```
1)
algorithm is equivalent to recursive prefix sum

initialize array A of size 1000, set every entry to x

function(X):
    create array Y of size  = size(X)/2
    parallelelize:
        for i in [0, size(Y)]
        Y[i] = X[2*i] * X[(2*i) - 1]
    create array Z of size = size(Y)
    let Z = function(Y)
    construct solution for X:
        parallelize:
            solution = {X[0], Z[0]. Z[0] * X[2], Z[1], Z[1] * X[4], ...}
                        (continue this pattern up to size of X)

    return solution

how it works:
    like any recursive function it helps to consider only
    the intermediary step and try to get closer to the solution.
    effectively, the intermediary step turns the problem of finding the
    partial products of the current list into finding the partial
    products of a smaller list.
    the result of the partial products of that small list are then
    interpolated (and multiplied) with the current list to produce the
    partial products of the current list.


analysis (same as that performed in class):
    runtime: T(n) = T(n/2) + a = T(n/4) + 2a = T(n/8) + 3a
        => alogn where a is some constant
        => O(logn)
    work: W(n) = W(n/2) + bn = W(n/4) + bn + bn/2 = W(n/8) + bn + bn/2 + bn/4
        => 2bn
        => O(n) work

    algorithm is optimal. both the parellel and nonparallel (trivial) implementations
        require O(n) work to find the result

    don't know if strongly optimal though


2)
-split list into chunks of size n/k where k is the number of processors
-for each chunk of size n/k, iterate through entire chunk to find max
    -each processor computes this for a chunk of size n/k
```

```
-result is k possible max values:
    iterate through possible max values in a single processor
        to find max.

how it works: effectively is splits the linear problem into k smaller problems
    -it doesn't do much more than this as n >> k means that splitting
    the problem again will have little improvement on efficiency (same
    time analysis would still apply however)
    -the trouble with attempting to optimize further is that every element
    must be read in order to determine the max since the list has
    not been sorted at all.

analysis:
    runtime: O(n/k) (which is basically O(n) for some constant k)
        each process does work in O(n/k) followed by O(k) work on a single processor
    total work: O(n).
        each processor does O(n/k) work, k processors => O(n) work

    total work of parallelized algorithm is same order as non parallel
     therefore it is an optimal algorithm

     however, it is unknown if this is strongly optimal
        (not equipped to perform that analysis)

3)
can turn this into the linked list partial sum problem covered in lecture

initialize every node to have value 1
parallelize: (rounds of the while loop must be synchronized to avoid race condition)
for all curr_node in nodes:
    while next_node is not null:
        next_node.value += curr_node.value
        next_node = next_node.next_node

how it works:
    within each round, a node contributes its value to another
    value further down the chain. Because every node follows the same
    progression of contribution, values aren't double counted due
    to the nature of jumping over sections of nodes. Therefore,
    a single node N will directly contribute its value to a subset of the
    total nodes. The nodes not in that subset (that require a contribution from N
    for the solution to be correct), will have N contributed to them via
    the nodes belonging to the subset of nodes that N contributed to.


analysis:
    runtime: cutting problem in half with each parallel iteration (and each node done in parallel)
        => O(logn)
    total work:
        follows same logic as problem 1.
        do half the number of additions in the current round as the previous round
        => O(n) work

    algorithm is optimal as it requires same order of work as the non parallel version.

4)
void histogram(int *a, int *h){
    for(int i = 0; i < m; i++){
        h[i] = 0; //initialize the array
    }
    for(int i = 0; i < N; i++){ //N is the number of elements that THIS PARTICULAR PROCESS HAS
        h[a[i] - 1]++; //compute the proper values within each process don't know what m is though
    }
    int num_proc;
    MPI_Comm_size(MPI_COMM_WORLD, &num_proc);

    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == 0){  //to prevent unnecessary computation, only the root processor will compute final histogram
        int * rec_buffer = (int *) malloc(num_proc * m * sizeof(int));
        MPI_Gather(h, m, MPI_INT, rec_buffer, m, MPI_INT, 0, MPI_COMM_WORLD);
        for(int i = 0; i < m; i++){
            h[i] = 0;
            for(int j = 0; j < num_proc; j++){
                h[i] += rec_buffer[j*m + i];
            }
        }
    } else{
        MPI_Gather(h, m, MPI_INT, NULL, m, MPI_INT, 0, MPI_COMM_WORLD);
    }
}


5)
with matrix multiplication the sequential problem is O(n^3)
parallelizing this among k^2 processes means that
before their O((n^3)/(k^2)) computation, they must receive
their initial portions of the matricees:
resulting in ((n^2)/(k))log(k) overhead per process.
this startup cost is heavier than the end cost considering the
that matrix multiplication takes two NxN matrices and produces
one NxN matrix. Therefore, we can ignore the cost of communicating
results

plugging this into the isoefficiency equation
C(n^2)klog(k) <= n^3
which gives us
```

```
Cklog(k) <= n

Memory utilization is M(n) = n^3
plugging this into the scalability equation

M(Cklog(k))/(k^2) = (C^3)(k^3)log3(k)/(k^2) = (C^3)klog3(k)


6)
for a list of length n, split the list into chunks of size n/p
    assign each chunk to a processor.

    each processor compares y to x_i and x_j where i and j are the
        respective start and endpoints of the processors sublist

    the processor that finds y to be within its sublist splits
        its sublist into p equally sized regions and assigns
        them accordingly.
    repeat this process until found the proper index of y

the idea is mimick binary search, but the next list is
of size 1/p rather than 1/2 of the current.

analysis:
    runtime: O(logn/logp)
        the algorithm is log base p of n (by same analysis as binary search).
        This assumes that the intermediary step of a processor
        dictating to the others the indices of the next round can be performed
        in constant time.
```