When we implemented the trapezoidal rule using MPI `mpi_trap.c`, we used a loop of receives and additions on process 0 to add up the results obtained by the different processes. When the loop of receives is completed, only process 0 has the complete trapezoidal rule estimate of the area. It’s often the case that a “global sum” is needed by all the processes. We could implement this by having process 0 send the complete result to all the other processes, but this would make the program even less scalable: process 0 would do essentially all the work in both computing the global sum and in sending it to the other processes.

A somewhat better approach is to use something called a ring-pass. In a ring-pass, each process executes a loop of sends, receives, and additions. The destination of the send is the “next-higher” ranked process: process 0 sends to 1, 1 sends to 2, . . . , and process \(p - 1\) sends to process 0. The source of the receives is the “next-lower” ranked process: process 0 receives from \(p - 1\), 1 receives from 0, . . . , and \(p - 1\) receives from \(p - 2\). In the first send, each process sends its “local” result. In subsequent sends, each process sends the result it received in the previous pass through the loop. So pseudo-code might look something like this.

```c
dest = Next_higher(my_rank, p);
src = Next_lower(my_rank, p);
global_sum = temp = my_calc;
/* Make p-1 passes */
for (pass = 1; pass < p; pass++) {
    Send temp to dest;
    Recv temp from src;
    global_sum += temp;
}
```

Note that if there is only one process, the global sum will already be on the only process, process 0. So the body of the loop will never be executed.

The computation of the source and the destination should be familiar from the circular shift program:

```c
dest = (my_rank + 1) % p;
src = (my_rank - 1 + p) % p;
```
For the homework assignment we’ll just compute the global sum of a collection of random ints:

```c
srandom(my_rank);
my_calc = random() % 20;
```

The call to `srandom` “seeds” the random number generator so that each process can get a different value. The call to `random` generates a “pseudo-random” number in the range from 0 to \(2^{31} - 1\), and taking the remainder after division by 20 will ensure that each process gets a relatively small value, and it will be easy to check that the sum is correct.

So each process will do the required MPI setup, and compute its pseudo-random int. Then each process will compute the sum of all the random ints, and each process will print the sum it computed. So there won’t be any input to the program, but every process will print to `stdout`. The output from each process should include the process rank, the process’ random number, and the global sum that it computed. Note that since the processes are all printing to `stdout`, the order of the output will be unpredictable. If, for example, there are four processes, it might happen that the output from process 3 will appear first, then the output from process 1, the output from process 0, and finally the output from process 2.

You should be sure to test your program on the penguin cluster before you submit: there are slight differences in the different implementations of MPI, and we will test your program on the penguin cluster.