Dynamic Programming History

- D*ynamic Programming* is a general algorithm design technique for solving (mostly optimization) problems **defined by recurrences with overlapping subproblems.**
- Bellman. Pioneered the systematic study of dynamic programming in the 1950s.
- Etymology.
  - Dynamic programming = planning over time.
  - Secretary of Defense was hostile to mathematical research.
  - Bellman sought an impressive name to avoid confrontation.
    - "it's impossible to use dynamic in a pejorative sense"
    - "something not even a Congressman could object to"

Dynamic Programming Applications

- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, systems, ….
- Some famous dynamic programming algorithms.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.
  - Smith-Waterman for sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger for parsing context free grammars.

Design paradigms

Greedy, Divide and Conquer, and Dynamic Programming all use solutions to subproblems to solve a larger problems

- Greedy. Build up a solution incrementally, myopically optimizing some local criterion.
- Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- Dynamic programming. Break up a problem into a series of overlapping/repeating sub-problems, and build up solutions to larger and larger sub-problems.

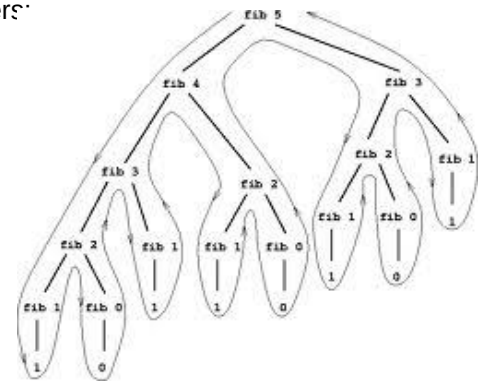
Design and Implementation of a Dynamic Programming Algorithm

Takes practice, good news formulaic in approach, key idea is optimal substructure of problem

- Characterize the solution to a problem in terms of the solution to smaller problems. Structure of an optimal solution.
- Define the value of the optimal solution in terms of the value of the optimal solutions of some smaller instances
- Compute the value of the optimal solution either in a bottom up way by recording solutions in a table or recursively using memoization
- Construct the optimal solution of the problem usually using information from the table

Fibonacci Numbers

- Definition of Fibonacci numbers:
  - f(0) = 0
  - f(1) = 1
  - f(n) = f(n-1) + f(n-2)



- Problem: Subproblems are recomputed multiple times.
- How can this be avoided??

Maximum weight independent set in path graphs

- Given a graph G = (V,E) and independent set is a subset of V, say V', such that no two vertices in V' are adjacent to each other.
- The path graph P is a tree with two nodes of vertex degree 1, and the other n-2 nodes of vertex degree 2. A path graph is therefore a graph that can be drawn so that all of its vertices and edges lie on a single straight line.

Coin Row Problem (An example of a maximum weight independent set in a path graph)

There is a row of *n* coins whose values are some positive integers c_1, c_2,.., c_n, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up

1. Characterize a solution in terms of smaller instances of the problem. For a given instance, what choice makes sense?
2. Write down a recurrence relation for the optimum value
3. Determine a table that will contain the solutions and fill it by solving the smallest problems first (bottom-up)
4. Determine the solution by tracing back in the table determining the choices made at each step

Approaches

- Brute Force – try all subsets of vertices – exponential time
- Greedy – e.g. at each step take the highest weight that does not violate the constraint. 1 – 4 – 5 – 4
- Divide and Conquer – crossing boundary – violating constraints can be done in quadratic time

Reasoning about Optimal substructure

- Thought experiments –
  - Suppose you already knew how to solve the problems of *smaller sizes. Is there a way to search through these smaller solutions to find candidates as partial solution to larger problem. AND/OR*
  - *Think about the problem of size n. Does the solution contain a solution to smaller subproblem.*
  -
- Looking at the last coin, c_n in the row (path graph) either it is or is not in the solution for c_1 , c_2 ,…., c_n , call this S_n !
  - Case 1 c_n is not in S_n, then the optimal solution must be the optimal solution for n-1 coins - S_{n-1} . Why?
  - Case 2 c_n is in S_n, then the optimal solution must be the optimal solution for n-2 coins - S_{n-2} + c_n Why?

Example: 5, 1, 2, 10, 6, 3

1. To pick or not pick up a coin. Is there knowledge of a subproblem that enables you to make that decision?

2. F(n) = the maximum amount you can pick up from n coins
   F(n) = max{c_n + F(n-2), F(n-1)} for $n > 1$ F(0)=0 and F(1)=c_1

3.

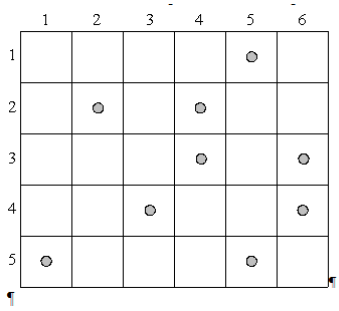
index	0	1	2	3	4	5	6
coins		5	1	2	10	6	3
F(n)	0	5	5	7	15	15	18

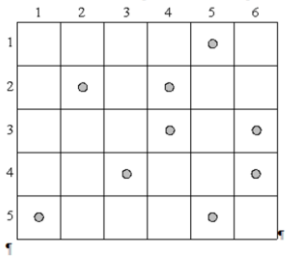
4.

Pick the last coin (3)?	yes – since 18 is not equal to 15 but is equal to 15+3
Pick the 4th coin?	yes – since 15 = 10+5
Pick the 2nd coin?	no – since 5=5 (the previous entry)
Pick the 1st coin?	yes – since 5 is not equal to 0

Coin Collecting in a grid by a robot

Coins are placed in cells of an *n*×*m* board. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the right or one cell down from its current location.





1. In this problem the "choice" is a bit harder to imagine. Think of the robot as bring in a given location. The question is which previous possible location did it come from – above or left
2. F(*i,j*) = max{F(*i-1,j*), F(*i, j-1*)} + $c_{i,j}$ for *i,j* > 0
   F(i,0)=0 and F(0,j))=0
3.

0¤	0¤	0¤	0¤	0¤	0¤	0¤
0¤	0¤	0¤	0¤	0¤	1¤	1¤
0¤	0¤	1¤	1¤	2¤	2¤	2¤
0¤	0¤	1¤	1¤	3¤	3¤	4¤
0¤	0¤	1¤	1¤	3¤	3¤	5¤
0¤	0¤	1¤	2¤	3¤	4¤	5¤

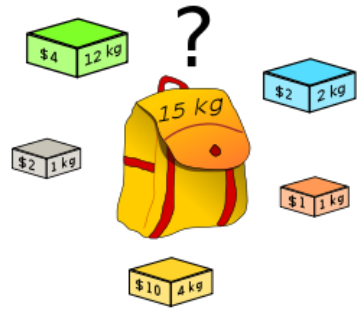
4. 5 in the lower right hand corner come from above
   Etc.

0-1 Knapsack

Given n items
integer weights: w_1 w_2 … w_n
integer values: v_1 v_2 … v_n
knapsack capacity: W

Find the most valuable subset of the items that fit into the knapsack.

Exhaustive enumeration: $\Theta(2^n)$

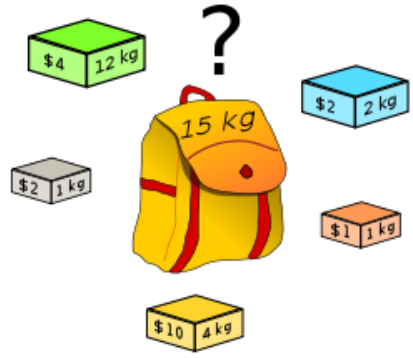


(2, $12) (1, $10) (3, $20) (2, $15)
and the overall capacity W = 5

Knapsack with repeats

Given *n* items
integer weights:
w_1 w_2 …
$w_n$values:
v_1 v_2 … v_n
knapsack capacity
W

Find the most valuable group of the items that fit into the knapsack. However now there are as many copies of each item as you want



(6, $30) (3, $14) (4, $16) (2, $9)
Capacity W = 10

Memoization or Memory functions

- Idea: only fill in entries of table if necessary.
- Use the recursive function but before executing a recursive call check to see if the desired value already exists in the table
  - Thus need to initialize the table in such a way that you know if the value has already been computed. Note this usually means that the order of growth of the initialization is still the size of the table but it may save a significant number of basic operations

Ques: When is this likely to be helpful?

Longest Increasing Subsequence Problem

Definition: Z is a subsequence of $X = x_1, ..., x_n$ if there exists a strictly increasing sequence of **indices** $i_1, ..., i_k$ so that $z_k = x_{i_k}$

For example: (A, C, F) is a subsequence of (A, B, C, D, E, F)
with $i_1 = 1$, . $i_2 = 3..$, $i_3 = 6$

Longest increasing subsequence problem: Given a sequence of ordered elements (usually numbers) , X , find the longest subsequence of X, call it Z, such that $z_k < z_{k+1}$ $\forall$ k

For example: X = 1, 5, 12, 10, 3, 11, 4, 3 has longest increasing subsequence of length 4 -- 1, 5, 10, 11

Longest Increasing Subsequence Problem

What is the objective function?

Let G(k) : the length of the longest increasing subsequence of a given sequence X of length k – is the obvious choice but ….

What is the recurrence relation? (this is where you need to be careful)

To have the recurrence work well, it should be modified to be

F(k) : the length of the longest increasing subsequence in X that includes x_k !? This allows us to write the following recurrence relation

F(k) = $\max_{i<k \text{ and } x_i<x_k}\{F(i)\}$+ 1.

What is the table?

A simple one dimensional array.

How do we Trace back?

by looking for an earlier entry i with $x_i < x_k$ and the correct length of a subsequence ending at that entry

Example: Longest Increasing Subsequence
X = 1, 5, 12, 10, 3, 11, 4, 3

Example: Longest Increasing Subsequence
X = 1, 5, 12, 10, 3, 11, 4, 3

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
X_i	1	5	12	10	3	11	4	3
F(i)	1							
F(i)	**1**	**2**	**3**	**3**				
F(i)	**1**	**2**	**3**	**3**	**2**	**4**	**3**	**2**

- Initialize with 1 in F(**1**)
- A 4 goes in F(4) since 10 is greater than any of the earlier entries in X_i
- A 5 goes into F(6) since 11 greater than all the preceding entries of X_i and the largest entry of F(i) is in F(4) , 4+1 gives the 5

Longest Common Subsequence Problem

Given two sequences:

X: x_1 x_2 … x_n Y: y_1 y_2 … y_m

A common subsequence of the two sequences can be written as

Z: z_1 z_2 … z_k *where Z is a subsequence of both X and Y.*

Longest common subsequence problem: Given two (or more) sequences X and Y, find a maximum length common subsequence of X and Y

Solving the Longest Common Subsequence

Always a good idea to write down a small example

- What are you trying to optimize? Describe the objective function in words?
- What is the argument that characterizes the size of problem?
- Developing the recurrence relation?

```
X = AGCAGT     Y=GACA     a longest common subsequence is ACA
What would be a subproblem?
```

- Table structure?
- How fill in?
- How trace back?

Matrix Chain Multiplication

- Suppose we want to multiply four matrices AxBxCxD of dimensions: 5x20, 20x1, 1x10, 10x100 and we would like to minimize the number of multiplications
- Two facts are important when considering the problem
  - When multiplying two matrices of dimensions n x m and m x p the number of multiplications is the product of the dimensions each appearing once – namely nmp.
  - Matrix multiplication is associative – that is we can multiply the matrices grouping them in any way we please and still get the same answer.
- Thus in this case there are 5 possible groupings A(B(CD), A((BC)D), (AB)(CD), ((AB)C)D, (A(BC))D. Note that the number of grouping grows at least as fast as 2^n.

Matrix Chain Multiplication: Example

- AxBxCxD of dimensions: 50x20, 20x1, 1x10, 10x100.
  Let ops = multiplications in what follows.
- A(B(CD)
  - CD requires 1,000 ops and results in a 1x100 matrix,
  - B(CD) this requires an additional 2,000 ops,
  - finally computing A(B(CD) requires 100,000 ops (50x20x100).
  - Thus the entire chain of multiplications requires 103,000 ops
- Carrying out similar calculations you find that:
  - A(B(CD) 103,000
  - A((BC)D) 120,200
  - (A(BC))D 60,200
  - (AB)(CD) 7,000
  - ((AB)C)D 51,500

Matrix Chain Multiplication Optimal Substructure: Finding the recurrence relation

- What might be a reasonable subproblem?
- Notation:
  - To multiply the matrices A_1, A_2, …A_n where the dimensions of A_i are d_{i-1} x d_i . Thus multiplying
    - » A_1 x A_2 requires d_0 x d_1 x d_2 operations
    - » A_k x A_{k+1} requires d_{k-1} x d_k x d_{k+1} operations
  - Let M(i,j) = the least number of multiplications need to multiply matrices from i to j.

- M(i, j) = **min** {M(i,k) + M(k+1,j) + d_{i-1} d_k d_j } overall all the indices k that split the problem.

- The base case is M(i, i) = 0 for 1 ≤ i ≤ n.

Matrix Chain Multiplication: example

Computing M(i,j)

A_1 be 30 x 35
A_2 be 35 x 15
A_3 be 15 x 5
A_4 be 5 x 10
A_5 be 10 x 20
A_6 be 20 x 25

j ¶ i¤	1¤	2¤	3¤	4¤	5¤	6¤
1¤	0¤	15,750¤	7875¤	9375¤	11,875¤	15,125¤
2¤	¤	0¤	2,625¤	4375¤	7125¤	10500¤
3¤	¤	¤	0¤	750¤	2500¤	5375¤
4¤	¤	¤	¤	0¤	1,000¤	3500¤
5¤	¤	¤	¤	¤	0¤	5,000¤
6¤	¤	¤	¤	¤	¤	0¤

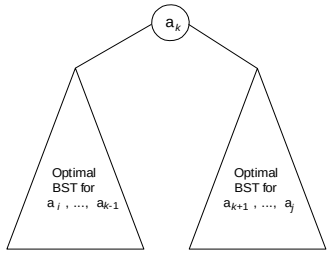
Optimal Binary Search Trees

- Problem: Given n keys $a_1 < \ldots < a_n$ and probabilities $p_1 \leq \ldots \leq p_n$ searching for them, find a BST with a minimum average number of comparisons in successful search.
- Since total number of BSTs with n nodes is given by C(2n,n)/(n+1), which grows exponentially, brute force is hopeless.
- Example: What is an optimal BST for keys A, B, C, and D with search probabilities 0.1, 0.2, 0.4, and 0.3, respectively?

DP for Optimal BST Problem

Let C[i,j] be minimum average number of comparisons made in T[i,j], optimal BST for keys $a_i < \ldots < a_j$, where $1 \le i \le j \le n$.

Consider optimal BST among all BSTs with some a_k $(i \le k \le j)$ as their root; T[i,j] is the best among them.



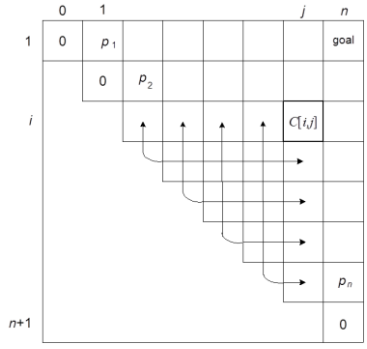
$$C[i,j] = \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \text{ (level } a_s \text{ in } T[i,k-1] +1) + \sum_{s=k+1}^{j} p_s \text{ (level } a_s \text{ in } T[k+1,j] +1)\}$$

DP for Optimal BST Problem (cont.)

After simplifications, we obtain the recurrence for C[i,j]:

$$C[i,j] = \min_{i \le k \le j} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n$$

$$C[i,i] = p_i \quad \text{for } 1 \le i \le j \le n$$



Example: key *A B C D*

probability 0.1 0.2 0.4 0.3

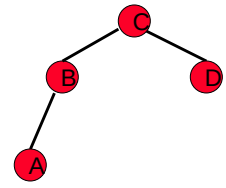
The tables below are filled diagonal by diagonal: the left one is filled using the recurrence

$$C[i,j] = \min_{i \le k \le j} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^{j} p_s, \quad C[i,i] = p_i;$$

the right one, for trees' roots, records k's values giving the minima

i \ j	0	1	2	3	4
1	0	.1	.4	1.1	1.7
2		0	.2	.8	1.4
3			0	.4	1.0
4				0	.3
5					0

i \ j	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					



optimal BST

CAL POLY SAN LUIS OBISPO Computer Science Department

ALGORITHM *OptimalBST*$(P[1..n])$

```
//Finds an optimal binary search tree by dynamic programming
//Input: An array P[1..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
//        optimal BST and table R of subtrees' roots in the optimal BST
for i ← 1 to n do
    C[i, i − 1] ← 0
    C[i, i] ← P[i]
    R[i, i] ← i
C[n + 1, n] ← 0
for d ← 1 to n − 1 do //diagonal count
    for i ← 1 to n − d do
        j ← i + d
        minval ← ∞
        for k ← i to j do
            if C[i, k − 1] + C[k + 1, j] < minval
                minval ← C[i, k − 1] + C[k + 1, j]; kmin ← k
        R[i, j] ← kmin
        sum ← P[i]; for s ← i + 1 to j do sum ← sum + P[s]
        C[i, j] ← minval + sum
return C[1, n], R
```

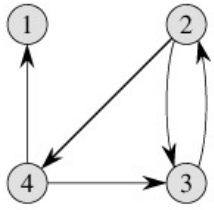
32

Analysis DP for Optimal BST Problem

- Time efficiency: $\Theta(n^3)$ but can be reduced to $\Theta(n^2)$ by taking advantage of monotonicity of entries in the root table, i.e., R[i,j] is always in the range between R[i,j-1] and R[i+1,j]
- Space efficiency: $\Theta(n^2)$
- Method can be expended to include unsuccessful searches

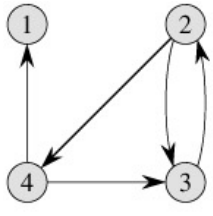
Warshall's Algorithm: Transitive Closure

- Computes the transitive closure of a relation
- Alternatively: existence of all nontrivial paths in a digraph



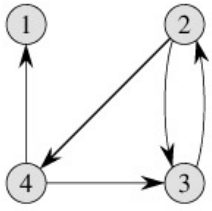
Warshall's Algorithm

- Constructs transitive closure T as the last matrix in the sequence of n-by-n matrices $R^{(0)}$, … , $R^{(k)}$, … , $R^{(n)}$ where
- $R^{(k)}[i,j]$ = 1 iff there is nontrivial path from i to j with only first k vertices allowed as intermediate
- Note that R(0) = A (adjacency matrix), R(n) = T (transitive closure
- Alternatively: existence of all nontrivial paths in a digraph



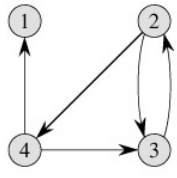
Warshall's Algorithm: Recurrence Relation

- On the k-th iteration, the algorithm determines for every pair of vertices i, j if a path exists from i and j with just vertices 1,…,k allowed as intermediate
- $R^{(k)}[i,j] = R^{(k-1)}[i,j] \vee (R^{(k-1)}[i,k] \wedge R^{(k-1)}[k,j])$



- If an element in row i and column j is 1 in $R^{(k-1)}$ it remains 1 in $R^{(k)}$
- If an element in row i and column j is 0 in $R^{(k-1)}$, change to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$

Warshall's Algorithm: Transitive Closure



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Warshall's Algorithm - pseudocode

ALGORITHM *Warshall*$(A[1..n, 1..n])$

```
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
R^(0) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R^(k)[i, j] ← R^(k−1)[i, j] or (R^(k−1)[i, k] and R^(k−1)[k, j])
return R^(n)
```

Time efficiency: $\Theta(n^3)$

Space efficiency: Matrices can be written over their predecessors

Floyd's Algorithm: All pairs shortest paths

- Problem: In a weighted (di)graph, find shortest paths between every pair of vertices
- Same idea: construct solution through series of matrices $D^{(0)}$, …, $D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

Floyd's Algorithm: Matrix computation

- On the k-th iteration, the algorithm determines shortest paths between every pair of vertices i, j that use only vertices among 1,…,k as intermediate
- $D^{(k)}[i,j] = \min \{D^{(k-1)}[i,j], D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$

Floyd's Algorithm - pseudocode

ALGORITHM *Floyd*$(W[1..n, 1..n])$

```
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

Time efficiency: $\Theta(n^3)$

Space efficiency: Matrices can be written over their predecessors

Matrix Chain Multiplication

- Suppose we want to multiply four matrices AxBxCxD of dimensions: 5x20, 20x1, 1x10, 10x100 and we would like to minimize the number of multiplications
- Two facts are important when considering the problem
  - When multiplying two matrices of dimensions n x m and m x p the number of multiplications is the product of the dimensions each appearing once – namely nmp.
  - Matrix multiplication is associative – that is we can multiply the matrices grouping them in any way we please and still get the same answer.
- Thus in this case there are 5 possible groupings A(B(CD), A((BC)D), (AB)(CD), ((AB)C)D, (A(BC))D. Note that the number of grouping grows at least as fast as 2^n.

Matrix Chain Multiplication: Example

- AxBxCxD of dimensions: 50x20, 20x1, 1x10, 10x100.
  Let ops = multiplications in what follows.
- A(B(CD)
  - CD requires 1,000 ops and results in a 1x100 matrix,
  - B(CD) this requires an additional 2,000 ops,
  - finally computing A(B(CD) requires 100,000 ops (50x20x100).
  - Thus the entire chain of multiplications requires 103,000 ops
- Carrying out similar calculations you find that:
  - A(B(CD) 103,000
  - A((BC)D) 120,200
  - (A(BC))D 60,200
  - (AB)(CD) 7,000
  - ((AB)C)D 51,500

Matrix Chain Multiplication Optimal Substructure: Finding the recurrence relation

- What might be a reasonable subproblem?
- Notation:
  - To multiply the matrices A_1, A_2, …A_n where the dimensions of A_i are d_{i-1} x d_i . Thus multiplying
    - » A_1 x A_2 requires d_0 x d_1 x d_2 operations
    - » A_k x A_{k+1} requires d_{k-1} x d_k x d_{k+1} operations
  - Let M(i,j) = the least number of multiplications need to multiply matrices from i to j.

- M(i, j) = **min** {M(i,k) + M(k+1,j) + d_{i-1} d_k d_j } overall all the indices k that split the problem.

- The base case is M(i, i) = 0 for 1 ≤ i ≤ n.

Matrix Chain Multiplication: example

Computing M(i,j)

A_1 be 30 x 35
A_2 be 35 x 15
A_3 be 15 x 5
A_4 be 5 x 10
A_5 be 10 x 20
A_6 be 20 x 25

j ¶ i¤	1¤	2¤	3¤	4¤	5¤	6¤
1¤	0¤	15,750¤	7875¤	9375¤	11,875¤	15,125¤
2¤	¤	0¤	2,625¤	4375¤	7125¤	10500¤
3¤	¤	¤	0¤	750¤	2500¤	5375¤
4¤	¤	¤	¤	0¤	1,000¤	3500¤
5¤	¤	¤	¤	¤	0¤	5,000¤
6¤	¤	¤	¤	¤	¤	0¤

Rod Cutting

Design a dynamic programming algorithm for the following problem. Find the maximum total sale price that can be obtained by cutting a rod of n units long into integer-length pieces if the sale price of a piece i units long is p_i for i = 1, 2, …, n . If p_n is large enough you may not need to cut the rod at all. What are the time and space efficiencies of your algorithm?

- Notes:
  - Brute force would require checking all the different ways to divide the rod. This would require checking 2^{n-1} different possibilities
  - What is optimal substructure. What are the different smaller problems that if you knew the answers then one of them might give you the answer to the larger problem?

Rod Cutting – Optimal substructure

- How to characterize the size of a problem?
  - Length of the rod
- What choice or option will create a smaller problem?
  - Cut at each possible point then for each piece use the optimal solution.
  - Take the optimal solution of each of these smaller problems.
  - Then claim the max of these along with “not cutting option” is the optimal solution.
- Proof: Two cases
  - Optimal solution contains a cut. Then each of the subproblems must be optimal otherwise could get a better solution by exchange argument
  - Optimal solution does not contain a cut. Then it must be better than any of two subproblems combined

Recurrence Relations : options

1. Val(j) = max {p_j, max $_{i=1 \text{ to } j-1}$ { Val(i)+Val(j-i) } Val(1)= p_1 for j>1
2. Val(j) = max {p_j, max $_{i=1 \text{ to } n}$ {p_i +Val(j-i) } where Val(0)=0 } for j>0

- Either of these will be exponential time - draw call tree for 2 if implemented in a straightforward recursive way
- Use a table to contain results to avoid recalculation
  - Bottom up straightforward
  - Memoization – check table to see if already calculated then recurse

Counting instructions and subproblem graphs

- See Corman
- DAG with
  - V = subproblems
  - E = dependencies between subproblems
- Reverse top sort gives a picture of the bottom-up computation
- DFS shows the computation top-down (need correct ordering of the adjacency list
- O(|V| + |E|) running time
- Does example for rod cutting