The Change Making Problem: A Lens into Dynamic Programming & Greedy Algorithms

CSCI-432: Advanced Algorithms

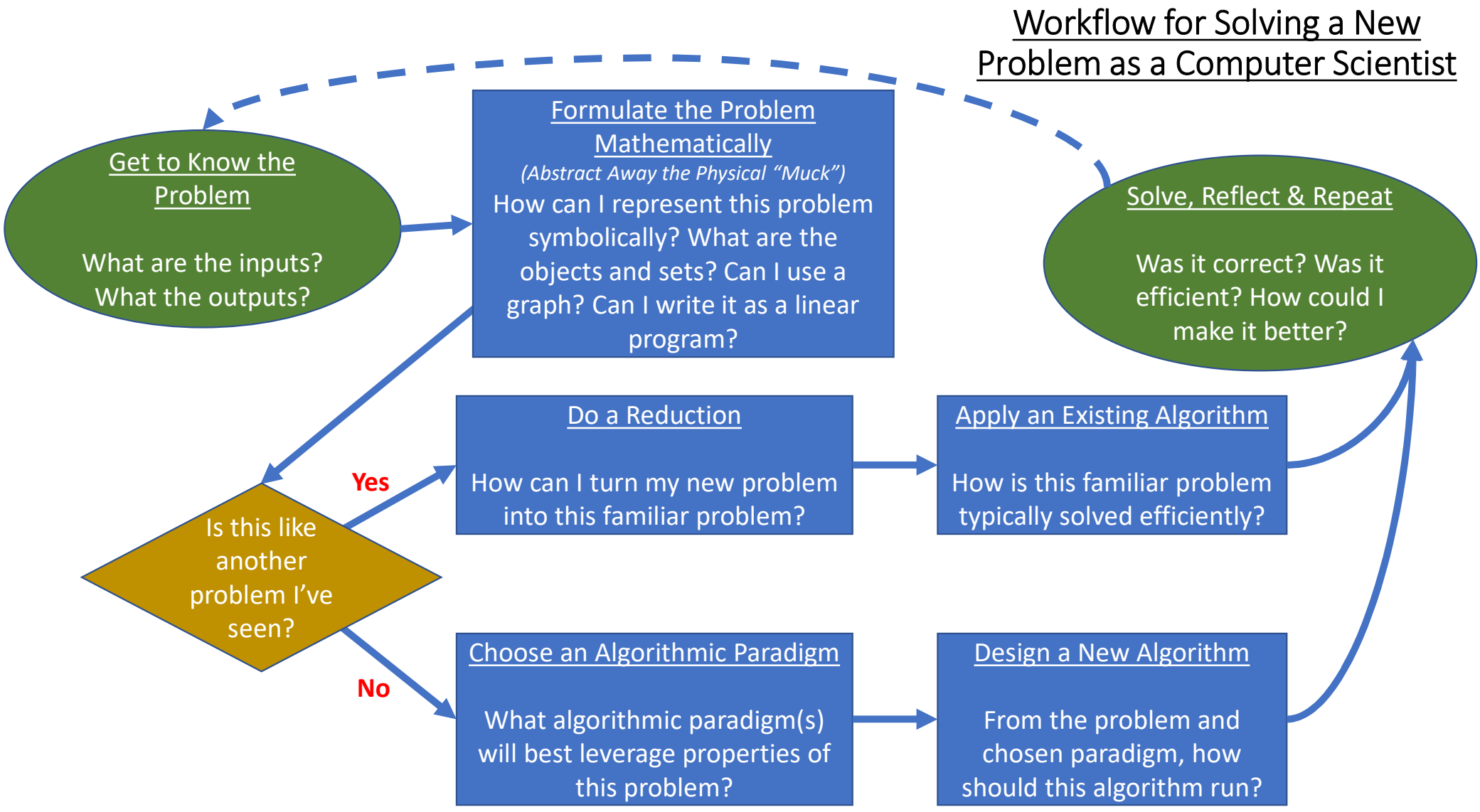
Fall 2021

Dalton Gomez

(With a big shoutout to Sean Yaw for providing me with a lot of content for this presentation)

General Q&A or Feedback

A Quick Review of Recursion


Workflow for Solving a New Problem as a Computer Scientist
Get to Know the Problem
What are the inputs? What the outputs?
Formulate the Problem Mathematically
(Abstract Away the Physical "Muck")
How can I represent this problem symbolically? What are the objects and sets? Can I use a graph? Can I write it as a linear program?
Solve, Reflect & Repeat
Was it correct? Was it efficient? How could I make it better?
Is this like another problem I've seen?
Yes
No
Do a Reduction
How can I turn my new problem into this familiar problem?
Apply an Existing Algorithm
How is this familiar problem typically solved efficiently?
Choose an Algorithmic Paradigm
What algorithmic paradigm(s) will best leverage properties of this problem?
Design a New Algorithm
From the problem and chosen paradigm, how should this algorithm run?

Some Algorithmic Paradigms:
Tools in a Computer Scientists Toolbox

- An algorithmic paradigm is a model for designing and classifying algorithms.
  - Think of paradigms as the inherited class above any specific algorithm (i.e. an "IS A" relationship).
- An algorithm can inherit multiple paradigms.
  - For example, QuickSort is both a divide & conquer and a randomized algorithm.
- As recursion and iteration are computationally equivalent, any of these paradigms can be implemented recursively or iteratively.
- This list is not exhaustive- it is only the paradigms relevant to this lecture.

Paradigm	Description	Pros & Cons	Example Algorithm
Brute Force	Create all possible solutions to the problem and select the best	Correct and easy but very slow	Linear Search
Divide & Conquer	Breaking the problem into ***independent (non-overlapping)*** subproblems, solving each, and then recombing into a solution	Correct and faster but requires that the problem can be broken up into subproblems that are independent	Merge Sort
Dynamic Programming	Breaking the problem into ***dependent (overlapping)*** subproblems and solving them in order of their dependencies to arrive at a solution	Correct and faster but requires that the problem has an ***optimal substructure***, where the optimal solution to one subproblem depends upon another	Longest Increasing Subsequence
Greedy Algorithms	Solving the problem by iteratively selecting the best "here-and-now" decisions (local optima) to arrive at a final global optimum. Note this also requires ***dependent (overlapping)*** subproblems.	Fastest (typically) but can be difficult (or impossible) to define a correct greedy choice, often leading to suboptimal solutions	Dijkstra's Shortest Path

Introducing the Change Making Problem

Imagine that you just got hired as a cashier at <INSERT BZN STORE>. A customer buys a <THING>. Their total is $10.63, and they pay with $11.00, leaving $0.37 as change.

What procedure do you do to count out the 37 cents in the least number of coins as possible?

Unfortunately, this isn't just a thought experiment… This is the world we live in…

Photo taken at the Town Pump on N 19th and Baxter…

September 25th, 2021!!!

The Change Making Problem (More Formally)

- **Input Parameters:**
  - $p =$ the amount of change to make
  - $D = \{d_1, d_2, \ldots, d_n\}$
    - The denomination set of all available coin values.
  - $d_i =$ a denomination value (coin) available in the currency
- **Decision Variables:**
  - $x_i =$ the number of coins of d_i used in the optimal solution
- **Objective:**
  - $\mathrm{C(p)} = \min\left(\sum_{i=1}^{n} x_i\right)$
    - Minimize the total number of coins, $\mathrm{C(p)}$, used in the optimal solution.
- **Constraints:**
  - $\sum_{i=1}^{n} d_i x_i = p$
    - The total value of all the coins must exactly equal the amount of change to make.
  - $x_i \in \mathbb{Z}^+, \forall\, d_i \in D$
    - The number of coins for each denomination must be a positive integer.
    - There is an infinite number of each coin denomination in the "cash drawer."

- **Example:**
  - $p = 37¢$
  - $D = \{1¢, 5¢, 10¢, 25¢\}$
    - Pennies, nickels, dimes, and quarters only!

Write down pseudocode for how you would solve this problem.

If you finish early, analyze the run time of your algorithm. Then analyze its correctness.

The Change Making Problem Solved(?)

- What did you come up with?
- My Hypothesis:
  - Everyone provided a ***greedy algorithm*** to solve this problem!
- Does your algorithm:
  - Start at the amount of change to be made and iterative or recursively work down to zero?
  - Use as many of the largest coin as possible and then repeat iteratively/recursively?

Pseudocode for Greedy Change:

```
greedyChange(p, D)
    X[0,…,n] = [0,…,0]
    i = n
    for d_i to d_1
        x_i = ⌊p/d_i⌋
        p = p % d_i
    return sum(X)
```

Initialize an array for the amounts of each coin

Start at the largest denomination and work down

Use as many of that coin as you can via the floor of the quotient

Update the remaining amount via the mod function

Return the total sum of coins used

Great!

But is it fast? Okay… how fast?

Is it correct? Okay… how correct?

The Change Making Problem Solved(?)

- **How fast?**
  - $\text{Run Time} = O(|D|)$
    - Where |D| is the number of available denominations in the currency
- **How correct?**
  - Well… what is the best way to make change for 37 cents?
    - $\text{1-Quarter} + \text{1-Dime} + \text{2-Pennies} = 37 \text{ cents}$
      - Can you beat this??? No! Must be good, right?
- **Consider the following instances:**
  - **Instance 1:**
    - P = 6
    - D = {1, 3, 4}
  - **Instance 2:**
    - $P = 455$
    - $D = \{1, 4, 7, 13, 28, 52, 91, 365\}$

```
greedyChange(p, D)
    x[0,…,n] = [0,…,0]
    i = n
    for d_i to d_1
        x_i = ⌊p/d_i⌋
        p = p % d_i
    return sum(X)
```

Initialize an array for the amounts of each coin

Start at the largest denomination and work down

Use as many of that coin as you can via the floor of the quotient

Update the remaining amount via the modulo function

Return the total sum of coins used

Greedy = 3 coins ⇒ {4, 1, 1}
Optimal = 2 coins ⇒ {3, 3}

These proofs by counterexample show that greedyChange is not optimal over all instances of the Change Making Problem!

Greedy = 7 coins ⇒ {365, 52, 28, 7, 1, 1, 1}
Optimal = 5 coins ⇒ {91, 91, 91, 91, 91}

Yeah, but just don't create dumb currency systems and we're good, right?

Canonical vs. Non-Canonical Currency Systems

- Let **`greedyChange`** be the greedy algorithm described previously. Let **`optimalChange`** be an algorithm (yet to be shown) that always returns the minimum number of coins needed to make change on all denomination sets.
- A currency system is said to be ***canonical*** if and only if:

  $\forall\ p \in \mathbb{N},\ \texttt{greedyChange(p)} = \texttt{optimalChange(p)}$

- A currency system is said to be ***non-canonical*** if and only if:

  $\exists\ p \in \mathbb{N},\ \text{s.t.}\ \texttt{greedyChange(p)} > \texttt{optimalChange(p)}$

- Over human history, we've created non-canonical currency systems. Therefore, all new and existing currency systems are now developed to be canonical.
- A telling example of this is the pre-1971 English currency system, where:

  $$D = \{\tfrac{1}{2}, 1, 3, 6, 12, 24, 30, 60, 240\}$$

  (all expressed in units of a British penny)

- Read more here: *Canonical Coin Systems for the Change Making Problem* by Xuan Cai

Implications Much Beyond Currency Systems…

- Don't write this problem off as niche or too trivial!!!
- "Change Making" is a placeholder name for any problem where you have a set of objects, each with a cost of 1 and a unique value, and you need to meet or exceed a target value while minimizing cost.
- Change Making is a special instance of the "Knapsack Problem," where you have a set of objects, each with a unique cost and value, and you want to minimize cost while meeting a value or maximize value while meeting a cost.
- Often ***BIG*** optimization problems get elementary names, but those elementary instances are just familiar realizations of an abstract problem with broad applications.
- Organizations make decisions daily that reduce to Change Making/Knapsack problems, and each come with ***BIG*** implications!
- Essentially, optimization problems like these ask, the fundamental question:

  How can I do the best I can with what I have?

- So when faced with a problem, make sure to abstract away the physical manifestations (e.g. let the "coins" become "objects") and ask yourself, "Abstractly, what does this problem do?" Then when you find another problem, involving "vehicles" for example, and you abstract those vehicles to "objects" and, if those objects behave just like the coin objects from the Change Making problem, then you already have your algorithm.

  You own a transportation company with several cars, trucks, vans, and buses, each of which can hold a unique number of people. You must shuttle 123 people for an event. Minimize the total number of drivers (i.e. vehicles) you send.

What is Dynamic Programming?

- Developed by Richard Bellman in the 1950's
- Named so to appease the political pressures of the time.

> "Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."
>
> — *Richard Bellman, Eye of the Hurricane: An Autobiography (1984, page 159)*

- Dynamic programming *really* is:
  - ***"Smart recursion"*** or ***"recursion with memory"***
- You solve the simplest subproblems first, save the answers, and then solve all the subproblems that depend upon those previous problems by looking up previous answers when needed. This is repeated until you reach the solution.
- Central Tenant of Dynamic Programming: ***Leverage the optimal substructure!***
  - **Optimal Substructure –** When the answer to a problem depends upon the answer(s) to a subproblem(s), and so on and so forth.

Steps to Writing a Dynamic Program

1. Identify Optimal Substructure
2. Formulate Recursively
3. Identify a Memoization Structure & Dependencies
4. Write an Algorithm
5. Analyze Complexity (Running Time & Space)
6. Prove Correctness

Adapted from Erikson's Algorithms Textbook, Chapter 4, pages 106-107

1. Identify Optimal Substructure

Central Tenant of DP: *Leverage the optimal substructure!*

Someone calls you into a mysterious and smokey room in the back of your local Town Pump and says,

"2 buttons + 4 "cool" rocks + 1 broom are the smallest number of items possible to get to $10. One broom costs $3."

What can you conclude?

2 buttons + 4 "cool" rocks are the smallest number of items possible to get to $7.

If I have a set of objects that represent the minimum way to achieve a value, and I remove one thing, then I now how a new set that represents the minimum way to achieve the new value (i.e. the old value – the value of the removed object).

2. Formulate Recursively

- **Input Parameters:**
  - $p =$ the amount of change to make
  - $D = \{d_1, d_2, \ldots, d_n\}$
  - $d_i =$ a denomination value (coin) available in the currency
- **Decision Variables:**
  - $x_i =$ the number of coins of d_i used in the optimal solution
- **Objective:**
  - $\mathrm{C(p)} = \min\left(\sum_{i=1}^{n} x_i\right)$
- **Constraints:**
  - $\sum_{i=1}^{n} d_i x_i = p$
  - $x_i \in \mathbb{Z}^+, \forall\, d_i \in D$

Can we characterize $C(p)$ in terms of $C(p - d_i)$?

$$C(p) = 1 + C(p - d_i)$$

The cost of making change for p is equal to the cost of 1 coin plus the cost of making change for $p - d_i$, the amount minus the coin you just added.

2. Formulate Recursively

- **Input Parameters:**
  - p = the amount of change to make
  - $D = \{d_1, d_2, \ldots, d_n\}$
  - d_i = a denomination value (coin) available in the currency
- **Decision Variables:**
  - x_i = the number of coins of d_i used in the optimal solution
- **Objective:**
  - $\mathrm{C(p)} = \min\left(\sum_{i=1}^{n} x_i\right)$
- **Constraints:**
  - $\sum_{i=1}^{n} d_i x_i = p$
  - $x_i \in \mathbb{Z}^+, \forall\, d_i \in D$

Can we characterize $C(p)$ in terms of $C(p - d_i)$?

$$C(p) = 1 + C(p - d_i)$$

The cost of making change for p is equal to the cost of 1 coin plus the cost of making change for $p - d_i$, the amount minus the coin you just added.

How do we know account for finding the optimal solution?

Example: Let p = 19 cents.

C(p) = 1 + minimum of:

- least change for 19-10 = 9 cents
- least change for 19-5 = 14 cents
- least change for 19-1 = 18 cents

$$C(p) = \begin{cases} \min\limits_{i:d_i \leq p} C(p - d_i) + 1, p > 0 \\ 0, p = 0 \end{cases}$$

Making Change with Brute Force Recursion

```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

Making Change with Brute Force Recursion

```
change(p)
   if p == 0
      return 0
   else
      min = ∞
      for d_i ≤ p
         a = change(p-d_i)
         if a < min
            min = a
      return 1 + min
```

Running time?

Making Change with Brute Force Recursion

```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

Running time?

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for

Making Change with Brute Force Recursion

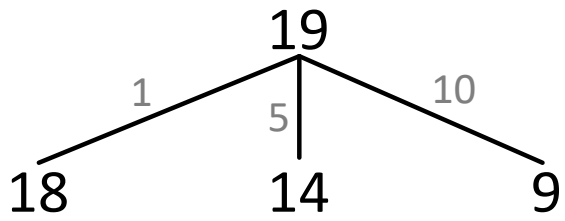
```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

Running time?

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



Making Change with Brute Force Recursion

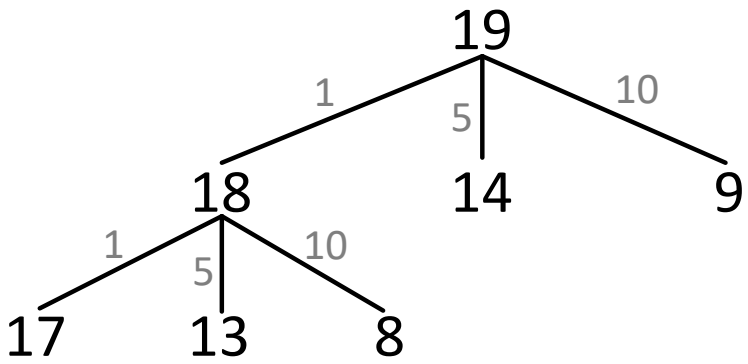
```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

Running time?

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



Making Change with Brute Force Recursion

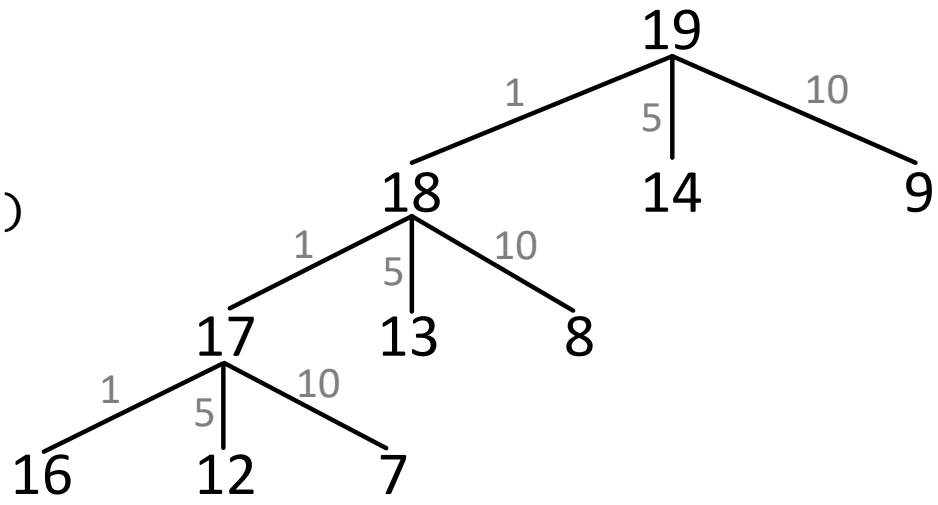
```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

Running time?

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



Making Change with Brute Force Recursion

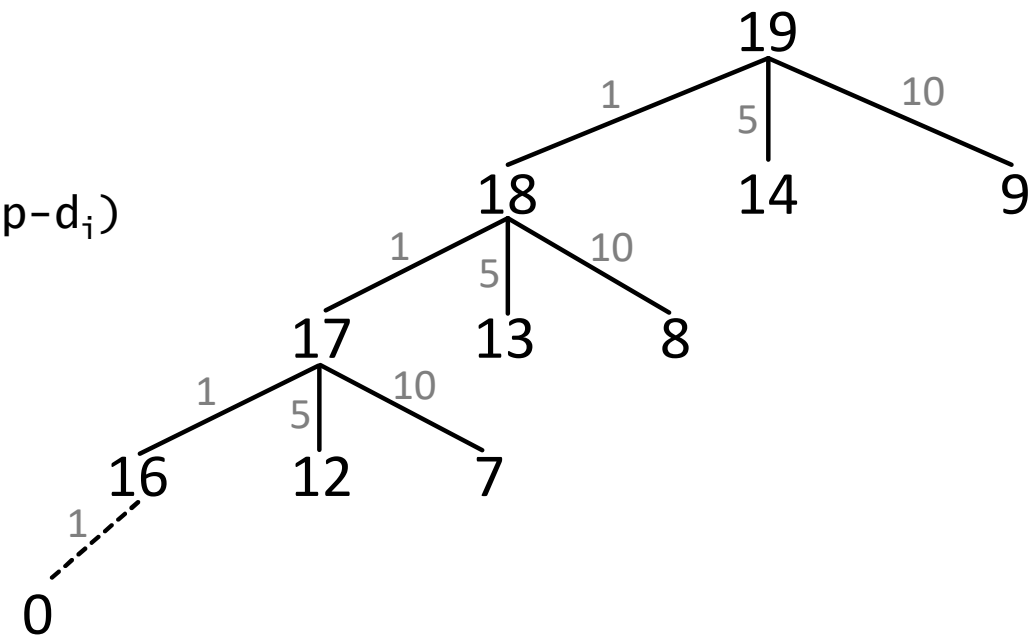
```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

Running time?

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



Making Change with Brute Force Recursion

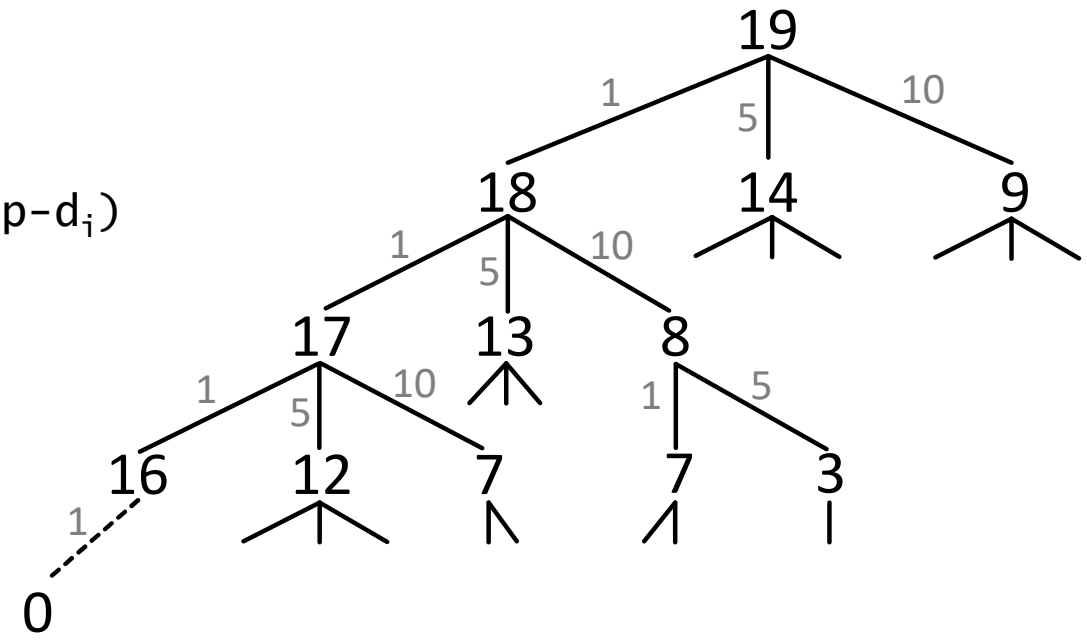
```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

Running time?

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



Making Change with Brute Force Recursion

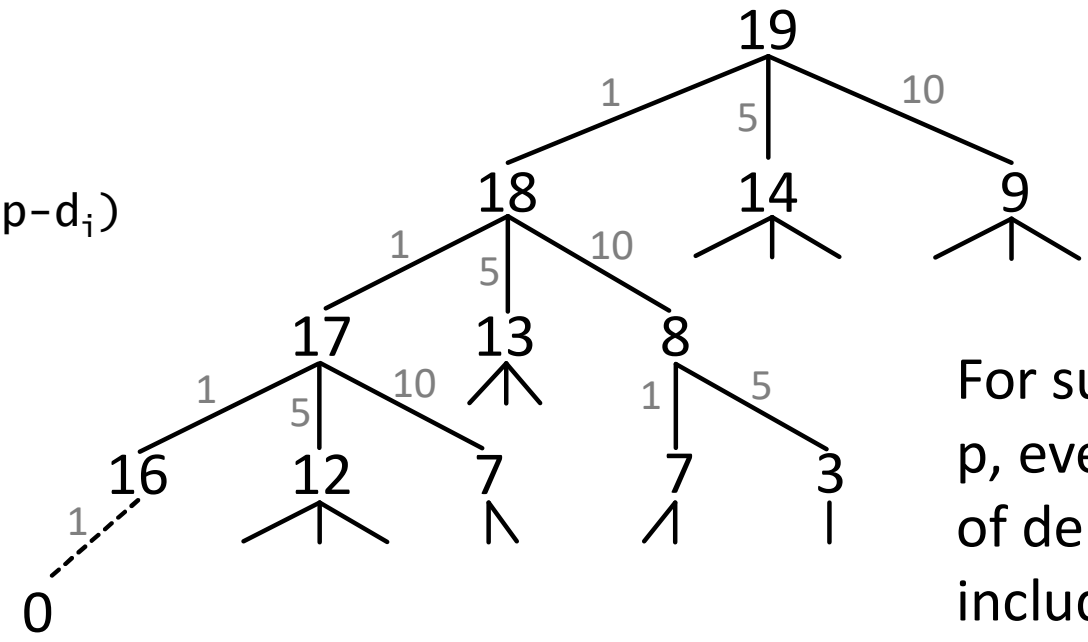
```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

Running time?

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



For sufficiently large p, every permutation of denominations is included.

Making Change with Brute Force Recursion

```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

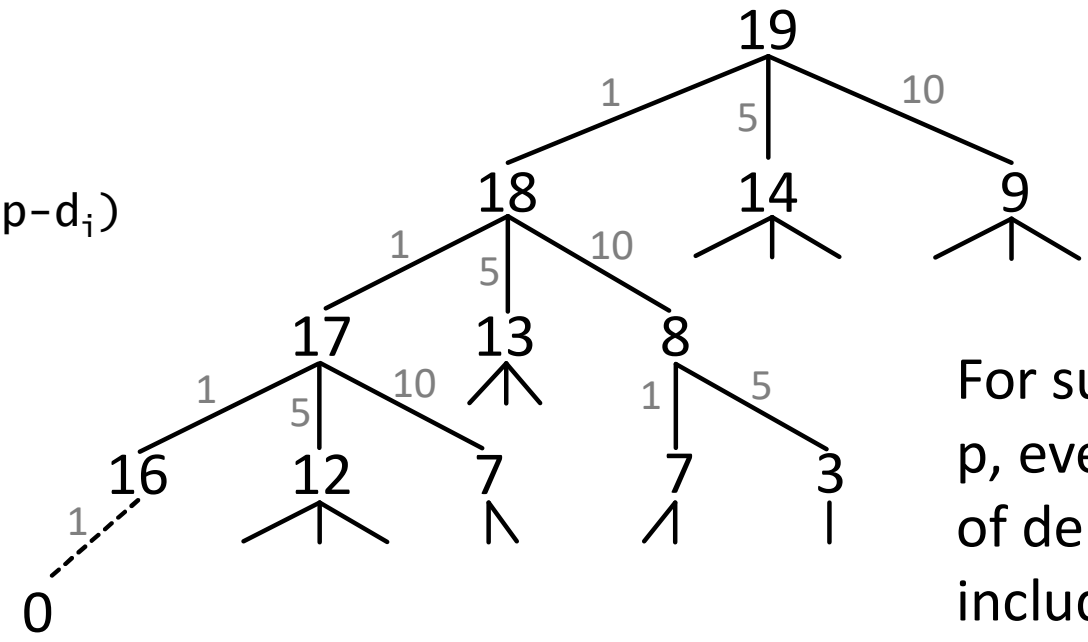
Running time?

$k!$

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



For sufficiently large p, every permutation of denominations is included.

Making Change with Brute Force Recursion

```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

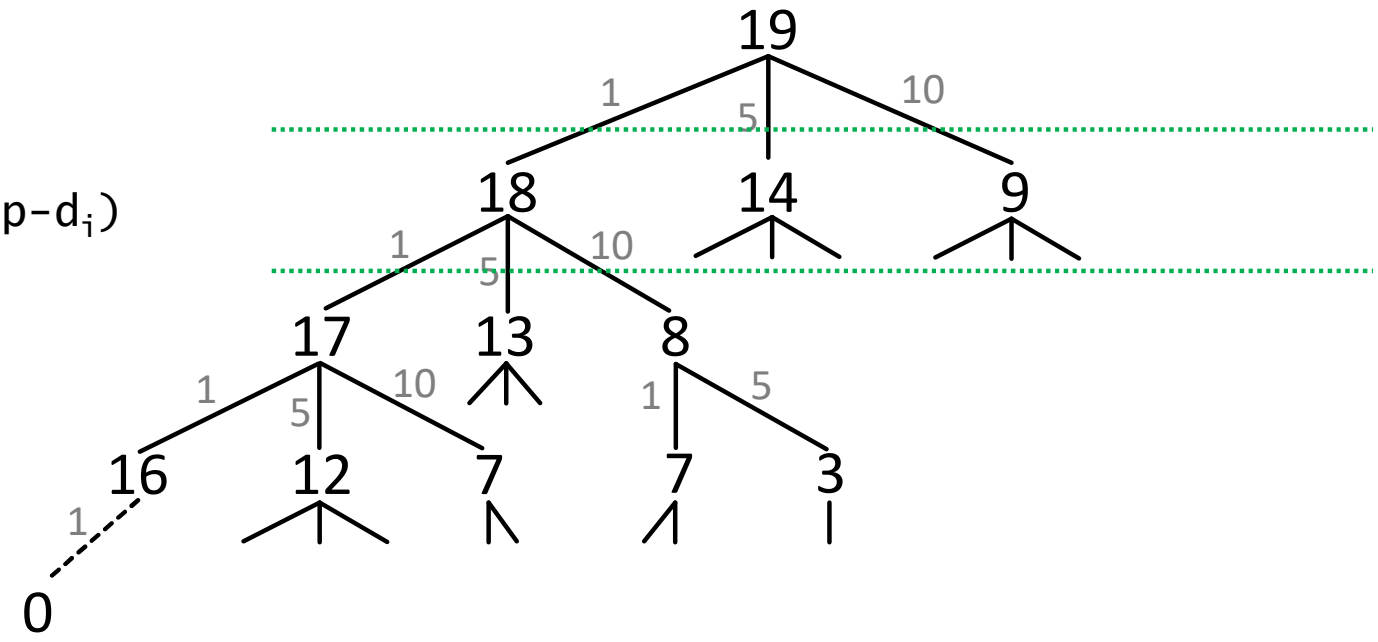
Running time?

$k!$

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



Making Change with Brute Force Recursion

```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

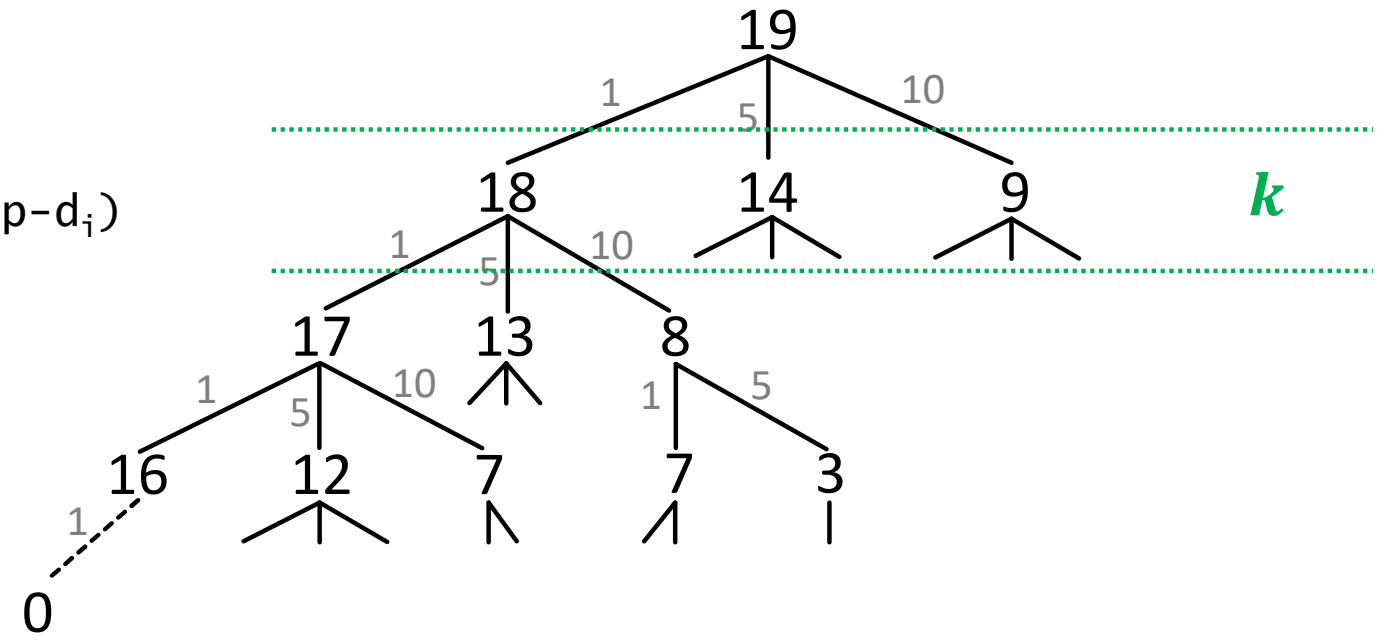
Running time?

$k!$

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



Making Change with Brute Force Recursion

```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

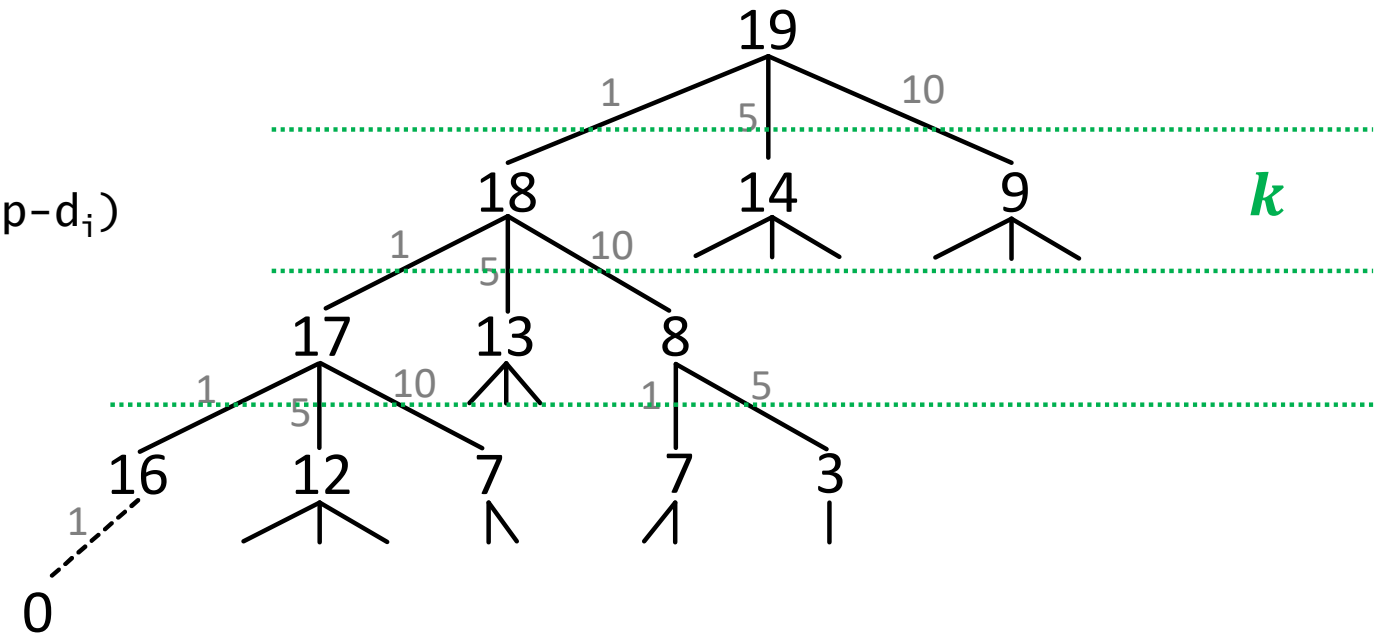
Running time?

$k!$

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



Making Change with Brute Force Recursion

```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

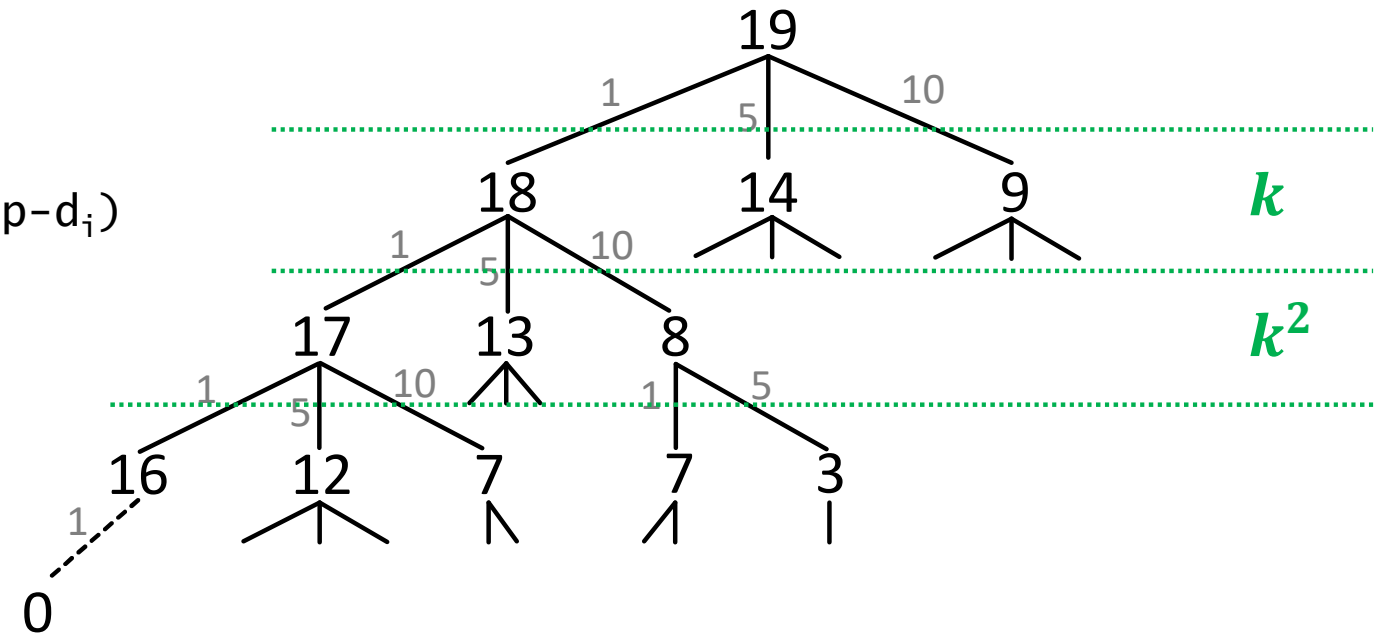
Running time?

$k!$

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



Making Change with Brute Force Recursion

```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

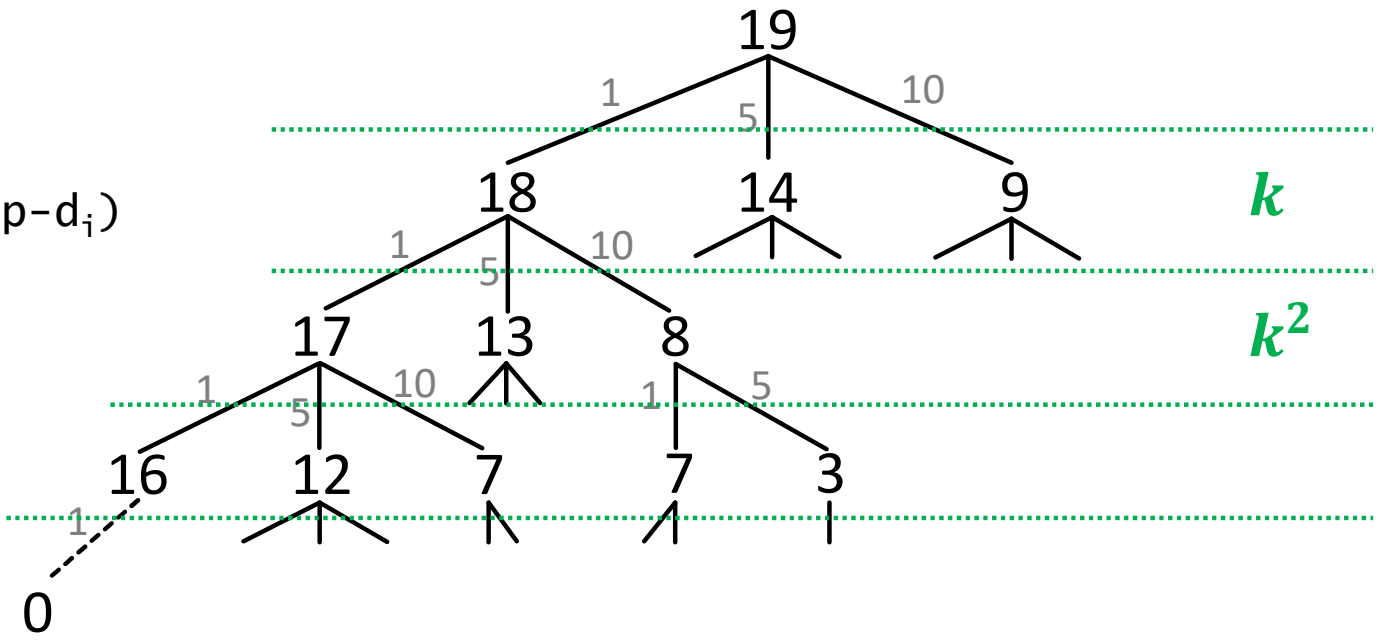
Running time?

$k!$

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



Making Change with Brute Force Recursion

```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

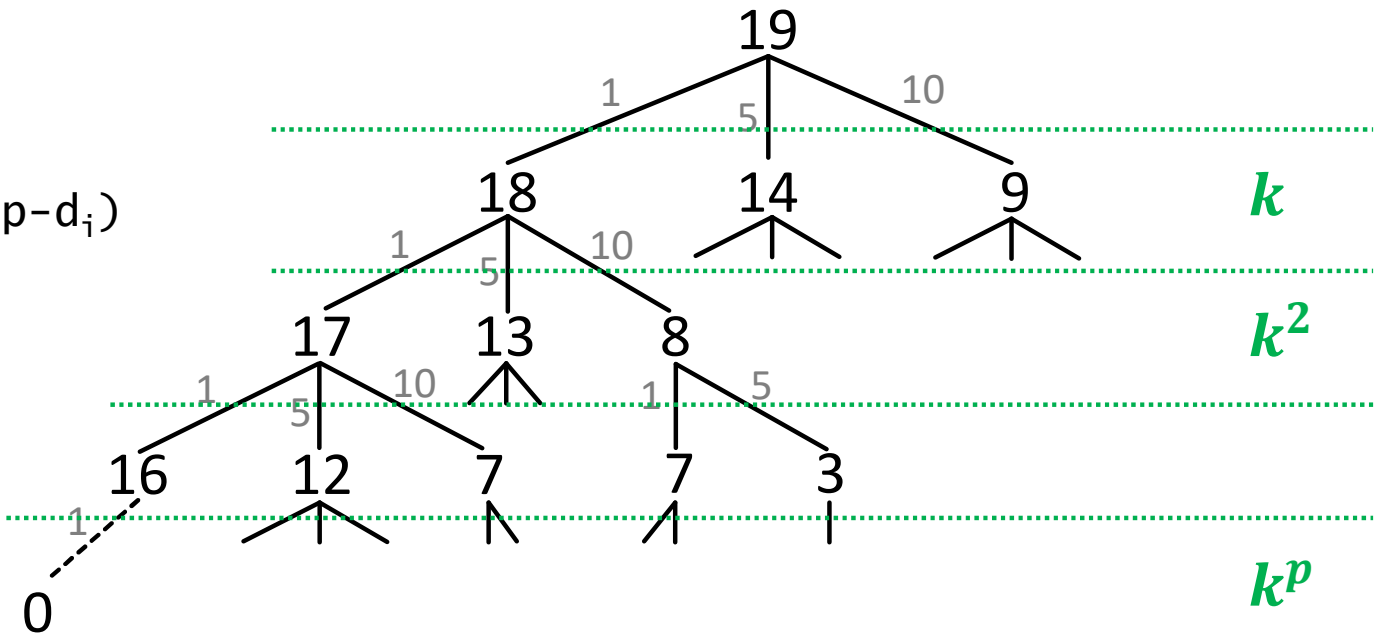
Running time?

$k!$

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



Making Change with Brute Force Recursion

```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

Running time?

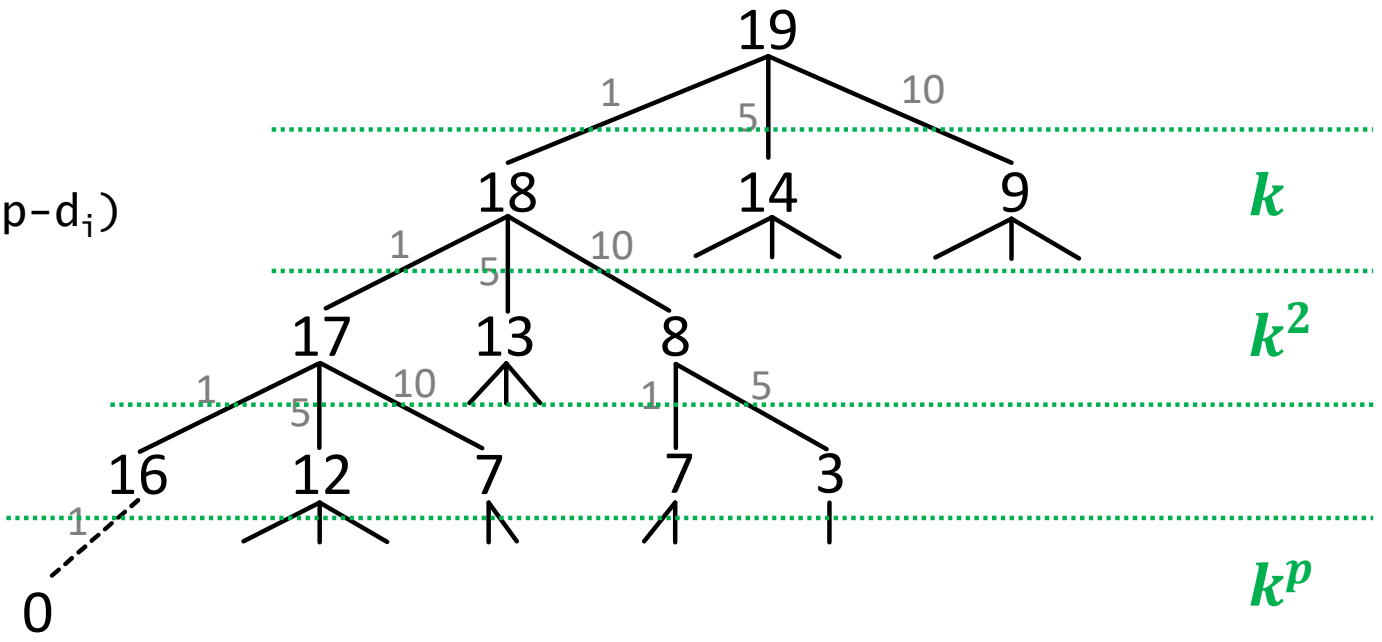
$k!$

k^p

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



Making Change with Brute Force Recursion

```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

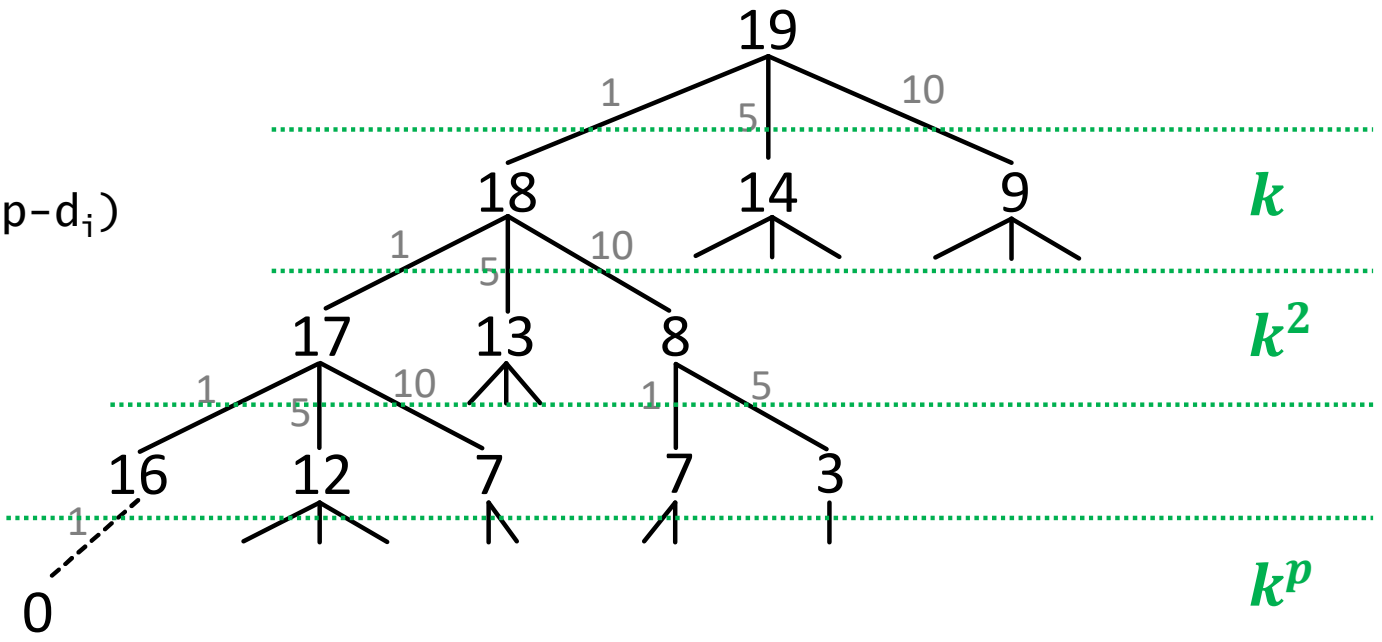
Running time?

$$\frac{k!}{k^p} \in \Omega(\text{scary})$$

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



Making Change with Brute Force Recursion

```
change(p)
    if p == 0
        return 0
    else
        min = ∞
        for all d_i ≤ p
            a = change(p-d_i)
            if a < min
                min = a
        return 1 + min
```

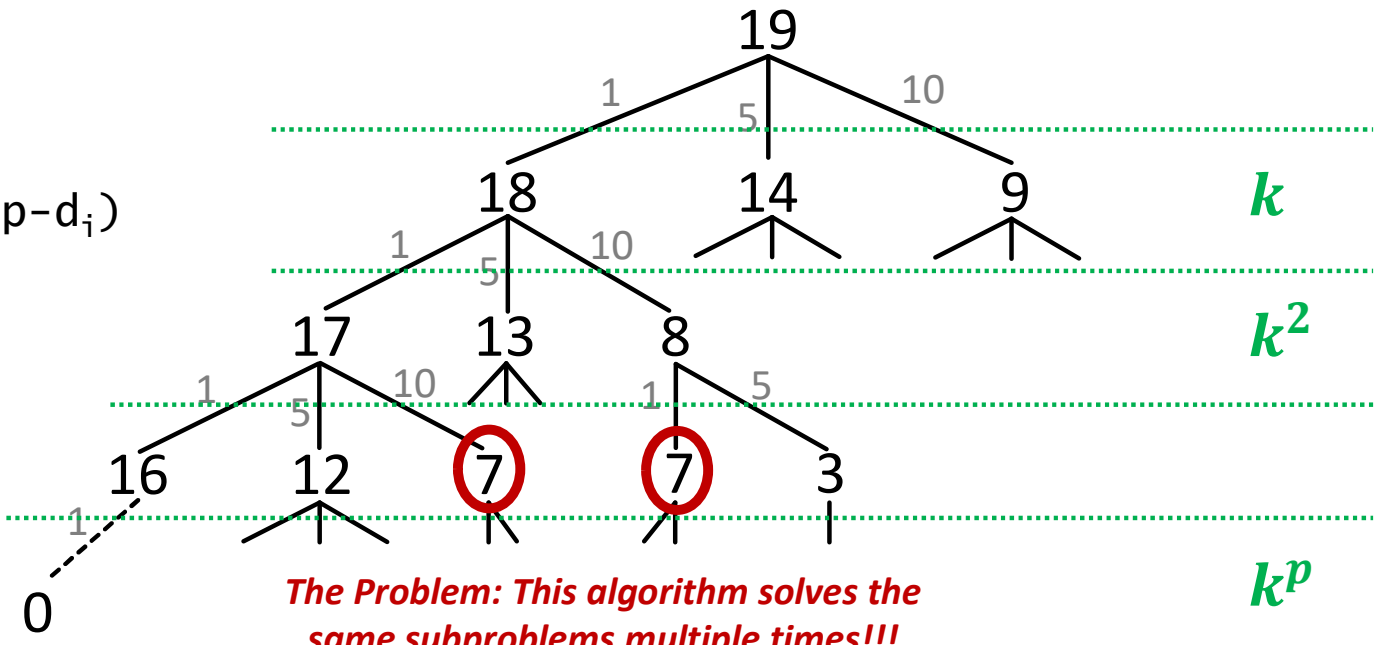
Running time?

$$\frac{k!}{k^p} \in \Omega(\text{scary})$$

Make \$0.19 with \$0.01, \$0.05, \$0.10

k = # denominations

p = value to make change for



The Problem: This algorithm solves the same subproblems multiple times!!!

3. Identify a Memoization Structure & Dependencies

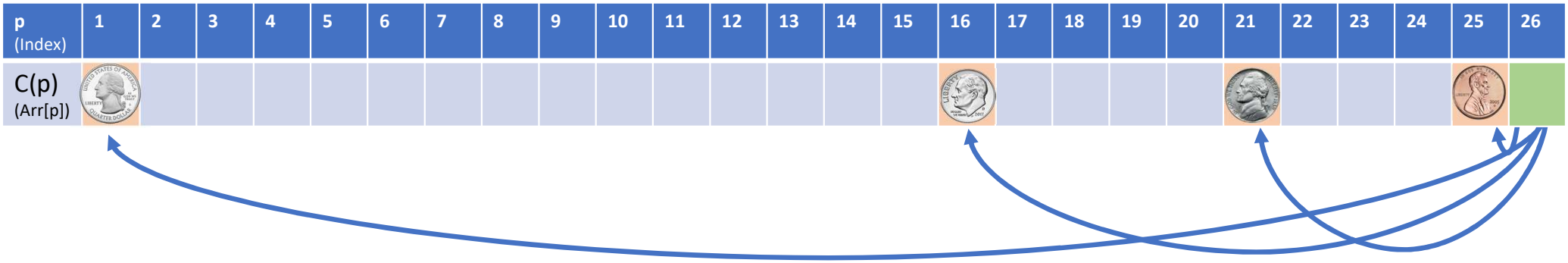
If I know the best way to make change for $0.01 - $0.32, how can I figure out the best way to make change for $0.33?

$$\text{change}(\$0.33) = \min \begin{cases} \text{change}(\$0.08) + 1 \\ \text{change}(\$0.23) + 1 \\ \text{change}(\$0.28) + 1 \\ \text{change}(\$0.32) + 1 \end{cases}$$

If only I had a table with the solutions to every possible subproblem!

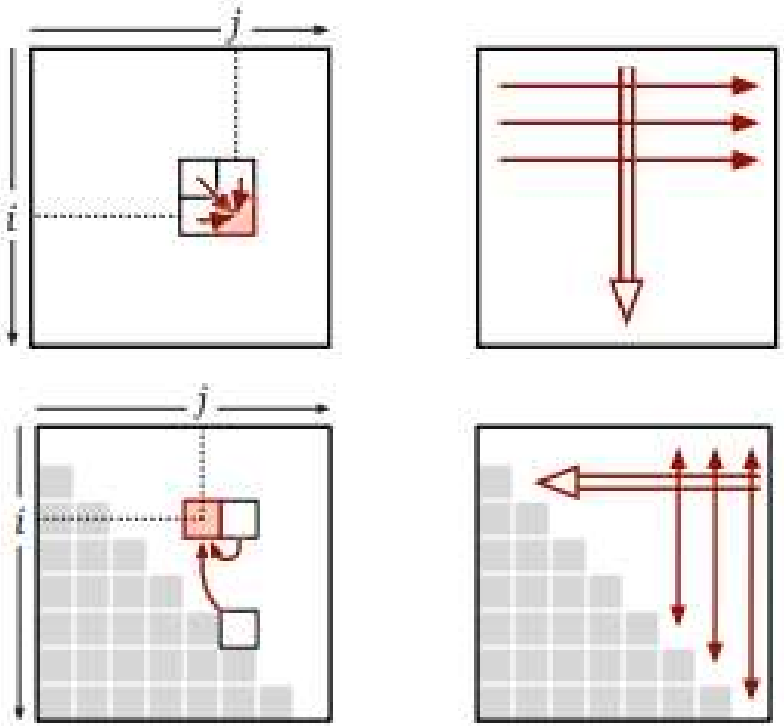
3. Identify a Memoization Structure & Dependencies

- A ***memoization structure*** is a table that holds the values to every previously solved subproblem.
  - These are arrays or matrices.
    - Indexing into an array/matrix is constant time.
- ***Dependencies*** are the subproblems that a given cell needs to look up to solve that cell.
- Consider the problem: C($0.26)



3. Identify a Memoization Structure & Dependencies

- Dependencies help you determine the order in which you need to solve subproblems.
- In more complex Dynamic Programming algorithms, this may be less obvious.
- In the Change Making Problem, our dependencies are always below us, so it makes sense to start at the bottom and work up to the final solution.



Dependency tables for the Edit Distance Problem (above) and the Longest Increasing Substring (below).

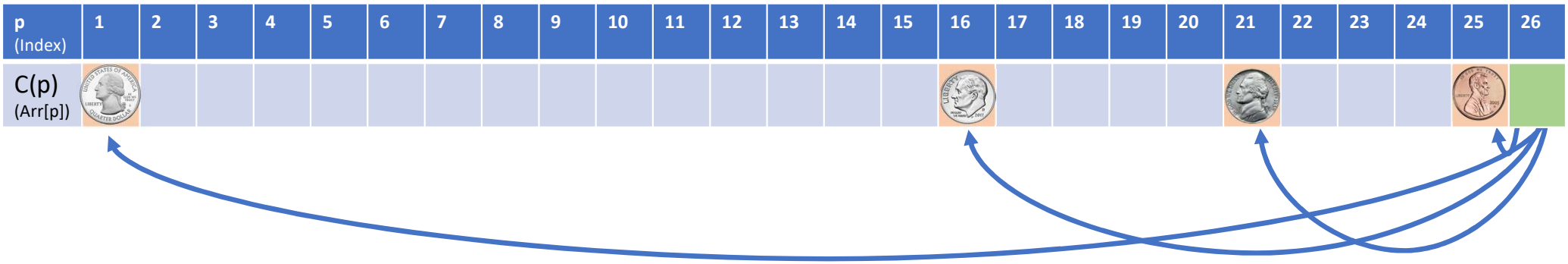
Source: Erickson's Algorithms Textbook

4. Write an Algorithm

Given this recurrence definition:

$$\text{change}(p) = \min \begin{cases} \text{change}(p - d_1) + 1 \\ \text{change}(p - d_2) + 1 \\ \dots \\ \text{change}(p - d_n) + 1 \end{cases}$$

- Where should we start?
  - At the beginning!!!
- PLAN:
  - Let $p_s = the\ starting\ amount\ of\ change\ to\ make$
  - Solve the first instance $p = 1$
  - Solve every instance where $p > 1$
    - Save the subproblem's solution in the table
    - Look up the saved solution to a subproblem when needed
  - Stop when $p = p_s$
  - Return the value saved at p_s



What if I want to know coins that go into the solution?

Add another table!

- Whenever you save a value to the original table, save the value of d_i that was used in the minimization problem to the other table.

$$\text{change}(p) = \min \begin{cases} \text{change}(p - d_1) + 1 \\ \text{change}(p - d_2) + 1 \\ \dots \\ \text{change}(p - d_n) + 1 \end{cases}$$

p (Index)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
C(p) (Arr[p])																			

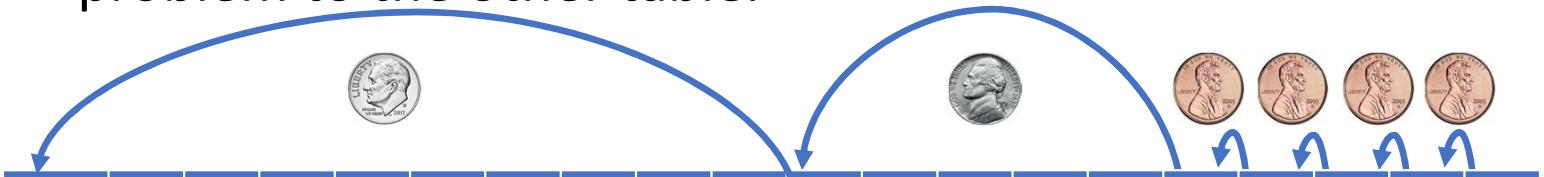
p (Index)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
d_i																			

What if I want to know coins that go into the solution?

Add another table!

- Whenever you save a value to the original table, save the value of d_i that was used in the minimization problem to the other table.

$$\text{change}(p) = \min \begin{cases} \text{change}(p - d_1) + 1 \\ \text{change}(p - d_2) + 1 \\ \cdots \\ \text{change}(p - d_n) + 1 \end{cases}$$



p (Index)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
C(p) (Arr[p])	1	2	3	4	1	2	3	4	5	1	2	3	4	5	2	3	4	5	6

p (Index)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
d_i	1	1	1	1	5	1	1	1	1	10	1	1	1	1	5	1	1	1	1

4. Write an Algorithm

```
changeDP(p)
   Chng[0,...,p] = [0,...,0]
   for m = 1 to p
      min = ∞
      for d_i ≤ m
         if Chng[m – d_i] + 1 < min
            min = Chng[m – d_i] + 1
      Chng[m] = min
   return Chng[p]
```

array holding optimal values for all values $\leq p$

find optimal change amounts starting with 1

for each denomination value.

check optimal value of $m - d_i$

5. Analyze Complexity (Running Time & Space)

```
changeDP(p)
   Chng[0,...,p] = [0,...,0]
   for m = 1 to p
      min = ∞
      for d_i ≤ m
         if Chng[m - d_i] + 1 < min
            min = Chng[m - d_i] + 1
      Chng[m] = min
   return Chng[p]
```

Running time?

Outer for loop $\in O(p)$

Inner for loop $\in O(k)$

Total $\in O(pk)$

Space?

Total $\in O(p)$

We could also use a decrementing function to show that the algorithm will terminate!

6. Prove Correctness: Overview

Algorithm 1 : makeChangeDP(int[ ] denominations, int amount)

Result: Returns the minimum number of bills needed to make change on an input amount given a set of available denominations

```
create dpData = int[amount+1]
dpData[0] ← 0
create i ← 1
while i ≤ amount do
    dpData[i] ← INFINITY
    i ++
end
create subAmount ← 0
while subAmount ≤ amount do
    create denom ← 0
    while denom ≤ denominations[ ].length do
        if demnominations[denom] ≤ subAmount then
            create subSoln ← dpData[subAmount - denominations[denom]]
            if subSoln + 1 < dpData[subAmount then
                dpData[subAmount] = subSoln + 1
            end
        end
        denom ++
    end
    subAmount ++
end
return dpData[amount]
```

If there is a loop or recursion, what is the loop/recursion invariant? Provide the proof.

Proof: We are to show the correctness of the loop invariant for the *makeChangeDP* Algorithm.

- To show the correctness of an algorithm using proofs of initialization, maintenance, and completion, I think it's best to have the pseudocode ready to be referenced. Use the verbiage from the pseudocode to prove each case!

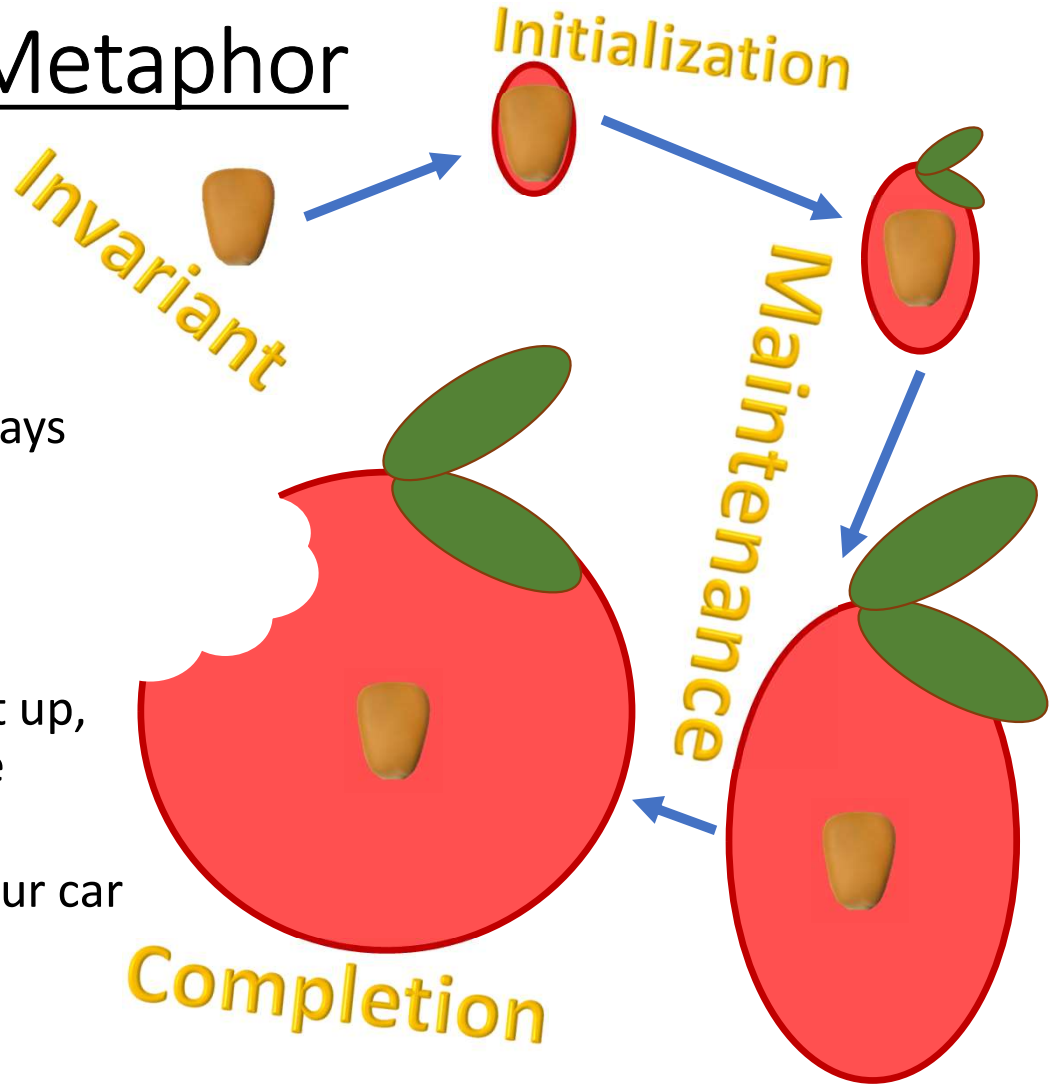
- Ultimately, these proofs work via propositional logic, where we have 4 initial propositions that we must define before starting the proof:
  - *Loop/Recursion Guard, G*
  - *Pre-Condition, P*
  - *Post-Condition, Q*
  - *Loop/Recursion Invariant, L*

How Algorithms Work: A Metaphor

- Imagine that you are a wizard…
  - That grows crops…
  - So really you're just a “cool” farmer.
- Regular (“uncool”) famers use seeds that are “destroyed” as their plants begin to grow. Instead, you magically grow from kernels!
- A kernel is indestructible and unchanging. When you plant a kernel, it grows a fruit around it, but itself never changes.
- That kernel can then be extracted from the fruit and reused to make more fruit.
- Sweet trick “cool” farmer! You’re gunna go far!

How Algorithms Work: A Metaphor

- **So what does that kernel represent?**
  - It's the loop/recursion invariant!
  - It's the kernel of ***truth*** that moves the algorithm forward.
  - And while the fruit grows, the kernel always remains the same.
- **So what does the fruit represent?**
  - The fruit is the solution!
  - As an algorithm runs, the solution is built up, fueled by the "magical" properties of the kernel.
  - The invariant is the engine that brings your car (the solution) up to speed!
    - *>> Error: mixed metaphors*



A Quick Review of Propositional Logic

- A ***proposition*** is a statement that you can evaluate to be either ***true*** or ***false***.
- Propositions can be combined via operators:
  - **NOT:** ¬ ~
  - **OR:** ∨
  - **AND:** ∧
  - **IMPLIES:** ⇒
    - Also known as IF… THEN

Let:
W = It is warm
R = It is raining
J = Wear a jacket

The proposition
¬ **W** ∨ **R** ⇒ **J**

Says,
"If it is not warm or raining then wear a jacket."

6. Prove Correctness: Relevant Propositions

Algorithm 1 : makeChangeDP(int[ ] denominations, int amount)

Result: Returns the minimum number of bills needed to make change on an input amount given a set of available denominations

```
create dpData = int[amount+1]
dpData[0] ← 0
create i ← 1
while i ≤ amount do
    dpData[i] ← INFINITY
    i ++
end
create subAmount ← 0
while subAmount ≤ amount do
    create denom ← 0
    while denom ≤ denominations[ ].length do
        if demnominations[denom] ≤ subAmount then
            create subSoln ← dpData[subAmount - denominations[denom]]
            if subSoln + 1 < dpData[subAmount then
                dpData[subAmount] = subSoln + 1
            end
        end
        denom ++
    end
    subAmount ++
end
return dpData[amount]
```

If there is a loop or recursion, what is the loop/recursion invariant? Provide the proof.

Proof: We are to show the correctness of the loop invariant for the *makeChangeDP* Algorithm.

- **Loop/Recursion Guard, G**
  - This is the evaluation that determines if the loop/recursion should continue. When G is no longer true, then the loop/recursion should be at termination.
- **Pre-Condition, P**
  - This is the base case or where the algorithm starts. The invariant must be true when the pre-condition is true (i.e. the algorithm must be correct at the base case) for the whole proof to work.
- **Post-Condition, Q**
  - This is the case (i.e. what you know) at the end of the loop/recursion. It should represent the correct solution to the problem.
- **Loop/Recursion Invariant, L**
  - This is a proposition whose value can be evaluated to true regardless of what step the algorithm is at.
  - A.K.A. The truth of the invariant does not vary- It is always true!!!

6. Prove Correctness: Assigning the Propositions

Algorithm 1 : makeChangeDP(int[ ] denominations, int amount)

Result: Returns the minimum number of bills needed to make change on an input amount given a set of available denominations

```
create dpData = int[amount+1]
dpData[0] ← 0
create i ← 1
while i ≤ amount do
    dpData[i] ← INFINITY
    i ++
end
create subAmount ← 0
while subAmount ≤ amount do
    create denom ← 0
    while denom ≤ denominations[ ].length do
        if demnominations[denom] ≤ subAmount then
            create subSoln ← dpData[subAmount - denominations[denom]]
            if subSoln + 1 < dpData[subAmount then
                dpData[subAmount] = subSoln + 1
            end
        end
        denom ++
    end
    subAmount ++
end
return dpData[amount]
```

If there is a loop or recursion, what is the loop/recursion invariant? Provide the proof.

Proof: We are to show the correctness of the loop invariant for the *makeChangeDP* Algorithm.

Loop Guard G: $subAmount \leq amount$
Pre Condition P: $subAmount = 0$
Post Condition Q: Array $dpData$ = {optimal change for all positive integers less than and equal to the starting amount}
Loop Invariant L: Array $dpData[0 : subAmount]$ = {optimal change for all integers less than and equal to the sub-problem amount} and $dpData[subAmount + 1 : amount]$ are all strictly greater than $subAmount$

6. Prove Correctness: How the Proofs Work

Loop Guard G: $subAmount \leq amount$
Pre Condition P: $subAmount = 0$
Post Condition Q: Array $dpData$ = {optimal change for all positive integers less than and equal to the starting amount}
Loop Invariant L: Array $dpData[0 : subAmount]$ = {optimal change for all integers less than and equal to the sub-problem amount} and $dpData[subAmount + 1 : amount]$ are all strictly greater than $subAmount$

Algorithm 1 : makeChangeDP(int[ ] denominations, int amount)
Result: Returns the minimum number of bills needed to make change on an input amount given a set of available denominations

```
create dpData = int[amount+1]
dpData[0] ← 0
create i ← 1
while i ≤ amount do
    dpData[i] ← INFINITY
    i ++
end
create subAmount ← 0
while subAmount ≤ amount do
    create denom ← 0
    while denom ≤ denominations[ ].length do
        if demnominations[denom] ≤ subAmount then
            create subSoln ← dpData[subAmount - denominations[denom]]
            if subSoln + 1 < dpData[subAmount then
                dpData[subAmount] = subSoln + 1
            end
        end
        denom ++
    end
    subAmount ++
end
return dpData[amount]
```

- **Proof of Initialization**
  - Shows that at the start of the loop the invariant does indeed hold true.
  - P ⇒ L
  - Analogous to the base case in a proof by induction.
- **Proof of Maintenance**
  - Shows that the invariant is true at any given step of the loop if we assume that the previous step was true and the loop is still running.
  - $L_i \wedge G \Rightarrow L_{i+1}$
  - Analogues to the inductive step in a proof by induction.
- **Proof of Completion**
  - Shows that if the loop is at termination and the invariant is still true then you have your final solution.
  - $L \wedge \neg G \Rightarrow Q$
  - No analogue to a proof by induction.
- ***In each of the proofs, you get to assume the propositions on the LHS are true. Your proof is then an argument that those assumptions, along with your pseudocode, imply the truth of the RHS.***

Algorithm 1 : makeChangeDP(int[ ] denominations, int amount)

Result: Returns the minimum number of bills needed to make change on an input amount given a set of available denominations

```
create dpData = int[amount+1]
dpData[0] ← 0
create i ← 1
while i ≤ amount do
    dpData[i] ← INFINITY
    i ++
end
create subAmount ← 0
while subAmount ≤ amount do
    create denom ← 0
    while denom ≤ denominations[ ].length do
        if demnominations[denom] ≤ subAmount then
            create subSoln ← dpData[subAmount - denominations[denom]]
            if subSoln + 1 < dpData[subAmount then
                dpData[subAmount] = subSoln + 1
            end
        end
        denom ++
    end
    subAmount ++
end
return dpData[amount]
```

6. Prove Correctness: Proof of Initialization

Loop Guard G: $subAmount \leq amount$
Pre Condition P: $subAmount = 0$
Post Condition Q: Array $dpData$ = {optimal change for all positive integers less than and equal to the starting amount}
Loop Invariant L: Array $dpData[0 : subAmount]$ = {optimal change for all integers less than and equal to the sub-problem amount} and $dpData[subAmount + 1 : amount]$ are all strictly greater than $subAmount$

Proof of Initialization $(P \Rightarrow L)$

If $subAmount = 0$, then $dpData[subAmount]$ has already been initialized to 0. This represents a true proposition: The optimal number of bills required to make change for \$0 is 0. As such, the proof of initialization is vacuously true.

6. Prove Correctness: Proof of Maintenance

Algorithm 1 : makeChangeDP(int[ ] denominations, int amount)

Result: Returns the minimum number of bills needed to make change on an input amount given a set of available denominations

```
create dpData = int[amount+1]
dpData[0] ← 0
create i ← 1
while i ≤ amount do
    dpData[i] ← INFINITY
    i ++
end
create subAmount ← 0
while subAmount ≤ amount do
    create denom ← 0
    while denom ≤ denominations[ ].length do
        if demnominations[denom] ≤ subAmount then
            create subSoln ← dpData[subAmount - denominations[denom]]
            if subSoln + 1 < dpData[subAmount then
                dpData[subAmount] = subSoln + 1
            end
        end
        denom ++
```

Loop Guard G: $subAmount \leq amount$
Pre Condition P: $subAmount = 0$
Post Condition Q: Array $dpData$ = {optimal change for all positive integers less than and equal to the starting amount}
Loop Invariant L: Array $dpData[0 : subAmount]$ = {optimal change for all integers less than and equal to the sub-problem amount} and $dpData[subAmount + 1 : amount]$ are all strictly greater than $subAmount$

Proof of Maintenance ($L_i \wedge G \Rightarrow L_{i+1}$)

Given that we are on the i^{th} iteration and can assume that L_i holds and G holds. We need to prove that the L_{i+1} holds. By L_i, $dpData[0 : i]$ is the set of optimal change for integer amounts 0 to i. The calculation of $dpData[i + 1]$ will consider all relevant denominations by looking back to a known correct value (i.e. $dpData[x]|x \leq i$) equal to the current value minus the denomination (i.e. $dpData[x] = dpData[i+1-denomination]$) and add a 1 to represent that known correct value becoming the current value by the addition of one bill of the denomination. By taking the minimum number of these evaluations, the algorithm has found the optimal way to make change on $dpData[i+1]$. As such, it has been proven that L_{i+1} is true.

Algorithm 1 : makeChangeDP(int[ ] denominations, int amount)

Result: Returns the minimum number of bills needed to make change on an input amount given a set of available denominations

```
create dpData = int[amount+1]
dpData[0] ← 0
create i ← 1
while i ≤ amount do
    dpData[i] ← INFINITY
    i ++
end
create subAmount ← 0
while subAmount ≤ amount do
    create denom ← 0
    while denom ≤ denominations[ ].length do
        if demnominations[denom] ≤ subAmount then
            create subSoln ← dpData[subAmount - denominations[denom]]
            if subSoln + 1 < dpData[subAmount then
                dpData[subAmount] = subSoln + 1
            end
        end
        denom ++
    end
    subAmount ++
end
return dpData[amount]
```

6. Prove Correctness: Proof of Completion

Loop Guard G: $subAmount \leq amount$
Pre Condition P: $subAmount = 0$
Post Condition Q: Array $dpData$ = {optimal change for all positive integers less than and equal to the starting amount}
Loop Invariant L: Array $dpData[0 : subAmount]$ = {optimal change for all integers less than and equal to the sub-problem amount} and $dpData[subAmount + 1 : amount]$ are all strictly greater than $subAmount$

Proof of Completion $(L \wedge \neg G \Rightarrow Q)$

If the loop guard, G, is defined as $subAmount \leq amount$, then the negation, $\neg G$, is $subAmount > amount$. By closure of the integers, this means $subAmount - 1 \geq amount$. Since the loop invariant in the maintenance step proved that all L_i, including L_{i-1}, was true, therefore, the truth of $L_{subAmount-1}$ implies the truth of L_{amount}, thus proving that the algorithm completes correctly, resulting in Q.

Algorithm 1 : makeChangeDP(int[ ] denominations, int amount)

Result: Returns the minimum number of bills needed to make change on an input amount given a set of available denominations

```
create dpData = int[amount+1]
dpData[0] ← 0
create i ← 1
while i ≤ amount do
    dpData[i] ← INFINITY
    i ++
end
create subAmount ← 0
while subAmount ≤ amount do
    create denom ← 0
    while denom ≤ denominations[ ].len
        if demnominations[denom] <
            create subSoln ← dp
            if subSoln + 1
                dpData[subAm
            end
        end
        denom ++
    end
    subAmount ++
end
return dpData[amount]
```

6. Pro ectness: letion

Note: A proof of completion is NOT a proof of termination.

A proof of completion assumes that the recursion/loop stops and shows that the post-condition, therefore, is true.

A proof of termination, done via a discussion of running time or a decrementing function, shows that from the pre-condition you will eventually arrive at a false loop/recursion guard: P ⇒ ¬G

ll positive integers less than and equal to

change for all integers less than and *amount*] are all strictly greater than

Proof of Completio

If the loop guard, G, is en the negation, $\neg G$, is $subAmount > amount$. By closure of the $nt - 1 \geq amount$. Since the loop invariant in the maintenance step prove $_{i-1}$, was true, therefore, the truth of $L_{subAmount-1}$ implies the truth of L_{amount}, the algorithm completes correctly, resulting in Q.

Algorithm 1 : makeChangeDP(int[ ] denominations, int amount)

Result: Returns the minimum number of bills needed to make change on an input amount given a set of available denominations

```
create dpData = int[amount+1]
dpData[0] ← 0
create i ← 1
while i ≤ amount do
    dpData[i] ← INFINITY
    i ++
end
create subA
while subA
    create de
    while
        if de
            c
            i
            |
            e
        end
        deno
    end
    subAmo
end
return dpDa
```

6. Pro ectness: letion

tion is NOT a proof of

The Complete Proof of Correctness:

Partial Correctness ∧ Termination ⇒ Correctness

where

Initialization ∧ Maintenance ∧ Completion ⇒ Partial Correctness

running
from the pre-
false loop?

l equal to

than and
ater than

Proc

If th $t >$ *amount*. By closure of the $nt - 1 \geq amount$. Since the loop invariant in the maintenance step prove$_{i-1}$, was true, therefore, the truth of $L_{subAmount-1}$ implies the truth of L_{amount}, the algorithm completes correctly, resulting in Q.