SOFTWARE FOUNDATIONS

VOLUME 3: VERIFIED FUNCTIONAL ALGORITHMS

TABLE OF CONTENTS INDEX ROADMAP

SORT

INSERTION SORT

Sorting can be done in O(N log N) time by various algorithms (quicksort, mergesort, heapsort, etc.). But for smallish inputs, a simple quadratic-time algorithm such as insertion sort can actually be faster. And it's certainly easier to implement — and to prove correct.

Recommended Reading

If you don't already know how insertion sort works, see Wikipedia or read any standard textbook; for example:

Sections 2.0 and 2.1 of *Algorithms, Fourth Edition*, by Sedgewick and Wayne, Addison Wesley 2011; or

Section 2.1 of *Introduction to Algorithms, 3rd Edition*, by Cormen, Leiserson, and Rivest, MIT Press 2009.

The Insertion-Sort Program

Insertion sort is usually presented as an imperative program operating on arrays. But it works just as well as a functional program operating on linked lists!

```
Require Import Perm.

Fixpoint insert (i:nat) (l: list nat) :=
  match l with
  | nil ⇒ i::nil
  | h::t ⇒ if i <=? h then i::h::t else h :: insert i t
 end.

Fixpoint sort (l: list nat) : list nat :=
  match l with
  | nil ⇒ nil
```

```
  | h::t ⇒ insert h (sort t)
end.

Example sort_pi: sort [3;1;4;1;5;9;2;6;5;3;5]
                    = [1;1;2;3;3;4;5;5;5;6;9].
+
```

What Sedgewick/Wayne and Cormen/Leiserson/Rivest don't acknowlege is that the arrays-and-swaps model of sorting is not the only one in the world. We are writing *functional programs*, where our sequences are (typically) represented as linked lists, and where we do *not* destructively splice elements into those lists. Instead, we build new lists that (sometimes) share structure with the old ones.

So, for example:

```
Eval compute in insert 7 [1; 3; 4; 8; 12; 14; 18].
(* = 1; 3; 4; 7; 8; 12; 14; 18 *)
```

The tail of this list, `12::14::18::nil`, is not disturbed or rebuilt by the `insert` algorithm. The nodes `1::3::4::7::_` are new, constructed by `insert`. The first three nodes of the old list, `1::3::4::_` will likely be garbage-collected, if no other data structure is still pointing at them. Thus, in this typical case,

- Time cost = 4X
- Space cost = (4-3)Y = Y

where X and Y are constants, independent of the length of the tail. The value Y is the number of bytes in one list node: 2 to 4 words, depending on how the implementation handles constructor-tags. We write (4-3) to indicate that four list nodes are constructed, while three list nodes become eligible for garbage collection.

We will not *prove* such things about the time and space cost, but they are *true* anyway, and we should keep them in consideration.

Specification of Correctness

A sorting algorithm must rearrange the elements into a list that is totally ordered.

```
Inductive sorted: list nat → Prop :=
| sorted_nil:
    sorted nil
| sorted_1: ∀ x,
    sorted (x::nil)
| sorted_cons: ∀ x y l,
   x ≤ y → sorted (y::l) → sorted (x::y::l).
```

Is this really the right definition of what it means for a list to be sorted? One might have thought that it should go more like this:

```
Definition sorted' (al: list nat) :=
 ∀ i j, i < j < length al → nth i al 0 ≤ nth j al 0.
```

This is a reasonable definition too. It should be equivalent. Later on, we'll prove that the two definitions really are equivalent. For now, let's use the first one to define what it means to be a correct sorting algorithm.

```
Definition is_a_sorting_algorithm (f: list nat → list nat) :=
  ∀ al, Permutation al (f al) ∧ sorted (f al).
```

The result `(f al)` should not only be a `sorted` sequence, but it should be some rearrangement (Permutation) of the input sequence.

Proof of Correctness

Exercise: 3 stars (insert_perm)

Prove the following auxiliary lemma, `insert_perm`, which will be useful for proving `sort_perm` below. Your proof will be by induction, but you'll need some of the permutation facts from the library, so first remind yourself by doing `Search`.

```
Search Permutation.

Lemma insert_perm: ∀ x l, Permutation (x::l) (insert x l).
Proof.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (sort_perm)

Now prove that sort is a permutation.

```
Theorem sort_perm: ∀ l, Permutation l (sort l).
Proof.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 4 stars (insert_sorted)

This one is a bit tricky. However, there just a single induction right at the beginning, and you do *not* need to use `insert_perm` or `sort_perm`.

```
Lemma insert_sorted:
  ∀ a l, sorted l → sorted (insert a l).
Proof.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 2 stars (sort_sorted)

This one is easy.

```
Theorem sort_sorted: ∀ l, sorted (sort l).
Proof.
(* FILL IN HERE *) Admitted.
```

□

Now we wrap it all up.

```
Theorem insertion_sort_correct:
    is_a_sorting_algorithm sort.
Proof.
  split. apply sort_perm. apply sort_sorted.
Qed.
```

Making Sure the Specification is Right

It's really important to get the *specification* right. You can prove that your program satisfies its specification (and Coq will check that proof for you), but you can't prove that you have the right specification. Therefore, we take the trouble to write two different specifications of sortedness (`sorted` and `sorted'`), and prove that they mean the same thing. This increases our confidence that we have the right specification, though of course it doesn't *prove* that we do.

Exercise: 4 stars, optional (sorted_sorted')

```
Lemma sorted_sorted': ∀ al, sorted al → sorted' al.
```

Hint: Instead of doing induction on the list `al`, do induction on the *sortedness* of `al`. This proof is a bit tricky, so you may have to think about how to approach it, and try out one or two different ideas.

```
(* FILL IN HERE *) Admitted.
```
□

Exercise: 3 stars, optional (sorted'_sorted)

```
Lemma sorted'_sorted: ∀ al, sorted' al → sorted al.
```

Here, you can't do induction on the sorted'-ness of the list, because `sorted'` is not an inductive predicate.

```
Proof.
(* FILL IN HERE *) Admitted.
```
□

Proving Correctness from the Alternate Spec

Depending on how you write the specification of a program, it can be *much* harder or easier to prove correctness. We saw that the predicates `sorted` and `sorted'` are equivalent; but it is really difficult to prove correctness of insertion sort directly from `sorted'`.

Try it yourself, if you dare! I managed it, but my proof is quite long and complicated. I found that I needed all these facts:

- `insert_perm`, `sort_perm`
- `Forall_perm`, `Permutation_length`
- `Permutation_sym`, `Permutation_trans`
- a new lemma `Forall_nth`, stated below.

Maybe you will find a better way that's not so complicated.

DO NOT USE `sorted_sorted'`, `sorted'_sorted`, `insert_sorted`, or `sort_sorted` in these proofs!

Exercise: 3 stars, optional (Forall_nth)

```
Lemma Forall_nth:
  ∀ {A: Type} (P: A → Prop) d (al: list A),
     Forall P al ↔ (∀ i, i < length al → P (nth i al d)).
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 4 stars, optional (insert_sorted')

```
Lemma insert_sorted':
  ∀ a l, sorted' l → sorted' (insert a l).
(* FILL IN HERE *) Admitted.
```

□

Exercise: 4 stars, optional (insert_sorted')

```
Theorem sort_sorted': ∀ l, sorted' (sort l).
(* FILL IN HERE *) Admitted.
```

□

The Moral of This Story

The proofs of `insert_sorted` and `sort_sorted` were easy; the proofs of `insert_sorted'` and `sort_sorted'` were difficult; and yet `sorted al` ↔ `sorted' al`. *Different formulations of the functional specification can lead to great differences in the difficulty of the correctness proofs*.

Suppose someone required you to prove `sort_sorted'`, and never mentioned the `sorted` predicate to you. Instead of proving `sort_sorted'` directly, it would be much easier to design a new predicate (`sorted`), and then prove `sort_sorted` and `sorted_sorted'`.