P0 (15pt): Give an algorithm to print out all possible subsets of an n-element set, such as $\{1, 2, 3 \ldots, n\}$.

As sets that have their order rearranged but with the same elements are equal, we only have to print out all combinations of the n - element set, and not every permutation. To print out each combination, I employ the following algorithm.

```
#recursive function that takes in a set and returns all possible subsets
#this recursive function always returns a list of lists
def subsets(set):
    #if subset is empty, return itself as the only subset of this set
    #as an empty set contains no subset
    if set == []:
        return [set]
    #set x equal to all subsets of the rest of the set after index 0
    x = subsets(set[1:])
    # all possible subsets after 0 + (all possible subsets after 0 + 0 for
each subset) = all possible subsets
    return x + [[set[0]] + y for y in x]
```

So for example

```
S = [1,2,3,4]
print(subsets(S))
```

Will print:

[[], [4], [3], [3, 4], [2], [2, 4], [2, 3], [2, 3, 4], [1], [1, 4], [1, 3], [1, 3, 4], [1, 2], [1, 2, 4], [1, 2, 3], [1, 2, 3, 4]]

Which is each subset recursively generated. This algorithm has a time complexity of O(n^2). As there exists only N x N possible subsets.

P1 (15pt): A *derangement* is a permutation p of $\{1, \ldots, n\}$ such that no item is in its proper position; i.e. $p_i \neq i$ for all $1 \leqslant i \leqslant n$. For example, $p = \{3, 1, 2\}$ is a derangement of $\{1,2,3\}$, whereas $p = \{3, 2, 1\}$ is not.

Write an efficient backtracking program that prints out all derangements of $\{1, \ldots, n\}$.

```
# Python function to return permutations of a given list
def permutations(set):
    # If set is empty then there are no permutations
    if len(set) == 0:
        return []

    # If there is only one element in set then, only
    # one permutation is possible
    if len(set) == 1:
        return [set]

    # Find the permutations for set if there are
    # more than 1 characters

     l = [] # empty list that will store current permutation
    # Iterate the input(set) and calculate the permutation
    for i in range(len(set)):
        m = set[i]

       # Extract set[i] or m from the list.  remset is
       # remaining list
        remset = set[:i] + set[i+1:]

       # Generating all permutations where m is first
       # element
        for p in permutations(remset):
            l.append([m] + p)
    return l

def derangements(set):
    #return all deranged permutations
    return list(perm for perm in permutations(set) if all(set[indx] !=
perm[indx] for indx in range(len(set))))
```

I break up the algorithm into two functions, one generates all permutations of the set, and the other returns only the deranged permutations

The time complexity of this algorithm is O(n!) as there are always n! Permutations of a set with n being the size of the set.

Proof:

```
set = [1, 2 ,3, 4]
print(derangements(set))
```

Prints:
[[2, 1, 4, 3], [2, 3, 4, 1], [2, 4, 1, 3], [3, 1, 4, 2], [3, 4, 1, 2], [3, 4, 2, 1], [4, 1, 2, 3], [4, 3, 1, 2], [4, 3, 2, 1]]

P2 (20pt): Describe recursive algorithms for the following generalizations of the SubsetSum problem:

(a) Given an array $X[1..n]$ of positive integers and an integer T, compute the number of subsets of X whose elements sum to T.

(b) Given two arrays $X[1..n]$ and $W[1..n]$ of positive integers and an integer T, where each $W[i]$ denotes the weight of the corresponding element $X[i]$, compute the maximum weight subset of X whose elements sum to T. If no subset of X sums to T, your algorithm should return $-\infty$.

a)

Using my previous subset generating algorithm:

```
#recursive function that takes in a set and returns all possible subsets
#this recursive function always returns a list of lists
def subsets(set):
    #if subset is empty, return itself as the only subset of this set
    #as an empty set contains no subset
    if set == []:
        return [set]
    #set x equal to all subsets of the rest of the set after index 0
    x = subsets(set[1:])
    # all possible subsets after 0 + (all possible subsets after 0 + 0 for
each subset) = all possible subsets
    return x + [[set[0]] + y for y in x]
```

I generate each subset, then using another function

```
def sumSets(subsets, T):
    return list(subset for subset in subsets if sum(subset) == T)
```

To return each subset which elements sum to T.

The number of subsets which sum to T is simply the length of the list of subsets.

This algorithm has a Time Complexity of O(n^2) as the upper bound is generating the list of possible subsets, and the time complexity has not changed from the original function as I am referencing the same function.

Proof:

```
X = [1, 2, 3, 4]
T = 4
sumsets = sumSets(subsets(X), T)
print(len(sumsets))
Prints 2 as sumsets == [[4], [1, 3]]
```

b)

Employing the same algorithms above, I can use the following function

```
def maxWeight(X, W, T):
    #fetch all subsets of X which sum to T
    sumsets = sumSets(subsets(X), T)

    #if no subset of X sums to T, return negative infinity
    if sumsets == []:
        return float('-inf')

    #calculate the sum of weights from each viable sumset
    weightsets = []
    for sumset in sumsets:
        sum = 0
        for e in sumset:
            sum += W[X.index(e)]
        weightsets.append(sum)

    #return highest weight set
    return max(weightsets)
```

Which will return the highest weightsum of a subset who sums to T.

Proof:

```
X = [1, 2, 3, 4]
W = [5, 4, 6, 8]
T = 4

print(maxWeight(X, W, T))
Prints 11 as the subset [1, 3] sums to 4 and its weights = [5, 6] which
have the sum of 11 which is higher than subset 4 which weights sum to 8
```

The time complexity remains O(n^2) as the upper bound still remains at generating the subsets themselves.

Dynamic Programming:

P3 (20pt): In a strange country, the currency is available in the following denominations: \$1, \$4, \$7, \$13, \$28, \$52, \$91, \$365. Find the minimum bills that add up to a given sum $\$k$.

(a) The greedy change algorithm repeatedly takes the largest bill that does not exceed the target amount. For example, to make \$122 using the greedy algorithm, we first take a \$91 bill, then a \$28 bill, and finally three \$1 bills. Give an example where this greedy algorithm uses more bills than the minimum possible. [Hint: It may be easier to write a small program than to work this out by hand.]

(b) Describe and analyze a recursive algorithm that computes, given an integer k, the minimum number of bills needed to make $\$k$. (Don't worry about making your algorithm fast; just make sure it's correct.)

(c) Describe a dynamic programming algorithm that computes, given an integer k, the minimum number of bills needed to make $\$k$. (This one needs to be fast.)

a)

The greedy algorithm fails whenever subtracting large bills results in a remainder that is split up less optimally by the smaller change.

My greedy algorithm

```
def greedychange(k):
    global bills

    if k == 0:
        return []

    lowest = 1
    for b in reversed(bills):
        if b <= k:
            lowest = b
            break

    return list([lowest] + greedychange(k-lowest))
```

b)

```
def print_all_sum_rec(target, current_sum, start, output, result):
 if current_sum == target:
   output.append(copy.copy(result))

 for i in range(start, target):
   temp_sum = current_sum + i
   if temp_sum <= target:
     result.append(i)
     print_all_sum_rec(target, temp_sum, i, output, result)
     result.pop()
   else:
     return

def print_all_sum(target):
 output = []
 result = []
 print_all_sum_rec(target, 0, 1, output, result)
 return output
```

I can use the following function to get all of the sums, then simply return the sum with the smallest list of change c)

```
prevsums = {}

def sums(k, curr, output, result):
    global prevsums

    if k == curr:
        output.append(result.copy())

    for i in range(k - curr, 0, -1):
        temp = curr + i
        if temp in prevsums:
            sums(k, temp, output, prevsums[temp])
            return
        elif i in prevsums:
            comb = result + prevsums[i]
            if (not temp in prevsums.keys()) or (len(prevsums[temp]) >
len(comb)):
                prevsums[temp] = comb

            sums(k, temp, output, comb)

def getChange(k):
    global prevsums

    for b in bills:
        prevsums[b] = [b]

    out = []
    res = []
    sums(k, 0, out, res)
    ls = out[0]
    for lst in out:
        if len(lst) < len(ls):
            ls = lst
    return ls
```

The following method remembers previous sums using my prevsums dictionary

```
prevsums[number] = sum
```

Which in use will reduce the runtime exponentially using dynamic programming

P4 (30pt): Eggs break when dropped from great enough height. Specifically, there must be a floor f in any sufficiently tall building such that an egg dropped from the fth floor breaks, but one dropped from the $(f-1)$st floor will not. If the egg always breaks, then $f=1$. If the egg never breaks, then $f=n+1$.

You seek to find the critical floor f using an n-story building. The only operation you can perform is to drop an egg off some floor and see what happens. You start out with k eggs, and seek to drop eggs as few times as possible. Broken eggs cannot be reused. Let $E(k,n)$ be the minimum number of egg droppings that will always suffice.

1. (a) Show that $E(1,n)=n$.
2. (b) Show that $E(k,n)=\Theta(n^{1/k})$. [Hint1: Try to recurse; Hint2:mathematical induction on k. Hint3: treat variables as real numbers and minimize via derivative.]
3. (c) Find a recurrence for $E(k,n)$. What is the running time of the dynamic pro- gram to find $E(k,n)$? [Note: this is different from (b). k, n, $E(k,n)$ have to be integers.]

a)

Since you cannot afford to break your only egg without finding f, simply do a linear scan

```
For f = 1, f++
        Drop egg on f floor
        If breaks
                Return f
```

And therefore the min droppings that will always suffice is N as you are searching linearly

b)

```
int E(int k, int n)
{
    // If there are no floors,
    // then no trials needed.
    // OR if there is one floor,
    // one trial needed.
    if (n == 1 || n == 0)
        return n;

    // We need n trials for one
    // egg and n floors
    if (k == 1)
        return n;

    int min = INT_MAX, x, res;

    // Consider all droppings from
    // 1st floor to nth floor and
    // return the minimum of these
    // values plus 1.
    for (x = 1; x <= n; x++) {
        res = max(E(k - 1, x - 1), E(k, n - x));
        if (res < min)
            min = res;
    }

    return min + 1;
}
```

The time complexity for this recursive algorithm is O(n^k) as you have to calculate
n droppings 1/k times as each recur there is n - 1 more calculations of k -1 to make until n or k reaches 1

c)
Dynamic Programming saves the computer time by not forcing it to recalculate subproblems over again.

```
int E(int k, int n)
{
    /* A 2D table where entry
    eggFloor[i][j] will represent
    minimum number of trials needed for
    i eggs and j floors. */
    int eF[k + 1][n + 1];
    int res;
    int i, j, x;

    // We need one trial for one floor and 0
    // trials for 0 floors
    for (i = 1; i <= k; i++) {
        eggFloor[i][1] = 1;
        eggFloor[i][0] = 0;
    }

    // We always need j trials for one egg
    // and j floors.
    for (j = 1; j <= n; j++)
        eggFloor[1][j] = j;

    // Fill rest of the entries in table using
    // optimal substructure property
    for (i = 2; i <= k; i++) {
        for (j = 2; j <= n; j++) {
            eF[i][j] = INT_MAX;
            for (x = 1; x <= j; x++) {
                res = 1 + max(eF[i - 1][x - 1], eF[i][j - x]);
                if (res < eggFloor[i][j])
                    eggFloor[i][j] = res;
            }
        }
    }

    // eggFloor[k][n] holds the result
    return eggFloor[k][n];
}
```

The time complexity of this algorithm is O(k*n^2)