Design and Analysis of Algorithms – CSE 101
Using Data Structures to Design Efficient Algorithms

High-level strategies vs. efficient algorithms Many times, an algorithmic idea is presented as a high-level strategy; for example, Kruskal's algorithm: "At each step, we add the edge of least cost that connects two new components". This description does not answer some basic questions: How do we find this edge? How do we test whether an edge connects two components? Sometimes there are obvious answers to these questions in isolation. If we just had to do this once, we would scan through all edges, and for each test whether it connected two components using a Depth First Search algorithm. But that would take a fair amount of time, $O(mn)$ where m is the number of edges. Plus, Kruskal's algorithm says to do this for n steps, so it would get another factor of n in there. This kind of distinction is particularly common for greedy algorithms, since dynamic programming and divide-and-conquer algorithms usually specify the order to do the computation.

The key observation is that the work done in different steps isn't unrelated work. The graph where we are testing connectivity changes only slowly, and is built up gradually by the edges we add. We're free to restructure the algorithm, so that the same work is done, but in a different order. For example, say that we put the edges in a sorted list, sorted by cost. Then we could keep an index in that list as a marker of the edges we have looked at. We wouldn't have to look at any edge twice, because once an edge's endpoints are in the same component, they stay that way. Thus, we'd have to do at most $O(m)$ same component checks total, not for each step of the algorithm. This doesn't change the strategy, and hence doesn't change the correctness, but dramatically affects the time analysis.

Also, because the relevant structures are only changing gradually, we can use data structures to avoid duplicating work. We need to identify what kind of structure is used in the strategy, what information the algorithm needs at each step, and how the structure might change at each step. This points to the appropriate data structure operations. Either we can use a standard data structure to support these operations, or maybe develop our own. Many of the standard data structures were first thought up to speed up a specific algorithm, but then were found to be useful for a variety of different algorithms.

Basic Design Steps

1. Is there some useful pre-processing that can be done? In other words, would it be easier to perform the strategy if the input is in some format? Then your algorithm's first step could be to put it in that format. Typically, this involves something like sorting the input according to some field. Don't just assume the input is in the format; specify that your algorithm puts it in the format as the first step. You have to include the pre-processing time in the overall time analysis. Sometimes it is the dominating term. In Kruskal's algorithm, we preprocess by sorting edges according to cost, an $O(m \log m)$ operation. This turns out to be more expensive than the rest of the algorithm.
2. Re-structure the algorithm taking advantage of the format. Here, we replace scanning through all edges to find the minimum at each step with, " At each step, check the next edge to see if it connects two new components. If so, add it. If not, skip to the next edge." This is a different notion of step, but is really the same strategy: the next edge that eventually gets added is the minimum cost edge that connects two new components.
3. Identify structures that come out of the strategy. What is the strategy defined in terms of? Here, we want to know, repeatedly, whether the endpoints of an edge are in the same component. So the structure that matters is how the graph partitions the nodes into components. Each node is in exactly one component. Thus, the relevant structure is a partition, a collection of disjoint sets whose union is an underlying set of elements.
4. Identify what information we need to know at each step of the strategy. Use this to define Access operations to the data structure. At each step, we are given two elements, and we need to know whether they are in the same component. The simplest way to do this is to have a name for each component, and given an element, get the name of its component. This

gives us our notion of Access operation : We need to be able, given a node x, to find a name for the set in the partition that x is in.

5. Identify how the structure changes in each step. This defines Update queries. Here, if we don't add an edge, the set of components doesn't change. If we do, then the two components get merged into one. So we need a Merge operation that replaces two components with their union.

6. Re-write the algorithm in terms of these operations. How many times is each operation used? Here we do a Merge when we add an edge, so $n - 1$ times. We do two find set operations every time we consider an edge, for a total of $2m$ times.

7. Find a data structure that supports these operations. Some common, easy data structures: If the strategy is "Find any element with property blah", we frequently want to put all such elements in a list or stack. To find such an element, we just use the head of the list. "Find the least (greatest) element with property blah": often a heap of such elements is useful. For "Find the middle element" or "Find the first element greater than x", we might want a binary search tree or to get guaranteed performance, a B-tree. For Kruskal's algorithm, we want to use a data structure for disjoint sets. (This leaped out at us!) If you can't find a standard data structure that supports all operations, think about how you could combine data structures or modify an existing data structure to handle a new operation.

8. If there are several data structures for the same operations, think about which ones best balance the times for the different operations. For example, here we are doing a FindLeader m times, and Merge n times. So we do FindLeader more often, up to a factor of n. Therefore, it is more important to do FindLeader quickly than Merge, as long as we don't make the Merge operation too expensive.

9. Once we've found a data structure that supports our operations, we can use the time analysis for the data structure to give the time analysis for the algorithm. Compute costs for each operation in the data structure, and multiply these costs with the number of times each operation is performed. Be sure to include the pre-processing stages in the time analysis! Identify which parts are taking the most time, and delete the other parts in the O notation.

10. Correctness follows from the correctness of the high-level strategy, as long as our re-ordering performs the same computation. You need to show that any restructuring doesn't change the output, just the time.