1. Ordered lists of numbers (with duplicates):

   (a) Write a function `mergelists[x, y]` which takes two ordered lists `x` and `y`, and makes one ordered list from them, for example,
   
   `mergelists['(2 3 4), '(1 4)] = (1 2 3 4 4).`
   
   Your algorithm should run in time proportional to the number of elements in the two lists.
   
   (b) Using `mergelists`, write a function `sortlist[l]` which takes an unordered list `l` and makes an ordered list of it, for example,
   
   `sortlist['(1 7 3 5 3)] = (1 3 3 5 7).`
   
   For an initial list of `n` elements, your algorithm should run in \(O(n \log n)\) time and not \(O(n^2)\) time.
   
   (c) Write a predicate `dup[l]` which indicates if any atom occurs more than once in an unordered list `l`, for example,
   
   `dup['(1 3 5 3)] = t`
   
   The algorithm should be as efficient as `sortlist`. Make sure you compare numbers with `equal` and not `eq` (`eq` will generally not return `t` for two equal numbers if they are sufficiently large.)

2. Lists of lists of numbers:

   (a) Write a function `countlists[l]` which counts the number of top level lists in a list of lists, for example,
   
   `countlists['((1 2) (1 3) (1 4))] = 3.`
   
   (b) Lexical ordering on lists of numbers is a binary relation defined by:
   
   ```lisp
   lex-lt[x, y] = if null[x] or null[y] then null[x]
                     else if car[x]=car[y] then lex-lt[cdr[x], cdr[y]]
                     else car[x]<car[y]
   ```
   
   Write a function `duplist_of_lists[l]` which returns `t` if any of the lists in a list of lists `l` are identical. `duplist_of_lists` should be as efficient as `sortlist`.
   
   (c) Given a list of numbers, there are several ways to obtain a list of all permutations of these numbers. For example, the set of permutations of `'(1 2 3)` is `'((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1)).` Note that a list of `n` numbers has `n!` permutations. There are no duplications in the list and all sublists have the same number of elements. One strategy, though not very efficient, is to take out each element, say `a`, in turn from the list, permute the rest, and then attach `a` to the front of each permutation. Write a function `permute[l]` to implement this method.
3. S-expressions of numbers:

(a) A cons_tree of a nonempty list x containing no nils, is defined as

\[
\text{cons_tree}[x] = \\
\text{if \ null}[\text{cdr}[x]] \text{ then car}[x] \\
\text{if \ x \ has \ } 2^n \text{ elements then} \\
\quad \text{cons}[	ext{cons_tree}[\text{first } 2^{n-1} \text{ elements of } x]] \\
\quad \text{cons_tree}[\text{second } 2^{n-1} \text{ elements of } x]] \\
\text{else cons_tree}[\text{append}[x, '(\text{nil . nil})]].
\]

For example,
\[
\text{cons_tree}['(1 2 3 4 5)] \\
= ((((1 . 2) . (3 . 4)) . ((5 . nil) . (nil . nil))).
\]

Write a function \(\text{make\_cons\_tree}[l]\) that sorts an unordered list of numbers \(l\) and then returns the corresponding cons_tree, for example,
\[
\text{make\_cons\_tree}['(3 2 1)] = ((1 . 2) . (3 . nil)) \\
= ((1 . 2) 3)
\]

(b) The natural way to write \(\text{make\_cons\_tree}\) is to follow the definition closely. This can be rather inefficient due to the use of operations that repeatedly scan the list in a top-down manner in order to construct lists whose lengths are powers of two. Write a function \(\text{squeeze}[l]\) that returns the cons_tree of a list \(l\), but doesn’t use any operations that compute the length of a list. Thus you will do it in a bottom-up manner. Assume that the list is already sorted. [Hint: your function should take time \(O(n)\).]

(c) Write a predicate \(\text{cons\_treep}[s]\) that determines whether or not an arbitrary s-expression \(s\) is a cons_tree. For example,
\[
\text{cons\_treep}['(\text{nil . 3})] = \text{cons\_treep}['(3 . \text{nil})] = \text{nil}.
\]
The first one is nil because all occurrences of nil must be at the end, while the second is nil because the true cons_tree for a set consisting of just one atom is the atom itself.

(d) Write a function \(\text{bin}[n]\) that returns the list of 1’s and 0’s that correspond to the binary representation of an integer \(n\), for example,
\[
\text{bin}[6] = (1 1 0).
\]

(e) Write a function \(\text{kth\_least}[x, k]\) which takes a cons_tree \(x\) and an integer \(k\) as arguments and returns the \(k^{th}\) least element of \(x\) (nil if no such element exists). For example,
\[
\text{kth\_least}['((1 . 2) . (5 . \text{nil}))', 3] = 5.
\]
[Hint: look at the binary representation of \(k-1\).]