The Art of Data Structures

Dynamic Programming

Alan Beadle
CSC 162: The Art of Data Structures

Agenda

- Understand Dynamic Programming as a technique used to solve optimization problems

Dynamic Programming

Dynamic Programming

- Many programs in computer science are written to optimize some value:
  - Find the shortest path between two points,
  - Find the line that best fits a set of points
  - Find the smallest set of objects that satisfies some criteria

Dynamic Programming

- There are many strategies that computer scientists use to solve these optimization problems
- Dynamic programming is one strategy

Dynamic Programming

- Making change using the fewest coins is one classic optimization problem

Coin Optimization

Using Fewest Coins

Coin Optimization

Using Fewest Coins

- For a currency with coins C_1, C_2, ..., C_n (cents) what is the minimum number of coins needed to make K cents of change?
- US currency has 1, 5, 10, and 25 cent denominations. (Also 50 cent and 1 dollar denominations).

Coin Optimization

Using Fewest Coins

- A purchase is made for $0.37
- Change due is $0.63 cents
- We can make 63 cents using two quarters, one dime and 3 pennies
- This is done using a **greedy method** of choosing as many of largest coins as possible before choosing smaller coins

Coin Optimization

Using Fewest Coins

- What if US currency had a $0.21 coin?
- Does this greedy method still work?
- No, it still chooses the same 6 coins, missing the fact the three 21 cent coins are the optimal solution to the problem

Coin Optimization

Recursive Solution

1. **Base Case**: If we can make change that is satisfied by exactly one coin, then that is a minimum
2. **Otherwise:** minimum of a penny plus the number of coins needed to make change for the original amount minus one cent, nickel, minus five cents; or a dime; minus ten cents, and so on…

Coin Optimization

Recursive Solution

$$numCoins = min \begin{cases} 1 + numCoins(original amount - 1) \\ 1 + numCoins(original amount - 5) \\ 1 + numCoins(original amount - 10) \\ 1 + numCoins(original amount - 25) \end{cases}$$

Coin Optimization

Recursive Solution

```
def rec_mc(coin_values, change):
    min_coins = change
    if change in coin_values:
        return 1
    else:
        for i in [c for c in coin_values if c <= change]:
            num_coins = 1 + rec_mc(coin_values, change-i)
            if num_coins < min_coins:
                min_coins = num_coins
    return min_coins


value = 63
rec_mc([1, 5, 10, 25], value)
```

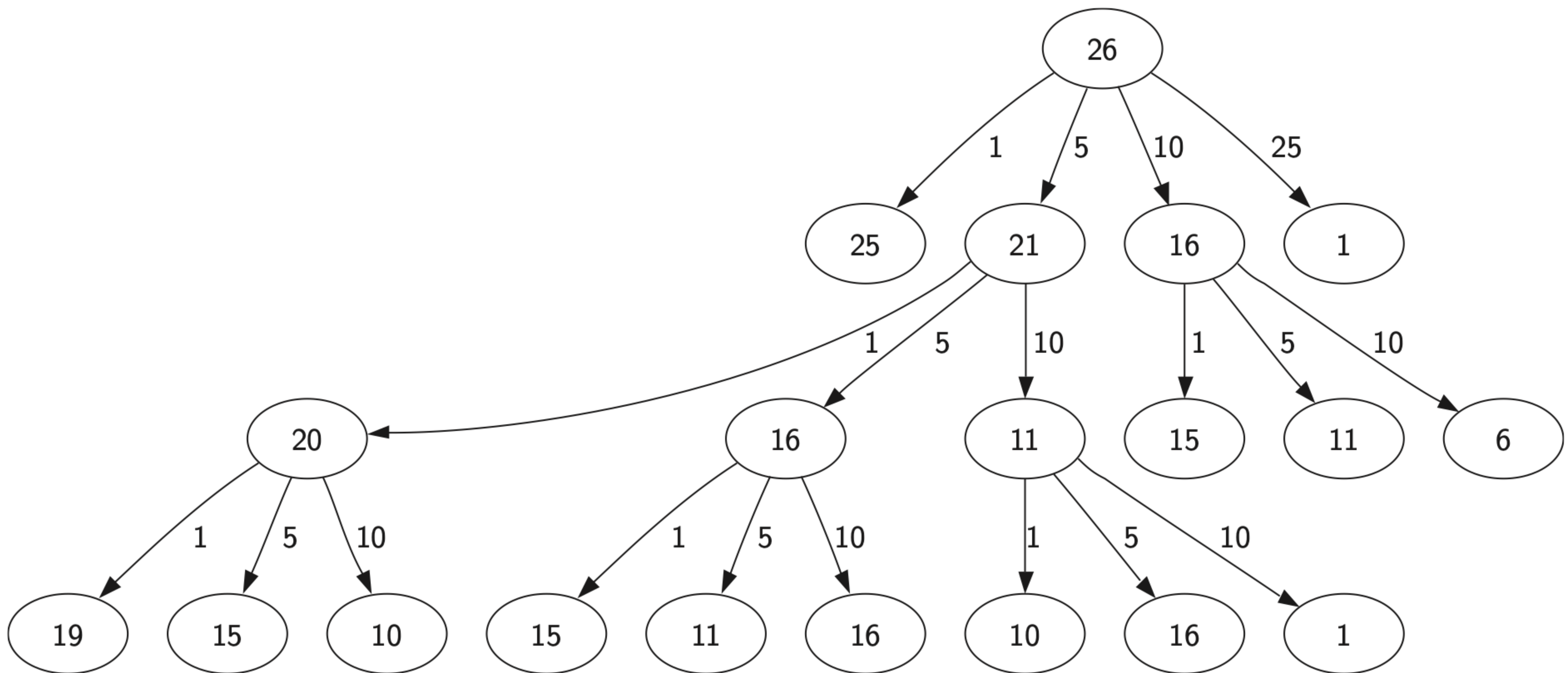
Coin Optimization

Recursive Results

- This takes a few minutes to run
- 67,716,925 recursive calls to `rec_mc`!

Coin Optimization

Call Tree



Coin Optimization

Recursive Performance

- The each graph node represents a call to `rec_mc`
- Represents a small fraction of the 377 function calls needed for 26 cents
- Each node indicates the amount of change for which we need to computer the number of coins
- Arrow label is coin just used

Coin Optimization

Recursive Performance

- Graph traces path of coin combinations
- Main problem is we are re-doing too many calculations
- Finding optimal change for 15 cents occurs three times
  - Itself depends into 52 function calls, each!

Coin Optimization

Recursive Performance

- We need to remember past results to avoid re-computation
- Store results for minimum number of coins in a table
- Before commuting a new minimum, we check to see if the table already has a value

Coin Optimization

Recursive Solution, Using Table Lookup

```
def rec_dc(coin_values, change, known_results):
    min_coins = change
    if change in coin_values:
        known_results[change] = 1
        return 1
    elif known_results[change] > 0:
        return known_results[change]
    else:
        for i in [c for c in coin_values if c <= change]:
            num_coins = 1 + rec_dc(coin_values,
                                   change-i,
                                   known_results)
            if num_coins < min_coins:
                min_coins = num_coins
                known_results[change] = min_coins
    return min_coins


value = 63
rec_dc([1, 5, 10, 25], value, [0]*(value+1))
```

Coin Optimization

Recursive Performance

- We now check to see if table contains the minimum number of coin for a certain amount of change
- Otherwise, we recursively compute and store the result
- Curiously, this is not Dynamic Programming; this is **memoization** — a type of caching
- However, recursion drops to 221 calls!

Coin Optimization

Recursive Performance

- Dynamic Programming is more systematic
- Starts with one cent and systematically works up
- At each step of algorithm, we already know the minimum number of coins for any smaller amount

Coin Optimization

Minimum Number of Coins Needed

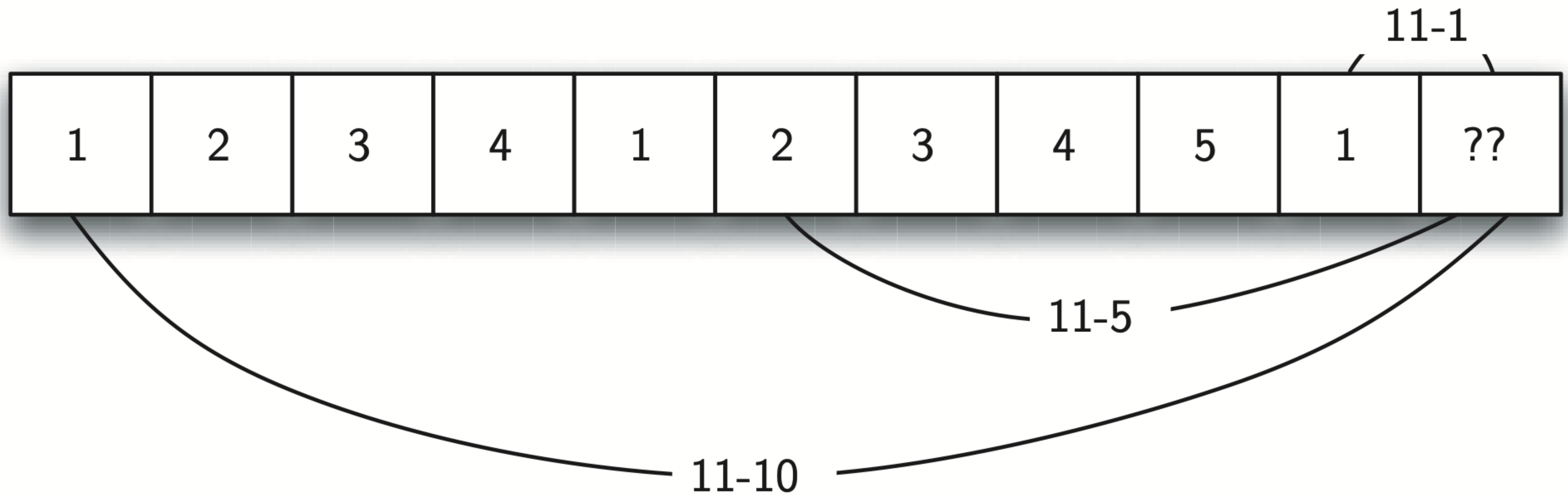
Change to Make

Step of the Algorithm

1	2	3	4	5	6	6	8	9	10	11
1										
1	2									
1	2	3								
1	2	3	4							
1	2	3	4	1						
...										
1	2	3	4	1	2	3	4	5	1	
1	2	3	4	1	2	3	4	5	1	2

Coin Optimization

Minimum Coins for 11 Cents



Coin Optimization

Dynamic Programming Solution

- Find optimum solution for 1 cent
- Find optimum solution for 2 cents using previous
- Find optimum solution for 3 cents using previous
- ...etc.

Coin Optimization

Dynamic Programming Solution

- At any amount *a*, for each denomination *d*, check the minimum coins for the (previously calculated) amount *a-d*
- We can always get from *a-d* to a with one more coin

Coin Optimization

Dynamic Programming Solution

```
def dp_make_change(coin_values, change, min_coins):
    for cents in range(change+1):
        coin_count = cents
        for j in [c for c in coin_values if c <= cents]:
            if min_coins[cents-j] + 1 < coin_count:
                coin_count = min_coins[cents-j]+1
        min_coins[cents] = coin_count
    return min_coins[change]


value = 63
dp_make_change([1, 5, 10, 25], value, [0]*(value+1))
```

Coin Optimization

Modified Dynamic Programming Solution

```
def dp_make_change_2(coin_values, change, min_coins, coins_used):
    for cents in range(change+1):
        coin_count = cents
        new_coin = 1
        for j in [c for c in coin_values if c <= cents]:
            if min_coins[cents-j] + 1 < coin_count:
                coin_count = min_coins[cents-j]+1
                new_coin = j
        min_coins[cents] = coin_count
        coins_used[cents] = new_coin
    return min_coins[change]
```

Coin Optimization

Modified Dynamic Programming Solution Cont.

```
def print_coins(coins_used, change):
    coin = change
    coin_dict = {}
    while coin > 0:
        this_coin = coins_used[coin]
        print(this_coin)
        coin = coin - this_coin
```

Coin Optimization

Modified Dynamic Programming Solution Cont.

```
cl = [1, 5, 10, 21, 25]
coins_used = [0]*64
coin_count = [0]*64
dp_make_change_2(cl, 63, coin_count, coins_used)
print_coins(coins_used, 63)
print(coins_used)
print_coins(coins_used, 52)
print(coins_used)
```

Coin Optimization

Dynamic Programming Solution

- O(NK)
  - N denominations
  - K amount of change
- By *backtracking* through the `coins_used` list, we can generate the sequence needed for the amount in question

Coin Optimization

Dynamic Programming Solution

- Do note this is **not** a recursive algorithm
- While we started with a recursive algorithm, an iterative solution is better here
- Bulk of work in `dp_make_change_2` work is handled on line 4 of the function
- All possible coins are considered here for making change for an amount `cents`

Coin Optimization

Dynamic Programming Solution

- This course is meant to expose you to a variety of different problem solving strategies
- While recursion does work here, we discover an iterative solution to this dynamic programming problem is more optimal

Questions?