Basic C programming skills test

This is a sequence of programming exercises to be completed before you start classes in September. You should bring the assignments with you when you come to New York, either on a floppy or printed. This should include four programs together with sample output. If you bring a floppy, make sure the C code is written in four separate files *in ASCII* (i.e. character or .txt) format. It is very important that you take this seriously. Not only is the project worth 5% of your grade in the course Computing in Finance, but the course instructors (Kishor Laud and Lee Macklin) will assume programming ability at this level from the start of the course. We firmly believe that anyone who has been admitted to the program will have no difficulty learning C at this level.

If you already know C programming, these should be very easy. If you do not, you can do these as you read through a book on programming. The exercises progress through the aspects of programming in the order they occur in Kernighan and Ritchie.

1. Compiling and running a program

This is just to make sure you have the mechanics of editing and running a C program. Write a simple program, based on the "Hello, world" example, that produces the output:

```
(my name)
Financial Mathematics
Courant Institute
New York University
```

2. Variables and control structures

To understand that you know how to use variables and control structures, write a `for` or `while` loop to compute the total discounted payments value of n:

$$P = 1 + r + r^2 + \cdots + r^{n-1} \ . \tag{1}$$

Do not use the formula for the sum of a geometric series (except possibly when you are checking that your program gives the right answer). In your program, compute r^k using $r^k = r * r^{k-1}$ rather than the `pow` function. Give the program the values of n and r using a `#define` statement. The program should print the values of n and r, for example, in the format

```
The discounted value of    20 payments with r =   .900 is   8.7842
```

When this is done, add to your program a feature that halts the program and prints an error message if P exceeds a given value `PMAX`, with `PMAX` defined in a `#define` statement. Now, if I use `#define PMAX 10` and take $r = .95$, the program should produce something like:

```
P exceeded    10 with     13 payments,   r =   .950
```

Do this by adding a few more lines of code. You should hand in the before (without the feature) and after (with the feature) programs. On the job, you will probably spend more time adding features to existing programs than writing programs from scratch.

3. Arrays and procedures

To do this, you will need to understand arrays, pointers, and the ways procedures communicate information. The *normalized* binomial coefficients are

$$B(n,k) = 2^{-n} \frac{n!}{k!(n-k)!} \quad , \tag{2}$$

for k in the range $0 \leq k \leq n$. They can be computed using the relationship that is the basis of Pascal's triangle:

$$B(n,k) = \frac{1}{2} \left(B(n-1,k) + B(n-1,k-1) \right) \; . \tag{3}$$

The formula (3) holds for $k = 1, \cdots, n-1$, and needs to be supplemented with

$$B(n,0) = 2^{-n} \quad , \quad \text{and} \quad B(n,n) = 2^{-n}. \tag{4}$$

When $n = 1$, the conditions (4) define B completely. In this exercise, you will use the recurrence relations (3) and boundary conditions (4) to compute values of $B(n,k)$. Do not use (2) in your program. The main ingredient is a procedure `NextB` that computes all the values $B(n,0)$, ..., $B(n,n)$ (the values for a fixed n and all integers k in the range $0 \leq k \leq n$), given that the values $B(n-1,k)$ for $0 \leq k \leq n-1$ have already been computed. Your procedure should have signature

```
double NextB( double B[], int *firstK, int *lastK, int n);
```

When the procedure is called, the array entry `B[k]` should hold the value $B(n-1,k)$ for k in the range $0 \leq k \leq n-1$. When the procedure returns, `B[k]` should have the value $B(n,k)$ for k in the range $0 \leq k \leq n$. To do this, the procedure will use both (3) and (4). The procedure should return the value of the largest binomial coefficient at level n, that is, $B_{\max} = \max_k B(n,k)$. For large n, most of the binomial coefficients are very small relative to the largest one. The largest one is for k about $n/2$, and they get smaller as k increases or decreases from that value. When the procedure returns, `firstK` should hold the first k value with $B(n,k) > .01 * B_{\max}$ and `lastK` should have the last k value with $B(n,k) > .01 * B_{\max}$.

Test your procedure by having it called by the following main program. It may be interesting to see just how many large B values there are for a given n. Feel free to experiment with larger values of `NMAX`.

```
double NextB( double B[], int *firstK, int *lastK, int n);

#define NMAX 100

int main() {

   double B[NMAX+1];
   double maxB;
   int    n, firstK, lastK,

   B[0] = .5;  /*   Start by putting in the binomial          */
   B[1] = .5;  /*   coefficients for n=1, and k=0 and k=1.    */
               /*   Why do you get a different answer using   */
               /*   1/2 instead of .5?                        */


   for ( n = 2; n <= NMAX; n++ ) {
      maxB = nextB( B, &firstK, &lastK, n);
      printf('n = %5d has max = %10.2f', n, maxB);
      printf(', and large values between %5d  and %5d.\n', firstK, lastK);
    }
   return(0);
 }
```

Hints and comments:

- We are using just one array, while the formula (3) uses two arrays (one for level n and one for level $n-1$). The statement

  ```
  B[k] = .5*(B[k] + B[k-1]);
  ```

  will produce the wrong answer. You need to save the old $B(n, k-1)$.

- To avoid computing 2^{-n} for $B(n, 0)$ and $B(n, n)$, you can use the relation $2^{-n} = .5 \cdot B(n-1, 0)$.

- A simple way to find `firstK` and `lastK` is to use a `while` loop, or a `for` loop with a `break` statement, *after* the $B(n, k)$ have been computed. That is, loop over k at least twice.

- The normalization factor 2^{-n} in (2) allows us to compute to much larger values of n without overflowing the range of floating point arithmetic.

- The binomial tree method for pricing simple stock options leads to a program that is very similar to this one.