



National Cheng Kung University



Lecture Notes for Chapter 15: Dynamic Programming

Sun-Yuan Hsieh

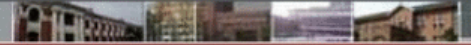
謝孫源 教授

成功大學資訊工程學系



Dynamic Programming

- ▶ Not a specific algorithm, but a technique (like divide-and-conquer).
- ▶ Developed back in the day when “programming” meant “tabular method” (like linear programming). Doesn’t really refer to computer programming.
- ▶ Used for optimization problems:
 - ▷ Find a solution with the optimal value.
 - ▷ Minimization or maximization. (We’ll see both.)



Four-step method

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



Rod cutting

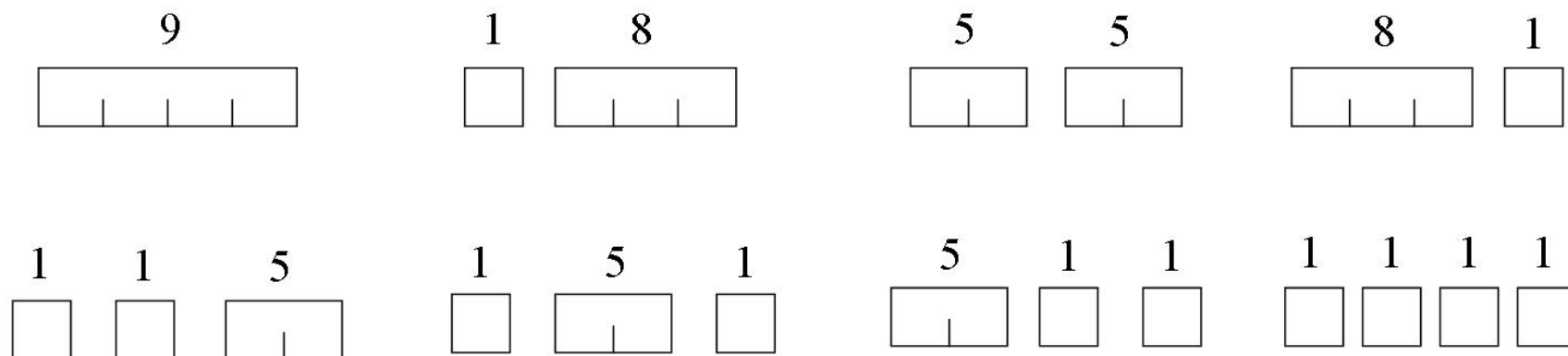
- ▶ How to cut steel rods into pieces in order to maximize the revenue you can get?
- ▶ Each cut is free . Rod lengths are always an integral number of inches.
- ▶ **Input:** A length n and table of prices p_i , for $i=1, 2, \dots, n$.
- ▶ **Output:** the maximum revenue obtainable for rods whose lengths sum to n , computed as the sum of the prices for the individual rods .
- ▶ If p_n is large enough , an optimal solution might require no cuts, i.e., just leave the rod as n inches long.



- ▶ **Example:**[Using the first 8 values from the example in the book.]

Length i	1	2	3	4	5	6	7	8
Price p_i	1	5	8	9	10	17	17	20

- ▶ Can cut up a rod in 2^{n-1} different ways, because can choose to cut or not cut after each of the first $n-1$ inches.
- ▶ Here are all 8 ways to cut a rod of length 4, with the costs from the example:



- ▶ The best way is to cut it into two 2-inch pieces, getting a revenue of $p_2 + p_2 = 5 + 5 = 10$.
- ▶ Let r_i be the maximum revenue for a rod of length i . Can express a solution as a sum of individual rod lengths.



- Can determine optimal revenues r_i for the example, by inspection:

i	r_i	optimal solution
1	1	1(no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	2 + 2
5	13	2 + 3
6	17	6 (no cuts)
7	18	1 + 6 or 2 + 2 + 3
8	22	2 + 6



- ▶ Can determine optimal revenue r_n by taking the maximum of
 - ▷ p_n : the price we get by not making a cut,
 - ▷ $r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod $n-1$ inches,
 - ▷ $r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inches and a rod of $n-2$ inches, ...
 - ▷ $r_{n-1} + r_1$.
- ▶ That is,
$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$



Optimal substructure

- ▶ **Optimal substructure:** To solve the original problem of size n , solve subproblems on smaller sizes. After making a cut, we have two subproblems. The optimal solution to the original problem incorporates optimal solutions to the subproblems. We may solve the subproblems independently.
- ▶ *Example:* For $n = 7$, one of the optimal solutions makes a cut at 3 inches, giving two subproblems, of lengths 3 and 4. We need to solve both of them optimally. The optimal solution for the problem of length 4, cutting into 2 pieces, each of length 2, is used in the optimal solution to the original problem with length 7.

A simpler way to decompose the problem



成功大學

COPYRIGHT 2002 NATIONAL CHENG KUNG UNIVERSITY



- ▶ **A simpler way to decompose the problem:** Every optimal solution has a leftmost cut. In other words, there's some cut that gives a first piece of length i cut off the left end, and a remaining piece of length $n - i$ on the right.
 - ▷ Need to divide only the remainder, not the first piece.
 - ▷ Leaves only one subproblem to solve, rather than two subproblems.



- ▷ Say that the solution with no cuts has first piece size $i = n$ with revenue p_n , and remainder size 0 with revenue $r_0 = 0$.
- ▷ Gives a simpler version of the equation for r_n :

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) .$$



Recursive top-down

► Recursive top-down

- ▷ Direct implementation of the simpler equation for r_n .
- ▷ The call $\text{CUT-ROD}(p, n)$ returns the optimal revenue r_n :

$\text{CUT-ROD}(p, n)$

If $n = 0$

return 0

$q = -\infty$

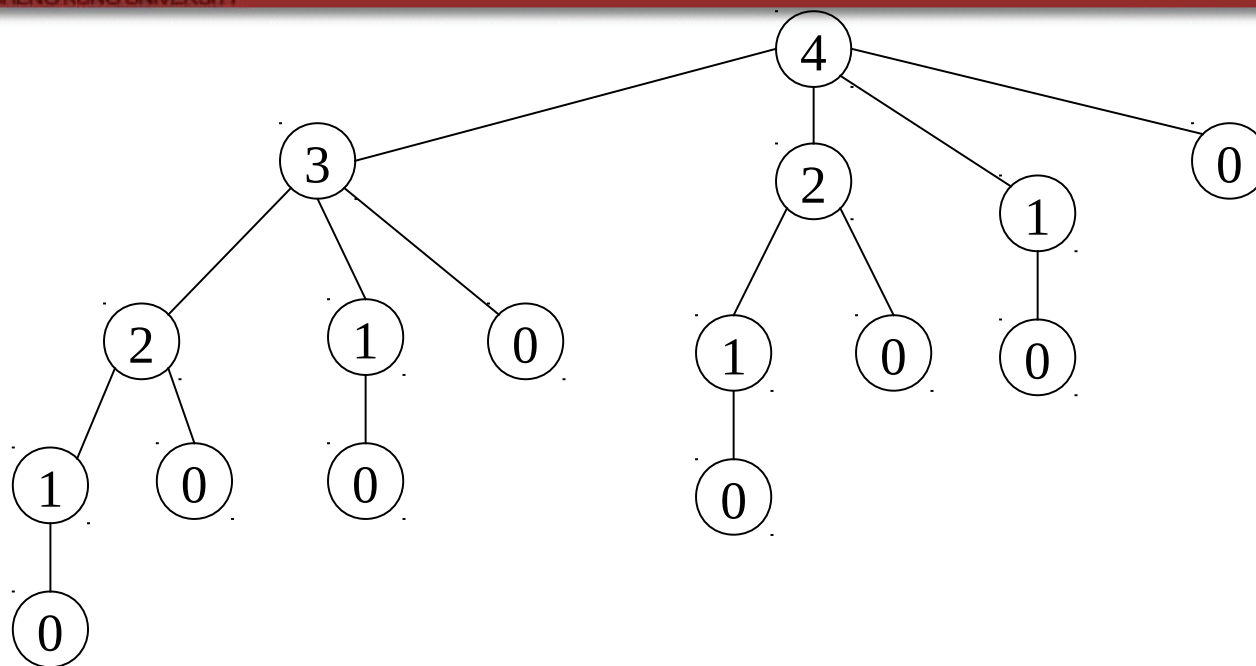
for $i = 1$ **to** n

$q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$

return q



- ▶ This procedure works, but it is terribly *inefficient*. If you code it up and run it, it could take more than an hour for $n = 40$. Running time almost doubles each time n increases by 1.
- ▶ ***Why so inefficient?***: CUT-ROD calls itself repeatedly, even on subproblems it has already solved. Here's a tree of recursive calls for $n = 4$. Inside each node is the value of n for the call represented by the node:



- Lots of repeated sub problems. Solve the subproblem for size 2 twice, for size 1 four times, and for size 0 eight times.

- ▶ *Exponential growth*: Let $T(n)$ equal the number of calls to CUT-ROD with second parameter equal to n . Then

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n \geq 1 \end{cases}$$

- ▶ Summation counts calls where second parameter is $j = n - i$.
- ▶ Solution to recurrence if $T(n) = 2^n$.



Dynamic-programming solution

- ▶ Instead of solving the same subproblems repeatedly, arrange to solve each subproblem just once.
- ▶ Save the solution to a subproblem in a table , and refer back to the table whenever we revisit the subproblem.
- ▶ “Store, don’t recompute” => time-memory trade-off.
- ▶ Can turn an exponential-time solution into a polynomial-time solution.
- ▶ Two basic approaches: top-down with memoization, and bottom-up.



Top-down with memoization

- ▶ Solve recursively, but store each result in a table.
- ▶ To find the solution to a sub problem, first look in the table. If the answer is there, use it. Otherwise, compute the solution to the sub problem and then store the solution in the table for future use.



- ▶ **Memorizing** is remembering what we have computed previously.
- ▶ Memorized version of the recursive solution, storing the solution to the subproblem of length i in array entry $r[i]$:
- ▶ **MEMOIZED-CUT-ROD(p, n)**
 - let $r[0..n]$ be a new array
 - for** $i = 0$ **to** n
 - $r[i] = -\infty$
 - return** **MEMOIZED-CUT-ROD-AUX(p, n, r)**



► MEMOIZED-CUT-ROD-AUX(p, n, r)

if $r[n] \geq 0$

return $r[n]$

if $n == 0$

$q = 0$

else $q = -\infty$

for $i = 1$ **to** n

$q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n-i, r))$

$r[n] = q$

return q



Bottom-up

- ▶ Sort the subproblems by size and solve the smaller ones first. That way, when solving a subproblem, have already solved the smaller subproblems we need.

► BOTTOM-UP-CUT-ROD(p, n)

let $r[0..n]$ be a new array

$r[0] = 0$

for $j = 1$ **to** n

$q = -\infty$

for $i = 1$ **to** j

$q = \max(q, p[i] + r[j - i])$

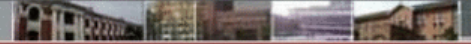
$r[j] = q$

return $r[n]$



Running time

- ▶ Both the top-down and bottom-up versions run in $\Theta(n^2)$ time.
 - ▷ Bottom-up: Doubly nested loops. Number of iterations of inner for loop forms an arithmetic series.
 - ▷ Top-down: MEMOIZED-CUT-ROD solves each subproblem just once, and it solves subproblems for sizes $0, 1, \dots, n$. To solve a subproblem of size n , the **for** loop iterates n times \Rightarrow over all recursive calls, total number of iterations forms an arithmetic series. [*Actually using aggregate analysis, which Chapter 17 covers.*]



Subproblem graphs

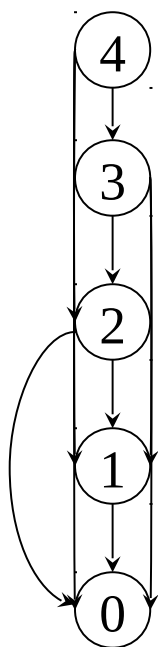
- ▶ How to understand the subproblems involved and how they depend on each other.

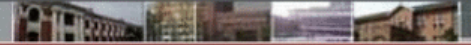
Directed graph:

- ▷ One vertex for each distinct subproblem.
- ▷ Has a direct edge (x,y) if computing an optimal solution to subproblem x *directly* requires knowing an optimal solution to subproblem y .



- **Example:** For rod-cutting problem with $n = 4$:





- ▶ Can think of the subproblem graph as a collapsed version of the tree of recursive calls, where all nodes for the same subproblem are collapsed into a single vertex, and all edges go from parent to child.
- ▶ Subproblem graph can help determine running time. Because we solve each subproblem just once, running time is sum of times needed to solve each subproblem.

- ▷ Time to compute solution to a subproblem is typically linear in the out-degree(number of outgoing edges) of its vertex.
- ▷ Number of subproblems equals number of vertices.
- ▶ When these conditions hold, running time is linear in number of vertices and edges.



Reconstructing a solution

- ▶ So far, have focused on computing the value of an optimal solution, rather than the *choices* that produced an optimal solution.
- ▶ Extend the bottom-up approach to record not just optimal values, but optimal choices. Save the optimal choices in a separate table. Then use a separate procedure to print the optimal choices.



► EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

let $r[0..n]$ and $s[0..n]$ be new arrays

$r[0] = 0$

for $j = 1$ **to** n

$q = -\infty$

for $i = 1$ **to** j

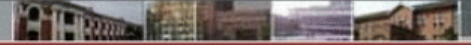
if $q < p[i] + r[j - i]$

$q = p[i] + r[j - i]$

$s[j] = i$

$r[j] = q$

return r and s



- ▶ Saves the first cut made in an optimal solution for a problem of size i in $s[i]$.
- ▶ To print out the cuts made in an optimal solution:
- ▶ PRINT-CUT-ROD-SOLUTION(p, n)
 $(r, s) = \text{EXTENDED-BOTTOM-UP-CUT-ROD}(p, n)$
 while $n > 0$
 print $s[n]$
 $n = n - s[n]$

- **Examples:** For the example, EXTENDED-BOTTOM-UP-CUT-ROD returns

i	0	1	2	3	4	5	6	7	8
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$	0	1	2	3	2	2	6	1	2

- A call to PRINT-CUT-ROD-SOLUTION($p, 8$) calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the above r and s tables. Then it prints 2, sets n to 6, prints 6, and finishes (because n becomes 0).



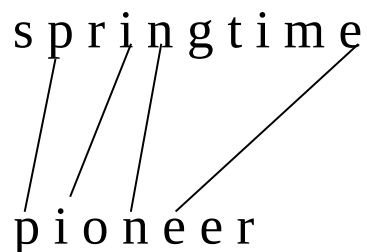
Longest common subsequence

Problem: Given 2 sequences, $X = \langle x_1, \dots, x_m \rangle$ and $Y = \langle y_1, \dots, y_n \rangle$, find a subsequence common to both whose length is longest. A subsequence doesn't have to be consecutive, but it has to be in order.

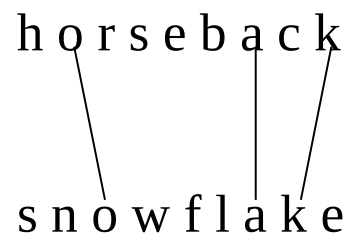


Examples: [The examples are of different types of trees.]

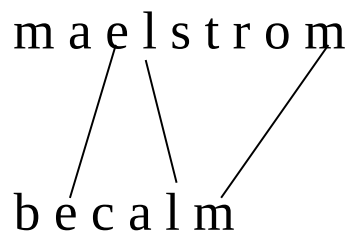
s p r i n g t i m e
p i o n e e r



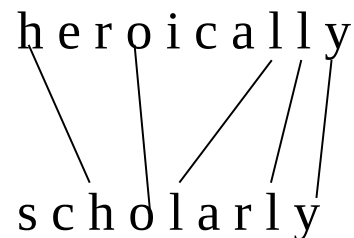
h o r s e b a c k
s n o w f l a k e



m a e l s t r o m
b e c a l m



h e r o i c a l l y
s c h o l a r l y



Brute-force algorithm:

For every subsequence of X , check whether it's a subsequence of Y .

Time $\Theta(n2^m)$:

- ▶ 2^m subsequences of X to check.
- ▶ Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, from there scan for second, and so on.

Optimal substructure

Notation:

$X_i = \text{prefix } \langle x_1, \dots, x_i \rangle$

$Y_i = \text{prefix } \langle y_1, \dots, y_i \rangle$

Theorem

Let $Z = \langle z_1, \dots, z_k \rangle$ be any LCS of X and Y .

If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .

If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y

If $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow Z$ is an LCS of X and Y_{n-1}



If $x_m = y_n$, then $z_k = x_m = y_n$

X	X_{m-1}	x_m
Y	Y_{n-1}	y_n
Z	Z_{k-1}	z_k

If $x_m \neq y_n$, then $z_k \neq x_m$

X	X_{m-2} X_{m-1} x_{m-1}	x_m
Y	Y_{n-1}	y_n
Z	Z_{k-1}	z_k

If $x_m \neq y_n$, then $z_k \neq y_n$

X	X_{m-1}	x_m
Y	Y_{n-2} Y_{n-1} y_{n-1}	y_n
Z	Z_{k-1}	z_k

Proof

1. First show that $z_k = x_m = y_n$. Suppose not. Then make a subsequence $Z' = \langle z_1, \dots, z_k, x_m \rangle$. It's a common subsequence of X and Y and has length $k + 1$
 $\Rightarrow Z'$ is a longer common subsequence that $Z \Rightarrow$ contradicts Z being an LCS.
 Now show Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} . Clearly, it's a common subsequence.
 Now suppose there exists a common subsequence W of X_{m-1} and Y_{n-1} that's longer than $Z_{k-1} \Rightarrow$ length of $W \geq k$. Make subsequence W' by appending x_m to W . W' is common subsequence of X and Y , has length $\geq k + 1 \Rightarrow$ contradicts Z being an LCS.
2. If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y . Suppose there exists a subsequence W of X_{m-1} and Y with length $> k$. Then W is a common subsequence of X and $Y \Rightarrow$ contradicts Z being an LCS.
3. Symmetric to 2.

Therefore, an LCS of two sequences contains as a prefix an LCS of prefixes of the sequences.



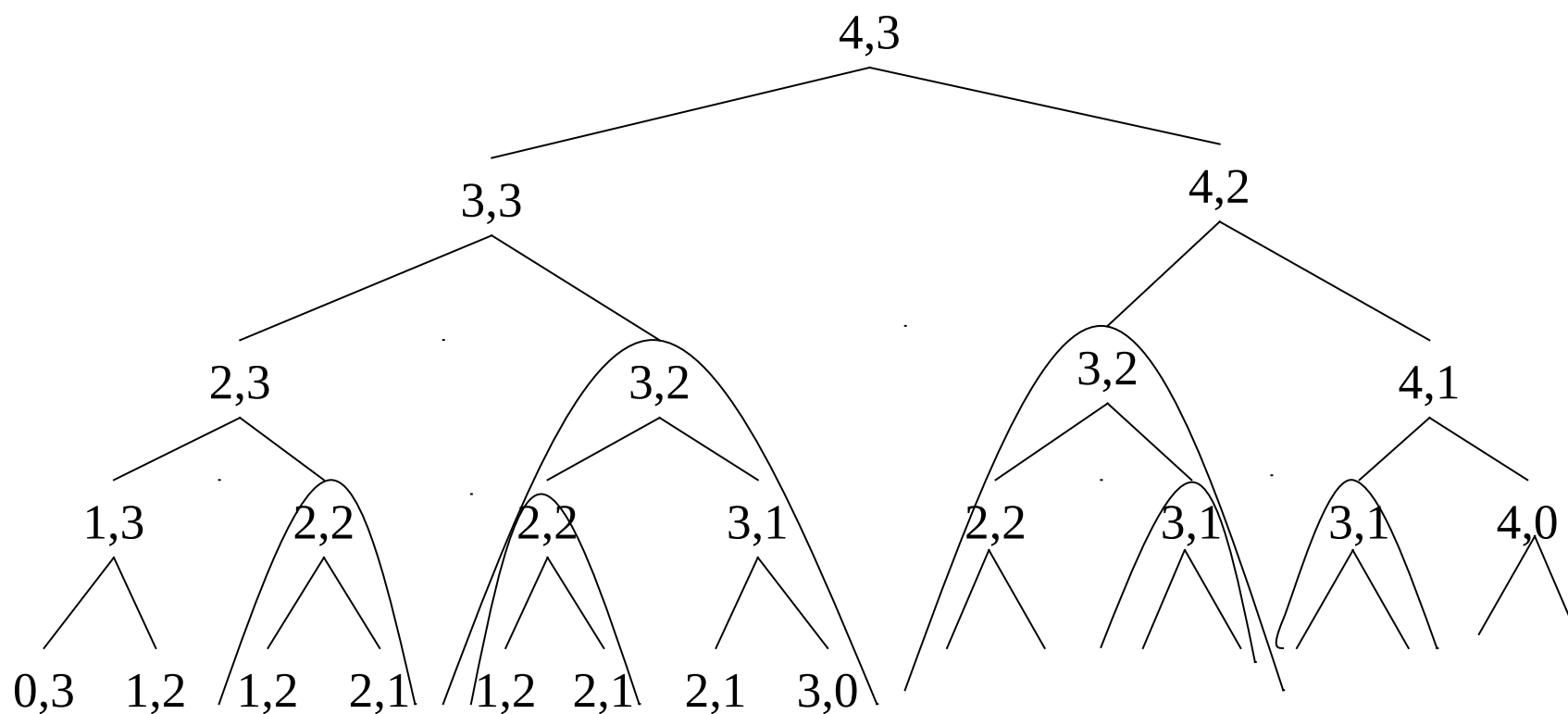
Recursive formulation

Define $c[i, j]$ = length of LCS of X_i and Y_j . We want $c[m, n]$.

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Again, we could write a recursive algorithm based on this formulation.

Try with bozo, bat.



- Lots of repeated subproblems
- Instead of recomputing, store in a table.



Compute length of optimal solution

LCS-Length(X, Y, m, n)

1. **for** $i \leftarrow 1$ **to** m
2. **do** $c[i, 0] \leftarrow 0$
3. **for** $j \leftarrow 0$ **to** n
4. **do** $c[0, j] \leftarrow 0$
5. **for** $i \leftarrow 1$ **to** m
6. **do for** $j \leftarrow 1$ **to** n
7. **do if** $x_i = y_j$
8. **then** $c[i, j] \leftarrow c[i - 1, j - 1] + 1$
9. $b[i, j] \leftarrow \text{“}\nwarrow\text{”}$
10. **else if** $c[i - 1, j] \geq c[i, j - 1]$
11. **then** $c[i, j] \leftarrow c[i - 1, j]$
12. $b[i, j] \leftarrow \text{“}\uparrow\text{”}$
13. **else** $c[i, j] \leftarrow c[i, j - 1]$
14. $b[i, j] \leftarrow \text{“}\leftarrow\text{”}$
15. **return** c and b

PRINT-LCS(b, X, i, j)

1. **if** $i = 0$ or $j = 0$
2. **then return**
3. **if** $b[i, j] = \text{“}\nwarrow\text{”}$
4. **then** PRINT-LCS($b, X, i - 1, j - 1$)
5. print x_i
6. **else if** $b[i, j] = \text{“}\uparrow\text{”}$
7. **then** PRINT-LCS($b, X, i - 1, j$)
8. **else** PRINT-LCS($b, X, i, j - 1$)

- ▶ Initial call is PRINT-LCS(b, X, m, n).
- ▶ $b[i, j]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j .
- ▶ When $b[i, j] = \nwarrow$, we have extended LCS by one character. So longest common subsequence = entries with \nwarrow in them.

Demonstration: show only $c[i, j]$:

		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

Time: $\Theta(mn)$



Optimal binary search trees

[Also new in the second edition.]

- ▶ Given sequence $K = (k_1, k_2, \dots, k_n)$ of n distinct keys, sorted ($k_1 < k_2 < \dots < k_n$).
- ▶ Want to build a binary search tree from the keys.
- ▶ For k_i , have probability p_i that a search is for k_i .
- ▶ Want BST with minimum expected search cost.
- ▶ Actual cost = # of items examined.

For key k_i , cost = $\text{depth}_T(k_i) + 1$, where $\text{depth}_T(k_i)$ = depth of k_i in BST T .

E [search cost in T]

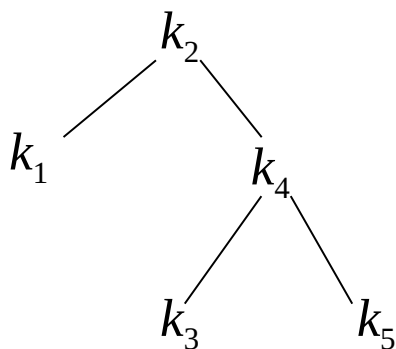
$$\begin{aligned} &= \sum_{i=1}^n (\text{depth}_T(k_i) + 1) \cdot p_i \\ &= \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^n p_i \\ &= 1 + \sum_{i=1}^n \text{depth}_T(k_i) \cdot p_i \quad (\text{since probabilities sum to 1}) \quad (*) \end{aligned}$$

[Similar to optimal BST problem in the book, but simplified here: we assume that all searches are successful. Book has probabilities of searches between keys in tree.]



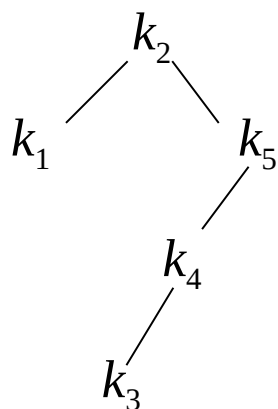
i	1	2	3	4	5
p_i	.25	.2	.05	.2	.3

Example:



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	.25
2	0	0
3	2	.1
4	1	.2
5	2	.6
		<hr/> 1.15

Therefore, $E[\text{search cost}] = 2.15$.



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	.25
2	0	0
3	3	.15
4	2	.4
5	1	.3

1.10

Therefore, $E[\text{search cost}] = 2.10$, which turns out to be optimal.



Observations:

- ▶ Optimal BST might not have smallest height.
- ▶ Optimal BST might not have highest-probability key at root.

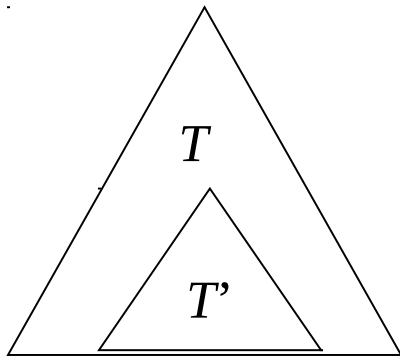
Build by exhaustive checking?

- ▶ Construct each n -node BST.
- ▶ For each, put in keys.
- ▶ Then compute expected search cost.
- ▶ But there are $\Omega(4^n/n^{3/2})$ different BSTs with n nodes.



Optimal substructure

Consider any subtree of a BST. It contains keys in a contiguous range k_i, \dots, k_j
for some $1 \leq i \leq j \leq n$.

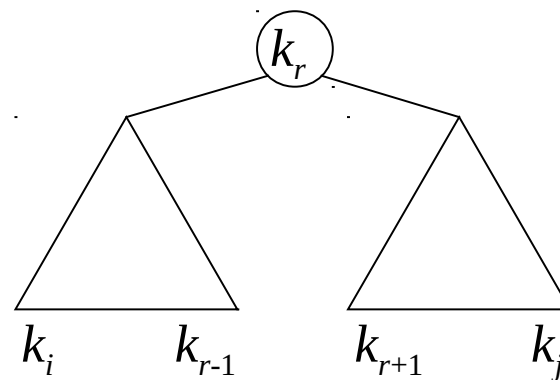


If T is an optimal BST and T contains subtree T' with keys k_i, \dots, k_j , then T' must be an optimal BST for keys k_i, \dots, k_j .

Proof Cut and paste. ■

Use optimal substructure to construct an optimal solution to the problem from optimal solutions to subproblems:

- ▶ Given keys k_i, \dots, k_j (the problem).
- ▶ One of them, k_r , where $i \leq r \leq j$, must be the root.
- ▶ Left subtree of k_r contains k_i, \dots, k_{r-1} .
- ▶ Right subtree of k_r contains k_{r+1}, \dots, k_j .



- ▶ If
 - ▷ examine all candidate roots k_r , for $i \leq r \leq j$, and
 - ▷ we determine all optimal BSTs containing k_i, \dots, k_{r-1} and containing k_{r+1}, \dots, k_j ,then we're guaranteed to find an optimal BST for k_i, \dots, k_j .



Recursive solution

Subproblem domain:

- ▶ Find optimal BST for k_i, \dots, k_j , where $i \geq 1, j \leq n, j \geq i - 1$.
- ▶ When $j = i - 1$, the tree is empty.

Define $e[i, j]$ = expected search cost of optimal BST for k_i, \dots, k_j .

If $j = i - 1$, then $e[i, j] = 0$.

If $j \geq i$,

- ▶ Select a root k_r , for some $i \leq r \leq j$.
- ▶ Make an optimal BST with k_i, \dots, k_{r-1} as the left subtree.
- ▶ Make an optimal BST with k_{r+1}, \dots, k_j as the right subtree.
- ▶ Note: when $r = i$, left subtree is k_i, \dots, k_{i-1} ; when $r = j$, right subtree is k_{j+1}, \dots, k_j .

When a subtree becomes a subtree of a node:

- ▶ Depth of every node in subtree goes up by 1.
- ▶ Expected search cost increases by

$$w(i, j) = \sum_{l=i}^j p_l \quad (\text{refer to equation } (*)).$$

If k_r is the root of an optimal BST for k_i, \dots, k_j :

$$e[i, j] = p_r + (e[i, r - 1] + w(i, r - 1)) + (e[r + 1, j] + w(r + 1, j)).$$

$$\text{But } w(i, j) = w(i, r - 1) + p_r + w(r + 1, j).$$

$$\text{Therefore, } e[i, j] = e[i, r - 1] + e[r + 1, j] + w(i, j).$$



This equation assumes that we already know which key is k_r .

We don't.

Try all candidates, and pick the best one:

$$e[i, j] = \begin{cases} 0 & \text{if } j = i-1, \\ \min_{i \leq r \leq j} \{ e[i, r-1] + e[r+1, j] + w(i, j) \} & \text{if } i \leq j. \end{cases}$$

Could write a recursive algorithm...



Computing an optimal solution

As “usual,” we’ll store the values in a table:

$$e[1..n+1, 0..n]$$

can store $e[n+1, n]$ can store $e[1, 0]$

- ▶ Will use only entries $e[i, j]$, where $j \geq i - 1$.
- ▶ Will also compute
 $\text{root}[i, j] = \text{root of subtree with keys } k_i, \dots, k_j, \text{ for } 1 \leq i \leq j \leq n.$



One other table...don't recompute $w(i, j)$ from scratch every time we need it.

(Would take $\Theta(j - i)$ additions.)

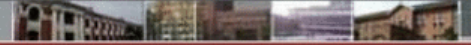
Instead:

- ▶ Table $w[1..n + 1, 0..n]$
- ▶ $w[i, i - 1] = 0$ for $1 \leq i \leq n$
- ▶ $w[i, j] = w[i, j - 1] + p_j$ for $1 \leq i \leq j \leq n$

Can compute all $\Theta(n^2)$ values in $O(1)$ time each.

OPTIMAL-BST(p, q, n)

1. **for** $i \leftarrow 1$ **to** $n + 1$
2. **do** $e[i, i - 1] \leftarrow 0$
3. $w[i, i - 1] \leftarrow 0$
4. **for** $l \leftarrow 1$ **to** n
5. **do for** $i \leftarrow 1$ **to** $n - l + 1$
6. **do** $j \leftarrow i + l - 1$
7. $e[i, j] \leftarrow \infty$
8. $w[i, j] \leftarrow w[i, j - 1] + p_j$
9. **for** $r \leftarrow i$ **to** j
10. **do** $t \leftarrow e[i, r - 1] + e[r + 1, j] + w[i, j]$
11. **if** $t < e[i, j]$
12. **then** $e[i, j] \leftarrow t$
13. $root[i, j] \leftarrow r$
14. **return** e and $root$



First **for** loop initializes e , w entries for subtrees with 0 keys.

Main **for** loop:

- ▶ Iteration for l works on subtrees with l keys.
- ▶ Idea: compute in order of subtree sizes, smaller (1 key) to larger (n keys).

For example at beginning:

i	1	2	3	4	5
p_i	.25	.2	.05	.2	.3

		j					
i	e	0	1	2	3	4	5
		0	.25	.65	.8	1.25	2.10
2		0	.2	.3	.75	1.35	
3			0	.05	.3	.85	
4				0	.2	.7	
5					0	.3	
6						0	

p_i



		<i>j</i>					
<i>i</i>	<i>w</i>	0	1	2	3	4	5
	1	0	.25	.45	.5	.7	1.0
	2		0	.2	.25	.45	.75
	3			0	.05	.25	.55
	4				0	.2	.5
	5					0	.3
	6						0

		<i>j</i>				
<i>i</i>	<i>root</i>	1	2	3	4	5
	1	1	1	1	2	2
	2		2	2	2	4
	3			3	4	5
	4				4	5
	5					5

Time: $O(n^3)$: for loops nested 3 deep, each loop index takes on $\leq n$ values.
Can also show $\Omega(n^3)$. Therefore, $\Theta(n^3)$.

Construct an optimal solution

CONSTRUCT-OPTIMAL-BST(*root*)

$r \leftarrow \text{root}[1, n]$

print “ k ” _{r} “is the root”

CONSTRUCT-OPT-SUBTREE(1, $r - 1$, r , “left”, *root*)

CONSTRUCT-OPT-SUBTREE($r + 1$, n , r , “right”, *root*)

CONSTRUCT-OPT-SUBTREE(*i*, *j*, *r*, *dir*, *root*)

if $i \leq j$

 then $t \leftarrow \text{root}[i, j]$

 print “ k ” _{t} “is” *dir* “child of k ” _{r}

 CONSTRUCT-OPT-SUBTREE(*i*, $t - 1$, t , “left”, *root*)

 CONSTRUCT-OPT-SUBTREE($t + 1$, *j*, t , “right”, *root*)



Elements of dynamic programming

Mentioned already:

- ▶ optimal substructure
- ▶ overlapping subproblems



Optimal substructure

- ▶ Show that a solution to a problem consists of making a choice, which leaves one or subproblems to solve.
- ▶ Suppose that you are given this last choice that leads to an optimal solution. [We find that students often have trouble understanding the relationship between optimal substructure and determining which choice is made in an optimal solution. One way that helps them understand optimal substructure is to imagine that “God” tells you what was the last choice made in an optimal solution.]
- ▶ Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
- ▶ Show that the solutions to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste:
 - ▷ Suppose that one of the subproblem solutions is not optimal.
 - ▷ *Cut* it out
 - ▷ *Paste* in an optimal solution.
 - ▷ Get a better solution to the original problem. Contradicts optimality of problem solution.

That was optimal substructure.

Need to ensure that you consider a wide enough range of choices and subproblems that you get them all. [“God” is too busy to tell you what that last choice really was.] Try all the choices, solve all the subproblems resulting from each choice, and pick the choice whose solution, along with subproblem solutions, is best.

How to characterize the space of subproblems?

- ▶ Keep the space as simple as possible.
- ▶ Expand it as necessary.

Examples:

Rod cutting

- ▶ Space of subproblems was rods of length $n-i$, for $1 \leq i \leq n$.
- ▶ No need to try a more general space of subproblems.

Optimal binary search trees

- ▶ Suppose we had tried to constrain space of subproblems to subtrees with keys k_1, k_2, \dots, k_j .
- ▶ An optimal BST would have root k_r , for some $1 \leq r \leq j$.
- ▶ Get subproblems k_1, \dots, k_{r-1} and k_{r+1}, \dots, k_j .
- ▶ Unless we could guarantee that $r = j$, so that subproblem with k_{r+1}, \dots, k_j is empty, then this subproblem is *not* of the form k_1, k_2, \dots, k_j .
- ▶ Thus, needed to allow the subproblems to vary at “both ends,” i.e., allow both i and j to vary.

Optimal substructure varies across problem domains:

1. *How many subproblems* are used in an optimal solution.
2. *How many choices* in determining which subproblem(s) to use.
 - ▶ Rod cutting:
 - ▷ 1 subproblem (of size $n - i$)
 - ▷ n choices
 - ▶ Longest common subsequence:
 - ▷ 1 subproblem
 - ▷ Either
 - 1 choice (if $x_i = y_j$, LCS of X_{i-1} and Y_{j-1}), or
 - 2 choices (if $x_i \neq y_j$, LCS of X_{i-1} and Y_j , and LCS of X_i and Y_{j-1})
 - ▶ Optimal binary search tree:
 - ▷ 2 subproblems (k_1, \dots, k_{r-1} and k_{r+1}, \dots, k_j)
 - ▷ $j - i + 1$ choices for k_r in k_i, \dots, k_j . Once we determine optimal solutions to subproblems, we choose from among the $j - i + 1$ candidates for k_r .

Informally, running time depends on (# of subproblems overall) × (# of choices).

- ▶ Rod cutting : $\Theta(n)$ subproblems, $\leq n$ choices for each
⇒ $O(n^2)$ running time.
- ▶ Longest common subsequence: $\Theta(mn)$ subproblems, ≤ 2 choices for each
⇒ $\Theta(mn)$ running time.
- ▶ Optimal binary search tree: $\Theta(n^2)$ subproblems, $O(n)$ choices for each
⇒ $O(n^3)$ running time.

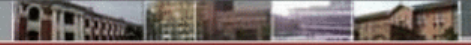


Can use the subproblem graph to get the same analysis:
count the number of edges.

- ▶ Each vertex corresponds to a subproblem.
- ▶ Choices for a subproblem are vertices that the subproblem has edges going to.
- ▶ For rod cutting, subproblem graph has n vertices and $\leq n$ edges per vertex $\Rightarrow O(n^2)$ running time.

In fact, can get an exact count of the edges: for $i = 0, 1, \dots, n$, vertex for subproblem size i has out-degree $i \Rightarrow \# \text{ of edges} = \sum_{i=0}^n i = n(n+1)/2$

- ▶ Subproblem graph for matrix-chain multiplication would have $\Theta(n^2)$ vertices, each with degree $\leq n - 1$
 $\Rightarrow O(n^3)$ running time.



Dynamic programming uses optimal substructure *bottom up*.

- ▶ *First* find optimal solutions to subproblems.
- ▶ *Then* choose which to use in optimal solution to the problem.



When we look at greedy algorithms, we'll see that they work *top down*: *first* make a choice that looks best, *then* solve the resulting subproblem.

Don't be fooled into thinking optimal substructure applies to all optimization problems. It doesn't.

Here are two problems that look similar. In both, we're given an *unweighted, directed* graph $G = (V, E)$.

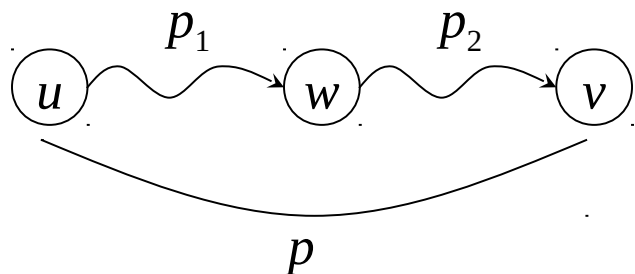
- ▶ V is a set of *vertices*.
- ▶ E is a set of *edges*.

And we ask about finding a ***path*** (sequence of connected edges) from vertex u to vertex v .

- ▶ ***Shortest path***: find path $u \rightsquigarrow v$ with fewest edges. Must be ***simple*** (no *cycles*), since removing a cycle from a path gives a path with fewer edges.
- ▶ ***Longest simple path***: find *simple* path $u \rightsquigarrow v$ with most edges. If didn't require *simple*, could repeatedly traverse a cycle to make an arbitrarily long path.



Shortest path has optimal substructure.



- ▶ Suppose p is shortest path $u \rightsquigarrow v$.
- ▶ Let w be any vertex on p .
- ▶ Let p_1 be the portion of p , $u \rightsquigarrow w$.
- ▶ Then p_1 is a shortest path $u \rightsquigarrow w$.

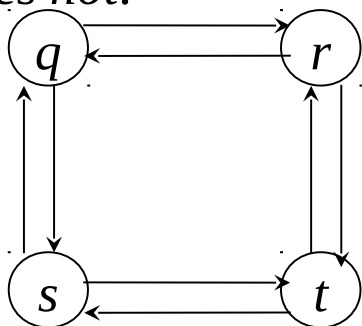
Proof Suppose there exists a shorter path p'_1 , $u \rightsquigarrow w$. Cut out p_1 , replace it with p'_1 , get path $u \xrightarrow{p'_1} w \xrightarrow{p_2} v$ with fewer edges than p . ■

Therefore, can find shortest path $u \rightsquigarrow v$ by considering all intermediate vertices w , then finding shortest paths $u \rightsquigarrow w$ and $w \rightsquigarrow v$.

Same argument applies to p_2 .

Does longest path have optimal substructure?

- It seems like it should.
- It does *not*.



Consider $q \rightarrow r \rightarrow t =$ longest path $q \rightsquigarrow t$. Are its subpaths longest paths?

- No!



- ▶ Subpath $q \rightsquigarrow r$ is $q \rightarrow r$.
- ▶ Longest simple path $q \rightsquigarrow r$ is $q \rightarrow s \rightarrow t \rightarrow r$.
- ▶ Subpath $r \rightsquigarrow t$ is $r \rightarrow t$.
- ▶ Longest simple path $r \rightsquigarrow t$ is $r \rightarrow q \rightarrow s \rightarrow t$.

Not only isn't there optimal substructure, but we can't even assemble a legal solution from solutions to subproblems.

Combine longest simple paths:

$q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t$

Not simple!

In fact, this problem is NP-complete (so it probably has no optimal substructure to find.)



What's the big difference between shortest path and longest path?

- ▶ Shortest path has *independent* subproblems.
- ▶ Solution to one subproblem does not affect solution to another subproblem of the same problem.
- ▶ Longest simple path: subproblems are *not* independent.
- ▶ Consider subproblems of longest simple paths $q \rightsquigarrow r$ and $r \rightsquigarrow t$.
- ▶ Longest simple path $q \rightsquigarrow r$ uses s and t .
- ▶ Cannot use s and t to solve longest simple path $r \rightsquigarrow t$, since if we do the path isn't simple.
- ▶ But we *have* to use t to find longest simple path $r \rightsquigarrow t$!
- ▶ Using resources (vertices) to solve one subproblem renders them unavailable to solve the other subproblem.

[For shortest paths, if we look at a shortest path $u \xrightarrow{p_1} w \xrightarrow{p_2} v$, no vertex other than w can appear in p_1 and p_2 . Otherwise, we have a cycle.]

Independent subproblems in our examples:

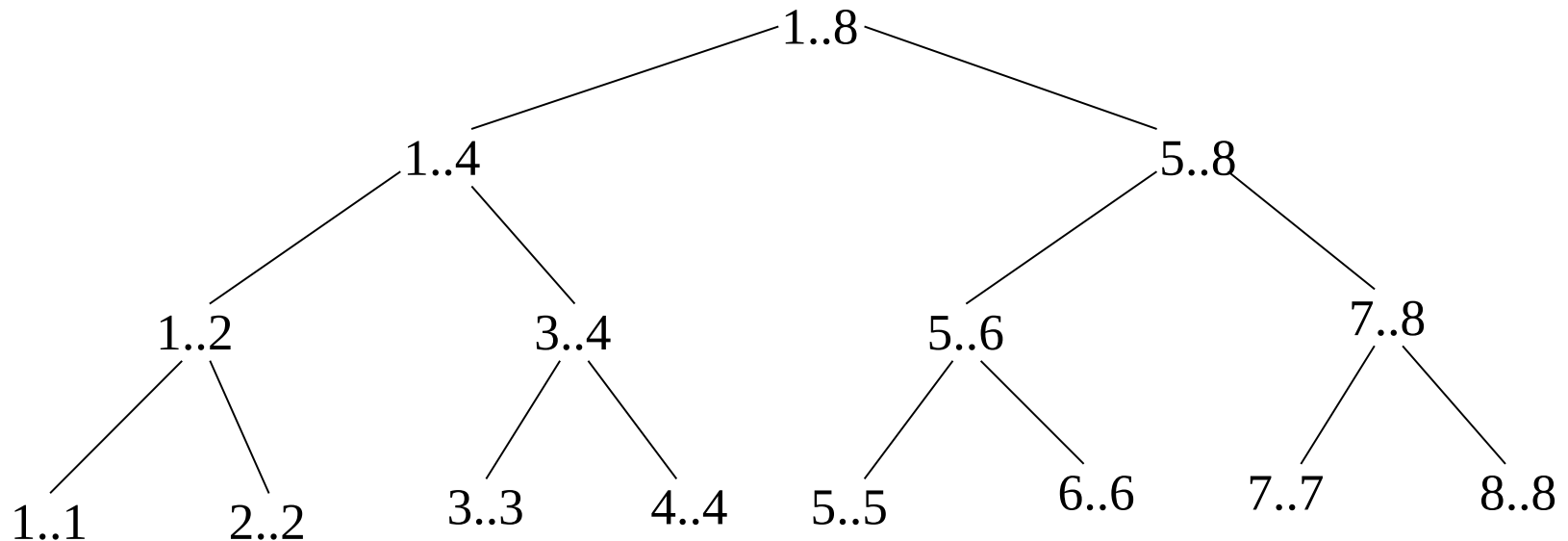
- ▶ Rod cutting and longest common subsequence
 - ▷ 1 subproblem \Rightarrow automatically independent.
- ▶ Optimal binary search tree
 - ▷ k_i, \dots, k_{r-1} and $k_{r+1}, \dots, k_j \Rightarrow$ independent.

Overlapping subproblems



These occur when a recursive algorithm revisits the same problem over and over. Good divide-and-conquer algorithms usually generate a brand new problem at each stage of recursion.

Example: merge sort





Won't go through exercise of showing repeated subproblems.

Book has a good example for matrix-chain multiplication.

Alternative approach: ***memoization***

- ▶ “Store, don't recompute.”
- ▶ Make a table indexed by subproblem.
- ▶ When solving a subproblem:
 - ▷ Lookup in table.
 - ▷ If answer is there, use it.
 - ▷ Else, compute answer, then store it.
- ▶ In dynamic programming, we go one step further. We determine in what order we'd want to access the table, and fill it in that way.