CS61A Notes – Week 9: Recursion and tree recursion (solutions)

1. Write a procedure `expt(base, power)`, which implements the exponent function. For example, `expt(3, 2)` returns 9, and `expt(2, 3)` returns 8. Use recursion!

   We know that $a^0 = 1$ for all a != 0, so we can use this fact for our base case. From there, the recursive call just relies on the fact that $a^b = a*a^{b-1}$.

   ```
   def expt(base, power):
       if power == 0:
           return 1
       else:
           return n * expt(base, power – 1)
   ```

2. Write a procedure `merge(s1, s2)` which takes two sorted (smallest value first) lists and returns a single list with all of the elements of the two lists, in ascending order. (*Hint*: if you can figure out which list has the smallest element out of both, then we know that the resulting merged list will be that smallest element, followed by the merge of the two lists with the smallest item removed. Don't forget to handle the case where one list is empty!). Use recursion.

   Since the two sequences we get as arguments are already sorted, we know the smallest element out of both sequences must lie at the front of one of the two. Thus, at each step, we take the smaller of the two first elements, put it at the front of our result, and make a recursive call.

   We stop when either of the sequences becomes empty, at which point we can simply return the other, non-empty sequence. (This also works if both sequences are empty; can you see why?)

   ```
   def merge(s1, s2):
       if not s1:  # s1 is empty
           return s2
       elif not s2:  # s2 is empty
           return s1
       elif s1[0] < s2[0]:
           return [s1[0]] + merge(s1[1:], s2)
       else:
           return [s2[0]] + merge(s1, s2[1:])
   ```

Multiple recursive calls at a time: Tree recursion

1. I want to go up a flight of stairs that has `n` steps. I can either take 1 or 2 steps each time. How many different ways can I go up this flight of stairs? Write a procedure `count_stair_ways` that solves this problem for me.

   ```
   This problem demonstrates the power of recursion. We solve this problem
   by brute force; we simply try every single method of climbing up the
   staircase and see how many of them bring us to the top exactly!

   Our first base case is n == 0. If this is the case, then we know we
   have reached the top of the staircase exactly, which means there is 1
   way to climb the staircase. If n < 0, then we have overshot the top of
   the staircase, which doesn't count as a way (thus we return 0).

   Now the recursive call simply tries to take one step (recursive call
   with argument n – 1), then tries to take two steps (recursive call with
   argument n – 2), and sums the two results together to get our final
   result.

       def count_stair_ways(n):
           if n == 0:
               return 1
           elif n < 0:
               return 0
           else:
               return count_stair_ways(n – 1) + count_stair_ways(n – 2)
   ```

2. Consider the `subset_sum` problem: you are given a list of integers and a number k. Is there a subset of the list that adds up to k? For example:

   ```
   This is actually a very famous problem. Our approach to this problem is
   going to be to generate every possible subset and see if any of them
   ever sum up to k.

   Let’s talk about base cases. If I asked you if whether there is a
   subset of a list that sums to 0, the answer is always yes – all I have
   to do is take an empty subset, which then sums to 0! This is one of our
   base cases.

   What if I asked you if an empty list could sum to some k != 0? The
   answer is clearly no, since the only subset of an empty list is the
   empty list, and we already know that sums to 0. This is our second base
   case.

   Note that the ordering of our base cases in the following solution is
   important, since subset_sum([], 0) should return True.

   Now we consider the recursive case, looking only at the first item in
   the list. We can either choose to include this in the sum, which would
   reduce k by lst[0] in the recursive call, or we leave it out of the
   sum, which would leave k the same. In both cases, we no longer need to
   look at the first element, so we can use lst[1:] as our new list in the
   recursive call. If either of these recursive calls returns true, then
   we know there is a subset that sums to the value, which is why we
   combine using or.

   A lot of people asked if you could use k <= 0 as a base case. My answer
   to this is no, since your list might contain negative integers too!
   (Consider subset_sum([8, -2], 6).)
   ```

   ```
   >>> subset_sum([2, 4, 7, 3], 5)  # 2 + 3 = 5
   True
   >>> subset_sum([1, 9, 5, 7, 3], 2)
   False

   def subset_sum(lst, k):
       if k == 0:
           return True
       elif not lst:  # lst is empty
           return False
       else:
           return subset_sum(lst[1:], k - ls[0} \
                   or subset_sum(lst[1:], k)
   ```

3. We will now write one of the faster sorting algorithms commonly used, named Mergesort. Merge sort works like this:
   a. If there's only one (or zero) item(s) in the sequence, it's already sorted!
   b. If there's more than one item, then we can split the sequence in half, sort each half recursively, then `merge` the results (using the `merge` procedure from earlier in the notes). The result will be a sorted sequence.

   Using the described algorithm, write a function `mergesort(s)` that takes an unsorted sequence `s` and sorts it.

   The code here basically follows the directions, word for word.

```
    def mergesort(s):
        if len(s) <= 1:
            return s
        else:
            mid = len(s) // 2
            return merge(mergesort(s[:mid]), mergesort(s[mid:]))
```

4. Pascal's triangle is a useful recursive definition that tells us the coefficients in the expansion of the polynomial *$(x + a)^n$*. Each element in the triangle has a coordinate, given by the row it is on and its position in the row (which you could call a column). Every number in Pascal's triangle is defined as the sum of the item above it and the item above it and to the left (its position in the row, minus one). If there is a position that does not have an entry, we treat it as if we had a 0 there. Below are the first few rows of this triangle:

```
Item:    0   1   2   3   4   5  ...
Row 0:   1
Row 1:   1   1
Row 2:   1   2   1
Row 3:   1   3   3   1
Row 4:   1   4   6   4   1
Row 5:   1   5   10  10  5   1
...
```

Define the procedure `pascal(row, column)` which takes a row and a column, and finds the value at that position in the triangle. Don't use the closed-form solution, even if you know it!

```
There are a number of base cases we could've used for this problem;
here's one example. The base cases here basically say "if you go out of
bounds, return 0, and if you're in the upper left corner, return 1."
That alone is good enough to make the full program work!

    def pascal(row, column):
        if row < 0 and column < 0:
            return 0
        elif row == 0 and column == 0:
            return 1
        else:
            return pascal(row – 1, column) \
                    + pascal(row – 1, column – 1)
```