test :: ExerciseSheet(6) (18.5.2010)

1. **Exercise (Complexity of Naive Reverse)**
   
   Show that the naive reverse function needs quadratically many
   Cons operations in the length of the input list.
   (Note that \(\text{[x]}\) is syntax sugar for \(\text{Cons} \ x \ \text{[]}\).)

   ```
   fun reverse where
   "reverse [x] = [x]"
   
   (** Define cost functions and prove that they are equal to quadratic function **)  
   ```

2. **Exercise (Simple Paths)**
   
   Recall the definition of paths from last exercise sheet:

   ```
   data Path = Path [Int]
   ```

   ```
   list last :: [a] -> Maybe [a]
   list last xs | last xs = Nothing |
   list last xs = Just (tail xs)
   ```

   ```
   lemma set_rec :: "\text{set} \ x \ = \ \text{rec_list} () (\text{if} _ \ \text{then} \ \text{else} \ \text{but_last} \ x)"
   by (induct xs) auto
   ```

   ```
   lemma coset :: "\text{coset} \ x \ = \ \text{set} \ x"
   proof
     \text{apple coset} \ x \ = \ \text{set} \ x
   qed
   ```

   ```
   lemma rev :: "\text{rev} \ x \ = \ \text{rev} \ x"
   proof
     \text{rev} \ [x] = [x]
     \text{rev} \ x \ \# \ x = \ \text{rev} \ x \ \# \ x
   ```

   ```
   lemma cons :: "\text{cons} \ x \ y \ = \ \text{rev} \ \text{cons} \ y \ x"
   proof
     \text{cons} \ x \ y = \ \text{rev} \ \text{cons} \ y \ x
   ```

   ```
   lemma rev1 :: "\text{rev} \ x \ = \ \text{rev} \ x"
   proof
     \text{rev} \ x \ \# \ x = \ \text{rev} \ x \ \# \ x
   ```

   ```
   test :: ExerciseSheet(6) (18.5.2010)
   ```

   ```
   fun reverse where
   "reverse [x] = [x]"
   
   (** Define cost functions and prove that they are equal to quadratic function **)  
   ```
fun reverse \_.

fun reverse \_.

fun reverse \_.

fun reverse \_.

fun reverse \_.

fun reverse \_.

fun reverse \_.

fun reverse \_.

fun reverse \_.

fun reverse \_.

fun reverse \_.

fun reverse \_.

fun reverse \_.

fun reverse \_.

fun reverse \_.

fun reverse \_.
fun reverse where
  "reverse [] = []"
  "reverse (x#xs) = reverse xs @ [x]"

fun t_reverse where
  "t_reverse [] = 0"
  "t_reverse (x#xs) = t_reverse xs + 1 + t_append (x, y) @ [x]"

(* Define cost functions and prove that they are equal to quadratic function *)

(t - Exercise(Simple Paths))

consts
  reverse :: "'a list ⇒ 'a list"

Found termination order: "length ◦ rlex" ()

(t - Exercise(Simple Paths))

Note that (\{x\}) is syntax sugar for (Cons x [])

fun t_append :: "'a list ⇒ 'a list ⇒ nat" where
  "t_append [] y = 0"
  "t_append (x#xs) ys = 1 + t_append xs ys"

fun append.simps

fun reverse where
  "reverse [] = []"
  "reverse (x#xs) = reverse xs @ [x]"

fun t_reverse where
  "t_reverse [] = 0"
  "t_reverse (x#xs) = t_reverse xs + t_append (x, y) @ [x]"

(* Define cost functions and prove that they are equal to quadratic function *)

(t - Exercise(Simple Paths))

consts
  reverse :: "'a list ⇒ 'a list"

Found termination order: "length ◦ rlex" ()

(t - Exercise(Simple Paths))

Note that (\{x\}) is syntax sugar for (Cons x [])

fun t_append :: "'a list ⇒ 'a list ⇒ nat" where
  "t_append [] y = 0"
  "t_append (x#xs) ys = 1 + t_append xs ys"

fun append.simps

fun reverse where
  "reverse [] = []"
  "reverse (x#xs) = reverse xs @ [x]"

fun t_reverse where
  "t_reverse [] = 0"
  "t_reverse (x#xs) = t_reverse xs + t_append (x, y) @ [x]"

(* Define cost functions and prove that they are equal to quadratic function *)

(t - Exercise(Simple Paths))

consts
  reverse :: "'a list ⇒ 'a list"

Found termination order: "length ◦ rlex" ()
fun t_append : "a list \to \text{nat} \to \text{nat} where \\
\text{t_append} [\text{ys} \mid \text{ys}] \equiv 1 + \text{t_append} \text{xs} \text{ys}
\\
lemma \text{t_append} \text{xs} \text{ys} \equiv \text{length} \text{x} + \text{length} \text{y}
\\
fun reverse where \\
\text{reverse} [] \equiv []
\\
lemma \text{simp} : \text{length} (\text{reverse} \text{xs}) = \text{length} \text{xs} \text{by induction on} \text{xs}
\\
fun t_reverse where \\
\text{t_reverse} [] \equiv 0
\\
\text{t_reverse} \text{[xs]} \equiv \text{t_reverse} \text{xs} + \text{t_append} (\text{reverse} \text{xs}) [0]
\\
const \\
\text{t_reverse} [] : \text{"a list \to \text{nat}"

"\text{lemma} \text{simp} : \text{length} (\text{reverse} \text{xs}) = \text{length} \text{xs} \text{by induction on} \text{xs}

"\text{fun} \text{t_reverse} \text{where}

"\text{fun} \text{t_append} \text{where} \\
\text{\text{t_append} [\text{ys} \mid \text{ys}] \equiv 1 + \text{t_append} \text{xs} \text{ys}
fun t_reverse where
  | t_reverse [] = 0
  | t_reverse (xs:xs) = t_reverse xs + t_append (reverse xs) [x] + 1

  t n ml = t n + n + 1

(* Define cost functions and prove that they are equal to quadratic function *)

consts
  t_reverse :: "a list → nat"

Found termination order: "length <"rel> {}"

("0, 1, 3, 6, 10, 15, 21")
  []: "nat list"

proof (prove)
  goal (1 subgoal):
  1. Ax xs. t_reverse xs = (length xs + length xs + length xs) div 2 →
     (length xs + length xs + length xs) div 2 = t_append (reverse xs) [x] =
     (length xs + (length xs + (length xs + length xs + length xs))) div 2

(* Define cost functions and prove that they are equal to quadratic function *)
fun t_reverse where
   t_reverse [] = 0
   | t_reverse (xs:xss) = t_reverse xs + t_append (reverse xs) [x] + 1

val = map (\i. t_reverse (replicate n (i::int))) [0,1,2,3,4,5,6]

consts
   t_reverse :: "a list ⇒ nat"
   length :: "('a list) ⇒ nat"

fun path :: "('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a list ⇒ 'a ⇒ bool"
where
   "path G U [] v = true"
   | "path G U (x:xs) v = G U (x v) ∧ path G U xs v"

lemma path_append_Isimp : "path G U (pin p2) v =⇒ (G v. path G U p1 v ∧ path G U p2 v)"
   by (induction p1 arbitrary: u) auto

consts
   path :: "('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a list ⇒ 'a ⇒ bool"

fun simple_path where
   "simple_path x y (length (fst v (snd v))) (\x. x y) () = x y"
Show that for every path, there is a corresponding simple path.

Hint: Induction on the length of the path.

- \( \text{measure_induct_rule} \)
- \( \text{measure_induct_rule} \) (where \( f = \text{length} \), \( \text{case_names shorter} \))
- \( \text{not_distinct_decomp} \)

Lemma exists_simple_path:
- assumes "path \( G \cup p' \)"
- shows "\( \exists p'. \text{ path } G \cup p' \cap \text{ distinct } p' \)"

\( \forall x, \forall y, \forall \gamma \in \text{length} x \rightarrow \exists \gamma \ y \rightarrow \exists \gamma \ x \rightarrow \exists \gamma \ a \)

\( \forall x, \forall y, \forall y \in \text{length} x \rightarrow \exists \gamma \ y \rightarrow \exists \gamma \ x \rightarrow \exists \gamma \ a \)
proof (prove)
  goal (1 subgoal):
  1. \( p' \cdot \text{path} \ 0 \ 0' \ p' \ \wedge \text{distinct} \ p' \)
proof (state)
goal (1 subgoal):
  1. \(\forall x. [\text{length } y < \text{length } x; \text{path } G \cup y \cup v] \rightarrow \exists p'. \text{path } G \cup p' \cup v \land \text{distinct } p'\)

  Proof outline with cases:
  - case (shorter $x$)
  - then show ?case sorry

qed
proof (state)

this:
. \[ \text{length } y < \text{length } p ; \text{ path } G u y v \] \implies \exists p'. \quad \text{path } G u p' v \wedge \text{distinct } p'

goal (1 subgoal):
1. \( \forall x. \ [\text{length } y < \text{length } x ; \text{ path } G u y v] \implies \exists p'. \quad \text{path } G u p' v \wedge \text{distinct } p'; \text{ path } G u x v \]
   \implies \exists p'. \quad \text{path } G u p' v \wedge \text{distinct } p'

proof (state)

this:
. \[ \text{length } y < \text{length } p ; \text{ path } G u y v \] \implies \exists p'. \quad \text{path } G u p' v \wedge \text{distinct } p'

goal (1 subgoal):
1. \( \forall x. \ [\text{length } y < \text{length } x ; \text{ path } G u y v] \implies \exists p'. \quad \text{path } G u p' v \wedge \text{distinct } p'; \text{ path } G u x v \]
   \implies \exists p'. \quad \text{path } G u p' v \wedge \text{distinct } p'

proof (state)

this:
. \[ \text{length } y < \text{length } p ; \text{ path } G u y v \] \implies \exists p'. \quad \text{path } G u p' v \wedge \text{distinct } p'

goal (1 subgoal):
1. \( \forall x. \ [\text{length } y < \text{length } x ; \text{ path } G u y v] \implies \exists p'. \quad \text{path } G u p' v \wedge \text{distinct } p'; \text{ path } G u x v \]
   \implies \exists p'. \quad \text{path } G u p' v \wedge \text{distinct } p'

proof (prove)

goal (1 subgoal):
1. \( \exists p'. \quad \text{path } G u p' v \wedge \text{distinct } p' \)
proof (state)
this:
- \text{distinct } p

goal (1 subgoal):
1. \text{\neg distinct } p \implies \exists p', \text{ path } 6 \cup p' \cup v \land \text{distinct } p'

proof (state)
this:
\begin{align*}
p & = p_1 \cup p_2 \cup p_3 \\
p & = p_1 \cup p_2 \cup p_3
\end{align*}

proof (state)
this:
\begin{align*}
p & = p_1 \cup p_2 \cup p_3 \\
p & = p_1 \cup p_2 \cup p_3
\end{align*}

proof (chain)
picking this:
\begin{align*}
x & = zs \\
x & = zs & x & = x & \exists y | y \in x & 0 & 0 & zs
\end{align*}
proof (prove)
using this:
\[ p = p_1 \# \# p_2 \# \# p_3 \]
goal (1 subgoal):
1. path \( U \cup (p_1 \# \# p_3) \) \( v \)
proof (prove)

goal (1 subgoal):
1. \( \forall w. \forall p. (p \neq p_1 \land \# p \neq \# p_2 \land \# p \neq p_3) \rightarrow \text{path } G \cup (p \oplus p_3) \cup v' \rightarrow \text{thesis} \)

proof (state)

this:

\( p \equiv p_1 \land \# p \equiv \# p_2 \land \# p \equiv p_3 \)

goal (1 subgoal):
1. distinct \( p \rightarrow \) \( \exists p'. \text{path } G \cup p' \cup v \land \text{distinct } p' \)
assume ~distinct p 
from not_distinct_decomp[of this] obtain p1 p2 p3 x where ~impl p1 p2 p3 x 
by auto  
with shorter_prem obtain s1 s2 where 
"path 6 u s1 x" "path 6 x s2 x" "path 6 x p3 x" 
by auto 
then have "path 6 u (p1#p2#p3) v" by auto 

...  
show thesis sorry 
qed

have path 6 u (p1 # x # p3) v 
proof (state): 
this:  
path 6 u (p1 # x # p3) v

proof (prove): 
using this: 
path 6 u v 

proof (prove) 
using this: 
path 6 u v 

proof (prove) 
using this: 
path 6 u v 

proof (prove) 
using this: 
path 6 u v 

proof (prove) 
using this: 
path 6 u v 

proof (prove) 
using this: 
path 6 u v
proof (prove)
  using assume
  proof (induction p rules len_induct)
    case (shorter p)
      the shorter_presume_assume
      show tcase proof cases
        assume "distinct p" then show ?thesis using shorter_presume_assume by blast
      next
    assume "distinct p" then show ?thesis using shorter_presume_assume by blast
next

proof (prove)
  using assume
  proof (induction p rules len_induct)
    case (shorter p)
      the shorter_presume_assume
      show tcase proof cases
        assume "distinct p" then show ?thesis using shorter_presume_assume by blast
      next
    assume "distinct p" then show ?thesis using shorter_presume_assume by blast
next

assumes "path p u v" shows "p', path p u v' \& \& distinct p'"
using assume
proof (induction p rules len_induct)
  case (shorter p)
    term x term p
    show tcase proof cases
      assume "distinct p" then show ?thesis using shorter_presume_assume by blast

proof (prove)
  using assume
  proof (induction p rules len_induct)
    case (shorter p)
      the shorter_presume_assume
      show tcase proof cases
        assume "distinct p" then show ?thesis using shorter_presume_assume by blast
      next
    assume "distinct p" then show ?thesis using shorter_presume_assume by blast
next

path 0 u v
path 0 u v

path 0 u v
path 0 u v
proof (induction p rules: len_induct)
  case (shorter p)
    the shorter.prems assume

  show thesis proof cases
    assume "distinct p" then show thesis using PATH oops by blast
  next
    assume "not distinct p"
    from not_distinct_decomp[OF this] obtain p1 p2 p x where [simp]: "p = p1::'a list @ p2"
    by auto
    from shorter.prems obtain x1 x2 where "path 0 u p1 x1" "0 x1 x" "path 0 x p x2" "x2 x p x" by auto
    then have "path 0 u (play p3) v" by auto
    from shorter.IH[OF this] show thesis by simp

qed

text {*

NumeralSort (Stability of Insertion Sort) [Key 25]

Have a look at Isabelle's standard implementation of sorting: @ (const sort_key),
(Use Ctrl-Click to jump to the definition in @ (file "~~/src/HOL/List.thy"))
Show that this function is a stable sorting algorithm, i.e., the order of elements
with the same key is not changed during sorting!

lemma "[x,x,x, x] = [x,x, x, x]"
  oops

theorem "[x,x, P x]"

theorem exists_simple_path
  path 76 u p v v" = >> path 76 u p v v' v' and distinct p'
text -insertsort.sty

4 Have a look at Isabelle’s standard implementation of sorting: (const sort.key).

4 (Use Ctrl-Click to jump to the definition in @file “~/src/MOD/List.thy”)

4 Show that this function is a stable sorting algorithm, i.e., the order of elements

4 with the same key is not changed during sorting!

4 lemma “[x=sort_key k x = a] = [x=x.s . k x = a]”

4 oops

4 term “[x=x.s . P x]”

4 hint: You do not necessarily need Isar, and the auxiliary lemmas

4 you need are already in Isabelle’s library. @command find_theorems is your friend!
Have a look at Isabelle's standard implementation of sorting: 

```isar
(Use Ctrl-Click to jump to the definition in @file "~/src/HOL/List.thy")
Show that this function is a stable sorting algorithm, i.e., the order of elements with the same key is not changed during sorting:

```isar
lemma "\{\text{-sort_key} k \cdot x = a\} = \{\text{-xs.} k \cdot x = a\}"
```isar
oops
```isar
term "\{\text{-xs.} P \cdot x\}"
```isar
Note: \(\text{term} \[\{\text{-xs.} P \cdot x\}\] \) is syntax sugar for \(\text{term} \[\{\text{filter} P \cdot \text{xss}\}\] \), where the filter function returns only the elements of list \(\cdot \text{xss}\) for which \(P \cdot \text{x} = \text{True}\).
```isar
Hint: You do not necessarily need \text{Isar}, and the auxiliary lemmas you need are already in Isabelle's library. \text{Command find_theorems} is your friend!
```isar
```

(Use Ctrl-Click to jump to the definition in @file "~/src/HOL/List.thy")
Show that this function is a stable sorting algorithm, i.e., the order of elements with the same key is not changed during sorting:

```isar
lemma "\{\text{-sort_key} k \cdot x = a\} = \{\text{-xs.} k \cdot x = a\}"
```isar
oops
```isar
term "\{\text{-xs.} P \cdot x\}"
```isar
Note: \(\text{term} \[\{\text{-xs.} P \cdot x\}\] \) is syntax sugar for \(\text{term} \[\{\text{filter} P \cdot \text{xss}\}\] \), where the filter function returns only the elements of list \(\cdot \text{xss}\) for which \(P \cdot \text{x} = \text{True}\).
```isar
Hint: You do not necessarily need \text{Isar}, and the auxiliary lemmas you need are already in Isabelle's library. \text{Command find_theorems} is your friend!
```isar
```
fun quickselect :: "'a list ⇒ nat ⇒ 'a" where
  "quickselect (x#xs) k = if k < length xs then quickselect xs k
   else if k = length xs then x
   else quickselect xs2 (k - length xs - 1)"

"quickselect [] = undefined"

text "Your first task is to prove the crucial idea of quicksort, i.e., that partitioning \( \text{rt} \setminus \{ pivot \} \) a pivot element \( p \) is correct.

consts quickselect :: "'a list ⇒ nat ⇒ 'a"

Found termination order: "(list, size (and p))\(<\text{lex}>(\text{Ap}, \text{length (fst p)})\)<\text{lex}> {}"

proof (prove)

proof (prove)

goal (1 subgoal):
  1. sort xs = sort [x # xs, x < p] \@ sort [x # xs, x > p]

lemma partition_correct: "sort xs = sort [x # xs, x < p] \@ sort [x # xs, x > p]"

lemma partition_correct: "sort xs = sort [x # xs, x < p] \@ sort [x # xs, x > p]"

text "Hint: Induction, and auxiliary lemmas to transform a term of the form \( \text{merge-insert-sort-insert}(\text{merge-insert}(x, y)) \) when you know that \( x \) is greater than all elements in \( \text{xs} \) / less than or equal all elements in \( \text{ys} \)."

text "Hint: Induction, and auxiliary lemmas to transform a term of the form \( \text{merge-insert-sort-insert}(\text{merge-insert}(x, y)) \) when you know that \( x \) is greater than all elements in \( \text{xs} \) / less than or equal all elements in \( \text{ys} \)."
Your first task is to prove the crucial idea of quicksort, i.e., that partitioning \( w r t . \) a pivot element \( s p \) is correct.

**Lemma partition_correct:** \(
\text{sort}_{x} = \text{sort}_{x, x < x_{p}} \uplus \text{sort}_{x < x_{p}}\)

**Proof (state)**

\( \text{if } k < \text{length} [y < x_{p}, y < x] \)
\( \Rightarrow \text{quickselect} [y < x_{p}, y < x] = \text{sort} [y < x_{p}, y < x] + k \)
\( k < \text{length} [y < x_{p}, y < x] \)
\( \Rightarrow \text{quickselect} [y < x_{p}, y < x] = (k - \text{length} [y < x_{p}, y < x]) + 1 \)
\( \Rightarrow \text{sort} [y < x_{p}, y < x] = (k - \text{length} [y < x_{p}, y < x]) + 1 \)

\( \text{Note: To make the induction hypothesis more readable, you can collapse the first two premises of the form } \text{sort}_{x} \text{ by reflexivity.} \)
\( \text{Note } H = \text{"I'' DH\ if refl refl"} \)

Your first task is to prove the crucial idea of quicksort, i.e., that partitioning \( w r t . \) a pivot element \( s p \) is correct.