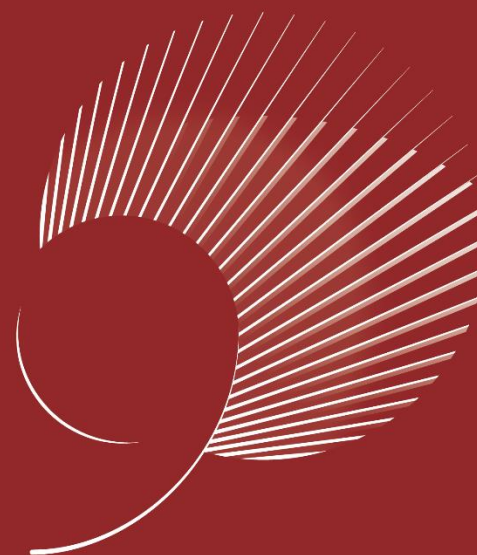


Chapter 15: Dynamic Programming (part II)

Chi-Yeh Chen

陳奇業

成功大學資訊工程學系



藏行顯光
成就共好

Achieve Securely
Prosper Mutually



國立成功大學 九十週年
90th Anniversary of NCKU

Optimal binary search trees



Optimal binary search trees

[Also new in the second edition.]

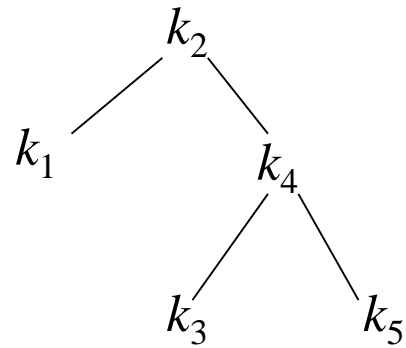
- Given sequence $K = (k_1, k_2, \dots, k_n)$ of n distinct keys, sorted ($k_1 < k_2 <$

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

[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.]

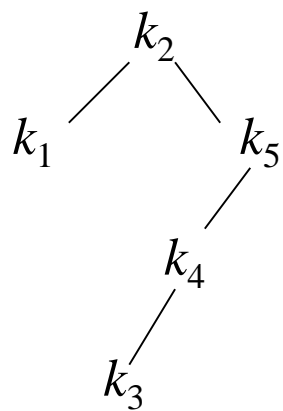
i	1	2	3	4	5
p_i	.25	.2	.05	.2	.3

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
		<hr/> 1.10

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

Observations:

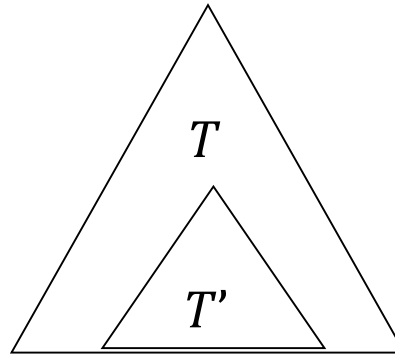
- Optimal BST might not have smallest height.
- Optimal BST might not have highest-probability key at root.

Build by exhaustive checking?

- Construct each n -node BST.
- 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 $1 \leq i \leq j \leq n$.



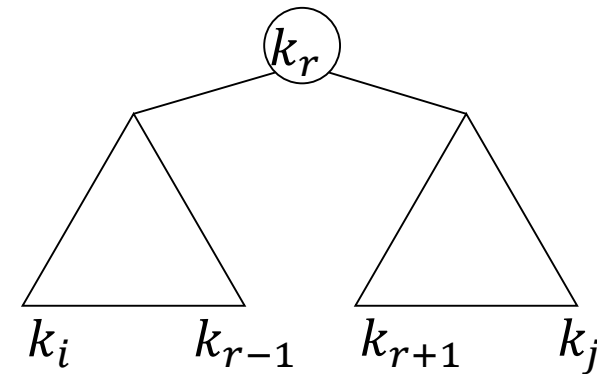
If T is an optimal BST and T contains subtree T' with keys k_i, \dots, k_j , then T' must be an optimal BST for keys k_i, \dots, k_j .

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...

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

$$\underbrace{e[1..n+1, \quad]}_{\substack{\text{can store} \\ e[n+1, n]}} \underbrace{0..n \quad]}_{\substack{\text{can store} \\ e[1, 0]}}$$

- 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 $\Theta(j - 1)$ additions.)

Instead:

- Table $w[1 \dots n + 1, 0 \dots n]$
- $w[i, i - 1] = 0$ 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  $n + 1$  do
2    $e[i, i - 1] \leftarrow 0$ 
3    $w[i, i - 1] \leftarrow 0$ 
4 for  $l \leftarrow 1$  to  $n$  do
5   for  $i \leftarrow 1$  to  $n - l + 1$  do
6      $j \leftarrow i + l - 1$ 
7      $e[i, j] \leftarrow \infty$ 
8      $w[i, j] \leftarrow w[i, j - 1] + p_j$ 
9     for  $r \leftarrow i$  to  $j$  do
10       $t \leftarrow e[i, r - 1] + e[r + 1, j] + w[i, j]$ 
11      if  $t < e[i, j]$  then
12         $e[i, j] \leftarrow t$ 
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

	e	j					
		0	1	2	3	4	5
i	1	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

<i>i</i>	<i>w</i>	<i>j</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>i</i>	<i>root</i>	<i>j</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-OPTIMAL-BST($root$)

```
1  $r \leftarrow root[1, n]$ 
2 print " $k$ " $r$  "is the root"
3 CONSTRUCT-OPT-SUBTREE(1,  $r - 1$ ,  $r$ , "left",  $root$ )
4 CONSTRUCT-OPT-SUBTREE( $r + 1$ ,  $n$ ,  $r$ , "right",  $root$ )
```

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

```
1 if  $i \leq j$  then
2      $t \leftarrow root[i, j]$ 
3     print " $k$ " $t$  "is"  $dir$  "child of  $k$ " $r$ 
4     CONSTRUCT-OPT-SUBTREE( $i, t - 1$ ,  $t$ , "left",  $root$ )
5     CONSTRUCT-OPT-SUBTREE( $t + 1, j, t$ , "right",  $root$ )
```



Longest common subsequence

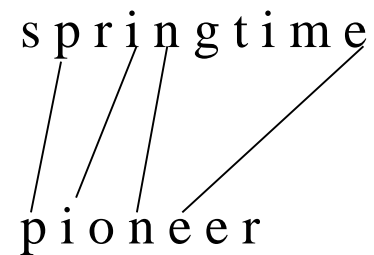


Longest common subsequence

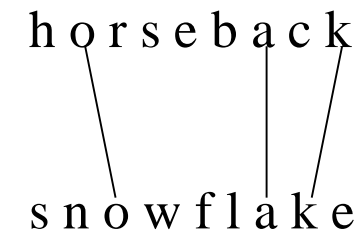
- **Problem:** Given 2 sequences, $X = \langle x_1, \dots, x_m \rangle$ and $Y = \langle y_1, \dots, y_n \rangle$, find a subsequence common to both whose length is longest. A subsequence doesn't have to be consecutive, but it has to be in order.

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

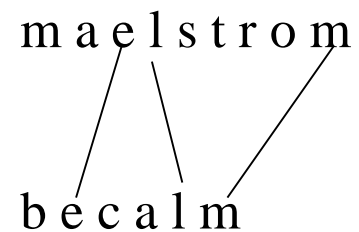
s p r i n g t i m e
p i o n e e r



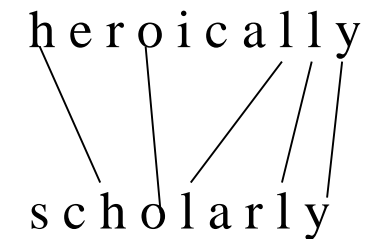
h o r s e b a c k
s n o w f l a k e



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

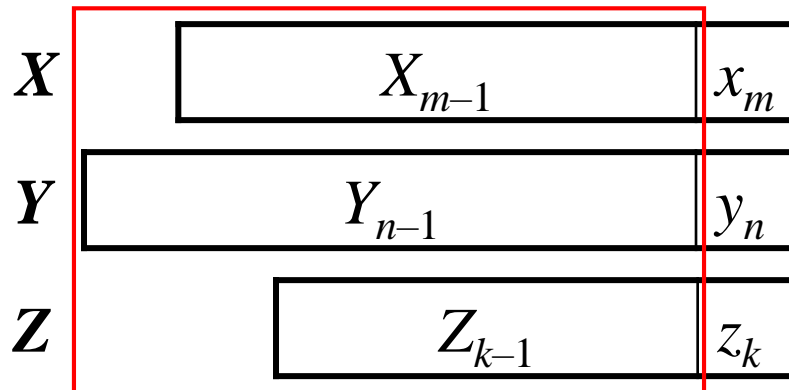
Let $Z = \langle z_1, \dots, z_k \rangle$ be any LCS of X and Y .

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} .

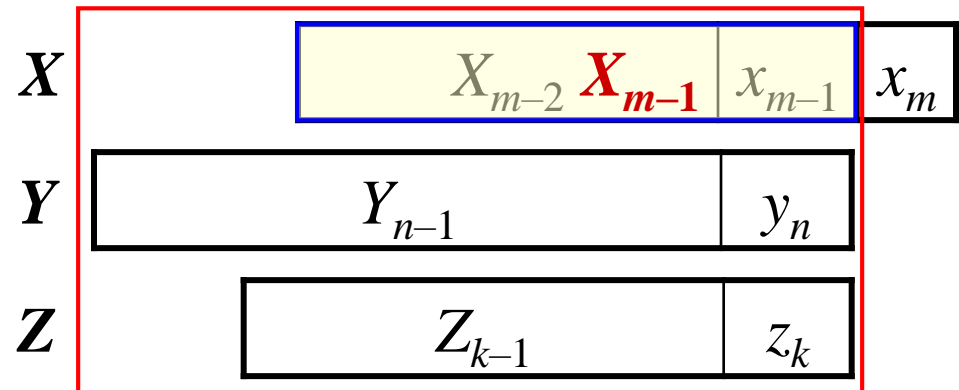
If $x_m \neq y_n$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y

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

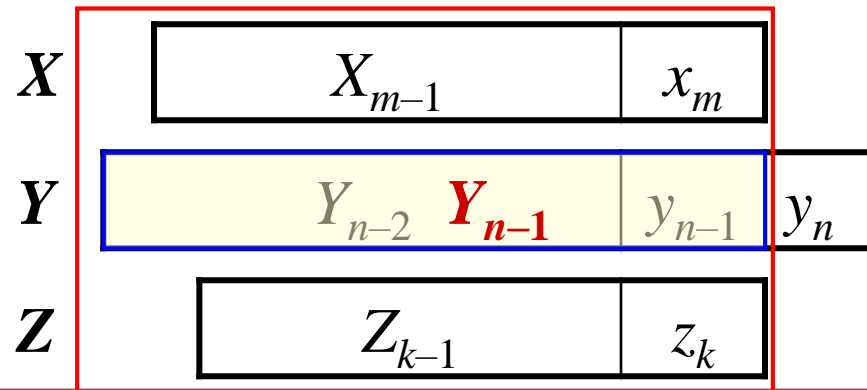
If $x_m = y_n$, then $z_k = x_m = y_n$



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



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



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 than $Z \Rightarrow$ 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 \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.

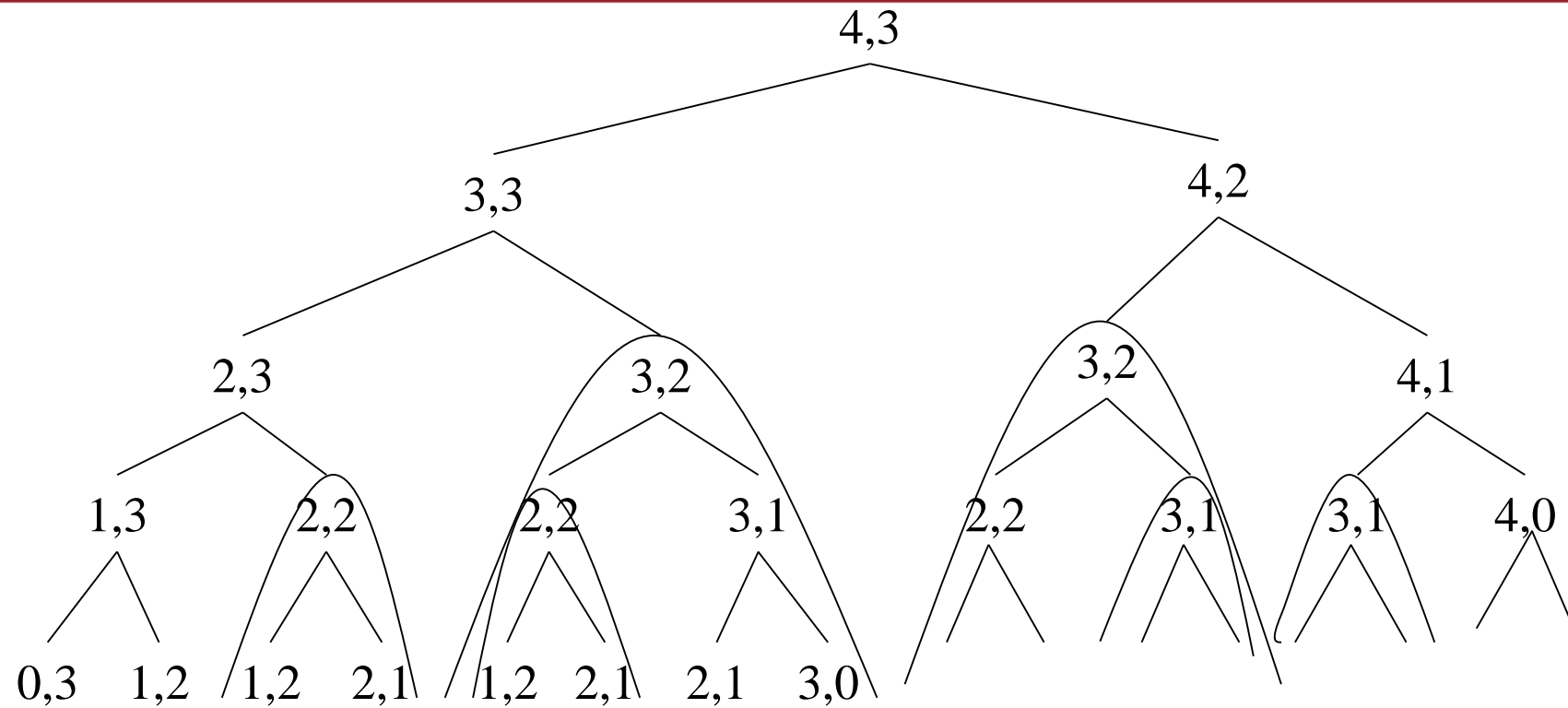
Recursive formulation

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  $m$  do
2    $c[i, 0] \leftarrow 0$ 
3 for  $j \leftarrow 1$  to  $n$  do
4    $c[0, j] \leftarrow 0$ 
5 for  $i \leftarrow 1$  to  $m$  do
6   for  $j \leftarrow 1$  to  $n$  do
7     if  $x_i = y_j$  then
8        $c[i, j] \leftarrow c[i - 1, j - 1] + 1$ 
9        $b[i, j] \leftarrow "$  ↖  $"$ 
10    else if  $c[i - 1, j] \geq c[i, j - 1]$  then
11       $c[i, j] \leftarrow c[i - 1, j]$ 
12       $b[i, j] \leftarrow "$  ↑  $"$ 
13    else
14       $c[i, j] \leftarrow c[i, j - 1]$ 
15       $b[i, j] \leftarrow "$  ←  $"$ 
16 return  $c$  and  $b$ 
```



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

```
1  if  $i = 0$  or  $j = 0$  then
2      return
3  if  $b[i, j] \leftarrow \nwarrow$  then
4      PRINT-LCS( $b, X, i - 1, j - 1$ )
5      print  $x_i$ 
6  else if  $b[i, j] \leftarrow \uparrow$  then
7      PRINT-LCS( $b, X, i - 1, j$ )
8  else
9      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
s		0	0	0	0	0	0	0	0	0	0
p		0	0	0	1	1	1	1	1	1	1
a		0	1	1	1	1	2	2	2	2	2
n		0	1	1	1	1	2	2	2	2	3
k		0	1	1	1	1	2	2	2	2	3
i		0	1	1	1	1	2	2	3	3	3
n		0	1	1	1	1	2	2	3	3	4
g		0	1	1	1	1	2	2	3	3	4
				p		a		i		n	

Time: $\Theta(mn)$



Elements of dynamic programming



Elements of dynamic programming

Mentioned already:

- optimal substructure
- overlapping subproblems

Optimal substructure

- Recall that a problem exhibits **optimal substructure** if an optimal solution to the problem contains within its optimal solutions to subproblems.
- We observed that the optimal way of cutting up a rod of length n involves optimally cutting up the two pieces resulting from the first cut.
- We observed that an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ that splits the product between A_k and A_{k+1} contains within its optimal solutions to the problems of parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$.



Optimal substructure

You will find yourself following a common pattern in discovering optimal substructure:

1. You show that a solution to the problem consists of making a choice, such as choosing an initial cut in a rod.
2. You suppose that for a given problem, you are given the choice that leads to an optimal solution.
3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
4. You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by using a “**cut-and-paste**” technique.

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_i, \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 $O(n^2)$ 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 : # 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 $\leq n-1$
 $O(n^3)$ 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.

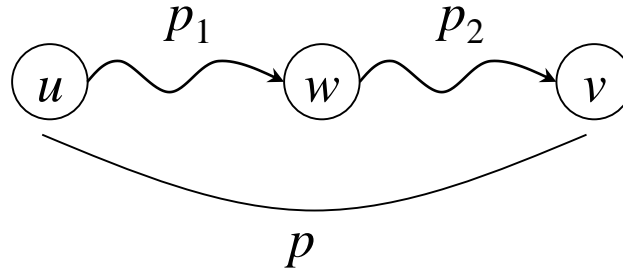
Here are two problems that look similar. In both, we're given an *unweighted, directed* graph $G = (V, E)$.

- 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 $u \rightsquigarrow w$.

Claim that if p is an optimal path from u to v , then p_1 must be a shortest path from u to w .

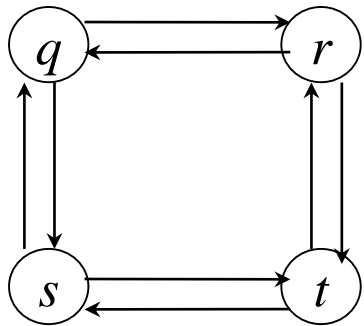
Proof

Suppose there exists a shorter path p'_1 , $u \rightsquigarrow w$. Cut out p_1 , replace it with p'_1 , get path $u \rightsquigarrow w \rightsquigarrow v$ 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 $w \rightsquigarrow v$, which contradicts the assumption that p is a shortest path.

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.]

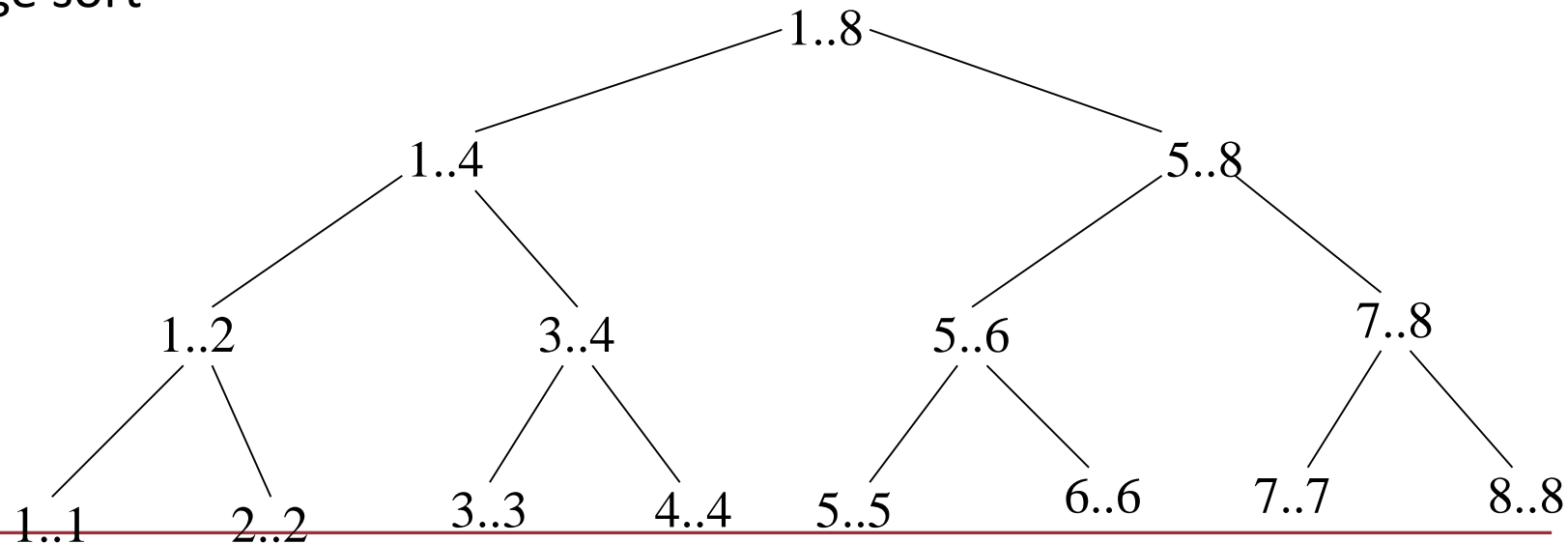
Independent subproblems in our examples:

- 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

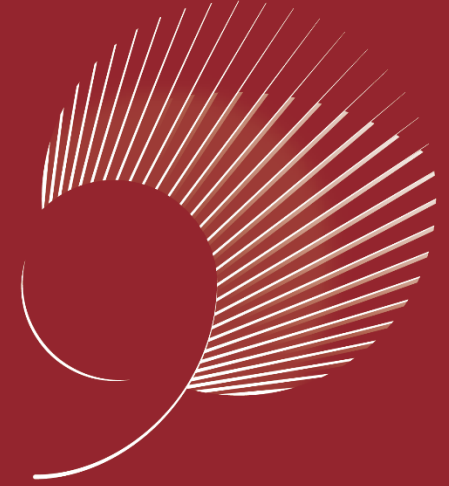
These occur when a recursive algorithm revisits the same problem over and over. Good divide-and-conquer algorithms usually generate a new problem at each stage of recursion.

Example: merge sort



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.



藏行顯光 成就共好

Achieve Securely
Prosper Mutually



國立成功大學 九十週年
90th Anniversary of NCKU



國立成功大學
National Cheng Kung University