1 Proposal Revisions

1.1 Overview

Problem: Using Coq, verify Timsort, Python's preferred sorting algorithm! Yet, implementations of Timsort – a hybrid of insertion sort and mergesort, plus some optimizations using heuristics about memory management – have been known to fail in subtle ways. We turn to Coq for its support of *verifiable* code and draw on its formal proof techniques, hoping to implement and verify (a version of) Timsort. Additionally, Coq supports the exporting of programs into other languages such as ocaml, so we expect to render a sorting procedure that is not only verified, but also usable in a broader context.

Solution sketch: We will take an incremental approach (and all of our algorithms will be functional, using persistent data structures). We're going to start with a library of fundamental data structures and verified methods. Then, we'll work toward verifying a simplified version of Timsort, hybridizing mergesort and insertion sort with a small subset of the heuristics used by Timsort in full. At first, all of these components of this simplified Timsort will be independently verified; the combined algorithm implementation will be thoroughly unit-tested. For short, call this algorithm Simsort. Implementing Simsort will be the conclusion of our core functionality.

Goals: Primarily, we'd like to verify Timsort (i.e., Simsort) as a way to learn more about Coq and certified programming. **First, for our core functionality, we'll work through the first five chapters of *Software Foundations* [2], producing a cohesive set of solutions while also building up the foundational data structures necessary for implementation and verification of our chosen sorting algorithms.**

1.2 Prioritized Feature List

Note: All algorithms and data structures used in this project will be functional; in particular, we'll use persistent data structures.

Core Features. The following feature list is a minimal list of the chapters of *SF* for which we will solve the exercises and refactor the code from the solutions into libraries to be used for our later verification of sorting algorithms.

1. **Basics.** Enumerated types: Booleans, Function types, Peano naturals. Proofs by simplification, rewriting, and case analysis. Notations.
2. **Induction.** Proofs by induction. Using lemmas.
3. **Lists.** Tuples of naturals. Lists of naturals. Variant and record types. Proofs by induction on lists.
4. **Polymorphism.** Polymorphic lists, functions, tuples, and variants. Essential function methods (`map`, `fold`, etc.). Proofs using the `unfold` tactic.
5. **More Coq.** More on induction hypotheses. Proof tactics `apply`, `apply with`, `inversion`, `destruct`.

Cool Extensions

1. **Verified insertion sort.** Verified insertion sort of lists of natural numbers.
2. **Verified merge sort.** Verified merge sort of lists of natural numbers.
3. **Simsort.** Fully tested implementation of Simsort (our hybridization of verified merge sort, verified insertion sort, and a modified subset of the heuristics used in Timsort). We still need to determine exactly which subset of heuristics we will use and how we will modify them.

4. **Verified Simsort.** This is our main goal beyond core functionality. We will improve the fully tested Simsort to a rigorously verified Simsort (using Coq).

Cooler Extensions

1. **Heaps.** Polymorphic priority queues. If it provides an advantage in asymptotics, we will use heaps to re-implement stacks.
2. **Heap Sort.** Verified heap sort of heaps of natural numbers. This will operate on lists of natural numbers, represented perhaps as trees or priority queues.
3. **Augmenting Simsort with heapsort.** Verified implementation of Simsort, with heapsort replacing insertion sort, for a slight improvement in asymptotics.
4. **Passing foreign tests.** We have come across a few known to be broken implementations of Timsort in certain languages (e.g., Java's clone of Timsort, early versions of Python 2.x's Timsort). For this cool extension, we would take some of the failing test cases for those other implementations, adapt them to use the same heuristic assumptions that we've used with Simsort, and show that our verified Simsort passes those tests.
5. **Adding more heuristics.** If we make it this far, we will add more heuristics to Simsort, showing that each addition passes verification and doesn't break invariants, working our way gradually to a verified, functional Timsort in full.

2 Progress Report

2.1 Progress

So far, we have worked out most of the exercises from the first five chapters. Our implementations work well, though they have yet to be refactored into modules for use by our proofs of sorting algorithms.

2.2 Problems

We had concerns at the beginning of the week regarding our original plan to launch right into verifying sorting algorithms without first builing a cohesive code library by completing the necessary early exercises from *SF*. We have since spoken to Prof. Morissett and to Ore about including this work as the core functionality part of our project, rather than simply as background, and we feel that this is now a more tenable project.

2.3 Teamwork

So far, teamwork has been pretty good. We've had a little trouble reconciling schedules and disparate project goals over the last week, but it's fair to say that, with the project revision as outlined above, we feel more on the same page. Early on, Michael and Ben were more available for the early project planning. Availabilities have fluctuated over the past week, so Alex and Michael now have some catching up to do with learning the Coq basics, but we feel confident that we will be on the same page by the end of the weekend, ready to tackle the sorting algorithms cohesively. Morale seems good.

2.4 Plan

We plan on finishing the remaining exercises and refactoring this weekend, possibly Monday. Then, we hope to have the two basic sorting algorithms implemented and verified by midweek. Then, we will make as much progress as possible toward verifying Simsort as time allows on Thursday and Friday.

We will continue our thus-far effective strategy of pair programming, possibly working all-together to design sorting algorithm proofs.

References

[1] Chlipala, Adam. *Certified Programming with Dependent Types*.

[2] Pierce, Benjamin, et al. *Software Foundations*.