Your program must have the following functions:

* ***int\* initializeArray(unsigned int size, unsigned int mode);*** This function allocates space for a one dimensional integer array with *size* elements in it. if *mode* is 0, the values in the array elements are initialized with the value of their index, i.e. A[0] = 0, A[1] = 1, and so on. If *mode* is 1, the values are initialized with random values between -1000 and 1000, inclusive (see the *random(3)*man page).
* ***double sumArray(int\* array, unsigned int size, long\* sum, unsigned int numBlocks, unsigned int step);***This function must sum up the whole array, returning the total sum in the variable pointed to by the third argument. The return value of this function is the total time in **milliseconds** that the summation computation took (see below). Finally, the *numBlocks* and *step* arguments determine the pattern that the summation must use to access the array. The *numBlocks* argument gives the number of blocks the array is divided into; the summation must proceed "simultaneously" through the blocks: i.e., add the first element of the first block, then the first element of the second block, etc. Secondly, the *step* argument is the amount to add to the current index each time the computation is moving on to the next element in each block that it needs to add. With a step of 1, a simple linear summation is performed through each block, and only one iteration through the blocks is needed. With a step of two, first the even-indexed elements in each block are added (indices 0, 2, 4, ...) and then a second iteration goes back and adds the odd elements (1, 3, 5, ...). With higher step values, the same idea is followed; your computation will need to make *step* number of passes through the blocks, each one adding elements that are *step* indices away in each block. The base case of *numBlocks=1* and *step=1* gives a single linear sum of the array. Your program can assume that the array size is an exact multiple of the number of blocks (i.e., the last block won't be an odd sized block).
* ***double sumArrayRandom(int\* array, unsigned int size, int\* sum);*** This function must sum *size* randomly accessed elements in the array. You do not need to worry about whether the random number generator will cause some elements to be accessed more than once, and others to be missed at all. Thus, this is not a true sum of the whole array. But you do need to access *size* number of elements, each one being indexed by a new randomly generated index. As with the other sum function, this one also returns the number of milliseconds that the computation takes.

Each execution of the program will create one array, invoke the two summation procedures on it, and print out the time the two computations took, and the sum values that they returned.

To time the computations in the functions (in order to return the time in milliseconds that they take), you must use the *clock\_gettime()* library function with the CLOCK\_THREAD\_CPUTIME\_ID clock specification. do "man clock\_gettime" to see more information, or search the web.

Your program must also accept (and use) the following command line options:

* *-s #*: Selects the array size to be # elements. Default size is 10000.
* *-m #*: Selects the array initialization mode to be #. Legal values of # are 0 and 1 only. Default is 0. The meaning is the same as the mode parameter on *initializeArray()*.
* *-b #*: Selects the number of blocks to be #. Default is 1. Negative values are not allowed.
* *-p #*: Selects the step size to be #. Default is 1. Negative values are not allowed.
* *-r #*: Selects the random number seed. The seed should be used once, at the beginning of the program. Multiple executions using the same seed should produce exactly the same summation results. If not given, do not seed the random number generator (the man page to srandom() says it will default to 1; that's ok; other random packages might do something else; that's ok, too).

The options can be in any order on the command line. There must be at least one space between the option flag and the option value.

For example, a run of your program could look like:

./sumprog -s 100000 -b 2 -p 4 -r 121 -m 1

will cause it to create a 100,000 element array (randomly initialized), sum it with 2 blocks and a step of 4, and use a seed of 121 for the random summation.

Submit your source code for this assignment. Code quality is important, so even if your program works perfectly, you may receive deductions for poor code quality (including internal documentation -- i.e., comments).