Isabelle/HOL Exercises
Advanced

Merge Sort

Sorting with lists

For simplicity we sort natural numbers.

Define a predicate *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 :: "nat ⇒ nat list ⇒ bool" where
  "le a []     = True"
| "le a (x#xs) = (a <= x & le a xs)"

primrec sorted :: "nat list ⇒ bool" where
  "sorted []     = True"
| "sorted (x#xs) = (le x xs & sorted xs)"
```

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

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

Merge sort

Implement *merge sort*: a list is sorted by splitting it into two lists, sorting them separately, and merging the results.

Define the two functions *merge* and *msort* for merging and sorting, respectively.

```
fun
 merge :: "nat list ⇒ nat list ⇒ nat list"
where
 "merge [] ys = ys" |
 "merge xs [] = xs" |
 "merge (x # xs) (y # ys) = (
   if x ≤ y
     then x # merge xs (y # ys)
     else y # merge (x # xs) ys
```

```
 )"

fun
 msort :: "nat list ⇒ nat list"
where
 "msort [] = []" |
 "msort [x] = [x]" |
 "msort xs = (
   let half = length xs div 2 in
   merge (msort (take half xs)) (msort (drop half xs))
 )"

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

lemma [simp]: "count (merge xs ys) x = count xs x + count ys x"
  apply(induct xs ys rule: merge.induct)
  apply auto
done

lemma [simp]: "le x (merge xs ys) = (le x xs ∧ le x ys)"
  apply (induct xs ys rule: merge.induct)
  apply auto
done

lemma [simp]: "sorted (merge xs ys) = (sorted xs ∧ sorted ys)"
  apply(induct xs ys rule: merge.induct)
  apply (auto simp add: linorder_not_le order_less_le)
done

lemma [simp]: "1 < x ⟹ min x (x div 2::nat) < x"
  by (simp add: min_def linorder_not_le)

lemma [simp]: "1 < x ⟹ x - x div (2::nat) < x"
  by arith

theorem "sorted (msort xs)"
  apply (induct_tac xs rule: msort.induct)
  apply auto
done
```

```
lemma count_append[simp]: "count (xs @ ys) x = count xs x + count ys x"
  apply (induct xs)
  apply auto
done

theorem "count (msort xs) x = count xs x"
  apply (induct xs rule: msort.induct)
      apply simp
    apply simp
  apply simp
  apply (simp del:count_append add:count_append[symmetric])
done
```

An alternative solution in Isabelle/Isar

If some element *x* is less than or equal to all elements of the lists *ys* and *zs*, then this also holds true for the merged lists.

```
lemma le_merge[simp]:
 assumes "le x ys" and "le x zs" shows "le x (merge ys zs)"
using assms by (induct ys zs rule: merge.induct) simp_all

lemma le_le_simps[simp]:
 "x ≤ y ⟹ le y ys ⟹ le x ys"
 "¬ x ≤ y ⟹ le x xs ⟹ le y xs"
by (induct ys, simp_all) (induct xs, simp_all)
```

Merging lists preserves sortedness.

```
lemma sorted_merge[simp]:
 assumes "sorted xs" and "sorted ys" shows "sorted (merge xs ys)"
using assms by (induct xs ys rule: merge.induct) simp_all
```

The result of *msort xs* is a sorted list.

```
theorem "sorted (msort xs)" by (induct xs rule: msort.induct) simp_all
```

Merging does neither remove nor add elements.

```
lemma count_merge: "count (merge xs ys) x = count xs x + count ys x"
by (induct xs ys rule: merge.induct) auto

lemma cnt_append: "count (xs @ ys) x = count xs x + count ys x"
by (induct xs) auto
```

lemma *take_drop_count: "count (take n xs) x + count (drop n xs) x = count xs x"*
unfolding *count_append[symmetric]* **by** *simp*

Sorting does neither remove nor add elements (important, since functions like *wrong_sort xs = []* would also satisfy *sol.sorted (wrong_sort xs)*).

theorem *"count (msort xs) x = count xs x"*
by *(induct xs rule: msort.induct) (simp_all add: take_drop_count)*