Problem 1

1. Describe a backtracking algorithm that decides whether an input string matches a parsed regular expression.

   To begin, define some helper functions to make the operation more smooth and the algorithm more readable. *Note: Algorithms use python notation.*

   To begin with, assume that the parsed python regex uses square brackets instead of parenthesis so that we can treat it as a list and can recursively call the lists as "sub-regex's". The first function we define is the getOps() function which takes a regex and returns the operation (either +, ., or *) and the one or two operands (which may be alphabet characters or other regex's).

```
# getOps
# returns an oprator and two operands
# example:
# r1 = ['+', ['A'], ['B']]
# reg(r1) returns the tuple:
# ('+', ['A'], ['B'])
def getOps(reg):
    # grab the operator
    operation = reg[0]
    if (len(reg) == 1) and (reg[0] in alphabet):
        return ('const', reg[0], None)
    elif operation == '+':
        return (operation, reg[1], reg[2])
    elif operation == '.':
        return (operation, reg[1], reg[2])
    elif operation == '*':
        return (operation, reg[1], None)
```

The next function to define is a helper function that will help with the kleene-star operation. Given an index number and a string, it will slice up the string into 1, 2, or more substrings, with the number and length of substrings determined by the index function. For a string of length n, there are n-1 cuts we can make (in between characters) to slice up the string, so to do this we take the index, convert it to binary, and use the location of the 1's to decide where to slice the string.

```
# splitString breaks up a string
# into smaller sub-strings given
# an index from 0 to 2^(n - 1).
# i.e. it maps an index to all
# possible ways to split up a string
# and then returns a list of the
# string s broken up in the way
# matched to index
def splitString(idx, s):
    slices = [0]
    # convert index to binary
    idx = bin(idx)
    idx = "".join(reversed(idx))
    # iterate over binary string
    for i in range(len(idx) - 2):
        # cut the string where the
        # binary number has a 1
        if idx[i] == '1':
            slices.append(i + 1)
    slices.append(len(s))
    ret = []
    #Now actually cut up the string
    for i in range(len(slices)-1):
        ret.append(s[slices[i]:slices[i+1]])
    return ret
```

Finally, making the backtracking regex algorithm we start by getting the operation and operands. Depending on the operation, the way in which we recursively split changes. See comments for more information.

```
# Check whether or not the
# given string s matches the
# regex reg
def parseRegex(s, reg):
    # get the operation
    ops = getOps(reg)
    # check if we're looking at a single
    # constant
    if ops[0] == 'const':
        if s == ops[1]:
            return True
        else:
            return False
    # check for + action
    elif ops[0] == '+':
        return parseRegex(s, ops[1]) or parseRegex(s, ops[2])
    # check for concatenation
    elif ops[0] == '.':
        # Check all possible combos of concatenation
        for i in range(0, len(s)):
            end = len(s)
            t = parseRegex(s[0:i], ops[1]) and parseRegex(s[i:end], ops[2])
            if t == True:
                return True
        return False
    # Check kleene star
    elif ops[0] == '*':
        # check empty string
        if s == "":
            return True
        # check all possible combos of ways to slice
        # up the string and see if it matches
        for i in range(0, 2**(len(s) - 1) - 1):
            t = True
            for sub in splitString(i, s):
                t = t and parseRegex(sub, ops[1])
            if t == True:
                return True
        return False
    return False
```

2. Calculate the runtime of your algorithm in terms of the length of the input n and the length of the regular expression m. Justify your answer.

*Assume that m is the number of operations in the regular expression, with operations meaning ., +, or *.*

Notice that the worst-case time complexity operation is the kleene-star operation. For a string of length n, it may require 2^{n-1} recursive calls. However, one issue is that this bound is only reached whenever each individual character is checked. If the kleene-star operation is performed on subexpressions that produce strings greater than length one, this bound will not be reached.

However, this bound will be reached whenever the kleene-star operator is performed on sub-expressions that produce a string of length one. In other words, the worst case is a kleene-star that requires 2^{n-1} recursions followed by m sub-expressions. This leads to the follow recursion:

$$T(n) = 2^{n-1} + \sum_{i=1}^{m} T(1)$$

Looking at the + algorithm, T(1) may require up to 2 recursive calls and the . operation may have T(1) call up to 1 recursive call. Therefore, we have the following bound:

$$T(n) \in O(2^{n-1} + 2(m-1))$$

The (m - 1) term comes from the operation used on the initial kleene-star algorithm.

3. For the fixed regular expression example above, calculate the runtime complexity of your algorithm in terms of the length of the input n.

The runtime complexity is still bounded by the problem 1 part 2 (aka b). Therefore, the runtime complexity is bounded again by:

$$O(2^{n-1} + 2(m-1))$$

skipping problems 2 and 3 because they're too hard and I have a job interview I need to focus on