EEEE1042 - **Practical 4**
Conditionals and Loops.

1 Pascal's triangle

Pascal's triangle is a mathematical construct named after Blaise Pascal. It takes on the shape of a triangle as shown below:

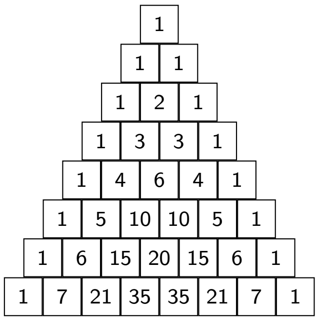

Figure 1: Pascal's Triangle

The edges of the triangle are all set to a value of one, while every internal node in Pascal's Triangle is the sum of the two nodes above it as can be seen in Figure 1. Use the loop constructs taught in class, to write a program that calculates and displays the first N rows of Pascal's triangle, where N is a constant `#define`'d at the start. You do not have to reproduce the exact structure as such positioning of the digits is cumbersome, rather your output can look as in Figure 2

```
1
1      1
1      2      1
1      3      3      1
1      4      6      4      1
1      5     10     10      5      1
1      6     15     20     15      6      1
```

Figure 2: Expected sample display from the Pascal-Triangle program

You may come up with your own method and internal data structures to perform your computations for each row of the triangle. One suggested (although not very memory efficient) structure would be to keep the entire triangle in a 2D array of int's ranging from 1 to N in both dimensions. Then you could compute the i'th row by summing the appropriate two elements in the row $i-1$ above. Save your code to the file `yourName_practical4_EEEE1042.1.cpp` and submit it on Moodle.

2 Prime numbers

1. A prime number is any positive integer greater than 1, that is divisible only by 1 and itself. Note that 1 itself is not defined as a prime. Write a program to calculate and display the first N prime numbers, where N is a constant `#define`'d at the start. As with Pascal's triangle above, you may use any method you wish to try to compute the prime numbers. Here, we provide a possible method that you might consider adopting.

   Define an array of size N to hold your list of prime numbers. Initialize the first prime number to be 2. Iterate one outer `for` loop over the "test numbers" that will be tested to see if they are prime. For each test number, iterate a second `for` loop to check whether it is divisible by every prime number in the existing list. If it is divisible by any prime number in the list, the test-number is not prime. On the other hand if the test-number is not divisible by every prime number in your existing list, then the new test number is a prime and can be added to the list of prime numbers. The first 10 prime numbers generated by the program are shown in Figure 3.

```
0         2
1         3
2         5
3         7
4        11
5        13
6        17
7        19
8        23
9        29
```

Figure 3: Expected sample display from the prime-number program

2. What is the 100,000th prime? How long does it take your program to compute the first 100,000 primes? Is your program able to determine the 500,000th prime? How long does that take?

   In order to time things with your program, you need to first `#include<time.h>` in the header of your program. The `clock_t clock()` function in this library returns the CPU time used by your program (`clock_t` is a type defined in `time.h`, you can treat it similarly to an unsigned int). `clock()` really does return just the CPU time, and does not include the time processing I/O or printing to the screen, which incidentally is slow. So to time your code, you need to take your CPU time before calculating primes:

```
clock_t startTime = clock();
```

After your computationally intensive routine has finished, you can take the CPU time again, find the difference and print the CPU elapsed time with

```
printf("Time to calculate %d primes: %f s\n",N,(float)(clock()-startTime)/CLOCKS_PER_SEC);
```

where the constant `CLOCKS_PER_SEC` is defined in `time.h` and converts from CPU CLOCKs to seconds.

3. You should have found that the algorithm suggested above for computing primes is not very efficient and thus could end up taking a long time to compute the first half a million primes. Can you find a way to make the program more efficient and thereby compute the 500,000th prime in a more reasonable time? Hint: you can do it through making use of the `sqrt` function which also means you will need to `#include<math.h>`. When you include the math header file, you also have to include the math library in order for the linker to know where to find the definition of `sqrt`. You do this by appending `-lm` onto your compile line so it will look something like this:

```
gcc -o practical4.2 practical4.2.c -lm
```

When you are done, you can save your program as `yourName_practical4_EEEE1042.2.cpp` and upload it to Moodle.