Dynamic Programming

Heejin Park

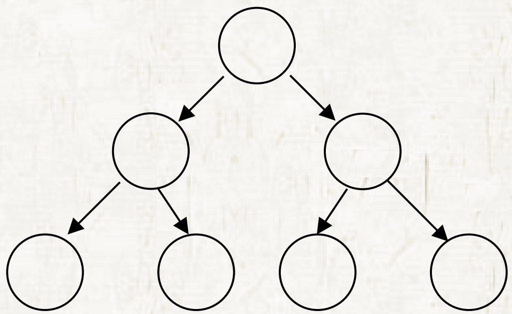
Division of Computer Science and Engineering

Hanyang University

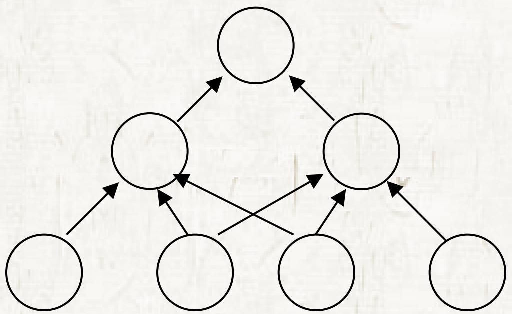
Contents

Dynamic Programming

- ***Dynamic programming*** solves a problem by partitioning the problem into subproblems.
  - The subproblems are ***independent***: divide-and-conquer method.
  - The subproblems are ***not independent***: dynamic programming.



divide-and-conquer



Dynamic Programming

- A dynamic programming algorithm solves every subproblem just once and ***saves its answer in a table*** and then reuse it.

Dynamic Programming

- Dynamic programming is typically to solve **optimization problems**.
- **Optimization problems**
  - There can be many possible solutions.
  - Each solution has a value.
  - We find a solution with ***the*** optimal (minimum or maximum) value.
  - Such a solution is called ***an*** optimal solution to the problem.
    - Shortest path example

Dynamic Programming

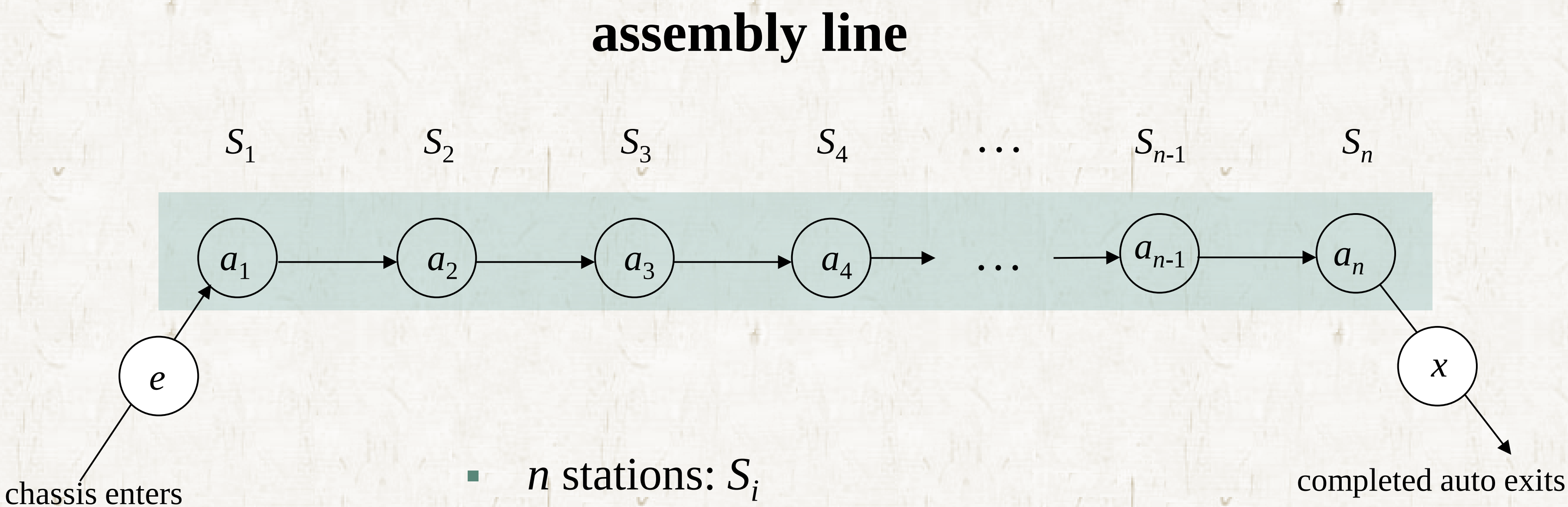
- The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

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.

Contents

- *Introduction*
- **Assembly-line scheduling**
- **Rod cutting**
- **Matrix-chain multiplication**
- **Elements of dynamic programming**
- **Longest common subsequence**

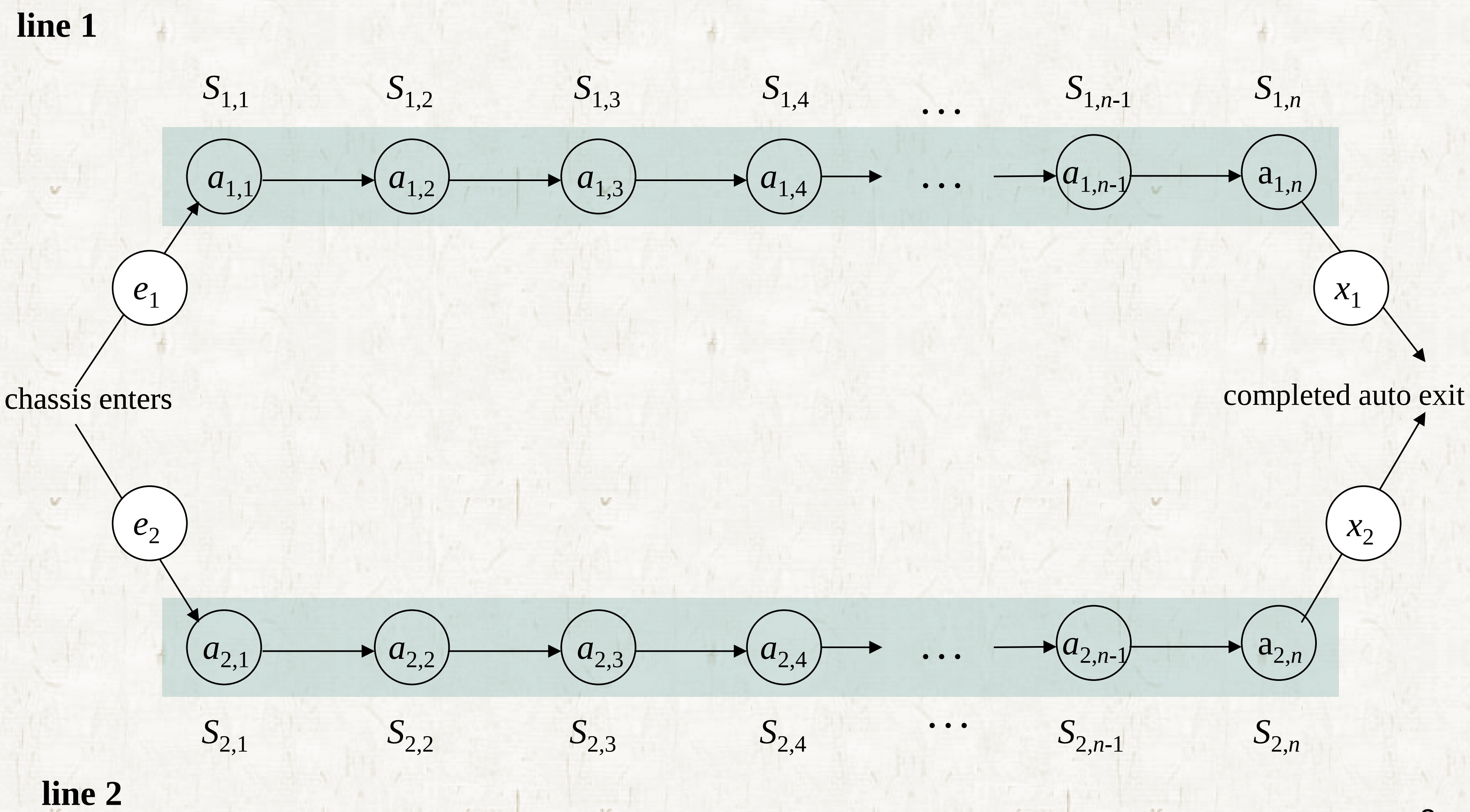
Assembly-line scheduling



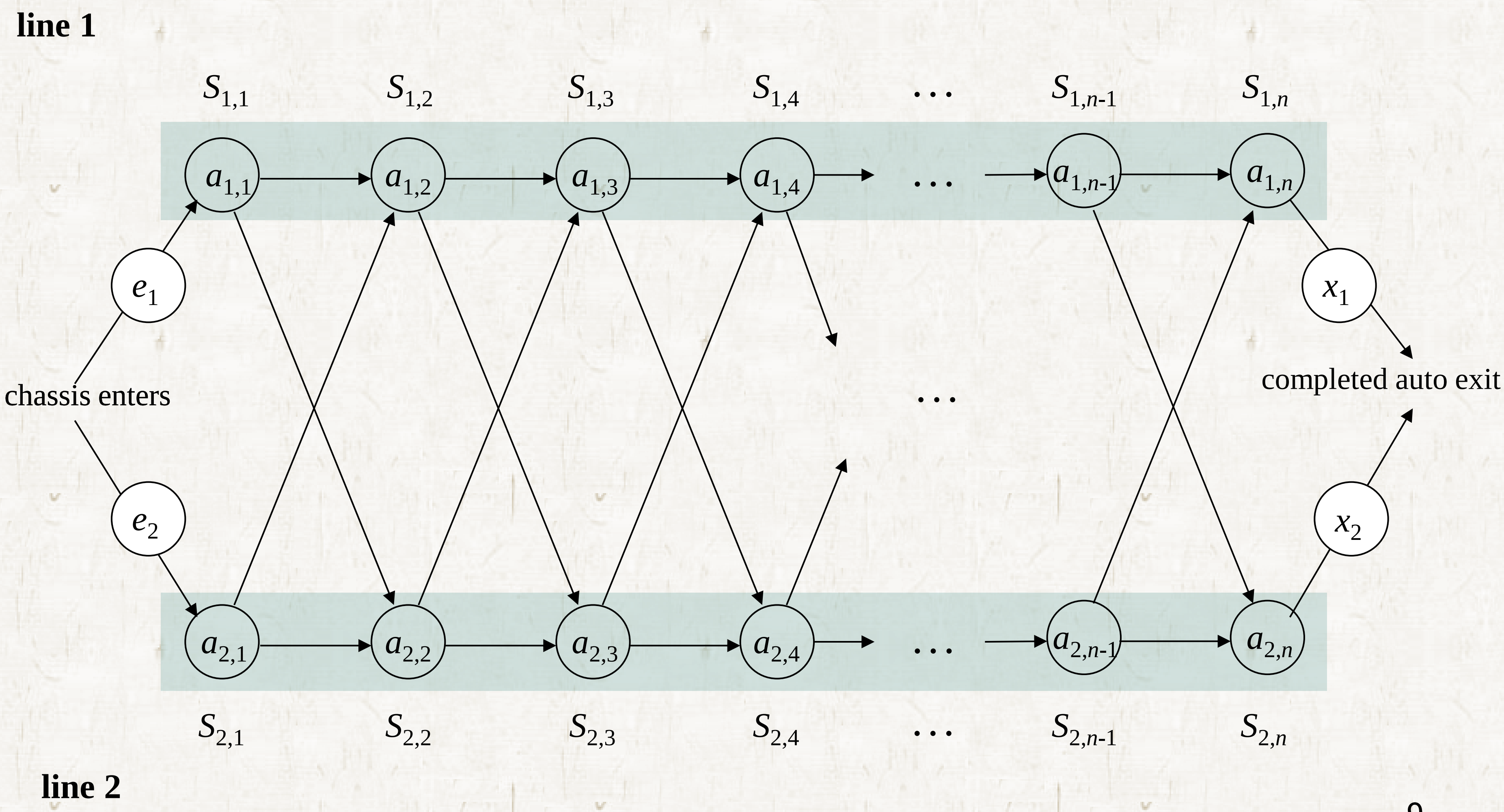
- n stations: S_i
- Assembly time in the ith station: a_j
- Entry time : e
- Exit time : x

- **The problem:** Determine the fastest assembly time

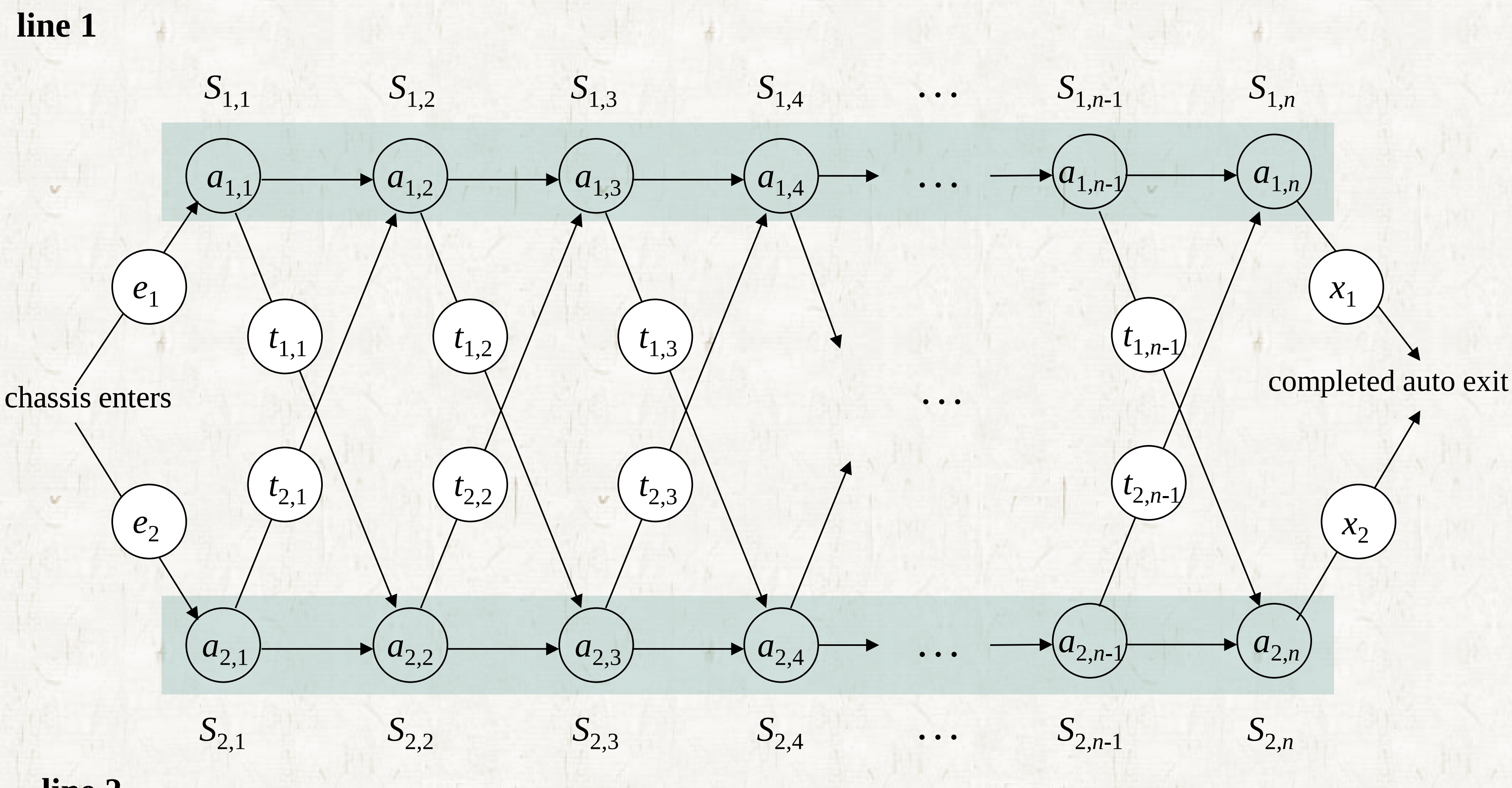
Assembly-line scheduling



Assembly-line scheduling



Assembly-line scheduling



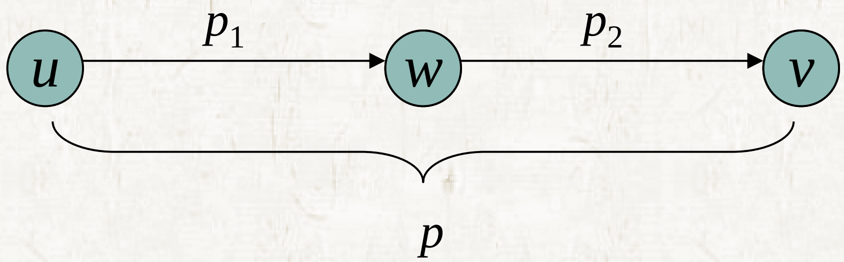
- Transfer time: $t_{i,j}$

Assembly-line scheduling

- **Brute-force approach**
  - Enumerate all possible ways and find a fastest way.
  - There are 2^n possible ways: Too many.

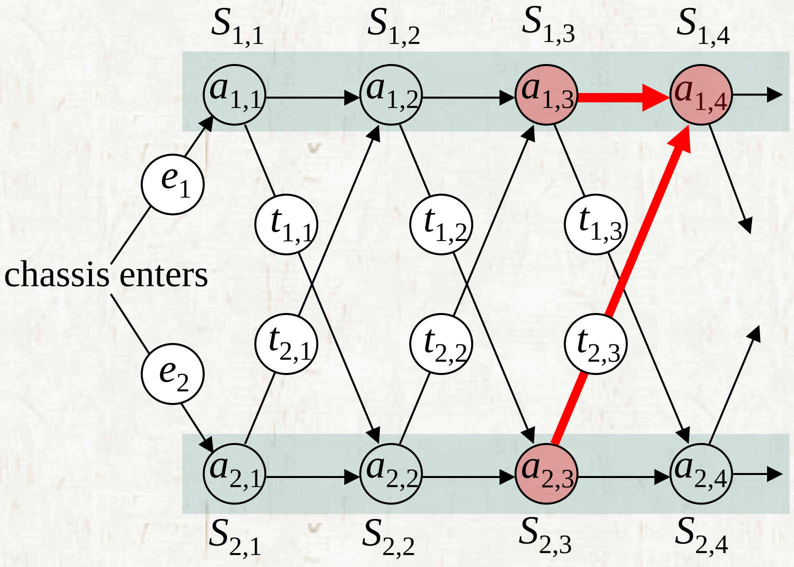
Assembly-line scheduling

- **Step 1: The structure of the fastest way through the factory**
  - **Optimal substructure**
    - An optimal solution to a problem contains within it an optimal solution to subproblems.
    - For example, shortest path problem in a graph.



Assembly-line scheduling

- In this case, the fastest way to $S_{1,4}$ contains the fastest way through either $S_{1,3}$ or $S_{2,3}$.
- Generally, the fastest way through station $S_{i,j}$ contains the fastest way through either $S_{1,j\text{-}1}$ or $S_{2,j\text{-}1}$.



Assembly-line scheduling

- **Step 2: A recursive solution**
  - $\boldsymbol{f_i[j]}$ denotes the fastest time to finish station $S_{i,j}$.
    - $f_1[4]$ is the fastest time to finish the 4th station in line 1.
    - $f_2[3]$ is the fastest time to finish the 3th station in line 2.
  - $\boldsymbol{f^*}$ denotes the fastest time to finish all stations.

Assembly-line scheduling

- **$f_i[j]$ for j = 1**
  - $f_1[1] = e_1 + a_{1,1}$
  - $f_2[1] = e_2 + a_{2,1}$
- **$f_i[j]$ for j >1**
  - $f_1[j] = \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j})$
  - $f_2[j] = \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j})$

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

Assembly-line scheduling

- **Step 3: Computing the fastest times**
  - ***Simple recursive solution***
    - The running time is $\Theta(2^n)$.
      - Let $r_i(j)$ be the number of references made to $f_i[j]$.
      - $r_i(j)$ for $j = n$

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

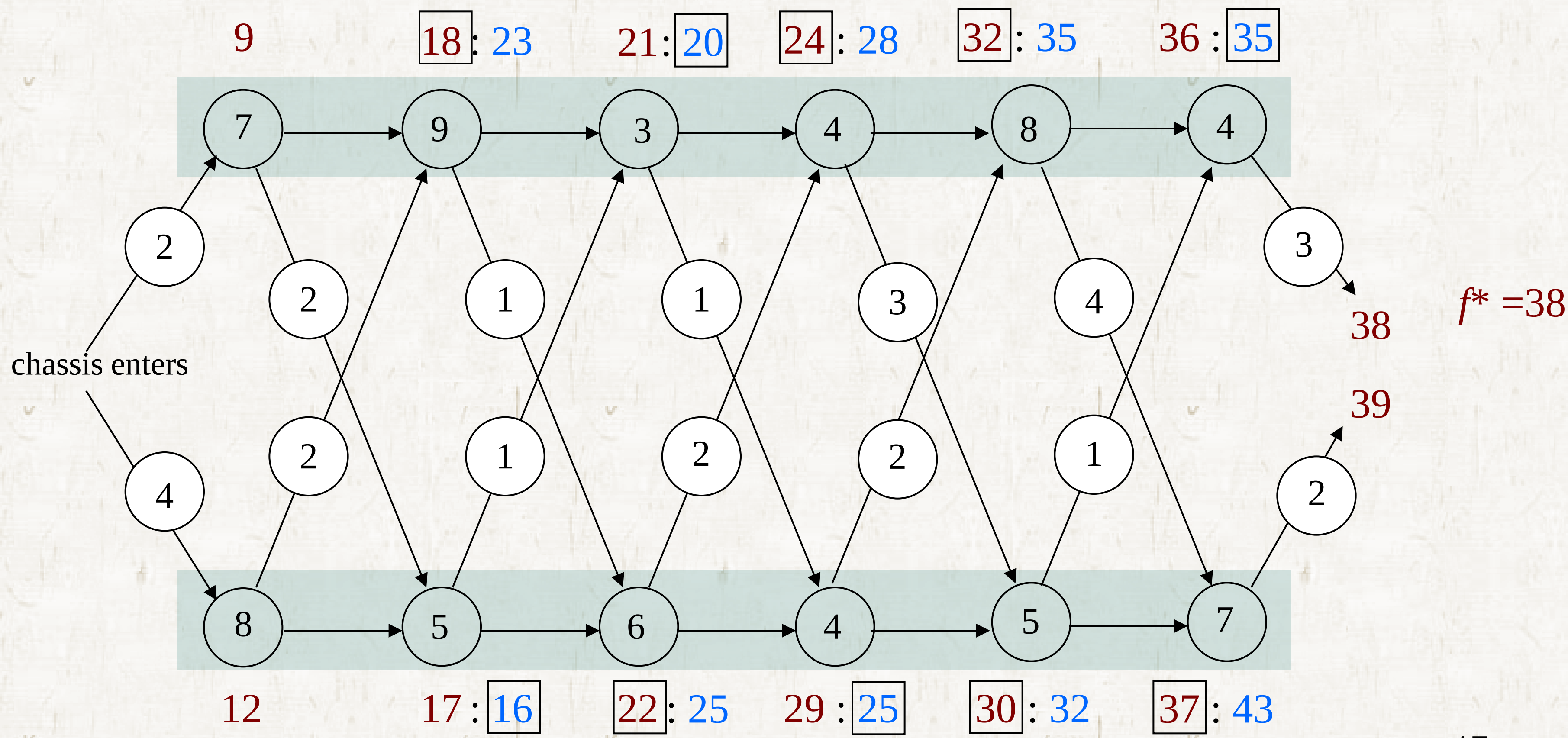
      - $r_i(j)$ for $j < n$
        - $r_i(j) = 2^{n-j}$

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

    - $\boldsymbol{f_1}$**[1]** and $\boldsymbol{f_2}$**[1]** are referred 2^{n-1} times, respectively.

Assembly-line scheduling

- ***Dynamic programming***

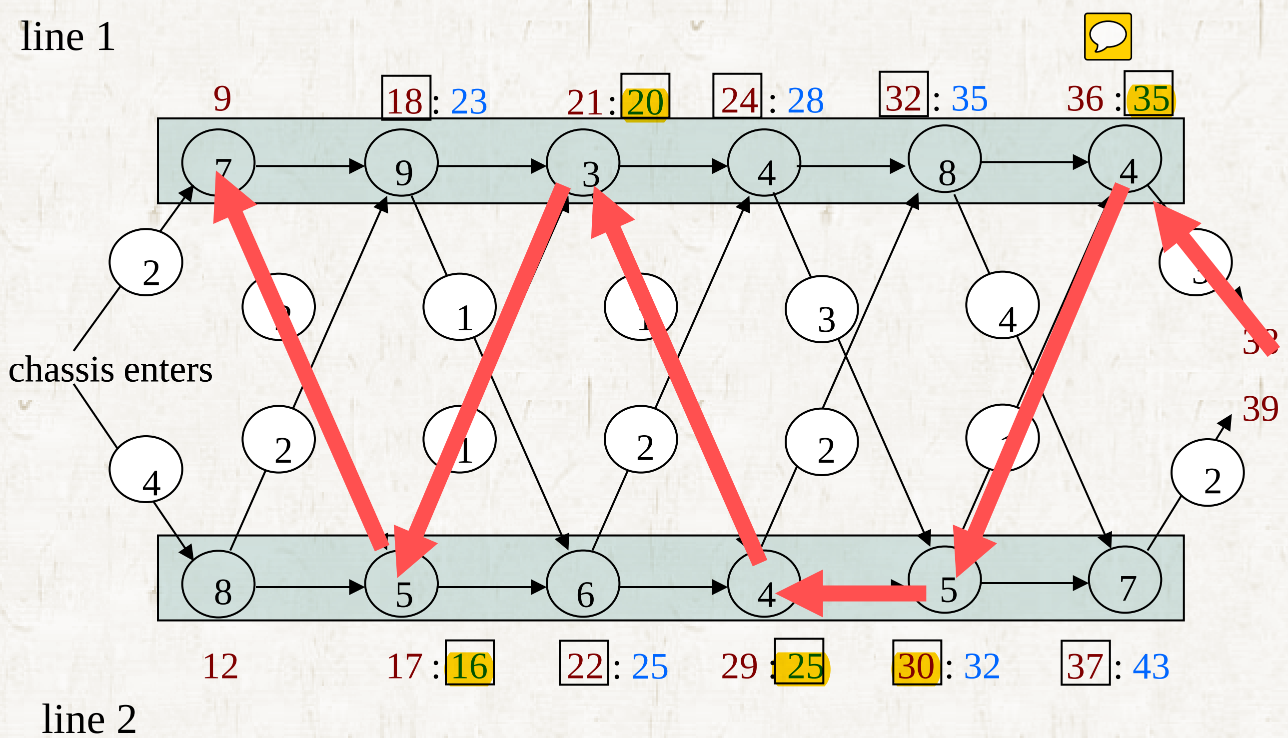


Assembly-line scheduling

- **Running time**
  - We can compute the fastest time in $\Theta(n)$ time.

Assembly-line scheduling

- **Step 4: Constructing the fastest way through the factory**
  - The $l_i[j]$ values help us trace a fastest way.



$l^* = 1$

	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

Assembly-line scheduling

- **Space consumption**
  - Table ***f*****:** $2n$
  - Table ***l*****:** $2n$-2
- **Space reduction**
  - Table ***f***
    - $2n \rightarrow 4$
  - Table ***l***
    - $2n-2 \rightarrow 0$
    - If table *f* table is preserved.

Contents

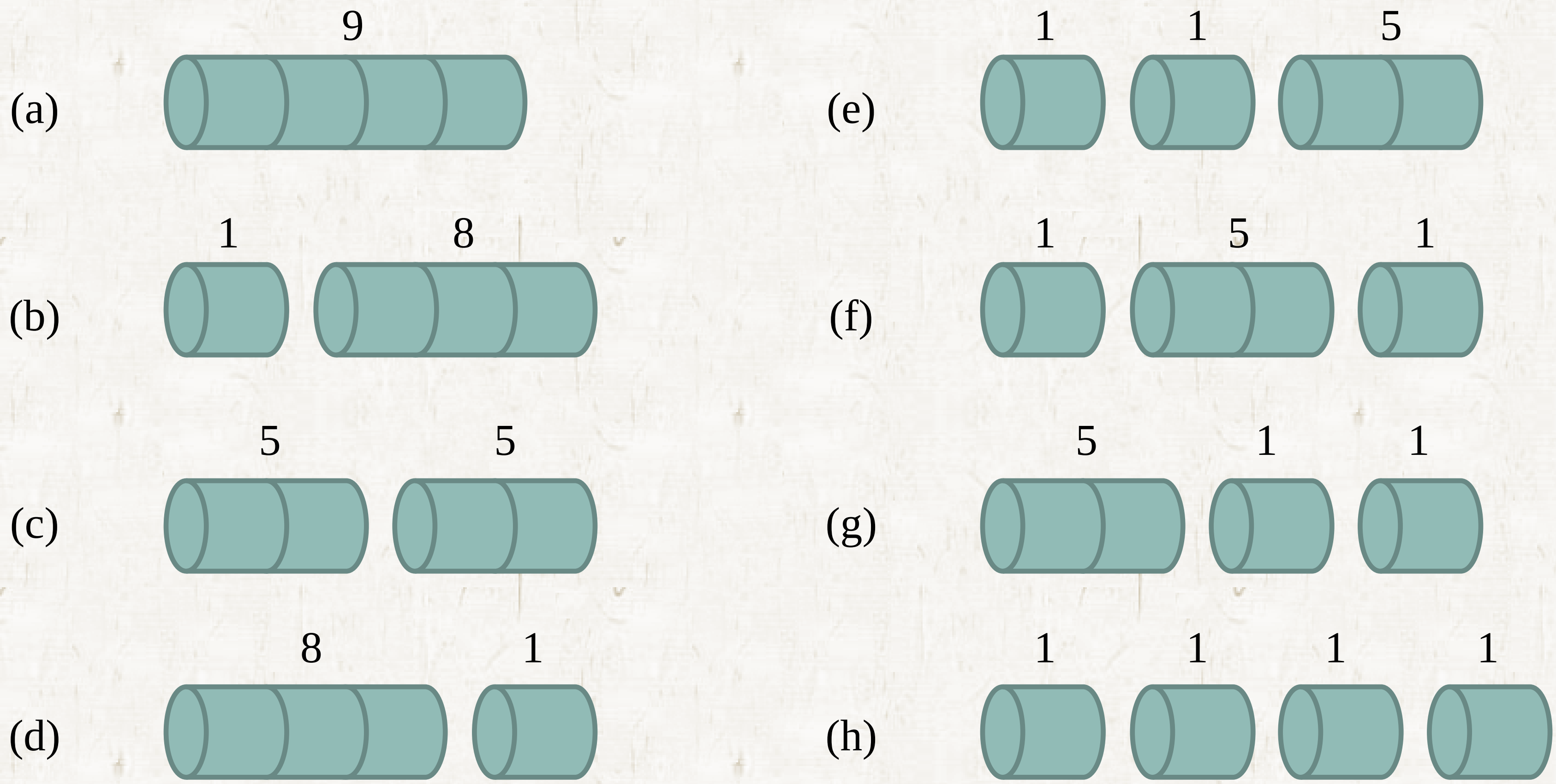
- *Introduction*
- *Assembly-line scheduling*
- **Rod cutting**
- **Matrix-chain multiplication**
- **Elements of dynamic programming**
- **Longest common subsequence**

Rod cutting

- The ***rod-cutting problem*: ** Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

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

Rod cutting



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

Rod cutting

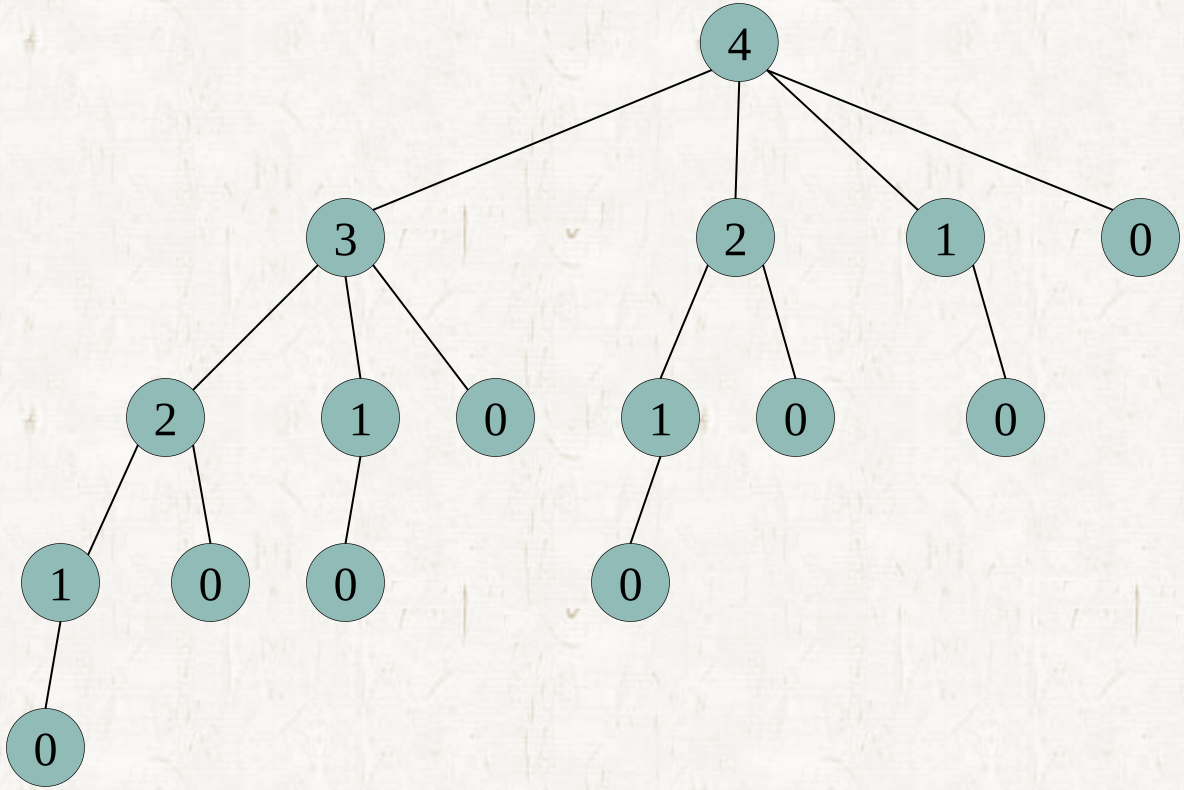
$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

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

Rod cutting

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

Rod cutting



Rod cutting

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

$$T(n) = 2^n$$

Rod cutting

MEMOIZED-CUT-ROD (*p*, *n*)

let $r[0 \,..\, n]$ be a new array

for $i = 0$ **to** n

    $r[i] = -\infty$

return MEMOIZE-CUT-ROD-AUX (*p*, *n*, *r*)

Rod cutting

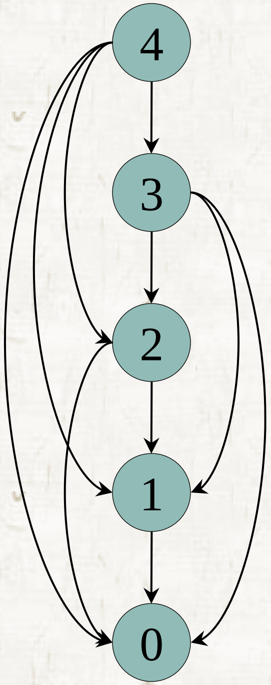
```
MEMOIZED-CUT-ROD-AUX (p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

Rod cutting

```
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]
```

Rod cutting

- Subproblem graphs



Rod cutting

```
EXTENDED-BOTTOM-UP-CUT-ROD (p, n)
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
```

Rod cutting

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD (p, n)
while n > 0
    print s[n]
    n = n – s[n]
```

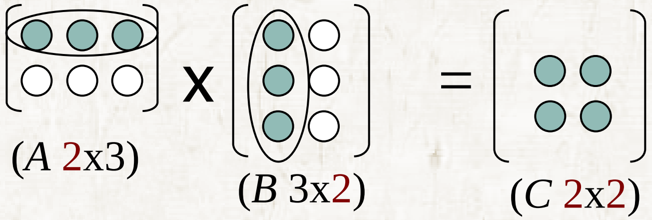
Rod cutting

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

Contents

Matrix-chain multiplication

- **Multiplying two matrices *A* and *B***
  - We can multiply them if they are compatible: the number of columns of *A* must equal the number of rows of *B*.
  - If *A* is a $p \times q$ matrix and *B* is a $q \times r$ matrix, the resulting matrix is a $p \times r$ matrix.



Matrix-chain multiplication

- **The number of scalar multiplications to multiply *A* and *B*.**
  - **It is *pqr*** because we compute ***pr*** elements and computing each element needs ***q*** scalar multiplications.

Matrix-chain multiplication

- **The order of multiplications**
  - The order of multiplications does not change ***the value of the product*** because matrix multiplication is associative.
  - For example, whether the left multiplication is done first or the right multiplication is done first does not matter.

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

  - However, the order of multiplication affects ***the number of scalar multiplications*** needed to compute the product.

Matrix-chain multiplication

- **The order of multiplications affects the number of scalar multiplications.**
  - Computing $\boldsymbol{A_1 A_2 A_3}$ where $\boldsymbol{A_1}$: 10×100 $\boldsymbol{A_2}$: 100×5 $\boldsymbol{A_3}$: 5×50
  - $\boldsymbol{(A_1\, A_2)\, A_3}$
    - $(A_1\, A_2)$ = 10*100*5 = 5000, $(10 \times 5)\, A_3$ = 10*5*50 = 2500
      =>5000 + 2500 = **7,500**
  - $\boldsymbol{A_1\, (A_2\, A_3)}$
    - $(A_2\, A_3)$ = 100*5*50 =25000, $A_1(100 \times 50)$ = 10 *100*50 = 50000
      =>25000 + 50000 = **75,000**
  - **Computing $\boldsymbol{(A_1\, A_2)\, A_3}$ is 10 times faster.**

Matrix-chain multiplication

- **Matrix-chain multiplication problem**
  - Given a chain $A_1, A_2, ..., A_n$ of n matrices, where matrix A_i has dimension $p_{i-1} \times p_i$, find the order of matrix multiplications minimizing the scalar multiplications to compute the product.
  - That is, to fully parenthesize the product of matrices minimizing scalar multiplications.

Matrix-chain multiplication

- The product $A_1 A_2 A_3 A_4$ can be fully parenthesized in five distinct ways.

  $A_1(A_2(A_3 A_4))$, $A_1((A_2 A_3) A_4)$,

  $(A_1 A_2)(A_3 A_4)$,

  $(A_1(A_2 A_3))A_4$, $((A_1 A_2) A_3) A_4$.

Matrix-chain multiplication

- **Solutions of the matrix-chain multiplication problem**
  - **Brute-force approach**
    - Enumerate all possible parenthesizations.
    - Compute the number of scalar multiplications of each parenthesization.
    - Select the parenthesization needing the least number of scalar multiplications.

Matrix-chain multiplication

- **The Brute-force approach is inefficient.**
  - The number of parenthesizations of a product of n matrices, denoted by $P(n)$, is as follows.

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

  - The number of enumerated parenthesizations is $\Omega(4^n/n^{3/2})$.

Matrix-chain multiplication

- **Dynamic programming**
- **Optimal substructure**
  - $m[i,j]$: The minimum number of scalar multiplications for computing $A_i A_{i+1} \ldots A_j$.

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

  - matrix $A_i : p_{i-1} \times p_i$
  - computing $A_{i..k}\, A_{k+1..j}$ takes $p_{i-1}\, p_k\, p_j$ scalar multiplications.
  - $s[i, j]$ stores the optimal k for tracing the optimal solution.

Matrix-chain multiplication

i \ j	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

m

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

s

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

$i=2,\ \ j=5,\ \ i \le k < j$

Matrix-chain multiplication

- **Running time**
  - O(n^3) time in total
    - $\Theta(n^2)$ subproblems
    - $O(n)$ time for each subproblem

- **Space consumption**
  - $\Theta(n^2)$ space to store the m and s tables.

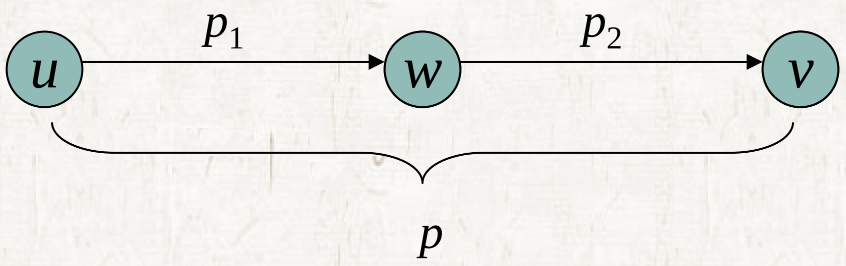
Contents

Elements of dynamic programming

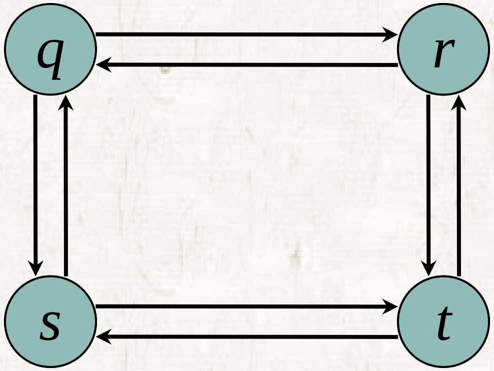
- **Elements of dynamic programming**
  - *Optimal substructure*
  - *Overlapping subproblems*

Element of dynamic programming

- **Subtleties**
  - Unweighted longest simple path problem
    - Does it have optimal substructure?



Elements of dynamic programming

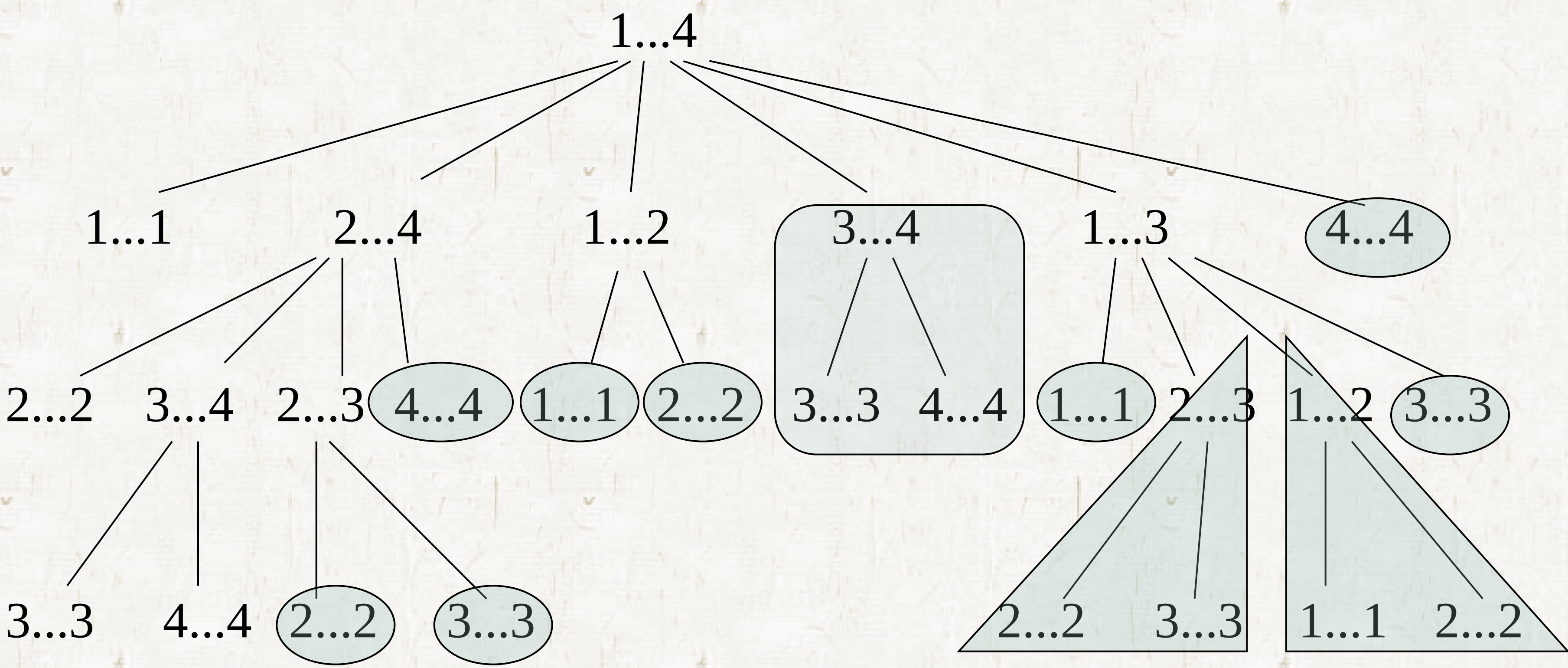


$q \rightarrow r \rightarrow t$

Elements of dynamic programming

- **Overlapping subproblems**
  - When a recursive algorithm revisits the same problem over and over again, the optimization problem has ***overlapping subproblems***.

Elements of dynamic programming



Matrix chain multiplication: top-down vs. bottom-up

Elements of dynamic programming

- **Memoization**
  - Recursive solution but solve each subproblem only once.
  - Fills the table in recursive way.
  - In most cases, it is slower than dynamic programming.
  - It is useful when only a part of subproblems are solved.

Elements of dynamic programming

- **The running time** of a dynamic-programming algorithm depends on the product of two factors.
  - ***The number of subproblems overall***.
  - ***How many choices each subproblem has***.

  - Assembly line scheduling
    - $\Theta(n)$ subproblems $\cdot$ 2 choices = $\Theta(n)$
  - Matrix chain multiplication
    - $\Theta(n^2)$ subproblems $\cdot$ $(n\text{-}1)$ choices = $O(n^3)$

Contents

- *Introduction*
- *Assembly-line scheduling*
- *Rod cutting*
- *Matrix-chain multiplication*
- *Elements of dynamic programming*
- **Longest common subsequence**

Longest common subsequence

- **Definition**
  - ***Character***
  - ***Alphabet***: A set of characters
    - English alphabet: {*A*, *B*, …*Z*}
    - Korean alphabet: { ㄱ,ㄴ,…ㅎ, ㅏ,… ㅣ}
  - ***String*** (or ***sequence***): A list of characters from an alphabet
    - ex> strings over {0,1}: Binary strings
    - ex> strings over {*A*,*C*,*G*,*T*}: DNA sequences

Longest common subsequence

- ***Substring***
  - *CBD* is a substring of *ABCBDAB*
- ***Subsequence***
  - *BCDB* is a subsequence of *ABCBDAB*
- ***Common subsequence***
  - *BCA* is a common subsequence of *X*=*ABCBDAB* and *Y*=*BDCABA*

Longest common subsequence

- ***Longest common subsequence* (LCS)**
  - *BCBA* is the longest common subsequence of *X* and *Y*

X = A B C B D A B

Y = B D C A B A

- ***LCS problem***
  - Given two sequences $X=<x_1,x_2,\ldots,x_m>$ and $Y=<y_1, y_2,..,y_n>$ to find an LCS of X and Y.

Longest common subsequence

- **Brute force approach**
  - Enumerate all subsequences of X and check each subsequence if it is also a subsequence of Y and find the longest one.
  - Infeasible!
    - The number of subsequences of X is 2^m.

Longest common subsequence

- **Dynamic programming**

  - **The *i*th *prefix* X_i of *X*** is X_i=<$x_1, x_2, \ldots, x_i$>.

  - If X = <A, B, C, B, D, A, B>
    - X_4=< A, B, C, B>
    - X_0=< >

Longest common subsequence

- **Optimal substructure**
  - Let $X = x_1, x_2, ..., x_m$ and $Y = y_1, y_2, ..., y_n$ be sequences, and let $Z = z_1, z_2, ..., z_k$ 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 $x_m \neq y_n$, Z is an LCS of X_{m-1} and Y or an LCS of X and Y_{n-1}.

Longest common subsequence

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}.

- Suppose $z_k \neq x_m$.
- Then, we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k + 1$, Thus, $z_k = x_m$.

$X = A\ B\ C\ B\ D\ A\ B$

$Y = B\ D\ C\ A\ B$

Longest common subsequence

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}.

- Since $z_k = x_m$, Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1}.
- We show Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} by contradiction.
  - Suppose that there is a common subsequence W of X_{m-1} and Y_{n-1} with length greater than k - 1.
  - Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k.

$$X = A\ B\ C\ B\ D\ A\ B$$

$$Y = B\ D\ C\ A\ B$$

Longest common subsequence

2. If $x_m \neq y_n$, Z is an LCS of X_{m-1} and Y or an LCS of X and Y_{n-1}.

$X = A\ B\ C\ B\ D\ A\ B$

$Y = B\ D\ C\ A\ A$

$X = A\ B\ C\ B\ D\ A\ B$

$Y = B\ D\ C\ A\ A$

Longest common subsequence

- $c[i, j]$: The length of an LCS of the sequences X_i and Y_j .
- If either i = 0 or j = 0, so the LCS has length = 0.

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

Longest common subsequence

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

Longest common subsequence

- Computation time: $\Theta(mn)$
- Space: $\Theta(mn)$
- Space reduction: $\min(m,n)+1$