

1) As discussed in Section 14.3 (“Three Applications of Stacks”) we could use backtracking with a stack to maintain the list of unexplored choices, but we can also use recursion (with its run-time stack) to drive a backtracking algorithm. The general recursive backtracking algorithm for optimization problems (e.g., fewest number of coins) looks something like:

```
Backtrack( recursionTreeNode p ) {
    treeNode c;
    for each child c of p do
        if promising(c) then
            if c is a solution that's better than best then
                best = c
            else
                Backtrack(c)
            end if
        end if
    end for
} // end Backtrack
```

each c represents a possible choice
c is "promising" if it could lead to a better solution
check if this is the best solution found so far
remember the best solution
follow a branch down the tree

General Notes about Backtracking:

- The depth-first nature of backtracking only stores information about the current branch being explored so the memory usage is “low” even though the number of tree nodes might be exponential (2^n).
- Each node of the search-space (recursive-call) tree maintains the state of a partial solution. In general the partial solution state consists of potentially large arrays that change little between parent and child. To avoid having multiple copies of these arrays, a single “global” state is maintained which is updated before we go down to the child (via a recursive call) and undone when we backtrack to the parent.

For the coin-change problem:

a) What defines the current state of a search-space tree node?

b) When would a “child” search-space tree node NOT be promising?

```

backtrackingNodes = 0 # profiling variable to track number of state-space tree nodes

def main():
    changeAmt = int(raw_input("Enter the change amount: "))
    coinTypes = raw_input("Enter the coin types separated by spaces: ")
    coinTypes = coinTypes.split(" ")
    for index in xrange(len(coinTypes)):
        coinTypes[index] = int(coinTypes[index])
    print "Change Amount:", changeAmt, " Coin types:", coinTypes
    fewestCoins, numberOfEachCoinType = solveCoinChange(changeAmt, coinTypes)
    print "Fewest number of coins", fewestCoins
    print "The number of each type of coin in the solution is:"
    for index in xrange(len(coinTypes)):
        print "number of %s-cent coins is %s"%(str(coinTypes[index]), str(numberOfEachCoinType[index]))
    return

def solveCoinChange(changeAmt, coinTypes):

    def backtrack(changeAmt, numberOfEachCoinType, numberOfCoinsSoFar, solutionFound, bestFewestCoins, bestNumberOfEachCoinType):
        global backtrackingNodes
        backtrackingNodes += 1

        for index in xrange(len(coinTypes)):
            smallerChangeAmt = changeAmt - coinTypes[index]
            if promising(smallerChangeAmt, numberOfCoinsSoFar+1, solutionFound, bestFewestCoins):
                if smallerChangeAmt == 0: # a solution is found
                    if (not solutionFound) or numberOfCoinsSoFar + 1 < bestFewestCoins: # check if its best
                        bestFewestCoins = numberOfCoinsSoFar+1
                        bestNumberOfEachCoinType = [] + numberOfEachCoinType
                        bestNumberOfEachCoinType[index] += 1
                        solutionFound = True
                else:
                    # call child with updated state information
                    smallerChangeAmtNumberOfEachCoinType = [] + numberOfEachCoinType
                    smallerChangeAmtNumberOfEachCoinType[index] += 1

                    solutionFound, bestFewestCoins, bestNumberOfEachCoinType = backtrack(smallerChangeAmt, smallerChangeAmtNumberOfEachCoinType,
                                                                                          numberOfCoinsSoFar + 1, solutionFound,
                                                                                          bestFewestCoins, bestNumberOfEachCoinType)

        return solutionFound, bestFewestCoins, bestNumberOfEachCoinType
    # end def backtrack

    def promising(changeAmt, numberOfCoinsReturned, solutionFound, bestFewestCoins):
        if changeAmt < 0:
            return False
        elif changeAmt == 0:
            return True
        else: # changeAmt > 0
            if solutionFound and numberOfCoinsReturned+1 >= bestFewestCoins:
                return False
            else:
                return True

    # set-up initial "current state" information
    numberOfEachCoinType = []
    numberOfCoinsSoFar = 0
    solutionFound = False
    bestFewestCoins = -1
    bestNumberOfEachCoinType = None

    numberOfEachCoinType = []
    for coin in coinTypes:
        numberOfEachCoinType.append(0)
    numberOfCoinsSoFar = 0
    solutionFound = False
    bestFewestCoins = -1
    bestNumberOfEachCoinType = None

    solutionFound, bestFewestCoins, bestNumberOfEachCoinType = backtrack(changeAmt, numberOfEachCoinType, numberOfCoinsSoFar, solutionFound,
                                                                           bestFewestCoins, bestNumberOfEachCoinType)

    return bestFewestCoins, bestNumberOfEachCoinType

main()
print "Number of Backtracking Nodes:", backtrackingNodes

```

2) Consider the output of running the above backtracking code twice with a change amount of 399 cents.

```
>>>
Enter the change amount: 399
Enter the coin types separated by spaces: 1 5 10 12 25 50
Change Amount: 399   Coin types: [1, 5, 10, 12, 25, 50]
Run-time: 1289.869 seconds
Fewest number of coins 10
The number of each type of coin in the solution is:
number of 1-cent coins is 0
number of 5-cent coins is 0
number of 10-cent coins is 0
number of 12-cent coins is 2
number of 25-cent coins is 1
number of 50-cent coins is 7
Number of Backtracking Nodes: 302715760
>>> ===== RESTART =====
>>>
Enter the change amount: 399
Enter the coin types separated by spaces: 50 25 12 10 5 1
Change Amount: 399   Coin types: [50, 25, 12, 10, 5, 1]
Run-time: 8.360 seconds
Fewest number of coins 10
The number of each type of coin in the solution is:
number of 50-cent coins is 7
number of 25-cent coins is 1
number of 12-cent coins is 2
number of 10-cent coins is 0
number of 5-cent coins is 0
number of 1-cent coins is 0
Number of Backtracking Nodes: 2015539
>>>
```

a) Explain why ordering the coins from largest to smallest produced faster results.

b) For coins of {1, 5, 10, 12, 25, 50}, typical timings:

Change Amount	Run-Time (seconds)	Number of Tree Nodes
200	0.92	236,583
300	33.23	8,617,265
320	64.12	16,676,454
340	116.8	30,370,729

a) Why the exponential growth in run-time?

3) As with Fibonacci, the coin-change problem can benefit from dynamic program since it was slow due to solving the same problems over-and-over again. Recall the general idea of dynamic programming:

- Solve smaller problems before larger ones
- store their answers
- look-up answers to smaller problems when solving larger subproblems

Advantage:

Eliminates redundant work since each smaller problem solved only once!

a) How do we solve the coin-change problem using dynamic programming?

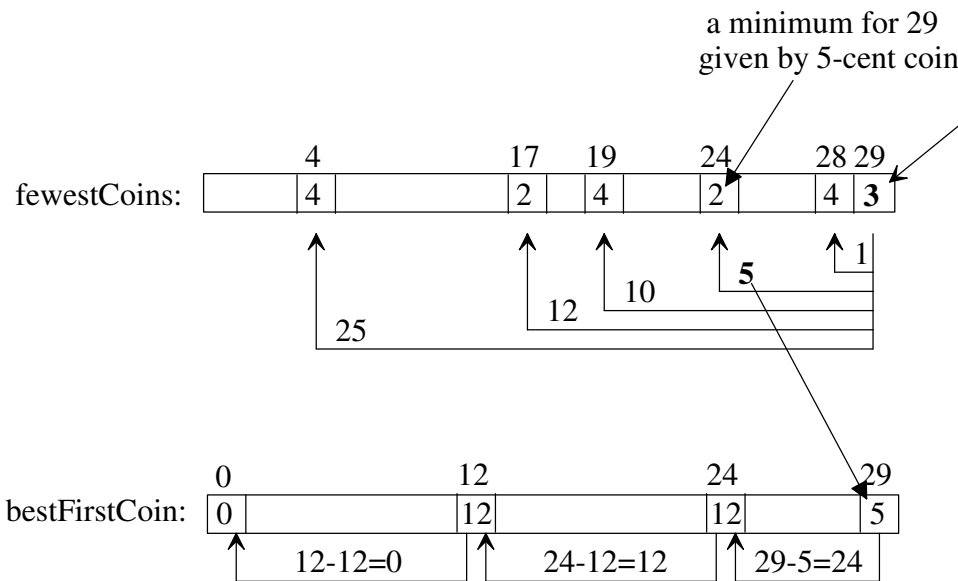
Dynamic Programming Coin-change Algorithm:

I. Fills an array fewestCoins from 0 to the amount of change. An element of fewestCoins stores the fewest number of coins necessary for the amount of change corresponding to its index value.

For 29-cents using the set of coin types {1, 5, 10, 12, 25, 50}, the dynamic programming algorithm would have previously calculated the fewestCoins for the change amounts of 0, 1, 2, ..., up to 28 cents.

II. If we record the best, first coin to return for each change amount (found in the “minimum” calculation) in an array bestFirstCoin, then we can easily recover the actual coin types to return.

$$\text{fewestCoins}[29] = \text{minimum}(\text{fewestCoins}[28], \text{fewestCoins}[24], \text{fewestCoins}[19], \text{fewestCoins}[17], \text{fewestCoins}[4]) + 1 = 2 + 1 = 3$$



Extract the coins in the solution for 29-cents from bestFirstCoin[29], bestFirstCoin[24], and bestFirstCoin[12]

b) Extend the lists through 32-cents.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
fewestCoins:	0	1	2	3	4	1	2	3	4	5	1	2	1	2	3	2	3	2	3	4	2	3	2	3	2	1	2	3	4	3			
bestFirstCoin:	0	1	1	1	1	5	1	1	1	1	10	1	12	1	1	5	1	5	1	1	10	1	10	1	12	25	1	1	1	5			

c) What coins are in the solution for 32-cents?