Isabelle/HOL Exercises
Advanced

Sorting with Lists and Trees

For simplicity we sort natural numbers.

Sorting with lists

The task is to define insertion sort and prove its correctness. The following functions are required:

```
consts
  insort :: "nat ⇒ nat list ⇒ nat list"
  sort   :: "nat list ⇒ nat list"
  le     :: "nat ⇒ nat list ⇒ bool"
  sorted :: "nat list ⇒ bool"
```

In your definition, `sol.insort x xs` should insert a number `x` into an already sorted list `xs`, and `sol.sort ys` should build on `insort` to produce the sorted version of `ys`.

To show that the resulting list is indeed sorted we need a predicate `sol.sorted` that checks if each element in the list is less or equal to the following ones; `le n xs` should be true iff `n` is less or equal to all elements of `xs`.

```
primrec
  "le a []     = True"
  "le a (x#xs) = (a <= x & le a xs)"

primrec
  "sorted []     = True"
  "sorted (x#xs) = (le x xs & sorted xs)"

primrec
  "insort a []     = [a]"
  "insort a (x#xs) = (if a <= x then a#x#xs else x # insort a xs)"

primrec
  "sort []     = []"
  "sort (x#xs) = insort x (sort xs)"
```

Start out by showing a monotonicity property of `le`. For technical reasons the lemma should be phrased as follows:

```
lemma [simp]: "x ≤ y ⟹ le y xs ⟶ le x xs"
  apply (induct_tac xs)
  apply auto
done
```

Now show the following correctness theorem:

```
lemma [simp]:
  "le x (insort a xs) = (x <= a & le x xs)"
  apply (induct_tac xs)
  apply auto
done

lemma [simp]:
  "sorted (insort a xs) = sorted xs"
  apply (induct_tac xs)
  apply auto
done

theorem "sorted (sort xs)"
  apply (induct_tac xs)
  apply auto
done
```

This theorem alone is too weak. It does not guarantee that the sorted list contains the same elements as the input. In the worst case, `sol.sort` might always return `[]` – surely an undesirable implementation of sorting.

Define a function `count xs x` that counts how often `x` occurs in `xs`.

```
consts
  count :: "nat list => nat => nat"
primrec
  "count []     y = 0"
  "count (x#xs) y = (if x=y then Suc(count xs y) else count xs y)"
```

Show that

```
lemma [simp]:
  "count (insort x xs) y =
  (if x=y then Suc (count xs y) else count xs y)"
  apply (induct_tac xs)
  apply auto
done
```

```
theorem "count (sort xs) x = count xs x"
  apply (induct_tac xs)
  apply auto
done
```

Sorting with trees

Our second sorting algorithm uses trees. Thus you should first define a data type *bintree* of binary trees that are either empty or consist of a node carrying a natural number and two subtrees.

```
datatype bintree = Empty | Node nat bintree bintree
```

Define a function *tsorted* that checks if a binary tree is sorted. It is convenient to employ two auxiliary functions *tge*/*tle* that test whether a number is greater-or-equal/less-or-equal to all elements of a tree.

Finally define a function *tree_of* that turns a list into a sorted tree. It is helpful to base *tree_of* on a function *ins n b* that inserts a number *n* into a sorted tree *b*.

```
consts
  tsorted :: "bintree ⇒ bool"
  tge :: "nat ⇒ bintree ⇒ bool"
  tle :: "nat ⇒ bintree ⇒ bool"
  ins :: "nat ⇒ bintree ⇒ bintree"
  tree_of :: "nat list ⇒ bintree"

primrec
  "tsorted Empty          = True"
  "tsorted (Node n t1 t2) = (tsorted t1 ∧ tsorted t2 ∧ tge n t1 ∧ tle n t2)"

primrec
  "tge x Empty          = True"
  "tge x (Node n t1 t2) = (n ≤ x ∧ tge x t1 ∧ tge x t2)"

primrec
  "tle x Empty          = True"
  "tle x (Node n t1 t2) = (x ≤ n ∧ tle x t1 ∧ tle x t2)"

primrec
  "ins x Empty          = Node x Empty Empty"
  "ins x (Node n t1 t2) = (if x ≤ n then Node n (ins x t1) t2 else Node n t1
(ins x t2))"
```

```
primrec
  "tree_of []     = Empty"
  "tree_of (x#xs) = ins x (tree_of xs)"
```

Show

```
lemma [simp]: "tge a (ins x t) = (x ≤ a ∧ tge a t)"
  apply (induct_tac t)
  apply auto
done

lemma [simp]: "tle a (ins x t) = (a ≤ x ∧ tle a t)"
  apply (induct_tac t)
  apply auto
done

lemma [simp]: "tsorted (ins x t) = tsorted t"
  apply (induct_tac t)
  apply auto
done

theorem [simp]: "tsorted (tree_of xs)"
  apply (induct_tac xs)
  apply auto
done
```

Again we have to show that no elements are lost (or added). As for lists, define a function `tcount x b` that counts the number of occurrences of the number `x` in the tree `b`.

```
consts
  tcount :: "bintree => nat => nat"
primrec
  "tcount Empty          y = 0"
  "tcount (Node x t1 t2) y = (if x=y then
                                Suc (tcount t1 y + tcount t2 y)
                              else
                                tcount t1 y + tcount t2 y)"
```

Show

```
lemma [simp]: "tcount (ins x t) y =
  (if x=y then Suc (tcount t y) else tcount t y)"
  apply(induct_tac t)
  apply auto
done
```

```
theorem "tcount (tree_of xs) x = count xs x"
  apply (induct_tac xs)
  apply auto
done
```

Now we are ready to sort lists. We know how to produce an ordered tree from a list. Thus we merely need a function `list_of` that turns an (ordered) tree into an (ordered) list. Define this function and prove

```
theorem "sorted (list_of (tree_of xs))"
theorem "count (list_of (tree_of xs)) n = count xs n"
```

Hints:

- Try to formulate all your lemmas as equations rather than implications because that often simplifies their proof. Make sure that the right-hand side is (in some sense) simpler than the left-hand side.
- Eventually you need to relate `sorted` and `tsorted`. This is facilitated by a function `ge` on lists (analogously to `tge` on trees) and the following lemma (that you will need to prove):

```
sol.sorted (a @ x # b) = (sol.sorted a ∧ sol.sorted b ∧ ge x a ∧ le x b)
```

```
consts
  ge      :: "nat ⇒ nat list ⇒ bool"
  list_of :: "bintree ⇒ nat list"

primrec
  "ge a []     = True"
  "ge a (x#xs) = (x ≤ a ∧ ge a xs)"

primrec
  "list_of Empty          = []"
  "list_of (Node n t1 t2) = list_of t1 @ [n] @ list_of t2"

lemma [simp]: "le x (a@b) = (le x a ∧ le x b)"
  apply (induct_tac a)
  apply auto
done

lemma [simp]: "ge x (a@b) = (ge x a ∧ ge x b)"
  apply (induct_tac a)
```

```
  apply auto
done

lemma [simp]:
  "sorted (a@x#b) = (sorted a ∧ sorted b ∧ ge x a ∧ le x b)"
  apply (induct_tac a)
  apply auto
done

lemma [simp]: "ge n (list_of t) = tge n t"
  apply (induct_tac t)
  apply auto
done

lemma [simp]: "le n (list_of t) = tle n t"
  apply (induct_tac t)
  apply auto
done

lemma [simp]: "sorted (list_of t) = tsorted t"
  apply (induct_tac t)
  apply auto
done

theorem "sorted (list_of (tree_of xs))"
  by auto

lemma count_append [simp]: "count (a@b) n = count a n + count b n"
  apply (induct a)
  apply auto
done

lemma [simp]: "count (list_of b) n = tcount b n"
  apply (induct b)
  apply auto
done

theorem "count (list_of (tree_of xs)) n = count xs n"
  apply (induct xs)
  apply auto
done
```