Dynamic Programming

Dynamic Programming

- Typically applied to optimization problems
  - characterize the structure of an optimal solution
  - recursively define the value of an optimal solution
  - compute optimal value in a bottom-up fashion
  - construct optimal solution steps from computed information (if more than the value is required)
- Coverage
  - example problem: assembly line scheduling
  - example problem: matrix chain multiplication
  - general characteristics of suitable problems
  - another problem: longest common subsequence

Dynamic Programming

- An algorithm design method for optimization problems
- Find the value of an optimal solution
  - Characterize the structure of an optimal solut.
  - Recursively define value of optimal solution
  - Compute value from bottom-up
  - (From path followed to compute the value, construct the optimal solution)

LONGEST COMMON SUBSEQUENCE

Longest Common Subsequence

- A subsequence of a given sequence is the given sequence with some elements (maybe none) left out; for example, < B, M, K, L > is a subsequence of < A, B, M, G, P, K, D, L, R >
- Longest Common Subsequence
  - Given X=<A,B,C,B,D,A,B> and Y=<B,D,C,A,B,A>
  - <B,C,A> is a common subsequence of length 3
  - <B,C,B,A> is the longest common subsequence, it has length 4
  - Given two sequences we want to find the maximum length of any common subsequence

Optimal substructure of an LCS

- For a sequence of length m, there are 2^m subsequences, so enumeration is impractical
- Given a sequence $X=<x_1,x_2, .. ,x_m>$ the prefix $X_i=< x_1,x_2, .. x_i>$

Theorem 16.1 (Optimal substructure of an LCS)

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

1. 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}.
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.
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}.

- This shows the LCS problem has an optimal-substructure property

A recursive sol. to subproblems

- Overlapping subproblems
  - if $x_m = y_n$ then we must find the LCS of X_{m-1} and Y_{n-1} and then append $x_m = y_n$ to this result
  - if $x_m \neq y_n$ then we must find the LCS of X_{m-1} and Y and the LCS of X and Y_{n-1} ; whichever LCS is longer is the LCS of X and Y
- A recursive formula for c[i,j], the length of the LCS for X_i and Y_j

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases} \qquad (16.5)$$

A Sample Solution

i \ j		0	1	2	3	4	5	6
	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

- The answer is 4 from the c[m,n] entry
- If the letters match, the arrow points ↖ and the entry is one greater
- If the letters do not match the value is the maximum of left and up and the arrow points to the larger of left or up (and up if equal)

Finding the LCS Length

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

- If one sequence is empty the LCS has length 0
- The table b[1..m,1..n] points to the table entry corresponding to the optimal subproblem solution when computing c[i,j]
- Lines 7 through 16 compute the tables b and c in row major order; the structure matches the three cases outlined ($x_m = y_n$ and two cases for $x_m \neq y_n$)
- The figure on the next slide shows how the calculations are performed
- Since the table c is (m+1) x (n+1) and each entry takes O(1) to compute, the complexity of the algorithm is Θ(mn)

Constructing an LCS

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

- Intuitively you start at c[m,n] and follow the arrows; you print the character whenever the arrow is ↖
- This would print an LCS in reverse order, so the actual code is written recursively with the printing done on exit from recursion; this will print the LCS from left to right
- The complexity of printing an LCS is Θ(m + n)

Parenthasization in Matrix chain multiplication

MATRIX CHAIN MULTIPLICATION

Another example: matrix mult

- Compute the product of n mutually compatible matrices A_1 A_2 … A_n
- How many way of computing the matrix product?
- Is there an optimal way? (fewer scalar products)?

Example

- n = 3
- A1: 10 x 100
- A2: 100 x 5
- A3: 5 x 50

- (A1 A2) A3: 10x100x5+10x5x50 = 7500
- A1 (A2 A3): 100x5x50+10x100x50 = 75000

Number of Parenthesizations

- The basic recurrence is

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 . \end{cases}$$

- The solution is known as Catalan numbers

$$P(n) = C(n-1) ,$$

where

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n / n^{3/2}) .$$

- Since this is exponential in n, a brute force approach is not practical

Structure of Optimal Parenthesizatin

- To find the optimal parenthesization of A_1 .. A_n has some final multiplication A_1 .. A_k * A_{k+1} .. A_n
- The key observation is that the prefix chain A_1 .. A_k must also be optimal; the same is true of the postfix chain A_{k+1} .. A_n
- The optimal solution to the matrix-chain multiplication must contain optimal solutions to subproblem instances
- This is a general characteristic of problems for which dynamic programming is well suited

A Recursive Solution

- **Let m[i,j] be the minimum # mult to compute $A_{i..j}$**
- **A recursive solution**
  - **if i = j, there is only one matrix and 0 multiplications**
  - **if i < j then we must find the k such that the number of mult for $A_{i..k}$ plus the number for $A_{k+1..j}$ plus the number for performing the final mult. is minimal**

$$m[i,j] = \begin{cases} 0 & \text{if } i = j , \\ \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j . \end{cases}$$

  - For convenience we define s[i,j] equals a value k such that m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j

Overlapping Subproblems

- The total number of subproblems is relatively small, the number of i and j such that 1 ≤ i ≤ j ≤ n is

$$\binom{n}{2} + n = \Theta(n^2)$$

- A recursive algorithm would be exponential since it encounters each subproblem many times
- The key idea is to solve each subproblem only once and build more complex solutions from the bottom up

The Basic Algorithm

- Assume A_i has size p_{i-1} x p_i
- The algorithm is given the n+1 values <p_0,p_1, .. , p_n>
- A table m[1..n,1..n] stores each of the costs m[i,j]

```
n ← length[p] − 1
for i ← 1 to n
    do m[i,i] ← 0
for l ← 2 to n
    do for i ← 1 to n − l + 1
           do j ← i + l − 1
              m[i,j] ← ∞
              for k ← i to j − 1
                  do q ← m[i,k] + m[k+1,j] + p_{i−1} p_k p_j
                     if q < m[i,j]
                        then m[i,j] ← q
                             s[i,j] ← k
return m and s
```

- An auxiliary table s[1..n,1..n] contains the index k that obtained the minimum
- m[i,i] is set to 0 first; m[i,i+1] is computed next
- then m[i,i+2] is computed and the process continues in this fashion until all the m[i,j] are computed

Struct. of the optimal solution

- Suppose the optimal order to multiply optimally $A_i \ldots A_j$ is to:
  - first multiply $A_i \ldots A_k$
  - then $A_{k+1} \ldots A_j$
  - and finally multiply the two resulting matrices
- Key observation:
  the products $A_i \ldots A_k$ and $A_{k+1} \ldots A_j$ must be done in optimal order

Example

i \ j	1	2	3	4	5	6
1	0	30x 35x 15				
2		0	35x 15x 5			
3			0	15x 5x 10		
4				0	5x 10x 20	
5					0	10x 20x 25
6						0

Matrix	nxm
A1	30 x 35
A2	35 x 15
A3	15 x 5
A4	5 x 10
A5	10 x 20
A6	20 x 25

Example

i \ j	1	2	3	4	5	6
1	0	A_{12}=30x15 15.750	A_{123}=30x5 7.875			
2		0	A_{23}=35x5 2.625			
3			0	A_{34}=15x10 750		
4				0	A_{45}=5x20 1.000	
5					0	A_{56}=10x25 5.000
6						0

Min(A12*A3,
A1*A23)

Min: {15.750 + 30x15x5;
2.625 + 30x35x5}
= Min{18,000;
7.875}
= **7.875**

Matrix	nxm
A1	30 x 35
A2	35 x 15
A3	15 x 5
A4	5 x 10
A5	10 x 20
A6	20 x 25

Example

i \ j	1	2	3	4	5	6
1	0	A_{12}=30x15 15.750	A_{123}=30x5 7.875			
2		0	A_{23}=35x5 2.625	A_{234}=35x10 3.375		
3			0	A_{34}=15x10 750		
4				0	A_{45}=5x20 1.000	
5					0	A_{56}=10x25 5.000
6						0

Min: {2.625 + 35x5x10;
750 + 35x15x10}
= Min{43,750;
3.375}
= **3,375**

Min(A23*A4,
A2*A34)

Matrix	nxm
A1	30 x 35
A2	35 x 15
A3	15 x 5
A4	5 x 10
A5	10 x 20
A6	20 x 25

Example

i \ j	1	2	3	4	5	6
1	0	15.750	7.875			
2		0	2.625	4.375	7.125	
3			0	750	2.500	
4				0	1.000	3.500
5					0	5.000
6						0

Min: {A_2xA_{345};
A_{23}xA_{45};
A_{234}xA_5} =
Min{35x15x20 + 2.500;
2.625 + 35x5x20 + 1.000;
4.375 + 35x10x20}
= Min{13,000;
7.125;
11,375}
= **7.125**

Matrix	nxm
A1	30 x 35
A2	35 x 15
A3	15 x 5
A4	5 x 10
A5	10 x 20
A6	20 x 25

min(A2 * 345,
A23 * A45,
A234 * A5
)

Example

i \ j	1	2	3	4	5	6
1	0	15,750	7,875	9,375		
2		0	2,625	4,375	7,125	
3			0	750	2,500	5,375
4				0	1,000	3,500
5					0	5,000
6						0

Matrix	nxm
A1	30 x 35
A2	35 x 15
A3	15 x 5
A4	5 x 10
A5	10 x 20
A6	20 x 25

min(????????
)

What is dynamic programming?

- Dynamic programming is a method of solving optimization problems by combining the solutions of subproblems
- Developing these algorithms follows four steps:
  1. Characterize the structure of an optimal solution
  2. Recursively define the value of an optimal solution
  3. Compute the optimal solution, typically bottom-up
  4. Construct the path of an optimal solution (if desired)

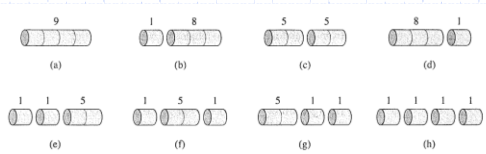
Example – Rod Cutting

- Problem: Given a rod of length *n* inches and a table of prices, determine the maximum revenue obtainable by cutting up the rod and selling the pieces
  - Rod cuts are an integral number of inches, cuts are free
  - Price table for rods

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

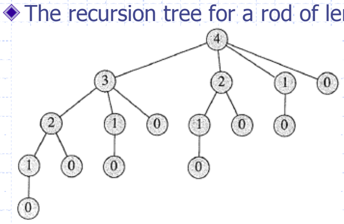
Example – Rod Cutting

- Eight possible ways to cut a rod of length 4 (prices shown on top)

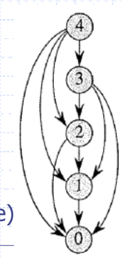


Example – Rod Cutting

- The recursion tree for a rod of length 4



- The subproblem graph (collapsed tree)



Example – Rod Cutting

- Recursive equation: $r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$
- Bottom-up algorithm – O(n²) from double nesting

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

Example – Rod Cutting

- Extended bottom-up algorithm obtains path

```
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = −∞
    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
```

- Print solution

Knapsack Problem

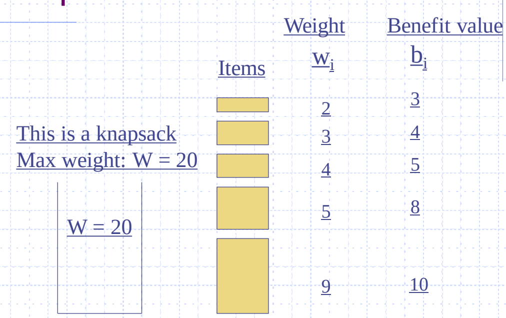
The Knapsack Problem

- The famous *knapsack problem*:
  - A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

0-1 Knapsack problem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i , b_i and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem: a picture



The Knapsack Problem

- More formally, the *0-1 knapsack problem*:
  - The thief must choose among n items, where the ith item worth v_i dollars and weighs w_i pounds
  - Carrying at most W pounds, maximize value
    - Note: assume v_i, w_i, and W are all integers
    - "0-1" b/c each item must be taken or left in entirety
- A variation, the *fractional knapsack problem*:
  - Thief can take fractions of items
  - Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust

0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is called a *"0-1"* problem, because each item must be entirely accepted or rejected.
- Just another version of this problem is the "*Fractional Knapsack Problem*", where we can take fractions of items.

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to W
- Running time will be $O(2^n)$

0-1 Knapsack problem: brute-force approach

- Can we do better?
- Yes, with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Let's try this:
If items are labeled $1..n$, then a subproblem would be to find an optimal solution for S_k = {items labeled 1, 2, .. k}

Defining a Subproblem

If items are labeled $1..n$, then a subproblem would be to find an optimal solution for S_k = {items labeled 1, 2, .. k}

- This is a valid subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we can't do that. Explanation follows....

Defining a Subproblem

$w_1=2$, $b_1=3$	$w_2=4$, $b_2=5$	$w_3=5$, $b_3=8$	$w_4=3$, $b_4=4$	?

Max weight: W = 20
For S_4:
Total weight: 14;
total benefit: 20

$w_1=2$, $b_1=3$	$w_2=4$, $b_2=5$	$w_3=5$, $b_3=8$	$w_4=9$, $b_4=10$

For S_5:
Total weight: 20
total benefit: 26

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

S_4 (items 1–4), S_5 (items 1–5)

Solution for S_4 is not part of the solution for S_5!!!

Defining a Subproblem (continued)

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!
- Let's add another parameter: w, which will represent the exact weight for each subset of items
- The subproblem then will be to compute $B[k,w]$

Recursive Formula for subproblems

- Recursive formula for subproblems:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- It means, that the best subset of S_k that has total weight w is one of the two:

1) the best subset of S_{k-1} that has total weight w, **or**

2) the best subset of S_{k-1} that has total weight $w-w_k$ plus the item k

Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight *w*, either contains item *k* or not.
- First case: $w_k > w$. Item *k* can't be part of the solution, since if it was, the total weight would be > *w*, which is unacceptable
- Second case: $w_k <= w$. Then the item *k* can be in the solution, and we choose the case with greater value

The Knapsack Problem And Optimal Substructure

- Both variations exhibit optimal substructure
- To show this for the 0-1 problem, consider the most valuable load weighing at most *W* pounds
  - *If we remove item j from the load, what do we know about the remaining load?*
  - A: remainder must be the most valuable load weighing at most $W - w_j$ that thief could take from museum, excluding item j

Solving The Knapsack Problem

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - *How?*
- The optimal solution to the 0-1 problem cannot be found with the same greedy strategy
  - Greedy strategy: take in order of dollars/pound
  - Example: 3 items weighing 10, 20, and 30 pounds, knapsack can hold 50 pounds
    - *Suppose item 2 is worth \$100. Assign values to the other items so that the greedy strategy will fail*

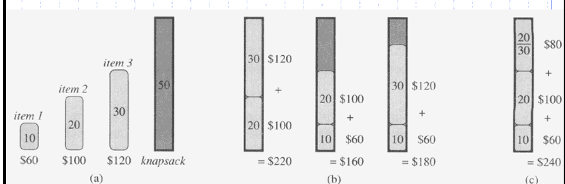


Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

The Knapsack Problem: Greedy Vs. Dynamic

- The fractional problem can be solved greedily
- The 0-1 problem cannot be solved with a greedy approach
  - As you have seen, however, it can be solved with dynamic programming

0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
    for w = 0 to W
        if w_i <= w // item i can be part of the solution
            if b_i + B[i-1,w-w_i] > B[i-1,w]
                B[i,w] = b_i + B[i-1,w- w_i]
            else
                B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // w_i > w
```

Running time

for w = 0 to W *O(W)*
  B[0,w] = 0
for i = 0 to n Repeat *n* times
  B[i,0] = 0
  for w = 0 to W *O(W)*
    < the rest of the code >

What is the running time of this algorithm?

O(n*W)

Remember that the brute-force algorithm takes $O(2^n)$

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example (2)

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

for w = 0 to W
  B[0,w] = 0

Example (3)

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

for i = 0 to n
  B[i,0] = 0

Example (4)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0 →	0			
2	0				
3	0				
4	0				
5	0				

i=1
b_i=3
w_i=2
w=1
w-w_i =-1

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Example (5)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	**3**			
3	0				
4	0				
5	0				

i=1
b_i=3
w_i=2
w=2
w-w_i =0

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (6)

W i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	**3**			
4	0				
5	0				

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=1
b_i=3
w_i=2
w=3
w-w_i=1

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (7)

W i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	**3**			
5	0				

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=1
b_i=3
w_i=2
w=4
w-w_i=2

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (8)

W i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0	**3**			

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=1
b_i=3
w_i=2
w=5
w-w_i=2

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (9)

W i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0**		
2	0	3			
3	0	3			
4	0	3			
5	0	3			

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=2
b_i=4
w_i=3
w=1
w-w_i=-2

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Example (10)

W i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	**3**		
3	0	3			
4	0	3			
5	0	3			

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=2
b_i=4
w_i=3
w=2
w-w_i=-1

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

Example (11)

W i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	**4**		
4	0	3			
5	0	3			

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i=2
b_i=4
w_i=3
w=3
w-w_i=0

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (12)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	**4**		
5	0	3			

$i=2$
$b_i=4$
$w_i=3$
$w=4$
$w-w_i=1$

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (13)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3	**7**		

$i=2$
$b_i=4$
$w_i=3$
$w=5$
$w-w_i=2$

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (14)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	**0**	
2	0	3	3	**3**	
3	0	3	4	**4**	
4	0	3	4		
5	0	3	7		

$i=3$
$b_i=5$
$w_i=4$
$w=1..3$

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (15)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	**5**	
5	0	3	7		

$i=3$
$b_i=5$
$w_i=4$
$w=4$
$w- w_i=0$

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (15)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	**7**	

$i=3$
$b_i=5$
$w_i=4$
$w=5$
$w- w_i=1$

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (16)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	**0**
2	0	3	3	3	**3**
3	0	3	4	4	**4**
4	0	3	4	5	**5**
5	0	3	7	7	

$i=3$
$b_i=5$
$w_i=4$
$w=1..4$

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (17)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	3	3	3	3
3	0	3	4	4	4
4	0	3	4	5	5
5	0	3	7	7 →	7

i=3
b_i=5
w_i=4
w=5

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
- To know the items that make this maximum value, an addition to this algorithm is necessary
- Please see LCS algorithm from the previous lecture for the example how to extract this data from the table we built

ASSEMBLYLINE SCHEDULING (DO YOURSELF)

Optimization problem

- There are two "parallel" assembly lines
  - Each assembly line can perform any job
  - There is a cost to switch between assembly lines
- Find an optimal sequence of stations from assembly line 1 or 2 so that the total transit time is minimized
- Brute force attempt is infeasible
  (requires to examine $\Omega(2^n)$ possibilities)

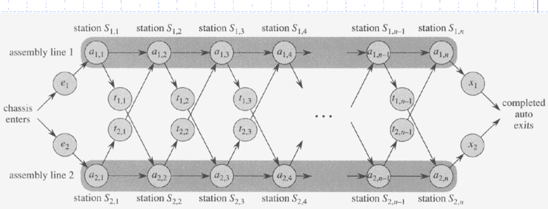


Figure 15.1 A manufacturing problem to find the fastest way through a factory. There are two assembly lines, each with n stations; the jth station on line i is denoted $S_{i,j}$ and the assembly time at that station is $a_{i,j}$. An automobile chassis enters the factory, and goes onto line i (where $i = 1$ or 2), taking e_i time. After going through the jth station on a line, the chassis goes on to the $(j+1)$st station on either line. There is no transfer cost if it stays on the same line, but it takes time $t_{i,j}$ to transfer to the other line after station $S_{i,j}$. After exiting the nth station on a line, it takes x_i time for the completed auto to exit the factory. The problem is to determine which stations to choose from line 1 and which to choose from line 2 in order to minimize the total time through the factory for one auto.

Optimal Substructure

- An optimal solution to the entire problem depends on optimal solutions to subproblems
- We formulate a solution for the assembly line problem stage-by-stage

Thus, the fastest way through station $S_{1,j}$ is either

- the fastest way through station $S_{1,j-1}$ and then directly through station $S_{1,j}$, or
- the fastest way through station $S_{2,j-1}$, a transfer from line 2 to line 1, and then through station $S_{1,j}$.

Using symmetric reasoning, the fastest way through station $S_{2,j}$ is either

- the fastest way through station $S_{2,j-1}$ and then directly through station $S_{2,j}$, or
- the fastest way through station $S_{1,j-1}$, a transfer from line 1 to line 2, and then through station $S_{2,j}$.

A Recursive Solution

- After some manipulation, we find the following recurrence relation

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1\,, \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1\,, \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

- Where $f_i[j]$ is the fastest possible time to $S_{i,j}$
  $t_{i,j}$ is the transfer time from $S_{i,j}$
  $a_{i,j}$ is the time at $S_{i,j}$
  e_i is the entry time for assembly line i

Recursion is not a good solution technique

- The same subproblems are solved again & again
- Let $r_i(j)$ be the number of references to $f_i[j]$

$r_1(n) = r_2(n) = 1$.

From the recurrences (15.6) and (15.7), we have

$r_1(j) = r_2(j) = r_1(j+1) + r_2(j+1)$

- It can be shown $r_i(j) = 2^{n-j}$
- So $f_i[1]$ is referenced 2^{n-1} times
- Our improved strategy is to build a table to save previous results so that they don't have to be recomputed each time

Inefficient implementation

```
public int f1(int j) {
  if (j==1) return e1+a1[1];
  else {
     int temp1 = f1(j-1);
     int temp2 = f2(j-1)+t2[j-1];
     return min(temp1,temp2)+a1[j];
  }
}
```

Similarly for `public int f2(int j)`

Continuation

All that is still needed is to:

- Initialize the cost vectors
  - a1[...]
  - a2[...]
  - t1[...]
  - t2[...]
- Initialize the entry/exit costs x1, x2, e1, e2
- Do the initial call:

```
fStar = min(f1(n) + x1;   f2(n) + x2);
```


Complexity

- Claim:
  - f1(1) and f2(1) both require execution time $\Theta(2^{n-1})$

Sketch of proof:

Let T(j) be the time taken to execute fx(j) (x=1 or 2)

T(1) = c (no further recursive calls)

T(j) = c + 2 T(j-1) (fx(j) calls both f1(j-1) and f2(j-1))

Expand the expression until T(1)

A better method

- Compute fx(j) in a better order
- Each fx(j) depends only on f1(j-1) and f2(j-1)
- Compute fx(j) in increasing order of j
- Fill out two vectors f1[j] and f2[j] for j=1,2,...,n

Example of code

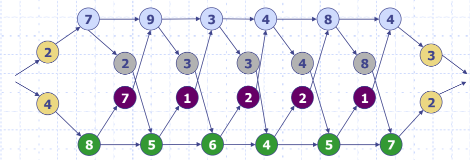
```
public int fastestWay() {
  f1[1] = e1+a1[1];
  f2[1] = e2+a2[1];
  for (int j=2; j<=n; j++) {
      f1[j] = min(f1[j-1],
                  f2[j-1] + t2[j-1])
               +a1[j-1];

      f2[j] = min(f1[j-1] + t1[j-1],
                  f2[j-1])
               +a2[j-1];
  }
  return min(f1[n]+x1; f2+x2);
}
```

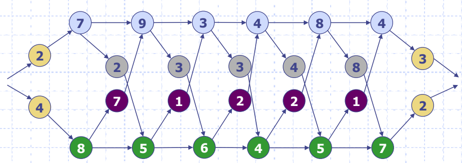

For example in book

j	1	2	3	4	5	6
a1	7	9	3	4	8	4
t2	2	3	1	3	4	
t1	2	1	2	2	1	
a2	8	5	6	4	5	7

e1	2
e2	4
x1	3
x2	2



Example - continued



j	1	2	3	4	5	6
f1	9	18	20	24	32	35
f2	12	16	22	25	30	37

Min finish time: 35+3 = 38

Additional details

- Keep track of which line (1 or 2) is used at each step
- Construct the optimal solution from such sequence of line choices

Notice: solution will be built by going backward from exit to stations 6, 5, …1, to start

- Complexity: $\Theta(n)$

More applications for Dyn. Prog.

- Find the optimal order of multiplying matrices
- Longest common subsequence
- Graph algorithms
  - Optimal paths
- Knapsack problem

More on dynamic programming

When is a problem suitable for D.P.?

- Optimization problems
- Overlapping subproblems

- Memoization:
  - Use the (easy) recursive definition
  - Keep track of what has already been computed and read it off a table (memo) instead of recomputing it

Complexity of Dynamic Solution

- Matrix-chain-order has three nested loops with indices having at most n values; so the complexity is $O(n^3)$
- The tables m and s each require n^2 space
- Overall the algorithm is much more efficient than the exponential time method of enumerating all possible parentheses and checking each one

Constructing an Optimal Solution

- The table s[1..n,1..n] helps reconstruct the solution
- s[i,j] records the value k that splits $A_{i..j}$ optimally

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
  if j > i
     then X ← MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
          Y ← MATRIX-CHAIN-MULTIPLY(A, s, s[i, j] + 1, j)
          return MATRIX-MULTIPLY(X, Y)
     else return A_i
```


Dynamic Programming Characteristics

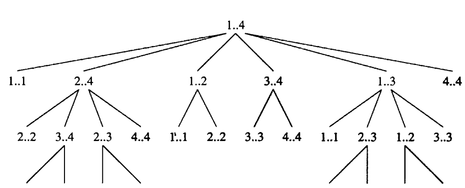
- Optimal Substructure
  - the optimal solution to the problem contains within it optimal solutions to subproblems
  - proof that subproblems also have to be optimal is usually done by contradiction
  - you must then find a suitable space for the subproblems
    - for the matrix chain problem the subspace of all arbitrary sequences of the input chain is unnecessarily large
    - chains of the form A_i through A_j where $1 \le i \le j \le n$ were adequate to solve the problem

Overlapping Subproblems - 1

- A recursive solution will typically solve the same subproblem again and again

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
  if i = j
     then return 0
  m[i, j] ← ∞
  for k ← i to j − 1
      do q ← RECURSIVE-MATRIX-CHAIN(p, i, k)
               + RECURSIVE-MATRIX-CHAIN(p, k + 1, j) + p_{i−1} p_k p_j
         if q < m[i, j]
            then m[i, j] ← q
  return m[i, j]
```


Overlapping Subproblems - 2



- The total number of *distinct* problems is usually polynomial in size

Complexity of Recursive Solution

- The recursive code has the complexity:

$$T(1) \ge 1,$$

$$T(n) \ge 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{for } n > 1$$

- Solving this recurrence relation shows:

$$\begin{aligned} T(n) &\ge 2\sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2\sum_{i=0}^{n-2} 2^i + n \\ &= 2(2^{n-1} - 1) + n \\ &= (2^n - 2) + n \\ &\ge 2^{n-1}, \end{aligned}$$

Top Down with Memoization

```
MEMOIZED-MATRIX-CHAIN(p)
n ← length[p] − 1
for i ← 1 to n
    do for j ← i to n
           do m[i, j] ← ∞
return LOOKUP-CHAIN(p, 1, n)

LOOKUP-CHAIN(p, i, j)
if m[i, j] < ∞
   then return m[i, j]
if i = j
   then m[i, j] ← 0
   else for k ← i to j − 1
            do q ← LOOKUP-CHAIN(p, i, k)
                     + LOOKUP-CHAIN(p, k + 1, j) + p_{i−1} p_k p_j
               if q < m[i, j]
                  then m[i, j] ← q
return m[i, j]
```

- The form of the program looks recursive, but once a problem is solved, the solution is stored to avoid resolving the same problem
- Each of the n^2 calls to lookup-chain takes O(n) time, so the overall complexity is $O(n^3)$

Top-down vs. Bottom-up

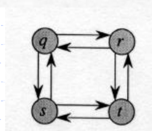
- If all subproblems must be solved to find an optimal solution then the bottom-up approach of dynamic programming is usually faster
- If some subproblems don't need be solved, then the top-down memorized solution can be faster (why?)

Not All Problems Are Suitable - 1

- Consider the following problem statement:

Unweighted shortest path:[2] Find a path from u to v consisting of the fewest edges. Such a path must be simple, since removing a cycle from a path produces a path with fewer edges.

- This problem has optimal substructure, two shorter paths can be combined to give a shortest path
- This works because the two shorter paths, say from q → r and r → t, combine to give the shortest path from q → t



Not All Problems Are Suitable - 2

- Consider the following problem statement:

Unweighted longest simple path: Find a simple path from u to v consisting of the most edges. We need to include the requirement of simplicity because otherwise we can traverse a cycle as many times as we like to create paths with an arbitrarily large number of edges.

- This problem does not have optimal substructure, combining two longest paths may produce an "illegal" path with a cycle
- The problem is the subproblems are not independent and they may share edges