1 Names and Emails

- Aleksandar Makelov — `amakelov@college.harvard.edu`
- Ben Wetherfield — `bwetherfield@college.harvard.edu`
- Chan Kang — `chankang@college.harvard.edu`
- Michael Fountaine — `mfount@college.harvard.edu`

2 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.

3 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.** 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).

Cool Extensions

- **Heaps.** Priority queues of natural numbers.
- **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.
- **Timsort.** Verified timsort; that is, simplified timsort plus heuristics. This version would have the same time asymptotics as python's implementation of Timsort: $\Theta(n)$ best case, $\Theta(n \log n)$ average case, $\Theta(n \log n)$ worst case, where n is the length of the list.
- **[Everything until here in this section is correct; will finish after class. —mfount]**
- **Python's timsort.** This final possible extension could be to modify our verified Timsort algorithm to use exactly the heuristics found in the current Python 3.x release, as opposed to the heuristics we end up using in our implemented version of the full Timsort algorithm. We could also attempt to use the same space complexity of Timsort, $O(n)$.

4 Technical Specification

Data Structures We will, of course, prove all methods associated with these types/structures.

- Ordered set.
- Natural numbers.
- List of natural numbers.
- Array of natural numbers. (Unclear if necessary.)
- Stack.
- Tree.
- Heap (a.k.a. Priority Queue).

Algorithms These are the algorithms and subroutines we'll need for the mutable (and often in-place) versions of the sorts, which we expect to be harder; if we decide to do the pure versions, we will make adjustments accordingly. The ideas are based on the paper [3], which describes how to prove correctness for insertion sort, quicksort and mergesort. It also provides some general predicates for formally verifying sorting algorithms, like `(sorted t)` which tests for sortedness, and `(permutation t t')` which tests whether t' is a permutation of t.

- **Insertion sort:** The subroutine `(insertion n t)` takes as input an array t where the first $n-1$ elements are sorted, and returns an array where the first n elements are sorted, by inserting the n-th element in its place. Then insertion sort itself proceeds by running `(insertion n t)` for $n = 1, 2, 3, \ldots, N$ for a list t of length N.
- **Mergesort:** Here the basic subroutine is `(merge t)` which takes a list t in which the first and second halves are sorted, and merges them. Here we'll need to use some extra memory. Then mergesort proceeds by calling itself recursively on each half, and then merging them together.
- **Tree sort:** We first build a binary search tree from the array by a subroutine `(bst_build t)`; then we have another subroutine `(bst_traverse t)` that traverses it in-order.

- **Heap sort:** The algorithm proceeds by making the array into a heap in-place, and then gradually pushing the largest elements to the right side of the heap. Here, we're representing a binary tree implicitly as an array, where the children of the i-th element are the $2i+1$-th and $2i+2$-th elements. We have the following subroutines:
  - Predicate `(heap t n k)` checks if in a list t, the tree of elements of index $\leq n$ rooted at the k-th element.
  - Predicate `(inftree t n v k)` checks if in a list t, every element in the tree of elements of index $\leq n$ rooted at the k-th element is less than v.
  - Subroutine `(downheap t k n)` takes a list t where the tree of elements of index $\leq n$ rooted at the k-th element is a heap *except* possibly for the root node. It then makes a bunch of swaps that make this tree into a heap.

  Having these subroutines, heapsort is easy: first build the heap using `downheap` a bunch of times, and then swap the 1-st element with the i-th and rebuild the heap up to index $i-1$, for $i = N, N-1, \ldots, 1$

- **Simple timsort:** The idea of timsort is to use the sorted chunks (called *runs*) often present in real-world data. At a high level, the algorithm works by first making sure all runs are of some minimum length (using insertion sort if necessary), and then passing over the list and merging the runs in an intelligent way.

  As we do the second pass, we push the starting index and length of each run on a stack, and merge consecutive runs in order to preserve a certain invariant of the stack: if A, B, C are the lengths of three consecutive runs on the stack (with A being the topmost), we require that $B > C$ and $A > B + C$.

  So at any point we end up with a bunch of runs whose sizes grow faster than the Fibonacci sequence, i.e. at least exponentially fast; so it's easy to see there are at most logarithmically many runs at each point in time, which is important for memory reasons and for running time (it's faster to merge a sequence of lists with exponentially-increasing size than to naively merge a list split into equal parts).

  The way the invariant is preserved is where things get weird: apparently, there is a bug in the algorithm on the wikipedia page (which attempts to restore the invariant by only fixing the top three runs), but there is a fix [4] we have to look into.

  It's hard to know in advance what interface would be best for our proof, but here's a guess:
  - Predicate `(run t i j)` - this checks if the subarray between the i-th and j-th index of t is a run, i.e. is already in sorted order.
  - Subroutine `insertion-sort` - this is just as insertion sort above.
  - Subroutine `merge` - this is supposed to take the next run and do any merges necessary to preserve the invariant; of course, it will be based on the merge procedure.
  - Now piecing together all the parts should be easy. In the end, we end up with a sequence of runs that we merge in the obvious way.

- **Timsort:** Additionally, there are some memory optimizations ('galloping') during the merges that we might try to include if we end up having the time for it.

Proofs *Note:* We will be able to better characterize proofs after working through *Software Foundations* [2] (See "Next Steps" below.)

At this stage, we know:

- Because Coq proofs are very formal, they suffer from an inability to convey human-to-human understanding that is afforded by semi-formal, traditional mathematical proof. To aid debugging and authoring of proofs in Coq, *SF* [2] recommends two strategies, "Informal Proof" and "Human Proof", that by encouraging verbosity and a high degree of refactoring help to restore the "understanding" aspect of a proof.

- Proofs in coq are guided by tactics. These are like hints you give to the prover, such as "rewrite this as the other side of this equality we already proved", "rewrite both sides in a canonical way", etc.
- We expect to use the "Reflexivity" keyword frequently for verifying more basic properties of our data structures. Proofs in Coq that use reflexivity as a tactic are generally very simple proofs using, say, rewrite rules.
- When we get onto proofs of correctness of algorithms, we will have to break proofs into subcomponents.
- There are various ways of doing this:
    1. Example, Theorem, Lemma, Fact and Remark keywords (all functionally the same) break up larger proof into subproofs.
    2. The Case proof tactic (chapter 3 of Software Foundations) can be used to break a proof clearly into cases (as in the familiar structure of an induction proof).
    3. This could break a proof into say a case of the Reflexivity tactic and the Induction tactic. Our reading of the references and some example Coq code indicates that this is a common formulation that appears when doing structural induction.
    4. Ltac can be used to create common tactics.

5 Next Steps

[Revise these!]

- Install Coq 8.4-pl5 and ensure that it's running correctly. Also install an IDE, most likely emacs with Proof General.
- Read early parts of *SF* [2] and *CPDT* [1], doing excercises to familiarize ourselves with Gallina (Coq's functional language, similar to Caml) syntax and Coq's Proof functionality.

  Specifically, we will initially work through at least the 1-star and 2-star exercises in the opening three chapters of *SF*. (This will give us the implementations of lists and nats.)
- In a similar vein, get an idea of how long it would take to verify correctness of the basic sorting algorithms
- Understand timsort in more depth.

References

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

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

[3] Filliatre, Jean-Christophe and Magaud, Nicolas. *Certification of sorting algorithms in the system COQ*

[4] de Gouw, Stijn and Rot, Jurriaan and de Boer, Frank S and Bubel, Richard and Hahnle, Reiner. *OpenJDK's java. utils. Collection. sort () is broken: The good, the bad and the worst case*