Returning to Dynamic Programming

**Brute-Force Solution:**
\[ O(n!) \]

**Dynamic Programming Algorithms:**
\[ O(n^2 2^n) \]

**Selling on eBay:**
\[ O(1) \]

Still working on your route?

Shut the hell up!
What is an Algorithm?

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

- An algorithm is correct only if it produces correct result for all input instances.
  - If the algorithm gives an incorrect answer for one or more input instances, it is an incorrect algorithm.
- Coin change problem
  - **Input:** an amount of money $M$ in cents
  - **Output:** the smallest number of coins
- 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

\[
\begin{align*}
M & \rightarrow 100 \\
r & \rightarrow M \\
q & \rightarrow r / 25 \\
r & \rightarrow r - 25 \cdot q \\
d & \rightarrow r / 10 \\
r & \rightarrow r - 10 \cdot d \\
n & \rightarrow r / 5 \\
r & \rightarrow r - 5 \cdot n \\
p & \rightarrow r
\end{align*}
\]

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

```plaintext
M = 40
c = (25, 20, 10, 5, 1)
r = M
n = 0
for k = 1 to d
    i_k = r \\ c_k
    n = n + i_k
    r = r - c_k \times i_k
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?

```python
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(42, [1,5,10,20,25])
[2, 0, 0, 2, 0]
```
Other Tricks?

- 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):
            continue
        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 branchAndBoundChange(42, [1, 5, 10, 20, 25])

[2, 0, 0, 2, 0]
```

- Correct, and works well for most cases, but might be as slow as an exhaustive search for some inputs.
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 previous 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
  - $\text{BestChange}(67¢) = 1¢ + \text{BestChange}(66¢)$
A Recursive Coin-Change Algorithm

```python
def RecursiveChange(M, c):
    if (M == 0):
        return [0 for i in xrange(len(c))]
    smallestNumberOfCoins = M+1
    for i in xrange(len(c)):
        if (M >= c[i]):
            thisChange = RecursiveChange(M - c[i], c)
            thisChange[i] += 1
            if (sum(thisChange) < smallestNumberOfCoins):
                bestChange = thisChange
                smallestNumberOfCoins = sum(thisChange)
    return bestChange

print RecursiveChange(42, [1,5,10,20,25])
```

[2, 0, 0, 2, 0]

- The only problem is... it is too slow
- Let’s see why...
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!

\[
\text{Change}(40) = 25 + \text{Change}(15)
\]
\[
25 + 10 + \text{Change}(5)
\]
\[
25 + 5 + \text{Change}(10)
\]
\[
20 + \text{Change}(20)
\]
\[
20 + 20 + \text{Change}(0)
\]
\[
20 + 10 + \text{Change}(10)
\]
\[
20 + 5 + \text{Change}(15)
\]
\[
10 + \text{Change}(30)
\]
\[
10 + 25 + \text{Change}(5)
\]
\[
10 + 20 + \text{Change}(10)
\]
\[
10 + 10 + \text{Change}(20)
\]
\[
10 + 5 + \text{Change}(25)
\]
\[
5 + \text{Change}(35)
\]
\[
5 + 25 + \text{Change}(15)
\]
\[
5 + 20 + \text{Change}(10)
\]
\[
5 + 10 + \text{Change}(25)
\]
\[
5 + 5 + \text{Change}(30)
\]
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

| 1c = [0,0,0,0.1] | 2c = [0,0,0,0.2] | 3c = [0,0,0,0,3] | ... | Mc = [?,?,?,?] |
def DPChange(M, c):
    change = [[0 for i in xrange(len(c))]]
    for m in xrange(1,M+1):
        bestNumCoins = m+1
        for i in xrange(len(c)):
            if (m >= c[i]):
                thisChange = [x for x in change[m - c[i]]]
                thisChange[i] += 1
                if (sum(thisChange) < bestNumCoins):
                    change[m:m] = [thisChange]
                    bestNumCoins = sum(thisChange)
    return change[M]

print DPChange(42, [1,5,10,20,25])

[2, 0, 0, 2, 0]

- Recall, BruteForceChange() was O(Md)
- DPChange() is O(Md)
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
- We'll see it again, and again
Next Time

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