CS1112-206/211/212–Fall 2009

Lab 4 Solution

Friday, September 25

1 Binomial Coefficients

(a) This part is easy: just use a single `for` loop and code in the formula correctly.

(b) This part is more challenging. The intuition is that, if $n \leq 5$, then we print all the values of $\binom{n}{k}$, where $k = 1, \ldots, 5$. Otherwise, if $n > 5$, then we need to stop printing after $k = 5$. One way to do this is to simply modify Part (a).

Another way which avoids using `for` and `if` together is to use a `while` loop with condition `k<=n && k<=5`. Since we use `k` in the `while`'s logical expression, we need to instantiate it first. The first `k` is 1, so we just need to have `k=1;` before the `while` loop. Also, because `while` loop does not update `k` automatically, we have to update it ourselves: `k=k+1;` is needed as the last line of the loop.

The main part would look like the following:

```
k=1;
while k<=n && k<=5
    nChooseK=factorial(n)/factorial(k)/factorial(n-k);
    fprintf('%d choose %d = %d\n',n,k,nChooseK);
    k=k+1;
end
```

Yet, another way is to use the MATLAB's built-in function `min` to determine the minimum between n and 5 and use this value as the limit for the `for` loop. This solution is on the course webpage.

(c) Instead of calling `factorial` repeatedly, which—in some sense—is an expensive operation to do, we could use our intelligence to observe the relationship between our current binomial coefficient and the next one. Suppose we know the value of $\binom{n}{k-1}$. We could derive the formula as follows:

$$\begin{aligned}
\binom{n}{k} &= \frac{n!}{k!(n-k)!} \\
&= \frac{n-k+1}{n-k+1}\frac{n!}{k!(n-k)!} \\
&= (n-k+1)\frac{n!}{k!(n-k+1)!} \\
&= \frac{n-k+1}{k}\frac{n!}{(k-1)!(n-k+1)!} \\
&= \frac{n-k+1}{k}\binom{n}{k-1}.
\end{aligned}$$

Now, when we calculate $\binom{n}{k}$, we must have calculated $\binom{n}{k-1}$, so why not use this value? To use the value, we must save it somewhere. The place to save is an additional variable that stores the last binomial coefficient calculated. In fact, we can use the variable originally used to store the value of $\binom{n}{k}$ and keep updating it when we calculate the next coefficient. The solution is on the course webpage.

One might wonder how we could obtain the first binomial coefficient if we need the one before. The answer is quite intuitive: We just have to know this value. More concretely, to calculate $\binom{n}{1}$, we must know $\binom{n}{0}$. For this issue, it is acceptable (and indeed necessary) to assume that $\binom{n}{0=1}$ for any n. (Note that the solution on the course webpage starts from $k = 2$ instead of 1, so the value assumed is n instead of 1.)

(d) This is the most difficult task to understand in the first place, but once we do, it becomes not so bad. Just to clarify: This part does *not* depend on the input n. Instead, we always run from $n = 1$ to 10 and, for each n, print out the number of digits of $\binom{n}{k}$ for $k = 1, \ldots, n$ on the same line. That means the output for different n's should be on separate lines. If programmed correctly, your solution should output 55 numbers that form a triangular shape, with base at the bottom.

A question that popped up was how to print several numbers on the same line. The trick is to use `fprintf`; `disp` does not work because it always prints a new-line character at the end. We do not print the new-line character (`\n`) until we are done with a line. At that point, `fprintf('\n');` should do the job.

The remaining of this Part is more technical and the reader may skip this if preferred. For those who are interested, read on.

Instead of using the approach as in the posted solution, one might try to use Part (c) to help calculate the number of digits. That is, the following program should output correct results:

```
for n=1:10
    % log value of binomial coefficient
    digits=0;
    for k=1:n
        % update log value using formulas:
        % - log10(x*y)=log10(x)+log10(y)
        % - log10(x/y)=log10(x)-log10(y)
        digits=digits+log10(n-k+1)-log10(k);
        fprintf('%d ',floor(digits)+1);
    end
    fprintf('\n');
end
```

As we might have learned from our programming experience so far, the last task to do for programming is testing. To test this program is easy: just run it! Here is the output:

```
1
1 1
```

```
1 1 1
1 1 1 1
1 1 1 1 0
1 2 2 2 1 1
1 2 2 2 2 1 1
1 2 2 2 2 2 1 1
1 2 2 3 3 2 2 1 1
2 2 3 3 3 3 3 2 1 0
```

Obviously, something is wrong here. No binomial coefficients have no digits! What went wrong is the inaccuracy in adding floating-point numbers. As we just learned from class this week, there is a limit on how precise a decimal can be represented in a computer, and this program illustrates the problem! Solutions to this are out of our control; we have to blame the language designer or people who standardized these things, or just put up with it and find another, "correct" solution.

2 Fibonacci Numbers

(a) Easy: Instead of printing in every iteration, move the print statement to after the loop so it prints the final value only once.

(b) This is, again, a `while`-loop program. We just have to change the condition for the loop to limit on the *value* of the Fibonacci number instead of the argument n. (The last value printed is f_{30}.)

(c) Part (b) already sets the upper bound. We just have to add in the lower bound. The first number we start printing is one just greater than 10,000. So, if the value has not exceeded 10,000, do not print it out! ($f_{21}, \ldots, f_{30}$ are printed.)

(d) This is not too difficult, but there is a catch if the programmer is not careful enough. In specifying $\frac{1+\sqrt{5}}{2}$, we have to put parentheses correctly. We also have to make sure to enter the correct error threshold. Last but not least, we should associate f_n and f_{n+1} with `f_old` and `f_cur` correctly, but this totally depends on which values are printed, etc. That is, this part should look like

```
abs(f_cur/f_old-(1+sqrt(5))/2)>0.000001
```

Incorrect expressions will lead to $n = 1476$, $n = 17$, or $n = 15$. (The result is $n = 16$.) Note that the solution on the course webpage is incorrect: The logical expression for the `while` loop should be exactly as above, or (as another fix) is to change to `f_old=0;` instead, but do only one of these, not both. As we see, it is really easy to make a mistake in this kind of problems! Does your program divide anything by zero?

3 Challenge Problems

For convenience, we provide the problems below. The solutions to these problems are posted on the section webpage.

3.1 P3.1.7

Write a script that "draws" the figure below in the *Command Window* using `fprintf` statements. Prompt the user to input an integer n for the number of asterisks on each side of the square. Assume $n > 3$.

```
*****
**  *
* * *
*  **
*****
```

Recall that `fprintf(' ');` prints a single blank (space) while `fprintf('\n');` starts a new line.

Solution: Just to clarify, the figure above is for $n = 5$. Here comes the solution. First, observe that we need to print at all columns in the first and last rows, and the first and last columns always need to be printed. The more difficult part is the diagonal inside. But this is not that hard: The second row is at the second position, the third row is at the third position, and so on. In general, for the i^{th} row, where $2 \leq i \leq n-1$, the asterisk should be printed at the i^{th} column. Lastly, do not forget to print a new line after a line is completely printed.

3.2 P3.2.7

Define

$$\begin{aligned}
t_0 &= \sqrt{1+0} \\
t_1 &= \sqrt{1+1} \\
t_2 &= \sqrt{1+2} \\
t_3 &= \sqrt{1+2\sqrt{1+3}} \\
t_4 &= \sqrt{1+2\sqrt{1+3\sqrt{1+4}}} \\
t_5 &= \sqrt{1+2\sqrt{1+3\sqrt{1+4\sqrt{1+5}}}}.
\end{aligned}$$

Pick up the pattern and develop a program that prints $t_1, \ldots, t_{26}$. A loop is required for each t_k.

Solution: As a hint from the problem statement, a loop is required for each t_k. But we also need to loop through all the k's so this is, yet, another nested-loop problem. As anticipated, the most difficult task for this problem is to determine the pattern. We see that the only special cases occur

when $k = 0, 1$. For these values, it is simple enough to just hard-code the values. Now we concentrate on determining the pattern for the rest.

Because the way function call works (in any programming language), we need to calculate the innermost expression first before calculating outer ones. We start with 1 as running result. Multiply by k, add 1 to this result and take a square root. This is one iteration of the loop. The next iteration takes the running result, multiplies by $k-1$, add 1, and take a square root. The successive iterations does exactly the same with decrementing values of the number we multiply. We stop multiplying after the number multiplied is 2. Then we have the final answer.