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

Original Problem

Possible First Coin
- 1-cent coin
- 5-cent coin
- 10-cent coin
- 25-cent coin
- 50-cent coin

Smaller problems
- 28 cents
- 24 cents
- 19 cents
- 17 cents
- 4 cents

4. 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?

5. Complete a recursive relationship for the fewest number of coins.

\[
\text{FewestCoins}(\text{change}) = \begin{cases} 
\min( \text{FewestCoins}(\text{change} - \text{coin})) + 1 & \text{if change} \notin \text{CoinSet} \\
1 & \text{if change} \in \text{CoinSet and coin} \leq \text{change} 
\end{cases}
\]

6. Complete a couple levels of the recursion tree for 29-cents change using the set of coins \{1, 5, 10, 12, 25, 50\}.
Data Structures

Lecture 13

1. The textbook solves the coin-change problem with the following code (note the “set-builder-like” notation):
\[ \{ c | c \in \text{coinValueList and } c \leq \text{change} \} \]

Results of running this code:

Change Amount: 63 Coin types: [1, 5, 10, 25]
Run-time: 70.689 seconds
Fewest number of coins: 6
Number of Backtracking Nodes: 67,716,925

I removed the fancy set-builder notation and replaced it with a simple if-statement check:

```python
def recMC(change, coinValueList):
    global backtrackingNodes
    backtrackingNodes += 1
    minCoins = change
    if change in coinValueList:
        return 1
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recMC(change - i, coinValueList)
        if numCoins < minCoins:
            minCoins = numCoins
        return minCoins
```

Results of running this code:

Change Amount: 63 Coin types: [1, 5, 10, 25]
Run-time: 45.815 seconds
Fewest number of coins: 6
Number of Backtracking Nodes: 67,716,925

a) Why is the second version so much “faster”? The 1st version rebuilds the list of valid coins on each of 67 million calls. The 2nd version use the original list with an if-statement check.

b) Why does it still take a long time? As with recursive fibonacci we solve the smaller change problems from scratch each time we encounter them. Thus, the recursion tree grows exponentially.

2. To speed the recursive backtracking algorithm, we can prune unpromising branches. The general recursive backtracking algorithm for optimization problems (e.g., fewest number of coins) looks something like:

```python
def Backtrack( recursionTreeNode p ) {
    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 for
}
```

General Notes about Backtracking:

- The depth-first nature of backtracking only stores information about the current branch being explored on the run-time stack, so the memory usage is “low” even though the # of recursion 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 reference to a single “global” array can be maintained which is updated before we go down to the child (via a recursive call) and undone when we backtrack to the parent.

a) For the coin-change problem, what defines the current state of a search-space tree node?
b) When would a “child” tree node NOT be promising?

1. Change amount goes negative (29 + change + 50 * coin = -2)
2. If tree node cannot do better than a previously found solution

(See attached page)

3. Consider the output of running the backtracking code with pruning (next page) twice with a change amount of 63 cents.

\[
\begin{array}{|c|c|c|}
\hline
\text{Change Amount: 63} & \text{Coin types: [1, 5, 10, 25]} & \text{Change Amount: 63} \\
\text{Run-time: 0.036 seconds} & \text{Coin types: [25, 10, 5, 1]} & \text{Run-time: 0.003 seconds} \\
\text{Fewest number of coins 6} & \text{The number of each type of coins is:} \\
\text{number of 1-cent coins is 3} & \text{number of 25-cent coins is 2} \\
\text{number of 5-cent coins is 0} & \text{number of 10-cent coins is 1} \\
\text{number of 10-cent coins is 1} & \text{number of 5-cent coins is 0} \\
\text{number of 25-cent coins is 2} & \text{number of 1-cent coins is 3} \\
\text{Number of Backtracking Nodes: 4831} & \text{Number of Backtracking Nodes: 310} \\
\hline
\end{array}
\]

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

With \([1, 5, 10, 25]\) the 1st solution found has 29 coins (all pennies) but with \([25, 10, 5, 1]\) the 1st solution found is a 5 coin solution that is more helpful in pruning the tree.

b) For coins of \([50, 25, 12, 10, 5, 1]\) typical timings:

<table>
<thead>
<tr>
<th>Change Amount</th>
<th>Run-Time (seconds)</th>
<th>Number of Tree Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>399</td>
<td>8.88</td>
<td>2,015,539</td>
</tr>
<tr>
<td>409</td>
<td>55.17</td>
<td>12,093,221</td>
</tr>
<tr>
<td>419</td>
<td>318.56</td>
<td>72,558,646</td>
</tr>
</tbody>
</table>

Why the exponential growth in run-time?

Recalculation of smaller problems many times.

4. As with Fibonacci, the coin-change problem can benefit from dynamic programming 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, so each problem is solved only once

a) To solve the coin-change problem using dynamic programming, we need to answer the questions:

- What is the smallest problem?

Where do we store the answers to the smaller problems?
Suppose we already have a 10-coin solution.

If we are following a branch and already have given back some coins, then it cannot do better than the best 10-coin solution. Thus, stop following it!

Thus, we can prune all other branches to 4-coins at most.

If 33 coins fit from biggest to smallest, 1st solution is 6, 5, 3, 1, 1.

If 44 coins are there, then cannot do better than the best 5-coin solution.

If 44 coins are there, then cannot do better than the best 5-coin solution.
Data Structures

Lecture 13

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

def solveCoinChange(changeAmt, coinTypes):
    global backtrackingNodes
    backtrackingNodes += 1

    for index in range(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:
                    bestFewestCoins = numberOfCoinsSoFar + 1
                    bestNumberOfEachCoinType = [i] + numberOfEachCoinType
                    bestNumberOfEachCoinType[index] += 1
                    solutionFound = True
            else:
                # call child with updated state information
                smallerChangeAmtNumberOfEachCoinType = [i] + 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:
        if solutionFound and numberOfCoinsReturned + 1 >= bestFewestCoins:
            return False
        else:
            return True

    # Body of solveCoinChange

    numberOfEachCoinType = []  # set-up initial "current state" information
    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
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.

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

\[10\$1, 10\$1, 12\$2\]
1. Consider the following sequential search (linear search) code:

<table>
<thead>
<tr>
<th>Textbook’s Listing 5.1</th>
<th>Faster sequential search code</th>
</tr>
</thead>
<tbody>
<tr>
<td>def sequentialSearch(alist, item):</td>
<td></td>
</tr>
</tbody>
</table>
|     """Sequential search of unordered list"""
|     pos = 0 |
|     found = False |
|     while pos < len(alist) and not found: |
|         if alist[pos] == item: |
|             found = True |
|         else: |
|             pos = pos + 1 |
|     return found |
| def linearSearch(alist, target):  |
|     """Returns the index of target in alist or -1 if target is not in alist"""
|     for position in range(len(alist)): |
|         if target == alist[position]: |
|             return position |
|     return -1 |

a) What is the basic operation of a search? **Comparison of items to target with list item.**

b) For the following alist value, which target value causes linearSearch to loop the fewest (“best case”) number of times?

```
alist: 10 0 1 2 3 4 5 6 7 8 9 10
```

O(1) best case

c) For the above alist value, which target value causes linearSearch to loop the most (“worst case”) number of times?

unsuccessful O(n)

d) For a successful search (i.e., target value in alist), what is the “average” number of loops? O(n)

e) The above version of linear search assumes that alist is sorted in ascending order. When would this version perform better than the original linearSearch at the top of the page?

Stops early if run across list item that’s bigger than target item.
2. Consider the following binary search code:

<table>
<thead>
<tr>
<th>Textbook’s Listing 5.3</th>
<th>Faster binary search code</th>
</tr>
</thead>
<tbody>
<tr>
<td>def binarySearch(alist, item):</td>
<td></td>
</tr>
<tr>
<td>first = 0</td>
<td></td>
</tr>
<tr>
<td>last = len(alist) - 1</td>
<td></td>
</tr>
<tr>
<td>found = False</td>
<td></td>
</tr>
<tr>
<td>while first &lt;= last and not found:</td>
<td></td>
</tr>
<tr>
<td>midpoint = (first + last) // 2</td>
<td></td>
</tr>
<tr>
<td>if alist[midpoint] == item:</td>
<td></td>
</tr>
<tr>
<td>found = True</td>
<td></td>
</tr>
<tr>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>if item &lt; alist[midpoint]:</td>
<td></td>
</tr>
<tr>
<td>last = midpoint - 1</td>
<td></td>
</tr>
<tr>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>first = midpoint + 1</td>
<td></td>
</tr>
<tr>
<td>return found</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>def binarySearch(target, lyst):</td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;&quot;Returns the position of the target item if found, or -1 otherwise.&quot;&quot;&quot;</td>
<td></td>
</tr>
<tr>
<td>left = 0</td>
<td></td>
</tr>
<tr>
<td>right = len(lyst) - 1</td>
<td></td>
</tr>
<tr>
<td>while left &lt;= right:</td>
<td></td>
</tr>
<tr>
<td>midpoint = (left + right) // 2</td>
<td></td>
</tr>
<tr>
<td>if target == lyst[midpoint]:</td>
<td></td>
</tr>
<tr>
<td>return midpoint</td>
<td></td>
</tr>
<tr>
<td>elif target &lt; lyst[midpoint]:</td>
<td></td>
</tr>
<tr>
<td>right = midpoint - 1</td>
<td></td>
</tr>
<tr>
<td>else:</td>
<td></td>
</tr>
<tr>
<td>left = midpoint + 1</td>
<td></td>
</tr>
<tr>
<td>return -1</td>
<td></td>
</tr>
</tbody>
</table>

a) “Trace” binary search to determine the worst-case basic total number of comparisons?

b) What is the worst-case big-oh for binary search?

c) What is the best-case big-oh for binary search?

d) What is the average-case (expected) big-oh for binary search?

e) If the list size is 1,000,000, then what is the maximum number of comparisons of list items on a successful search?

f) If the list size is 1,000,000, then how many comparisons would you expect on an unsuccessful search?