1. (a) $O(\log n)$ algorithm for finding the logical AND of $n$ bits with $n$ processors on an EREW PRAM:

See solution for the next problem. Omit the step where each processor sequentially computes the AND of $\log n$ bits. The efficiency with $n$ processors is $E(n, n) = n / (n \cdot \log n) = 1 / \log n$.

(b) $O(\log n)$ algorithm for finding the logical AND of $n$ bits with $n / \log n$ processors on an EREW PRAM:

Each processor $p_i$ sequentially computes the AND of $\log n$ bits and stores them in $n / \log n$ variables $r_1, r_2, \ldots$. The AND of these $n / \log n$ results are computed in parallel as follows: Each processor $p_i$, $i = 1, 3, 5, 7, \ldots n / \log n$ computes $r_i$ AND $r_{i+1}$ and stores the result in $r_{(i+1)/2}$. By repeatedly "AND-ing" the result bits, we halve the number of bits with each recursive activation; after $\log(n / \log n)$ activations, we obtain the result. The total time is $T(n) = \log n + \log(n / \log n) = \Theta(\log n)$. The best sequential algorithm is linear, so $E(n, n / \log n) = n / (n / \log n \cdot \log n) = 1$.

(c) $O(1)$ algorithm for finding the logical AND of $n$ bits with $n$ processors on a CRCW PRAM:

Assume we have a shared variable RESULT, which is initialized to TRUE.

Algorithm for each processor $p_i$: read $n_i$

if $n_i = 0$ RESULT = FALSE

This clearly takes constant time. Since the best known sequential algorithm is linear, $E(n, n) = n / n = 1$.

2. Find a parallel algorithm $A$ to solve the Boolean Formula Value problem that (a) lacks only code for problem $N$, and (b) runs in poly-log time with a polynomial number of processors. Then if you could find code for problem $N$ that ran in poly-log time with a polynomial number of processors, this would then immediately give you code for the Boolean Formula Value Problem that runs in poly-log time with a polynomial number of processors, since polynomial and poly-log functions are closed under addition, multiplication and composition. So find good parallel code for $N$ is at least as hard as finding good parallel code for the Boolean Formula Value Problem.

3. (a) $O(\log n)$ algorithm for indexing of an $n$-cell array with $n$ processors on an EREW PRAM:

Similar to the following solution, except we have a processor for each array entry. The efficiency is $E(n, n) = n / (n \cdot \log n) = 1 / \log n$.

(b) $O(\log n)$ algorithm for indexing of an $n$-cell array with $n / \log n$ processors on an EREW PRAM:
Assume that you have a processor $P_i$ for array entry of an array
$B[1] \ldots B[n/\log n]$. At step 0, $B[1]$ is initialized to $x$. At step
$\lceil \log_2 i \rceil + 1, 1 \leq i \leq \log n$, $P_i$ writes $B[i]$ to locations $B[2i]$ and $B[2i+1]$. Then $P_i$ copies $B[i]$ to locations $A[(i - 1) \log n + 1]$ to $A[i \log n]$.
The best known algorithm takes linear time, so $E(n, n/\log n) = n/(n/\log n) \log n = 1$.

(c) Here is CRCW code for filling an array. Code for $P_i$, $1 \leq i \leq n$:
read $x; A[i] = x$.

4. Here is how to solve the parallel prefix problem in $\log n$ time with
$n/\log n$ processors on an EREW machine. Each processor $P_i$ is given a contiguous
$log n$ subarray $x_{(i-1) \log n}, \ldots, x_{i \log n}$. Each $P_i$ computes the sum $S_i$ of its
$log n$ sized subarray. This takes $\log n$ time. Then EREW parallel prefix
algorithm from the notes is then applied on $S_1, \ldots, S_{n/\log n}$. Note that
the number of processors is equal to the size of the $S$ list. Hence, in time
$log n$, you have the parallel prefixes of the $S$ list. Then $P_i$ can calculate
the $[(i - 1) \log n + j]$th prefix, $1 \leq j \leq \log n$, as the $(i - 1)$ prefix of the $S$
list plus $x_{(i-1) \log n}, \ldots, x_{(i-1) \log n + j}$.

5. Since multiplication is associative, you can use the EREW algorithm from
the notes for associative operators to solve this problem in $O(\log n)$ time
with $n/\log n$ processors.

6. Assume that we are trying to compute the $n \times n$ matrix $C$ that is the
product of $n \times n$ matrices $A$ and $B$

(a) Compute each of the $n^2$ entries in the resultant matrix in linear time
using the standard formula $C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$.

(b) Use $n$ processors per entry $C_{i,j}$. In constant time compute the $n$
products $A_{i,k} B_{k,j}$, $k \in [1, n]$. Then in $\log n$ time using the di-
vide algorithm to evaluate an associate operator, compute the sum
$\sum_{k=1}^{n} A_{i,k} B_{k,j}$.

(c) $O(\log n)$ algorithm for multiplying two $n \times n$ matrices with $n^3/\log n$
processors on a CREW PRAM:
Assume that we are trying to compute $AB = C$. Each of the
$n^2$ entries of $C$ are computed in parallel using $n/\log n$ processors.
To compute $C_{i,j}$ each processor computes $\sum_{k=a}^{b} A_{i,k} B_{k,j}$ for a sub-
range $a \ldots b$ of $1 \ldots n$ of length $\log n$. These sums are then added
in time $\log n$ using the tree addition algorithm. Assuming (falsely)
that the best known algorithm runs in time $O(n^3)$, the efficiency is
$E(n, n^3/\log n) = n^3/(n^3/\log n) \cdot \log n = 1$.

(d) Before running the algorithm in the previous subproblem, use $n/\log n$
processors per entry in $A$ and $B$ to make $n$ copies of each of these
entries in $O(\log n)$ time. Then when each reach to $A$ and $B$ in the algorithm from the previous subproblem can reach a different memory location.

7. A $(\log n)$ time algorithm for evaluating a polynomial $p(x) = \sum_{i=0}^{n} a_i x^i$ with $n/\log n$ processors on a EREW machine. In time $O(\log n)$ find $1, k, k^2, \ldots, k^n$ using parallel prefix algorithm (we can make $n$ copies of $k$ using problem 3). In constant time compute $a_0, a_1 * k, a_2 k^2, \ldots, a_n * k^n$. In time $O(\log n)$ compute $p(k) = \sum_{i=0}^{n} a_i k^i$ using tree-algorithm (note plus is associative).

8. By expanding the recurrence relation,

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

As matrix multiplication of $2 \times 2$ matrices is associative, and takes constant time, all of the Fibonacci numbers can be computed using parallel prefix.

9. (Solution courtesy of Daniel Cole) We give an EREW algorithm for finding the longest prefix that is equal to a suffix in poly-log time using a polynomial number of processors. We’ll assume that we have $n^2$ processors and we’ll run in about 4 phases (note we don’t try $k = n$):

- Each letter is assigned $n$ processors and we run our copy algorithm from previous homeworks (double then number of copied items at each of the steps) to get $n$ copies of each item. Time $O(\log n)$
- Note that there are $n - 1$ possible answers to this problem so we ”assign” one of these possible $k$ values to each of the $n$ copies, further each copy is ”assigned” a set of $n$ processors. For each set of processors they have some $k$ value and each processor $p_i$ in this set checks if $C[i]$ is equal to $C[n - k + i]$ for $i \leq k$ (some processors don’t do anything). This is checking if a unique character of the prefix is equal to the corresponding unique character of the suffix. Because there is overlap here, we can just have the $n$ processors make a copy of the input and then we’re fine because each character can only appear once in the prefix and once in the suffix. In any case each $p_i$ now has a 0 or 1 bit based on whether or not their character’s matched. Note that each set of $n$ processors does all this in constant time. Thus for each $k$, we need only to know if all the bits are 1’s, if so then the prefix and suffix match and that $k$ is valid, if not the max.
- In this phase we run our algorithm for ANDing bits together. Thus each group of $n$ processors (each with a unique $k$), and their bits in the naive way which takes $O(\log n)$ time and at the end we have a 1 or a 0 for each $k$ value.
• The last phase is to find the max \(k\) value. We only need \(n\) processors here and assume that for each 1 for a \(k\) value in the last phase, instead we just have \(k\) and a 0 for a 0. Thus we are simply trying to find the max value. Again we just do a simple naive parallel max finding algorithm where each processor passes on the max of it’s two assigned numbers and at the next level those numbers are used by half the processors. This will run in \(O(\log n)\) time with \(n\) processors and produce the maximum \(k\) value for the problem.

As we can see, each phase ran in \(O(\log n)\) time assuming \(n^2\) processors and having a constant number of steps, our total time is still \(O(\log n)\).

10. (Solution courtesy of Daniel Cole) We give an CRCW algorithm for finding the longest prefix that is equal to a suffix in constant time using a polynomial number of processors. In this problem we just modify the previous algorithm again assuming \(n^2\) processors (note we don’t try \(k = n\) as previously). Firstly, we do not need the first phase as we can do concurrent read. Thus in the second phase, we do the exact same thing just that every group uses the same \(n\) characters. This takes constant time and again we are left with \(n\) bits for each group of \(n\) processors.

At this point we can run our constant time CRCW AND algorithm that has each processor simply write a 1 to the desired output and then if a processor’s bit is 0 it writes a 0 to that output, otherwise it does nothing. Thus the answer will be 1 if and only if no processor has a 0 bit. Each set of \(n\) runs this algorithm in constant time and thus we have, for each \(k\) whether or not it is feasible. Now we simply need to find the maximum \(k\).

Again here we assume \(k\) instead of 1, which is a constant time update. Now to find the max \(k\), we have \(n\) processors assigned to each \(k\) value and within each group, a processor will first write a \(k\) to the common output and will then look at its group \(k\) and one other \(k\), unique for each processor in a particular \(k\) group. If its \(k\) is bigger it will do nothing else it will write a 0 to the common output for that \(k\) group. Thus at this step, each \(k\) that was 0 will still be 0 but each \(k\) that is not 0 will have been compared to every other \(k\) and if a single other \(k\) was larger then this \(k\) will in effect be 0 now. Thus only one \(k\) will have a value and the rest will be 0. This step takes constant time as we assigned each \(k\) value \(n\) processors.

In the final phase we will simply find the \(k\) value that is not 0 as we have established that all of them are 0 except the largest, whose \(n\) group of processors did not find any \(k\) larger. To do this we assign 1 processor to each of these locations of potential maximum \(k\) values and each processor
first writes a 0 to the common output and then checks their \( k \), if it is not 0 they write it to the common output. This will only be true for one processor thus the common output will contain the largest \( k \). This step was done in constant time as well.

Each step of our process (each \( n \) group tests a \( k \) value, ANDing results in each \( n \) group, clear all \( k \) except the max and find the non-zero \( k \) left over) run in constant time thus our algorithm is a constant number of steps of constant time each for a total runtime that is constant on \( n^2 \) processors.

11. \( O(\log n) \) algorithm for adding two \( n \) bit integers on an CREW PRAM:

Generalize the inductive hypothesis to compute the sum both assuming no carry and assuming a carry of 1. Assume that we want to add integers \( A \) and \( B \) and \( A = A_1 2^{n/2} + A_2 \) and \( B = B_1 2^{n/2} + B_2 \). Now recursively solve the problems of adding \( A_1 \) and \( B_1 \), and of adding \( A_2 \) and \( B_2 \). Depending on whether or not there is a carry from adding \( A_2 \) and \( B_2 \) one can compute the final sum in constant time. Once the number of bits is less than \( \log n \) we can add in log time with one processor. This running time is time \( O(\log n) \) using \( n/\log n \) processors.

12. We explain how to convert the CREW shortest path algorithm into an EREW algorithm. Initially make \( n \) copies of the \( n \times n \) matrix holding the edge distances. This takes \( O(\log n) \) time with \( n^3 \) processors since the distance matrix has \( n^2 \) entries. As only \( n \) processors read an edge distance in any one step, this means that concurrent reading is no longer necessary so we have an EREW algorithm. Note that within each iteration of the \( \log n \) rounds of distance updates you must again do this \( O(\log n) \)-time copying procedure so that none of the distance info being read from the copies of the distance matrix becomes stale. And note that the entire algorithm still runs in \( O(\log^2 n) \) time.

13. (Solution courtesy of Daniel Cole) Here we explain how to have the shortest path algorithm actually compute the shortest paths. Here the idea will be to save a bit of extra data during the main algorithm and then rebuild all the paths at the end. It doesn’t seem like we can build the paths as we go because although we know after each iteration that we have completed the shortest paths of length up to \( 2^{\text{iteration} - 1} \) in length, we don’t know which paths these are necessarily. Thus we have to wait until the end.

For extra data we will keep the ”midpoint” used to create the shortest path, i.e. the common point of the two shortest paths that we used to create the new shortest path. We will also keep number of hops in the path. We can do both of these items in our original algorithm without
adding any time as they take an addative constant longer than just keep-
ing track of the length. To see this note that when we are doing the min
of a set of combined paths for some \(i, j\) path, we use \(n\ "midpoints"\) each
with a unique value \(k\). When we do this min operation we can just keep
the winner’s \(k\) value as well as his length. To keep the hop length we
can see that it can easily be created initially in the first round as 1 for
any paths that are created and thereafter we can just calculated it as the
hop-length of \(i, k\) plus that of \(k, j\) for the winning sub paths. Thus from
now on we will assume that in addition to having shortest path length
in our two-dimensional array, we also have the \(k\) value used to create the
path as well as the number of hops on the path.

After our EREW algorithm from the last question, modified according to
our requirements, finishes we can commence constructing the paths them-
selves. We essentially follow the same procedure as above except that we
know the optimal paths and we simply need to construct them. The idea
thus is again to run for \(\log n\) iterations guarenteeing that after each iter-
atation we complete all paths of length \(2^{\text{iteration}-1}\). We have two phases,
the path output phase and the path copy phase. The output outputs the
completed paths, the copy phase, copies these paths so that they may be
used in subsequent rounds.

- **Output Phase:** Each entry, \(i, j\), in the table is assigned \(n\) processors
  and in iteration \(p\) any entry with hop-count \(2^{p-2} + 1\) to \(2^{p-1}\) will
  be calculated. This will be done by copying unique copies of the
  previously finished paths \(i, k\) and \(k, j\) to the proper output location
  for \(i, j\). We know that these paths were finished in the last round
  thus they are available. The copy of these paths takes at most \(\log n\)
time per path as each path can be at most \(n\) in length and we can
copy that in \(\log n\) time with \(n\) processors which is exactly what we
have per entry. In the next phase we will explain the unique copies
of \(i, k\) and \(k, j\).

- **Copy Phase:** At this point we have created all paths of hop-count
  \(2^{p-2} + 1\) to hop-count \(2^{p-1}\) we just need to make these paths available
to the next iteration. Note that for a path \(i, j\) that it can only be
  used in the next iteration as either \(i, (k = j)\) or \((k = i), j\). Because \(k\)
is fixed in both cases the other half of the path can only be one of \(n\)
paths meaning our current \(i, j\) path can be used at most \(2n\) times so
we’d like \(2n\) copies so that everyone can have their own unique copy
in phase 1 of the next iteration.
Now we must consider that we want to make \(2n\) copies of an unknown
number of paths that can be up to \(2^{p-1}\) in length. But consider that
in each round we double the number of paths that are finished and even if we have the max possible $2^{p-2}+1$ paths all of length $2^{p-1}$ then we still only need $2n$ copies of $\frac{n^2}{2^{p-1}}$ paths of length $2^{p-1}$ which just gives that we need order $n$ copies of data order $n^2$. By assigning $n^2$ processors for each final copy of the data we can make all our copies in $\log n$ time by our standard doubling of data each round. Further we can identify these path copies by having one copy for all possible $i$ and one copy for all possible $j$ so that in phase one the processors will know where to look.

14. Use the same method as used for the minimum edit distance problem, except with different weights on the arcs, e.g., 0 on horizontal/vertical arcs and $-1$ on diagonal arcs.

15. (Solution courtesy of Daniel Cole) Here we just create a graph and then run the all pairs shortest path problem from class followed by a step to extract the minimum edit distance. First we construct the graph:

- Consider an $n$ by $n$ matrix, $M$ where each entry is a vertex and $[i, j]$ has $B[j]$ as the column and $A[i]$ as the row. We’ll assign a single processor to each vertex to create the edges going out of that vertex. Processor $p_{i,j}$ will look at $A[i]$ and $B[j]$. If they are equal, then it will create an arc from $M[i,j]$ to $M[i+1, j+1]$ with cost 0. If they are not equal it will create 3 arcs:
  - $M[i,j]$ to $M[i+1, j]$ with cost 3 or more generally, $\text{cost}(OP_{\text{delete}})$.
  - $M[i,j]$ to $M[i, j+1]$ with cost 4 or more generally, $\text{cost}(OP_{\text{insert}})$.
  - $M[i,j]$ to $M[i+1, j+1]$ with cost 5 or more generally, $\text{cost}(OP_{\text{replace}})$.

- Notice that we need an extra row and an extra column which we place in the $n+1$ position for both, i.e. we have an extra $n+1$ row and $n+1$ column. These are needed because comparing $A[n]$ with $B[i]$ or $A[i]$ with $B[n]$ would cause edges to go outside our $n$ by $n$ matrix. But consider what this means; it means that we have either removed or replaced all letters in $A$ ($n+1$ row) and are left with some of $B$ left or that we have covered all of $B$ ($n+1$ column) through insertions or replacements and are left with some of $A$ left. If the first case we just need to insert the rest of $B$ thus we simply add edges from $M[n+1, j]$ to $M[n+1, j+1$ with cost 4 ($\text{cost}(OP_{\text{insert}})$). In the second case we just need to remove the rest of $A$ thus we add edges from $M[j, n+1]$ to $M[j+1, n+1]$ with cost 3 ($\text{cost}(OP_{\text{delete}})$). We can assign a single processor to each of these entries as well.

Conceptually we think of the minimum edit distance problem as starting at $M[1,1]$ and making decisions on conversion. For the first entry the
optimal either removes \( A[1] \) in which case \( A[2] \) is now being compared to \( B[1] \) hence we go down. Likewise for inserting or replacing. Once we’ve reached \( M[n + 1, n + 1] \) all letters in \( B \) have been matched (either through insert or replace) and we’ve made decisions about ever letter in \( A \) (either through delete or replace). And that’s the key, replace handles both a letter in \( B \) and \( A \) while insert and delete only handle a letter in one string. Thus once we run all pairs shortest path, we need to simply find the minimum distance between point \( M[1, 1] \) and \( M[n + 1, n + 1] \).

The construction of this graph takes constant time on a CREW machine with \( n^2 + 2n \) processors as each processor is assigned a single vertex and creates arcs out of that vertex. If there is a concurrent write issue with where the arcs are stored then we can just do it in two shifts such that there is a gap between the head and tail of any arc. This will still take a constant amount of time. In either case, we have just constructed a graph such that the shortest path distance from \( 1, 1 \) to \( n + 1, n + 1 \) is in fact the minimum edit distance. Thus we can run the all pairs shortest path algorithm using \( n^2 \) processors and get the result in \( O(\log^2 n) \) time using our in-class algorithm for all pairs shortest path. Then we can look up the shortest path distance from node \( 1, 1 \) to \( n + 1, n + 1 \) in constant time, thus our run time is bound by the all pairs shortest path computation making our run time \( O(\log^2 n) \) with \( O(n^2) \) processors.

16. (Solution courtesy of Daniel Cole) Here we would like a CRCW algorithm for merging that runs in constant time. Assume we have two sorted arrays of length \( n \), namely, \( x_1, x_2, \ldots, x_n \) and \( y_1, y_2, \ldots, y_n \). We give an algorithm for placing \( y_i \) correctly in its final location in the \( 2n \) output array in constant time.

- We use \( n \) processors and each processor is assigned an \( x_j \) value. We have an array, \( A_i[n] \) that we initialize to all zeros in constant time. Each processor compares \( x_j \) and \( y_i \) and if \( y_i > x_j \), then set \( A_i[j] = 0 \) else \( A_i[j] = 1 \). This takes constant time.
- \( A_i[] \) now has a 1 for every \( x_j \) that is larger than \( y_i \) and a 0 for all smaller. Thus, all 0 values represent \( x \) values that are less than or equal to \( y_i \). Each processor, \( p_j \), now looks at \( j - 1 \) and \( j \) and only one processor will see that \( A_i[j] = 1 \) and \( A_i[j - 1] = 0 \) as the \( x_j \) values are in sorted order. Thus the final position of \( y_i \) is in index \( (i - 1) + (j - 1) + 1 \) (index from 1) of the output array because we know \( i - 1 \) \( y \) values are smaller than \( y_i \) and \( j - 1 \) \( x \) values are smaller than \( y_i \). Because only a single processor knows this, it can safely write \( y_i \) to the correct output location.

Creating and populating \( A_i[] \) with 1’s takes constant time and finding the
largest \( x_j \) that is smaller than \( y_i \) (the largest indexed 0) took constant time, thus we have placed \( y_i \) in constant time. Thus we can simply assign \( n \) processors to every \( x \) and \( y \) value to place each value in the output array in parallel in constant time. Note that we should write a 0 for \( x > y \) instead of \( x \geq y \) to give the \( y \) values precedence over \( x \) values and avoid concurrent writes and empty entries in the output. We thus need a total of \( O(n^2) \) processors and our run time is \( O(1) \).

17. (Solution courtesy of Daniel Cole) Here we give a parallel algorithm that finds the max of \( n \) numbers in \( O(\log \log n) \) time with \( n \) processors on a CRCW PRAM. Here we simply break our problem up into \( \sqrt{n} \) subproblems of \( \sqrt{n} \) size each and recursively call, in parallel, our max function on each of these chunks with \( \sqrt{n} \) processors each. We receive a single max value from each of these \( \sqrt{n} \) calls and because we have \( n \) processors we can solve the max problem for \( \sqrt{n} \) values in constant time with \( n \) processors using the method from class, i.e., we have processors equal to the number of values squared. Thus our recurrence relation is \( T(n, n) = T(\sqrt{n}, \sqrt{n}) + C \), where \( C \) is the constant time max parallel algorithm on CRCW. To solve this we just guess \( T(n, n) = C \log \log n \) to get \( T(n, n) = C \log \log(\sqrt{n}) + C = C(\log(\frac{1}{2} \log(n)) + C = C(\log(\frac{1}{2}) + \log \log n + C = -C + C \log \log n + C = C \log \log n \) which is what we guessed. Thus we have a running time of \( O(\log \log n) \).

18. Here we give a parallel algorithm that finds the max of \( n \) numbers, all of which are in the range 1 to \( n \), in constant time on a CRCW-priority. Assume we have an array, \( A \), of size \( n \), and denote a processor’s integer identifier as \( p_i \) where \( 1 \leq i \leq n \). Given \( n \) processors on a CRCW priority machine, we give the following code, for processor \( p_i \), to find the maximum of numbers \( x_1, \ldots, x_n \), in the range 1 to \( n \):

\[
\begin{align*}
A[p_i] &= 0; \\
A[x_i] &= x_i; \\
\text{If } A[n-p_i+1] \neq 0 \\
\text{Answer} &= A[n-p_i+1]
\end{align*}
\]

19. (Solution courtesy of Daniel Cole) Here we give a parallel algorithm that finds the max of \( n \) numbers in the range 1 to \( n \) in \( O(1) \) time with \( n \) processors on a CRCW PRAM. We perform the following steps:

- Each processor has a number and there exists some predetermined array, \( A \), of size \( n \), with all zeros. Processor \( p_i \) looks at its number, \( x_i \), and writes a 1 to \( A[x_i] \). After this step \( A \) has a 1 for each unique number in the input. Note that only 1’s are written so CW with identical writes is not violated.
• There exists an array, \( R \), of size \( \sqrt{n} \), with all zeros. Each entry, \( j \), represents whether or not \( A \) has a 1 in any entry from \( A[j \ast \sqrt{n}] \) to \( A[(j + 1) \ast \sqrt{n}] \). Thus each processor \( p_i \) looks at \( A[i] \) and if \( A[i] == 1 \), writes a 1 to the appropriate entry in \( R \). At this point \( R \) has a 1 for each of the \( \sqrt{n} \) divisions of the input that has at least 1 number.

• Now the \( n \) processors perform a max operation on the array \( R \). In this case "max" simply means finding the largest index in \( R \) with a 1. The array is only \( \sqrt{n} \) in size and because we have \( n \) processors we can solve this in constant time. Thus at the end of this step we know which \( \sqrt{n} \)-sized division of \( A \) contains the largest number.

• In the final step we simply perform the same "max" operation on the \( \sqrt{n} \)-sized division of \( A \) found in the previous step. Again we have processors equal to the number of entries squared so we can perform this in constant time as before. The value found is the right-most 1 in the right-most \( \sqrt{n} \)-sized division of \( A \) with any 1’s in it, or in other words, the right-most 1 in \( A \). By our first step, this is also the largest value in the input thus the max value.

Each of these steps has taken constant time, because we have a constant number of steps our total time is constant.

20. No solution given.

21. (Solution courtesy of Daniel Cole) We consider the problem of how to number the leaves in a binary tree. Create a list representing the Eulerian Tour with each leaf having a value of 1 and all other nodes having a value of 0 then perform parallel-prefix (pointer doubling) and each leaf will have a value equal to its left to right leaf order. Note that we don’t care about the number on internal nodes and they can just ignore the numbers when we’re all done, this is possible because a node knows whether it is a leaf or not (whether or not it has children). To see why this method is correct, consider the order in which the leaves are visited in a Eulerian tour: the left-most is visted first, followed by the second from left, with the right-most leaf being the last leaf visited. Thus our list will have the leaves in left to right order and because they are leaves they will only be in the list one time. Because they are the only list nodes with numbers their final numbers will just be a count of how many leaf nodes were previously in the list plus one. This is exactly the same as their left to right order number in the binary tree.

As this method is identical to what was done in class, albeit with a different numbering scheme, construction of the list takes \( O(\log n) \) with \( n \) processors, and the pointer doubling problem takes \( O(\log n) \) with \( n \) processors, giving us a total time of \( O(\log n) \).
22. No solution given.

23. (Solution courtesy of Daniel Cole) We consider the problem of how to evaluate an expression tree consisting of the four standard arithmetic operations: addition, subtraction, multiplication and division. The class of functions used in class to solve this problem, \( ax + b \), for subtraction and addition works here for multiplication as any constant, \( c \), times \( ax + b \) is \( acx + bc \) which is also of the form \( ax + b \) so that does not cause a problem. Division does, however present a problem. Dividing \( ax + b \) by \( c \) produces \( \frac{ax}{c} + \frac{b}{c} \) which is still in the correct form, however dividing by a constant is not the same as dividing a constant by a function as division is not commutative. Thus in our problem we must handle the case of \( \frac{c}{ax+b} \) which is not a function of the form \( ax + b \).

The class of functions we choose to solve this problem is \( ax + b cx + d \). Note that we can meet all of our requirements:

- The identify function: \( f(x) = x = \frac{1x + 0}{0x + 1} \)
- Constants: \( c = \frac{0x + c}{0x + 1} \)
- Addition with constants: \( \frac{ax+b}{cx+d} + e = \frac{ax+b}{cx+d} + \frac{e(cx+d)}{cx+d} = \frac{(a+e)c+(b+ed)}{cx+d} \)
- Multiplication with constants: \( \frac{ax+b}{cx+d} \times e = \frac{ax+bc}{cx+d} \)
- Division with constants: \( \frac{ax+b}{cx+d} \div e = \frac{ax+be}{cx+d} \times \frac{cx+d}{ax+b} = \frac{cex+de}{ax+b} \)
- Composition: \( f(x) = \frac{ax+b}{cx+d}, \ h(x) = \frac{ex+f}{gx+i}, \ f(h(x)) = f(\frac{ex+f}{gx+i}) = \frac{ae+bf}{ce+df} \times \frac{aZ+bY}{cZ+dY} = \frac{(ae+bf)Z+(af+be)}{(ce+df)Z+(cf+di)} \)

Note that subtraction can be done directly or as a multiplication by \(-1\) and an addition both of which have already been defined. We have show that our functions have all the necessary properties for the \( O(\log n) \) algorithm described in class that halves the number of leaves each round. Note that because all of our operations are constant time operations and because the functions can be represented in constant space, they can be used with the algorithm given in class to give an \( O(\log n) \) timed algorithm using \( n \) processors for a tree with addition, subtraction, multiplication, and division.

24. No solution given.

25. No solution given.