

# Parallel And Sequential Data Structures and Algorithms

**Dynamic Programming I**

# Learning Objectives

- Understand **dynamic programming**, a very prevalent and powerful algorithm design technique
- See how to specify and analyze solutions to dynamic programming problems
- Compare the various ways to implement dynamic programming algorithms
- Understand the three main categories of problems we can solve with dynamic programming

# **Dynamic Programming: Coin Change**

# Coin Change Problem

**Problem (Coin Change-Feasibility):** Given a currency (\$) with denominations  $c_1, c_2, \dots, c_n$ , is it possible to make exactly  $\$V$ ? Denominations may be used multiple times.

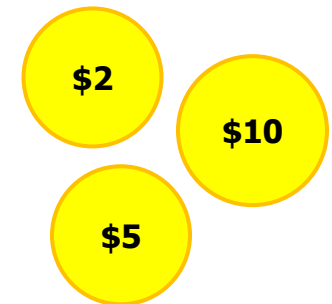
- Example:  $c = [2, 5, 10]$

- Can I make \$6?
- Can I make \$12?
- Can I make \$3?

**Yes: \$2 + \$2 + \$2**

**Yes: \$2 + \$5 + \$5 or \$2 + \$10**

**No**



# Problem Solving: Subproblems

- There's a common theme in the techniques we've studied so far: divide-and-conquer and contraction
- Divide-and-conquer and contraction reduce a problem to **smaller version(s) of itself!**
- This strategy works so well, let's continue using it
- I have denominations \$2, \$5, and \$10. **Can I make \$42?**

**Key idea:** express the solution in terms of **smaller problems**

**I can make \$42 if and only if I can make \$40, or \$37, or \$32**

# Step 1 of DP: Define Subproblems

**Step 1** of any dynamic programming algorithm is **always** to **define your set of subproblems**, precisely and unambiguously

Define, for all  $0 \leq v \leq V$ :

$$\text{Possible}(v) := \begin{cases} \mathbf{True} & \text{if exactly \$}v \text{ can be made} \\ \mathbf{False} & \text{otherwise} \end{cases}$$

- Give your subproblems reasonable names
- Clearly state the value of the subproblem (e.g., True/False)
- Describe the meaning of the parameters (e.g.,  $v$ )
- Give the domain of the parameters (e.g.,  $0 \leq v \leq V$ )

# Step 2 of DP: Recursive Solution

**Step 2** of any dynamic programming algorithm is to solve a subproblem by writing a recursive definition of the solution in terms of solutions to smaller subproblems. We write this as a **recurrence relation**.

- With coins  $c = [2, 5, 10]$ , we know that (roughly)

$$\text{Possible}(v) = \text{Possible}(v - 2) \textbf{ or } \text{Possible}(v - 5) \textbf{ or } \text{Possible}(v - 10)$$

$$\text{Possible}(v) := \begin{cases} \text{True} & \text{if } v = 0 \\ \bigvee_{\substack{i \in [n] \\ c_i \leq v}} \text{Possible}(v - c_i) & \text{otherwise} \end{cases}$$

# Step \_ of DP: Implement Recursion?

```
fun coin_change(c : sequence<int>, V : int) -> bool:
  fun possible(v):
    if v == 0: return True
    else:
      answer = False
      for denomination in c:
        if denomination <= v:
          answer ← answer or possible(v - denomination)
      return answer
  return possible(V)
```

*Note: This solution is imperative for simplicity*

# Step \_ of DP: Implement Recursion?

```
fun coin_change(c : sequence<int>, V : int) -> bool:
    fun possible(v):
        return v == 0 or any(map(fn(x) => (x <= v and possible(v - x)), c))
    return possible(V)
```

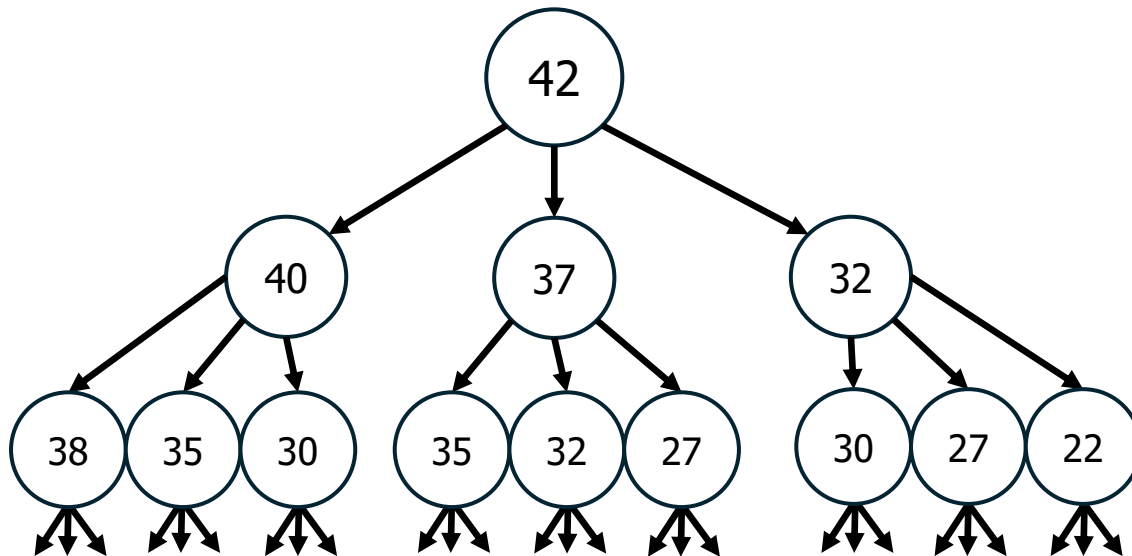
*Here is a functional version if you want<sup>1</sup>*

---

<sup>1</sup>any is a helper function that returns True if any element of the given Boolean sequence is True. It can be implemented as reduce(or, False, S)

# Step \_ of DP: Cry Because its Slow??

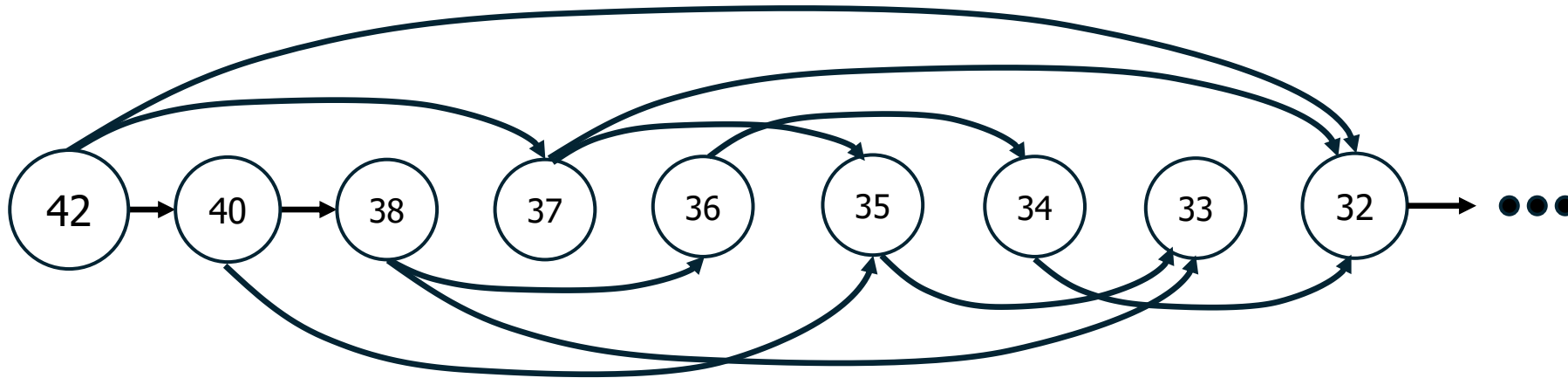
- What happens when we call Possible(42)?



- This tree contains 34833 nodes!
- But there are only 43 distinct subproblems...
- There is an absurd amount of redundancy in this computation!

# Step \_ of DP: Realize it Can Be Fast

- The subproblems aren't really a tree...
- They are a directed graph!



# Step 3 of DP: Make it Faster!

**Dynamic Programming** is about solving our subproblems without ever evaluating the same one more than once

*There are two (very different) ways to implement DP*

**Definition (Bottom-Up DP):** **Bottom-up** DP solves the subproblems in an order where every dependency is solved first (often smallest to largest)

**Definition (Top-Down DP):** **Top-down** DP solves the subproblems recursively but keeps a cache of previously solved subproblems.

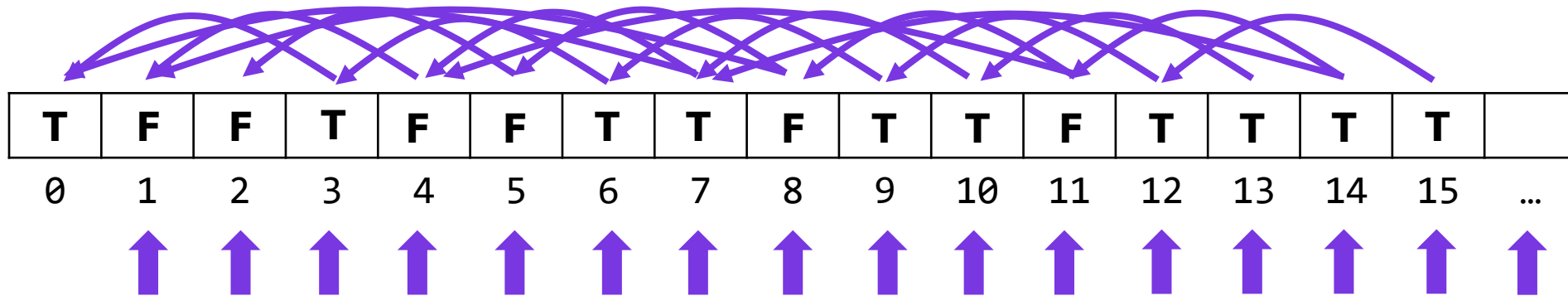
# Bottom-up Dynamic Programming

**Definition (Bottom-Up DP):** **Bottom-up** DP solves the subproblems in an order where every dependency is solved first (often smallest to largest)

```
fun coin_change(c : sequence<int>, V : int) -> bool:
    possible = [False for v in 0...V]
    for v in 0...V:
        if v == 0:
            possible[v] ← True
        else:
            answer = False
            for denomination in c:
                if denomination <= v:
                    answer ← answer or possible[v - denomination]
            possible[v] ← answer
    return possible[V]
```

# Bottom-up DP as "Table Filling"

- Suppose  $c = [3,7]$



```
answer = False
for denomination in c:
    if denomination <= v:
        answer ← answer or possible[v - denomination]
    possible[v] ← answer
```

# Cost Analysis of Bottom Up

**Theorem (Cost of Bottom-Up Coin Change):** Given  $n$  denominations, to make  $\$V$ , bottom-up coin change costs  $O(nV)$  work.

- *There are  $V + 1$  subproblems*
- *Each subproblem considers the  $n$  denominations*
- *That's  $O(nV)$  total work*

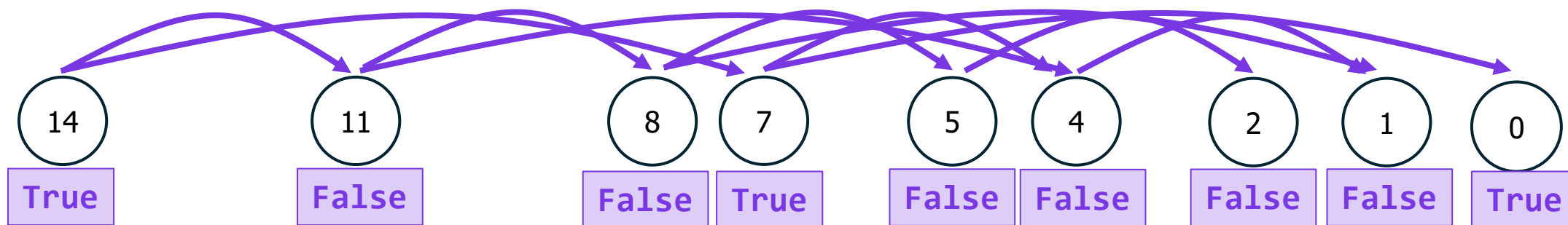
# Top-down Dynamic Programming

**Definition (Top-Down DP):** **Top-down** DP solves the subproblems recursively but keeps a cache of previously solved subproblems.

```
fun coin_change(c : sequence<int>, V : int) -> bool:
    memoized = [⊥ for v in 0...V]
    fun possible(v):
        if memoized[v] == ⊥:
            if v == 0: memoized[v] ← True
            else:
                answer = False
                for denomination in c:
                    if denomination <= v:
                        answer ← answer or possible(v - denomination)
                memoized[v] ← answer
        return memoized[v]
    return possible(V)
```

# Top-down DP as Memoization

- Suppose  $c = [3,7]$ , and we call **possible(14)**



# Cost Analysis of Top Down

**Theorem (Cost of Top-Down Coin Change):** Given  $n$  denominations, to make  $\$V$ , top-down coin change costs  $O(nV)$  work.

- *Each subproblem is solved at most once*
- *There are  $V + 1$  subproblems*
- *Each subproblem considers the  $n$  denominations*
- *That's  $O(nV)$  total work*

Top-down and bottom-up **have the same worst-case cost**

# Top-down versus bottom-up

- Top-down versus bottom-up is **mostly** a matter of personal preference. There are a few minor pros and cons:

## Advantages of top-down

- Don't need to know the order to evaluate subproblems in.  
Recursion does it automatically!
- Only evaluates necessary subproblems: can be faster if many subproblems aren't needed

## Advantages of bottom-up

- Can sometimes use advanced data structures or extra information to speed up (see **15-451**)
- Slightly faster in practice due to lack of recursion overhead
- No stack overflow

# What About Functional??

- So far, we have presented dynamic programming algorithms using imperative code!
- You can implement dynamic programming using purely functional code, but its... a *little* harder

```
fun coin_change(c : sequence<int>, V : int) -> bool:
  fun possible(table : Dict<int,bool>, v : int) -> Dict<int,bool>:
    answer = any(map(fn x => (x <= v and
                           find(table, v - x) == SOME(True)), c))
    return insert(table, v, answer)
  table = fold_left(possible, {0: True}, 1...V)
  return find(table, V) == SOME(True)
```

---

\*assuming that Dict is a **persistent** dictionary data type



# **Reconstructing the Solution**

# Reconstructing the Solution

So far, we have only computed the **value** of a solution (e.g., is it possible to make  $V$ : True/False) but not the combinatorial solution itself (e.g., return a sequence of coins that sums to  $V$ )

There are three common ways to do this (one is bad)

- **Add combinatorial state to your subproblem (this is bad!)**
- Backtrack through the subproblems and reconstruct after the fact
- Store an additional table during the algorithm that stores decisions

# The Bad Way (Avoid This)

- For example, storing the list of coins as additional information:

Define your subproblem type as:

$\text{Possible}(v) \in (\text{bool}, \text{sequence}\langle \text{int} \rangle)$

Each subproblem is no longer constant size! Subproblems are now  $O(V)$  size so the cost of solving the problem becomes  $O(nV^2)$ .

- You could probably get away with this by storing a **persistent list**... but that's more complicated and likely slower

# Backtrack Through Subproblems

Essentially, solve the recurrence "backwards" after the fact to figure out which coins were used

```
// Assume that possible has been computed and possible[V] == True

coin_list = [] // this must be a mutable dynamic array for efficiency
v = V
while v > 0:
    for denomination in c:
        if denomination <= v and possible[v - denomination]:
            coin_list.append(denomination) // assuming O(1) amortized append
            v ← v - denomination
return coin_list
```

# Store Decisions in Another Table

Each time we find a coin that makes a particular value of `possible[v] = True`, remember which coin it was

```
possible = [False for v in 0...V]
choice   = [⊥ for v in 0...V]
```

```
for v in 0...V:
    if v == 0:
        possible[v] ← True
    else:
        for denomination in c:
            if denomination <= v and
               possible[v - denomination]:
                possible[v] ← True
                choice[v] ← denomination
```

```
// Assuming possible[V] == True
```

```
coin_list = [] // mutable dynamic array
v = V
while v > 0:
    denomination = choice[v]
    coin_list.append(denomination)
    v ← v - denomination
return coin_list
```

# **Feasibility, Optimization, and Counting Problems**

# Problems Solvable with DP

- Dynamic programming can be used to solve many kinds of problems. Most frequently:
  - **Feasibility/decision problems**: Does this problem have a solution?
  - **Optimization problems**: Find the solution with the best value
  - **Counting problems**: Count the number of solutions

In all three cases, dynamic programming is similar:

- Choose the right set of subproblems
- Design a recurrence that solves the subproblems
- Analyze the cost

# Coin Change (Optimization)

**Problem (Coin Change-Optimization):** Given a currency (\$) with denominations  $c_1, c_2, \dots, c_n$ , what is the **fewest number of coins** needed to make exactly \$ $V$ ? Denominations may be used multiple times.

Define, for all  $0 \leq v \leq V$ :

$\text{MinCoins}(v) :=$  fewest coins needed to make \$ $v$

For optimization problems, we typically define the value of an infeasible problem as  $\infty$  (for minimization problems) or  $-\infty$  (for maximization).

# Coin Change (Optimization)

**Problem (Coin Change-Optimization):** Given a currency (\$) with denominations  $c_1, c_2, \dots, c_n$ , what is the **fewest number of coins** needed to make exactly  $\$V$ ? Denominations may be used multiple times.

$$\text{MinCoins}(v) = \begin{cases} 0 & \text{if } v = 0 \\ \infty & \text{if } c_i > v \text{ for all } i \\ \min_{\substack{i \in [n] \\ c_i \leq v}} (1 + \text{MinCoins}(v - c_i)) & \text{otherwise} \end{cases}$$

# Coin Change (Optimization)

**Problem (Coin Change-Optimization):** Given a currency (\$) with denominations  $c_1, c_2, \dots, c_n$ , what is the **fewest number of coins** needed to make exactly  $\$V$ ? Denominations may be used multiple times.

**Theorem (Cost of Coin Change-Optimization):** Given  $n$  denominations, to make  $\$V$ , the optimization problem costs  $O(nV)$  work.

- *There are  $V + 1$  subproblems*
- *Each subproblem takes  $O(n)$  cost to solve*
- *The total cost is  $O(nV)$  work*

# Coin Change (Counting)

**Problem (Coin Change-Counting):** Given a currency (\$) with denominations  $c_1, c_2, \dots, c_n$ , **how many distinct ways** are there to make exactly  $\$V$ ? Order matters ( $[2,5]$  and  $[5,2]$  are different).

Define, for all  $0 \leq v \leq V$ :

$\text{CountWays}(v) :=$  number of ways to make  $\$v$

For counting problems, we must be careful to not **double count**. This was not an issue for optimization or feasibility.

# Coin Change (Counting)

**Problem (Coin Change-Counting):** Given a currency (\$) with denominations  $c_1, c_2, \dots, c_n$ , **how many distinct ways** are there to make exactly  $\$V$ ? Order matters ( $[2,5]$  and  $[5,2]$  are different).

$$\text{CountWays}(v) := \begin{cases} 1 & \text{if } v = 0 \\ \sum_{\substack{i \in [n] \\ c_i \leq v}} \text{CountWays}(v - c_i) & \text{otherwise} \end{cases}$$

# Coin Change (Counting)

**Problem (Coin Change-Counting):** Given a currency (\$) with denominations  $c_1, c_2, \dots, c_n$ , **how many distinct ways** are there to make exactly  $\$V$ ? Order matters ( $[2,5]$  and  $[5,2]$  are different).

**Theorem (Cost of Coin Change-Counting):** Given  $n$  denominations, to make  $\$V$ , the counting problem costs  $O(nV)$  work.

- *There are  $V + 1$  subproblems*
- *Each subproblem takes  $O(n)$  cost to solve*
- *The total cost is  $O(nV)$  work*

# Summary

- Dynamic programming (like D&C and contraction) reduces a problem to smaller version(s) of itself (**subproblems**)
- The key steps for designing a DP algorithm are:
  - Define a set of subproblems (specifying their value and parameters)
  - Define a recurrence to solve the subproblems in terms of each other
  - Analyze the cost in terms of the number of subproblems and the cost to solve each subproblem
- Two implementation strategies: **top down** and **bottom up**
- Works for **feasibility**, **optimization**, and **counting** problems