Output

```
---------------------------------------------------------------------
       HOL-4 [Kananaskis 11 (stdknl, built Thu May 05 10:43:02 2016)]

       For introductory HOL help, type: help "hol";
       To exit type <Control>-D
---------------------------------------------------------------------
[extending loadPath with Holmakefile INCLUDES variable]
[Use-ing configuration file /Users/josephchan/.hol-config.sml]
> >
(* ------------------------------------------------------------------------- *)
(* LCM Bound by Triangle - Demo                                              *)
(* ------------------------------------------------------------------------- *)

(* prefix *)

val _ = HOL_Interactive.toggle_quietdec();

val _ = load "lcsymtacs";
open lcsymtacs;

val _ = load "jcLib";
open jcLib;

val _ = load "SatisfySimps"; (* for SatisfySimps.SATISFY_ss *)

(* Get dependent theories local *)

val _ = load "triangleTheory";
open triangleTheory;
open binomialTheory;

(* Get dependent theories in lib *)
open helperNumTheory;
open helperListTheory;

(* open dependent theories *)
open arithmeticTheory;
open pred_setTheory;
open listTheory;

(* open dependent theories *)
open dividesTheory gcdTheory;

open listRangeTheory; (* use listRange: [1 .. 3] = [1; 2; 3], [1 ..< 3] = [1; 2] *)
open rich_listTheory; (* for EVERY_REVERSE *)
open relationTheory;  (* for RTC, Reflexive Transitive Closure *)

set_trace "Unicode" 1;                    (* display using Unicode *)
set_trace "Goalstack.print_goal_at_top" 0;     (* display goal at bottom *)

val _ = clear_overloads_on "leibniz_vertical"; (* better display *)
val _ = overload_on("EVERY_POSITIVE", ``\l. EVERY (λk. 0 < k) l``); (* better display *)

val _ = HOL_Interactive.toggle_quietdec();
>
(* Press SPACE bar to continue *)
(* ------------------------------------------------------------------------- *)
(* LCM of a list of numbers                                                  *)
(* ------------------------------------------------------------------------- *)

(* Our definition of the LCM of a list of numbers. *)
list_lcm_def;
val it =
   ⊢ (list_lcm [] = 1) ∧ ∀h t. list_lcm (h::t) = lcm h (list_lcm t):
   thm
>
(* Note that it covers two cases:
   when the list is empty [], and
   when the list consists of head (h) and tail (t).
*)

(* Let's compute some values. *)
EVAL ``list_lcm [1]``;
# # # # val it = ⊢ list_lcm [1] = 1: thm
> EVAL ``list_lcm [1; 2]``;
val it = ⊢ list_lcm [1; 2] = 2: thm
> EVAL ``list_lcm [1; 2; 3]``;
val it = ⊢ list_lcm [1; 2; 3] = 6: thm
> EVAL ``list_lcm [1; 2; 3; 4]``;
val it = ⊢ list_lcm [1; 2; 3; 4] = 12: thm
> EVAL ``list_lcm [1; 2; 3; 4; 5]``;
val it = ⊢ list_lcm [1; 2; 3; 4; 5] = 60: thm
> EVAL ``list_lcm [1; 2; 3; 4; 5; 6]``;
val it = ⊢ list_lcm [1; 2; 3; 4; 5; 6] = 60: thm
```

```
>
(* Basic properties of list_lcm. *)
list_lcm_sing;
val it = ⊢ ∀x. list_lcm [x] = x: thm
>
list_lcm_append;
val it =
   ⊢ ∀l1 l2. list_lcm (l1 ++ l2) = lcm (list_lcm l1) (list_lcm l2):
   thm
>
list_lcm_reverse;
val it =
   ⊢ ∀l. list_lcm (REVERSE l) = list_lcm l:
   thm
>
list_lcm_is_common_multiple;
val it =
   ⊢ ∀x l. MEM x l ⇒ x divides list_lcm l:
   thm
>
(* Note this pattern: after introducing something new, collect its properties. *)

(* Since each member x divides the list_lcm, each x ≤ list_lcm, giving: *)
list_lcm_lower_bound;
val it =
   ⊢ ∀l. EVERY_POSITIVE l ⇒ SUM l ≤ LENGTH l * list_lcm l:
   thm
>
(* This establishes a general lower bound for list_lcm. *)

(* ------------------------------------------------------------------------- *)
(* Leibniz Triangle (Denominator form)                                       *)
(* ------------------------------------------------------------------------- *)

(* Leibniz Triangle definition: L n k = entry at n-th row, k-th column. *)
leibniz_def;
val it =
   ⊢ ∀n k. leibniz n k = (n + 1) * binomial n k:
   thm
>

(* Basic properties of Leibniz Triangle entries. *)
leibniz_n_0;
val it = ⊢ ∀n. leibniz n 0 = n + 1: thm
>
leibniz_n_n;
val it =
   ⊢ ∀n. leibniz n n = n + 1:
   thm
>
leibniz_sym;
val it =
   ⊢ ∀n k. k ≤ n ⇒ (leibniz n k = leibniz n (n − k)):
   thm
>
leibniz_pos;
val it =
   ⊢ ∀n k. k ≤ n ⇒ 0 < leibniz n k:
   thm
>

(*
Picture of Leibniz Triplet:

     b = L (n-1) k
     a = L n     k   c = L n (k+1)

with a * b = c * (a - b).
*)

(* A Leibniz triplet. *)
triplet_def;
# # # # # # val it =
   ⊢ ∀n k.
     triplet n k =
     <|a := leibniz n k; b := leibniz (n + 1) k;
       c := leibniz (n + 1) (k + 1)|>:
   thm
>

(* Overload elements of a triplet *)
val _ = overload_on("ta", ``(triplet n k).a``);
> val _ = overload_on("tb", ``(triplet n k).b``);
> val _ = overload_on("tc", ``(triplet n k).c``);
>
(* Basic properties *)
```

```
leibniz_triplet_property;
val it = ⊢ ∀n k. ta * tb = tc * (tb − ta): thm
>

(* This result matches the condition for LCM exchange. *)
LCM_EXCHANGE;
val it =
   ⊢ ∀a b c. (a * b = c * (a − b)) ⇒ (lcm a b = lcm a c):
   thm
>
(* We want to apply this to the Leibniz triplet, and prove: lcm tb ta = lcm tb tc *)

(* State what we want to prove as a goal. *)

(* <<leibniz_triplet_lcm>> *)
g `lcm tb ta = lcm tb tc`;
val it =
   Proof manager status: 1 proof.
1. Incomplete goalstack:
     Initial goal:


     lcm tb ta = lcm tb tc

:
   proofs
> (* Simplify by property of triplet and library theorem. *)
e (simp[leibniz_triplet_property, LCM_EXCHANGE]);
<<HOL message: Initialising SRW simpset ... done>>
OK..
val it = Initial goal proved.
⊢ lcm tb ta = lcm tb tc: proof
> (* Theorem proved, save it with a name, then clean up workspace. *)
val leibniz_triplet_lcm = save_thm("leibniz_triplet_lcm", top_thm());
val leibniz_triplet_lcm = ⊢ lcm tb ta = lcm tb tc: thm
> drop();
OK..
val it = There are currently no proofs.: proofs
>
(* Now we have a theorem for use later. *)
leibniz_triplet_lcm;
val it = ⊢ lcm tb ta = lcm tb tc: thm
>
(* ------------------------------------------------------------------------ *)
(* Paths in Leibniz Triangle                                                *)
(* ------------------------------------------------------------------------ *)

(* Define a path type. *)
val _ = type_abbrev ("path", Type `:num list`);
>
(* We can display the Leibniz Triangle by its rows. *)

(* Overload the n-th horizontal row as leibniz_horizontal n *)
EVAL ``leibniz_horizontal 0``;
val it = ⊢ leibniz_horizontal 0 = [1]: thm
> EVAL ``leibniz_horizontal 1``;
val it = ⊢ leibniz_horizontal 1 = [2; 2]: thm
> EVAL ``leibniz_horizontal 2``;
val it = ⊢ leibniz_horizontal 2 = [3; 6; 3]: thm
> EVAL ``leibniz_horizontal 3``;
val it = ⊢ leibniz_horizontal 3 = [4; 12; 12; 4]: thm
> EVAL ``leibniz_horizontal 4``;
val it = ⊢ leibniz_horizontal 4 = [5; 20; 30; 20; 5]: thm
> EVAL ``leibniz_horizontal 5``;
val it = ⊢ leibniz_horizontal 5 = [6; 30; 60; 60; 30; 6]: thm
> EVAL ``leibniz_horizontal 6``;
val it = ⊢ leibniz_horizontal 6 = [7; 42; 105; 140; 105; 42; 7]: thm
> EVAL ``leibniz_horizontal 7``;
val it = ⊢ leibniz_horizontal 7 = [8; 56; 168; 280; 280; 168; 56; 8]: thm
> EVAL ``leibniz_horizontal 8``;
val it = ⊢ leibniz_horizontal 8 = [9; 72; 252; 504; 630; 504; 252; 72; 9]:
   thm
>
(* Overload the leftmost edge as leibniz_up n for a triangle with n rows. *)
EVAL ``leibniz_up 8``;
val it = ⊢ leibniz_up 8 = [9; 8; 7; 6; 5; 4; 3; 2; 1]: thm
> EVAL ``leibniz_up 7``;
val it = ⊢ leibniz_up 7 = [8; 7; 6; 5; 4; 3; 2; 1]: thm
> EVAL ``leibniz_up 6``;
val it = ⊢ leibniz_up 6 = [7; 6; 5; 4; 3; 2; 1]: thm
> EVAL ``leibniz_up 1``;
val it = ⊢ leibniz_up 1 = [2; 1]: thm
> EVAL ``leibniz_up 0``;
val it = ⊢ leibniz_up 0 = [1]: thm
>
(* Basic properties *)
```

```
leibniz_horizontal_0;
val it = ⊢ leibniz_horizontal 0 = [1]: thm
> leibniz_horizontal_len;
val it =
   ⊢ ∀n. LENGTH (leibniz_horizontal n) = n + 1:
   thm
>
leibniz_horizontal_head;
val it =
   ⊢ ∀n. TAKE 1 (leibniz_horizontal (n + 1)) = [n + 2]:
   thm
>
leibniz_up_0;
val it = ⊢ leibniz_up 0 = [1]: thm
> leibniz_up_len;
val it =
   ⊢ ∀n. LENGTH (leibniz_up n) = n + 1:
   thm
>
leibniz_up_cons;
val it =
   ⊢ ∀n. leibniz_up (n + 1) = n + 2::leibniz_up n:
   thm
>

(* ------------------------------------------------------------------------- *)
(* Transform from Vertical LCM to Horizontal LCM using Triplet and Paths     *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* Zigzag Path in Leibniz Triangle                                           *)
(* ------------------------------------------------------------------------- *)

(* Define zig-zag paths, which we have overloaded with an infix 'zigzag'. *)
val _ = overload_on("↝", ``leibniz_zigzag``);
> val _ = set_fixity "↝" (Infix(NONASSOC, 450)); (* same as relation *)
>
(* Zig-zag paths are related by a Leibniz triplet:

                        ta (suffix y)
             (prefix x) tb tc

   triplet = (ta tb tc), with (ta tb) vertical, (tb tc) horizontal.
   path p1 = (prefix x) ++ [tb ta] ++ (suffix y)
   path p2 = (prefix x) ++ [tb tc] ++ (suffix y)
*)
leibniz_zigzag_def;
# # # # # # # val it =
   ⊢ ∀p1 p2.
     p1 ↝ p2 ⇔
     ∃n k x y. (p1 = x ++ [tb; ta] ++ y) ∧ (p2 = x ++ [tb; tc] ++ y):
   thm
>

(* Basic properties *)
leibniz_zigzag_tail;
val it =
   ⊢ ∀p1 p2. p1 ↝ p2 ⇒ ∀x. [x] ++ p1 ↝ [x] ++ p2:
   thm
>
(* This says: adding the same prefix to zig-zag paths gives zig-zag paths, too. *)
list_lcm_zigzag;
val it =
   ⊢ ∀p1 p2. p1 ↝ p2 ⇒ (list_lcm p1 = list_lcm p2):
   thm
>
(* This shows LCM invariance by zig-zag. *)

(* ------------------------------------------------------------------------- *)
(* Wriggle Paths in Leibniz Triangle                                         *)
(* ------------------------------------------------------------------------- *)

(* Introduce wriggle paths, which we shall overload with an infix 'wriggle'. *)

(* Wriggle is the transitive closure of zig-zag. *)
val _ = overload_on("↝*", ``RTC leibniz_zigzag``); (* RTC = reflexive transitive closure *)
> val _ = set_fixity "↝*" (Infix(NONASSOC, 450)); (* same as relation *)
>
(* Basic properties *)
leibniz_zigzag_wriggle;
val it =
   ⊢ ∀p1 p2. p1 ↝ p2 ⇒ p1 ↝* p2:
   thm
>
leibniz_wriggle_tail;
```

```
val it =
   ⊢ ∀p1 p2. p1 ⇝ p2 ⇒ ∀x. [x] ++ p1 ⇝ [x] ++ p2:
   thm
>
leibniz_wriggle_refl;
val it = ⊢ ∀p1. p1 ⇝ p1: thm
>
leibniz_wriggle_trans;
val it =
   ⊢ ∀p1 p2 p3. p1 ⇝ p2 ∧ p2 ⇝ p3 ⇒ p1 ⇝ p3:
   thm
>
list_lcm_wriggle;
val it =
   ⊢ ∀p1 p2. p1 ⇝ p2 ⇒ (list_lcm p1 = list_lcm p2):
   thm
>
(* The last one shows LCM invariance by wriggle. *)

(* ------------------------------------------------------------------------- *)
(* Path Transform keeping LCM                                                *)
(* ------------------------------------------------------------------------- *)

(* First, we establish an intermediate step.
   For successive rows of Leibniz Triangle:

           n-th row: b b b b b
       (n+1)-th row: a c c c c c

   [a] ++ [b b b b b] can wriggle to [a c c c c c]
*)
leibniz_horizontal_wriggle;
# # # # # # val it =
   ⊢ ∀n.
     [leibniz (n + 1) 0] ++ leibniz_horizontal n ⇝
     leibniz_horizontal (n + 1):
   thm
>

(* Then, we use induction to prove the following, making use of the result above.
   For the Leibniz Triangle:

                     b
                     b
                     b
                     b
                     b
       (n+1)-th row: a c c c c c

   [a b b b b b] can wriggle to [a c c c c c]
*)
(* Theorem: (leibniz_up n) wriggle (leibniz_horizontal n) *)
(* Proof:
   By induction on n.
   Base: leibniz_up 0 wriggle leibniz_horizontal 0
      Since leibniz_up 0 = [1]                           by leibniz_up_0
        and leibniz_horizontal 0 = [1]                   by leibniz_horizontal_0
      Hence leibniz_up 0 wriggle leibniz_horizontal 0    by leibniz_wriggle_refl
   Step: leibniz_up n wriggle leibniz_horizontal n ==>
         leibniz_up (SUC n) wriggle leibniz_horizontal (SUC n)
         Let x = leibniz (n + 1) 0.
         Then x = n + 2                                  by leibniz_n_0
          Now leibniz_up (n + 1) = [x] ++ (leibniz_up n)  by leibniz_up_cons
        Since leibniz_up n wriggle leibniz_horizontal n  by induction hypothesis
           so ([x] ++ (leibniz_up n)) wriggle
              ([x] ++ (leibniz_horizontal n))             by leibniz_wriggle_tail
          and ([x] ++ (leibniz_horizontal n)) wriggle
              (leibniz_horizontal (n + 1))                by leibniz_horizontal_wriggle
        Hence leibniz_up (SUC n) wriggle
              leibniz_horizontal (SUC n)                 by leibniz_wriggle_trans, ADD1
*)
(* This is a showcase of my proof style: state the theorem and draft a conventional proof. *)

(* Now check my idea by the theorem-prover. *)
(* First, state the desired goal. *)
(* <<leibniz_up_wriggle_horizontal>> *)
g `!n. (leibniz_up n) wriggle (leibniz_horizontal n)`;
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # val it =
   Proof manager status: 1 proof.
1. Incomplete goalstack:
     Initial goal:


     ∀n. leibniz_up n ⇝ leibniz_horizontal n

:
```

```
   proofs
>
(* By induction on n, this gives two subgoals: *)
e (Induct); (* >> *)
OK..
2 subgoals:
val it =

  leibniz_up n ⟿* leibniz_horizontal n
------------------------------------
leibniz_up (SUC n) ⟿* leibniz_horizontal (SUC n)



leibniz_up 0 ⟿* leibniz_horizontal 0

2 subgoals
:
   proof
>
(* Note that if there are premises, these are listed above a dash-separator,
   and the subgoal is located below the dash-separator. *)

(* Base case: *)
(* Recall these: *)
leibniz_up_0;
# # val it = ⊢ leibniz_up 0 = [1]: thm
> leibniz_horizontal_0;
val it = ⊢ leibniz_horizontal 0 = [1]: thm
> (* so this is simple. *)
e (rw[leibniz_up_0, leibniz_horizontal_0]); (* << *)
OK..

Goal proved.
⊢ leibniz_up 0 ⟿* leibniz_horizontal 0

Remaining subgoals:
val it =

  leibniz_up n ⟿* leibniz_horizontal n
------------------------------------
leibniz_up (SUC n) ⟿* leibniz_horizontal (SUC n)
:
   proof
>
(* The base case is done, next is the step case. *)

(* Step case: *)
(* Introduce an abbreviation to simplify work. *)
e (qabbrev_tac `x = leibniz (n + 1) 0`);
OK..
1 subgoal:
val it =

  0.  leibniz_up n ⟿* leibniz_horizontal n
  1.  Abbrev (x = leibniz (n + 1) 0)
------------------------------------
leibniz_up (SUC n) ⟿* leibniz_horizontal (SUC n)
:
   proof
>
e (`x = n + 2` by rw[leibniz_n_0, Abbr`x`]);
OK..
1 subgoal:
val it =

  0.  leibniz_up n ⟿* leibniz_horizontal n
  1.  Abbrev (x = leibniz (n + 1) 0)
  2.  x = n + 2
------------------------------------
leibniz_up (SUC n) ⟿* leibniz_horizontal (SUC n)
:
   proof
>
(* Put LHS into head and tail form. *)
e (`leibniz_up (n + 1) = [x] ++ (leibniz_up n)` by rw[leibniz_up_cons, Abbr`x`]);
OK..
1 subgoal:
val it =

  0.  leibniz_up n ⟿* leibniz_horizontal n
  1.  Abbrev (x = leibniz (n + 1) 0)
  2.  x = n + 2
  3.  leibniz_up (n + 1) = [x] ++ leibniz_up n
------------------------------------
```

```
leibniz_up (SUC n) ⥲ leibniz_horizontal (SUC n)
:
   proof
>
(* The tail, which is shorter, can be transformed by induction hypothesis (assumption #0). *)
e (`([x] ++ (leibniz_up n)) wriggle ([x] ++ (leibniz_horizontal n))` by rw[leibniz_wriggle_tail]);
OK..
1 subgoal:
val it =

  0.  leibniz_up n ⥲ leibniz_horizontal n
  1.  Abbrev (x = leibniz (n + 1) 0)
  2.  x = n + 2
  3.  leibniz_up (n + 1) = [x] ++ leibniz_up n
  4.  [x] ++ leibniz_up n ⥲ [x] ++ leibniz_horizontal n
------------------------------------
leibniz_up (SUC n) ⥲ leibniz_horizontal (SUC n)
:
   proof
>
(* The head ++ (transformed tail) can be transformed by the previous theorem. *)
e (`([x] ++ (leibniz_horizontal n)) wriggle (leibniz_horizontal (n + 1))` by rw[leibniz_horizontal_wriggle, Abbr`x`]);
OK..
1 subgoal:
val it =

  0.  leibniz_up n ⥲ leibniz_horizontal n
  1.  Abbrev (x = leibniz (n + 1) 0)
  2.  x = n + 2
  3.  leibniz_up (n + 1) = [x] ++ leibniz_up n
  4.  [x] ++ leibniz_up n ⥲ [x] ++ leibniz_horizontal n
  5.  [x] ++ leibniz_horizontal n ⥲ leibniz_horizontal (n + 1)
------------------------------------
leibniz_up (SUC n) ⥲ leibniz_horizontal (SUC n)
:
   proof
>
(* Thus LHS can be transformed to RHS. *)
e (metis_tac[leibniz_wriggle_trans, ADD1]); (* << *)
OK..
metis: r[+0+13]+0+0+0+0+0+0+0+0+0+0+0+8+2+1+1#

Goal proved.
 [......] ⊢ leibniz_up (SUC n) ⥲ leibniz_horizontal (SUC n)

Goal proved.
 [.....] ⊢ leibniz_up (SUC n) ⥲ leibniz_horizontal (SUC n)

Goal proved.
 [....] ⊢ leibniz_up (SUC n) ⥲ leibniz_horizontal (SUC n)

Goal proved.
 [...] ⊢ leibniz_up (SUC n) ⥲ leibniz_horizontal (SUC n)

Goal proved.
 [..] ⊢ leibniz_up (SUC n) ⥲ leibniz_horizontal (SUC n)

Goal proved.
 [.] ⊢ leibniz_up (SUC n) ⥲ leibniz_horizontal (SUC n)
val it =
   Initial goal proved.
⊢ ∀n. leibniz_up n ⥲ leibniz_horizontal n:
   proof
>
(* Name the theorem. *)
val leibniz_up_wriggle_horizontal = save_thm("leibniz_up_wriggle_horizontal", top_thm());
val leibniz_up_wriggle_horizontal =
   ⊢ ∀n. leibniz_up n ⥲ leibniz_horizontal n:
   thm
>
drop();
OK..
val it = There are currently no proofs.: proofs
>
(* This leads directly to the following significant result. *)

(* Theorem: list_lcm (leibniz_up n) = list_lcm (leibniz_horizontal n) *)
(* Proof:
   Since leibniz_up n wriggle leibniz_horizontal n                by leibniz_up_wriggle_horizontal
   Hence list_lcm (leibniz_up n) = list_lcm (leibniz_horizontal n)   by list_lcm_wriggle
*)
(* <<leibniz_lcm_property>> *)
g `!n. list_lcm (leibniz_up n) = list_lcm (leibniz_horizontal n)`;
# # # # val it =
```

```
   Proof manager status: 1 proof.
1. Incomplete goalstack:
     Initial goal:


     ∀n. list_lcm (leibniz_up n) = list_lcm (leibniz_horizontal n)

:
   proofs
>
e (simp[leibniz_up_wriggle_horizontal, list_lcm_wriggle]);
OK..
val it =
   Initial goal proved.
⊢ ∀n. list_lcm (leibniz_up n) = list_lcm (leibniz_horizontal n):
   proof
>
val leibniz_lcm_property = save_thm("leibniz_lcm_property", top_thm());
val leibniz_lcm_property =
   ⊢ ∀n. list_lcm (leibniz_up n) = list_lcm (leibniz_horizontal n):
   thm
>
drop();
OK..
val it = There are currently no proofs.: proofs
> (* This is indeed a milestone theorem! *)

(* ------------------------------------------------------------------------- *)
(* Lower Bound of LCM                                                        *)
(* ------------------------------------------------------------------------- *)

(* Now the main theorem *)

(* Theorem: 2 ** n <= list_lcm [1 .. (n + 1)] *)
(* Proof:
     list_lcm [1 .. (n + 1)]
   = list_lcm (REVERSE [1 .. (n + 1)])  by list_lcm_reverse
   = list_lcm (leibniz_up n)            by notation
   = list_lcm (leibniz_horizontal n)    by leibniz_lcm_property
   = (n + 1) * list_lcm (binomial_horizontal n)                      by leibniz_horizontal_lcm_alt
   = LENGTH (binomial_horizontal n) * list_lcm (binomial_horizontal n)   by binomial_horizontal_len
   >= SUM (binomial_horizontal n)       by list_lcm_lower_bound
   = 2 ** n                             by binomial_horizontal_sum
*)
(* Draft the proof, then check the proof. *)
(* <<lcm_lower_bound>> *)
g `!n. 2 ** n <= list_lcm [1 .. (n + 1)]`;
# # # # # # # # # # val it =
   Proof manager status: 1 proof.
1. Incomplete goalstack:
     Initial goal:


     ∀n. 2 ** n ≤ list_lcm [1 .. n + 1]

:
   proofs
>
e (rpt strip_tac);
OK..
1 subgoal:
val it =

2 ** n ≤ list_lcm [1 .. n + 1]
: proof
>
(* First, perform the LCM transforms. *)
e (`list_lcm [1 .. (n + 1)] = list_lcm (leibniz_up n)` by rw[list_lcm_reverse]);
OK..
1 subgoal:
val it =

  list_lcm [1 .. n + 1] = list_lcm (leibniz_up n)
------------------------------------
2 ** n ≤ list_lcm [1 .. n + 1]
:
   proof
>
e (`_ = list_lcm (leibniz_horizontal n)` by rw[leibniz_lcm_property]);
OK..
1 subgoal:
val it =

  list_lcm [1 .. n + 1] = list_lcm (leibniz_horizontal n)
------------------------------------
2 ** n ≤ list_lcm [1 .. n + 1]
:
```

```
   proof
> 
e (`_ = (n + 1) * list_lcm (binomial_horizontal n)` by rw[leibniz_horizontal_lcm_alt]);
OK..
1 subgoal:
val it =

  list_lcm [1 .. n + 1] = (n + 1) * list_lcm (binomial_horizontal n)
------------------------------------
2 ** n ≤ list_lcm [1 .. n + 1]
:
   proof
>
(* Next, the aim is to apply the following: *)
list_lcm_lower_bound;
val it =
   ⊢ ∀l. EVERY_POSITIVE l ⇒ SUM l ≤ LENGTH l * list_lcm l:
   thm
>
(* So make these assertions. *)
e (`EVERY_POSITIVE (binomial_horizontal n)` by rw[binomial_horizontal_pos]);
OK..
1 subgoal:
val it =

  0.  list_lcm [1 .. n + 1] = (n + 1) * list_lcm (binomial_horizontal n)
  1.  EVERY_POSITIVE (binomial_horizontal n)
------------------------------------
2 ** n ≤ list_lcm [1 .. n + 1]
:
   proof
>
e (`LENGTH (binomial_horizontal n) = n + 1` by rw[binomial_horizontal_len]);
OK..
1 subgoal:
val it =

  0.  list_lcm [1 .. n + 1] = (n + 1) * list_lcm (binomial_horizontal n)
  1.  EVERY_POSITIVE (binomial_horizontal n)
  2.  LENGTH (binomial_horizontal n) = n + 1
------------------------------------
2 ** n ≤ list_lcm [1 .. n + 1]
:
   proof
>
e (`SUM (binomial_horizontal n) = 2 ** n` by rw[binomial_horizontal_sum]);
OK..
1 subgoal:
val it =

  0.  list_lcm [1 .. n + 1] = (n + 1) * list_lcm (binomial_horizontal n)
  1.  EVERY_POSITIVE (binomial_horizontal n)
  2.  LENGTH (binomial_horizontal n) = n + 1
  3.  SUM (binomial_horizontal n) = 2 ** n
------------------------------------
2 ** n ≤ list_lcm [1 .. n + 1]
:
   proof
>
(* Apply the theorem. *)
e (metis_tac[list_lcm_lower_bound]);
OK..
metis: r[+0+7]+0+0+0+0+0+0+1#

Goal proved.
 [....] ⊢ 2 ** n ≤ list_lcm [1 .. n + 1]

Goal proved.
 [...] ⊢ 2 ** n ≤ list_lcm [1 .. n + 1]

Goal proved.
 [..] ⊢ 2 ** n ≤ list_lcm [1 .. n + 1]

Goal proved.
 [.] ⊢ 2 ** n ≤ list_lcm [1 .. n + 1]

Goal proved.
 [.] ⊢ 2 ** n ≤ list_lcm [1 .. n + 1]

Goal proved.
 [.] ⊢ 2 ** n ≤ list_lcm [1 .. n + 1]

Goal proved.
⊢ 2 ** n ≤ list_lcm [1 .. n + 1]
val it =
   Initial goal proved.
⊢ ∀n. 2 ** n ≤ list_lcm [1 .. n + 1]:
```

```
    proof
>
(* Success! *)
val lcm_lower_bound = save_thm("lcm_lower_bound", top_thm());
val lcm_lower_bound =
    ⊢ ∀n. 2 ** n ≤ list_lcm [1 .. n + 1]:
    thm
>
drop();
OK..
val it = There are currently no proofs.: proofs
>
(* Our main theorem is now in the system. *)
lcm_lower_bound;
val it =
    ⊢ ∀n. 2 ** n ≤ list_lcm [1 .. n + 1]:
    thm
>

(* That's how an interactive theorem-prover works! *)
(* Bye! *)

Session Terminated.
- Goodbye.
```

Input

Type your input here.

Info

[load script=[lcmTheorem-demo-long.hol]] wait [0 ms]