

CSU290 Homework 5 Due March 24, 2008 at 10 AM

- You **have to** work in pairs for this homework and you have to work with someone you have not worked with yet. Send the name of your partner to Zhifeng (austin@ccs.neu.edu) by the end of the day, on Thursday March 20th. If you do not do this, you will get a 0 on your homework. If you need a partner, send email to Zhifeng by the end of the day on Wednesday, March 19th.
- Each pair should submit their homework via blackboard. Each pair should submit just once. That is, one person per team should submit the homework via their blackboard account. Your submission should be a text file. The format of the file should be:
 - the names of the team
 - clear solutions to the problems

In this homework, you will proceed to state and prove the correctness of the both insertion-sort and merge-sort, two common sorting algorithms.

You can assume that $<$ is a linear ordering on the elements of the ACL2 universe (not just on the rationals). In other words, for any values a, b in the ACL2 universe, exactly one of $(< a b)$, $(> a b)$, or $(= a b)$ holds, and that $<$ is transitive. If $(< a b)$ and $(< b c)$, then $(< a c)$. Again, this is just an extension of what $<$ provides for the rationals. (For those interested, there is such a function in ACL2. It is called `lexorder`; see the ACL2 documentation.)

You may use the following definitions for `app` and `elem-of`:

```
(defun app (x y)
  (if (endp x)
      y
      (cons (car x) (app (cdr x) y))))
```

```
(defun elem-of (x l)
  (if (endp l)
      nil
      (if (equal x (car l))
          t
          (elem-of x (cdr l)))))
```

All these problems are straightforward, except the ones marked with *, which are harder, but by no means impossible. Proofs often require theorems in previous problems. (For instance, proofs of the theorems about insertion-sort requires results established in the problem about properties of permutations.) Feel free to use theorems in previous problems in a proof you are working on, even if you did not actually figure out the proof of those theorems.

Problem 1: Reasoning by Cases

Before we start, let us learn a new way to prove theorem, that comes in handy when we need to simplify `if` expressions and we have no information about the test in the conditional expression. Consider the following definitions, taken from the wiki page of the course:

```
(defun andp (x y)
  (if x
      (if y t nil)
      nil))
```

```
(defun orp (x y)
  (if x
      t
      (if y t nil)))
```

Suppose we wanted to prove the following theorem: `(booleanp (if (> x 10) t nil))`. Clearly, a direct approach will not work, because we have no information about `(> x 10)`.

However, an informal argument would run as follow: consider `(> x 10)`; it is either true or false. If true, then `(if (> x 10) t nil)` is equal to `t`, which is a Boolean. If false, then `(if (> x 10) t nil)` is equal to `nil`, which is also a Boolean. Thus, in all cases, the result is a Boolean, so `(booleanp (if (> x 10) t nil))` is true.

Case analysis simply formalizes the above informal argument. To reason by cases on a formula `(booleanp (if (> x 10) t nil))`, we choose a Boolean expression to do case analysis on, here `(> x 10)`, and we prove the following two formulas:

- (a) `(> x 10) ⇒ (booleanp (if (> x 10) t nil))`
- (b) `(not (> x 10)) ⇒ (booleanp (if (> x 10) t nil))`

If we prove those two facts, then we have proved `(booleanp (if (> x 10) t nil))`.

(Why? To justify reasoning by cases, note that a formula ϕ is always equivalent to $(p \vee \neg p) \Rightarrow \phi$, for any choice of p , which you can easily establish using Boolean logic. Similarly, $(p \vee \neg p) \Rightarrow \phi$ is equivalent to $(p \Rightarrow \phi) \wedge (\neg p \Rightarrow \phi)$. Thus, if you let p be the formula on which you want to do case analysis, you see that to prove ϕ , it suffices to prove $p \Rightarrow \phi$ and $\neg p \Rightarrow \phi$.)

Prove the following, by cases (generally, the case to consider is of the form $x = \text{nil}$, maybe with a different variable):

- (1) `(booleanp (andp x y))`
- (2) `(andp x y) = (andp y x)`
- (3) `(booleanp (orp x y))`
- (4) `(orp x y) = (orp y x)`
- (5) `(booleanp z) ⇒ (andp z t) = z`
- (6) `(booleanp z) ⇒ (andp t z) = z`
- (7) `(booleanp z) ⇒ (orp z nil) = z`
- (8) `(booleanp z) ⇒ (orp nil z) = z`

Problem 2: A Warm-Up

Prove the following property of the `elem-of` function:

- (1) `(elem-of a x) ⇒ (elem-of a (app x y))`

Hint: this is a proof by induction; the base case (the first proof obligation) is a bit tricky; the inductive case (the second proof obligation), you should prove by case analysis on `a = (car x)`.

Problem 3: Orderings

One property of sorting algorithms is that they produce sorted lists of values. To specify the correctness of sorting algorithms, therefore, we will need to state that at the very least they produce sorted lists. The following predicate checks that a list is an ordered true list of elements, in increasing order.

```
(defun ordered (l)
  (if (endp l)
      (equal l nil)
      (if (endp (cdr l))
          (equal (cdr l) nil)
          (and (<= (car l) (car (cdr l)))
               (ordered (cdr l)))))))
```

Prove the following properties of `ordered`:

- (1) $(\text{ordered } y) \wedge (\text{endp } y) \Rightarrow y = \text{nil}$
- (2) $(\text{ordered } y) \Rightarrow (\text{ordered } (\text{cdr } y))$

Problem 4: Permutations

The second property of sorting algorithms is that they output a permutation of their inputs. (That is, they do not lose elements during sorting, nor do they introduce spurious elements.) We say that x is a permutation of y if the first element of x is in y , and the rest of x is a permutation of y from which the first element of x has been removed. This is captured by the following definitions:

```
(defun remove-elt (x l)
  (if (endp l)
      nil
      (if (equal x (car l))
          (cdr l)
          (cons (car l) (remove-elt x (cdr l))))))
```

```

(defun perm (l1 l2)
  (if (and (endp l1) (endp l2))
      t
      (if (not (endp l1))
          (and (elem-of (car l1) l2)
               (perm (cdr l1) (remove-elt (car l1) l2)))
          nil)))

```

Prove the following basic properties of `remove-elt`:

- (1) $(\text{remove-elt } (\text{car } l) l) = (\text{cdr } l)$
- (2) $(\text{ordered } y) \Rightarrow (\text{ordered } (\text{remove-elt } a y))$

Prove the following basic properties of `perm`:

- (3) $(\text{perm } x y) \wedge (\text{endp } x) \Rightarrow (\text{endp } y)$
- (4) $(\text{perm } x y) \wedge (\text{consp } x) \Rightarrow (\text{consp } y)$
- (5) $(\text{perm } x y) \wedge (\text{consp } x) \Rightarrow (\text{elem-of } (\text{car } x) y)$
- (6) $(\text{perm } x y) \wedge (\text{consp } x) \Rightarrow (\text{perm } (\text{cdr } x) (\text{remove-elt } (\text{car } x) y))$
- (7) $(\text{perm } (\text{cons } a b) (\text{cons } a c)) = (\text{perm } b c)$

* Problem 5: Perm is an Equivalence Relation

Here, we show that `perm` is an equivalence relation.

Prove that `perm` is reflexive (actually pretty straightforward):

- (1) $(\text{perm } x x)$

To show that `perm` is symmetric, we need a sequences of lemmas that talk about the relationship between membership and remove elements from a list, some of which actually come in handy later on.

First, prove that remove elements can be done in any order:

$$(2) \text{ (remove-elt a (remove-elt b c)) = (remove-elt b (remove-elt a c))}$$

Next, prove that deleting an element different than `a` from a list does not affect the membership of `a` in the list:

$$(3) \text{ (not (equal a b))} \Rightarrow \text{(elem-of a (remove-elt b x)) = (elem-of a x)}$$

Next, prove that deleting the same member from two lists preserves the fact that these two lists are permutations:

$$(4) \text{ (elem-of x a) } \wedge \text{ (elem-of x b) } \wedge \text{ (perm a b)} \\ \Rightarrow \text{(perm (remove-elt x a) (remove-elt x b))}$$

Next, prove that deleting an element of a list and re-introducing it at the beginning of the list preserves the original list's status as a permutation of another list:

$$(5) \text{ (elem-of a y) } \wedge \text{ (perm (cons a (remove-elt a y)) x)} \Rightarrow \text{(perm y x)}$$

Finally, prove that `perm` is symmetric:

$$(6) \text{ (perm x y)} \Rightarrow \text{(perm y x)}$$

It remains to show transitivity. First, prove the following result that simply says that two lists that are permutations of each other have the same members:

$$(7) \text{ (perm x y)} \Rightarrow \text{(elem-of a x) = (elem-of a y)}$$

From this, prove that `perm` is transitive:

$$(8) \text{ (perm x y) } \wedge \text{ (perm y z)} \Rightarrow \text{(perm x z)}$$

Problem 6: Insertion Sort

We have enough machinery to attack our first sorting algorithm, insertion-sort. Recall that insertion-sort works by repeatedly inserting elements of the list into an ordered list, obtained by recursively sorting the rest of the list as we proceed.

Insertion-sort relies on the following auxiliary function that inserts an element into an ordered list:

```
(defun insert (n l)
  (if (endp l)
      (cons n l)
      (if (<= n (car l))
          (cons n l)
          (cons (car l) (insert n (cdr l))))))
```

Reasoning about `insert` involves reasoning about ordering. Prove the following property of `insert`:

- (1) $(\leq a (\text{car } l)) \wedge (\leq a b) \Rightarrow (\leq a (\text{car } (\text{insert } b l)))$
Hint: by case analysis on $(\leq b (\text{car } l))$.

(Similar properties can come in handy, and are proved essentially similarly.)

Prove the following (easy) property of `insert`:

- (2) $(\text{consp } (\text{insert } a l))$

Prove the following properties of the `insert` function:

- (3) $(\text{elem-of } a (\text{insert } a x))$
Hint: for the second proof obligation, use case analysis on $(\leq a (\text{car } x))$.
- (4) $(\text{ordered } l) \Rightarrow (\text{ordered } (\text{insert } a l))$
Hint: again, for the second proof obligation, use case analysis on $(\leq a (\text{car } x))$.

The next properties show that `insert` and `remove-elt` are essentially inverses of each other. Prove:

(5) `(remove-elt a (insert a l)) = l`

Hint: for the second proof obligation, use case analysis on `(<= a (car l))`.

(6) `(ordered l) ∧ (elem-of a l) ⇒ (insert a (remove-elt a l)) = l`

Hint: for the second proof obligation, use case analysis on `(= a (car l))`.

Additionally, `insert` and `perm` interact. Prove:

(7) `(perm x y) ⇒ (perm (cons a x) (insert a y))`

That's all the properties of `insert` we need. We can now define insertion-sort and prove it correct.

```
(defun isort (l)
  (if (endp l)
      nil
      (insert (car l) (isort (cdr l)))))
```

Prove the following (easy) property of `isort`:

(8) `(consp x) ⇒ (consp (isort x))`

Correctness of insertion-sort is given by two theorems, stating that it returns an ordered list of its inputs. Prove:

(9) `(ordered (isort x))`

(10) `(perm x (isort x))`

Problem 7: Permutations and List Concatenation

Before tackling merge-sort, we need more interesting properties of permutations. In particular, we need to understand how permutations interact with `app`. Prove the following properties:

(1) `(perm y1 y2) ⇒ (perm (app x y1) (app x y2))`

* (2) $(\text{perm } x1 \ x2) \Rightarrow (\text{perm } (\text{app } x1 \ y) \ (\text{app } x2 \ y))$

(3) $(\text{perm } (\text{app } x \ (\text{cons } a \ y)) \ (\text{cons } a \ (\text{app } x \ y)))$

Hint: for the second proof obligation, use case analysis on $a = (\text{car } x)$.

Problem 8: Merge Sort

Merge-sort is more difficult to reason about simply because the recursion pattern in merge-sort is not a simple true-list recursion pattern, meaning, in particular, that we cannot use our standard true-list induction principle.

First, recall that merge-sort relies on two auxiliary functions, one that merges two ordered lists, and one that splits a list into two lists of roughly the same length.

```
(defun merge (l1 l2)
  (if (endp l1)
      l2
      (if (endp l2)
          l1
          (if (<= (car l1) (car l2))
              (cons (car l1) (merge (cdr l1) l2))
              (cons (car l2) (merge l1 (cdr l2))))))))
```

Our standard true-list induction principle for `merge` will also not work. Rather than spending time on coming up with an appropriate principle for `merge`, you may simply assume the following two properties of `merge`:

(a) $(\text{ordered } l1) \wedge (\text{ordered } l2) \Rightarrow (\text{ordered } (\text{merge } l1 \ l2))$

(b) $(\text{perm } (\text{merge } x \ y) \ (\text{app } x \ y))$

(They are indeed provable, but we do not have the machinery to prove them just yet.)

Here is our function to split a list. Note that it returns a cons-pair of two lists:

```
(defun split (lst)
  (if (endp lst)
```

```

      (cons nil nil)
    (if (endp (cdr lst))
        (cons lst nil)
        (let ((p (split (cdr (cdr lst)))))
            (cons (cons (car lst) (car p))
                  (cons (car (cdr lst)) (cdr p)))))))

```

Recall that

```
(let ((p exp)) ...p... ...p...)
```

is just an abbreviation for

```
...exp... ...exp...
```

that is, substituting the expression `exp` for `p` everywhere in the body of the `let`.

Make sure that you understand how `split` works, by trying it out on a few examples. You may also want to code it up in ACL2s and try it out live.

Proving properties for `split` relies on the following variant of the induction principle, which we will call `split`-induction here:

- To prove ϕ by `split`-induction, it suffices to prove the following three proof obligations:
 - (a) $(\text{endp } x) \Rightarrow \phi$
 - (b) $(\text{consp } x) \wedge (\text{endp } (\text{cdr } x)) \Rightarrow \phi$
 - (c) $(\text{consp } x) \wedge (\text{consp } (\text{cdr } x)) \wedge \phi(x (\text{cdr } (\text{cdr } x))) \Rightarrow \phi$

(You can argue that `split`-induction is a valid rule in the logic by the same argument we use to justify true-list induction. Exercise for the interested reader.)

Prove the following property of `split`, which simply says that concatenating the two lists that `split` returns gives you a permutation of the original list, by `split`-induction:

```
(1) (perm (app (car (split x)) (cdr (split x))) x)
```

Well, the hard work is mostly done. Let's define merge-sort:

```
(defun msort (l)
  (if (endp l)
```

```

nil
(if (endp (cdr l))
    (list (car l))
    (let ((p (split l)))
        (merge (msort (car p)) (msort (cdr p)))))))

```

Just like for `split`, proving properties for `msort` relies on the following variant of the induction principle, which we will call `msort-induction` here:

- To prove ϕ by `msort-induction`, it suffices to prove the following three proof obligations:
 - (a) $(\text{endp } x) \Rightarrow \phi$
 - (b) $(\text{consp } x) \wedge (\text{endp } (\text{cdr } x)) \Rightarrow \phi$
 - (c) $(\text{consp } x) \wedge (\text{consp } (\text{cdr } x))$
 $\wedge \phi(x (\text{car } (\text{split } x))) \wedge \phi(x (\text{cdr } (\text{split } x)))$
 $\Rightarrow \phi$

(As before, you can argue that `msort-induction` is a valid rule in the logic by the same argument we use to justify true-list induction. Another exercise for the interested reader.)

Now, just like with insertion-sort, correctness of merge-sort is given by two theorems, stating that it returns an ordered list of its inputs. Prove the following two theorems by `msort-induction`:

- (2) $(\text{ordered } (\text{msort } x))$
- (3) $(\text{perm } (\text{msort } x) x)$

Problem 9: Relating Sorting Algorithms

The goal now is simply to show that insertion-sort and merge-sort returns the same result when given the same input list.

We do so by first establishing a very general result about insertion-sort, that completely characterizes its output. Prove:

- * (1) $(\text{ordered } y) \wedge (\text{perm } x y) \Rightarrow (\text{isort } x) = y$

Using the above, prove that:

$$(2) \text{ (isort } x) = \text{(msort } x)$$

Problem 10: Stump your TA

This is optional. Send directly to the TA running your lab.

Come up with one conjecture (it can be true or false), that (1) can be expressed using 300 characters or less and (2) only uses functions mentioned in this homework. Then, challenge your TA to either exhibit a counterexample or a proof. If you come up with a really challenging conjecture, then you will get 20 extra points on your homework grade. Send this directly to the TA in charge of your lab.

Problem 11: Use ACL2

This is optional and worth an extra 30 points.

Warning: we haven't covered all you need to know to complete this exercise, but for those of you who like to live dangerously, who want a challenge, and who are willing to do some amount of reading, this is one is for you.

Use ACL2s to prove all the theorems you proved in your homework and submit the ACL2s proof script. You have to use ACL2s in ACL2s mode; see the ACL2s Web page for details. You can also submit partial solutions and we can give you partial credit.