_merged.py

```
CoinChange.py
def coinChange(coins, amount):
    inf = float('inf')
    m = len(coins)
    n = amount + 1
    dP = [0] + [inf]*amount

    for j in range(coins[0], n):
        dP[j] = dP[j-coins[0]] + 1

    for i in range(1, m):
        for j in range(1, n):
            if j >= coins[i]:
                dP[j] = min(dP[j], dP[j-coins[i]] + 1)

    if dP[n-1] == inf:
        return -1
    return dP[n - 1]


print(coinChange([2,5,10,1], 27))

InFixToPostFix.py
"""
A * B + C

A B * C +

Code follows http://csis.pace.edu/~wolf/CS122/infix-postfix.htm
"""

def isOperator(op):
    return op in "+-*/"

def getOpRank(op):
    if op in "()":
        return 0
    elif op in "+-":
        return 1
    return 2

def inFixToPostFix(inFix):
    postFix = ""
    opStack = []
    for c in inFix:
        if c.isalnum():
            postFix += c
        elif isOperator(c): # is operand
            # if stack is empty, append
            # elif c has lower or equal precedence to stack's top, pop
            # else append
            if len(opStack) and getOpRank(c) <= getOpRank(opStack[-1]):
                postFix += opStack.pop()
            opStack.append(c)
        else: # is ( or )
            # if c is a ), pop all ops until (
            if c == "(":
                opStack.append(c)
            else:
                while opStack[-1] != "(":
                    postFix += opStack.pop()
                opStack.pop()

    while len(opStack):
        postFix += opStack.pop()

    return postFix

print(inFixToPostFix("(A+D)*(B+C*F)"))

Knapsack.py
def knapsack(values, weights, weightLimit):
    return knapsackRecursive(values, weights, weightLimit, 0, weightLimit, {})

def knapsackRecursive(values, weights, weightLimit, i, remainingCapacity, memo):
    if i >= len(weights):
        return 0
    if remainingCapacity < 0:
        return 0

    if (i, remainingCapacity) in memo:
        return memo[(i, remainingCapacity)]

    if remainingCapacity < weights[i]:
        maxValue = knapsackRecursive(values, weights, weightLimit, i+1, remainingCapacity, memo)
    else:
        valueIfIncludeCurrent = values[i] + knapsackRecursive(values, weights, weightLimit, i+1, remainingCapacity - weights[i], memo)
        valueIfNoIncludeCurrent = knapsackRecursive(values, weights, weightLimit, i+1, remainingCapacity, memo)
        maxValue = max(valueIfIncludeCurrent, valueIfNoIncludeCurrent)

    memo[(i, remainingCapacity)] = maxValue
    return maxValue

print(knapsack([22, 20, 15, 30, 24, 54, 21, 32, 18, 25], [4, 2, 3, 5, 5, 6, 9, 7, 8, 10], 30))

NQueensProblem.py
"""
[] <- [a,b,c,d,e,f,g,h], where index i is the ith column and
            a,b,c... is the row corresponding to the column

rec(curSolution, N)

rec([], 4)
    rec([0], 4)
        rec([0,2], 4)
            stuck
        rec([0,3], 4)
            rec([0,3,1], 4)
                stuck
            stuck
        stuck
    rec([1], 4)
        rec([1,3], 4)
            rec([1,3,0], 4)
                rec([1,3,0,2, 4)
                    true!
        rec([1,4], 4)
    rec([2], 4)
    rec([3], 4)
"""

def conflictRow(curSolution, toAddRow):
    for row in curSolution:
        if row == toAddRow:
            return True
    return False

def conflictDiagonal(curSolution, toAddRow):
    toAddCol = len(curSolution)
    for col, row in enumerate(curSolution):
        if abs(col-toAddCol) == abs(row-toAddRow):
            return True
    return False

def nQueens(curSolution, N):
    if len(curSolution) == N:
        return curSolution
    else:
        for i in range(N):
            # check valid
            if not conflictRow(curSolution, i) and not conflictDiagonal(curSolution, i):
                curSolution.append(i)
                if nQueens(curSolution, N):
                    return curSolution
                curSolution.pop()
    return None

print(nQueens([], 8))

Parentheses.py
def addParen(list, leftRem, rightRem, str, index):
    if leftRem < 0 or rightRem < leftRem:
        return

    if leftRem == 0 and rightRem == 0:
        list.append(''.join(str))
    else:
        str[index] = '('
        addParen(list, leftRem - 1, rightRem, str, index + 1)

        str[index] = ')'
        addParen(list, leftRem, rightRem - 1, str, index + 1)

def generateParents(count):
    str = [None]*(count*2)
    list = []
    addParen(list, count, count, str, 0)
    return list

print(generateParents(5))

PowerSet.py
"""
[1,2,3]

rec(set, 0)
    rec(set, 1)
        rec(set, 2)
            rec(set, 3) = [[]]
        [[], [2]]
    [[],[3],[2],[2,3]]
[[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]
"""
```

```
def getSubsetsRecursive(mySet, index):
    if index < 0:
        return [[]]
    allSubsets = getSubsetsRecursive(mySet, index-1)

    subsetsFromHere = []
    for set in allSubsets:
        subsetsFromHere.append(set + [mySet[index]])

    allSubsets += subsetsFromHere
    return allSubsets

def getSubsets(mySet):
    return getSubsetsRecursive(mySet, len(mySet) - 1)


def bitsToSet(mySet, bits):
    thisSet = []

    index = 0
    while bits > 0:
        if bits & 1:
            thisSet.append(mySet[index])
        bits >>= 1
        index += 1

    return thisSet

def getSubsetsIterative(mySet):
    allSubsets = []

    noOfSubsets = 1 << len(mySet) # 2**n

    for bits in range(noOfSubsets):
        allSubsets.append(bitsToSet(mySet, bits))

    return allSubsets


print(getSubsets([1,2,3]))
print(getSubsetsIterative([1,2,3]))

StringPermutations.py
"""
abcd

rec(s, 0)
    [a]
    rec(s, 1)
        [ab, ba]
"""
def getPerms(string):
    if string is None:
        return None

    permutations = []
    if len(string) == 0:
        permutations.append("")
        return permutations

    first = string[0]
    remainder = string[1:]

    words = getPerms(remainder)
    for word in words:
        for j in range(len(word)+1):
            s = insertCharAt(word, first, j)
            permutations.append(s)
    return permutations

def insertCharAt(word, c, i):
    start = word[0:i]
    end = word[i:]
    return start + c + end

print(getPerms("abcd"))

TowersOfHanoi.py
class Tower:
    def __init__(self, index):
        self.discs = []
        self.index = index

    def getIndex(self):
        return self.index

    def add(self, d):
        if len(self.discs) and d > self.discs[-1]:
            raise Exception("Cannot put a bigger disc on a smaller disc")
        self.discs.append(d)

    def moveTopTo(self, t):
        top = self.discs.pop()
        t.add(top)
    def moveDiscs(self, n, destination, buffer, stacks):
        if n > 0:
            self.moveDiscs(n-1, buffer, destination, stacks)
            self.moveTopTo(destination)
            print([stacks[0].discs, stacks[1].discs, stacks[2].discs])
            buffer.moveDiscs(n-1, destination, self, stacks)

towers = []
for i in range(3):
    towers.append(Tower(i))

n = 4
for i in range(n-1, -1, -1):
    towers[0].add(i)

print("Hey! I'm gna move {} discs from {} to {} using {} as a buffer".format(n, 0, 2, 1))
towers[0].moveDiscs(n, towers[2], towers[1], towers)

print(towers[2].discs)
```