1. (a) The subsets function we defined in class used functions extend and distrib to compute all possible subsets of a list of distinct values. However, the implementation we used does not order the subsets in any particular way. Rewrite subsets, extend and distrib so that subsets are generated in order of increasing size. That is, the empty subset must come first, followed by all subsets of size one, then all subsets of size two, etc. Any ordering among subsets of the same size is OK, but a smaller subset must always precede a larger one. For example (subsets ' (1 2)) may produce (() (1) (2) (1 2)) or (() (2) (1) (1 2)) but not (() (1 2) (2) (1)).

(b) The subsets function is sometimes used to enumerate all the possible selections from a list of distinct values. Since the number of subsets possible can be very large, a predicate is often used to filter out unwanted subsets. If the predicate is true, the subset is kept; if it is false the subset is discarded. For example, a filter predicate might be used to discard subsets that are too large or too expensive.

If we use a filter predicate after all subsets are generated, we may waste a great deal of time and space generating subsets we don’t really want. An alternative is to filter subsets as they are generated. Assume we limit our attention to filter predicates that are monotone. This means that if fp is false for set S, it will also be false for any larger set that contains S. Filter functions that limit a set to a given size or cost are monotone.

Write a Scheme function (filtered-subsets L fp) that generates all subsets of list L for which fp is true. This function should filter subsets as they are generated to avoid generation of large numbers of unwanted subsets. Test your solution on

(filtered-subsets
  ' (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)
  (lambda (S) (<= (reduce + S 0) 8)))

This call produces all subsets of the first 20 integers that sum to 8 or less (there are 25 such subsets, including the empty subset). This function should execute in just a few seconds, even though there are $2^{20}$ subsets that might be considered.
2. You are to write a Scheme function `make-queue` that generates a function that implements a queue data structure. Recall that a queue operates like a “waiting line.” New entries are added to the rear end of the queue, and entries are removed from the front of the queue.

The call `(make-queue)` should return a function that represents an initially empty queue. If `q` is a queue function, it should accept the following calls:

(q enter! val₁ val₂ ...)  
q adds `val₁, val₂, ...` to the end of the queue it represents, with `val₁` being the last value entered (and thus at the very end of the queue). The list `(val₁ val₂ ...)` is returned.

(q remove! cnt)  
cnt is a non-negative integer. `cnt` values are removed from the queue `q` represents. A list containing the values removed is returned (with the first value removed at the right end of the list). If the queue does not contain `cnt` values, then `#f` is returned (as an error indication).

(q contents)  
Returns a list representing the contents of `q`. The last element of the queue appears at the left end of the list; the first element of the queue appears at the right end of the list.

(q clone)  
Returns a new queue function whose contents are (initially) identical to the contents of `q`.

The following calls illustrate how queue functions are expected to behave:

```
(define my-queue (make-queue))
(my-queue 'enter! 4 5 6) ⇒ (4 5 6)
(my-queue 'enter! 1 2 3) ⇒ (1 2 3)
(my-queue 'contents) ⇒ (1 2 3 4 5 6)
(my-queue 'remove! 2) ⇒ (5 6)
(my-queue 'contents) ⇒ (1 2 3 4)
(define your-queue (my-queue 'clone))
(your-queue 'contents) ⇒ (1 2 3 4)
(my-queue 'remove! 1) ⇒ (4)
(your-queue 'contents) ⇒ (1 2 3)
(your-queue 'contents) ⇒ (1 2 3 4)
```

3. (a) In Scheme sets can be represented as lists. However, unlike lists, the order of values in a set is not significant. Thus both `(1 2 3)` and `(3 2 1)` represent the same set. Moreover, the same value may not appear more than once within a list that represents a set. Thus `(1 2 3)` is a valid set, but `(1 2 1)` is not.

Write a Scheme function `(valid-set? S)` that tests whether set `S` is valid. A set is valid if no value in the list appears more than once. In this part you may assume that sets contain only atomic values (numbers, strings, symbols, etc.). For example
(valid-set? '(1 2 3)) ⇒ #t  
(valid-set? '(1 2 1)) ⇒ #f  
(valid-set? '(1 1.0 "one")])) ⇒ #t  

(b) Write a Scheme function (equal-sets? S1 S2) that tests whether sets S1 and S2 (represented as lists) are equal. Two sets are equal if they contain exactly the same members, ignoring ordering. In this part you may assume that sets contain only atomic values (numbers, strings, symbols, etc.). If either S1 or S2 is invalid, return #f (as an error indication). For example

(equal-sets? '(1 2 3) '(3 2 1)) ⇒ #t  
(equal-sets? '(1 2) '(3 2 1)) ⇒ #f  
(equal-sets? '(curly larry moe) '(moe larry curly)) ⇒ #t  
(equal-sets? '(1 1) '(1 1)) ⇒ #f

(c) Two common operations on sets are union and intersection. The union of two sets is the set of all elements that appear in either set (with no repetitions). The intersection of two sets is the set of elements that appear in both sets.

Write Scheme functions (union S1 S2) and (intersect S1 S2) that implement set union and set intersection. You may again assume that set elements are always atomic. If either S1 or S2 is invalid, return #f (as an error indication). For example

(union '(1 2 3) '(3 2 1)) ⇒ (1 2 3)  
(union '(1 2 3) '(3 4 5)) ⇒ (1 2 3 4 5)  
(union '(a b c) '(3 2 1)) ⇒ (a b c 1 2 3)  
(union '(1 1) '(2 2)) ⇒ #f  
(intersect '(1 2 3) '(3 2 1)) ⇒ (1 2 3)  
(intersect '(1 2 3) '(4 5 6)) ⇒ ()  
(intersect '(1 2 3) '(2 3 4 5 6)) ⇒ (2 3)  
(intersect '(1 1) '(2 2)) ⇒ #f

(d) In general sets can contain other sets. Extend your solution to parts (a), (b) and (c) to allow sets to contain other sets. For example,

(valid-set? '(1 (1 2))) ⇒ #t  
(valid-set? '((1) 2 (1))) ⇒ #f  
(equal-sets? '(1 (2 3)) '(((3 2) 1))) ⇒ #t  
(equal-sets? '(1 2 3) '(((3 2) 1))) ⇒ #f  
(equal-sets? '(1 2 3) '(((1) 2 3)) ⇒ #f  
(union '(((1 2 3)) '(((3 2 1)))) ⇒ (1 2 3 (2))  
(union '(((1 2 3)) '(((3 4 5)))) ⇒ ((1 2 3) (3 4 5))  
(union '(((1 2 3)) '(((3 2 1)))) ⇒ ((1 2 3))  
(intersect '(((1 2 3)) '(((3 2 1)))) ⇒ ((1 2 3))  
(intersect '(((1 2 3)) '(((4 5 6)))) ⇒ ()  
(intersect '(((1) 2 (3)) '(((2) (3) (4)))) ⇒ ((2) (3))

-3-
4. (a) Write a pair of Scheme functions, \((\text{gen-list start stop step})\) and \((\text{pair-prod? list val})\). The function \(\text{gen-list}\) will generate a list of integers, from \(\text{start}\) to \(\text{stop}\), with consecutive values incremented by \(\text{step}\). (If \(\text{start} > \text{stop}\) then the empty list is generated). For example \((\text{gen-list 1 92}) \Rightarrow (1 \ 3 \ 5 \ 7 \ 9)\).

The predicate \(\text{pair-prod?}\) tests whether any two adjacent values in \(\text{list}\) have a product equal to \(\text{val}\). For example, \((\text{pair-prod? } ' (1 \ 2 \ 3) \ 6) \Rightarrow \#t\) since \(2 \times 3 = 6\). Similarly, \((\text{pair-prod? } (\text{gen-list 1 100 1}) \ 10) \Rightarrow \#f\) since no two adjacent integers in the range \(1\) to \(100\) have a product of \(10\).

(b) A problem with a function like \(\text{pair-prod?}\) is the fact that its \(\text{list}\) parameter must be fully computed in advance, even if all the values on the list are not needed. For example, \((\text{pair-prod? } (\text{gen-list 1 100000 1}) \ 2)\) will spend a lot of time and space building up a list of \(100000\) values, even though only the first two are needed!

An alternative to completely building a complex data structure is **lazy evaluation**. That is, part of the structure is built along with a **suspension**—a function that will supply a few more values upon request. The following two Scheme functions produce a sequence of integers in a lazy manner.

\[
\begin{align*}
\text{(define (return-one-val start stop step)} \ni (\text{if (> (+ start step) stop)} \ni (\text{cons start #f}) \ni (\text{cons start}
\ni \quad (\text{lambda () (return-one-val (+ start step) stop step))))
\ni )
\end{align*}
\]

\[
\begin{align*}
\text{(define (int-seq start stop step)} \ni (\text{if (> start stop)} \ni (\text{cons #f #f}) \ni (\text{return-one-val start stop step}))
\ni )
\end{align*}
\]

When called, \(\text{int-seq}\) returns a pair of values. The \textit{car} is the first integer in the sequence, or \#\(f\) if the sequence is empty. The \textit{cdr} is a function that, when called, will return another pair. That pair consists of the next integer in the sequence plus another suspension function. When the end of the sequence is reached, the \textit{cdr} of the pair is \#\(f\) (rather than a function), indicating that no more values can be produced.

Create a new version of \(\text{pair-prod?}\) called \(\text{pair-prod-seq?}\) that takes a lazy integer sequence (as defined by \(\text{int-seq}\)) rather than a list as a parameter. You can use the Scheme function \(\text{(time (f args))}\) to time the evaluation of \((f \ \text{args})\). Compare the execution times of \((\text{pair-prod? } (\text{gen-list 1 100000 1}) \ 90)\) and \((\text{pair-prod-seq? } (\text{int-seq 1 100000 1}) \ 90)\). Which is faster? Why?

Now compare the execution times of \((\text{pair-prod? } (\text{gen-list 1 100000 1}) \ 91)\) and \((\text{pair-prod-seq? } (\text{int-seq 1 100000 1}) \ 91)\). Again, which is faster, and why?
5. (a) Assume we have a list \( L \) of integers. Define a function \( \text{listify} \ L \) that divides \( L \) into one or more sublists so that each sublist contains integers in non-decreasing (sorted) order. That is, if \( v_1 \) and \( v_2 \) are adjacent in \( L \) and \( v_1 \leq v_2 \) then \( v_1 \) and \( v_2 \) are adjacent in the same sublist. However if \( v_1 > v_2 \) then \( v_2 \) ends one sublist and \( v_2 \) begins the next sublist. For example,

- \( \text{listify} \ (3 \ 5 \ 1 \ 8 \ 9 \ 2 \ 1 \ 0) \Rightarrow ((3 \ 5) \ (1 \ 8 \ 9) \ (2) \ (1) \ (0)) \)
- \( \text{listify} \ (1 \ 2 \ 3 \ 4 \ 5 \ 6) \Rightarrow ((1 \ 2 \ 3 \ 4 \ 5 \ 6)) \)
- \( \text{listify} \ (5 \ 4 \ 3 \ 2 \ 1) \Rightarrow ((5) \ (4) \ (3) \ (2) \ (1)) \)

(b) If the output of \text{listify} \ contains a single sublist, then we know the input list \( L \) was in sorted order. This makes it easy to test for duplicates within \( L \)—we just compare adjacent values. If \text{listify} \ returns more than one sublist, we can merge the first two lists into one sorted list, and repeat the process until we have one sorted list. Then testing for duplicates is easy.

For example, if \text{listify} \ initially produces

- \( ((3 \ 5) \ (1 \ 8 \ 9) \ (2) \ (1) \ (0)) \)
- \( ((1 \ 2 \ 3 \ 5 \ 8 \ 9) \ (2) \ (1) \ (0)) \)
- \( ((1 \ 1 \ 2 \ 3 \ 5 \ 8 \ 9) \ (0)) \)

Implement \( \text{duplicates?} \ L \), which sorts the values in \( L \) using \text{listify} \ as a subroutine and tests for duplicates.

(c) Our algorithm for testing for duplicates is somewhat inefficient. If a duplicate appears in \( L \) it can readily be detected when \text{listify} \ initially partitions \( L \) or when the sublists produced by \text{listify} \ are merged.

Create a function \( \text{duplicates-cc?} \) (and whatever auxiliary functions you require) that make duplicate checking integral to the implementation of \text{listify} \ and the merge component of the sort. As soon as a duplicate is seen, \#t should be immediately returned, without any further processing of \( L \). You should use \text{call-with-current-continuation} to implement the exception mechanism you will need to return when you see a duplicate value.