Name:_____

Consider the coin-change problem: Given a set of coin types and an amount of change to be returned, determine the **fewest number** of coins for this amount of change.

1) What "greedy" algorithm would you use to solve this problem with US coin types of {1, 5, 10, 25, 50} and a change amount of 29-cents?

2) Do you get the correct solution if you use this algorithm for coin types of {1, 5, 10, 12, 25, 50} and a change amount of 29-cents?

3) One way to solve this problem in general is to use an exhaustive search via a divide-and-conquer algorithm. Recall the idea of **Divide-and-Conquer** algorithms: Solve a problem by:

- dividing it into smaller problem(s) of the same kind
- solving the smaller problem(s) recursively
- use the solution(s) to the smaller problem(s) to solve the original problem

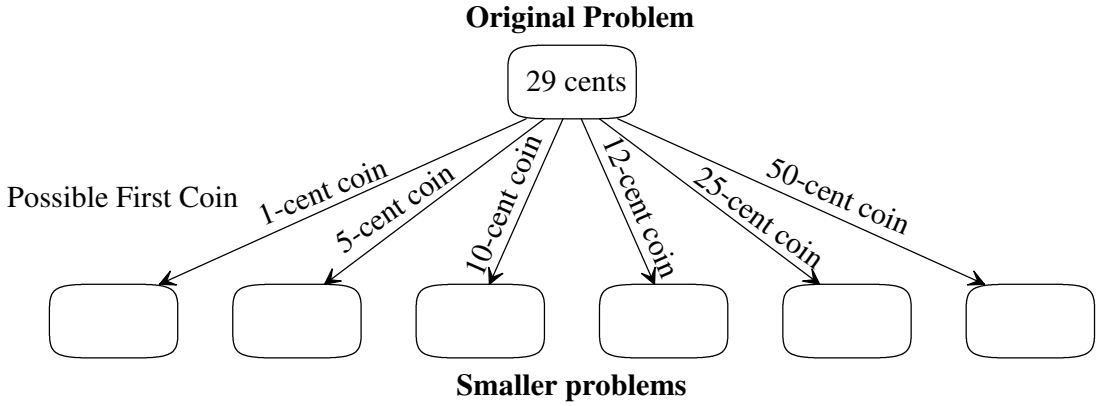
a) For the coin-change problem, what determines the size of the problem?

b) How could we divide the coin-change problem for 29-cents into smaller problems?

c) If we knew the solution to these smaller problems, how would be able to solve the original problem?

Name:_____

4) After we give back the first coin, which smaller amounts of change do we have?

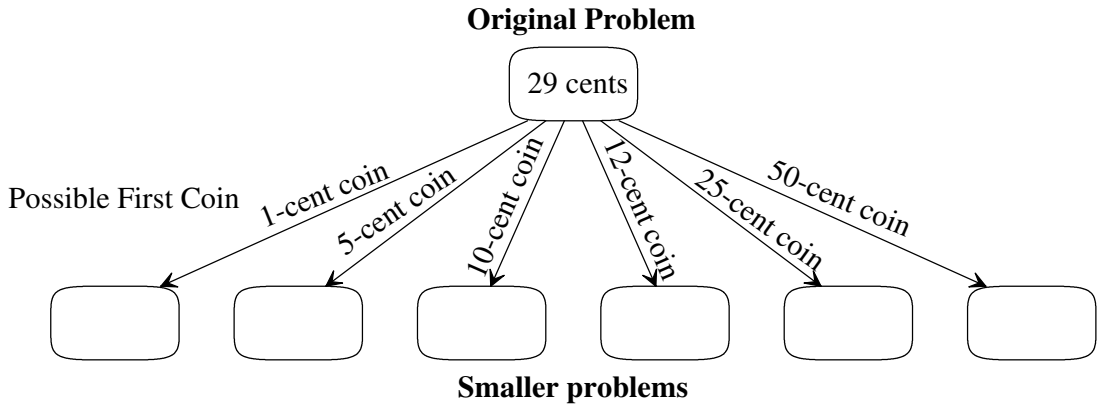


5) If we knew the fewest number of coins needed for each possible smaller problem, then how could determine the fewest number of coins needed for the original problem?

6) Complete a recursive relationship for the fewest number of coins.

$$\text{FewestCoins(change)} = \begin{cases} \min_{\text{coin} \in \text{CoinSet}}(\text{FewestCoins}(\quad)) + & \text{if change} \notin \text{CoinSet} \\ 1 & \text{if change} \in \text{CoinSet} \end{cases}$$

7) Complete a couple levels of the recursion tree for 29-cents change using the set of coin types {1, 5, 10, 12, 25, 50},



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

Change Amount	Run-Time (seconds)
50	1
55	4
60	19
65	93
70	449

Why the exponential growth in run-time?

```
int fewestCoins(int change, int coinTypes[], int numberOfCoinTypes) {

      int coin, minNumberOfCoins, numberOfCoins;

      // Find the min. number of coins after giving back one coin of each type
      minNumberOfCoins = INT_MAX;

      for (coin = 0; coin < numberOfCoinTypes; coin++){
            if (change == coinTypes[coin]) {
                  return 1;
            } else if (change > coinTypes[coin]) {
                  numberOfCoins = fewestCoins(change-coinTypes[coin], coinTypes,
                                              numberOfCoinTypes);
                  if (numberOfCoins < minNumberOfCoins) {
                        minNumberOfCoins = numberOfCoins;
                  } // end if (numberOfCoins...
            } // end if (change...
      } // end for (coin...

      return  minNumberOfCoins + 1;
} // end FewestCoins
```

8) Above is the recursive, exhaustive search, backtracking solution. Typically, we want to stop following non-promising branches of the search-space tree as soon as possible to speed the search. The general recursive backtracking algorithm *with pruning* for optimization problems (e.g., fewest number of coins) looks something like:

```
Backtrack( recursionTreeNode p ) {

   treeNode c;
   for each child c of p do                          // each c represents a possible choice
      if promising(c) then                           // c is "promising" if it could lead to a better solution
            if c is a solution that's better than best then   // check if this is the best solution found so far
                  best = c                           // remember the best solution
            else
                  Backtrack(c)                       // follow a branch down the tree
            end if
      end if
   end for
} // end Backtrack
```

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”

- Each node of the search space tree maintains the state of a partial solution. In general the 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.
- We could use the concept of backtracking without recursion by using a stack to maintain a collection of unexplored choices. Thus, we would simulate the run-time stack to drive the backtracking algorithm.

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?

9) 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?

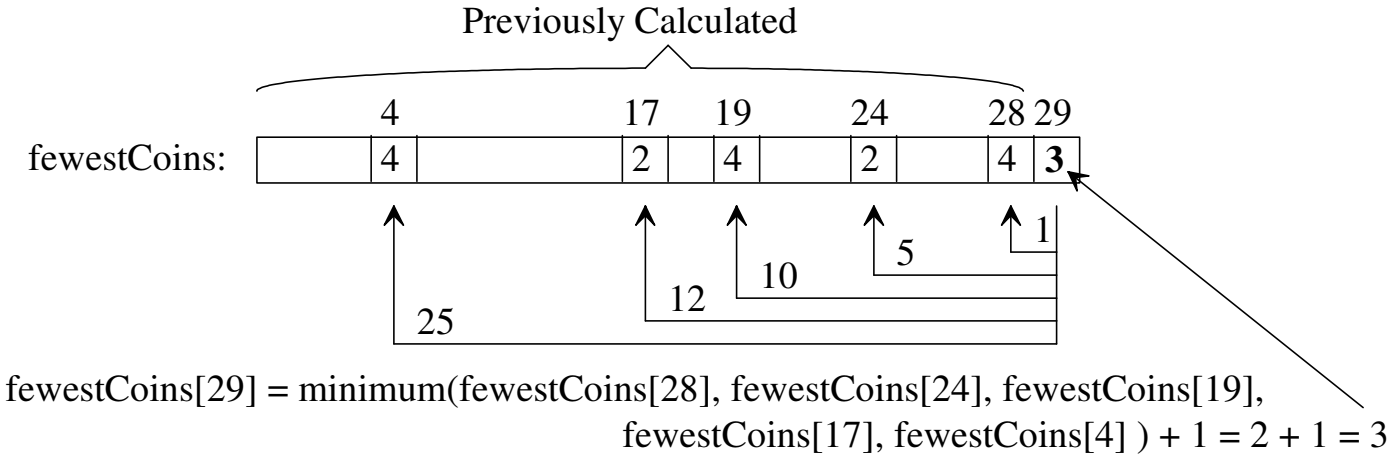
Name:_____

Dynamic Programming Coin-change Solution

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}:



fewestCoins[29] = minimum(fewestCoins[28], fewestCoins[24], fewestCoins[19], fewestCoins[17], fewestCoins[4]) + 1 = 2 + 1 = 3

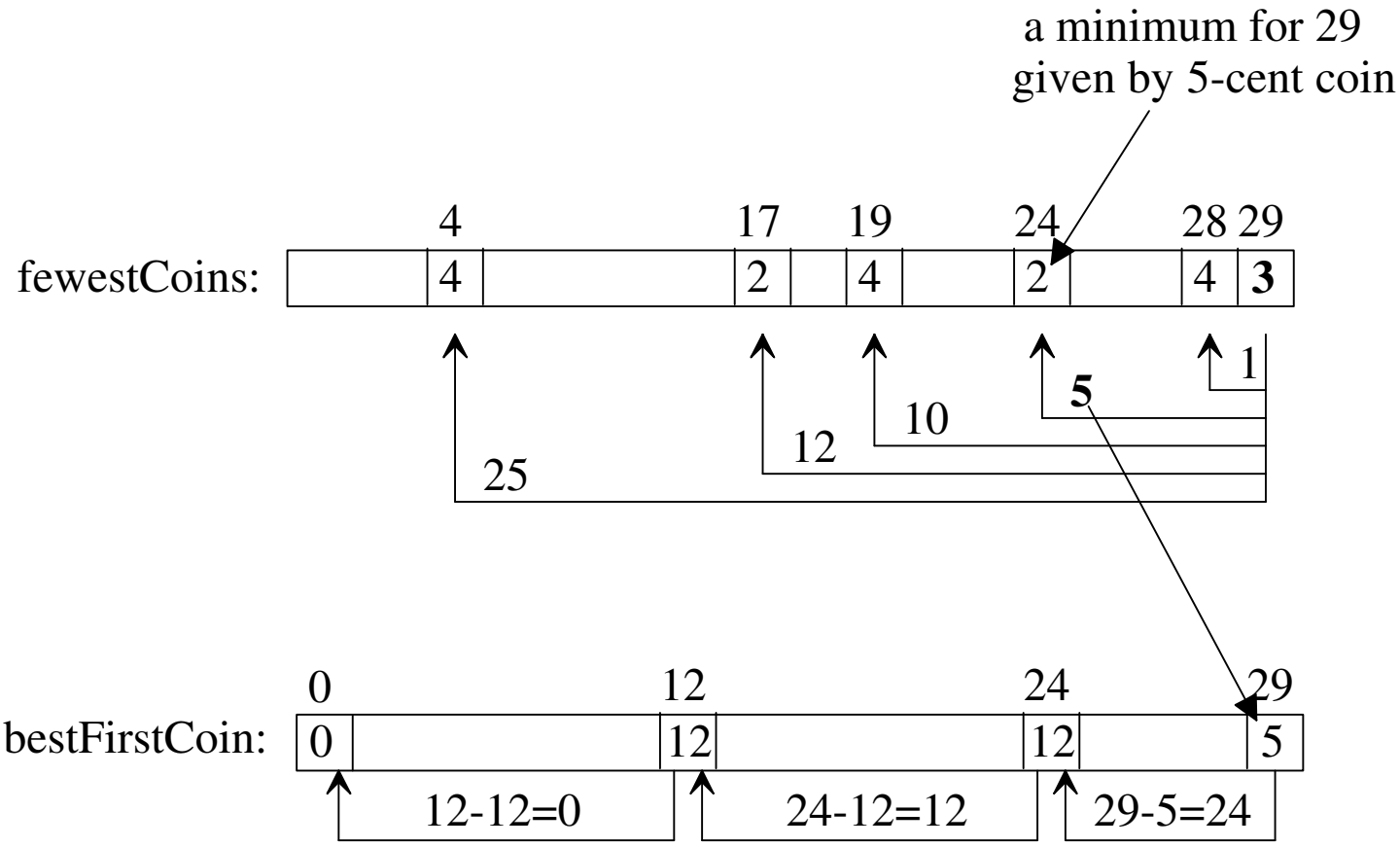
NOTICE: that the algorithm has previously calculated the fewestCoins for the change amounts of 0, 1, 2, ..., up to 28 cents.

Name:_____

Keeping Track of Coins in the Solution

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.

For the 29-cent solution with coin types {1, 5, 10, 12, 25, 50}.



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