Adventures in Dynamic Programming
What is an Algorithm?

An algorithm is a sequence of instructions that one must follow to solve a well-formulated problem.
Correctness

- An algorithm is correct only if it produces correct result for every input instance
  - An algorithm is incorrect answer if it cannot produce a correct result for one or more input instances,
- Coin change problem
  - Input: an amount of money $M$ in cents, and a list of coin denominations $[c_1, c_2, \ldots, c_n]$
  - Output: the smallest number of coins that add to $M$ (may not be unique)
- US coin change problem
US Coin Change

72 cents

US Coin Change

Two quarters, 22 cents left

Two dimes, 2 cents left

Two pennies

Classic Algorithm

\[
\begin{align*}
    r & \leftarrow M \\
    q & \leftarrow r / 25 \\
    r & \leftarrow r - 25 \cdot q \\
    d & \leftarrow r / 10 \\
    r & \leftarrow r - 10 \cdot d \\
    n & \leftarrow r / 5 \\
    r & \leftarrow r - 5 \cdot n \\
    p & \leftarrow r
\end{align*}
\]

Can we generalize it?

Is it correct?
Change Problem

- **Input:**
  - an amount of money $M$
  - an array of denominations $c = (c_1, c_2, ..., c_d)$ in order of decreasing value
- **Output:** the smallest number of coins

\[
\begin{align*}
M &= 40 \\
c &= (25, 20, 10, 5, 1)
\end{align*}
\]

1. $r = M$
2. $n = 0$
3. for $k$ from 1 to $d$
   1. $i_k = r / c_k$
   2. $n = n + i_k$
   3. $r = r - c_k \times i_k$
4. return $n$

Incorrect algorithm!
The correct answer should be 2.

Is it correct?
Another Approach?

- Let's bring back brute force
- Test every coin combination and see if it adds up to our target
- Is there exhaustive search algorithm?

```
In [12]: def exhaustiveChange(amount, denominations):
    bestN = 100
    count = [0 for i in range(len(denominations))]
    while True:
        for i, coinValue in enumerate(denominations):
            count[i] += 1
        if (count[i]*coinValue < 100):
            break
        count[i] = 0
        n = sum(count)
        if n == 0:
            break
        value = sum([count[i]*denominations[i] for i in range(len(denominations))])
        if (value == amount):
            if (n < bestN):
                solution = [count[i] for i in range(len(denominations))]
                bestN = n
        return solution

print(exhaustiveChange(40, [1, 5, 10, 20, 25]))

[0, 0, 0, 2, 0]
```
Correct, but costly

- Our algorithm now gets the right answer for every value 1..100
- It must, because it considers every possible answer (that’s the good thing about brute force)
- There is a downside though

In [19]:
%time print(exhaustiveChange(40, [1,5,10,25]))
%time print(exhaustiveChange(40, [1,5,10,20,25]))
%time print(exhaustiveChange(40, [1,5,5,7,11,13]))

[0, 1, 1, 1]
CPU times: user 209 ms, sys: 1e+03 μs, total: 210 ms
Wall time: 209 ms
[0, 0, 0, 2, 0]
CPU times: user 1 s, sys: 1 ms, total: 1.01 s
Wall time: 1 s
[0, 0, 0, 1, 3, 0]
CPU times: user 3min 42s, sys: 3 ms, total: 3min 42s
Wall time: 3min 42s
A Branch-and-bound algorithm

```python
def branchAndBoundChange(amount, denominations):
    bestN = amount
    count = [0 for i in range(len(denominations))]
    while True:
        for i, coinValue in enumerate(denominations):
            count[i] += 1
            if (count[i] * coinValue < amount):
                break
            count[i] = 0
        n = sum(count)
        if n == 0:
            break
        if (n > bestN):
            value = sum([count[i] * denominations[i] for i in range(len(denominations))])
            if (value == amount):
                solution = [count[i] for i in range(len(denominations))]
                bestN = n
        return solution
```

.. Correct, and it works well for many cases, but can be as slow as an exhaustive search for some inputs (try 99).
Is there another Approach?

Tabulating Answers

- If it is costly to compute the answer for a given input, then there may be advantages to caching the result of previous calculations in a table.
- This trades-off time-complexity for space.
- How could we fill in the table in the first place?
- Run our best correct algorithm.
- Can the table itself be used to speed up the process?
Solutions using a Table

- Suppose you are asked to fill-in the unknown table entry for 67¢
- It must differ from a previously known optimal result by at most one coin...
- So what are the possibilities?
  - \( \text{BestChange}(67¢) = 25¢ + \text{BestChange}(42¢) \), or
  - \( \text{BestChange}(67¢) = 20¢ + \text{BestChange}(47¢) \), or
  - \( \text{BestChange}(67¢) = 10¢ + \text{BestChange}(57¢) \), or
  - \( \text{BestChange}(67¢) = 5¢ + \text{BestChange}(62¢) \), or
  - \( \text{BestChange}(67¢) = 1¢ + \text{BestChange}(66¢) \)
Oops… it got slower. Why?
(Not to mention, it found another “different” correct answer.)
Recursion Recalculations

- Recursion often results in many redundant calls
- Even after only two levels of recursion 6 different change values are repeated multiple times
- How can we avoid this repetition?
- Cache precomputed results in a table!

\[
\text{Change}(40) = 25 + \begin{array}{c}
\text{Change}(15) \\
25 + 10 + \text{Change}(5) \\
25 + 5 + \text{Change}(10)
\end{array} \\
20 + \begin{array}{c}
\text{Change}(20) \\
20 + 20 + \text{Change}(0) \\
20 + 10 + \text{Change}(10) \\
20 + 5 + \text{Change}(15)
\end{array} \\
10 + \begin{array}{c}
\text{Change}(30) \\
10 + 25 + \text{Change}(5) \\
10 + 20 + \text{Change}(10) \\
10 + 10 + \text{Change}(20) \\
10 + 5 + \text{Change}(25)
\end{array} \\
5 + \begin{array}{c}
\text{Change}(35) \\
5 + 25 + \text{Change}(15) \\
5 + 20 + \text{Change}(10) \\
5 + 10 + \text{Change}(25) \\
5 + 5 + \text{Change}(30)
\end{array}
\]
When do we fill in the values of the table?

- We could do it lazily as needed... as each call to BestChange() progresses from M down to 1
- Or we could do it from the bottom-up – tabulating all values from 1 up to M
- Thus, instead of just trying to find the minimal number of coins to change M cents, we attempt the solve the superficially harder problem of solving for the optimal change for all values from 1 to M

<table>
<thead>
<tr>
<th>1¢ = [0,0,0,0,1]</th>
<th>2¢ = [0,0,0,0,2]</th>
<th>3¢ = [0,0,0,0,3]</th>
<th>...</th>
<th>M¢ = [?,?,?,?]</th>
</tr>
</thead>
</table>
Change via Dynamic Programming

- BruteForceChange() was $O(d^M)$
- DPChange() is $O(Md)$
A Hybrid Approach: Memoization

- Often we can simply modify a recursive algorithm to “cache” the result of previous invocations
- This “lazy evaluated” form of dynamic programming is often called “Memoization”

```python
In [34]: change = {}

def MemoizedChange(M, c):
    global change
    if (M in change):
        return [v for v in change[M]]
    if (len(change) == 0):
        change[0] = [0 for i in range(len(c))]
        smallestNumberOfCoins = M+1
    for i in range(len(c)):
        if (M == c[i]):
            thisChange = MemoizedChange(M - c[i], c)
            thisChange[i] += 1
            if (sum(thisChange) <= smallestNumberOfCoins):
                bestChange = [v for v in thisChange]
                smallestNumberOfCoins = sum(thisChange)
        change[M] = [v for v in bestChange]
    return bestChange

%time print(MemoizedChange(40, [1,3,5,7,11,13]))
```

```
[1, 0, 0, 0, 0, 3]
CPU times: user 541 µs, sys: 0 ns, total: 541 µs
Wall time: 477 µs
```
Dynamic Programming

- Dynamic Programming is a general technique for computing recurrence relations efficiently by storing partial or intermediate results.
- Three keys to constructing a dynamic programming solution:
  1. Formulate the answer as a recurrence relation
  2. Consider all instances of the recurrence at each step
  3. Order evaluations so you will always have precomputed the needed partial results
- Memoization is an easy way to convert recursive solutions to a DP
- We'll see it again, and again
Next Time

- Back to sequence alignment
- Another algorithm design approach: Divide and Conquer