Algorithm Write-up Guidelines

Throughout the class, you will be asked to write-up an algorithm you have designed to solve a problem. Whether on a homework assignment, project, or an exam, your solution is expected to contain the following sections. Each section should be long enough to contain a full, readable explanation of your solution, and no longer. Remember that the burden of proof is on you. If what you are explaining isn't clear to you, you will have no hope of convincing anyone else. On the other hand, when presenting algorithms, more detail is not necessarily better. The golden standard is concise but unambiguous.

You may assume that the reader is familiar with the problem. You may make use of any standard data structures (linked lists, binary trees, heaps, etc), as well as basic sorting and searching algorithms (linear search, binary search, insertion/selection/quick sort, etc) without explaining how to implement them.

- **Description**. Present a consice and unambiguous description of the algorithm in plain English. You want to explain your general strategy here so that if you make a mistake in writing it up more precisely in pseudo code (see below), the grader can ascertain your real intent and give partial credit. This is NOT the place to restate your pseudo code in English (actually there is never a place for that). In general, you should not explain *what* you are doing, but *why* you are doing it. Obvious technical details should be kept to a minimum so that the key computational issues stand out and implementation details (if any) should be left to the pseudocode section. It might be useful to include an example to illustrate your approach.

- **Pseudocode**. A pseudocode description of your algorithm, which does not rely on a specific programming language's syntax, but instead uses English phrases where appropriate. A guiding principal here is to remember that your description will be read by a human, not a compiler. You should leave out unnecessary low-level implementation details. For example, "insert x at the end of the list" is much clearer than "list.insertAtEnd(x)." On the other hand, the pseudocode should give sufficient detail to make the analysis straightforward (e.g. don't bury loops in English phrases) while remaining

at a high enough level so that it can be easily read and understood. Be sure that the interpretation of your pseudocode is unambiguous and that you include explicitly the input and output values.

Here are a few tips:

- Instead of giving a type declaration (e.g., "double x"), explain a variable's purpose (e.g., "x holds the price of the current commodity"). Type is an implementation detail that's not necessary here.
- Replace formal control structures (e.g., "`for (int i = 1; i <= 3*n; i += 3)`" with more intuitive explanations (e.g., "Let i run from 1 up to $3n$ in steps of 3")
- Directly employ standard algorithms (e.g., "Apply quick sort to sort the node labels in increasing order"), instead of writing out the quick-sort loop in pseudo code.

While you should avoid unnecessary explanations whenever possible, be sure to include enough information so that your intent is clear and unambiguous. For example, consider the instruction "Repeatedly remove the highest weight edge from G until the graph is no longer connected." While this is mathematically well defined, the questions of how to (a) find the highest weight edge and (b) determine whether the resulting graph is connected are both nontrivial. These steps would need to be explained in greater detail.

Trust that your readers are competent programmers who can implement the low-level structures if they understand what needs to happen. Thus the golden rule for the level of details is - can you implement the algorithm just from this pesudo-code specification?

- **Time Analysis**. A worst-case analysis of your algorithm's running time. This is where you should explicitly state the data structures that are being assumed and how their preprocessing and/or query times affect your analysis.
- **Proof of Correctness**. A proof that your algorithm does what you say it does and is optimal (if appropriate). If this is a proof by induction, be sure to state clearly what the base case, inductive hypothesis, and induction steps are. If this is a proof by contradiction, state clearly what the assumption to be contradicted is. In general, be sure to define any terms used and construct a logical argument using complete mathematical sentences. Try to avoid rambling about obvious or trivial elements and focus on the key elements. A good proof provides a high-level overview of what the algorithm does, and then focuses on any tricky elements that may not be obvious.