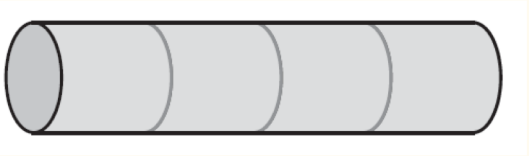
UMSL

Dynamic Programming

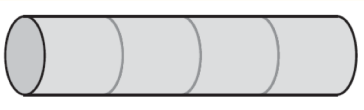
The rod-cutting problem



- You are in a company that buys long steel rods and cuts them into shorter rods, and then sells them. *Cutting is free!*
- They hired you to the find the best way to cut up the rods.
- The company charges p_i dollars for a length of rod i. Rods are always integral in length.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

The problem: Given a rod of length n inches and a table of prices p_i (i is in the range 1, 2, ..., n), determine the maximum revenue r_n obtained by cutting the rod and selling the pieces. If no cutting gives the best price, we don't cut at all.



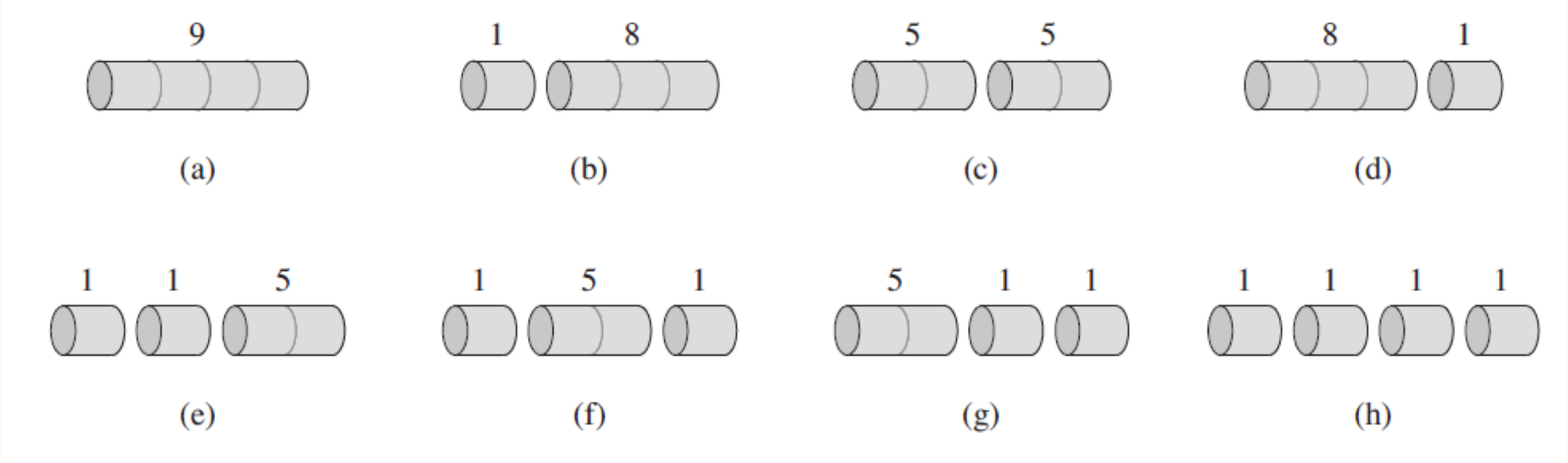
The rod-cutting problem

You are given a rod of 4 inches and a price table.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Should you cut the rod at all?

The rod-cutting problem



8 possible ways of cutting a rod of length 4

We can cut up a rod of length n in 2^{n-1} different ways.

The rod-cutting problem

Length(n)	Price (p_n)	Cut / Don't Cut	Revenue (r_n)
1	1	N	1
2	5	N	5
3	8	N	8
4	9	2 + 2	10
5	10		
6	17		
7	17		
8	20		
9	24		

$$
\begin{aligned}
r_1 &= 1 \quad \text{from solution } 1 = 1 \quad \text{(no cuts)}, \\
r_2 &= 5 \quad \text{from solution } 2 = 2 \quad \text{(no cuts)}, \\
r_3 &= 8 \quad \text{from solution } 3 = 3 \quad \text{(no cuts)}, \\
r_4 &= 10 \quad \text{from solution } 4 = 2 + 2, \\
r_5 &= 13 \quad \text{from solution } 5 = 2 + 3, \\
r_6 &= 17 \quad \text{from solution } 6 = 6 \quad \text{(no cuts)}, \\
r_7 &= 18 \quad \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3 \\
r_8 &= 22 \quad \text{from solution } 8 = 2 + 6, \\
r_9 &= 25 \quad \text{from solution } 9 = 3 + 6, \\
r_{10} &= 30 \quad \text{from solution } 10 = 10 \quad \text{(no cuts)}.
\end{aligned}
$$

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Defining the rod-cutting problem formally

The optimal revenue from a rod of length n, r_n

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

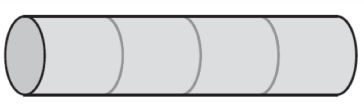
p_n => making no cut at all, $r_1 + r_{n-1}$ => a cut so that we have rods of length 1 and n-1

In other words,

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

p_i => price of rod length i, r_{n-1} => revenue from rod of length n-1

Is this a recursive approach?



Recursive (top-down) implementation

```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Python Implementation

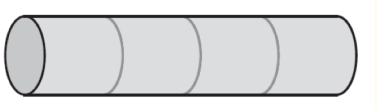
```
Recursive solution to the rod-cutting problem (top-down approach)
"""

#################################################################
p = (-1000, 1,5,8,9,10,17,17,20,24,30)

#################################################################
def CUT_ROD(p, n):
      if n == 0:
            return 0
      q = -1000000
      for i in range(1, n+1):
            q = max(q, p[i] + CUT_ROD(p, n-i))
      return q

#################################################################
print ("")
print ("L = " + str(len(p)))

#################################################################
print ("")
for i in range(1, len(p)):
      print (str(i) + " (p="+str(p[i])+") => " + str(CUT_ROD(p, i)))
```



Python Implementation

What is one limitation of the solution?

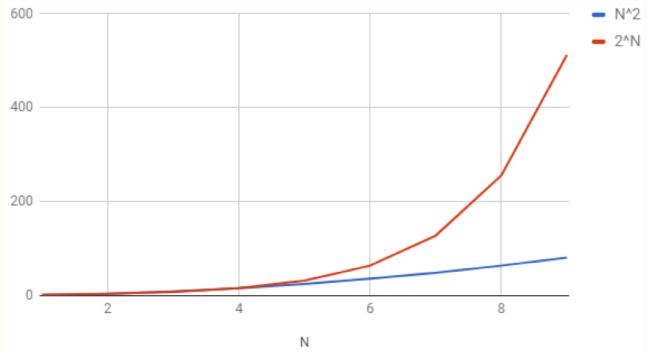
Why is the recursive program so inefficient (slow)?

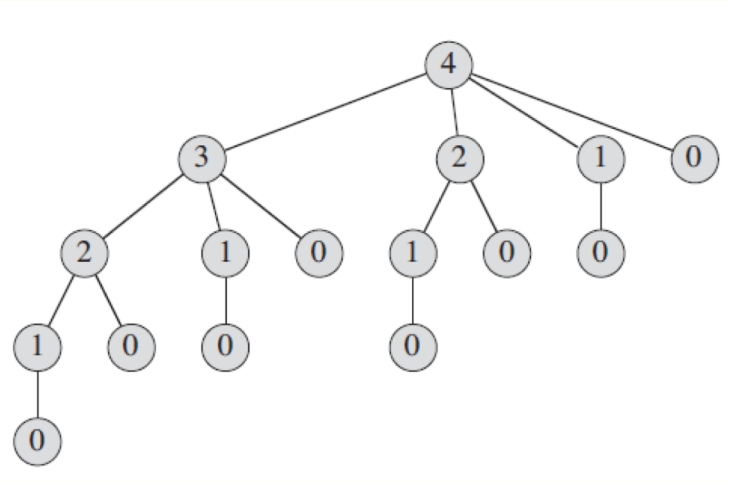
Analyze the recursive solution using recursion tree

The rod-cut algorithm calls itself recursively over and over again with the same parameter values; it solves the same problems repeatedly.

The running time of the algorithm is exponential in n.

$T(n) = 2^n$





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

Dynamic programming - intuition

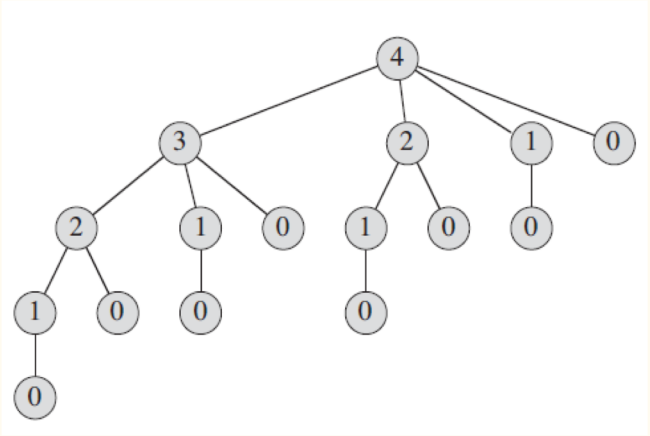
We apply dynamic programming to **optimization problems**, which typically have many possible solutions.

Each solution has a value and we wish to find a solution with an **optimal** (**maximum** or **minimum**) value.

Such a solution is called ***an* optimal solution** and not *the* optimal solution.

Programming refers to a **tabular method**, not writing computer code

DP is applicable when **subproblems have subproblems**



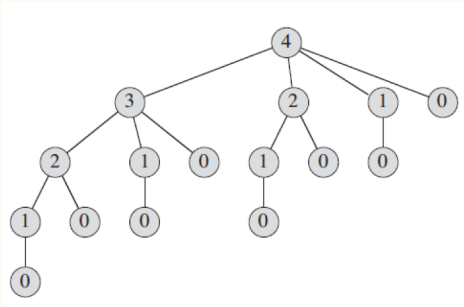
Dynamic programming for optimal rod-cutting

We arrange for each subproblem to be solved only **once**, saving its solution.

During the computation, if we need the solution to a subproblem again, we simply look it up rather than recomputing it.

Two dynamic programming approaches:

(a) Top-down dynamic programming approach with *memoization*
(b) Bottom-up dynamic programming approach



NOT memorized (but we could say so ☺)

Memoized rod-cutting algorithm

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

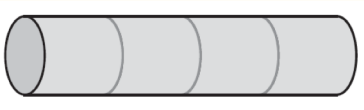
Initializes the new auxiliary array with -inf (unknown)

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

Recursive vs Dynamic programming (memoized)

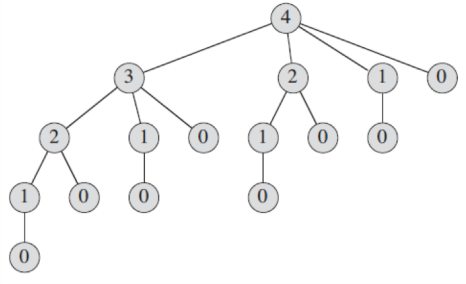
```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = −∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n − i))
6  return q
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2       return r[n]
3  if n == 0
4       q = 0
5  else q = −∞
6       for i = 1 to n
7            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```



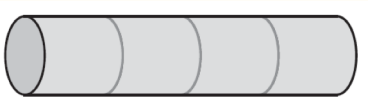
Bottom-up rod-cutting algorithm

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```



A subproblem i is smaller than subproblem j,

So solve subproblems of sizes $j = 0, 1, 2, ..., n$, in that order.



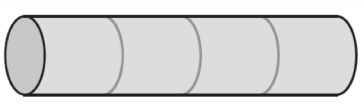
Memoization vs Bottom-up

The bottom-up and top-down versions have same asymptotic running time - $\Theta(n^2)$ because of the double for loops.

The bottom up approach usually outperforms the top-down approach by a small constant factor.

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = −∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7      r[j] = q
8  return r[n]
```



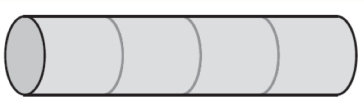
Returning actual solution

For each rod of size j, we would like to compute not only the maximum revenue r_j, but also s_j, the optimal size of the first piece to cut off.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Returning actual solution



```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n − s[n]
```

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = −∞
 5      for i = 1 to j
 6          if q < p[i] + r[j − i]
 7              q = p[i] + r[j − i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

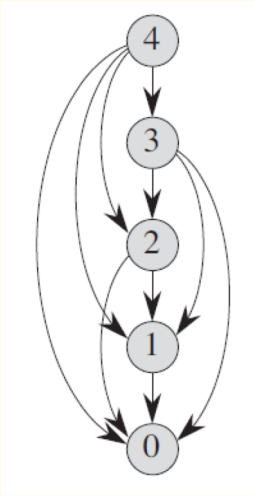
Subproblem graphs

When applying dynamic programming to a problem we should understand the set of subproblems involved.

How do the subproblems depend on each other?

A subproblem graph for the problem embodies this information.

It is like a ‘reduced’ or ‘collapsed’ version of the recursion tree.



- subproblem graph for n = 4 for the rod-cutting problem
- the vertex labels give the sizes of the corresponding subproblems.
- a directed edge x -> y indicates that we need a solution to subproblem y when solving subproblem x

When is dynamic programming applicable and how to apply it?

Elements of dynamic programming

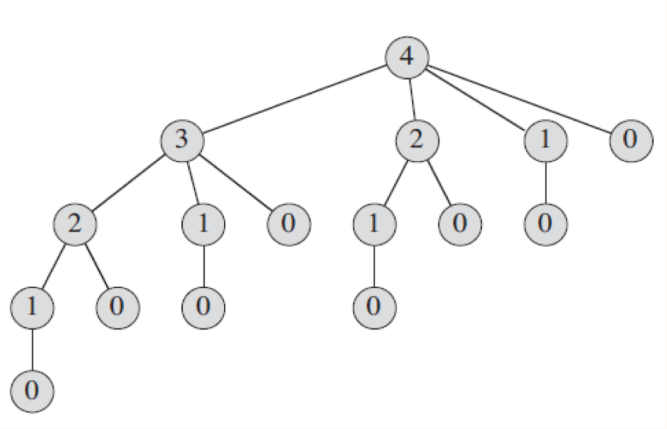
Two key ingredients that an optimization problem must have in order for dynamic programming to apply:

a) Optimal substructure

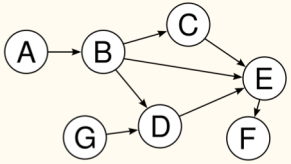
A problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems.

b) Overlapping subproblems

The space of subproblems must be ‘small’ in the sense that the recursive algorithm for the problem solves the same subproblems over and over (*not generating new subproblems*).



When is dynamic programming not applicable?



Given is a directed graph G = (V, E) and vertices u, v ∈ V.

(a) Unweighted shortest path: Find a path from u to v consisting of the fewest edges. Such a path must be simple (no cycles).

Any path *p* from u to v must contain w (say). Then we can decompose p into *p1* and *p2* so that *p1* is the path from u to w and *p2* from w to v.

Can we argue that *p1* is the shortest path from u to w, and *p2* from w to v?

(a) Unweighted longest path: Find a simple path from u to v consisting of the most edges.

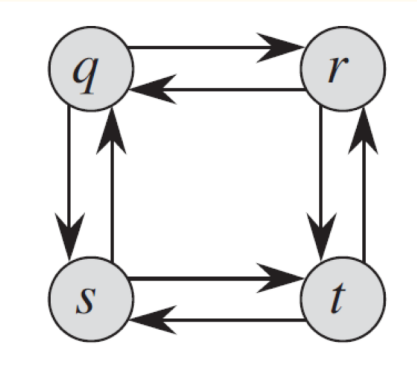
Unweighted longest path problem

Say we have a path *p* as the longest path from u to v. We have intermediate vertex w that lies in the path such that path *p* is composed of *p1* and *p2*. Does this imply that *p1* is the longest path from u to w and *p2* is the longest path from w to v?

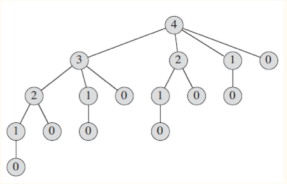
Consider path q → r → t $\Rightarrow$ the longest path from q to t.

Is q → r the longest path from q to r ?

Is r → t the longest path from r to t ?



Steps for developing a dynamic programming algorithm



1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution. $r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution. (*where to make a cut in the rod-cutting problem*)

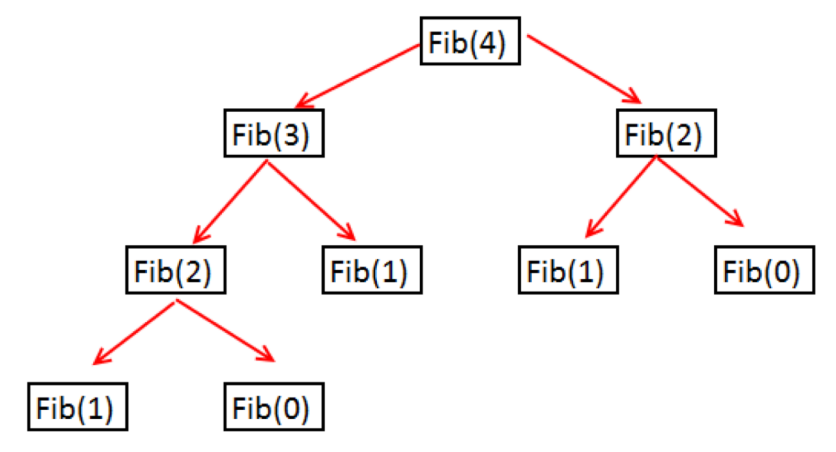
```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] and s[0..n] be new arrays
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          if q < p[i] + r[j - i]
7              q = p[i] + r[j - i]
8              s[j] = i
9      r[j] = q
10 return r and s
```

Example applications of dynamic programming

Calculating n^{th} Fibonacci number

$$F_1 = F_2 = 1; \quad F_n = F_{n-1} + F_{n-2}$$



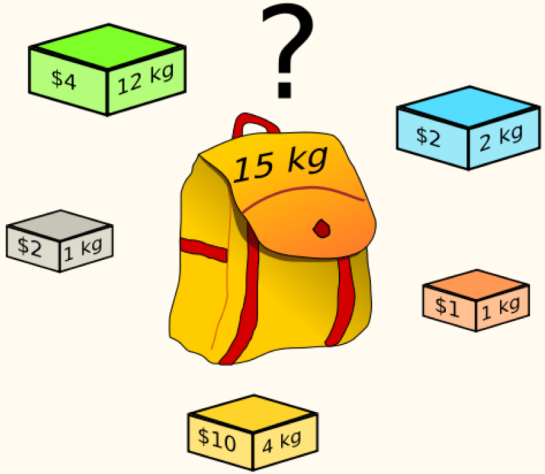
The 0-1 knapsack problem

A thief robbing a store find n items.

The i^{th} item is worth v_i dollars and weighs w_i pounds (v_i and w_i are integers).

The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack (W is an integer).

Which items should he take?

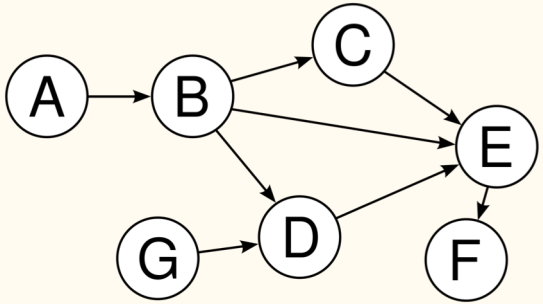


Shortest path in a DAG

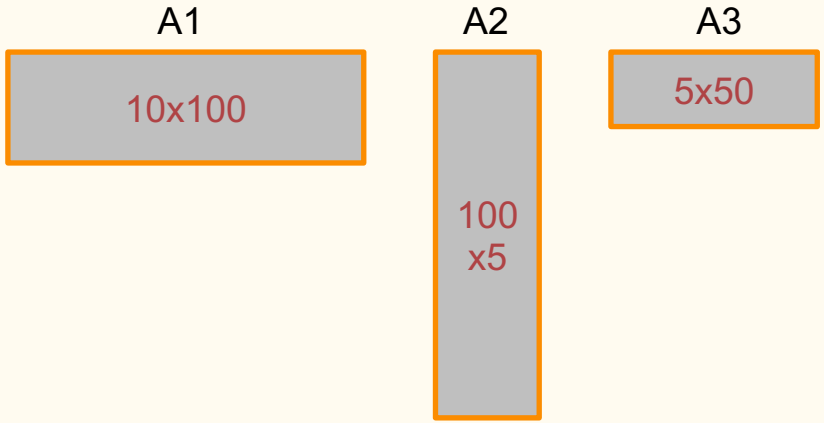
Find shortest path from vertex s to vertex v in a directed acyclic graph (DAG)

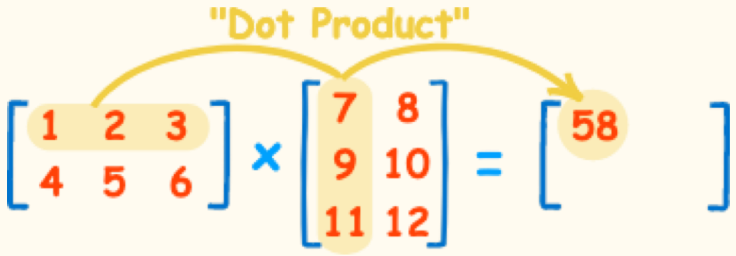
Recursive formulation of the single source shortest path problem:

$$\delta(s, v) = \texttt{min}\{\delta(s, u) + w(u, v) | (u, v) \in E\}$$



Matrix chain multiplication

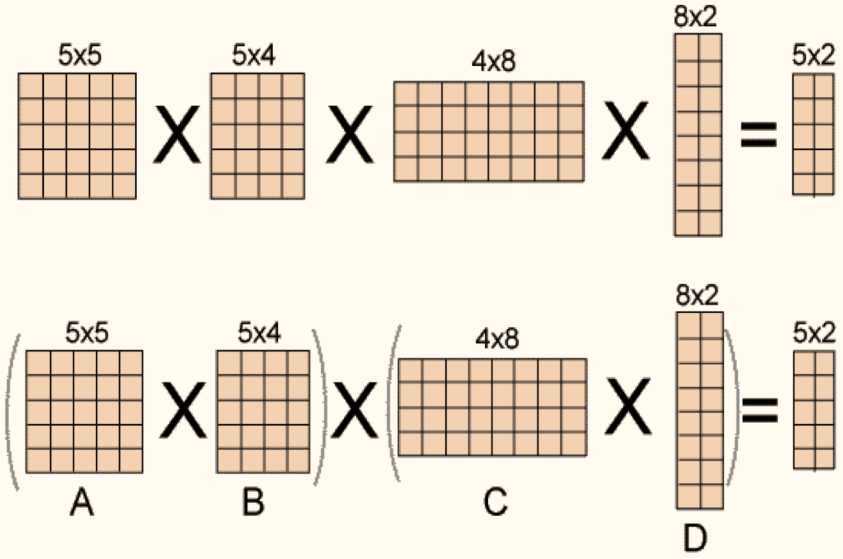




(A1 x A2) x A3 = (10 x 100 x 5) + 10 x 5 x 50
= 7,500 scalar multiplications

A1 x (A2 x A3) = (100 x 5 x 50) + 10 x 100 x 50
= 75,000 scalar multiplications

Matrix-chain multiplication



$$(A_1(A_2(A_3A_4)))$$
$$(A_1((A_2A_3)A_4))$$
$$((A_1A_2)(A_3A_4))$$
$$((A_1(A_2A_3))A_4)$$
$$(((A_1A_2)A_3)A_4)$$

Finding DNA similarity

A strand of DNA consists of string of molecules called *bases* (adenine, guanine, cytosine, and thymine).

DNA can be expressed as a string string of A, C, G, and T.

Compare the DNA to two (or more) different DNAs.

DNA similarity => how similar the two organisms are.

$S_1 = \texttt{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$

$S_2 = \texttt{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$

Measuring similarity: Find a third strand S_3 such that the bases in S_3 appear in both strands - S_1 and S_2; these bases must appear in the same order, but not necessarily consecutively.

GTCGTCGGAAGCCGGCCGAA

What is a subsequence?

Given a sequence $X = \langle x_1, x_2, ..., x_m \rangle$

A sequence $Z = \langle z_1, z_2, ..., z_k \rangle$ is a **subsequence** of X if there exists a strictly increasing sequence $i_1, i_2, ..., i_k$ of indices of X such that for all $j = 1, 2, ..., k$, we have $x_{ij} = z_j$.

Example: $X = \langle A, B, C, B, D, A, B \rangle$ then $Z = \langle B, C, D, B \rangle$ is a subsequence of X with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$.

What is a longest common subsequence?

Z is a **common subsequence** of X and Y if it is subsequence of both X and Y.

Example: $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$ then the sequence $\langle B, C, A \rangle$ is a common subsequence of X and Y.

$\langle B, C, A \rangle$ is not the longest common subsequence.

$\langle B, C, A, B \rangle$ and $\langle B, D, A, B \rangle$ are the two **longest common subsequences.**

The LCS problem: Given two sequences $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_m \rangle$, find the maximum length common subsequence of X and Y.

The brute-force approach

(1) Enumerate all subsequences of X and Y

- takes 2^m time, because X has that many subsequences

(1) Then for each subsequence in X check if is matches each subsequence in Y

(2) Find the longest matching subsequence

"Requires exponential time"

$S_1 = \texttt{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$

$S_2 = \texttt{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$

A recursive solution

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

$S_1 = \texttt{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$

$S_2 = \texttt{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$

Optimal-substructure property of LCS problem

Let $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, ..., z_k \rangle$ be any LCS of X and Y.

Then:

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

   $X = \langle A, B, C, B, D, A, A \rangle$ $Y = \langle B, D, C, A, B, A \rangle$ $Z = \langle B, C, B, A \rangle$

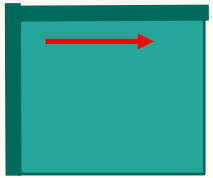
2) If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.

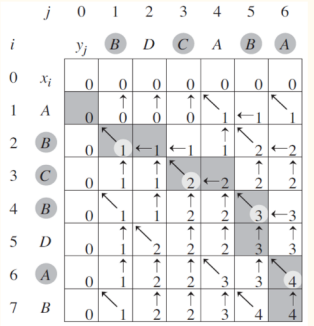
   $X = \langle A, B, C, B, D, A \rangle$ $Y = \langle B, D, C, A, B \rangle$ $Z = \langle B, C, B \rangle$

3) If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

   $X = \langle A, B, C, B, D, A, B \rangle$ $Y = \langle B, D, C, A, B, A \rangle$ $Z = \langle B, D, A, B \rangle$

LCS using dynamic programming





	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Inputs: $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$

Returns: b and c tables; c[m][n] contains the length of LCS of length X and Y

- Store c[i,j] values in a table c[0..m][0..n]
  - Compute entries in **row-major** order (i.e. fill in the first row of c from left to right, then second row, and so on.)
- Also maintain table b[0..m][0..n] to help construct an optimal solution
  - b[i][j] points to the table entry corresponding to optimal subproblem chosen when computing c[i][j]

Takes: $\Theta(m * n)$ time

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

Constructing LCS

Begin at b[m][n] and trace through the table by following the arrows.

Whenever we encounter a “ ” in the entry b[i,j], it implies that $x_i = y_i$ is an element of the LCS.

We encounter the elements of LCS in reverse order.

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2       return
3  if b[i, j] == "↖"
4       PRINT-LCS(b, X, i − 1, j − 1)
5       print x_i
6  elseif b[i, j] == "↑"
7       PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

LCS of “HUMAN” and “CHIMPANZEE”

http://wordaligned.org/articles/longest-common-subsequence