1. Bernard Chazelle described the following almost-trivial data structure, called a *window list*, to store a set I of real intervals. A window list is a partition of the real line into disjoint *windows*; each window w has a pointer to a linked list of the intervals $I_w \subseteq I$ that intersect w. (Any interval in I that intersects more than one window is stored multiple times.) Given a real value x, we can determine which intervals in S contain x by finding the window w containing x and then scanning through the corresponding list I_w.

   For any window w, we can partition the intervals I_w into *ending* intervals E_w, which have at least one endpoint in w, and *crossing* intervals C_w, which don't. A window list is *efficient* if $|\#C_w - \#E_w| \leq 1$ for every window w.

   (a) Prove that for any set of intervals, an efficient window list exists.

   (b) Analyze window lists in internal memory. Given a set of N intervals, show that an efficient window list using $O(N)$ space can be constructed in $O(N \lg N)$ CPU time, so that any stabbing query can be answered in $O(\lg N + K)$ CPU time, where K is the number of output intervals.

   (c) Now generalize window lists to external memory. Given a set of N intervals, show that an efficient window list can be constructed in $O(n \log_m n)$ I/Os and stored in $O(n)$ blocks, so that any stabbing query can be answered in $O(\log_B n + k)$ I/Os, where K is the number of output intervals. *[Hint: Beware of nearly-empty blocks.]*

   (d) Show that the same efficient window list can also be used to answer intersection queries ("Which intervals intersect this query interval?"), containment queries ("Which intervals contain this query interval?"), and reverse containment queries ("Which intervals are contained in this query interval?") in $O(\log_B n + k)$ I/Os.

2. In the early 1980s, Jon Bentley discovered the following technique, called the *logarithmic method*, to modify internal-memory data structures to support efficient insertions. The main idea is to partition the set of N elements into several subsets of different sizes and build a data structure for each subset. In Bentley's original scheme, the subset sizes are determined by writing N in binary—if the ith bit of N is 1, exactly one of the subsets has size 2^i, so there are at most $\lg N$ subsets altogether. Suppose the original structure uses $O(N)$ space to store N objects, can be built in $O(N \lg N)$ time, and supports queries in $O(\lg N)$ time.

   - The new structure uses $\sum_{i=0}^{\lg N} O(2^i) = O(N)$ space.
   - To answer a query, we simply query each of the component data structures in turn and combine the answers. The total query time is $\sum_{i=0}^{\lg N} O(\lg 2^i) = \sum_{i=0}^{\lg N} O(i) = O(\lg^2 n)$.
   - Inserting a new element mirrors incrementing a binary counter—we find the first i such that there is no subset of size 2^i, and then combine the new element and the elements in all subsets smaller than 2^i into a new structure of size $1 + \sum_{j<i} 2^j = 2^i$. This insertion requires $O(i2^i)$ time. Since we need 2^i insertions to create a new structure of size 2^i, the *amortized* cost of a single insertion is $\sum_{i=0}^{\lg N} O(i) = O(\lg^2 N)$.

   Generalize the logarithmic method to the external memory setting. Given an external data structure that uses $O(n)$ blocks, can be built in $O(n \log_m n)$ I/Os, and supports queries in $O(\log_B n)$ I/Os—for example, external window lists!—modify it to support efficient insertions. What are the amortized query and insertion times for your modified structure? *[Hint: For each i, you should have at most one subset whose size is between B^{i-1} and B^i.]*

3. Describe an algorithm that computes the minimum spanning tree of a weighted, undirected, connected graph that is represented by a *weight matrix*. The input to your algorithm is a $V \times V$ matrix W, where w_{ij} is the weight of the edge from vertex i to vertex j, or ∞ if there is no such edge. For full credit, the running time of your algorithm should be $O(V^2/B)$—linear in the size of the adjacency matrix.

*4. Describe an efficient external *topological sorting* algorithm. Given a directed acyclic graph as input, your algorithm should label the vertices with integers from 1 to V so that for any edge $u \to w$, the label of u is smaller than the label of w. *[Hint: Topological sorting is usually done in internal memory via depth-first search. Full credit will be given for an algorithm that runs in the same time as the external DFS algorithm of Buchsbaum* et al.*, but this may not be the most efficient approach.]*