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!

Solution sketch: We will take an incremental approach: We're going to start with an abstract "implementation" of Timsort, possibly black-boxing certian difficult-to-implement features such as heuristics used in the algorithm, and implement a verified version of that implementation to sort lists of natural numbers (defined inductively). Timsort is a hybrid sorting algorithm, requiring other sorting sub-algorithms and some simple data structures, so we will divide our project into interfaces for each of these, along with interfaces for proof tactics. We're going to hone our sorting algorithm verification chops by working through insertion sort and mergesort, both of which are part of Timsort, followed by tree sort and heap sort to test our data structures, followed by Timsort itself!

Goals: Primarily, we'd like to verify Timsort as a way to learn more about Coq and certified programming. (If Timsort proves to be too complicated (by our estimation by the time the final spec is due), we will be able to accomplish this same learning outcome by working through insertion, merge, tree, and heap sorts. Or, perhaps, we will include another hybrid search algorithm instead of Timsort.)

3 Prioritized Feature List

Note: Generally, we need to read more about Coq to understand where to draw the line between "core" and "cool"; that is, how long these features will take to implement, including proofs.

Additionally, we are unsure of whether to verify the in-place, imperative versions of these algorithms or the pure versions. The deciding factor will most likely be the difficulty of implementing arrays (and references) in Coq.

Core Features

- **Fundamentals.** Ordered sets, natural numbers (defined inductively), lists and arrays, stacks (to be used as Timsort's memory and temporary memory).
- **Insertion sort.** Verified insertion sort of lists of natural numbers.
- **Merge sort.** Verified merge sort of lists of natural numbers.
- **Trees.** Binary search trees of natural numbers.
- **Tree Sort.** Verified tree sort of binary search trees of natural numbers.
- **Heaps.** Priority queues of natural numbers.
- **Heap Sort.** Verified heap sort of binary search trees of natural numbers.
- **Simplified Timsort.** This will operate on lists of natural numbers, represented perhaps as trees or priority queues.

  Timsort is a hybrid of insertion sort and mergesort, plus some heuristics about memory management and other optimizations. Our "simplified timsort" will omit these heuristics when possible.

Cool Extensions

- **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.
- **Polymorphic timsort.** A verified timsort that works on polymorphic lists. (Unlike in OCaml, this seems actually to be a good bit of work in Coq, having to prove things about each subtype of our ordered set type.)
- **Export to OCaml.** One of Coq's initial purposes is to export verified OCaml (or Haskell or Scheme) code; for this cool extension, we can try to turn some or all of our verified algorithm implementations into usable OCaml.
- **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 All methods described below will have corresponding proofs.

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

Algorithms These are the algorithms and subroutines we'll need for the mutable versions of the algorithms, which we expect to be harder; if we decide to do the pure versions, we will make adjustments accordingly.

- **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:**
- **Tree sort:**
- **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 rooted at the k-th element of elements of index at most n is a heap

  - Predicate `(inftree t n v k)` checks for a tree represented as above whether all elements are $\leq v$.
  - Subroutine `(downheap t k n)` takes an array t where the tree of elements rooted at the k-th element and of indices $\leq n$ is a heap, *except* possibly for the root node. It then makes a bunch of swaps that make this tree into a heap (invariants need figuring out of course).

- Then, heapsort is easy: first build the heap using `downheap` a bunch of times, and then swap the 1st element with the N-th, $N-1$-th, etc. and rebuild the heap to the appropriate index between swaps.
- **Simple timsort:** Timsort, as described below, omitting heuristics.
- **Timsort:** First we pass over the list and make sure each run is of at least some minimum length c.

  Then we pass over it again and push the base address (that is, the index of the first element) and length of every run (this could be done in the above pass, but let's separate them for clarity). But as we push runs on the stack, we also sometimes merge consecutive runs until some invariant (that 'attempts to keep the run lengths as close to each other as possible to balance the merges' as the wikipedia page says) is satisfied. The condition is that if X, Y, Z are the lengths of the top three runs on the stack, we must have

  $$X > Y + Z \text{ and } Y > Z$$

  (There are also other invariants to maintain.) 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 seems to be important for memory reasons. It's also important for running-time reasons it seems - it's much faster to merge these exponentially-increasing guys than to naively merge a list split into equal parts! Additionally, there are some memory optimizations ('galloping').

Proofs *Note:* We will be able to better characterize proofs after working through Software Foundations (Pierce *et al.*) - Next Steps below.
At this stage, we know:

- Proofs in coq are guided by tactics. These are like hints you give to the prover, such as âĂŸrewrite this as the other side of this equality we already provedâĂŹ, âĂŸrewrite both sides in a canonical wayâĂŹ, etc.
- We expect to use the "Reflexivity" keyword a lot for more basic properties of our data structures.
- 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 steps (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 - this is a common formulation.
  4. Ltac can be used to make common tactics.

5 Next Steps

- 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.)

References

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

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