around a circular table? Note that because a round table has no obvious starting point, seatings are not "distinct" if they are rotations of one another.

H7. (8 points) Suppose we have infinite supplies of red balls, blue balls, and green balls. How many ways are there of choosing m balls? Order of the balls is unimportant. You must give a simple formula (and there *is* such a formula!) that works for any m to get full credit. However, we shall give partial credit for algorithms that compute the number or for calculations for specific m.

Programming Problems

This week, we are going to write another program for multiplying arbitrarily long integers using our Binary Integer (BI) representation. When we do, we'll discover

- The simplest algorithm sometimes is not the best algorithm.

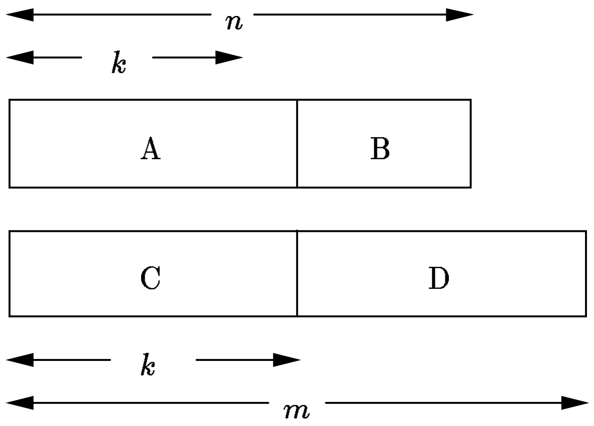
In fact, the obvious $O(n^2)$ algorithm that you wrote for Assignment 4 can be beaten considerably. The best known algorithm has running time $O(n \log n \log\log n)$. Unfortunately, another truism is

- The algorithms with the best big-oh running time often show their speed only for impractically large inputs.

In the world of algorithms for multiplying integers there is, happily, a good compromise: an algorithm that has a better big-oh running time than $O(n^2)$, yet whose efficiency we can see for reasonably sized numbers. The algorithm we shall implement was invented by Karatsuba and Ofman (the "KO Algorithm").[1] When I compared this algorithm with the standard, KO was faster for integers of 400 bits.

The trick behind KO is to arrange to do only three half-sized multiplications instead of four half-sized multiplications as in the standard approach. We pay a price, doing more additions, subtractions, and shifts in KO than in the standard algorithm, but those operations are $O(n)$ each, so they eventually become of negligible cost as n gets large.

Divide the two bit strings of length n and m, respectively, roughly in half as we did last week. The only difference is that this time we must pick a number k which will be the number of low-order bits selected from each of the numbers. If n and m are approximately equal, $k = (n + m)/4$ is about right. However, if one of n and m is much smaller than the other, we should probably pick k to be the smaller. Be careful not to allow k to be negative. The following picture gives the idea. As last week, we assume low-order bits are at the left, as in the BI representation.



The result we desire is $AC + 2^k(AD + BC) + 2^{2k}BD$. To obtain this result by KO, we compute the following three products:

[1] I erroneously referred to this algorithm as "Four Russians" in Assignment 4.

1. AC
2. BD
3. $(A+B)(C+D)$

We now have two of the three expressions (AC and BD) that we need. We obtain the third by

$$AD + BC = \Big((A+B)(C+D)\Big) - AC - BD$$

Note that neither subtraction can produce a negative result (why?), so your proper-subtraction function from Assignment 4 can be used here.

P1. (25 points) Write a function `koMult(x,y)` that multiplies two Binary Integers by the KO algorithm described above. Your function should catch erroneous data that is not caught by lower-level functions that you call. 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 `(2,[1,1])` by `(10,[0,1,0,1,0,1,0,1,0,1])`.

P2. (10 points) Analyze your program from (P1). No programming is required here. Rather,

a) Draw the expression tree for your function.

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 `koMult` 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 `koMult` 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) Compare the running time of your function `koMult` with the running time of your function `mult` from Assignment 4 on two binary integers of n-bits, for various values of n. Hand in a script showing your measurement of the running times of your functions for $n = 100$, 200, ... until the KO algorithm becomes faster than the ordinary method. However, don't go over $n = 1000$, which probably indicates that something is wrong with one of your functions.

How close to $O(n^{1.59})$ (1.59 is $\log_2 3$) does your measured running time come? Give the approximate value of a for which $an^{1.59}$ best approximates your measured running times (i.e., a minimizes the largest deviation of your data from $an^{1.59}$).

Suggestion: Review Assignments 3 and 4 for ideas about debugging complicated functions, using the UNIX `time` function, and eliminating the compile time from your measurements.