CMPT231

Lecture 8: ch15

Dynamic Programming

Ephesians 4:1-3 (NASB)

Therefore I, the **prisoner** of the Lord, implore you

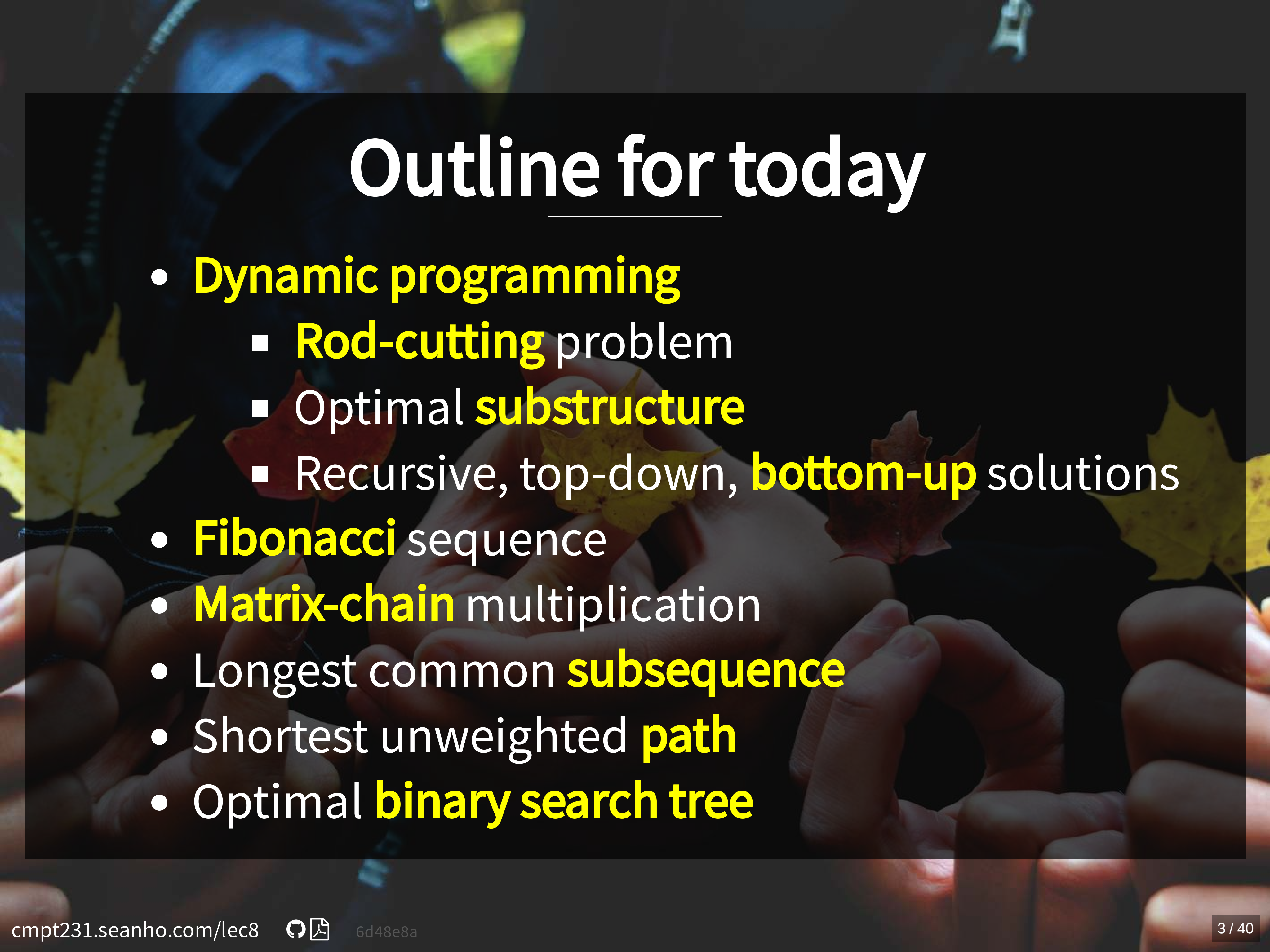
to **walk** in a manner **worthy of the calling**
with which you have been called,

with all **humility** and **gentleness**, with **patience**,
showing **tolerance** for one another in love,

being diligent to preserve the **unity** of the Spirit
in the bond of **peace**.

Outline for today

- **Dynamic programming**
  - **Rod-cutting** problem
  - Optimal **substructure**
  - Recursive, top-down, **bottom-up** solutions
- **Fibonacci** sequence
- **Matrix-chain** multiplication
- Longest common **subsequence**
- Shortest unweighted **path**
- Optimal **binary search tree**



Optimisation

- Large class of **real-world** problems consisting of
  - Finding the **max** (or min) value of some **goal** (cost) function
    over some **search space**
- **Search space**: may be discrete or continuous
  - **Dimension** of space may be low or very high (10^6 or more)
- **Goal** function: may be analytic or a black-box
  - Are its derivatives computable?

Exhaustive search is usually way too slow

![Saddle point between maxima, [Wikin
(https://commons.wikimedia.org/wiki/File%3ASaddle_Poi

Dynamic programming

- “Programming” here means **tables**
  - (e.g., linear “programming”)
- A form of **divide-and-conquer**, but
  - Store solutions to **sub-problems** for reuse
- Efficiency depends on:
  - Optimal **substructure**
  - **Overlapping** sub-problems
- Thought **process** to design solution:
  - **Recurrence** yields a recursive **top-down** solution (inefficient)
  - Top-down with **memoisation** (save sub-results)
  - **Bottom-up** (solve smaller sub-problems first,

Outline for today

- Dynamic programming
  - **Rod-cutting problem**
  - **Optimal substructure**
  - Recursive, top-down, bottom-up
- Fibonacci sequence
- Matrix-chain multiplication
- Longest common subsequence
- Shortest unweighted path
- Optimal binary search tree

Rod-cutting problem

- Steel **rods** of length i sell for $$p_i$ each (1≤i≤n)
- **Cut** a rod of length n so as to maximise **revenue**
  - Assume **cuts** are free
- Input: **price table** p = [1, 5, 8, 9]
  - So in this case rod **length** n = 4
  - **Exhaustive**: $9, 8+1, 1+8, 5+5, 5+1+1, 1+5+1, 1+1+5, 1+1+1+1
  - **Optimal** solution: two pieces of length 2: $5+5
- For fixed p, let r_n be the optimal **revenue** for length n
  - In this case, $r_n = 10$

Rod-cut: example

i:	1	2	3	4	5	6	7	8
p:	1	5	8	9	10	17	17	20

n:	0	1	2	3	4	5	6	7	8
r[n]:	0	1	5	8	10	13	17	18	22
cuts:	0	1	2	3	2+2	2+3	6	1+6, 2+2+3	2+6

May have **multiple** optimal solutions, with **same** r_n

Rod-cut: substructure

- Optimise **one cut** at a time, left to right
  - Assume the first piece **won't** be cut again
- Optimise **revenue** r_n by considering possible cuts:
  - cut of length 1: $r_n = p_1 + r_{n-1}$
  - cut of length 2: $r_n = p_2 + r_{n-2}$
  - …
  - cut of length n, i.e., **no** cuts: $r_n = p_n$
- **Recurrence** relation: $r_n = \max_{i=1\ldots n} (p_i + r_{n-i})$
- Decomposes overall task into **subproblems** r_i
- Translates directly into a **recursive** solution

Requirements for dyn prog

- To use **dynamic programming**, we need two properties:
- **Optimal substructure**:
  - Optimal solution to **subproblem** results in optimal solution to **overall** problem
  - I.e., any optimal solution can be **composed** of solutions to subproblems
- **Overlapping subproblems**:
  - Subproblems appear in **multiple** branches of recursion tree
  - Allows **reuse** of solutions, giving us efficiency

Rod-cut: optim substruct

- Let A_n be an optimal solution for **entire** length n
  - Let i be location of **first** cut in A_n
  - Let A_{n-i} be the **remaining** cuts in A_n
- **Claim**: A_{n-i} is optimal for length n-i
  - Assume **not**: let B_{n-i} be a **better** solution for n-i
    - revenue(B_{n-i}) > revenue(A_{n-i})
  - Then we can **improve** on A_n by combining this with i:
    - Let $B_n = [i, B_{n-i}]$, then
    - rev(B_n) = p[i] + rev(B_{n-i}) > p[i] + rev(A_{n-i}) = rev(A_n)
  - This **contradicts** that A_n was optimal for length

Overlapping subproblems

- Optimal substructure shows recursive solution is **correct**
- To get **efficiency** of dynamic programming, we also need to **reuse** subproblems
- **Taxonomy** of subproblems:
  - Index subproblems by **length** of rod (n)
- **Reuse** solutions to subproblems:
  - A solution for length 5 works **anywhere** within longer rods
    - Only depends on **length**, not **location**
  - So solutions to **small** rods like n=2 can be reused many times

Outline for today

- Dynamic programming
  - Rod-cutting problem
  - Optimal substructure
  - **Recursive, top-down, bottom-up solutions**
- Fibonacci sequence
- Matrix-chain multiplication
- Longest common subsequence
- Shortest unweighted path
- Optimal binary search tree

(1) Recursive top-down

```
def cutRod( p, n ):
  if ( n < 1 ): return 0
  q = -infinity
  for i = 1 .. n:
    q = max( q, p[ i ] + cutRod( p, n-i ) )
  return q
```

- **Naive** implementation of recurrence
- Recursion **tree**?
- T(n) = 2^n (#15.1-1)
  - Increase n by 1 $\Rightarrow$ **double** run time!
- E.g., cutRod(p, 2) run many times

(2) Top-down w/memoisation

```
revenue = array[ 0 .. n ] of -infinity
revenue [ 0 ] = 0
def cutRod( p, n ):
  if revenue[ n ] != -infinity:
    return revenue[ n ]
  for i = 1 .. n:
    revenue[ n ] = max( revenue[ n ], p[ i ] + cutRod( p, n-i ) )
  return revenue[ n ]
```

- **Memoisation**: cache previously-computed results
  - Need to **reset** cache for each new price table p
- cutRod(p, n) only computed **once** for each n
  - if result not in cache, takes Θ(n) to compute
  - **Complexity**: $\sum_i \Theta(i) = \Theta(n^2)$
- Can we eliminate the **recursion**?

(3) Bottom-up (dyn prog)

```
def cutRod( p, n ):
  revenue = array[ 0 .. n ] of -infinity
  revenue[ 0 ] = 0
  for j = 1 .. n:
    for i = 1 .. j:
      revenue[ j ] = max( revenue[ j ], p[ i ] + revenue[ j-i ] )
  return revenue[ n ]
```

- Start from **smaller** subproblems, caching as we go
- Doubly-nested **for** loop computes each cutRod(j)
- **Sequence** subproblems to satisfy dependencies
- **Complexity**: $\sum_i \Theta(j) = \Theta(n^2)$

Subproblem graph

- **Nodes** are the subproblems (e.g., cutRod(j))
- **Arrows** show dependencies:
  - Other nodes **needed** to compute this node
  - Like recursion **tree**, but collapsing reused nodes
- **Top-down**: performs a depth-first search down to leaves
- **Bottom-up**: must sequence nodes to resolve dependencies before reaching a node

`![node graph](static/img/Fig-15-4.svg)`

Outline for today

- Dynamic programming
  - Rod-cutting problem
  - Optimal substructure
  - Recursive, top-down, bottom-up
- **Fibonacci sequence**
- Matrix-chain multiplication
- Longest common subsequence
- Shortest unweighted path
- Optimal binary search tree

Fibonacci sequence

Recall: $F_n = F_{n-1} + F_{n-2}$, with $F_0 = F_1 = 1$

Closed-form solution: $\Theta(1)$

```
def fib( n ):
  return round( pow( phi, n ) )
```

Naive top-down: $\Theta(2^n)$

```
def fib( n ):
  if ( n < 2 ): return 1
  return fib( n-1 ) + fib( n-2 )
```

Fibonacci: dynamic prog

Top-down with memo: $\Theta(n)$

```
c = array[ 0 .. n ] of -1
c[ 0 ] = c[ 1 ] = 1
def fib( n ):
  if ( c[ n ] > 0 ): return c[ n ]
  c[ n ] = fib( n-1 ) + fib( n-2 )
  return c[ n ]
```

Bottom-up (dynamic programming): $\Theta(n)$

```
def fib( n ):
  c = array[ 0 .. n ] of -1
  c[ 0 ] = c[ 1 ] = 1
  for j = 2 .. n:
    c[ j ] = c[ j-1 ] + c[ j-2 ]
  return c[ n ]
```

Outline for today

- Dynamic programming
  - Rod-cutting problem
  - Optimal substructure
  - Recursive, top-down, bottom-up
- Fibonacci sequence
- **Matrix-chain multiplication**
- Longest common subsequence
- Shortest unweighted path
- Optimal binary search tree

Matrix-chain multiplication

- Given a **chain** of n matrices to multiply:
  - $A_1 * A_2 * A_3 * \ldots * A_n$
  - num **columns** of left matrix = num **rows** of right matrix
  - $(p_0 \times p_1)(p_1 \times p_2)\ldots(p_{n-1} \times p_n)$
- All **parenthesisations** are equivalent, but which **minimises** number of operations?
  - Recall dimensions of product matrix: $(p \times q)(q \times r) = (p \times r)$
- **Input** is a list of matrix dimensions: $\{p_i\}_0^n$
- e.g.: (5 x 500) (500 x 2) (2 x 50):
  - $(A_1 A_2)A_3$: (5)(500)(2) + (5)(2)(50) = 5500 ops

Optimal substructure

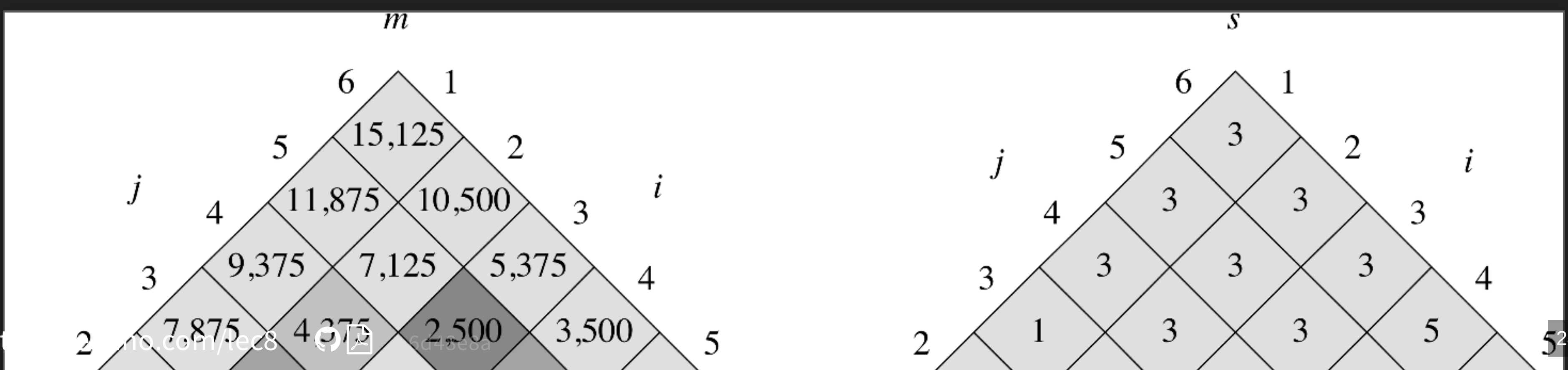
- Let c(i, j) be min **cost** to multiply $A_i, \ldots, A_j$
- Consider one **split** at a time (like rod-cut)
- If the chain from i to j is split at k, the cost is:

  $c(i, j) = c(i, k) + c(k + 1, j) + p_{i-1} p_k p_j$
- **Naive** solution uses 2n recursive calls per loop: $\Theta(2^n)$

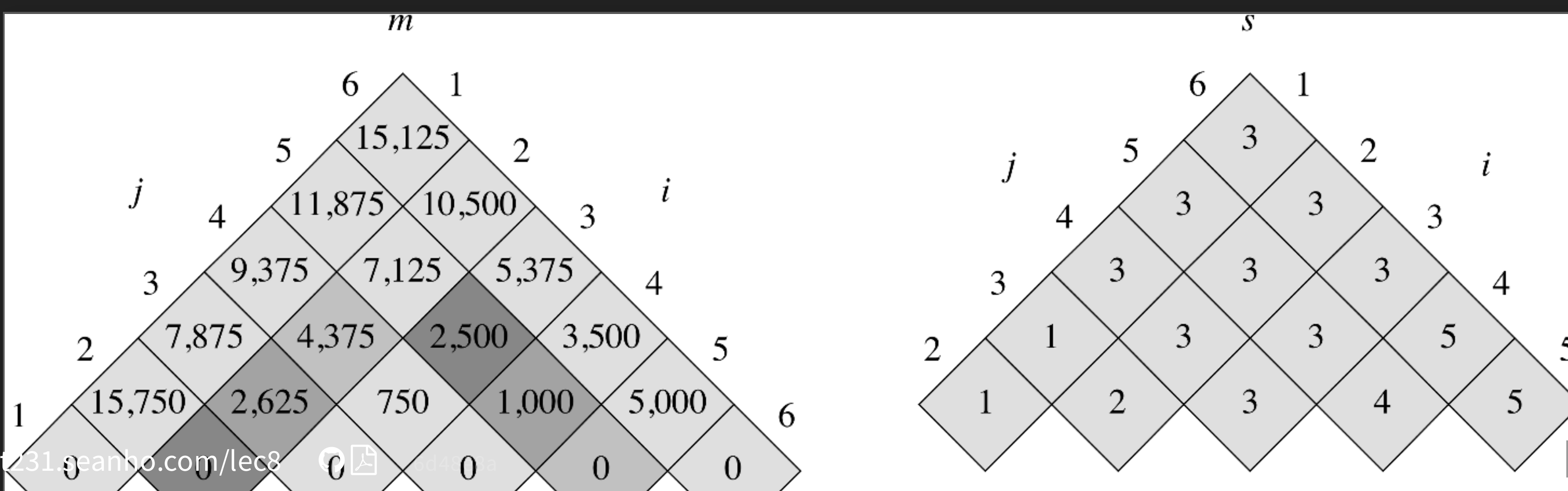
```
def matChain( p, i, j ):
  if (i == j): return 0
  cost = infinity
  for k = i .. j-1:
    cost = min( cost,
      matChain( p, i, k ) + matChain( p, k+1, j ) + p[ i-1 ] * p[
  return cost
```

Bottom-up solution

- Index subproblems by both **start** (i) and **end** (j):
  - Taxonomy is a 2D **grid** of nodes, not 1D line
- Save minimal **cost** in matrix c[i, j]
  - Save **split** point k in matrix s[i, j]
- p = [30, 35, 15, 5, 10, 20, 25] (n=7)
  - at (i, j) = (2, 5), k = 3: c[2, 5] = c[2, 3] + c[4, 5] + 35x5x20 = 7125



```
def matChain( p ):
  n = length(p) - 1
  c = array[ 1 .. n ][ 1 .. n ] of 0
  s = array[ 1 .. n-1 ][ 2 .. n ]
  for len = 2 .. n:
    for i = 1 .. n - len + 1:
      j = i + len - 1
      c[ i, j ] = infinity
      for k = i .. j-1:
q = c[ i, k ] + c[ k+1, j ] + p[ i-1 ] * p[ k ] * p[ j ]
if q < c[ i, j ]:
  c[ i, j ] = q
  s[ i, j ] = k
```



Outline for today

- Dynamic programming
  - Rod-cutting problem
  - Optimal substructure
  - Recursive, top-down, bottom-up
- Fibonacci sequence
- Matrix-chain multiplication
- **Longest common subsequence**
- Shortest unweighted path
- Optimal binary search tree

Longest common subsequence

- Given two **sequences**: $X = \{x_i\}_1^m, Y = \{y_i\}_1^n$
  - Find longest **subsequence** common to both X and Y
  - Need not be **consecutive**, but must be in **order**
- E.g.: LCS("springtime", "pioneer") = "pine"
  - LCS("horseback", "snowflake") = "oak"
  - LCS("heroically", "scholarly") = "holly"
- **Exhaustive** check:
  - for each of the 2^m **subsequences** of X,
  - **check** (in Θ(n) time) if it's a subsequence of Y

LCS: optimal substructure

- Let $X_k = \{x_i\}_1^k$ represent a **prefix** of X
- Let $Z = \{z_i\}_1^k$ be any **LCS** of X and Y
- **Theorem (part 1)**: If $x_m = y_n$,
  then (a) $z_k = x_m = y_n$
  and (b) Z_{k-1} is an **LCS** of X_{m-1} and Y_{n-1}
- **Theorem (part 2)**: If $x_m \neq y_n$ and $z_k \neq x_m$,
  then Z is an **LCS** of X_{m-1} and Y
- **Theorem (part 3)**: If $x_m \neq y_n$ and $z_k \neq y_n$,
  then Z is an **LCS** of X and Y_{n-1}
- This theorem says that an LCS of two sequences contains (as prefix) an LCS of prefixes of the two sequences

Proof of optimal substruct (part 1)

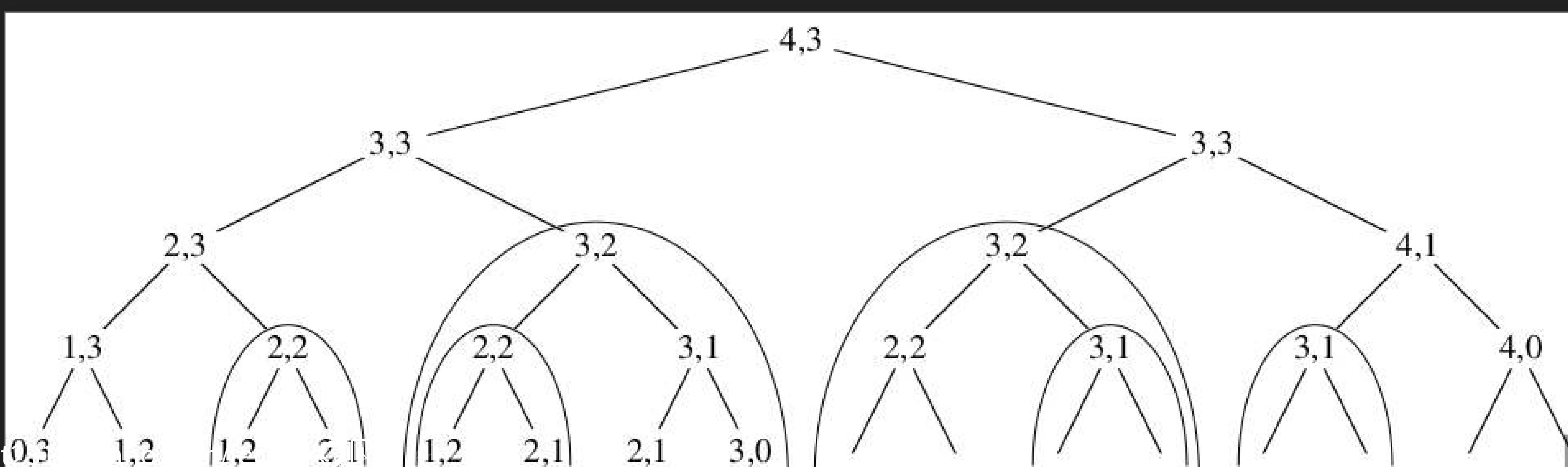
- Assume Z is an **LCS** of X and Y, and $x_m = y_n$
- **Part 1a**: Show $z_k = x_m = y_n$:
  - Assume **not**: then create Z' by **appending** x_m to Z:
    - i.e., let $Z' = (z_1, \ldots, z_k, x_m)$
  - Z' is also a subsequence of X and Y, and it's **longer** than Z
  - This **contradicts** assumption that Z was an LCS of X and Y
- **Part 1b**: Show Z_{k-1} is an **LCS** of X_{m-1} and Y_{n-1}:
  - It's certainly a **subseq** of X_{m-1} and Y_{n-1} (it just drops z_k)

Proof of opt substruct (parts 2-3)

- Assume Z is an **LCS** of X and Y, and $x_m \neq y_n$
- **Part 2** ($z_k \neq x_m$): Show Z is an LCS of of X_{m-1} and Y.
  - Let W be a subseq of X_{m-1} and Y, with length > k
  - Then W is also a subseq of X and Y, **longer** than Z
  - This **contradicts** assumption that Z was an LCS of X and Y
- **Part 3** ($z_k \neq y_n$) is **symmetric**
- Thus in all cases, any **LCS** of X and Y has a **prefix** which is an **LCS** of prefixes of X and Y

LCS recurrence

- Let c[i, j] = length of LCS of X and Y. Then
- $$c[i,j] = \begin{cases} 0 & \text{if} \quad i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if} \quad (i,j) > 0 \text{ and } x_i = \\ \max\left(c[i-1, j], c[j, i-1]\right) & \text{if} \quad (i,j) > 0 \text{ and } x_i \neq \end{cases}$$
- LCS only gets **extended** by a character in case 2
- e.g., LCS("bozo", "bat") [(3,3) on rhs should be (4,2)]



LCS solution

```
def LCSLength( x, y ):
  ( m, n ) = length( x, y )
  b[ 1 .. m ][ 1 .. n ] = new array
  c[ 0 .. m ][ 0 .. n ] = 0
```

```
  for i = 1 .. m:
    for j = 1 .. n:
      if x[ i ] == y[ j ]:
c[ i, j ] = c[ i-1, j-1 ] + 1
b[ i, j ] = “UL”
      elif c[ i-1, j ] >= c[ i, j-1
c[ i, j ] = c[ i-1, j ]
b[ i, j ] = “U”
      else:
c[ i, j ] = c[ i, j-1 ]
b[ i, j ] = “L”
```

Complexity: $\Theta(mn)$

What do "*spanking*" and "*amputation*" have in common?

		a	m	p	u	t	a	t	i	o	n
	0	0	0	0	0	0	0	0	0	0	0
s	0	0	0	0	0	0	0	0	0	0	0
p	0	0	0	1	1	1	1	1	1	1	1
a	0	1	1	1	1	1	2	2	2	2	2
n	0	1	1	1	1	1	2	2	2	2	3
k	0	1	1	1	1	1	2	2	2	2	3
i	0	1	1	1	1	1	2	2	3	3	3
n	0	1	1	1	1	1	2	2	3	3	4
g	0	1	1	1	1	1	2	2	3	3	4
				p			a		i		n

Outline for today

- Dynamic programming
  - Rod-cutting problem
  - Optimal substructure
  - Recursive, top-down, bottom-up
- Fibonacci sequence
- Longest common subsequence
- Matrix-chain multiplication
- **Shortest unweighted path**
- **Optimal binary search tree**

Shortest/longest path

- Given an unweighted **graph** (nodes + edges)
  - Find **shortest** path between given nodes u and v
- Optimal **substructure**:
  - If path is **split** at node w, then
  - **Concatenating** shortest paths u → w and w → v
  - Yields a **shortest** path from u to v through w
- What about **longest** path u → v?
  - Obviously need to rule out **cycles**
- **Concatenate** longest(u, w) + longest(w, v)?
- **Doesn't** work! Might not be **longest** u

![Counter-
example for
longest path]
(static/img/Fig-
15-6.svg)

Optimal BST

- Given sorted **keys** $\{k_i\}_1^n$ and **probabilities** $\{p_i\}_1^n$
  - Build tree to minimise expected **search cost**
- Recall cost for **successful** search is Θ(h(k)) (depth of key)
- To handle **unsuccessful** searches:
  - Add **dummy** keys $\{d_i\}_0^n$ as leaves
  - Dummy key d_i represents entire **interval** (k_{i-1}, k_i)
  - Input q_i as **probability** of d_i
- **Probabilities** over all search keys: $\sum p_i + \sum q_i = 1$
- Expected **search cost**:

$$\sum (h(k_i) + 1)p_i + \sum (h(d_i) + 1)q_i$$

Optimal substructure

- Consider one **split** at a time: choice of **root**
- Given keys $k_i, \ldots, k_j$:
  - Consider making k_r the **root** (i ≤ r ≤ j)
  - Recurse on **left** subtree: $k_i, \ldots, k_{r-1}$
  - Recurse on **right** subtree: $k_{r+1}, \ldots, k_j$
- **Demoting** a subtree increments **depth** to each node
  - Increases **search cost** by

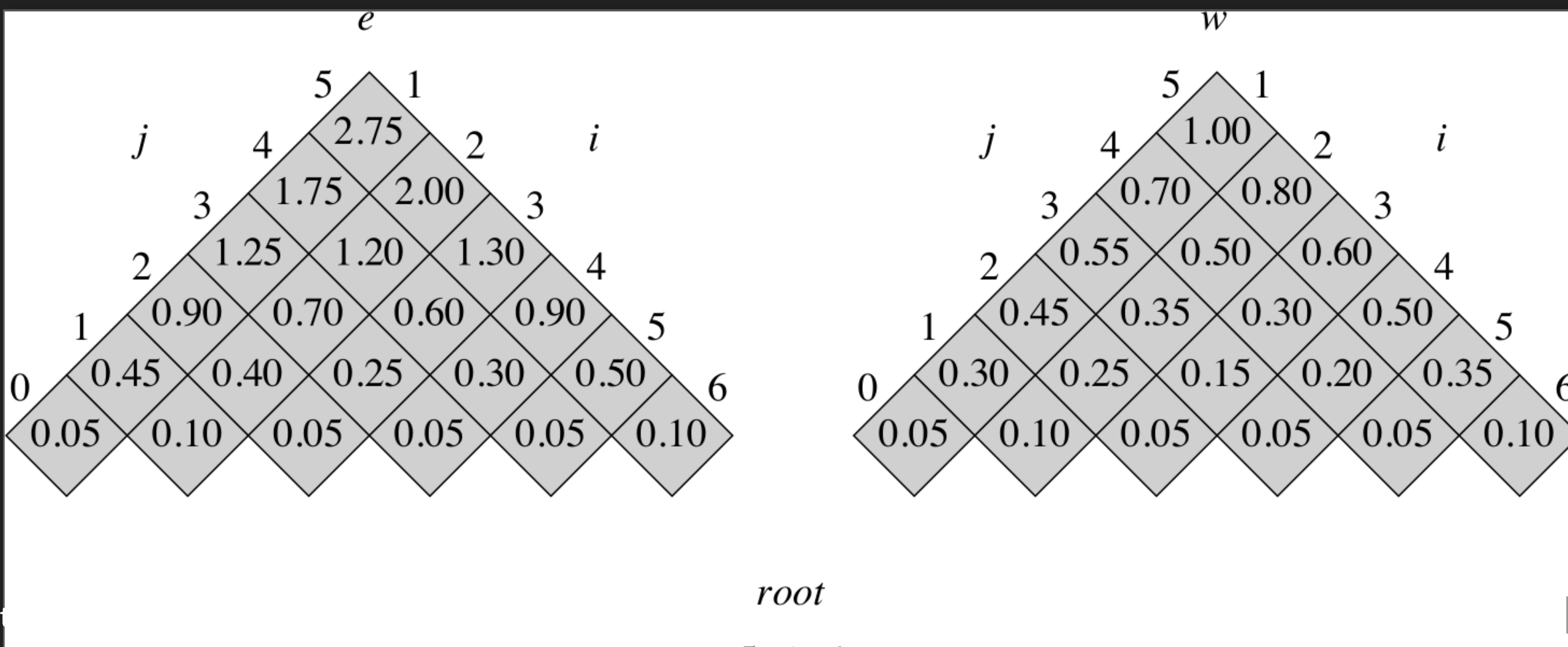
$$w(i, j) = \sum_{m=i}^{j} p_m + \sum_{m=i-1}^{j} q_m$$

- So **cost** is

$$e(i, j) = \min_{r=i \ldots j} \left[e(i, r-1) + e(r+1, j) + w(i, j) \right]$$

Optimal BST example

i	0	1	2	3	4	5
p	-	0.15	0.10	0.05	0.10	0.20
q	0.05	0.10	0.05	0.05	0.05	0.10



Complexity of dyn prog

- Optimal substructure **varies** according to
  - **Structure** of subproblem taxonomy (1D, 2D, etc)
  - How many **subproblems** used in solution
  - How many **choices** to consider per task
- **Complexity**: count **edges** in subproblem graph
- **Rod-cut**: 1D graph, 1 subprob, n choices: $\Theta(n^2)$
  - **Fibonacci**: 1D graph, 2 subprobs, no choices (not opt): Θ(n)
  - **Matrix-chain**: 2D graph, 2 subprobs, j-i choices: $\Theta(n^3)$
  - **LCS**: 2D graph, 1 subprob, ≤2 choices: Θ(mn)
  - **Optimal BST**: 2D graph, 2 subprobs, j-i+1

Outline for today

- **Dynamic programming**
  - **Rod-cutting** problem
  - Optimal **substructure**
  - Recursive, top-down, **bottom-up** solutions
- **Fibonacci** sequence
- **Matrix-chain** multiplication
- Longest common **subsequence**
- Shortest unweighted **path**
- Optimal **binary search tree**