COMP 550
Algorithm and Analysis

Dynamic Programming

Based on CLRS Sec. 14

Coin Change

- You have unlimited quantities of pennies (1 cent), nickels (5 cents), dimes (10 cents), and quarters (25 cents)
- You need to provide change of x cents. How to determine the minimum number of coins to equal x?
- Fill the table for $x = 61$.

Coin	Number	Value
Quarters (25¢)	2	50¢
Dimes (10¢)	1	10¢
Nickels (5¢)	0	0¢
Pennies (1¢)	1	1¢

Coin Change

- You have unlimited quantities of pennies (1 cent), nickels (5 cents), dimes (10 cents), and quarters (25 cents)
  - And there is one new coin denominations: 26 cents
- How to provide change of 61 cents?

Coin	Greedy	Optimal
Fictitious (26¢)	2 (52¢)	1 (26¢)
Quarters (25¢)	0 (0¢)	1 (25¢)
Dimes (10¢)	0 (0¢)	1 (10¢)
Nickels (5¢)	1 (5¢)	0 (0¢)
Pennies (1¢)	4 (4¢)	0 (1¢)

Coin Change

- Greedy doesn't work here
  - How to solve this problem?
  - We'll return to this problem later.
- Instead, let's consider the problem of calculating Fibonacci number

Calculate Fibonacci Number

- i-th Fibonacci number is defined as

$$F_i = \begin{cases} 0, & i = 0 \\ 1, & i = 1 \\ F_{i-1} + F_{i-2}, & i > 1 \end{cases}$$

- Fibonacci sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

Calculate Fibonacci Number

- We can compute n-th Fibonacci number recursively

```
Recursive-Fib(n)
    if (n = 0)
        return 0
    if (n = 1)
        return 1
    return Recursive-Fib(n-1)+ Recursive-Fib(n-2)
```

- $F_i = \begin{cases} 0, & i = 0 \\ 1, & i = 1 \\ F_{i-1} + F_{i-2}, & i > 1 \end{cases}$

Running Time?

- Assume that the running time is $T(n)$

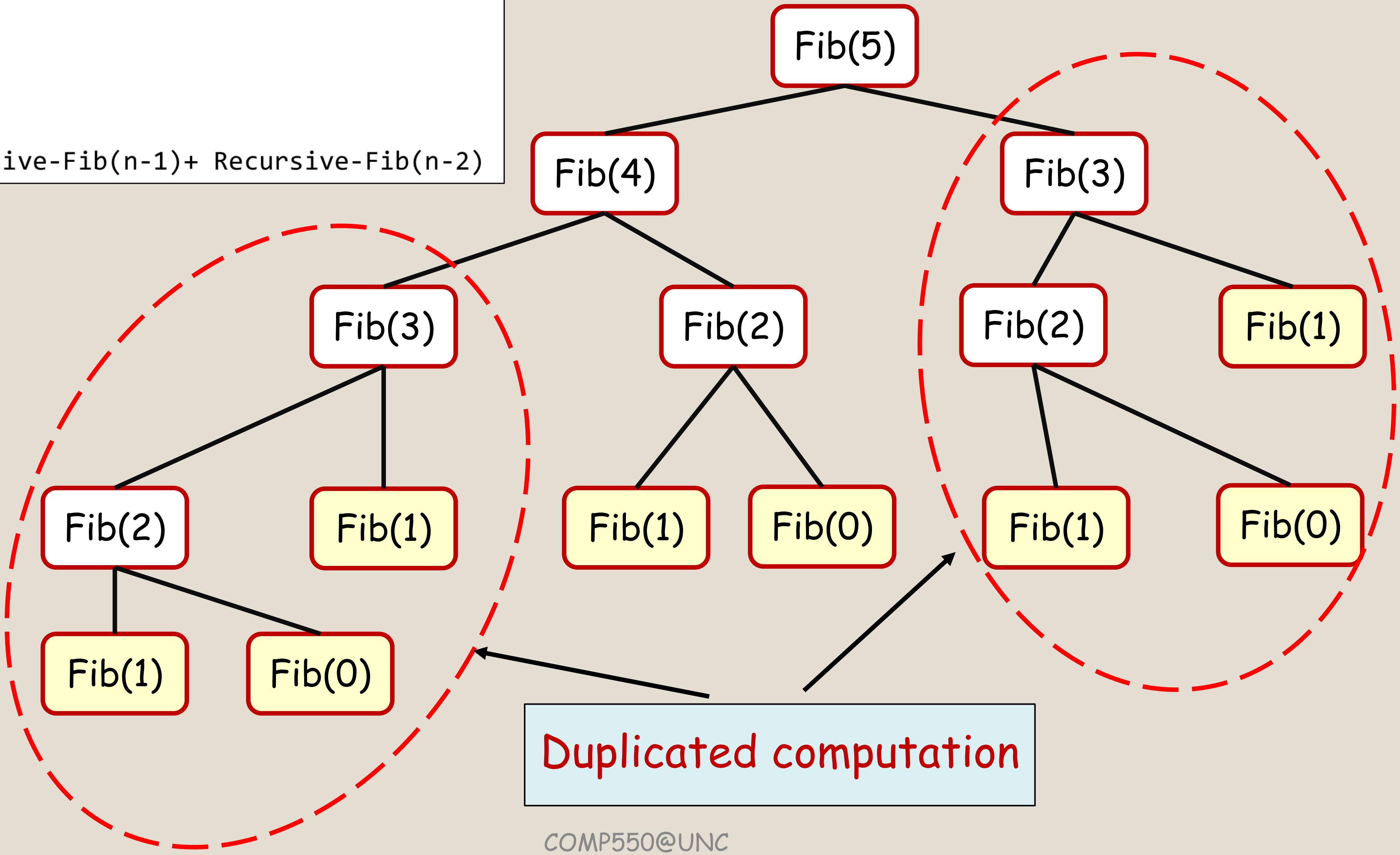
$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

- $T(n)$ is $O(2^n)$: This is a loose upper bound
- In fact, F_n and T_n has a similar recurrence:

$$T(n) = \Theta(F_n) = \Theta(\phi^n),\ \phi \text{ is the golden ratio}$$

Calculate Fibonacci Number

```
   Recursive-Fib(n)
1.    if (n = 0)
2.       return 0
3.    if (n = 1)
4.       return 1
5.    return Recursive-Fib(n-1)+ Recursive-Fib(n-2)
```



Calculate Fibonacci Number

- Computing Fibonacci number shouldn't be this inefficient
  - We need to avoid duplicated calculations!
- Idea: store the result of Fib(n) in a table after its computation and look up later if it is again needed
  - Look in the table first to check whether Fib(n) is already calculated

Top-Down Recursion with Memoization

- We can compute n-th Fibonacci number recursively

Not a typo.

```
Memoized-Fib(n)
    Fib[0:n] = array with elements initialized to NIL
    Fib[0] = 0
    Fib[1] = 1
    return Memoized-Recursive-Fib(n, Fib)
```

Don't compute if already done

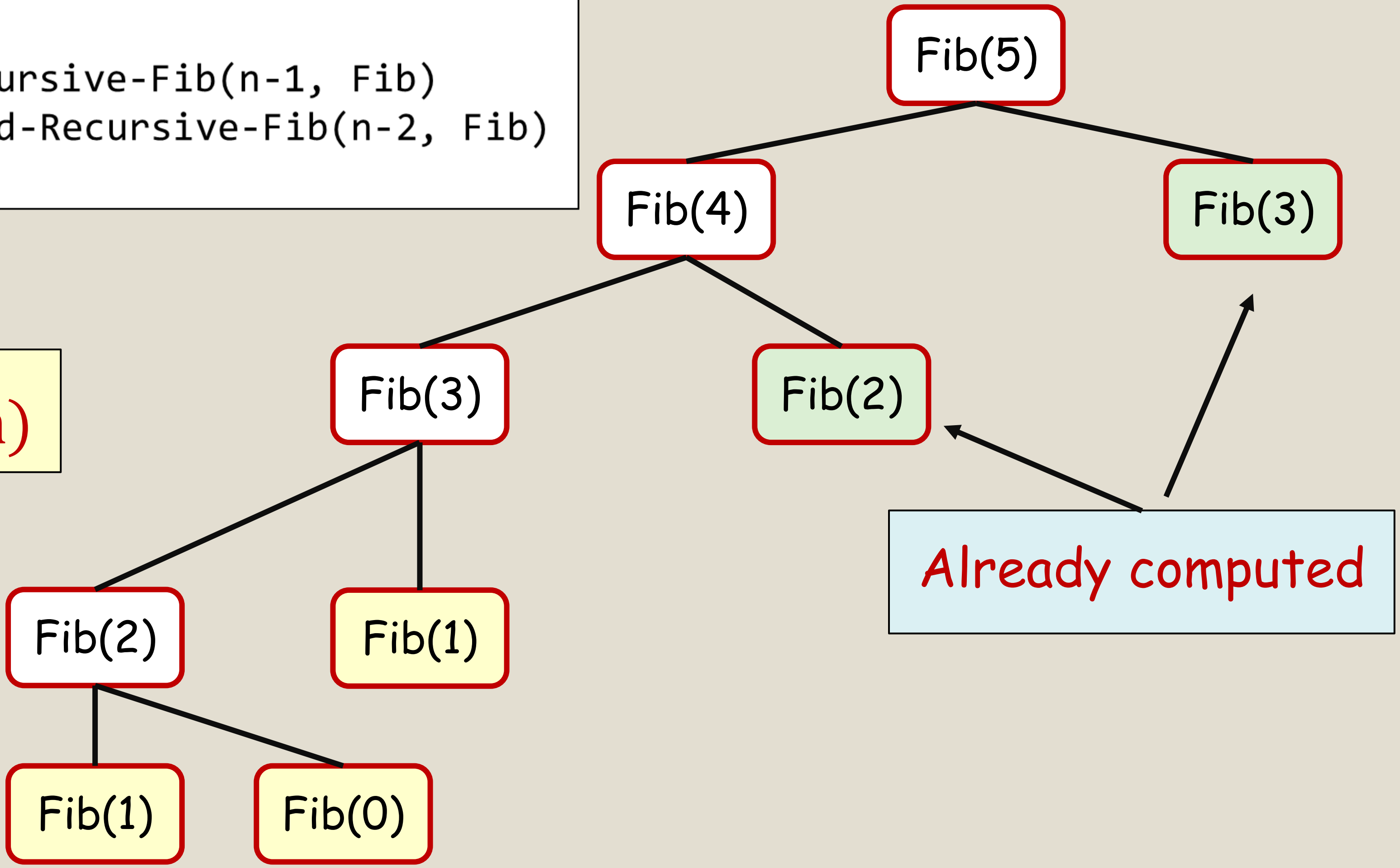
Store after computing

```
Memoized-Recursive-Fib(n, Fib)
    if Fib[n] ≠ NIL
        return Fib[n]
    Fib[n] = Memoized-Recursive-Fib(n-1, Fib)
                 + Memoized-Recursive-Fib(n-2, Fib)
    return Fib[n]
```

Top-Down Recursion with Memoization

```
Memoized-Recursive-Fib(n, Fib)
    if Fib[n] ≠ NIL
        return Fib[n]
    Fib[n] = Memoized-Recursive-Fib(n-1, Fib)
                 + Memoized-Recursive-Fib(n-2, Fib)
    return Fib[n]
```

Running Time: $\Theta(n)$



Bottom-Up Dynamic Programming

- Start with the base case: the smallest subproblem
  - The answer is easy
- Iteratively compute the larger subproblems (smallest to largest)
  - Fill up a cell of a table after each computation

```
   Bottom-Up-Fib(n)
1.     Fib[0:n] = (n+1)-element array
2.     Fib[0] = 0
3.     Fib[1] = 1
4.     for i = 2 to n
5.         Fib[i] = Fib[i-1] + Fib[i-2]
6.     return Fib[n]
```

Running Time: $\Theta(n)$

Dynamic Programming: How

- Identify a recurrence relation for your (optimal) solution
- Top-down recursive approach:
  - Store result in a table
  - Before solving a subproblem recursively, check whether that subproblem is already solved by checking the table entry
  - After solving a subproblem, store its result in the table

Dynamic Programming: How

- Identify a recurrence relation for your (optimal) solution
- Bottom-up iterative approach:
  - Start with a base case (smallest subproblem)
  - Iteratively solve the next largest subproblem that can be solved using already solved subproblems
    - Need to determine an ordering of subproblems
    - Store results in a table

Dynamic Programming: When

- The problem has optimal substructure property
  - We can solve large problems by solving smaller subproblems
- There are overlapping subproblems
  - Recursive algorithm would solve the same subproblem repeatedly (making recursive algorithm inefficient)
- The total number of distinct subproblems are "small" (i.e., polynomial w.r.t. input size)
  - If the number of distinct subproblems are "exponential", we are out of luck

Coin Change: Revisited

- You have unlimited quantities of pennies (1 cent), nickels (5 cents), dimes (10 cents), and quarters (25 cents)
  - And there is one new coin denominations: 26 cents
- How to provide change of 61 cents?

Coin	Greedy	Optimal
Fictitious (26¢)	2 (52¢)	1 (26¢)
Quarters (25¢)	0 (0¢)	1 (25¢)
Dimes (10¢)	0 (0¢)	1 (10¢)
Nickels (5¢)	1 (5¢)	0 (0¢)
Pennies (1¢)	4 (4¢)	0 (1¢)

Coin Change

- Generalized Coin Change:
  - There are m coin denominations $C = \{c_1, c_2, \dots, c_m\}$
  - Make a change of n cents.
  - Minimize the number of coins to provide n cents.
- We've seen that greedy doesn't work here
- The optimal solution in previous slide is $< 26¢,\ 25¢, 10¢ >$
  - Does this solution still have the optimal substructure property?
  - Optimal solution for change of $35¢$: $< 25¢, 10¢ >$
  - Optimal solution for change of $36¢$: $< 26¢, 10¢ >$
- With optimal substructure, there should be a way to solve recursively

Coin Change

- You have unlimited quantities of pennies (1 cent), nickels (5 cents), dimes (10 cents), quarters (25 cents), and fictitious (26 cents)
- Provide change of n cents using the minimum number of coins

- Let $change(x)$ be the minimum number of coins for x cents
  - Can we write a recurrence relation for $change(x)$?
  - First, let's consider $x \geq 26$ for simplicity.

$$change(x) = \mathbf{min} \begin{cases} change(x-26)+1 \\ change(x-25)+1 \\ change(x-10)+1 \\ change(x-5)+1 \\ change(x-1)+1 \end{cases}$$

Coin Change

- Let $change(x)$ be the minimum number of coins for x cents

$$change(x) = \mathbf{min} \begin{cases} change(x-26)+1 \\ change(x-25)+1 \\ change(x-10)+1 \\ change(x-5)+1 \\ change(x-1)+1 \end{cases}$$

- How to handle $x < 26$ cases?
  - For $x = 25$, we don't have $change(x-26)$
  - For $10 \leq x < 25$, we don't have $change(x-26)$ and $change(x-25)$
  - …

Coin Change

- Let $change(x)$ be the minimum number of coins for x cents

$$change(x) = \mathbf{min} \begin{cases} change(x-26)+1 \\ change(x-25)+1 \\ change(x-10)+1 \\ change(x-5)+1 \\ change(x-1)+1 \end{cases}$$

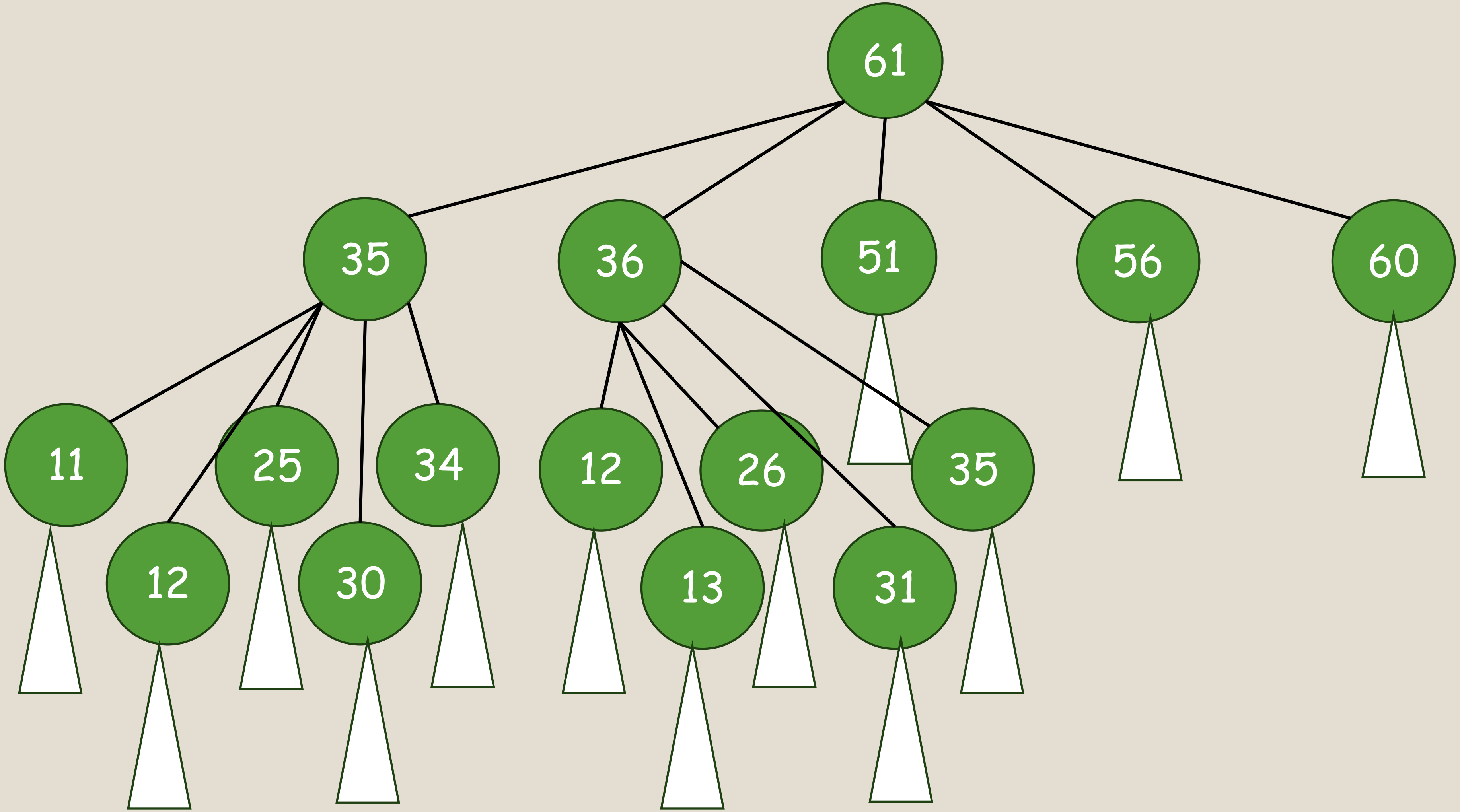
- How to handle $x < 26$ cases?
  - Compact way: define base case to allow $change(x)$ for negative x

$$change(x) = \begin{cases} 0, & x = 0 \\ \infty, & x < 0 \end{cases}$$

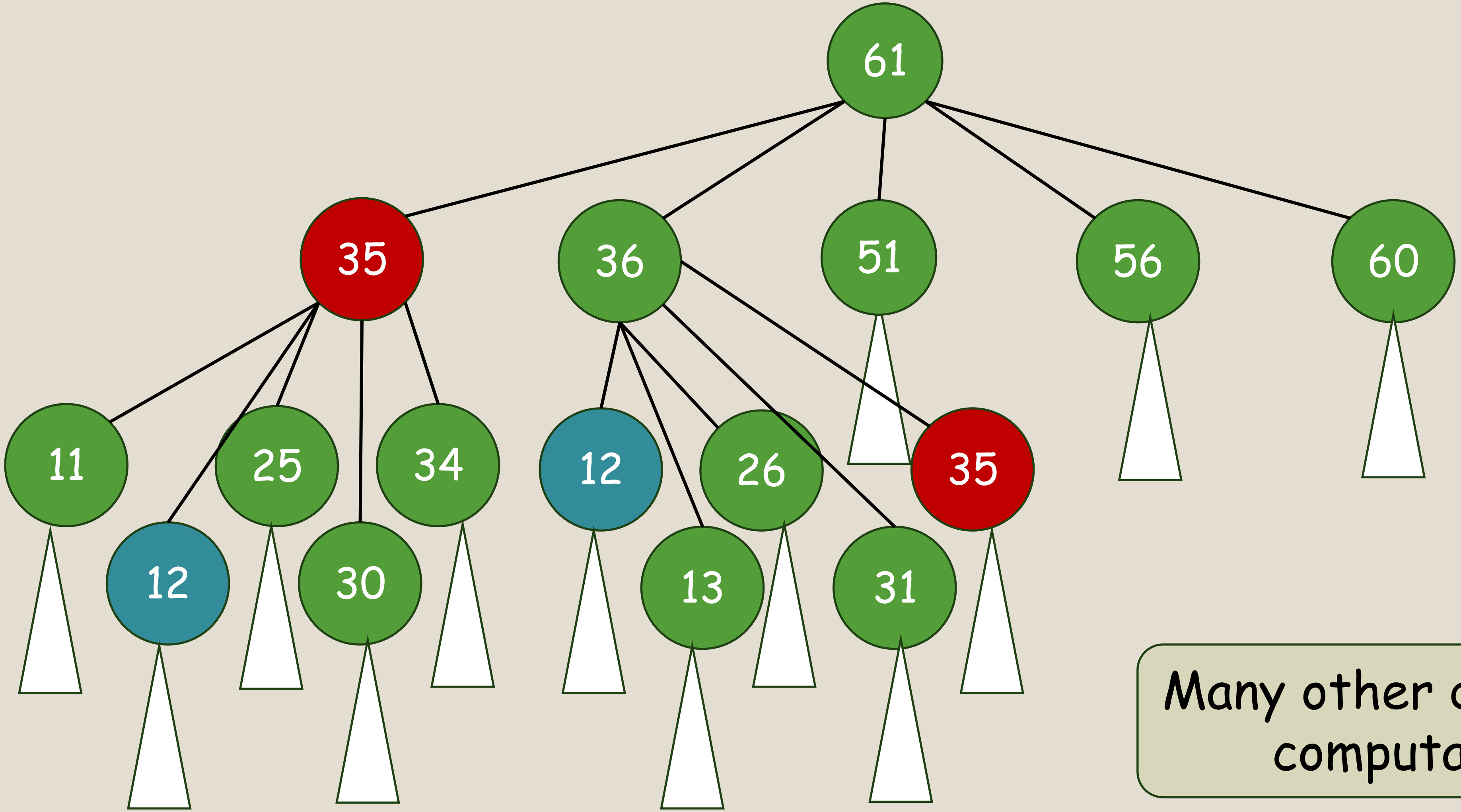
Coin Change

```
Recursive-Coin-Change(C, x)
    if (x < 0)
        return ∞
    if (x = 0)
        return 0
    min_coins = ∞
    for each c ∈ C
        num_coins = Recursive-Coin-Change(C, x-c) + 1
        min_coins = min(min_coins, num_coins)
    return min_coins
```

Coin Change



Coin Change



Many other duplicated computations

Top-Down DP

```
   Memoized-Recursive-Coin-Change(C, x, mem)
       if (x < 0)
           return ∞
       if (x = 0)
           return 0
       if mem[x] ≠ ∞   //mem[] initialized to ∞
           return mem[x]
       for each c ∈ C
           num_coins = Recursive-Coin-Change(C, x-c, mem) + 1
           mem[x] = min(mem[x], num_coins)
       return mem[x]
```

Array to store results.
What should be its size?

Bottom-Up DP

- Create a table $mem[0{:}\, n]$ and iteratively fill up the table from left to right. (Why left to right?)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0																					

Coins = {1,5,10,25,26}

Bottom-Up DP

- Create a table $mem[0{:}n]$ and iteratively fill up the table from left to right. (Why left to right?)

change(1) = **min**(change(1-1)+1, change(1-5)+1, change(1-10)+1, change(1-25)+1, change(1-26)+1)
= **min**$(0 + 1, \infty + 1, \infty + 1, \infty + 1, \infty + 1)$
= 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	1																				

Coins = {1,5,10,25,26}

Bottom-Up DP

- Create a table $mem[0{:}\,n]$ and iteratively fill up the table from left to right. (Why left to right?)

change(x) = **min**(change(x-1)+1, change(x-5)+1, change(x-10)+1, change(x-25)+1, chage(x-26)+1)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	1	2	3	4	1																

Coins = {1,5,10,25,26}

Bottom-Up DP

- Create a table $mem[0{:}n]$ and iteratively fill up the table from left to right. (Why left to right?)

change(x) = **min**(change(x-1)+1, change(x-5)+1, change(x-10)+1, change(x-25)+1, chage(x-26)+1)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	1	2	3	4	1	2	3	4	5	1	2	3	4	5	2	3	4	5	6	2	3

Coins = {1,5,10,25,26}

Bottom-Up DP

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	1	2	3	4	1	2	3	4	5	1	2	3	4	5	2	3	4	5	6	2	3

```
Bottom-Up-Coin-Change(C, n)
    mem[0:n] = an array with elements initialized to ∞
    mem[0] = 0    // 0 coins for 0 cents
    for i = 1 to n
        for each c ∈ C
            if i ≥ c
                mem[i] = min(mem[i], mem[i-c] + 1)
    return mem[n]
```

Running Time: $\Theta(n \cdot |C|)$

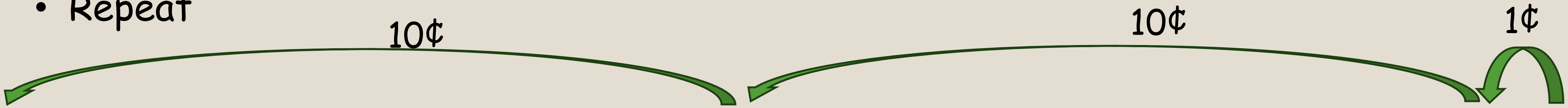
Note: this is NOT polynomial running time! It is pseudo-polynomial

Bottom-Up DP

- Steps:
  1. Determine a recurrence relation to solve the problem
  2. Determine table size and dimension to store the subproblem results
     - The recurrence relation will help
  3. Determine an order to solve the subproblems
     - Again, the recurrence relation will help
  4. Solve base cases first
  5. Iterate the subproblems in the determined ordering and solve each subproblem

How to Construct Optimal Solution?

- Track back the optimal solution.
- Suppose we want change for 21 cents.
  - We've constructed the table already.
  - Optimal solution for 21 is 3.
  - How was this "3" obtained? From one of $21-1, 21-5, 21-10, 21-25, 21-26$
    - Check which of the $21-1, 21-5, 21-10, 21-25, 21-26$ has "$3-1\ =\ 2$" in its entry
  - Repeat



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	1	2	3	4	1	2	3	4	5	1	2	3	4	5	2	3	4	5	6	2	3

Coin Change: A Different Recurrence

- Let $change(i, x)$ be the minimum number of coins for change of x cents using denominations $C[i], C[i+1], C[i+2], \dots, C[m]$

m = number of coin denom.
n = target cent value

- The solution of the main problem of changing for 61 cents is $\boldsymbol{change(1, 61)}$

$$change(i, x) = \mathbf{min} \begin{cases} change(i, x - C[i]) + 1 \\ change(i+1, x) \end{cases}$$

- To make x cents change from $C[i], C[i+1], \dots, C[m]$:
  - Case 1: We take a coin of $C[i]$ and make change of $x - C[i]$ using coins of $C[i], C[i+1], \dots, C[m]$
  - Case 2: We do not take a coin of $C[i]$ and make change of x using coins of $C[i+1], C[i+2], \dots, C[m]$

Coin Change: A Different Recurrence

- Let $change(i, x)$ be the minimum number of coins for change of x cents using denominations in $C[i], C[i+1], C[i+2], \ldots, C[m]$

m = number of coin denom.
n = target cent value

- Base cases:

$$change(i, x) = \begin{cases} 0, & x = 0 \\ \infty, & x < 0 \text{ or } i > m \end{cases}$$

- 0 coins for a change of 0 cents
- No solution for a change of –ve cents
- No solution if no denominations are left to consider

Coin Change: A Different Recurrence

Determine $change(i, x)$

m = number of coin denom.
n = target cent value

```
Another-Recursive-Coin-Change(C, m, n, i, x)
    if (x < 0 or i > m)
        return ∞
    if (x = 0)
        return 0
    taken = Another-Recursive-Coin-Change(C, m, n, i, x-C[i]) + 1
    not_taken = Another-Recursive-Coin-Change(C, m, n, i+1, x)
    min_coins = min(taken, not_taken)
    return min_coins
```

Second Bottom-Up DP

- Step 1: Determine a recurrence relation

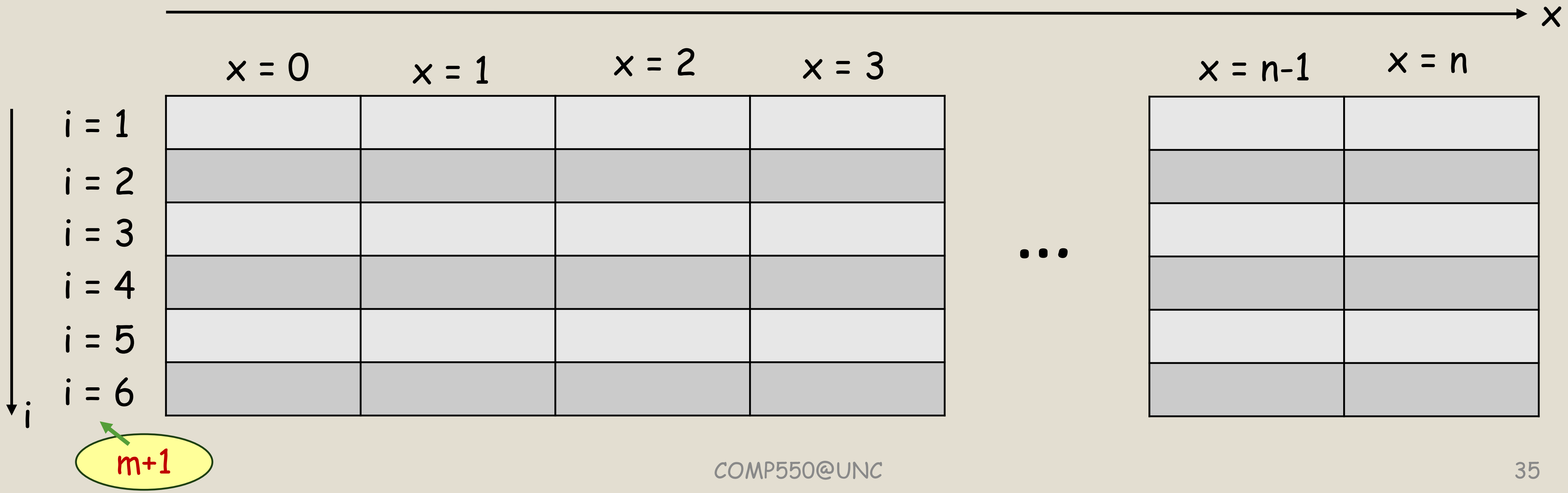
$$change(i, x) = \mathbf{min} \begin{cases} change(i, x - C[i]) + 1 \\ change(i + 1, x) \end{cases}$$

Second Bottom-Up DP

$$change(i,x) = \min \begin{cases} change(i, x - C[i]) + 1 \\ change(i+1, x) \end{cases}$$

- Step 2: Determine table size and dimension
  - Each subproblem is represented by a pair.
  - We can use a 2D table to store the result of change(i,x).
  - i (num coins) ranges from 1 to m+1, x (target) ranges from 0 to n

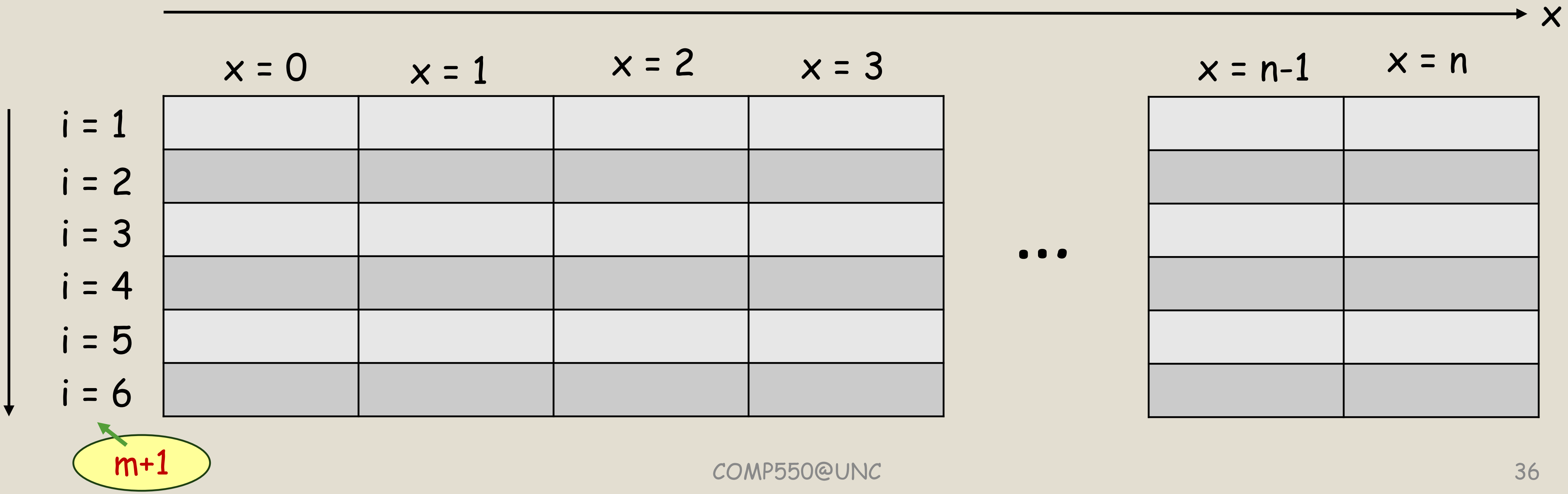
For simplicity



Second Bottom-Up DP

$$change(i,x) = \min \begin{cases} change(i, x - C[i]) + 1 \\ change(i+1, x) \end{cases}$$

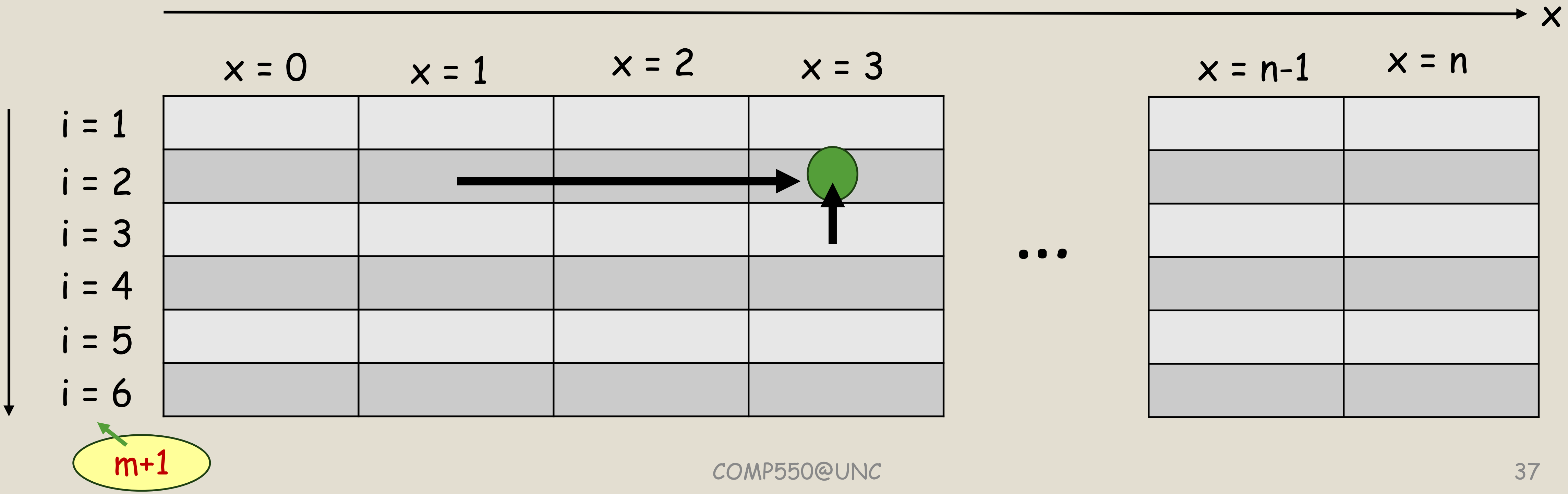
- Question:
  - Filling which cell is our main goal if we want to change n cents?
  - Cell (1,n)



Second Bottom-Up DP

$$change(i,x) = \min \begin{cases} change(i, x - C[i]) + 1 \\ change(i+1, x) \end{cases}$$

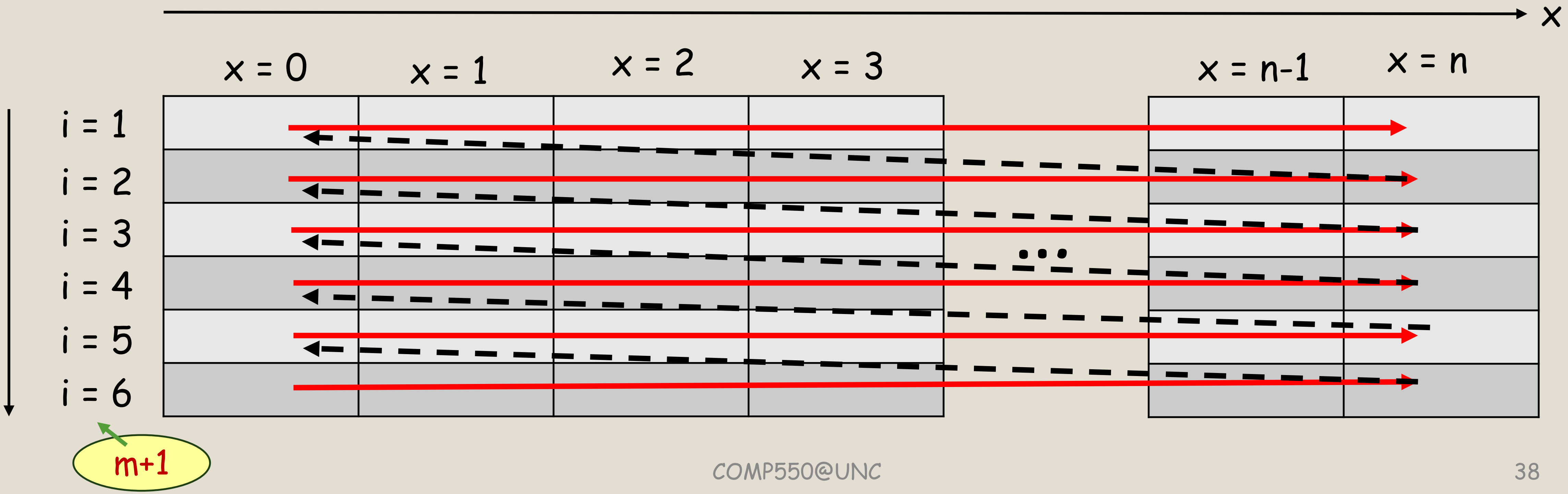
- Step 3: Determine an ordering of subproblems
  - Consider the cell $(i = 2, x = 3)$. Assume C[2]=2.
    - From the recurrence (2,3) depends on (2,1) and (3,3)



Second Bottom-Up DP

$$change(i,x) = \min \begin{cases} change(i, x - C[i]) + 1 \\ change(i+1, x) \end{cases}$$

- Step 3: Determine an ordering of subproblems
  - Fill table from bottom to up and left to right



Second Bottom-Up DP

$$change(i,x) = \min \begin{cases} change(i, x - C[i]) + 1 \\ change(i+1, x) \end{cases}$$

- Step 4: Solve base cases
  - 0 coin for 0 cents
  - ∞ coins for m+1 cents

x →

	x = 0	x = 1	x = 2	x = 3	…	x = n-1	x = n
i = 1	0						
i = 2	0						
i = 3	0						
i = 4	0						
i = 5	0						
i = 6	0	∞	∞	∞		∞	∞

m+1 (i = 6)

Second Bottom-Up DP

- **Step 5**: Iteratively fill up the table

$$change(i,x) = \min \begin{cases} change(i, x - C[i]) + 1 \\ change(i+1, x) \end{cases}$$

Example: C = {1, 6, 10}, n = 12 cents

i \ x	1	2	3	4	5	6	7	8	9	10	11	12
1	0											
2	0											
3	0											
4	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Second Bottom-Up DP

- Step 5: Iteratively fill up the table

$$change(i,x) = \min \begin{cases} change(i, x - C[i]) + 1 \\ change(i+1, x) \end{cases}$$

Example: C = {1, 6, 10}, n = 12 cents

i \ x	1	2	3	4	5	6	7	8	9	10	11	12
1	0	1	2	3	4	1	2	3	4	1	2	3
2	0	∞	∞	∞	∞	1	∞	∞	∞	1	∞	2
3	0	∞	∞	∞	∞	∞	∞	∞	∞	1	∞	∞
4	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Second Bottom-Up DP

- Step 5: Iteratively fill up the table

$$change(i,x) = \min \begin{cases} change(i, x - C[i]) + 1 \\ change(i+1, x) \end{cases}$$

```
Another-DP-Coin-Change(C, m, n)
    mem[1:m][0:n] = a new array with each cell initialized to ∞
    for i = 1 to m
        mem[i][0] = 0 //0 coins to return 0 cents
    for x = 2 to n
        mem[m+1][x] = ∞
    for i = m downto 1
        for x = 1 to n
            if(x - C[i] < 0)
                mem[i][x] = mem[i+1][x]
            else
                mem[i][x] = min(mem[i][x-C[i]] + 1 , mem[i+1][x])
    return mem[1][n]
```

Running Time: $\Theta(n \cdot |C|)$

Thank You!