Problem #1

a.) Assume the mesh grid is a d-dimensional architecture with k processors along each dimension.

- **Total # of processors**: $k^d$
- **Diameter**: $d(k-1)$
- **Max # of links per processor**: $2^d$

**Justification**

The total processors increases by a factor of $k$ for every new dimension that is added. This fact yields a closed form for calculating this number of $k^d$.

The diameter is traversed when going from one "corner" to another "corner". To completely traverse an entire dimension it takes $k-1$ steps and this must be done for each of the d dimensions. This yields the proposed closed form solution of $d(k-1)$.

The max links can be determined by looking at the interior nodes of the d-dimensional grid. These nodes allow someone starting at this processor to travel in either direction in each of d dimensions. This means there must $2^d$ links at such a node.
b.) Since \( P = K^d \) and the diameter is \( d(K-1) \) we want:

\[
\begin{align*}
\log_2 P &= d(K-1) \\
\log_2 K^d &= d(K-1) \\
\Rightarrow 2d \log_2 K &= d(K-1) \\
K &= 2^{(K-1)} \\
2K &= 2^K
\end{align*}
\]

This is true when \( K = 1 \quad 2^1 = 2^1 \)
and when \( K = 2 \quad 2^2 = 2^2 \)
and the dimension is irrelevant in both cases.

The max # of links per processor in this architecture is \( d \) because when \( K = 1 \) the max # of links per processor is 0 for all dimensions. This leaves only the case when \( K = 2 \) and the max # of links per processor in this case is \( d \).

Problem #2

The number of nodes in an \( n \)-dimensional hypercube is \( 2^n \) and there are always \( 2 \) \( K-1 \) dimensional hypercubes embedded in a \( K \) dimensional hypercube by its very construction. This means to find the number of \( K \) dimension hypercubes where \( K \in (0, n] \) in an \( n \) dimension hypercube you can divide \( 2^n \) by \( 2^K \) and this will yield the number of disjoint copies of a \( K \) dimensional hypercube embedded in
Finding the embedded hypercubes $n = 3, k \in \{1, 2, 3\}$

To find the 1-dimensional hypercubes in this 3-d hypercube:
- Look at the LSB of the pid and match adjacent 0/1 pairs.
- Since each 0 is connected to one and only one 1, this procedure will work, and yields all 4 disjoint 1-D hypercubes.

To find the 2-d hypercubes in this 3-d hypercube:
- Look at the two LSB of the pid. Start at each 00 pid and follow an edge to a pid that differs by one bit (has a Hamming distance of 1) until you have returned to your starting point.

These procedures essentially pick a node not already in an embedded hypercube of smaller or equal dimension and traverses the edges inside this embedded hypercube until it gets back where it started. If some nodes still remain outside of an embedded hypercube, this procedure will continue until all the embedded hypercubes of lesser or equal dimension have been located.
Problem #3

9. Assume that the N/2 processors (per sequence) will each operate as previously described in class, where each processor stores two elements and are configured in a systolic array.

Hypothetical input:
$S_0 = 3, 4, 9, 12, 19, 99, 103$
$S_1 = 3, 11, 13, 21, 23, 33$

**First Stage**
- Read in input
- Merge Proc. Element is idle

**Second Stage**
- Merge Proc. Element extracts the min from each and compares

$1, 2, 3, 4, 9$
$10, 11, 12, 13$
$17, 23, 99$
The smaller of the two is placed into the tail of the sorted list. This continues until the list of $2N$ values is sorted. The new processor must only be able to compare two elements and write the smaller one to memory and store the larger temporary in local memory until the next time step. This process is "cheap" in terms of its complexity because it has a need for only limited functionalities.

\[ T_p(N) = O(N) + O(N) = O(N) \]

\[ \text{Input stage} \quad \text{output stage} \]

\[
\text{Work} = (Nz + N/z + 1) \cdot T_p(N)
= (N+1) \cdot O(N)
= O(N^2)
\]

\[
\text{Speedup} = \frac{T_1^*(N)}{T_p(N)} = \frac{O(N + \log N)}{O(N)} = O(\log N)
\]

where \( p = N+1 \) and \( N = p-1 \)

\[
\text{Speedup} = O(\log P)
\]

The work efficiency of this algorithm is

\[
(2N) = \Theta(N \cdot N + N \cdot \log N) = \Theta(N \log N)
\]
Because the overhead is $\frac{N}{\log N}$ and this term is clearly $O(N)$ because $\log N \geq 1$ for $N \geq 3$ and $\frac{N}{\log N} = \Omega(N)$ where $0 < c < 1$. This means $\frac{N}{\log N}$ is in between $N^c \leq \frac{N}{\log N} \leq N$. This means the algorithm is too large to be polynomially efficient but it is less than $N^c$ so it is somewhat efficient.

**Problem #4**

(a) These represent which signal came which at the 1st stage of this decision made at this 1st stage.

(b) Omega networks utilize source routing, where each bit of the address corresponds to a decision made at each stage of the omega network. The signal can travel out of one of two directions leaving a switch at any given
level. This means that in order to route $P$ processors to $P$ memories, this routing scheme requires $\log_2 P$ stages each with $P/2$ switches to handle the routing decisions. My scheme routes 0 to the upper output and 1 to the lower output.

Example: $P = 4$

![Diagram of routing network]

Stage 1: Hypothetical routing from $P_1$ to address 11.

At stage 1 the LSB is checked, it is a 1 that output is chosen.

At stage 2 the MSB is checked, it is a 1 that output is chosen.

The destination is correctly reached.

In order to perform associative operations in such a way that utilizes the embedded binary tree structure of the omega network is to mimic the tree based computations we did in the EREW PRAM model.
If the switches are given more computational abilities, they can perform meaningful work as they are transmitting this data from memory to any given processor. (Most likely, $P_0$ will get the result.)

For instance, if a processor wanted to add $N$ values spread throughout shared memory, each processor could request one memory value, and as these are accessed on the way back a converse cast-type algorithm could be used to have their "sum" propagate through the network and converge to the one processor chosen to receive the sum.

This approach takes $\log N$ time for each memory access, and these are done in parallel. Then the convergence to compute the sum also takes $\log N$ time. The only upgrade is needed in the processors used to perform the switching. These must now be able to "compute" by performing an associative operation on two values as they route messages back from the memories to the processors.