Section Handout: All Things Scheme

Problem 1: Building Subsets of a Certain Size

Implement the **k-subsets** function, which accepts a **set** and a non-negative integer **k** and constructs a list of all those subsets of the incoming set whose size just happens to equal **k**. Your implementation should run in time that's proportional to the number of subsets in the final answer. In particular, you should not reuse the **power-set** implementation from lecture and then filter on length, because that's entirely too time-consuming when large sets are paired with small values of **k**.

```
;; Function: k-subsets
;; -------------------
;; k-subsets constructs a list of all those subsets
;; of the specified set whose size just happens to equal k.
;;
;; Examples: (k-subsets '(1 2 3 4) 2) -> ((1 2) (1 3) (1 4) (2 3) (2 4) (3 4))
;;           (k-subsets '(1 2 3 4 5 6) 1) -> ((1) (2) (3) (4) (5) (6))
;;           (k-subsets '(a b c d) 0) -> (())
;;           (k-subsets '(a b d d) 5) -> ()

(define (k-subsets set k)
   (
```

Problem 2: Up Down Permutations

a. An up-down list is a homogeneous list which alternates between local minima and local maxima—that is, the second element is larger than the first and third, the third element is smaller than the second and fourth, the fourth element is larger than the third and fifth, and so on. Informally, the list zig-zags up and down as you march over all of its elements.

   Using **car-cdr** recursion, implement the **is-up-down** routine, which returns **#t** if and only if the specified list of atoms is an up-down list according to the specified predicate. List of length 0 and 1 are automatically considered to be up-down lists.

```
;; Function: is-up-down?
;; ---------------------
;; Returns true if and only if the specified list is an up-down list
;; according to the specified predicate.
;;
;; Examples: (is-up-down? '() <) -> #t
;;           (is-up-down? '(1) <) -> #t
;;           (is-up-down? '(1 2 2 3) <) -> #f
;;           (is-up-down? '(1 6 2 4 3 5) <) -> #f
;;           (is-up-down? '(1 6 2 4 3 5) >) -> #f  ;; down-up, but not up-down
;;           (is-up-down? '(4 8 3 5 1 7 6 2) <) -> #f

(define (is-up-down? list comp)
   (
```

b. Implement the **up-down-permute** function, which takes of list of unique integers and generates all of permutations that are up-down permutations. Your implementation should take time that is proportional to the length of the answer—in particular, you may **not** generate all of the permutations and then filter out those that fail the predicate from part a. You may assume that the list being permuted is comprised of integers, and that each integer appears exactly once. You needn't generalize your implementation for various comparison functions; just assume that less than (<) and greater than (>) can be hard-coded into your implementation. This is difficult, but quite doable if you define **down-up-permute** as well and implement each in terms of the other. Feel free to make use of the **remove** function from lecture, which returns a list identical to the one specified, except that all instances of a specified element have been omitted. You should use mapping instead of exposed **car-cdr** recursion.

```
;; Function: up-down-permute
;; --------------------------
;; up-down-permute generates all those permutations of a list that
;; just happen to be up-down permutations.
;;
;; Examples: (remove 3 '(1 2 3 4 5 4 3 2 1)) -> (1 2 4 5 4 2 1)
;;           (up-down-permute '()) -> (())
;;           (up-down-permute '(1)) -> ((1))
;;           (up-down-permute '(1 2)) -> ((1 2))
;;           (up-down-permute '(1 2 3)) -> ((1 3 2) (2 3 1))
;;           (up-down-permute '(1 2 3 4 5)) ->
;;                ((1 3 2 5 4) (1 4 2 5 3) (1 4 3 5 2) (1 5 2 4 3) (1 5 3 4 2)
;;                 (2 3 1 5 4) (2 4 1 5 3) (2 4 3 5 1) (2 5 1 4 3) (2 5 3 4 1)
;;                 (3 4 1 5 2) (3 4 2 5 1) (3 5 1 4 2) (3 5 2 4 1)
;;                 (4 5 1 3 2) (4 5 2 3 1))

(define (up-down-permute num-list)
   (
```