A. Recall how merge sort works: The algorithm takes as input a list of numbers to be sorted. If the length of the list is 0 or 1, then the list is already sorted. Otherwise, the list is split into two lists of (roughly) equal length. Each of these sublists is sorted using a recursive call to merge sort. Then the two sorted sublists are merged, as follows. We create an empty result list. We inspect the first item in each sublist; we pick the lesser one (breaking ties arbitrarily), remove it from its sublist, and append it to our result list. We repeat this process until one or the other sublist is empty. If the other sublist is not empty, we append its contents to the result list. Now the sublists are merged; we output the result list. Here is a Python implementation:

```
def mergeSort(l):
    if len(l) <= 1:
        # The list is so short that it is already sorted.
        return l
    else:
        # Sort the front and back halves of the list.
        middle = len(l) / 2
        front = mergeSort(l[:middle])
        back = mergeSort(l[middle:])
        # Merge the two halves into a single sorted list.
        result = []
        i = 0
        j = 0
        while i < len(front) and j < len(back):
            if front[i] < back[j]:
                result.append(front[i])
                i += 1
            else:
                result.append(back[j])
                j += 1
        # Include any remaining elements from the lists.
        while i < len(front):
            result.append(front[i])
            i += 1
        while j < len(back):
            result.append(back[j])
            j += 1
        return result
```

Pages 413-414 of the DLN textbook prove that the quick sort algorithm is correct. Mimic that proof to prove that merge sort is correct.

B. Define a *fully parenthesized arithmetic expression (FPAE)* to be either a numeral or an expression of the form $(a + b)$, $(a - b)$, $(a * b)$, or (a/b), where a and b are FPAEs. For example, $(3 * ((4/5) + 1))$ is an

FPAE, and $(17/2 + 4)$ is not. Also, $(13/0)$ is an FPAE; we aren't worried about division by 0. FPAEs are a simple example of a *context-free language*, about which you'll learn more in CS 254 and CS 251.

1. Prove that for any FPAE the number of left parentheses equals the number of right parentheses.
2. Prove that for any FPAE, at any point in the FPAE the number of left parentheses before that point is greater than or equal to the number of right parentheses before that point.

C. For any set X of n objects and any k such that $0 \leq k \leq n$, a *k-combination* is a subset of X with exactly k objects. For example, if $X = \{a, b, c, d\}$ and $k = 3$ then $\{a, c, d\}$ is a 3-combination from X. By the way, the number of k-combinations from a set of n objects is denoted $\binom{n}{k}$, and it turns out that $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. (You don't need to prove this.)

In the language of your choice, write a function `comb` that takes in a set of size n and an integer k $(0 \leq k \leq n)$, and returns the set of all k-combinations from the given set. Depending on the language, you may represent sets as lists, arrays, vectors, etc. I did this exercise in Python and represented sets using Python lists. An example transcript follows. To give you an idea of the work involved, my implementation is 8 lines long (excluding comments) and could easily be shortened to 5 lines.

```
>>> comb(['a', 'b', 'c', 'd'], 0)
[[]]
>>> comb(['a', 'b', 'c', 'd'], 1)
[['d'], ['c'], ['b'], ['a']]
>>> comb(['a', 'b', 'c', 'd'], 2)
[['c', 'd'], ['b', 'd'], ['b', 'c'], ['a', 'd'], ['a', 'c'], ['a', 'b']]
>>> comb(['a', 'b', 'c', 'd'], 3)
[['b', 'c', 'd'], ['a', 'c', 'd'], ['a', 'b', 'd'], ['a', 'b', 'c']]
>>> comb(['a', 'b', 'c', 'd'], 4)
[['a', 'b', 'c', 'd']]
```

Print your function `comb`, along with a transcript that shows it working on lists of at least two different lengths.