Dynamic Programming

Dynamic Programming

- An algorithm design technique (like divide and conquer)
- Divide and conquer
  - Partition the problem into independent subproblems
  - Solve the subproblems recursively
  - Combine the solutions to solve the original problem

Dynamic Programming

- Applicable when subproblems are **not** independent
  - Subproblems share subsubproblems

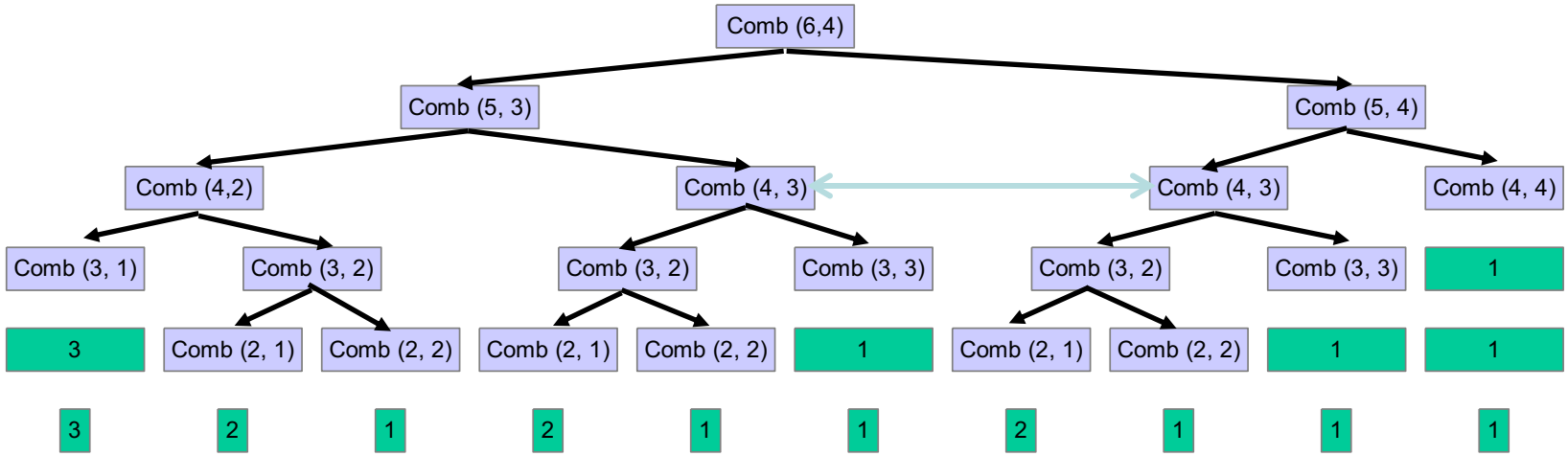
E.g.: Combinations:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

$$\binom{n}{1} = n \qquad \binom{n}{n} = 1$$

  - A divide and conquer approach would repeatedly solve the common subproblems
  - Dynamic programming solves every subproblem just once and stores the answer in a table

Example: Combinations



$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Dynamic Programming

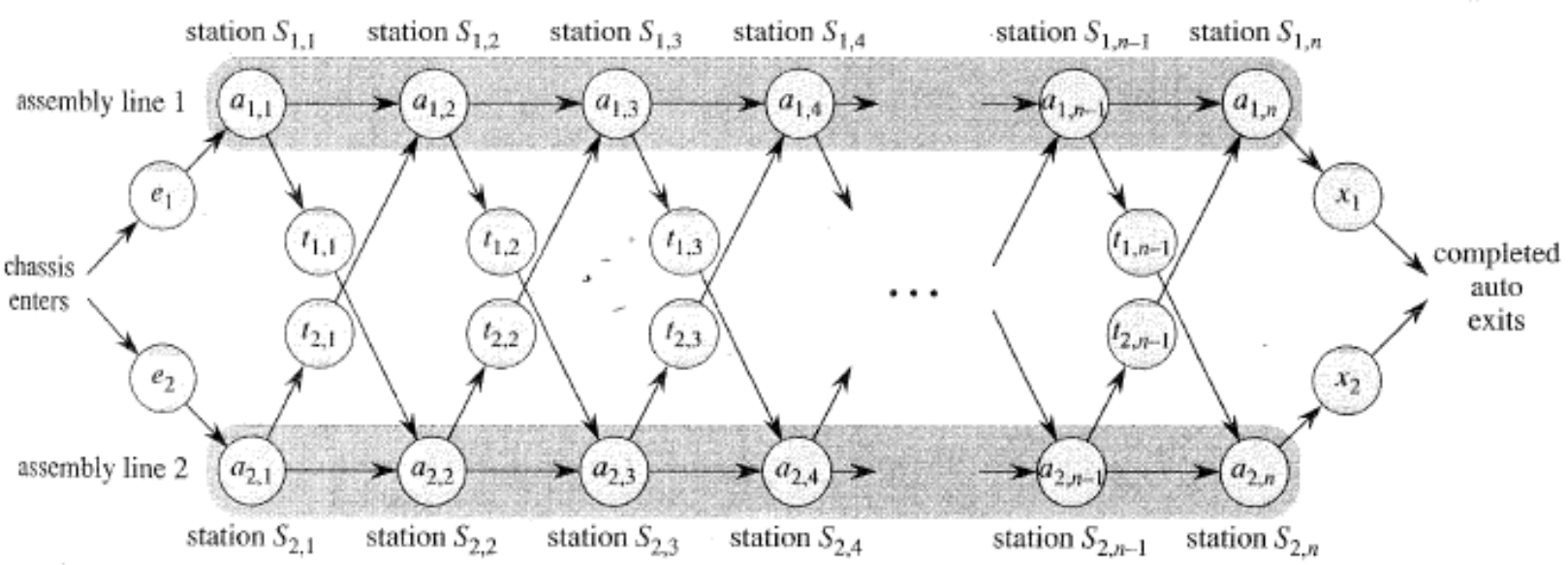
- Used for **optimization problems**
  - A set of choices must be made to get an optimal solution
  - Find a solution with the optimal value (minimum or maximum)
  - There may be many solutions that lead to an optimal value
  - Our goal: **find an optimal solution**

Dynamic Programming Algorithm

1. **Characterize** the structure of an optimal solution
2. **Recursively** define the value of an optimal solution
3. **Compute** the value of an optimal solution in a bottom-up fashion
4. **Construct** an optimal solution from computed information (not always necessary)

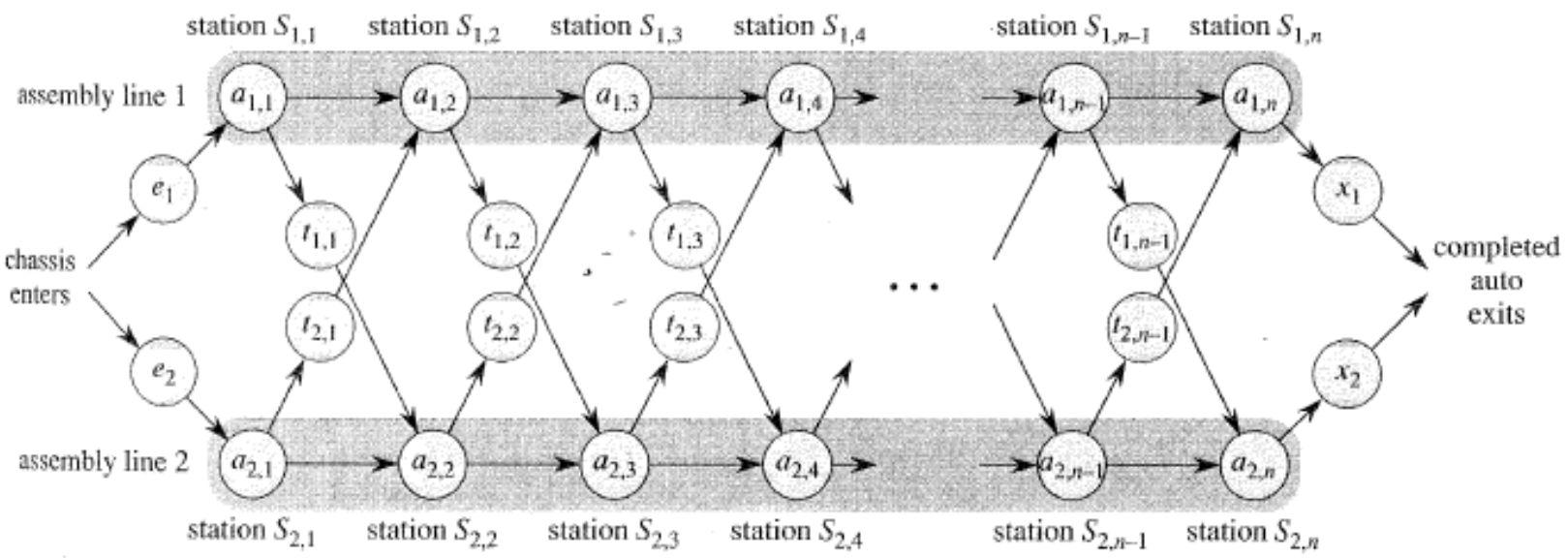
Assembly Line Scheduling

- Automobile factory with two assembly lines
  - Each line has n stations: $S_{1,1}, \ldots, S_{1,n}$ and $S_{2,1}, \ldots, S_{2,n}$
  - Corresponding stations $S_{1,j}$ and $S_{2,j}$ perform the same function but can take different amounts of time $a_{1,j}$ and $a_{2,j}$
  - Entry times are: e_1 and e_2; exit times are: x_1 and x_2



Assembly Line Scheduling

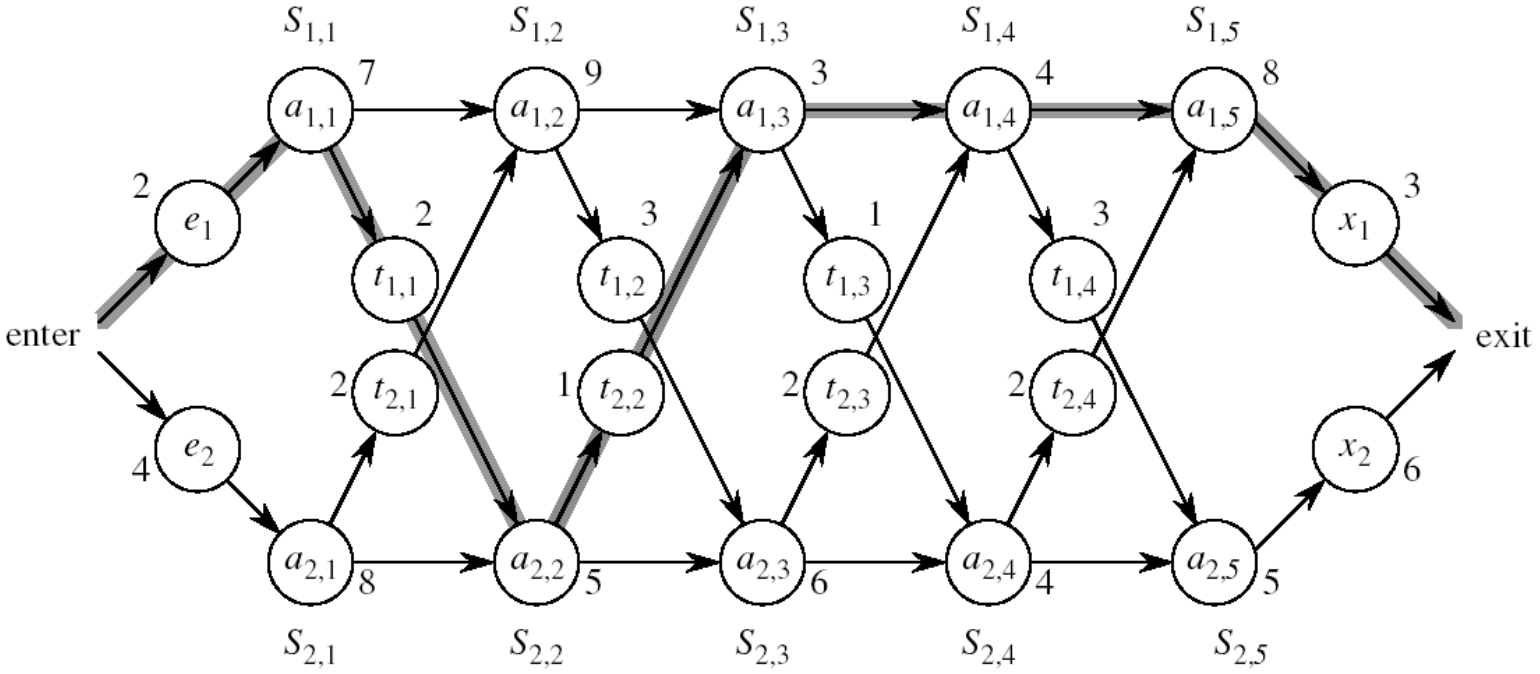
- After going through a station, can either:
  - stay on same line at no cost, or
  - transfer to other line: cost after $S_{i,j}$ is $t_{i,j}$, $j = 1, \ldots, n - 1$



Assembly Line Scheduling

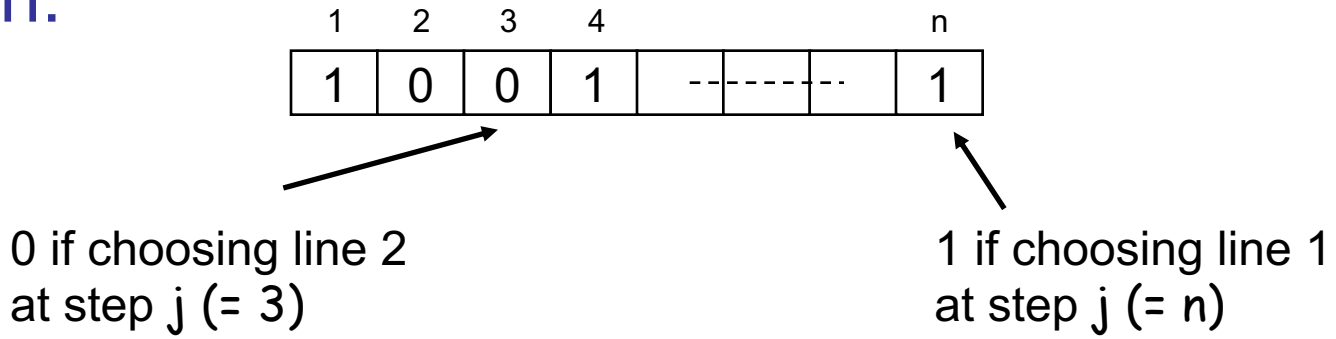
- Problem:

  what stations should be chosen from line 1 and which from line 2 in order to minimize the total time through the factory for one car?



One Solution

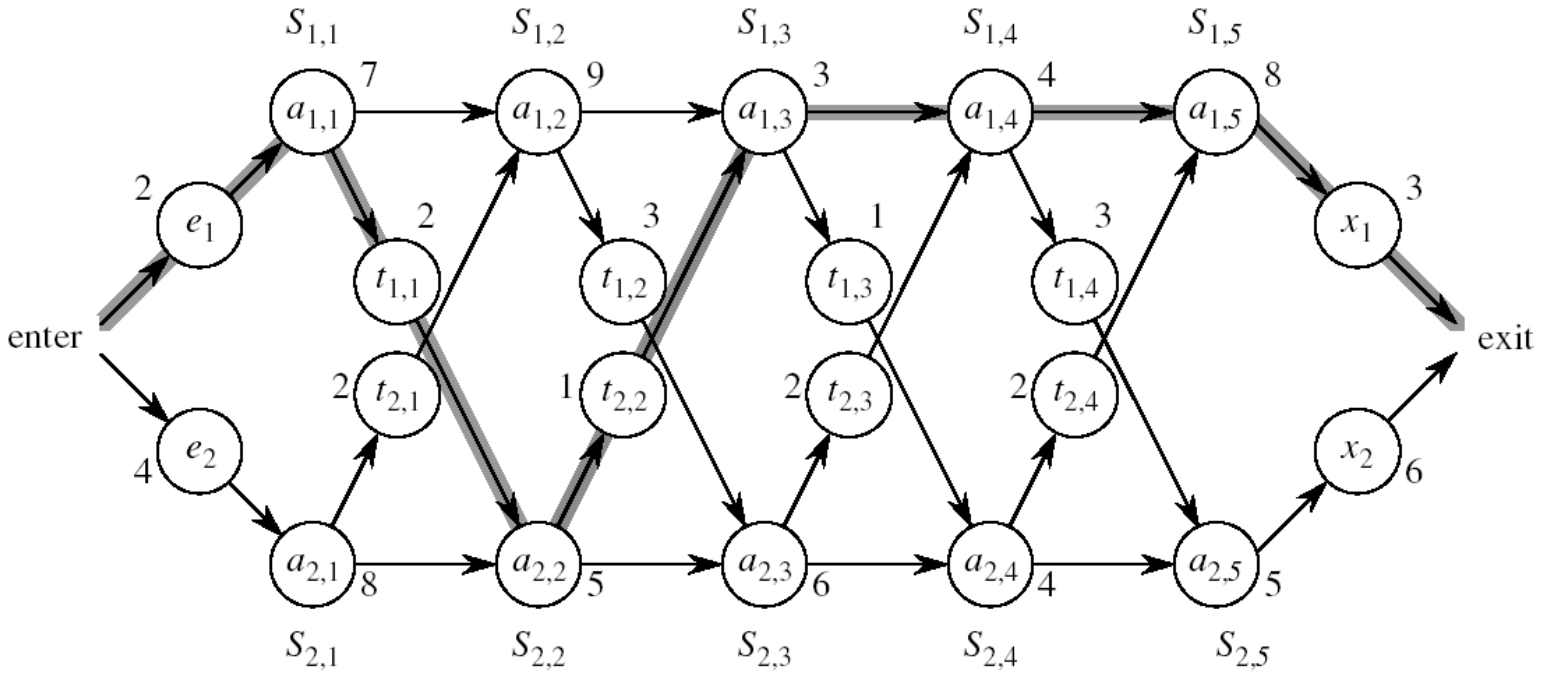
- Brute force
  - Enumerate all possibilities of selecting stations
  - Compute how long it takes in each case and choose the best one
- Solution:



  - There are 2^n possible ways to choose stations
  - Infeasible when n is large!!

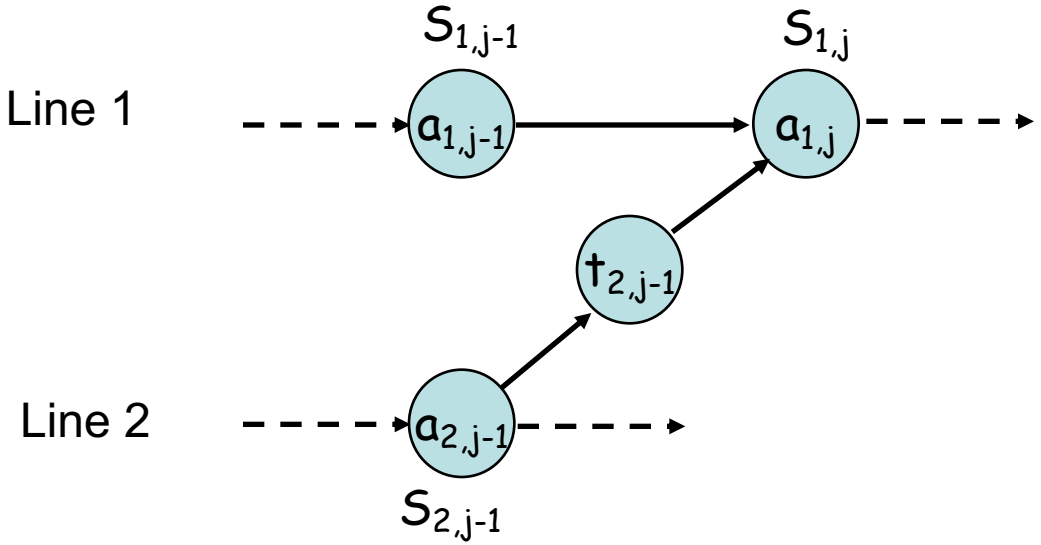
1. Structure of the Optimal Solution

- How do we compute the minimum time of going through a station?



1. Structure of the Optimal Solution

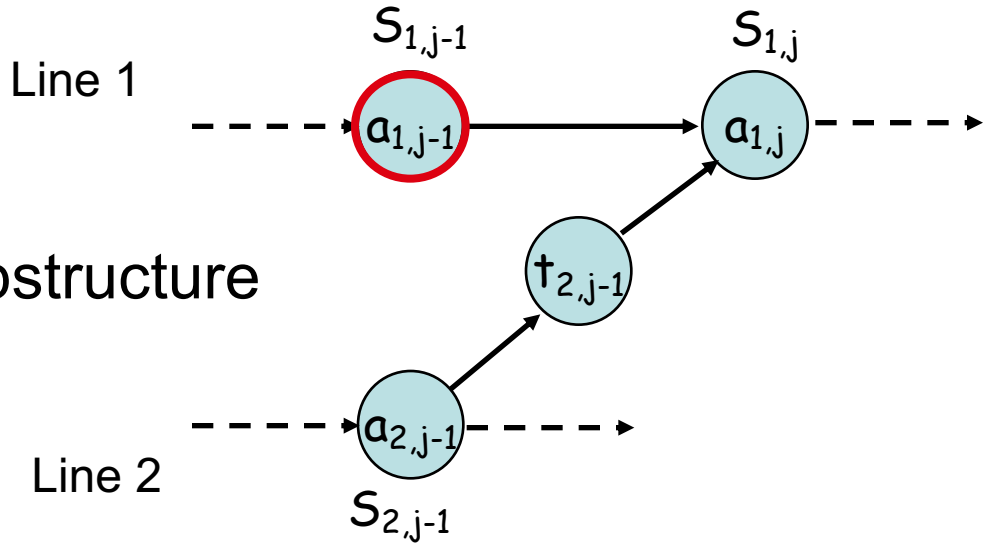
- Let's consider all possible ways to get from the starting point through station $S_{1,j}$
  - We have two choices of how to get to $S_{1,j}$:
    - Through $S_{1,j-1}$, then directly to $S_{1,j}$
    - Through $S_{2,j-1}$, then transfer over to $S_{1,j}$



1. Structure of the Optimal Solution

- Suppose that the fastest way through $S_{1,j}$ is through $S_{1,j-1}$
  - We must have taken a fastest way from entry through $S_{1,j-1}$
  - If there were a faster way through $S_{2,j-1}$, we would use it instead

Optimal Substructure



Optimal Substructure

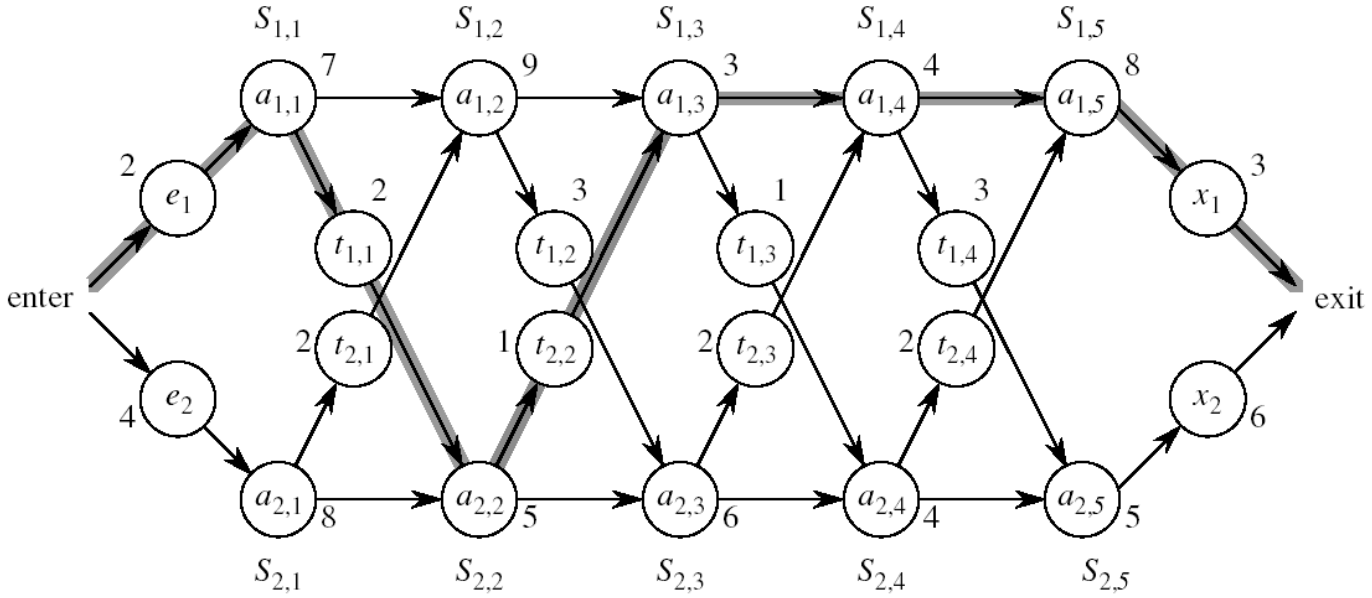
- **Generalization**: an optimal solution to the problem "*find the fastest way through* $S_{1,j}$" contains within it an optimal solution to subproblems: "*find the fastest way through* $S_{1,j-1}$ *or* $S_{2,j-1}$".

- This is referred to as the **optimal substructure** property

- We use this property to construct an optimal solution to a problem from optimal solutions to subproblems

2. A Recursive Solution

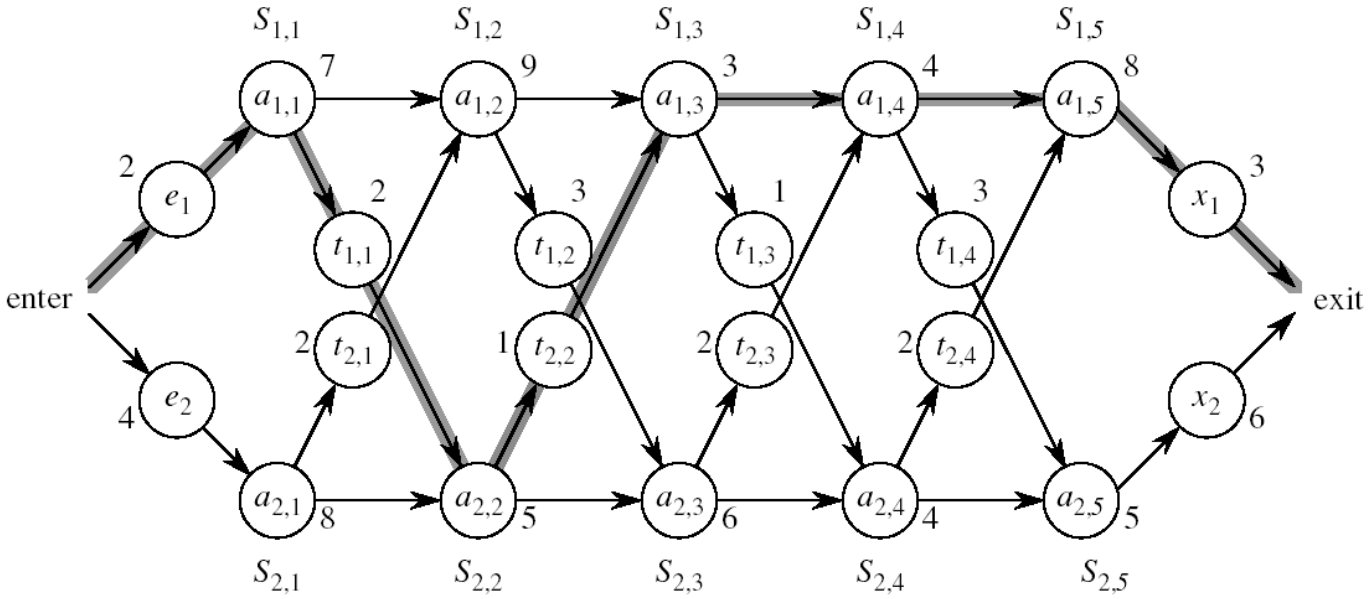
- Define the value of an optimal solution in terms of the optimal solution to subproblems



2. A Recursive Solution (cont.)

- Definitions:
  - f^* : the fastest time to get through the entire factory
  - $f_i[j]$: the fastest time to get from the starting point through station $S_{i,j}$

$$f^* = \min (f_1[n] + x_1, f_2[n] + x_2)$$

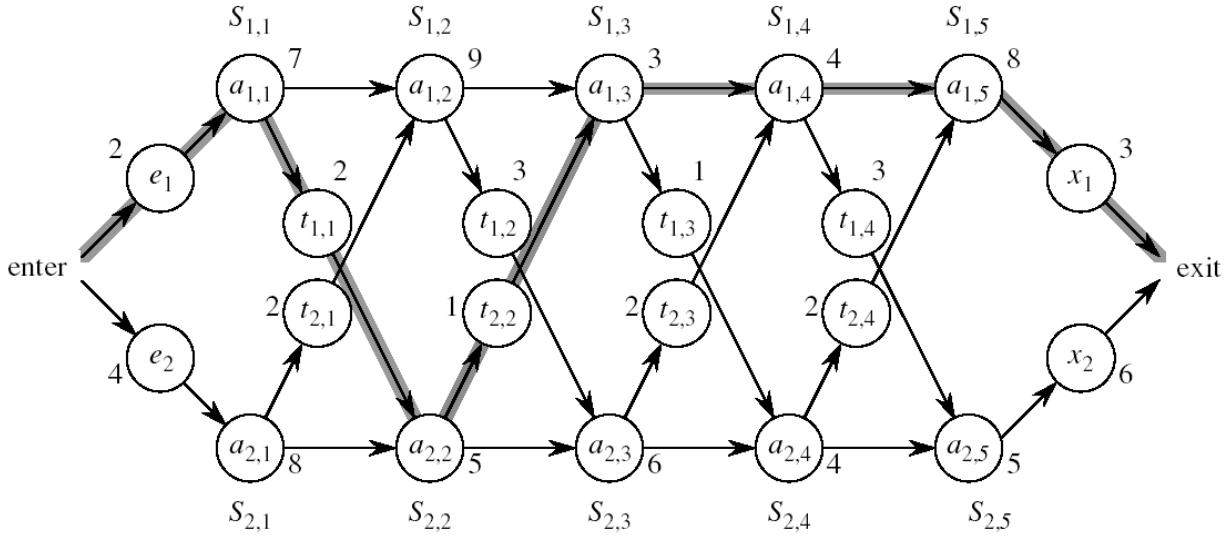


2. A Recursive Solution (cont.)

- Base case: j = 1, i=1,2 (getting through station 1)

  $f_1[1] = e_1 + a_{1,1}$

  $f_2[1] = e_2 + a_{2,1}$



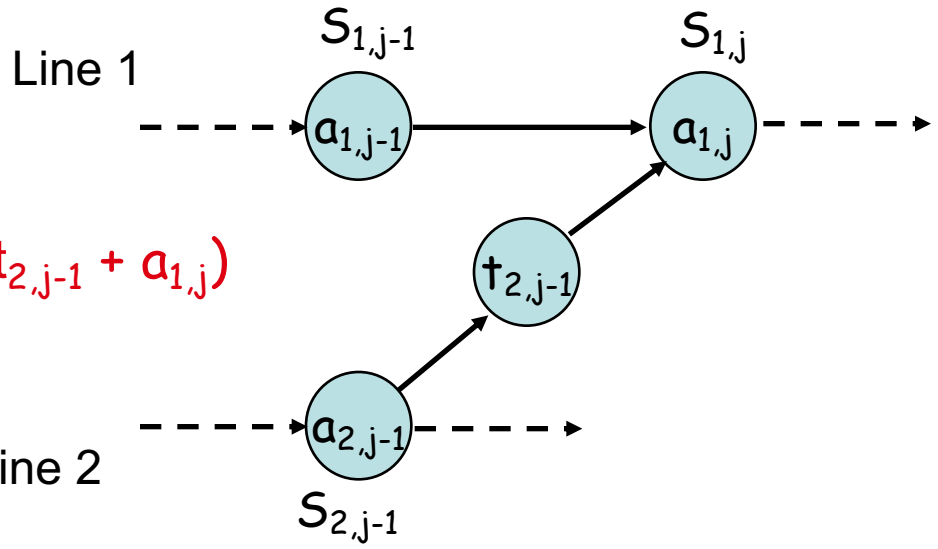
2. A Recursive Solution (cont.)

- General Case: j = 2, 3, …,n, and i = 1, 2
- Fastest way through $S_{1,j}$ is either:
  - the way through $S_{1,j-1}$ then directly through $S_{1,j}$, or

    $f_1[j-1] + a_{1,j}$
  - the way through $S_{2,j-1}$, transfer from line 2 to line 1, then through $S_{1,j}$

    $f_2[j-1] + t_{2,j-1} + a_{1,j}$

$$f_1[j] = \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j})$$



2. A Recursive Solution (cont.)

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

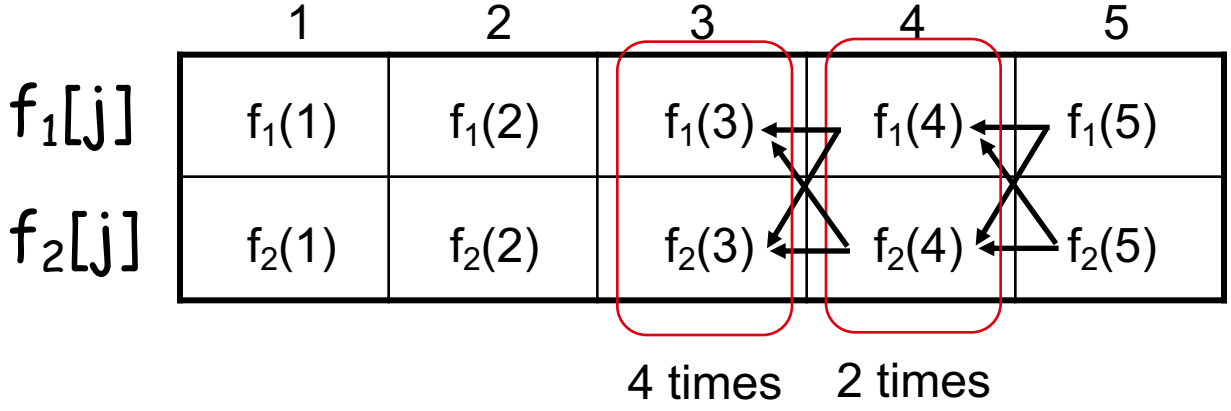
$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

3. Computing the Optimal Solution

$$f^* = \min(f_1[n] + x_1,\ f_2[n] + x_2)$$

$$f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j})$$

$$f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$$



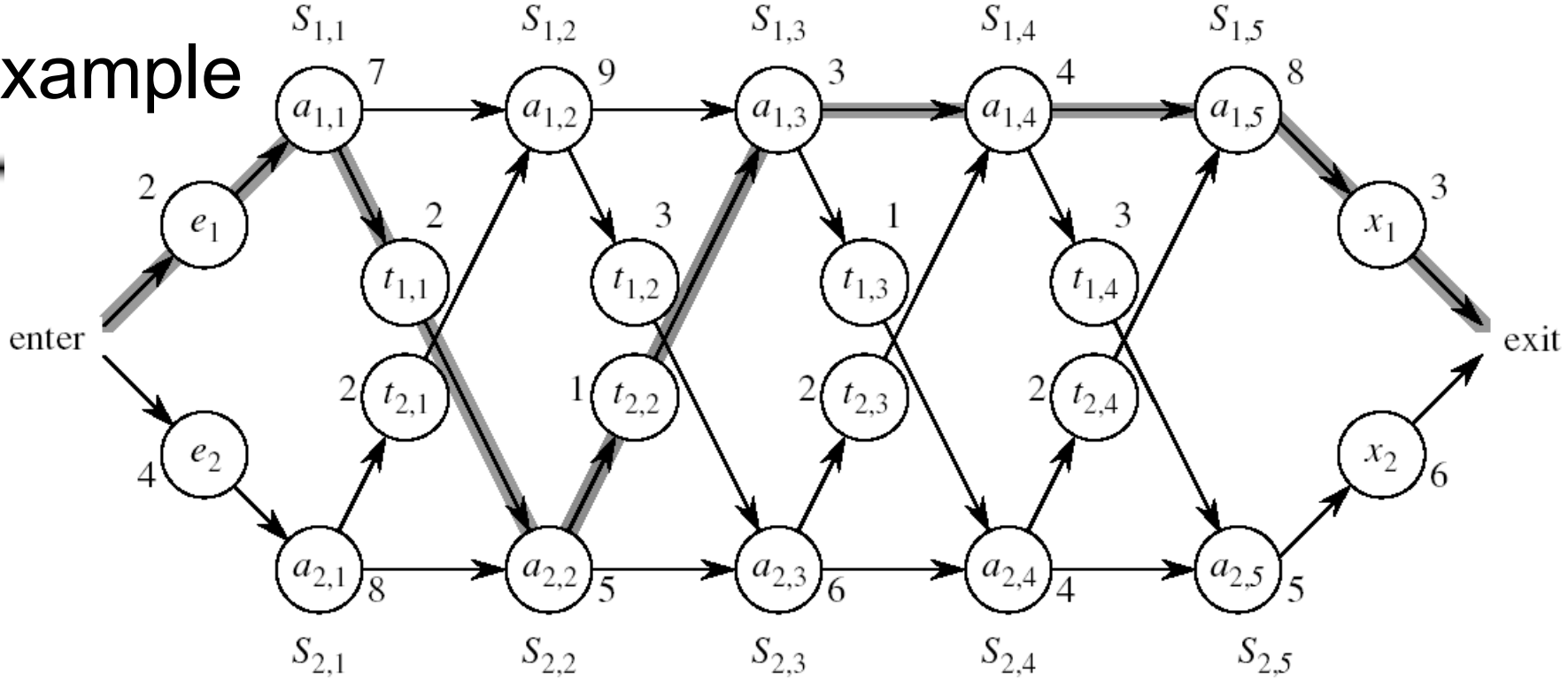
- Solving using simple top-down would result in exponential running time

3. Computing the Optimal Solution

- For $j \geq 2$, each value $f_i[j]$ depends only on the values of $f_1[j-1]$ and $f_2[j-1]$
- Idea: compute the values of $f_i[j]$ as follows:

- Bottom-up approach
  - First find optimal solutions to subproblems
  - Find an optimal solution to the problem from the subproblems

Example



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

	1	2	3	4	5
$f_1[j]$	9	$18^{[1]}$	$20^{[2]}$	$24^{[1]}$	$32^{[1]}$
$f_2[j]$	12	$16^{[1]}$	$22^{[2]}$	$25^{[1]}$	$30^{[2]}$

$f^* = 35^{[1]}$

FASTEST-WAY(a, t, e, x, n)

1. $f_1[1] \leftarrow e_1 + a_{1,1}$
2. $f_2[1] \leftarrow e_2 + a_{2,1}$

} Compute initial values of f_1 and f_2

3. **for** $j \leftarrow 2$ **to** n

O(N)

4. **do if** $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$
5. **then** $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$
6. $l_1[j] \leftarrow 1$
7. **else** $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$
8. $l_1[j] \leftarrow 2$

} Compute the values of $f_1[j]$ and $l_1[j]$

9. **if** $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$
10. **then** $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$
11. $l_2[j] \leftarrow 2$
12. **else** $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$
13. $l_2[j] \leftarrow 1$

} Compute the values of $f_2[j]$ and $l_2[j]$

FASTEST-WAY(a, t, e, x, n) (cont.)

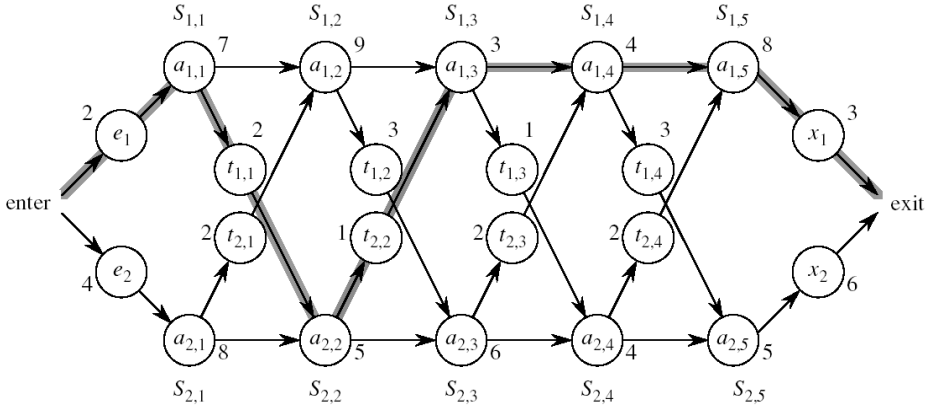
```
14.  if f1[n] + x1 ≤ f2[n] + x2
15.     then f* = f1[n] + x1
16.          l* = 1
17.     else f* = f2[n] + x2
18.          l* = 2
```

Compute the values of the fastest time through the entire factory

4. Construct an Optimal Solution

Alg.: PRINT-STATIONS(l, n)

```
i ← l*
print "line " i ", station " n
for j ← n downto 2
    do i ← l_i[j]
     print "line " i ", station " j - 1
```



	1	2	3	4	5
$f_1[j]/l_1[j]$	9	18[1]	20[2]	24[1]	32[1]
$f_2[j]/l_2[j]$	12	16[1]	22[2]	25[1]	30[2]

l* = 1

Matrix-Chain Multiplication

Problem: given a sequence $\langle A_1, A_2, \ldots, A_n \rangle$, compute the product:

$$A_1 \cdot A_2 \cdots A_n$$

- Matrix compatibility:

$C = A \cdot B$

$col_A = row_B$

$row_C = row_A$

$col_C = col_B$

$C = A_1 \cdot A_2 \cdots A_i \cdot A_{i+1} \cdots A_n$

$col_i = row_{i+1}$

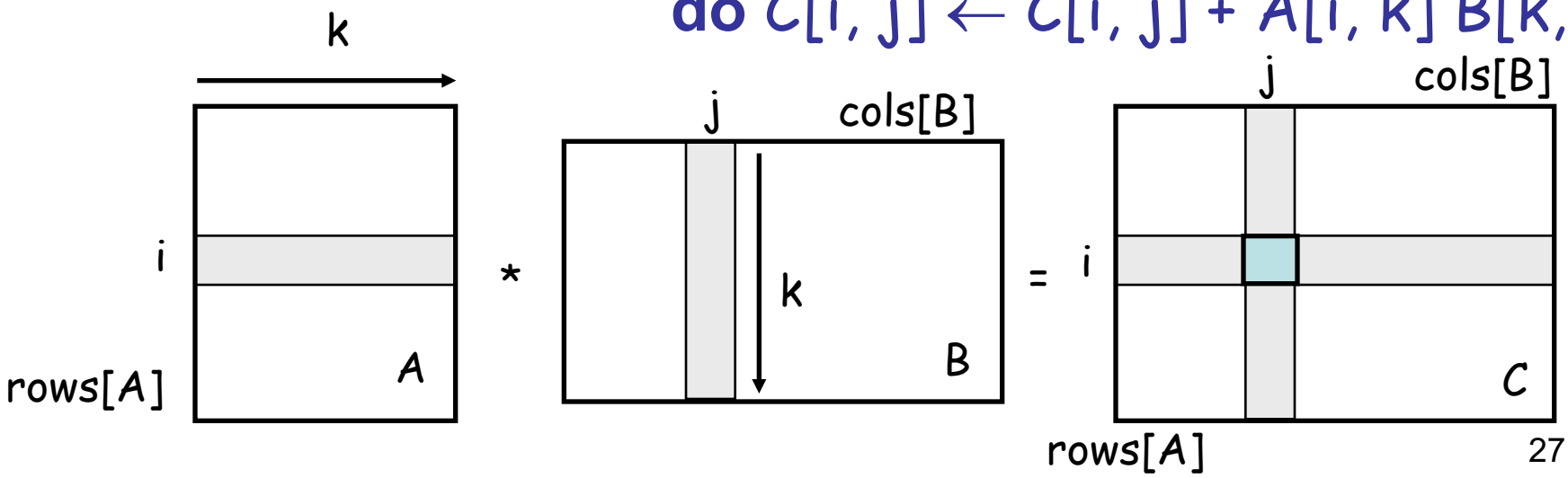
$row_C = row_{A1}$

$col_C = col_{An}$

MATRIX-MULTIPLY(A, B)

```
if columns[A] ≠ rows[B]
   then error "incompatible dimensions"
   else for i ← 1 to rows[A]
            do for j ← 1 to columns[B]
                   do C[i, j] = 0
                      for k ← 1 to columns[A]
                          do C[i, j] ← C[i, j] + A[i, k] B[k, j]
```

rows[A] · cols[A] · cols[B] multiplications



Matrix-Chain Multiplication

- In what order should we multiply the matrices?

$$A_1 \cdot A_2 \cdots A_n$$

- Parenthesize the product to get the order in which matrices are multiplied
- *E.g.:* $A_1 \cdot A_2 \cdot A_3 = ((A_1 \cdot A_2) \cdot A_3)$

$$= (A_1 \cdot (A_2 \cdot A_3))$$

- Which one of these orderings should we choose?
  - The order in which we multiply the matrices has a significant impact on the cost of evaluating the product

Example

$A_1 \cdot A_2 \cdot A_3$

- A_1: 10 x 100
- A_2: 100 x 5
- A_3: 5 x 50

1. $((A_1 \cdot A_2) \cdot A_3)$: $A_1 \cdot A_2 = 10 \times 100 \times 5 = 5{,}000$ (10 x 5)

   $((A_1 \cdot A_2) \cdot A_3) = 10 \times 5 \times 50 = 2{,}500$

   Total: 7,500 scalar multiplications

2. $(A_1 \cdot (A_2 \cdot A_3))$: $A_2 \cdot A_3 = 100 \times 5 \times 50 = 25{,}000$ (100 x 50)

   $(A_1 \cdot (A_2 \cdot A_3)) = 10 \times 100 \times 50 = 50{,}000$

   Total: 75,000 scalar multiplications

one order of magnitude difference!!

Matrix-Chain Multiplication: Problem Statement

- Given a chain of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, where A_i has dimensions $p_{i-1} \times p_i$, fully parenthesize the product $A_1 \cdot A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

$$\underset{p_0 \times p_1}{A_1} \cdot \underset{p_1 \times p_2}{A_2} \cdots \underset{p_{i-1} \times p_i}{A_i} \cdot \underset{p_i \times p_{i+1}}{A_{i+1}} \cdots \underset{p_{n-1} \times p_n}{A_n}$$

What is the number of possible parenthesizations?

- Exhaustively checking all possible parenthesizations is not efficient!
- It can be shown that the number of parenthesizations grows exponentially

1. The Structure of an Optimal Parenthesization

- Notation:

$$A_{i\ldots j} = A_i A_{i+1} \cdots A_j,\ i \le j$$

- Suppose that an optimal parenthesization of $A_{i\ldots j}$ splits the product between A_k and A_{k+1}, where $i \le k < j$

$$\begin{aligned} A_{i\ldots j} &= A_i A_{i+1} \cdots A_j \\ &= A_i A_{i+1} \cdots A_k A_{k+1} \cdots A_j \\ &= A_{i\ldots k} A_{k+1\ldots j} \end{aligned}$$

Optimal Substructure

$$A_{i\ldots j} = A_{i\ldots k}\, A_{k+1\ldots j}$$

- The parenthesization of the "prefix" $A_{i\ldots k}$ must be an optimal parentesization
- If there were a less costly way to parenthesize $A_{i\ldots k}$, we could substitute that one in the parenthesization of $A_{i\ldots j}$ and produce a parenthesization with a lower cost than the optimum $\Rightarrow$ contradiction!
- An optimal solution to an instance of the matrix-chain multiplication contains within it optimal solutions to subproblems

2. A Recursive Solution

- Subproblem:

  determine the minimum cost of parenthesizing

  $$A_{i \cdots j} = A_i A_{i+1} \cdots A_j \qquad \text{for } 1 \le i \le j \le n$$

- Let $m[i, j]$ = the minimum number of multiplications needed to compute $A_{i \cdots j}$
  - full problem ($A_{1..n}$): $m[1, n]$
  - $i = j$: $A_{i \cdots i} = A_i \Rightarrow m[i, i] = 0$, for $i = 1, 2, \ldots, n$

2. A Recursive Solution

- Consider the subproblem of parenthesizing

$$A_{i\ldots j} = A_i\, A_{i+1} \cdots A_j \qquad \text{for } 1 \le i \le j \le n$$

$$= (A_{i\ldots k})(A_{k+1\ldots j}) \qquad \text{for } i \le k < j$$

$p_{i-1}p_k p_j$

m[i, k] m[k+1,j]

- Assume that the optimal parenthesization splits the product $A_i\, A_{i+1} \cdots A_j$ at k $(i \le k < j)$

$$m[i, j] = \underbrace{m[i, k]}_{\text{min \# of multiplications to compute } A_{i\ldots k}} + \underbrace{m[k+1, j]}_{\text{min \# of multiplications to compute } A_{k+1\ldots j}} + \underbrace{p_{i-1}p_k p_j}_{\text{\# of multiplications to compute } A_{i\ldots k}A_{k\ldots j}}$$

2. A Recursive Solution (cont.)

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$$

- We do not know the value of k
  - There are j – i possible values for k: k = i, i+1, …, j-1
- Minimizing the cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes:

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

3. Computing the Optimal Costs

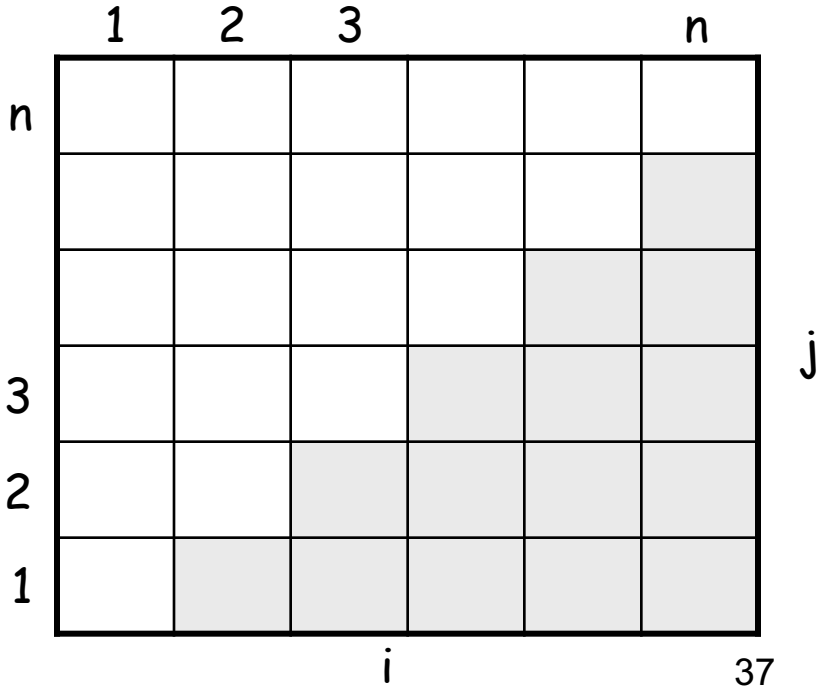
$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases}$$

- Computing the optimal solution recursively takes exponential time!
- How many subproblems?

  $\Rightarrow \Theta(n^2)$

  – Parenthesize $A_{i \ldots j}$ for $1 \le i \le j \le n$

  – One problem for each choice of i and j



3. Computing the Optimal Costs (cont.)

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

- How do we fill in the tables m[1..n, 1..n]?
  - Determine which entries of the table are used in computing m[i, j]

$$A_{i \ldots j} = A_{i \ldots k} \, A_{k+1 \ldots j}$$

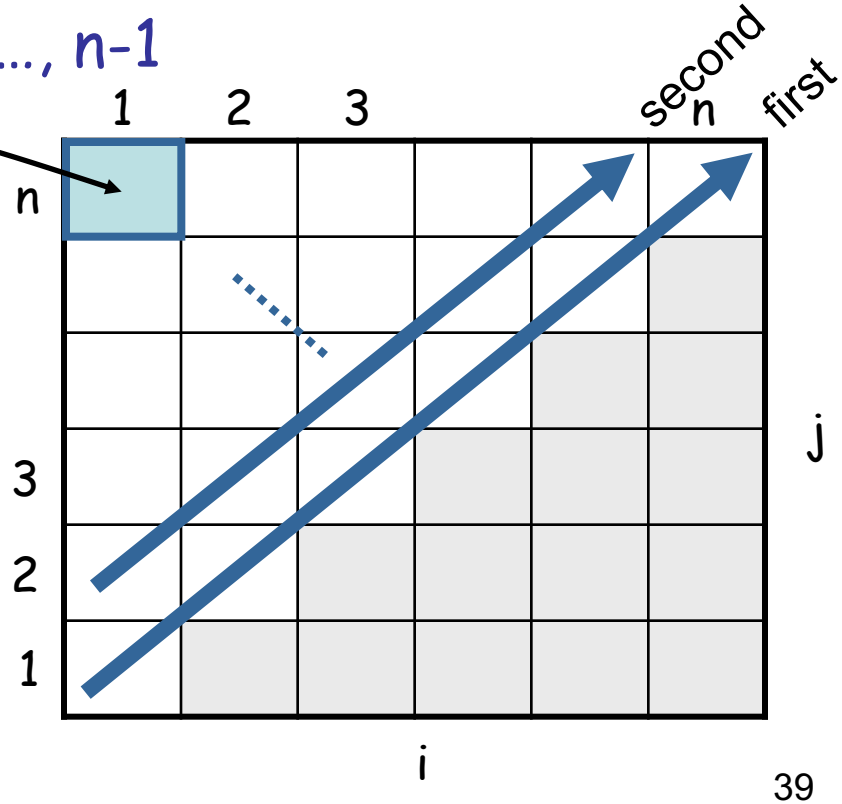
  - Subproblems' size is one less than the original size
  - **Idea:** fill in m such that it corresponds to solving problems of increasing length

3. Computing the Optimal Costs (cont.)

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

- Length = 1: $i = j$, $i = 1, 2, \ldots, n$
- Length = 2: $j = i + 1$, $i = 1, 2, \ldots, n-1$

$m[1, n]$ gives the optimal solution to the problem

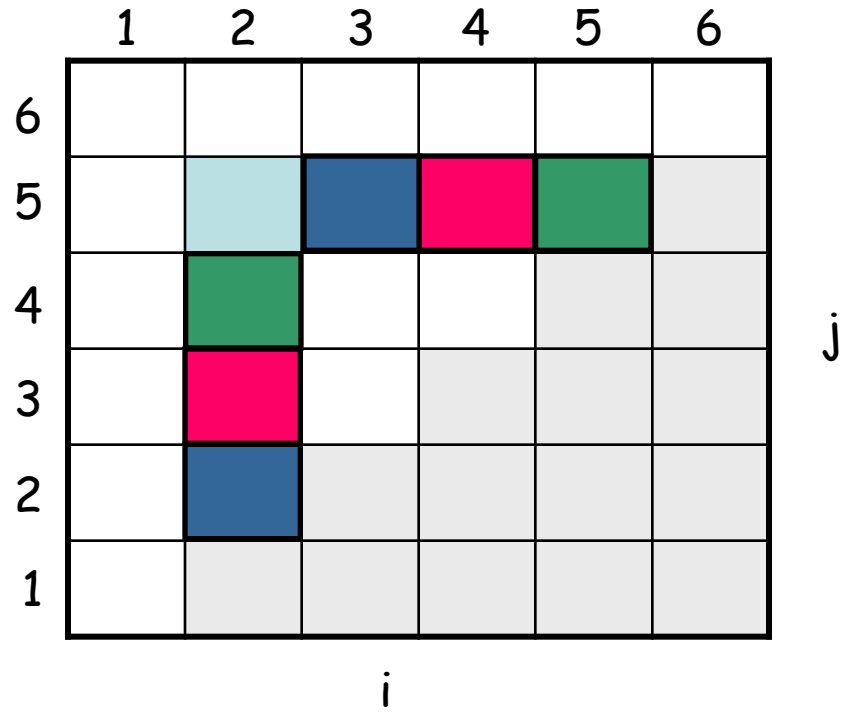


Compute rows from bottom to top and from left to right

Example: $\min \{m[i, k] + m[k+1, j] + p_{i-1}p_k p_j\}$ q1

q1

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p_1p_2p_5 & k = 2 \\ m[2, 3] + m[4, 5] + p_1p_3p_5 & k = 3 \\ m[2, 4] + m[5, 5] + p_1p_4p_5 & k = 4 \end{cases}$$



- Values m[i, j] depend only on values that have been previously computed

Example $\min \{m[i, k] + m[k+1, j] + p_{i-1}p_k p_j\}$

Compute $A_1 \cdot A_2 \cdot A_3$

- A_1: 10 x 100 (p_0 x p_1)
- A_2: 100 x 5 (p_1 x p_2)
- A_3: 5 x 50 (p_2 x p_3)

	1	2	3
3	2 7500	2 25000	0
2	1 5000	0	
1	0		

$m[i, i] = 0$ for $i = 1, 2, 3$

$m[1, 2] = m[1, 1] + m[2, 2] + p_0p_1p_2$ (A_1A_2)

$= 0 + 0 + 10 * 100 * 5 = 5{,}000$

$m[2, 3] = m[2, 2] + m[3, 3] + p_1p_2p_3$ (A_2A_3)

$= 0 + 0 + 100 * 5 * 50 = 25{,}000$

$$m[1, 3] = \min \begin{cases} m[1, 1] + m[2, 3] + p_0p_1p_3 = 75{,}000 & (A_1(A_2A_3)) \\ m[1, 2] + m[3, 3] + p_0p_2p_3 = 7{,}500 & ((A_1A_2)A_3) \end{cases}$$

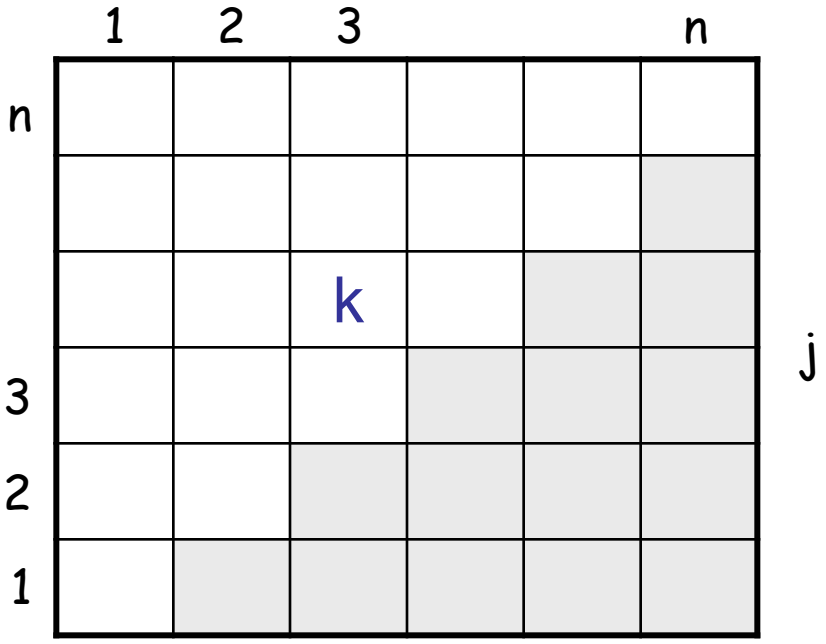
Matrix-Chain-Order(p)

```
MATRIX-CHAIN-ORDER(p)
 1  n ← length[p] − 1
 2  for i ← 1 to n
 3       do m[i, i] ← 0
 4  for l ← 2 to n                ▷ l is the chain length.
 5       do for i ← 1 to n − l + 1
 6            do j ← i + l − 1
 7               m[i, j] ← ∞
 8               for k ← i to j − 1
 9                    do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                       if q < m[i, j]
11                         then m[i, j] ← q
12                              s[i, j] ← k
13  return m and s
```

$O(N^3)$

4. Construct the Optimal Solution

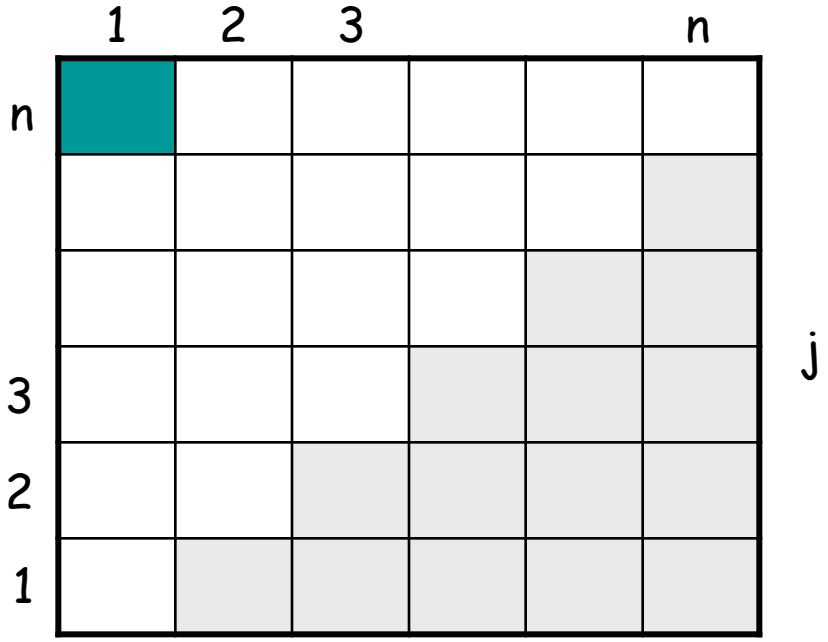
- In a similar matrix s we keep the optimal values of k
- $s[i, j]$ = a value of k such that an optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1}



4. Construct the Optimal Solution

- $s[1, n]$ is associated with the entire product $A_{1..n}$
  - The final matrix multiplication will be split at k = $s[1, n]$

    $A_{1..n} = A_{1..s[1, n]} \cdot A_{s[1, n]+1..n}$
  - For each subproduct recursively find the corresponding value of k that results in an optimal parenthesization



4. Construct the Optimal Solution

- $s[i, j]$ = value of k such that the optimal parenthesization of $A_i\ A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}

j \ i	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

- $s[1, n] = 3 \Rightarrow A_{1..6} = A_{1..3}\ A_{4..6}$
- $s[1, 3] = 1 \Rightarrow A_{1..3} = A_{1..1}\ A_{2..3}$
- $s[4, 6] = 5 \Rightarrow A_{4..6} = A_{4..5}\ A_{6..6}$

4. Construct the Optimal Solution (cont.)

```
PRINT-OPT-PARENS(s, i, j)
if i = j
   then print "A"_i
   else print "("
        PRINT-OPT-PARENS(s, i, s[i, j])
        PRINT-OPT-PARENS(s, s[i, j] + 1, j)
        print ")"
```

j \ i	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

Example: $A_1 \cdot \cdot \cdot A_6$ ((A_1 (A_2 A_3)) ((A_4 A_5) A_6))

```
PRINT-OPT-PARENS(s, i, j)
if i = j
   then print "A"i
   else print "("
        PRINT-OPT-PARENS(s, i, s[i, j])
        PRINT-OPT-PARENS(s, s[i, j] + 1, j)
        print ")"
```

s[1..6, 1..6]

	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

j

i

P-O-P(s, 1, 6) s[1, 6] = 3

i = 1, j = 6 "(" P-O-P (s, 1, 3) s[1, 3] = 1

i = 1, j = 3 "(" P-O-P(s, 1, 1) $\Rightarrow$ "A_1"

P-O-P(s, 2, 3) s[2, 3] = 2

i = 2, j = 3 "(" P-O-P (s, 2, 2) $\Rightarrow$ "A_2"

P-O-P (s, 3, 3) $\Rightarrow$ "A_3"

")"

")" ...

Memoization

- Top-down approach with the efficiency of typical dynamic programming approach
- Maintaining an entry in a table for the solution to each subproblem
  - **memoize** the inefficient recursive algorithm
- When a subproblem is first encountered its solution is computed and stored in that table
- Subsequent “calls” to the subproblem simply look up that value

Memoized Matrix-Chain

Alg.: MEMOIZED-MATRIX-CHAIN(p)

```
1.  n ← length[p] - 1
2.  for i ← 1 to n
3.      do for j ← i to n
4.          do m[i, j] ← ∞
5.  return LOOKUP-CHAIN(p, 1, n)
```

Initialize the m table with large values that indicate whether the values of m[i, j] have been computed

Top-down approach

Memoized Matrix-Chain

Alg.: LOOKUP-CHAIN(p, i, j)

Running time is $O(n^3)$

```
1.  if m[i, j] < ∞
2.      then return m[i, j]
3.  if i = j
4.     then m[i, j] ← 0
5.     else for k ← i to j – 1
6.              do q ← LOOKUP-CHAIN(p, i, k) +
                        LOOKUP-CHAIN(p, k+1, j) + p_{i-1} p_k p_j
7.                 if q < m[i, j]
8.                    then m[i, j] ← q
9.  return m[i, j]
```

Dynamic Progamming vs. Memoization

- Advantages of dynamic programming vs. memoized algorithms
  - No overhead for recursion, less overhead for maintaining the table
  - The regular pattern of table accesses may be used to reduce time or space requirements
- Advantages of memoized algorithms vs. dynamic programming
  - Some subproblems do not need to be solved

Rod Cutting Problem

- We are given prices p_i and rods of length i:

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

- **Question:** We are given a rod of length n, and want to maximize revenue, by cutting up the rod into pieces and selling each of the pieces.

- **Example:** we are given a 4 inches rod. Best solution to cut up? We'll first list the solutions:

- We'll first list all possible solutions:

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

1.) Cut into 2 pieces of length 2: $p_2 + p_2 = 5+5=10$

2.) Cut into 4 pieces of length 1: $p_1+p_1+p_1+p_1=1+1+1+1=4$

3-4.) Cut into 2 pieces of length 1 and 3 (3 and 1): $p_1+p_3=1+8=9$

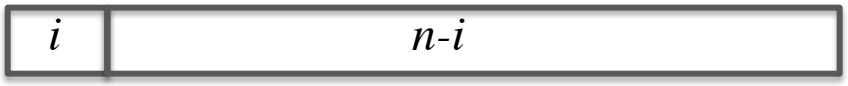
$p_3+p_1=8+1=9$

5.) Keep length 4: $p_4=9$

6-8.) Cut into 3 pieces, length 1, 1 and 2 (and all the different orders)

- $p_1+p_1+p_2=7$ $p_1+p_2+p_1=7$ $p_2+p_1+p_1=7$
- **Total:** 8 cases for $n = 4$ (= 2^{n-1}). We can slightly reduce by always requiring cuts in non-decreasing order. But still a lot!

- **Note:** We've computed a brute force solution; all possibilities for this simple small example. But we want more efficient solution!
- One solution:

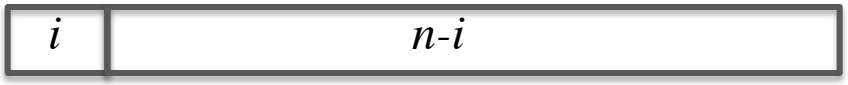


- What are we doing?
  - Cut rod into length i and n-i
  - Only remainder n-i can be further cut (recursed)

- We need to define:

a.) **Maximum revenue** for log of size *n*: $\boldsymbol{r_n}$ (that is the solution we want to find).

b.) **Revenue (price)** for single log of length *i*: $\boldsymbol{p_i}$

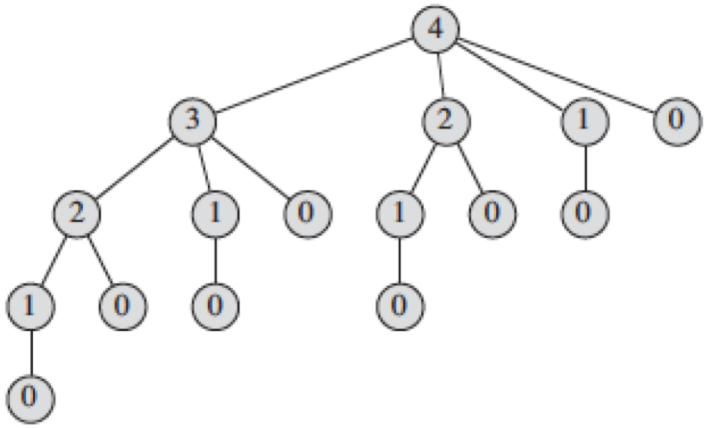


r_n: $p_i + r_{n-i}$

$$r_n = \max \begin{Bmatrix} p_1 + r_{n-1} \\ p_2 + r_{n-2} \\ \cdots \\ p_n + r_0 \end{Bmatrix}$$

```
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
```

- Problem? Slow runtime (it's essential brute force)!
- But why? Cut-rod calls itself repeatedly with the same parameter values (tree):



Bottom-up solution: **Bottom Up Cut-Rod**

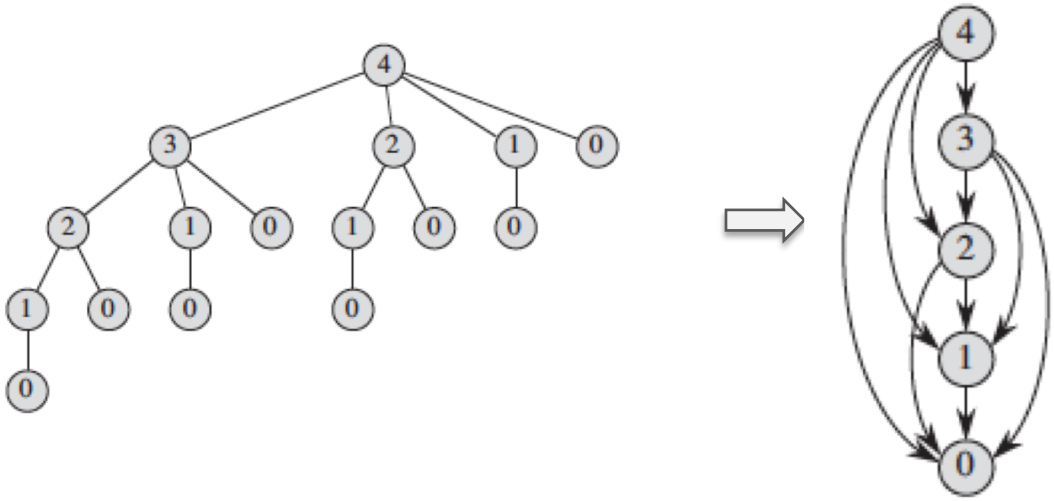
```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

For saving solution of subproblem

Compute maximum revenue for r[j].

Saving value

Dependency Tree



- each vertex represents subproblem of a given size
- Vertex label: subproblem size
- Edges from *x* to *y*: We need a solution for subproblem *x* when solving subproblem *y*

Memoized Approach

- **Step 1** Initialization

```
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)
```

```
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
```

Longest Common Subsequence

- Given two sequences

  $X = \langle x_1, x_2, \ldots, x_m \rangle$

  $Y = \langle y_1, y_2, \ldots, y_n \rangle$

  find a maximum length common subsequence (LCS) of X and Y

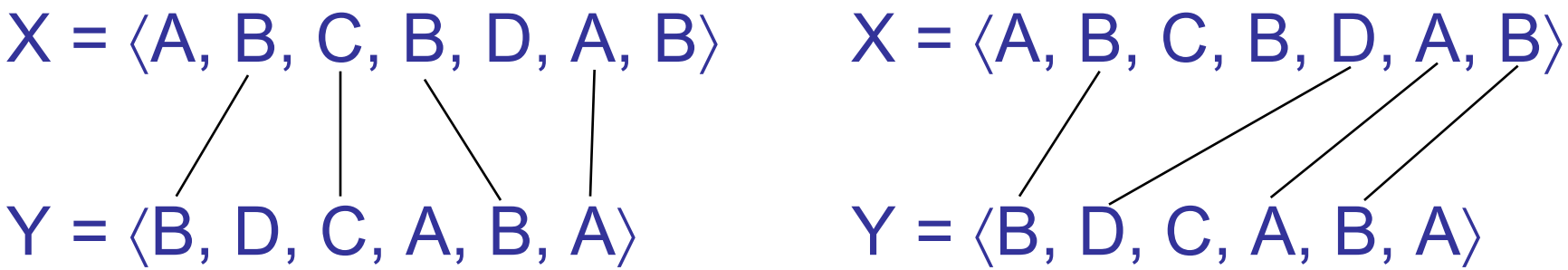
- *E.g.:*

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

- Subsequences of X:
  - A subset of elements in the sequence taken in order

    $\langle A, B, D \rangle$, $\langle B, C, D, B \rangle$, etc.

Example



- ⟨B, C, B, A⟩ and ⟨B, D, A, B⟩ are longest common subsequences of X and Y (length = 4)

- ⟨B, C, A⟩, however is not a LCS of X and Y

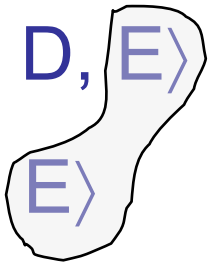
Brute-Force Solution

- For every subsequence of X, check whether it's a subsequence of Y
- There are 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
- Running time: $\Theta(n2^m)$

Making the choice

$X = \langle A, B, D, E \rangle$

$Y = \langle Z, B, E \rangle$



- Choice: include one element into the common sequence (E) and solve the resulting subproblem

  $X = \langle A, B, D, G \rangle$

  $Y = \langle Z, B, D \rangle$

- Choice: exclude an element from a string and solve the resulting subproblem

Notations

- Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ we define the i-th prefix of X, for i = 0, 1, 2, …, m

  $X_i = \langle x_1, x_2, \ldots, x_i \rangle$

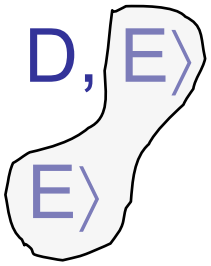
- $c[i, j]$ = the length of a LCS of the sequences $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ and $Y_j = \langle y_1, y_2, \ldots, y_j \rangle$

A Recursive Solution

Case 1: $x_i = y_j$

e.g.: $X_i = \langle A, B, D, E \rangle$

$Y_j = \langle Z, B, E \rangle$



$$c[i, j] = c[i - 1, j - 1] + 1$$

- Append $x_i = y_j$ to the LCS of X_{i-1} and Y_{j-1}
- Must find a LCS of X_{i-1} and $Y_{j-1} \Rightarrow$ optimal solution to a problem includes optimal solutions to subproblems

A Recursive Solution

Case 2: $x_i \neq y_j$

e.g.: $X_i = \langle A, B, D, G \rangle$

$Y_j = \langle Z, B, D \rangle$

$$c[i, j] = \max \{ c[i - 1, j], c[i, j-1] \}$$

- Must solve two problems
  - find a LCS of X_{i-1} and Y_j: $X_{i-1} = \langle A, B, D \rangle$ and $Y_j = \langle Z, B, D \rangle$
  - find a LCS of X_i and Y_{j-1}: $X_i = \langle A, B, D, G \rangle$ and $Y_j = \langle Z, B \rangle$

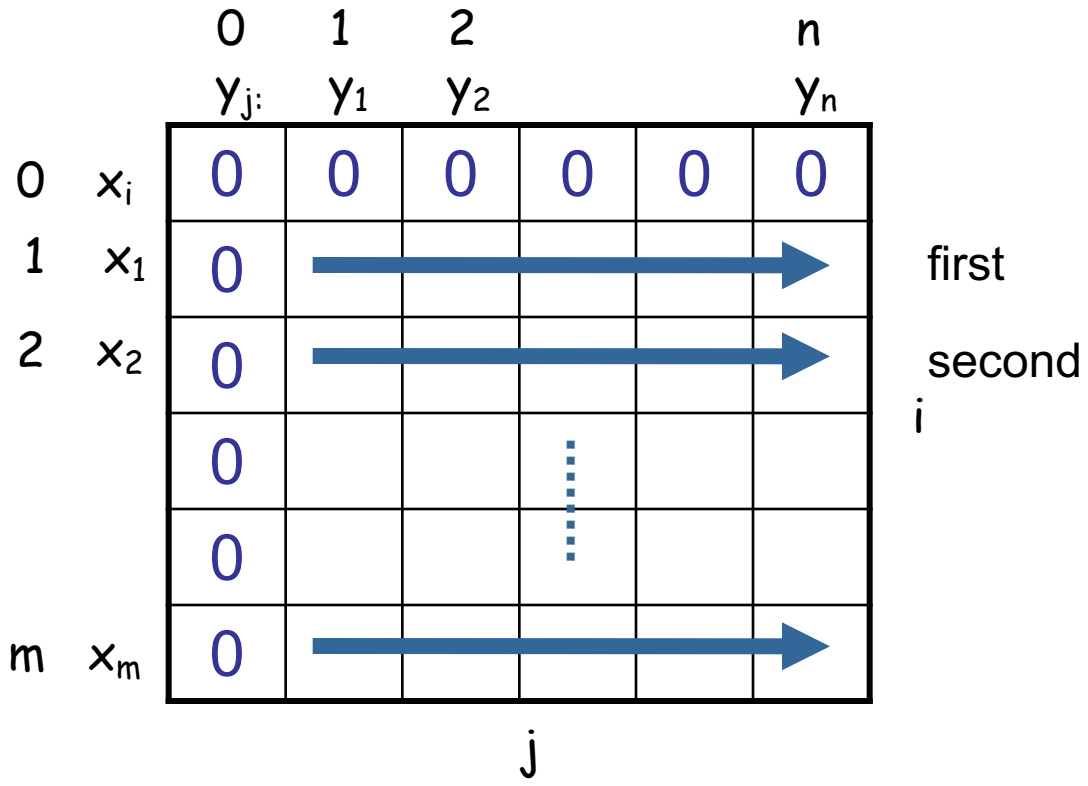
- Optimal solution to a problem includes optimal solutions to subproblems

Overlapping Subproblems

- To find a LCS of X and Y
  - we may need to find the LCS between X and Y_{n-1} and that of X_{m-1} and Y
  - Both the above subproblems has the subproblem of finding the LCS of X_{m-1} and Y_{n-1}
- Subproblems share subsubproblems

3. Computing the Length of the LCS

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



Additional Information

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

b & c:

		0	1	2	3		n
		y_j:	A	C	D		F
0	x_i	0	0	0	0	0	0
1	A	0					
2	B	0			c[i-1,j]		
3	C	0		c[i,j-1]			
		0					
m	D	0					

i

j

A matrix b[i, j]:

- For a subproblem [i, j] it tells us what choice was made to obtain the optimal value
- If $x_i = y_j$

  b[i, j] = "↖"
- Else, if $c[i-1, j] \geq c[i, j-1]$

  b[i, j] = "↑"

  else

  b[i, j] = "←"

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

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

The length of the LCS if one of the sequences is empty is zero

Case 1: $x_i = y_j$

Case 2: $x_i \neq y_j$

Running time: $\Theta(mn)$

Example

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

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

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

If $x_i = y_j$

$b[i, j] =$ "↖"

Else if

$c[i - 1, j] \geq c[i, j-1]$

$b[i, j] =$ "↑"

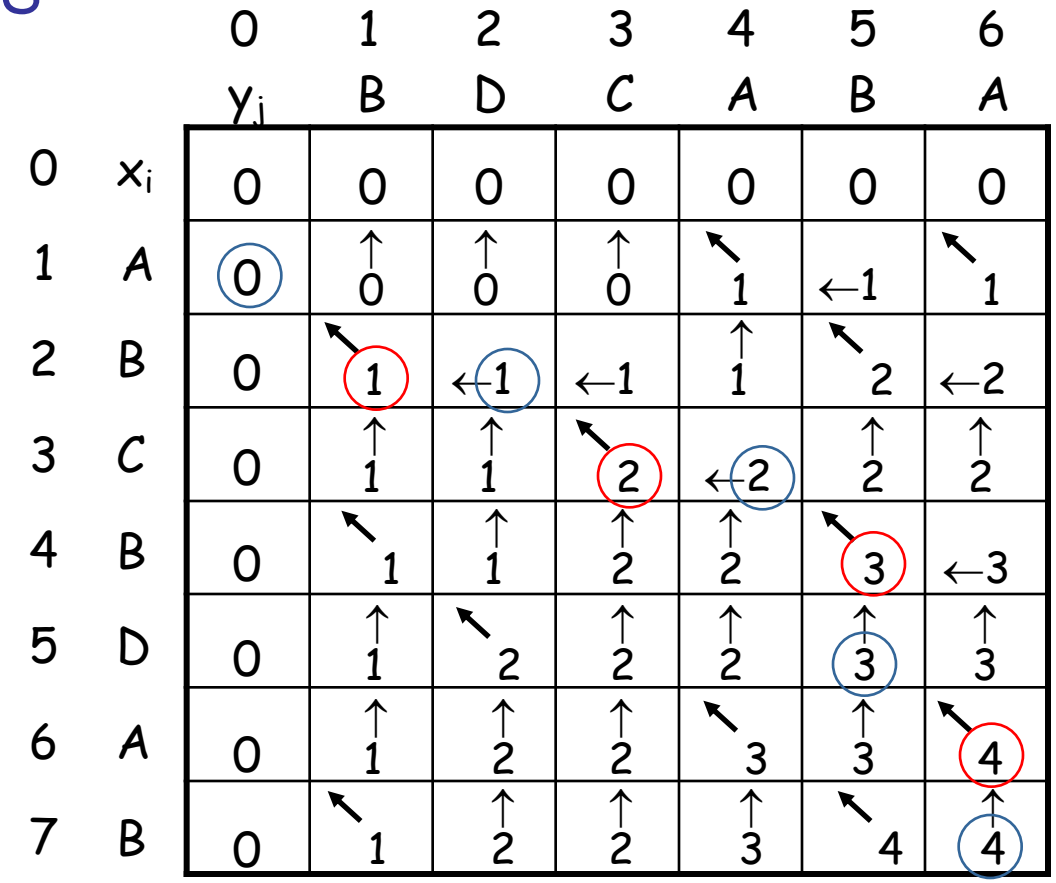
else

$b[i, j] =$ "←"

		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

4. Constructing a LCS

- Start at b[m, n] and follow the arrows
- When we encounter a "↖" in b[i, j] $\Rightarrow x_i = y_j$ is an element of the LCS



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

```
1.  if i = 0 or j = 0                          Running time: Θ(m + n)
2.    then return
3.  if b[i, j] = "↖"
4.     then PRINT-LCS(b, X, i - 1, j - 1)
5.          print x_i
6.  elseif b[i, j] = "↑"
7.          then PRINT-LCS(b, X, i - 1, j)
8.          else PRINT-LCS(b, X, i, j - 1)
```

Initial call: PRINT-LCS(b, X, length[X], length[Y])

Memoized Approach

```
LCS(X,Y,i,j,C) {
if (i == 0 || j == 0)
        return 0;
if (C[i][j] != -∝)
        return C[i][j];
if (X[i] == Y[j])
        C[i][j] = 1 + LCS(X, Y, i - 1, j - 1, C);
else
        C[i][j] = max(LCS(X, Y, i, j - 1, C),LCS(X, Y, i - 1, j, C));
return C[i][j];
}
```

Memoized Approach

```
LCS(X,Y,I,j,C) {
if (i == 0 || j == 0)
        return 0;
if (C[i][j] != -∝)
        return C[i][j];
if (X[i] == Y[j])
        C[i][j] = 1 + LCS(X, Y, i - 1, j - 1, C);
        B[i][j]='↖';
else     x1=LCS(X, Y, i, j - 1, C); y1= LCS(X, Y, i - 1, j, C);
        C[i][j] = max(x1,y1);
        if  (x1>=y1) B[i][j]='←'; else B[i][j]= '↑';
return C[i][j];
}
```

Optimal Binary Search Trees

- Problem
  - Given sequence $K = k_1 < k_2 < \cdots < k_n$ of n sorted keys, with a search probability p_i for each key k_i.
  - Want to build a binary search tree (BST) with minimum expected search cost.
  - Actual cost = # of items examined.
  - For key k_i, cost = $\text{depth}_T(k_i)+1$, where $\text{depth}_T(k_i)$ = depth of k_i in BST T .

Expected Search Cost

$$E[\text{search cost in } T]$$

$$= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i$$

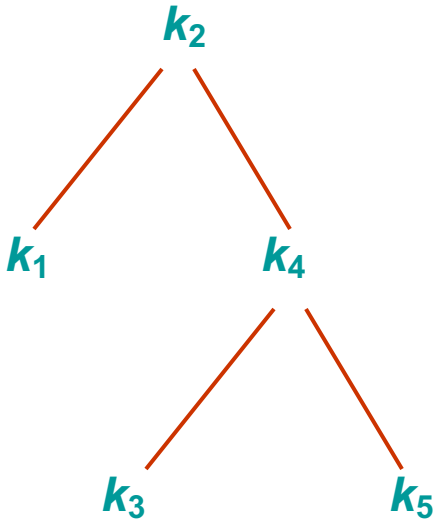
$$= \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i$$

$$= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i \quad (1)$$

Sum of probabilities is 1.

Example

- Consider 5 keys with these search probabilities: $p_1 = 0.25$, $p_2 = 0.2$, $p_3 = 0.05$, $p_4 = 0.2$, $p_5 = 0.3$.

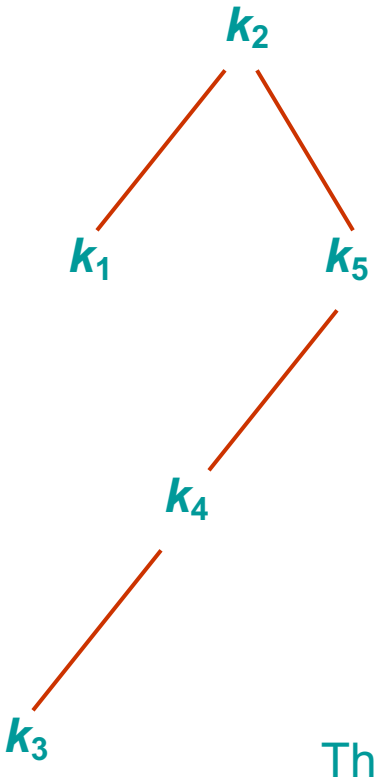


i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	0.25
2	0	0
3	2	0.1
4	1	0.2
5	2	0.6
		1.15

Therefore, E[search cost] = 2.15.

Example

- $p_1 = 0.25$, $p_2 = 0.2$, $p_3 = 0.05$, $p_4 = 0.2$, $p_5 = 0.3$.



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	0.25
2	0	0
3	3	0.15
4	2	0.4
5	1	0.3
		1.10

Therefore, E[search cost] = 2.10.

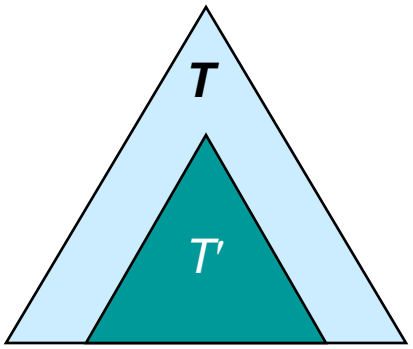
This tree turns out to be optimal for this set of keys.

Example

- **Observations:**
  - Optimal BST may not have smallest height.
  - Optimal BST may not have highest-probability key at root.
- Build by exhaustive checking?
  - Construct each *n*-node BST.
  - For each,
    assign keys and compute expected search cost.
  - But number of different BSTs with *n* nodes is exponential.

Optimal Substructure

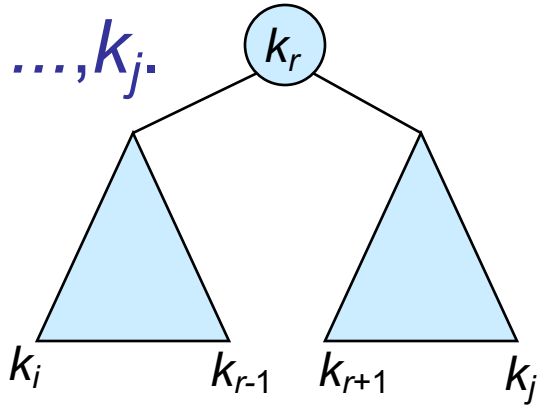
- Any subtree of a BST contains keys in a contiguous range $k_i, \ldots, k_j$ for some $1 \le i \le j \le n$.



- If T is an optimal BST and
  T contains subtree T' with keys $k_i, \ldots, k_j$,
  then T' must be an optimal BST for keys $k_i, \ldots, k_j$.
- **Proof:** Cut and paste.

Optimal Substructure

- One of the keys in $k_i, \ldots, k_j$, say k_r, where $i \le r \le j$,
  must be the root of an optimal subtree for these keys.
- Left subtree of k_r contains $k_i, \ldots, k_{r-1}$.
- Right subtree of k_r contains $k_r+1, \ldots, k_j$.



- To find an optimal BST:
  - Examine all candidate roots k_r , for $i \le r \le j$
  - Determine all optimal BSTs containing $k_i, \ldots, k_{r-1}$ and containing $k_{r+1}, \ldots, k_j$

Recursive Solution

- Find optimal BST for $k_i,\ldots,k_j$, 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 $k_i,\ldots,k_j$.

- If $j = i-1$, then $e[i, j] = 0$.
- If $j \geq i$,
  - Select a root k_r, for some $i \leq r \leq j$.
  - Recursively make an optimal BSTs
    - for $k_i,..,k_{r-1}$ as the left subtree, and
    - for $k_{r+1},..,k_j$ as the right subtree.

Recursive Solution

- When the OPT tree becomes a subtree of a node:
  - Depth of every node in OPT tree goes up by 1.
  - Expected search cost increases by

$$w(i,j) = \sum_{l=i}^{j} p_l$$ from (1)

- If k_r is the root of an optimal BST for $k_i,..,k_j$:
  - $e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j))$

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

(because $w(i, j) = w(i, r-1) + p_r + w(r + 1, j)$)

- But, we don't know k_r. Hence,

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

Computing an Optimal Solution

For each subproblem (i,j), store:

- expected search cost in a table $e[1..n+1, 0..n]$
  - Will use only entries $e[i, j]$, where $j \geq i-1$.
- root$[i, j]$ = root of subtree with keys $k_i,..,k_j$, for $1 \leq i \leq j \leq n$.
- $w[1..n+1, 0..n]$ = sum of probabilities
  - $w[i, i-1] = 0$ for $1 \leq i \leq n$.
  - $w[i, j] = w[i, j\text{-}1] + p_j$ for $1 \leq i \leq j \leq n$.

Pseudo-code

```
OPTIMAL-BST(p, q, n)
1.  for i ← 1 to n + 1
2.      do e[i, i – 1] ← 0
3.         w[i, i – 1] ← 0
4.  for l ← 1 to n
5.      do for i ← 1 to n – l + 1
6.         do j ← i + l – 1
7.            e[i, j ] ← ∞
8.            w[i, j ] ← w[i, j – 1] + p_j
9.            for r ← i to j
10.               do t ← e[i, r – 1] + e[r + 1, j ] + w[i, j
    ]
11.                  if t < e[i, j ]
12.                     then e[i, j ] ← t
13.                          root[i, j ] ← r
14. return e and root
```

Consider all trees with *l* key

Fix the first key.

Fix the last key

Determine the root of the optimal (sub)tree

Time: $O(n^3)$

Number of Subsequences with Even and Odd Sum

- ***Input:*** *arr[] = {1, 2, 2, 3}*
  Output: *EvenSum = 7, OddSum = 8*
- There are 2^N -1 total subsequences
- *Sequences with Even sum*

1. {1, 3} Sum =4
2. {1, 2, 2, 3} Sum =8

3-4. {1, 2, 3} {1, 2, 3} Sum =6

5-6. {2} {2} Sum =2

7. {2, 2} Sum=4

And the rest subsequences are of odd sum

Naïve Approach

- Generate all subsequences and check if the subsequence is odd or even
- Not Efficient!!

Dynamic Programming Approach

- CntEvn[i] → number of subsequences with even sum in the range of (0,i)
- CntOdd[i] → number of subsequences with odd sum in the range of (0,i)

- For odd number in i^{th} position
- CntEvn[i] = CntEvn[i-1] + CntOdd[i-1]
- CntOdd[i] = CntEvn[i-1] + CntOdd[i-1] + 1

- For even number in i^{th} position
- CntEvn[i] = CntEvn[i-1] + CntEvn[i-1] + 1
- CntOdd[i] = CntOdd[i-1] + CntOdd[i-1]

- Base Condition:
- CntEvn[0] = 0
- CntOdd[0] = 0

- For i -> 1 to n
  - if (A[i]%2 == 0)
    - CntEvn[i] = CntEvn[i-1] + CntEvn[i-1] + 1
    - CntOdd[i] = CntOdd[i-1] + CntOdd[i-1]
  - else
    - CntEvn[i] = CntEvn[i-1] + CntOdd[i-1]
    - CntOdd[i] = CntEvn[i-1] + CntOdd[i-1] + 1
- return

- Typedef struct s{
  - int even;
  - Int odd;
  - } X
- X Cnt[n+1]
- For i → 1 to n
  - X[i].even = -∝
  - X[i].odd = -∝
- FindCnt(A, Cnt, n)

- FindCnt(A, Cnt, n)
- X tmp;
- if (Cnt [n].even >= 0)
  - return Cnt[n];
- else
  - tmp= FindCnt(A, Cnt, n-1) ;
  - If (A[n]%2 == 0)
    - Cnt[n].even = 2*tmp.even +1;
    - Cnt[n].odd = 2*tmp.odd;
  - else
    - Cnt[n].even = tmp.even + tmp.odd
    - Cnt[n].odd = tmp.even + tmp.odd +1
  - return Cnt[n];

Knapsack 0-1 Problem

- The goal is to **maximize the value of a knapsack** that can hold at most W units (i.e. lbs or kg) worth of goods from a list of items $I_0, I_1, \ldots I_{n-1}$.

  - Each item has 2 attributes:
    1) Value – let this be v_i for item I_i
    2) Weight – let this be w_i for item I_i

Slides adopted from (http://www.cs.ucf.edu/~dmarino/ucf/cop3503/lectures/)

Knapsack 0-1 Problem

- The difference between this problem and the fractional knapsack one is that you CANNOT take a fraction of an item.
  - You can either take it or not.
  - Hence the name Knapsack 0-1 problem.

Knapsack 0-1 Problem

- Brute Force
  - The naïve way to solve this problem is to cycle through all 2^n subsets of the n items and pick the subset with a legal weight that maximizes the value of the knapsack.
  - We can come up with a dynamic programming algorithm that will USUALLY do better than this brute force technique.

Knapsack 0-1 Problem

- As we did before we are going to solve the problem in terms of sub-problems.
  - So let's try to do that…

- Our first attempt might be to characterize a sub-problem as follows:
  - Let S_k be the optimal subset of elements from $\{I_0, I_1, \ldots, I_k\}$.
    - What we find is that the optimal subset from the elements $\{I_0, I_1, \ldots, I_{k+1}\}$ may not correspond to the optimal subset of elements from $\{I_0, I_1, \ldots, I_k\}$ in any regular pattern.
  - Basically, the solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

Knapsack 0-1 Problem

- Let's illustrate that point with an example:

Item	Weight	Value
I_0	3	10
I_1	8	4
I_2	9	9
I_3	8	11

- **The maximum weight the knapsack can hold is 20.**

- The best set of items from $\{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$
- BUT the best set of items from $\{I_0, I_1, I_2, I_3\}$ is $\{I_0, I_2, I_3\}$.
  - In this example, note that this optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution, $\{I_0, I_1, I_2\}$.
    - (Instead it build's upon the solution, $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$ with weight 12 or less.)

Knapsack 0-1 problem

- So now we must re-work the way we build upon previous sub-problems…
  - Let **B[k, w]** represent the maximum total value of a subset S_k with weight w.
  - Our goal is to find **B[n, W],** where n is the total number of items and W is the maximal weight the knapsack can carry.

- So our recursive formula for subproblems:

$$B[k, w] = B[k-1, w], \text{ if } w_k > w$$
$$= \max\{ B[k-1, w], B[k-1, w-w_k] + v_k \}, \text{ otherwise}$$

- In English, this means that the best subset of S_k that has total weight w is:
  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

Knapsack 0-1 Problem – 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! If it was the total weight would be > w, which is unacceptable.

- **Second case:** $w_k \leq w$
  - Then the item *k* can be in the solution, and we choose the case with greater value.

Knapsack 0-1 Algorithm

```
for w = 0 to W {  // Initialize 1st row to 0's
   B[0,w] = 0
}
for i = 1 to n {  // Initialize 1st column to 0's
   B[i,0] = 0
}
for i = 1 to n {
   for w = 0 to W {
        if w_i <= w {  //item i can be in the solution
               if v_i + B[i-1,w-w_i] > B[i-1,w]
                      B[i,w] = v_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
   }
}
```

Knapsack 0-1 Problem

- Let's run our algorithm on the following data:
  - n = 4 (# of elements)
  - W = 5 (max weight)
  - Elements (weight, value):
    (2,3), (3,4), (4,5), (5,6)

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

```
// Initialize the base cases
for w = 0 to W
        B[0,w] = 0

for i = 1 to n
        B[i,0] = 0
```

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$w = 1$

$w\text{-}w_i = -1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

w = 2

$w\text{-}w_i = 0$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$w = 3$

$w\text{-}w_i = 1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$w = 4$

$w\text{-}w_i = 2$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$w = 5$

$w\text{-}w_i = 3$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
w = 1
$w\text{-}w_i = -2$

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

if $w_i <= w$ //item i can be in the solution
    if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
        $B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$
    else
        $B[i,w] = B[i\text{-}1,w]$
else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$w = 2$

$w\text{-}w_i = -1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else **$B[i,w] = B[i\text{-}1,w]$** // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
w = 3
$w\text{-}w_i = 0$

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

if $w_i <= w$ //item i can be in the solution
    if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
        $B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$
    else
        $B[i,w] = B[i\text{-}1,w]$
else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$w = 4$
$w\text{-}w_i = 1$

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

if $w_i <= w$ //item i can be in the solution
    if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
        $B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$
    else
        $B[i,w] = B[i\text{-}1,w]$
else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$w = 5$

$w\text{-}w_i = 2$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

$i = 3$

$v_i = 5$

$w_i = 4$

w = 1..3

$w\text{-}w_i = -3..-1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

$i = 3$

$v_i = 5$

$w_i = 4$

w = 4

$w\text{-}w_i = 0$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

$i = 3$

$v_i = 5$

$w_i = 4$

w = 5

$w\text{-}w_i = 1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$

else

B[i,w] = B[i-1,w]

else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

$i = 4$

$v_i = 6$

$w_i = 5$

w = 1..4

$w\text{-}w_i = -4..-1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 4$

$v_i = 6$

$w_i = 5$

$w = 5$

$w\text{-}w_i = 0$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	**7**

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

We're DONE!!

The max possible value that can be carried in this knapsack is ***$7***

Knapsack 0-1 Algorithm

- This algorithm only finds the max possible value that can be carried in the knapsack
  - The value in B[n,W]

- To know the ***items*** that make this maximum value, we need to trace back through the table.

Knapsack 0-1 Algorithm Finding the Items

- Let i = n and k = W

  if B[i, k] ≠ B[i-1, k] then

  mark the i^{th} item as in the knapsack

  i = i-1, k = k-w_i

  else

  i = i-1 // Assume the i^{th} item is not in the knapsack

  // Could it be in the optimally packed knapsack?

Knapsack 0-1 Algorithm Finding the Items

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

Knapsack:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i = 4
k = 5
$v_i = 6$
$w_i = 5$
B[i,k] = 7
B[i-1,k] = 7

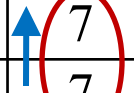
i = n , k = W
while i, k > 0
    if *B[i, k] ≠ B[i-1, k]* then
        mark the i^{th} item as in the knapsack
        i = i-1, k = k-w_i
    else
        i = i-1

Knapsack 0-1 Algorithm Finding the Items

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

Knapsack:

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7



$i = 3$
$k = 5$
$v_i = 5$
$w_i = 4$
B[i,k] = 7
B[i-1,k] = 7

$i = n$, $k = W$
while i, k > 0
    if $B[i, k] \neq B[i-1, k]$ then
        mark the i^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Knapsack 0-1 Algorithm Finding the Items

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

Knapsack:
Item 2

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 2$
$k = 5$
$v_i = 4$
$w_i = 3$
B[i,k] = 7
B[i-1,k] = 3
$k - w_i = 2$

$i = n$, $k = W$
while i, k > 0
    if *B[i, k] ≠ B[i-1, k]* then
        mark the i^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Knapsack 0-1 Algorithm Finding the Items

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

Knapsack:
Item 2
Item 1

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
B[i,k] = 3
B[i-1,k] = 0
$k - w_i = 0$

$i = n$, $k = W$
while i, k > 0
    if $B[i, k] \neq B[i-1, k]$ then
        mark the i^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Knapsack 0-1 Algorithm Finding the Items

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

Knapsack:
Item 2
Item 1

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
B[i,k] = 3
$B[i-1,k] = 0$
$k - w_i = 0$

k = 0, so we're DONE!

The optimal knapsack should contain:

Item 1 and Item 2

Knapsack 0-1 Problem – Run Time

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

O(*W*)

```
for i = 1 to n
    B[i,0] = 0
```

O(*n*)

```
for i = 1 to n
    for w = 0 to W
        < the rest of the code >
```

Repeat *n* times

O(*W*)

What is the running time of this algorithm?

O(*n*W*)

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

Knapsack Problem

1) Fill out the dynamic programming table for the knapsack problem to the right.
2) Trace back through the table to find the items in the knapsack.