DYNAMIC PROGRAMMING

CS6202 COMPUTABILITY, COMPLEXITY AND ALGORITHMS

KWANKAMOL NONGPONG, PH.D.

DYNAMIC PROGRAMMING

- Programming, in this context, refers to a tabular method, not writing the code.
- Similar to divide-and-conquer, it solves problems by combining solutions to subproblems.
- Applicable when the subproblems are not independent.
  - i.e., subproblems share subsubproblems.
- Solves every subsubproblem just once and saves its answer in a table to avoid recomputing the answer.

WHEN IS IT NORMALLY USED?

- Typically applied to optimization problems.
- There can be many possible solutions where each solution has a value.
- We wish to find an optimal solution (minimum or maximum value).

STEPS

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

- Note that step 4 can be omitted, if only the value of an optimal solution is required.

MATRIX CHAIN MULTIPLICATION

MATRIX-CHAIN MULTIPLICATION

- Given a sequence (chain) $<A_1, A_2, \ldots, A_n>$ of n matrices to be multiplied.
- Problem: Compute the product $A_1A_2\ldots A_n$

- For instance, $<A_1, A_2, A_3, A_4>$ can be fully parenthesized in 5 different ways:
  - $(A_1(A_2(A_3A_4)))$
  - $(A_1((A_2A_3)A_4))$
  - $((A_1A_2)(A_3A_4))$
  - $((A_1(A_2A_3))A_4)$
  - $(((A_1A_2)A_3)A_4)$

The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product.

MATRIX-CHAIN MULTIPLICATION

Matrix-Multiply(A, B)

```
if columns[A] != rows[B] then
    error "incompatible dimensions"
else for i = 1 to rows[A]  do
        for j = 1 to columns[B] do
            C[i, j] = 0
            for k = 1 to columns[A] do
                C[i, j] = C[i, j] + A[i, k] * B[k, j]
    return C
```

The running time shall be expressed in terms of the number of scalar multiplications.

EXAMPLE

- <A_1, A_2, A_3>
  - Dimension of A_1 is 10 x 100
  - Dimension of A_2 is 100 x 5
  - Dimension of A_3 is 5 x 50
- $(A_1A_2)A_3$) will require:
  - 10*100*5 = 5,000 scalar multiplications
  - 10*5*50 = 2,500 scalar multiplications
  - Total = 7,500 scalar multiplications

ELEMENTS OF DYNAMIC PROGRAMMING

- Optimal substructure
- Overlapping subproblems
  - Solving each subproblem once
  - Storing the solution in a table (i.e., look up is constant time)

MINIMUM COIN CHANGE

COIN CHANGE PROBLEM

- A problem of finding a number of ways of making changes for a target amount, n, using a given set of denominations $C = \{ c_1, c_2, \ldots, c_d \}$

- For instance, the US coin system is {1, 5, 10, 25, 50, 100}

EXAMPLE

- n = 4
- C = { 1, 2, 3 }
- Possible Solutions?
  - {1, 1, 1, 1}
  - {1, 1, 2}
  - {2, 2}
  - {1, 3}

MINIMUM COIN CHANGE

- Given
  - a particular amount of change n
  - a set of denominations $C = \{ c_1, c_2, \ldots, c_d \}$
- Goal:
  - Minimize the number of coins returned for a particular (given) quantity of change.

EXAMPLE 1

- n = 30
- C = { 1, 5, 10, 25, 50 }

- The minimum number of coins return is 2 coins.
- How do we obtain this number?
  - 25 + 5

EXAMPLE 2

- n = 67
- C = { 1, 5, 10, 25 }

- What is the minimum number of coins returned?

- <u>Answer</u>:   6 coins (25 + 25 + 10 + 5 + 1 + 1)

EXAMPLE 3

- n = 17
- C = { 1, 2, 3, 4 }

- What is the minimum number of coins returned?

- <u>Answer</u>: 5 coins
  - (4 + 4 + 4 + 4 + 1)
  - (4 + 4 + 3 + 3 + 3)

GREEDY ALGORITHM?

- Greedy solution does not always give an optimal solution.

- Consider the following example.
  - n = 7
  - C = { 1, 3, 4, 5 }

Greedy Solution = 3 coins
(5 + 1 + 1)

- What is the minimum number of coins returned?
- Answer: 2 coins (4 + 3)

DEMO EXAMPLE

- $n = 30$
- $C = \{ 1, 5, 10, 25, 50 \}$

DIVIDE AND CONQUER?

- Choose the smallest number of coins out of the following:
  - 1 + MinChange(29)
  - 1 + MinChange(25)
  - 1 + MinChange(20)
  - 1 + MinChange(5)

Note that
n = 30
C = { 1, 5, 10, 25, 50 }

- What seems to be the problem here?
  - What are the values of MinChange(29), MinChange(25), MinChange(20), MinChange(5) ?

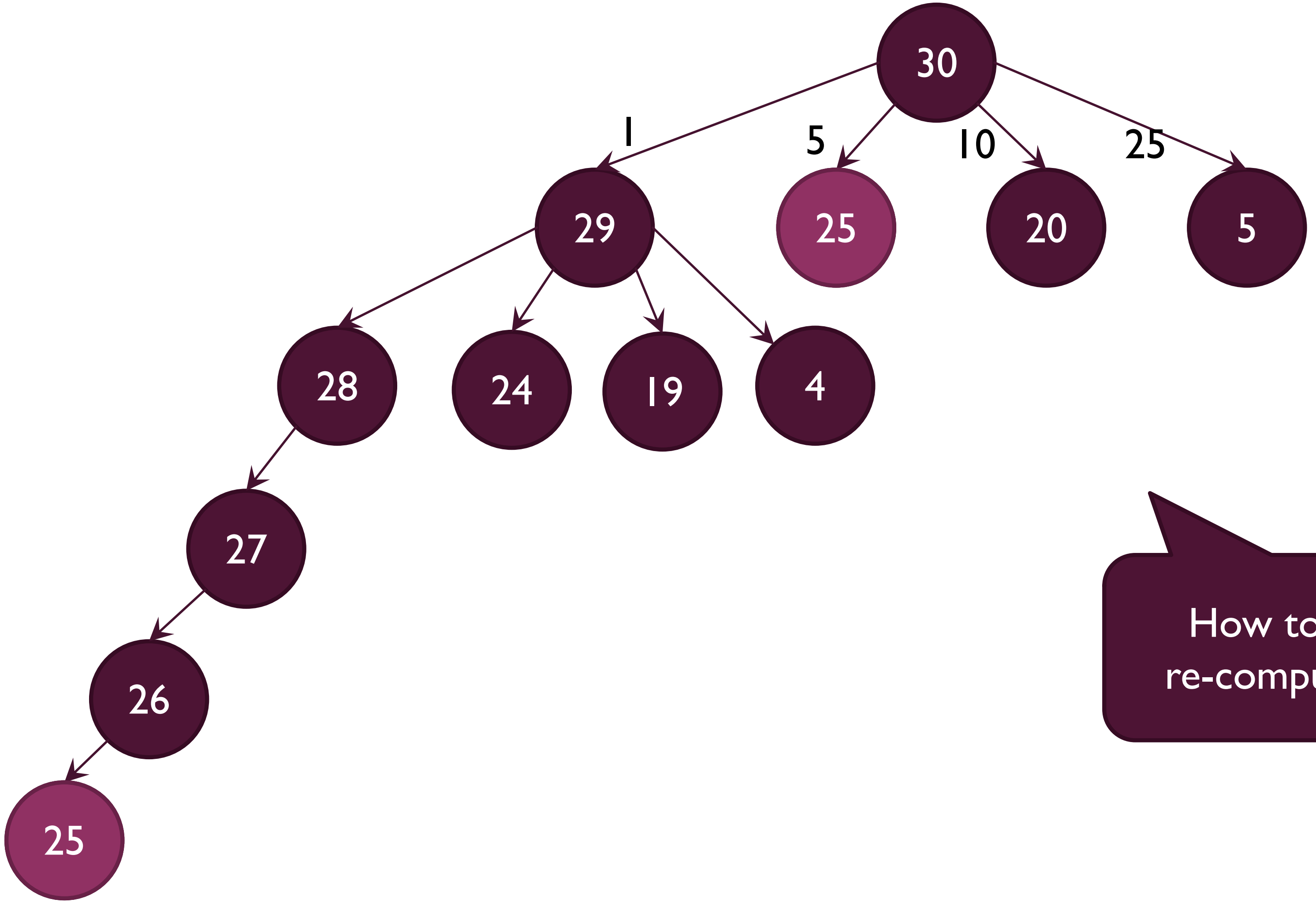
RECURSIVE ALGORITHM?

```
MinChange(n, C)
    if n = 0
        return 0
    v = ∞
    for each c in C ≤ n
        v = min { MinChange(n-c, C) + 1, v }
    return v
```

What do you think about this algorithm?

It recalculates the optimal coin combination for a given amount repeatedly!!

WHY IS IT INEFFICIENT?



How to avoid re-computation?

SAVE THE INTERMEDIATE RESULTS

```
MinChange(n, C)

    if minCoin[n] not empty
        return minCoin[n]
    if n = 0
        return 0
    for each c in C ≤ n
        v = min {MinChange(n-c) + 1, v}
    minCoin[n] = v
    return v
```

DYNAMIC PROGRAMMING

```
MinChange(n, C)

minCoin[0] = 0
for m = 1 to n
    minCoin[m] = ∞
    for each c in C ≤ m
        if minCoin[m-c] + 1 < minCoin[m]
            minCoin[m] = minCoin[m-c] + 1
return minCoin[n]
```

Let minCoin[m] be the minimum number of coins for an amount m.

EXERCISE 1: MINIMUM COIN CHANGE

Sample Input	Sample Output
30 5 1 5 10 25 50	2
67 4 1 5 10 25	6
17 4 1 2 3 4	5
7 4 1 3 4 5	2

EXERCISE 2: MINIMUM COIN CHANGE

Input	Sample Output
30 5 1 5 10 25 50	2 5 25
67 4 1 5 10 25	6 1 1 5 10 25 25
17 4 1 2 3 4	5 1 4 4 4 4
7 4 1 3 4 5	2 3 4

LONGEST INCREASING SUBSEQUENCE

LONGEST INCREASING SUBSEQUENCE

- Given a sequence of elements
- Goal
  - Find a subsequence of a given sequence in which:
    - the subsequence elements are in sorted order (lowest to highest) and
    - the subsequence is as long as possible.
- This subsequence is not necessarily contiguous, or unique.

EXAMPLE 1

- Given the sequence

5, 2, 8, 6, 3, 6, 9, 7

- The longest increasing subsequence is

2, 3, 6, 9

EXAMPLE 2

- Given the sequence

  0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15

- The longest increasing subsequence is

  0, 2, 6, 9, 13, 15.

- However, the longest increasing subsequence in this example is not unique, *i.e.*, there exists another longest increasing subsequence.

  0, 4, 6, 9, 11, 15

CONCEPT 1

- Process the sequence elements in order, maintaining the longest increasing subsequence found so far.
- Denote the sequence values as X[1], X[2], etc.
- After processing X[*i*], the algorithm will have stored values in two arrays:
  - M[*j*]
    - Stores the position *k* of the smallest value X[*k*] such that there is an increasing subsequence of length *j* ending at X[*k*] on the range $k \le i$
    - Note that we have $j \le k \le i$ here, because *j* represents the length of the increasing subsequence, and *k* represents the position of its termination.
    - Obviously, we can never have an increasing subsequence of length 13 ending at position 11. $k \le i$ by definition.
  - P[*k*]
    - Stores the position of the predecessor of X[*k*] in the longest increasing subsequence ending at X[*k*].
  - In addition the algorithm stores a variable L representing the length of the longest increasing subsequence found so far.

ALGORITHM: LONGEST INCREASING SUBSEQUENCE

```
L = 0
for i = 1 to N
    binary search for the largest positive j ≤ L
    such that X[M[j]] < X[i]
    (or set j = 0 if no such value exists)

    P[i] = M[j]
    if j == L or X[i] < X[M[j+1]]
        M[j+1] = i
        L = max(L, j+1)
```

Let X[i] be the element in the sequence

Let M[j] store the position i of the smallest value X[i]

EXAMPLE REVISITED

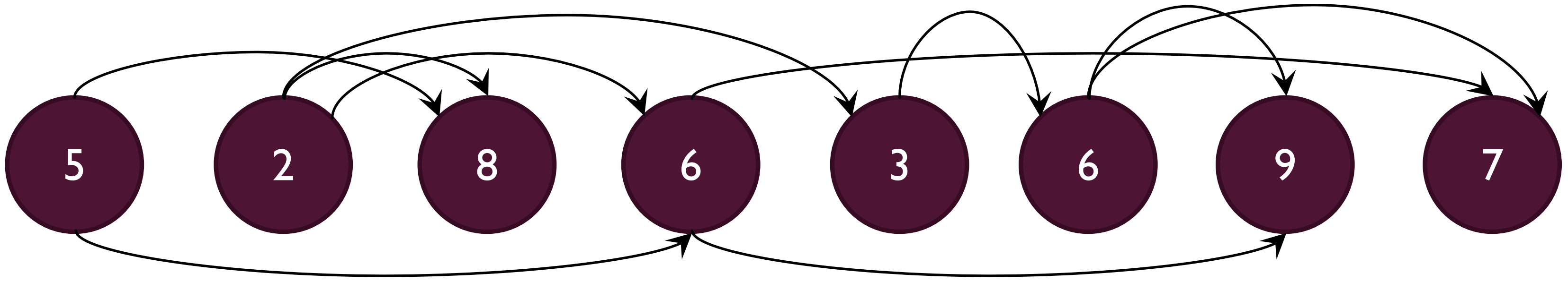
- Sequence: 5, 2, 8, 6, 3, 6, 9, 7

CONCEPT 2

- Using graph representation

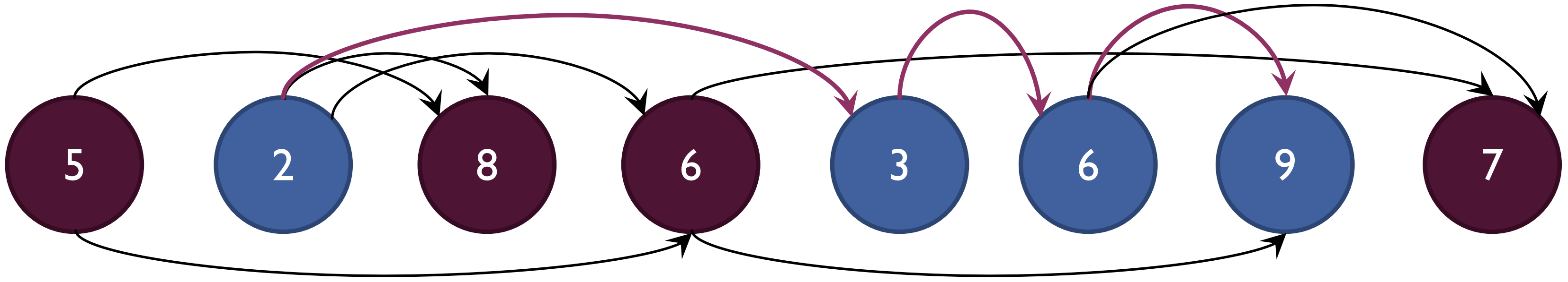
GRAPH REPRESENTATION

- Sequence: 5, 2, 8, 6, 3, 6, 9, 7



GRAPH REPRESENTATION

- Sequence: 5, 2, 8, 6, 3, 6, 9, 7



CONCEPT 2

- We define a collection of subproblems { L(j): 1 $\leq$ j $\leq$ n } with the key property that allows such subproblems to be solved in a single pass:
- Key property:
  - There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, i.e., subproblems that appear earlier in the ordering.

- In our case, each subproblem is solved using the relation

$$L(j) = 1 + \max \{ L(j) \mid (i, j) \in E \}$$

ALGORITHM

```
for j = 1 to n
    L(j) = 1 + max { L(j) : (i, j) ∈ E }
return max_j L(j)
```

Note that we only determine the length here!

What if we want to actually show the subsequence?

L(j) is the length of the longest path – the longest increasing subsequence – ending at j (plus one, why?)

EXERCISE 3: LONGEST INCREASING SUBSEQUENCE

Input	
8 5 2 8 6 3 6 9 7	4
16 0 8 4 12 2 10 6 14 1 9 5 13 3 11 7 15	6
9 2 6 3 4 1 2 9 5 8	5
5 9 7 5 3 1	1

EXERCISE 4: LONGEST INCREASING SUBSEQUENCE

Input	
8 5 2 8 6 3 6 9 7	4 2 3 6 9
16 0 8 4 12 2 10 6 14 1 9 5 13 3 11 7 15	6 0 2 6 9 11 15
9 2 6 3 4 1 2 9 5 8	5 2 3 4 5 8
5 9 7 5 3 1	1 (depends on algorithm)

0 4 6 9 13 15
depends on algorithm

QUESTIONS?