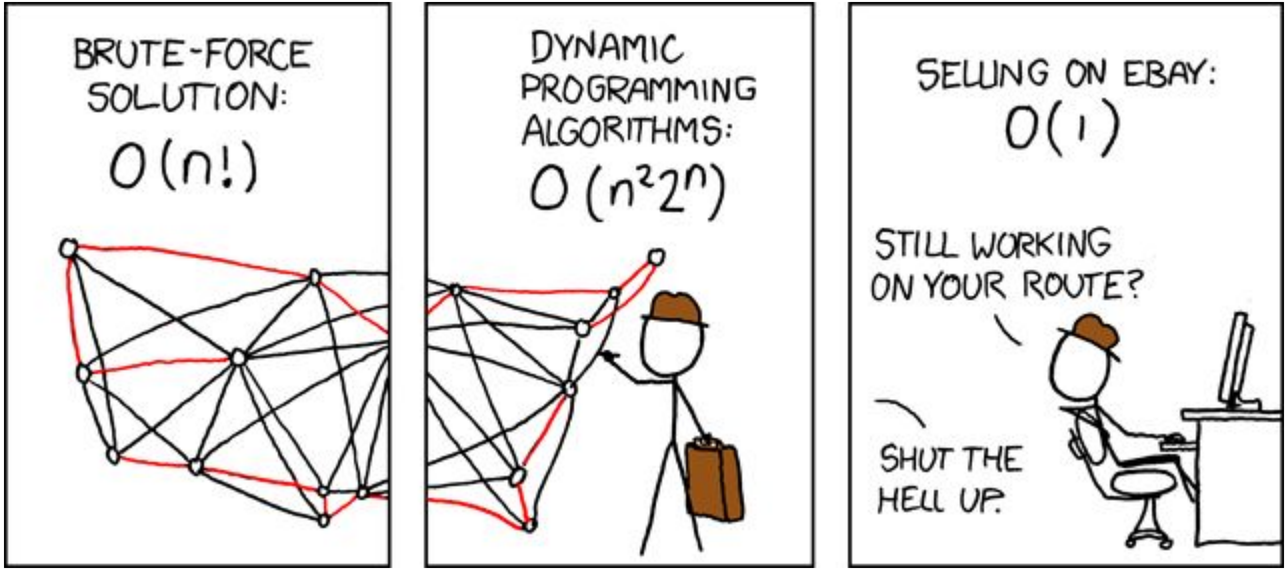
# Comp 555 - BioAlgorithms - Spring 2022



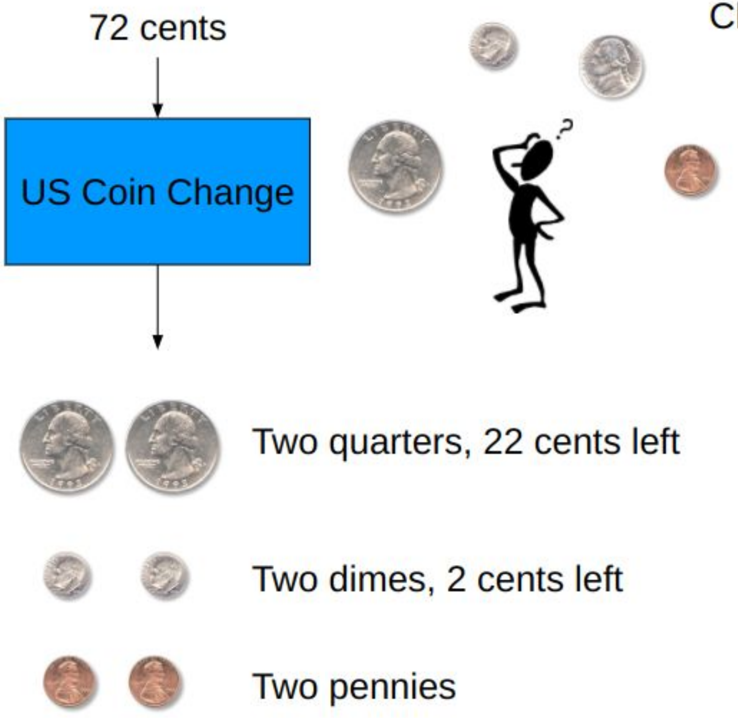

- Midterm next Thursday
- Study session Next Tuesday night?

Adventures in Dynamic Programming

# Algorithm Correctness

- An algorithm is correct only if it produces correct result for every valid 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 $[c_1, c_2, \dots, c_n]$
  - **Output:** the ***smallest number of coins***, t, that add to $M$ (may not be unique)
- US coin change problem

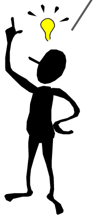
# US Coin Change




**Answer: 2+2+2 = 6 coins**

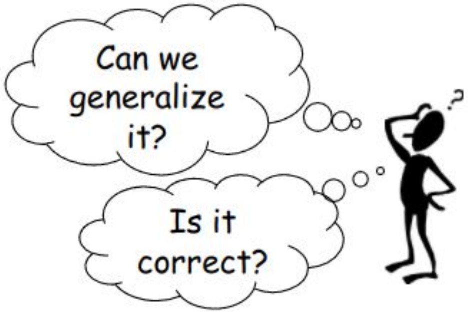
Classic Algorithm

```
r ← M
q ← r // 25
t ← q
r ← r - 25*q
d ← r // 10
t ← t + d
r ← r - 10*q
n ← r // 5
t ← t + n
r ← r - 5*n
t ← t + p
p ← r
```



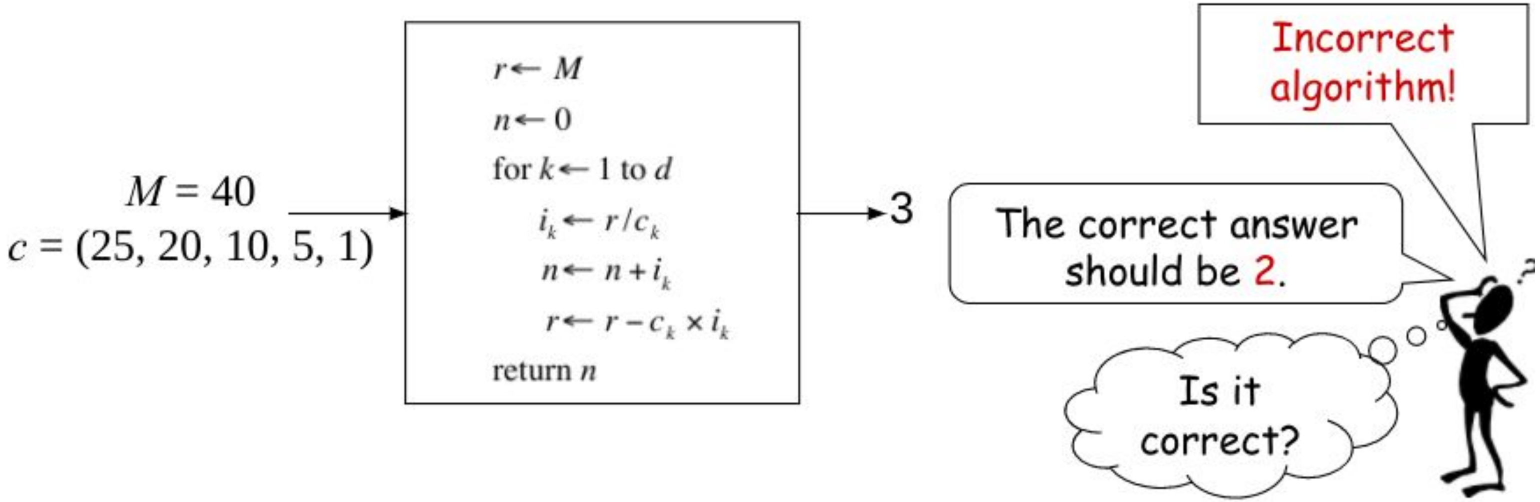
An aside: With this approach you will never output more than 2 dimes, 1 nickel, or 4 pennies.

How do you know this is true?



# 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



# A "Greedy" change approach

- **Key idea: Use as many of the largest available coin denomination so long as the sum is less than or equal to the change amount**

In [3]:

```
def greedyChange(amount, denominations):
    # Goal is to produce the fewest coins to achieve
    # given target "amount"
    # Strategy: Give as many of the largest coin
    # denomination that is less than amount.
    solution = []
    for coin in denominations:
        i = amount // coin        # truncating integer divide
        solution.append(i)
        amount -= coin * i
    return solution

s1 = greedyChange(72, [25,10,5,1])
print(s1, sum(s1))
s2 = greedyChange(40, [25,10,5,1])
print(s2, sum(s2))
s3 = greedyChange(40, [25,20,10,5,1])
print(s3, sum(s3))
```

```
[2, 2, 0, 2] 6
[1, 1, 1, 0] 3
[1, 0, 1, 1, 0] 3
```

# But first a trick

- How to iterate through products like itertools.products()
- How to count

```python
# A trick for creating counting sequences without recursion
# Idea, add one to the first digit until the base is reached
# if the base is reached add one to the next digit and test
# and so on

base = 3
digits = 3
count = [0 for i in range(digits)]
while True:
    print(count)
    for i in range(digits):
        count[i] += 1
        if (count[i] < base):
            break
        count[i] = 0
    if (sum(count) == 0):
        break
```

```
[0, 0, 0]
[1, 0, 0]
[2, 0, 0]
[0, 1, 0]
[1, 1, 0]
[2, 1, 0]
[0, 2, 0]
[1, 2, 0]
[2, 2, 0]
[0, 0, 1]
[1, 0, 1]
[2, 0, 1]
[0, 1, 1]
[1, 1, 1]
[2, 1, 1]
[0, 2, 1]
[1, 2, 1]
[2, 2, 1]
[0, 0, 2]
[1, 0, 2]
[2, 0, 2]
[0, 1, 2]
[1, 1, 2]
[2, 1, 2]
[0, 2, 2]
[1, 2, 2]
[2, 2, 2]
```

# Another Approach?

- Let's bring back brute force
- Test every coin combination (where each denomination is less than 100) to see if it adds up to our target
- Is there exhaustive search algorithm?

In [8]:

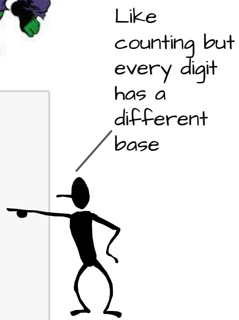
```
def exhaustiveChange(amount, denominations):
    bestN = 100
    count = [0 for i in range(len(denominations))]
    while True:
        for i, coinValue in enumerate(denominations):
            count[i] += 1
            if (count[i]*coinValue < 100):
                break
            count[i] = 0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

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

```
[0, 2, 0, 0, 0]
CPU times: user 688 ms, sys: 0 ns, total: 688 ms
Wall time: 672 ms
```

| | |
|---|---|
| [0,1,2,3] | 25 |
| [0,1,2,3,4] | 20 |
| [0,...,9] | 10 |
| [0,...,19] | 5 |
| [0,...,99] | 100 |

4*5*10*20*100 = 400000



# 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 [16]:
```
%time print(exhaustiveChange(40, [25,10,5,1]))
%time print(exhaustiveChange(40, [25,20,10,5,1]))
%time print(exhaustiveChange(40, [13,11,7,5,3,1]))
```

```
[1, 1, 1, 0]
CPU times: user 155 ms, sys: 0 ns, total: 155 ms
Wall time: 149 ms
[0, 2, 0, 0, 0]
CPU times: user 632 ms, sys: 0 ns, total: 632 ms
Wall time: 628 ms
[0, 3, 1, 0, 0, 0]
CPU times: user 2min 50s, sys: 0 ns, total: 2min 50s
Wall time: 2min 50s
```

# Other tricks?

A Branch-and-bound algorithm, almost identical to brute force

In [17]:

```
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):              # Set upper bound to amount rather than 100
                break
            count[i] = 0
        n = sum(count)
        if n == 0:
            break
        if (n > bestN):                                    # don't compute the amount if there are too many coins
            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

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

```
[0, 3, 1, 0, 0, 0]
CPU times: user 317 ms, sys: 0 ns, total: 317 ms
Wall time: 299 ms
```

BTW, this is an iterative branch-and-bound... meaning that branch-and-bound can be implemented without recursion

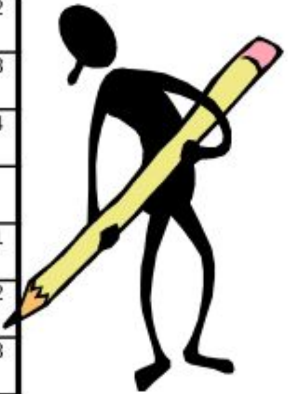
..Correct, and it works well for many cases, but can be as slow as an exhaustive search for some inputs (try 99).

# Is there another Approach?

Tabulating Answers

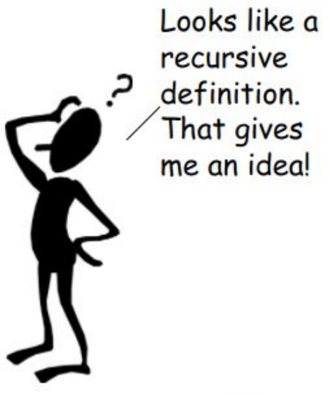
- If it is costly to compute the answer for a given input, then there may be advantages to caching the result of previous calculations in a table
- This trades-off time-complexity for space
- How could we fill in the table in the first place?
- Run our best correct algorithm
- Can the table itself be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

| Amt | 25 | 20 | 10 | 5 | 1 | Amt | 25 | 20 | 10 | 5 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1¢ | | | | | 1 | 42¢ | | 2 | | | 2 |
| 2¢ | | | | | 2 | 43¢ | | 2 | | | 3 |
| 3¢ | | | | | 3 | 44¢ | | 2 | | | 4 |
| 4¢ | | | | | 4 | 45¢ | | 2 | | 1 | |
| 5¢ | | | | 1 | | 46¢ | | 2 | | 1 | 1 |
| 6¢ | | | | 1 | 1 | 47¢ | | 2 | | 1 | 2 |
| 7¢ | | | | 1 | 2 | 48¢ | | 2 | | 1 | 3 |
| 8¢ | | | | 1 | 3 | 49¢ | | 2 | | 1 | 4 |
| 9¢ | | | | 1 | 4 | 50¢ | 2 | | | | |
| 10¢ | | | 1 | | | 51¢ | 2 | | | | 1 |
| 11¢ | | | 1 | | 1 | 52¢ | 2 | | | | 2 |



# 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?
  - BestChange(67¢) = 1 + BestChange(67-25 = 42¢), or
  - BestChange(67¢) = 1 + BestChange(67-20 = 47¢), or
  - BestChange(67¢) = 1 + BestChange(67-10 = 57¢), or
  - BestChange(67¢) = 1 + BestChange(67-5 = 62¢), or
  - BestChange(67¢) = 1 + BestChange(65-1 = 66¢)
- **The minimum of these must be the right answer**



# A Recursive Coin-Change Algorithm

```
In [23]: def RecursiveChange(M, c):
             if (M == 0):
                 return [0 for i in range(len(c))]
             smallestNumberOfCoins = M+1
             for i in range(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

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

         [1, 0, 0, 0, 0, 3]
         CPU times: user 6min 43s, sys: 16 ms, total: 6min 43s
         Wall time: 6min 43s
```

Oops… it got slower. Why?
(Not to mention, it found another "different" correct answer.)

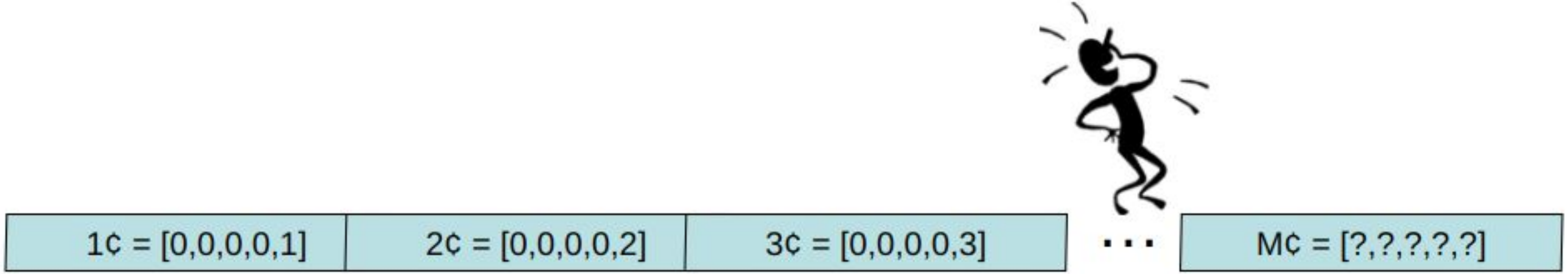
# Recursion Recalculations

- 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!

Change(40) = 25 + Change(15)
25 + 10 + Change(5)
25 + 5 + Change(10)
20 + Change(20)
20 + 20 + Change(0)
20 + 10 + Change(10)
20 + 5 + Change(15)
10 + Change(30)
10 + 25 + Change(5)
10 + 20 + Change(10)
10 + 10 + Change(20)
10 + 5 + Change(25)
5 + Change(35)
5 + 25 + Change(15)
5 + 20 + Change(10)
5 + 10 + Change(25)
5 + 5 + Change(30)

# Back to Table Evaluation

- When do we fill in the values of our table?
- We could solve for change for every value from 1 up to M, then we'd be guaranteed to have found the best change for any value less than M when needed
- 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



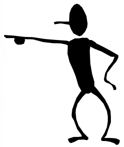
# Change via Dynamic Programming

```
In [21]: def DPChange(M, c):
             change = [[0 for i in range(len(c))]]
             for m in range(1,M+1):
                 bestNumCoins = m+1
                 for i in range(len(c)):
                     if (m >= c[i]):
                         thisChange = [x for x in change[m - c[i]]]
                         thisChange[i] += 1
                         if (sum(thisChange) < bestNumCoins):
                             bestCombination = [v for v in thisChange]
                             bestNumCoins = sum(thisChange)
                 change.append(bestCombination)
             return change[M]

         %time print(DPChange(40, [1,3,5,7,11,13]))
         %time print(DPChange(40, [1,3,5,7,11,13,17]))
         %time print(DPChange(40, [1,3,5,7,11,13,17,19]))
         %time print(DPChange(42, [13,11,7,5,3,1]))
```

```
0 [0, 0, 0, 0, 0, 0]
1 [1, 0, 0, 0, 0, 0]
2 [2, 0, 0, 0, 0, 0]
3 [0, 1, 0, 0, 0, 0]
4 [1, 1, 0, 0, 0, 0]
5 [0, 0, 1, 0, 0, 0]
6 [1, 0, 1, 0, 0, 0]
7 [0, 0, 0, 1, 0, 0]
8 [1, 0, 0, 1, 0, 0]
9 [2, 0, 0, 1, 0, 0]
10 [0, 1, 0, 1, 0, 0]
11 [0, 0, 0, 0, 1, 0]
12 [1, 0, 0, 0, 1, 0]
13 [0, 0, 0, 0, 0, 1]
14 [1, 0, 0, 0, 0, 1]
15 [2, 0, 0, 0, 0, 1]
16 [0, 1, 0, 0, 0, 1]
17 [1, 1, 0, 0, 0, 1]
18 [0, 0, 1, 0, 0, 1]
19 [1, 0, 1, 0, 0, 1]
20 [0, 0, 0, 1, 0, 1]
21 [1, 0, 0, 1, 0, 1]
22 [0, 0, 0, 0, 2, 0]
23 [1, 0, 0, 0, 2, 0]
24 [0, 0, 0, 0, 1, 1]
25 [1, 0, 0, 0, 1, 1]
26 [0, 0, 0, 0, 0, 2]
27 [1, 0, 0, 0, 0, 2]
28 [2, 0, 0, 0, 0, 2]
29 [0, 1, 0, 0, 0, 2]
30 [1, 1, 0, 0, 0, 2]
31 [0, 0, 1, 0, 0, 2]
32 [1, 0, 1, 0, 0, 2]
33 [0, 0, 0, 1, 0, 2]
34 [1, 0, 0, 1, 0, 2]
35 [0, 0, 0, 0, 2, 1]
36 [1, 0, 0, 0, 2, 1]
37 [0, 0, 0, 0, 1, 2]
38 [1, 0, 0, 0, 1, 2]
39 [0, 0, 0, 0, 0, 3]
40 [1, 0, 0, 0, 0, 3]
```

The contents of change just before the return after the first call



- 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
- Filling in the table lazily as needed… as each call to progresses from M down to 1
- This “*lazy evaluated*” form of dynamic programming is often called “*Memoization*”

In [34]:

```
change = {}                                              # This is a cache for saving bestChange[M]

def MemoizedChange(M, c):
    global change
    if (M in change):                                     # Check the cache first
        return [v for v in change[M]]
    if (len(change) == 0):                                # Initialize cache
        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]                   # Add new M to cache
    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

- On to sequence alignment
- But first we'll learn how to navigate in Manhattan