1 Overview

Problem: Using Coq, verify Timsort, python's preferred sorting algorithm! **[Improve motivation? Possibly include discussion of failures of implementations of Timsort in languages other than Python 3.x.]** Timsort is a hybrid of insertion sort and mergesort, plus some heuristics about memory management and other optimizations. Our simplified version will include a reduced version of these heuristics.

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 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.

Next, we will implement heaps and a verified heapsort; by replacing insertion sort with heapsort in Simsort, we should get a constant-factor time improvement. Then, our primary goal beyond core functionality will be verification of Simsort. From there, we will implement extra extensions, as discussed below, possibly adding more heuristics to Simsort, approaching verification of Timsort in full.

Goals: Primarily, we'd like to verify Timsort (i.e., Simsort) as a way to learn more about Coq and certified programming.

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

- **Fundamentals.** Booleans, natural numbers (defined inductively), polymorphic lists, stacks (for very basic representations of memory needed within Timsort heuristics).
- **Verified insertion sort.** Verified insertion sort of lists of natural numbers.
- **Verified merge sort.** Verified merge sort of lists of natural numbers.
- **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.

Cool Extensions

- **Heaps.** Polymorphic priority queues. If it provides an advantage in asymptotics, we will use heaps to re-implement stacks.
- **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.
- **Augmenting Simsort with heapsort.** Fully tested implementation of Simsort, with heapsort replacing insertion sort, for a slight improvement in asymptotics.
- **Verified Simsort.** This is our main goal beyond core functionality. We will improve the fully tested Simsort to a rigorously verified Simsort (using Coq).

- **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.

- **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.

3 Technical Specification

3.1 Interfaces.

References to code written thus far are included throughout this section. `chicken` is the root directory of our github repository, located at `https://github.com/mfount/chicken`.

Also, all libraries originating from SF [2] (which we've populated with exercise solutions), will eventually be refactored into neater, application-specific libraries once we have a clearer picture of how we will refactor code to optimize proofs.

Data Structures, related methods, and proofs of them.

- **Booleans.** Our code for bools so far is contained in `chicken/coq/Basics.v`, a library provided with SF [2], which we have filled out with solutions to the relevant exercises. This includes axioms and definitions pertaining to bools (e.g. `negb false = true`), proofs of simple lemmas (primarily done using the `destruct` tactic to prove by case exhaustion) familiar binary operations, and proofs of them.

- **Natural numbers.** There is code for nats in `chicken/coq/Basics.v` and `chicken/coq/Induction.v`, another SF library that we have supplemented with solutions to relevant exercises. Nats are defined inductively using a unary representation (as Peano naturals). Most of the lemmas pertaining to nats are proven using the `induction` tactic.

  For our purposes, the set of natural numbers is most usefully structured as an ordered set, so we've verified (or made progress toawrd verifying) the order properties of the natural numbers as comparison methods, for example `ble_nat` (<= for nats) as well as `blt_nat` (< for nats).

- **Polymorphic lists.** We will be sorting objects of the type `list nat` (informally, "natlists") throughout this project. Also, we will use lists of lists, `list (list nat)` in Simsort. Definitions of polymorphic lists are found in `chicken/coq/Poly.v` and `chicken/coq/Lists.v` other SF libraries modified as described above. (The former defines polymorphic lists, and the latter defines, specifically, lists of nats.)

- **Stack.** We're going to implement this using polymorphic lists. Methods will include `push` (implemented as the familiar list `Cons` construction) and `pop` (implemented as `hd_opt`). Proofs of consistency of these will be based on existing exercises in SF, which we have already done in working through the textbook. (see `chicken/coq/Poly.v`).

- **Heap. [Pseudocode here!]**

Algorithms and proofs of them.

- **Fundamental methods and proofs relevant to all sorting algorithms.**

- **Mergesort.**

- **Insertion sort.**
- **Heap sort.**
- **Simsort.**

4 Timeline

As of Friday 17 April at 5pm, we have exactly two weeks to finish our project.

Week One.

- Isolate axiom definitions from the first 4 chapters of Software Foundations that we will use. Get them working as a set of libraries that can be `Required`.
- As a group, do all of these exercises (proof of the consistency of the axiom definitions that we need). As individuals, learn the contents of the first 5 chapters.
- Complete proofs of `is_sorted_le` in `chicken/coq/IsSorted.v`. Complete proofs of `insertion_sort` in `chicken/coq/InsertSort.v`. (Both of these libraries are ours, not originating from SF, and already exist in part.)
- Do the same for `merge_sort`.
- Get an implementation of heaps working (in line with the pseudo-code signature given in the final spec) and prove half of the operations.

Week Two.

- Complete proof of the correctness of operations of heap.
- Implement `heap_sort` and prove it. Easy after the work we have done for heaps.
- combine merge and insert into `simsort`. Export `simsort` to OCaml and write test suite.
- Revise `simsort` tom implement heaps; export to OCaml; ensure that the test suite is still passed.
- Verify `simsort`.
- Further tasks if possible. (See "Cool Extensions", as outlined above.)

5 Detailed Progress

All code written so far is contained in the directory `chicken/coq`, where, again `chicken` is the root directory of our github repository, located at `https://github.com/mfount/chicken`.

References

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

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