Dynamic Programming

อ.ธนาวินท์ รักธรรมานนท์

ภาควิชาวิศวกรรมคอมพิวเตอร์

มหาวิทยาลัยเกษตรศาสตร์

Dynamic Programming

Adapted from notes by Anany Lavitin, Tao Jiang, Changhui Yan, Chuan Yi Tang, David Matuszek,

Dynamic Programming

Dynamic Programming is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping subinstances

- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS
- "Programming" here means "planning"
- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table

Divide-and-Conquer and Dynamic Programming

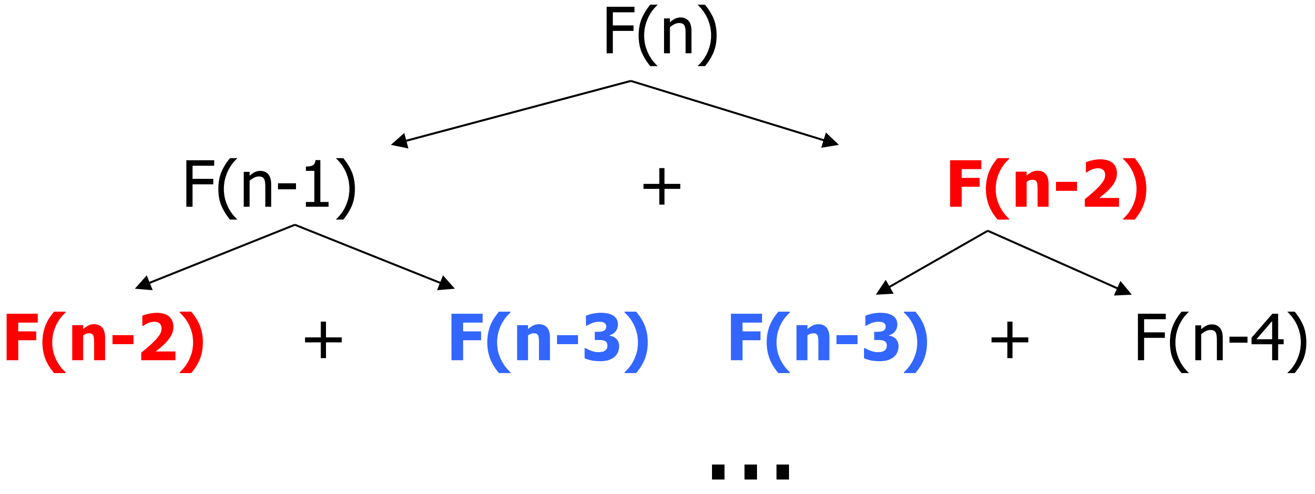
- Both algorithm methods solve a problem by combining solutions of *sub-problems*.
- However, there's a difference between the two:
    - In divide-and-conquer, the sub-problems don't overlap
        - Independent sub-problems, solve sub-problems independently and recursively, (so same sub(sub)problems solved *repeatedly*)
    - In dynamic programming, the sub-problems overlap
        - Sub-problems are dependent, i.e., sub-problems share sub-sub-problems, every sub(sub)problem solved just once, solutions to sub(sub)problems are stored in a table and used for solving higher level sub-problems.

Example: Fibonacci numbers

- Recall definition of Fibonacci numbers:

$$F(n) = F(n-1) + F(n-2)$$
$$F(0) = 0$$
$$F(1) = 1$$

- Computing the nth Fibonacci number recursively (top-down):



F(n)

F(n-1) + **F(n-2)**

F(n-2) + **F(n-3)** **F(n-3)** + F(n-4)

...

Example: Fibonacci numbers (cont.)

Computing the nth Fibonacci number using bottom-up iteration and recording results:

F(0) = 0
F(1) = 1
F(2) = 1+0 = 1
…
F(10) = F(9) + F(8)
…
F(n-1) =
F(n) = F(n-1) + F(n-2)

0	**1**	**1**	**. . .**	***F*(*n*-2)**	***F*(*n*-1)**	***F*(*n*)**

Efficiency:
- time O(n)
- space O(n)

What if we solve it recursively?

Fibonacci Number Pseudocode

A straightforward, but inefficient algorithm to compute the nth Fibonacci number would use a top-down approach:

```
Fibonacci (n)
if n = 0 then return 0
else if n = 1 then return 1
else return Fibonacci (n-1) + Fibonacci (n-2)
```

A more efficient, bottom-up approach starts with 0 and works up to *n*, requiring only *n* values to be computed:

```
Fibonacci(n)
$f[0] \leftarrow 0$
$f[1] \leftarrow 1$
for $i \leftarrow 2 \dots n$
    do $f[i] \leftarrow f[i-1] + f[i-2]$
return f[n];
```

Fibonacci Number Pseudocode

A straightforward, but inefficient algorithm to compute the nth Fibonacci number would use a top-down approach:

F	1	1	2	3	5	8	13	21

…

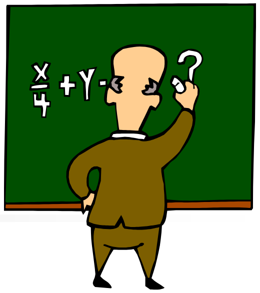
MEMOIZATION

```
Fibonacci (n)
if F(n) exists, return F(n)
if n = 0 then return 0
else if n = 1 then return 1
else return Fibonacci (n-1) + Fibonacci (n-2),   keep F(n)
```

A more efficient, bottom-up approach starts with 0 and works up to *n*, requiring only *n* values to be computed:

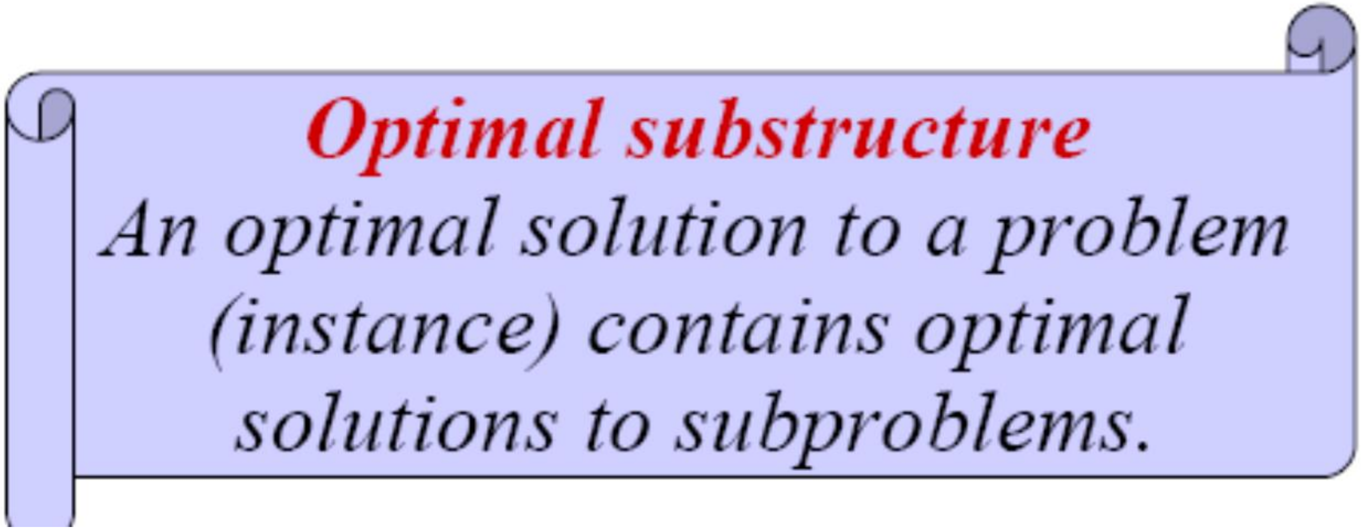
```
Fibonacci(n)
f[0] ← 0
f[1] ← 1
for i ← 2 … n
    do f[i] ← f[i-1] + f[i-2]
return f[n];
```

The General Dynamic Programming Technique



- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
    - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
    - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

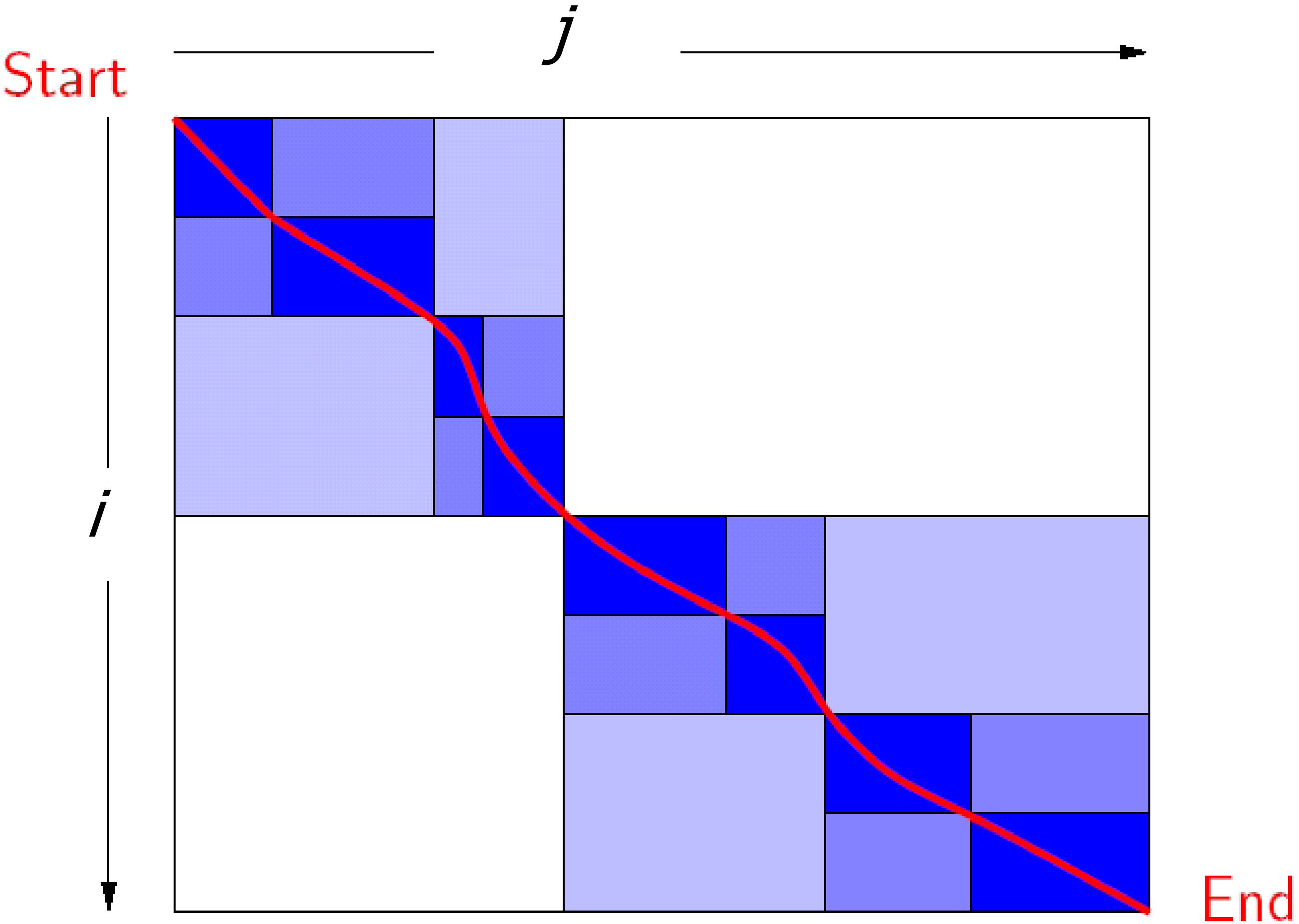
DP Properties



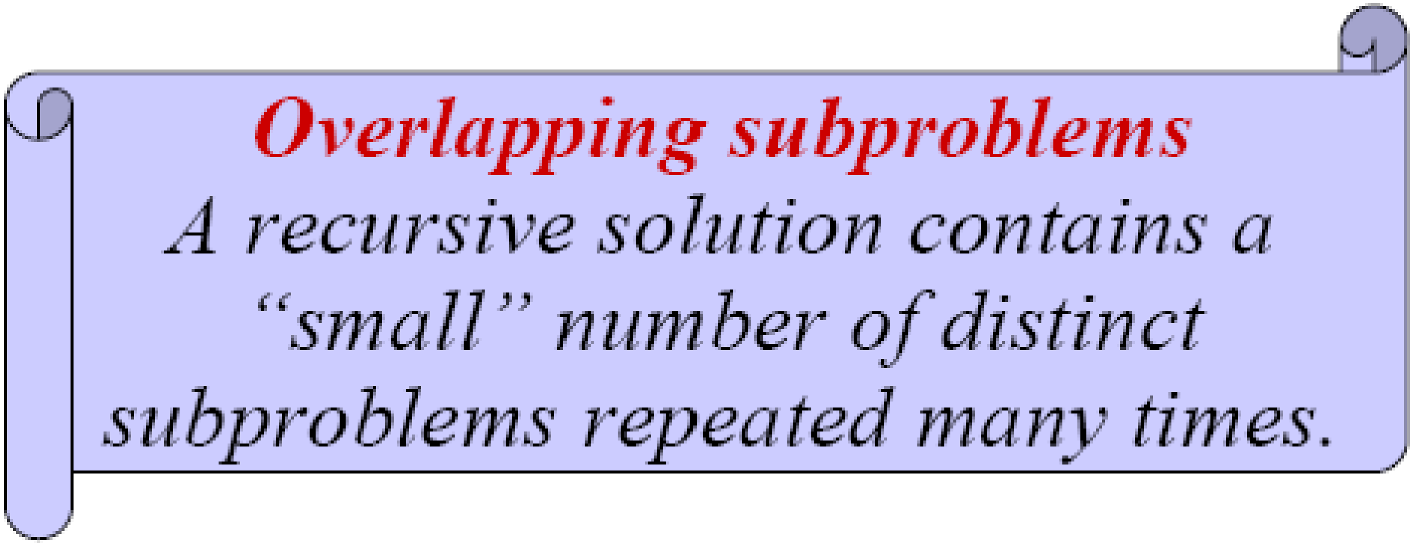
Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

Schmatic Diagram of DC Method



DP Properties

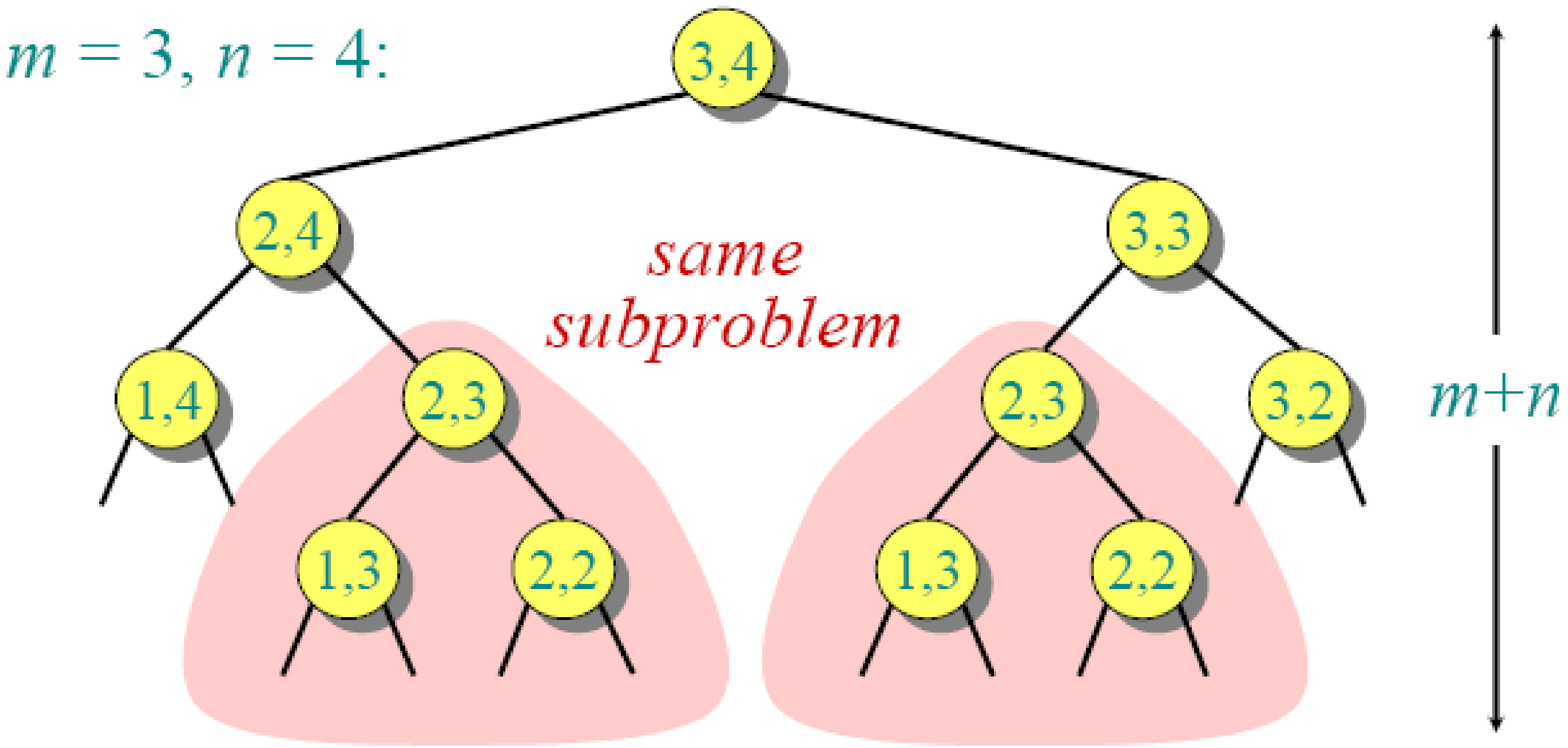


Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

DP Properties: Example

Recursion tree



$m = 3$, $n = 4$:

same subproblem

Height $= m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved!

DP Development

Development of a dynamic-programming algorithm can be broken into 4 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 from bottom-up**

4) **Construct an optimal solution from stored/computed information**

Assembly Line Scheduling Problem

Assembly Line Scheduling (ALS)

- 2 assembly lines at an auto assembly plant
- Normally they operate independently and concurrently
- But when there is a rush job the manager halts the assembly lines and use stations in both assembly lines in an optimal way, to be explained next

Assembly Line Scheduling (ALS)

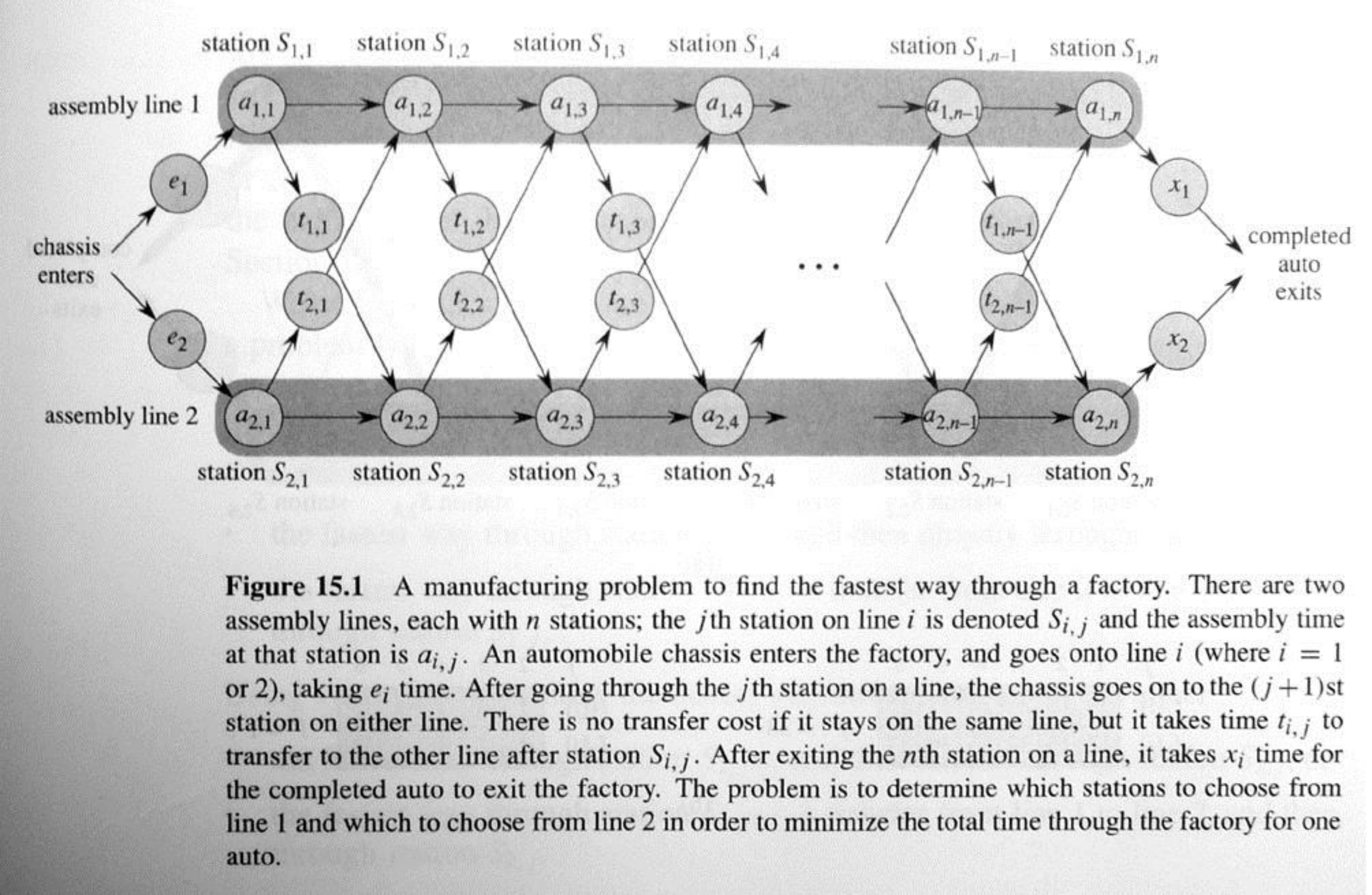
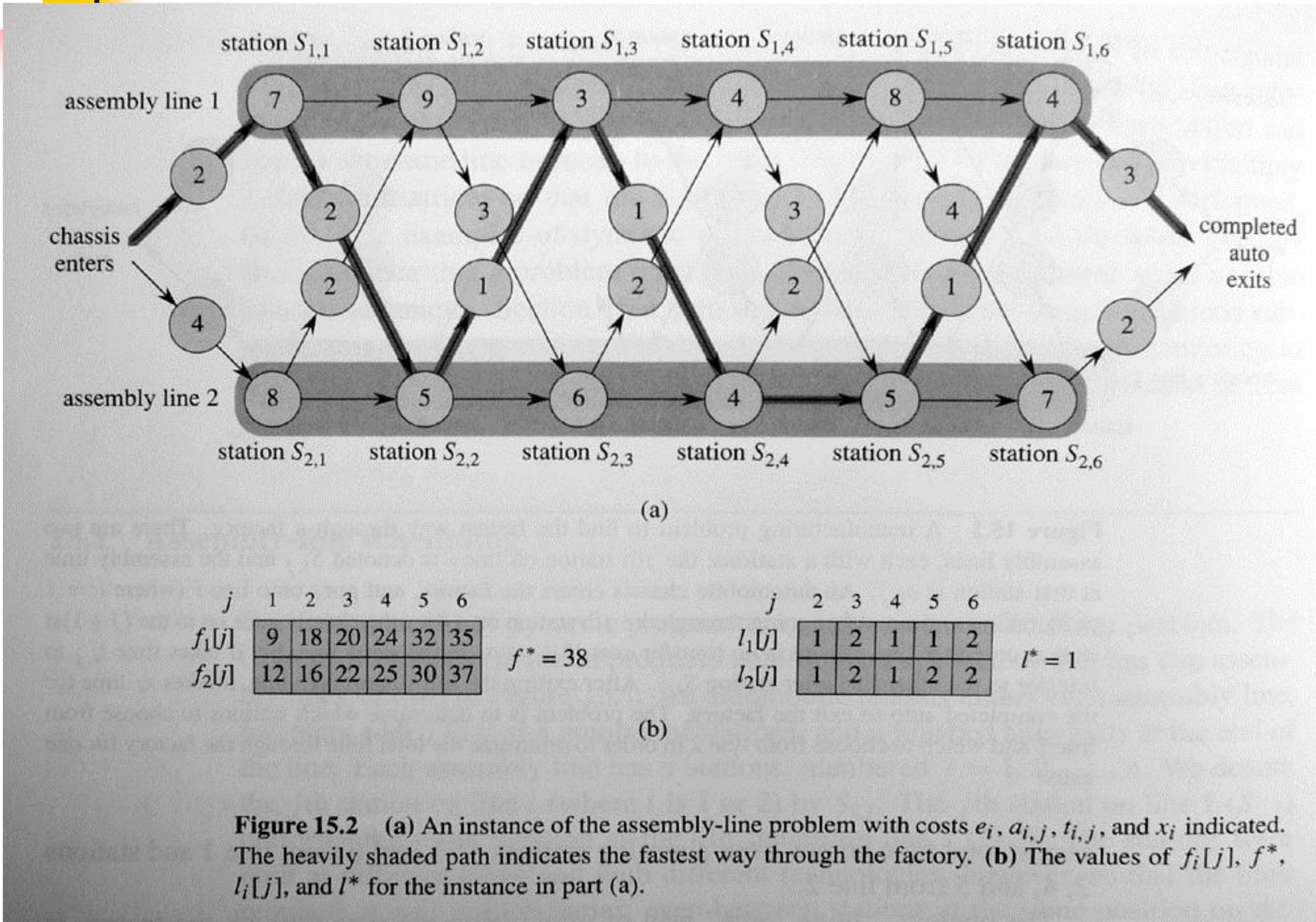


Figure 15.1 A manufacturing problem to find the fastest way through a factory. There are two assembly lines, each with n stations; the jth station on line i is denoted $S_{i,j}$ and the assembly time at that station is $a_{i,j}$. An automobile chassis enters the factory, and goes onto line i (where $i = 1$ 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.

Concrete Instance of ALS



(a)

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

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

$l^* = 1$

(b)

Figure 15.2 **(a)** An instance of the assembly-line problem with costs e_i, $a_{i,j}$, $t_{i,j}$, and x_i indicated. The heavily shaded path indicates the fastest way through the factory. **(b)** The values of $f_i[j]$, f^*, $l_i[j]$, and l^* for the instance in part (a).

Brute Force Solution

- List all possible sequences,
- For each sequence of n stations, compute the passing time. (the computation takes $\Theta(n)$ time.)
- Record the sequence with smaller passing time.
- However, there are total 2^n possible sequences.

ALS --DP steps: Step 1

- **Step 1**: find the structure of the fastest way through factory
  - Consider the fastest way from starting point through station $S_{1,j}$ (same for $S_{2,j}$)
    - j=1, only one possibility
    - j=2,3,…,n, two possibilities: from $S_{1,j-1}$ or $S_{2,j-1}$
      - from $S_{1,j-1}$, additional time $a_{1,j}$
      - from $S_{2,j-1}$, additional time $t_{2,j-1} + a_{1,j}$
    - suppose the fastest way through $S_{1,j}$ is through $S_{1,j-1}$, then the chassis must have taken a fastest way from starting point through $S_{1,j-1}$. Why???
    - Similarly for $S_{2,j-1}$.

DP step 1: Find Optimal Structure

- An optimal solution to a problem contains within it an optimal solution to subproblems.
- the fastest way through station $S_{i,j}$ contains within it the fastest way through station $S_{1,j-1}$ or $S_{2,j-1}$.
- Thus can construct an optimal solution to a problem from the optimal solutions to subproblems.

ALS --DP steps: Step 2

- **Step 2**: A recursive solution
- Let $f_i[j]$ (i=1,2 and j=1,2,…, n) denote the fastest possible time to finish station j of line i from starting point (through $S_{i,j}$).
- Let f* denote the fastest time for finishing all the way through the factory. Then

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

- $f_1[1]=e_1+a_{1,1}$, fastest time to get through $S_{1,1}$
- $f_1[j]=\min(f_1[j-1]+a_{1,j}, f_2[j-1]+ t_{2,j-1}+ a_{1,j})$
- Similarly to $f_2[j]$.

ALS --DP steps: Step 2

- Recursive solution:
  - $f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$
  - $f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j=1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j>1 \end{cases}$
  - $f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j=1 \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j>1 \end{cases}$
- $f_i[j]$ $(i=1,2; j=1,2,\ldots,n)$ records optimal values to the subproblems.
- To keep the track of the fastest way, introduce $l_i[j]$ to record the line number (1 or 2), whose station j-1 is used in a fastest way through $S_{i,j}$.
- Introduce l* to be the line whose station n is used in a fastest way through the factory.

ALS --DP steps: Step 3

- **Step 3**: Computing the fastest time
  - One possible option: *Recursive algorithm*
    - Let $r_i(j)$ be the number of references made to $f_i[j]$
      - $r_1(n) = r_2(n) = 1$
      - $r_1(j) = r_2(j) = r_1(j+1)+ r_2(j+1)$
      - $r_i (j) = 2^{n-j}$.
      - So $f_1[1]$ is referred to 2^{n-1} times.
      - Total references to all $f_i[j]$ is $\Theta(2^n)$.
    - Thus, the running time is exponential.
    - Memoization is needed.
  - Another possible option: *Non-recursive algorithm*

ALS Step 3 – Coding (Bottom-Up)

```
Fastest-Way(a, t, e, x, n)
 1.  $f_1[1] = e_1 + a_{1,1}$
 2.  $f_2[1] = e_2 + a_{2,1}$
 3.  for j = 2 to n
 4.      if $f_1[j-1] + a_{1,j} \le f_2[j-1] + t_{2,j-1} + a_{1,j}$
 5.          $f_1[j] = f_1[j-1] + a_{1,j}$
 6.          $l_1[j] = 1$
 7.      else
 8.          $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$
 9.          $l_1[j] = 2$

10-15.   // same as lines 4-9 for $f_2$ and $l_2$

16.  if $f_1[n] + x_1 \le f_2[n] + x_2$
17.          $f^* = f_1[n] + x_1$
18.          $l^* = 1$  // note: $l^*$ is the line whose station n is the last
19.  else  $f^* = f_2[n] + x_2$
20.          $l^* = 2$
```

Linear running time

ALS --DP steps: Step 4

- **Step 4**: Construct the fastest way through the factory
  - Use value $l_i[j]$ to trace back the solution

```
Print-Stations(l, n)
  i = l*
  print "line "+i+", station "+n
  for j = n downto 2
      i = $l_i[j]$
      print "line "+i+", station "+j - 1
```

Longest Increasing Subsequene

Longest increasing subsequence

INPUT: numbers $a_1, a_2, \ldots , a_n$

OUTPUT: longest increasing subsequence

1,9,2,4,7,5,6

↓

1,9,2,4,7,5,6

DP algorithm for Longest Increasing Subsequence

- Problem: Given a sequence $s_1,s_2,\ldots,s_N$ of integers, find a longest increasing subsequence

- Algorithm: We compute F*(j) for j=1,2,…,N where F*(j) is the length of the longest increasing subsequence of $s_1,s_2,\ldots,s_j$ that includes s_j

LIS Algorithm

- Step 1: Def of subproblem: previous slide
- Step 2: Solution obtained by taking Max{F*(1), F*(2), …, F*(N)}
- Step 3: Base case: F*(1)=1
- Step 4: Order of subproblems: F*(1), then F*(2), then F*(3)., etc., up to F*(N)
- Step 5: For j>1,

  $$F^*(j) = \max_k \{1, F^*(k)+1 : k<j, \text{ and } s_k < s_j\}$$

Longest Common Subsequence Problem

Longest Common Subsequence

LCS Problem: Given two strings, find a longest subsequence that they share.

- substring vs. subsequence of a string
  - Substring: the characters in a substring of S must occur *contiguously* in S
  - Subsequence: the characters can be interspersed with *gaps*.

- *Consider ababc* and *abdcb*

alignment 1

`ababc.`

`abd.cb` the longest common subsequence is ab..c with length 3

alignment 2

`aba.bc`

`abdcb.` the longest common subsequence is ab..b with length 3

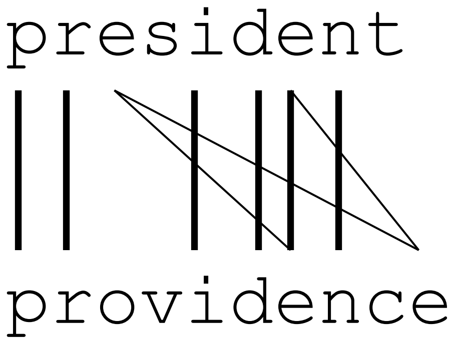
Longest Common Subsequence

For instance,

Sequence 1: president

Sequence 2: providence

Its LCS is priden.



president

providence

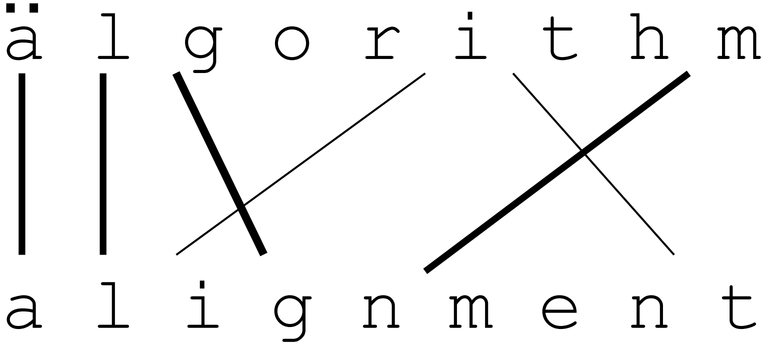
Longest Common Subsequence

For another example,

Sequence 1: algorithm

Sequence 2: alignment

Its LCS can be algm or alit or algt or …



a l g o r i t h m

a l i g n m e n t

Longest Common Subsequence

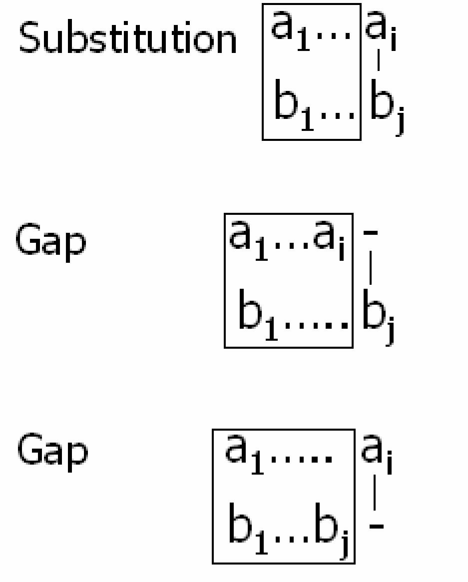
- Subproblem optimality

Consider two sequences

S_1: $a_1a_2a_3\ldots a_i$
S_2: $b_1b_2b_3\ldots b_j$

There are three possible cases

for the current pair :



Longest Common Subsequence

There are three cases for the last element:

S_1: $a_1a_2a_3\ldots a_i$
S_2: $b_1b_2b_3\ldots b_j$

Substitution $\begin{array}{|l|l} \hline a_1\ldots & a_i \\ & | \\ b_1\ldots & b_j \\ \hline \end{array}$ $M_{i,j} = M_{i-1,\, j-1} + S_{i,j}$ (match/mismatch)

Gap $\begin{array}{|l|l} \hline a_1\ldots a_i & - \\ & | \\ b_1\ldots\ldots & b_j \\ \hline \end{array}$ $M_{i,j} = M_{i,j-1} + w$ (gap in sequence #1)

Gap $\begin{array}{|l|l} \hline a_1\ldots\ldots & a_i \\ & | \\ b_1\ldots b_j & - \\ \hline \end{array}$ $M_{i,j} = M_{i-1,j} + w$ (gap in sequence #2)

$$M_{i,j} = \text{MAX}\ \{M_{i-1,\ j-1} + S\ (a_i, b_j)\ (\text{match} = 1\ /\ \text{mismatch} = 0)$$
$$M_{i,j-1} + 0\ (\text{gap in sequence \#1})$$
$$M_{i-1,j} + 0\ (\text{gap in sequence \#2}) \quad \}$$

$M_{i,j}$ is the score for optimal alignment between strings *a[1…i]* (substring of *a* from index *1* to *i*) and *b[1…j]*

Longest Common Subsequence

$$M_{i,j} = \text{MAX} \left\{ \begin{array}{l} M_{i-1,\, j-1} + S(a_i,b_j) \\ M_{i,j-1} + 0 \\ M_{i-1,j} + 0 \end{array} \right\}$$

$s(a_i,b_j)= 1$ if $a_i=b_j$

$s(a_i,b_j)= 0$ if $a_i \neq b_j$ or *any of them is a gap*

Examples:

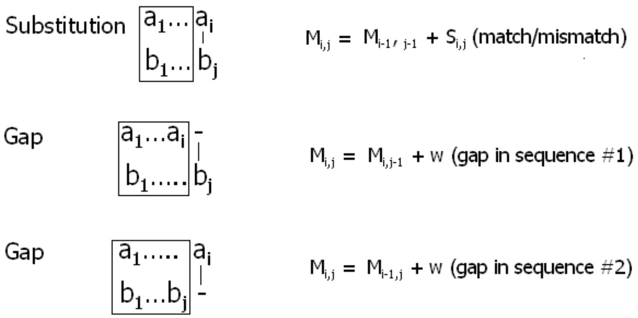
G A A T T C A G T T A (sequence #1)

G G A T C G A (sequence #2)

Longest Common Subsequence

Fill the score matrix M and trace back table B

	G	A	A	T	T	C	A	G	T	T	A
0	0	0	0	0	0	0	0	0	0	0	0
G											
G											
A											
T											
C											
G											
A											

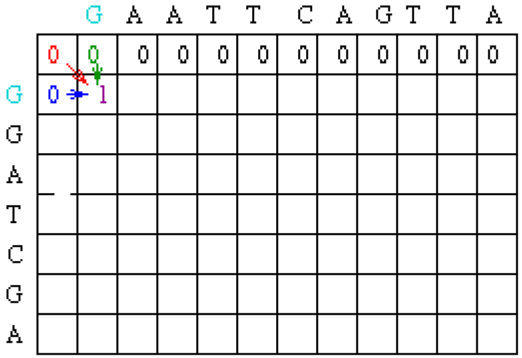


Substitution $a_1 \ldots a_i$ / $b_1 \ldots b_j$ $M_{i,j} = M_{i-1, j-1} + S_{i,j}$ (match/mismatch)

Gap $a_1 \ldots a_i$ - / $b_1 \ldots b_j$ $M_{i,j} = M_{i,j-1} + w$ (gap in sequence #1)

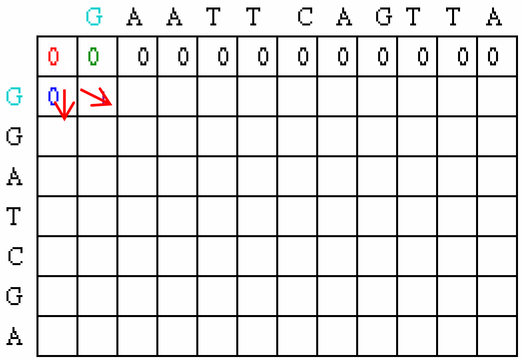
Gap $a_1 \ldots a_i$ / $b_1 \ldots b_j$ - $M_{i,j} = M_{i-1,j} + w$ (gap in sequence #2)

$$M_{1,1} = MAX[M_{0,0} + 1, M_{1, 0} + 0, M_{0,1} + 0] = MAX [1, 0, 0] = 1$$



	G	A	A	T	T	C	A	G	T	T	A
0	0	0	0	0	0	0	0	0	0	0	0
G	0	1									
G											
A											
T											
C											
G											
A											

Score matrix M

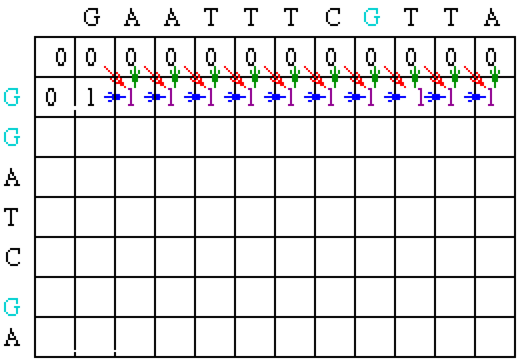


	G	A	A	T	T	C	A	G	T	T	A
0	0	0	0	0	0	0	0	0	0	0	0
G	0 ↓	↘									
G											
A											
T											
C											
G											
A											

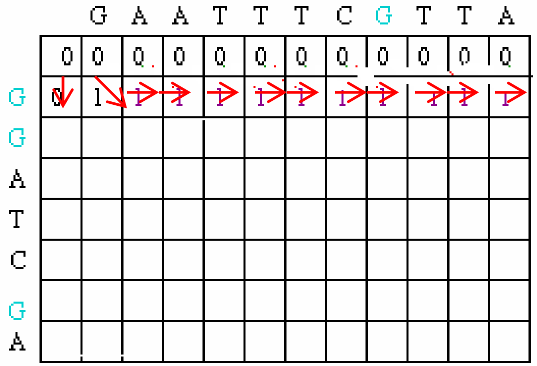
Trace back table B

Longest Common Subsequence

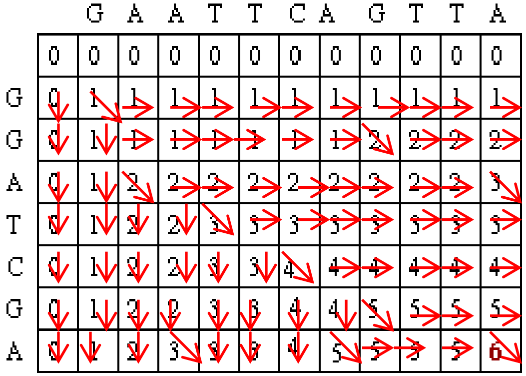
Score matrix M



Trace back table B



		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1	2	2	2	2
A	0	1	2	2	2	2	2	2	2	2	2	3
T	0	1	2	2	3	3	3	3	3	3	3	3
C	0	1	2	2	3	3	4	4	4	4	4	4
G	0	1	2	2	3	3	4	4	5	5	5	5
A	0	1	2	3	3	3	4	5	5	5	5	6



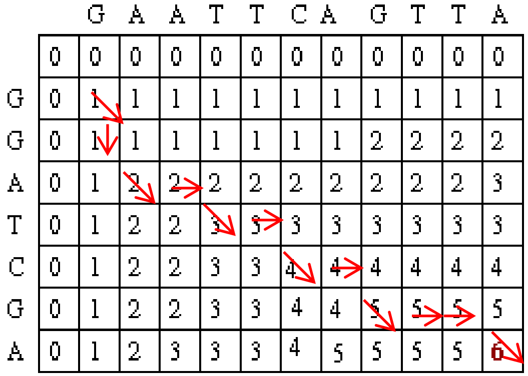
$M_{7,11}$=6 (lower right corner of Score matrix)
This tells us that the best alignment has a score of 6

What is the best alignment?

Longest Common Subsequence

We need to use trace back table to find out the best alignment, which has a score of 6

(1) Find the path from lower right corner to upper left corner



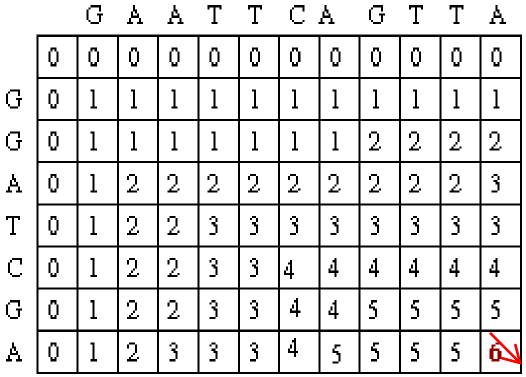
		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1	2	2	2	2
A	0	1	2	2	2	2	2	2	2	2	2	3
T	0	1	2	2	3	3	3	3	3	3	3	3
C	0	1	2	2	3	3	4	4	4	4	4	4
G	0	1	2	2	3	3	4	4	5	5	5	5
A	0	1	2	3	3	3	4	5	5	5	5	6

Longest Common Subsequence

(2) At the same time, write down the alignment backward

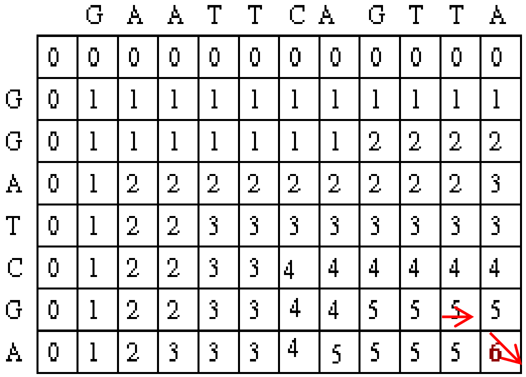
S_1

S_2



		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1	2	2	2	2
A	0	1	2	2	2	2	2	2	2	2	2	3
T	0	1	2	2	3	3	3	3	3	3	3	3
C	0	1	2	2	3	3	4	4	4	4	4	4
G	0	1	2	2	3	3	4	4	5	5	5	5
A	0	1	2	3	3	3	4	5	5	5	5	6 ↘

```
(Seq #1)        A
                |
(Seq #2)        A
```



		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1	2	2	2	2
A	0	1	2	2	2	2	2	2	2	2	2	3
T	0	1	2	2	3	3	3	3	3	3	3	3
C	0	1	2	2	3	3	4	4	4	4	4	4
G	0	1	2	2	3	3	4	4	5	5	5 →	5
A	0	1	2	3	3	3	4	5	5	5	5	6 ↘

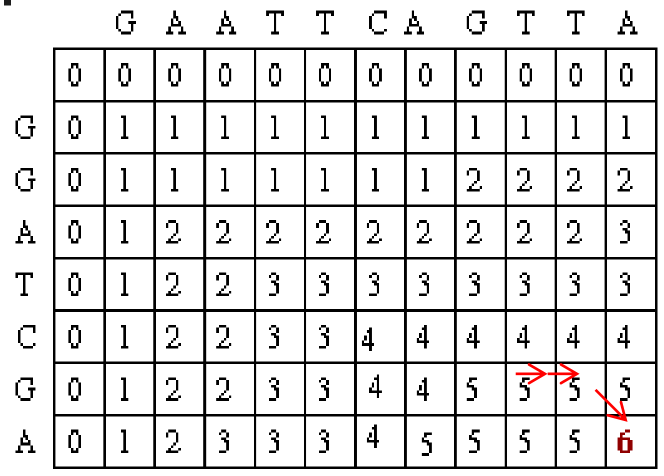
```
(Seq #1)        T  A
                   |
(Seq #2)        _  A
```

↘ :Take one character from each sequence

→:Take one character from sequence S_1 (columns)

↓:Take one character from sequence S_2 (rows)

Longest Common Subsequence



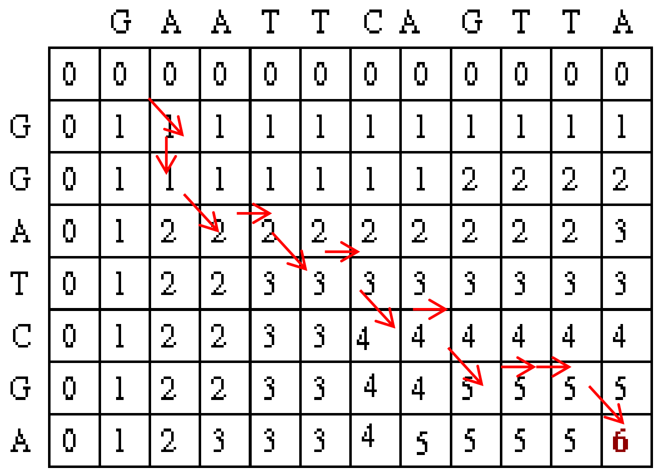
		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1	2	2	2	2
A	0	1	2	2	2	2	2	2	2	2	2	3
T	0	1	2	2	3	3	3	3	3	3	3	3
C	0	1	2	2	3	3	4	4	4	4	4	4
G	0	1	2	2	3	3	4	4	5	5	5	5
A	0	1	2	3	3	3	4	5	5	5	5	6

```
(Seq #1)        T T A
                    |
(Seq #2)        _ _ A
```

↘ :Take one character from each sequence

→ :Take one character from sequence S_1 (columns)

↓ :Take one character from sequence S_2 (rows)

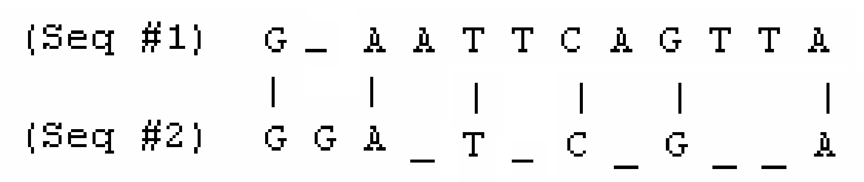


		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1	2	2	2	2
A	0	1	2	2	2	2	2	2	2	2	2	3
T	0	1	2	2	3	3	3	3	3	3	3	3
C	0	1	2	2	3	3	4	4	4	4	4	4
G	0	1	2	2	3	3	4	4	5	5	5	5
A	0	1	2	3	3	3	4	5	5	5	5	6

```
(Seq #1)  G _ A A T T C A G T T A
          |   |   |   |   |     |
(Seq #2)  G G A _ T _ C _ G _ _ A
```

Longest Common Subsequence

Thus, the optimal alignment is



```
(Seq #1)  G _ A A T T C A G T T A
          |   |   |   |   |     |
(Seq #2)  G G A _ T _ C _ G _ _ A
```

The longest common subsequence is
G.A.T.C.G..A

There might be multiple longest common subsequences (LCSs) between two given sequences.

These LCSs have the same number of characters (not include gaps)

Longest Common Subsequence

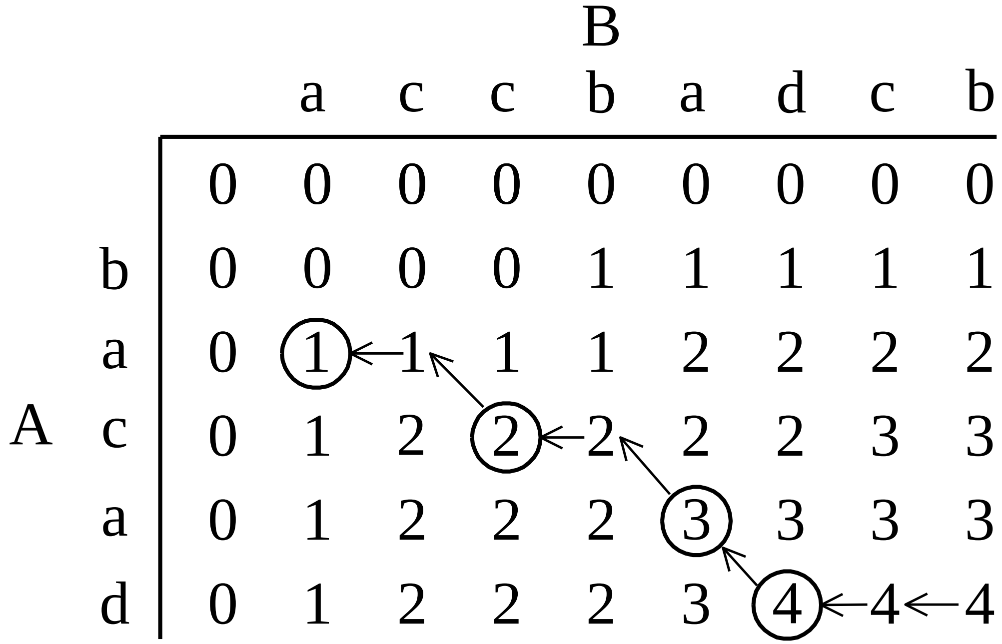
Algorithm LCS (string A, string B) {
Input strings A and B
Output the longest common subsequence of A and B

```
M: Score Matrix
B: trace back table (use letter a, b, c for ↘ → ↓          )
n=A.length()
m=B.length()
// fill in M and B
for (i=0;i<m+1;i++)
    for (j=0;j<n+1;j++)
         if (i==0) || (j==0)
                        then M(i,j)=0;
         else if (A[i]==B[j])
              M(i,j)=max {M[i-1,j-1]+1, M[i-1,j], M[i,j-1]}
             {update the entry in trace table B}
         else
             M(i,j)=max {M[i-1,j-1], M[i-1,j], M[i,j-1]}
            {update the entry in trace table B}

then use trace back table B to print out the optimal alignment
…
```

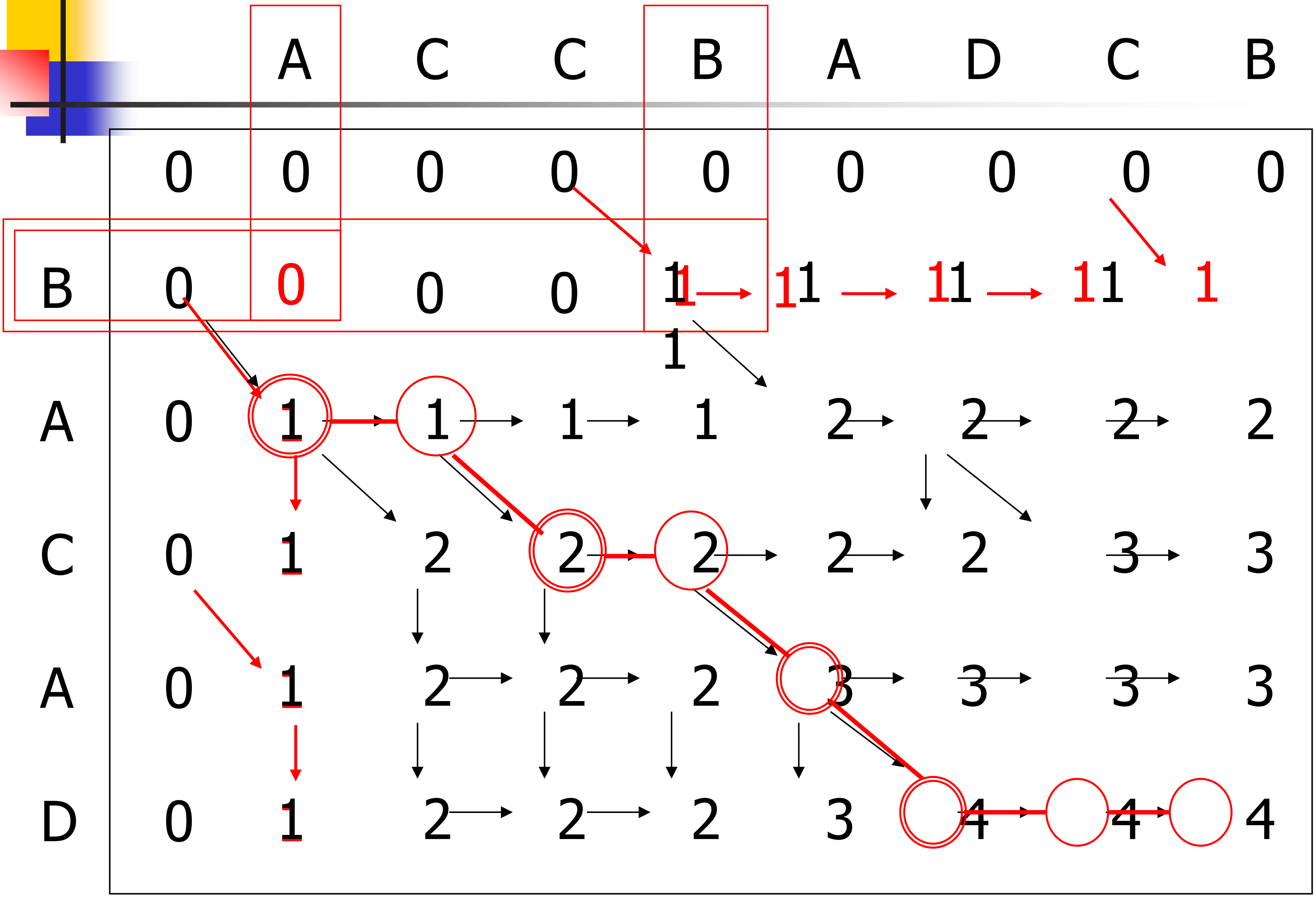
Tracing back in the LCS algorithm

- e.g. A = b a c a d, B = a c c b a d c b



						B				
			a	c	c	b	a	d	c	b
		0	0	0	0	0	0	0	0	0
	b	0	0	0	0	1	1	1	1	1
	a	0	1	1	1	1	2	2	2	2
A	c	0	1	2	2	2	2	2	3	3
	a	0	1	2	2	2	3	3	3	3
	d	0	1	2	2	2	3	4	4	4

- After all $L_{i,j}$'s have been found, we can trace back to find the longest common subsequence of A and B.



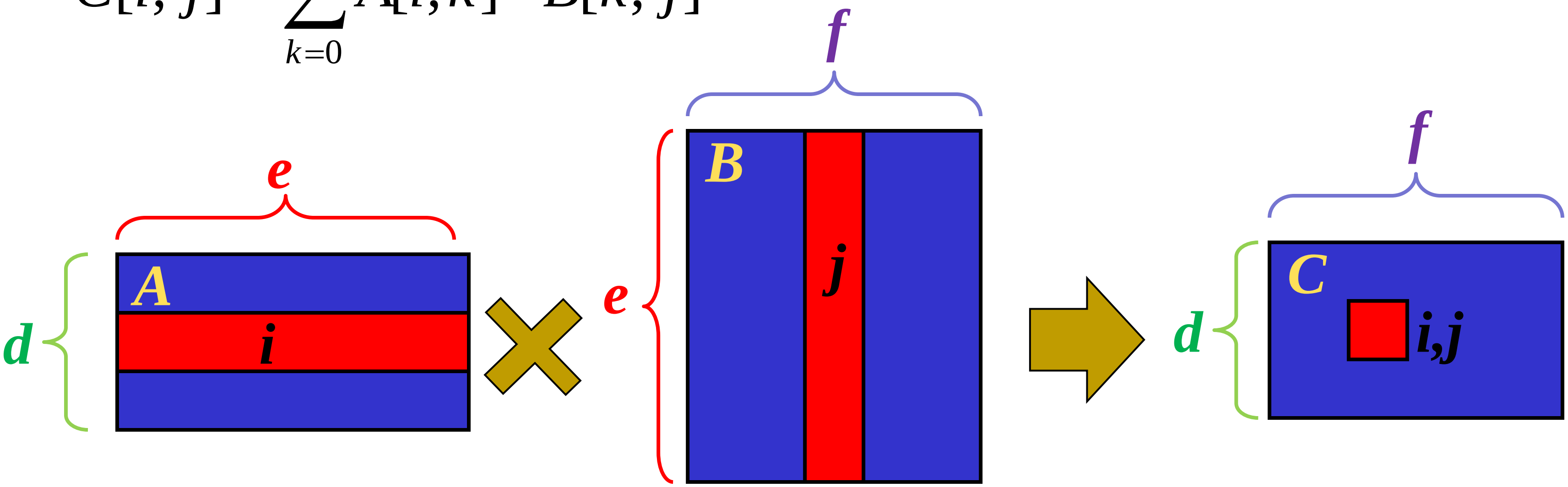
		A	C	C	B	A	D	C	B
	0	0	0	0	0	0	0	0	0
B	0	0	0	0	1	1	1	1	1
A	0	1	1	1	1	2	2	2	2
C	0	1	2	2	2	2	2	3	3
A	0	1	2	2	2	3	3	3	3
D	0	1	2	2	2	3	4	4	4

Matrix-Chain Multiplication

Matrix Chain-Products

- Review: Matrix Multiplication.
  - $C = A*B$
  - A is $d \times e$ and B is $e \times f$
  - computation time $O(d \cdot e \cdot f)$

$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$



Matrix Chain-Products

- **Matrix Chain-Product:**
  - Compute $A=A_0*A_1*\ldots*A_{n-1}$
  - A_i is $d_i \times d_{i+1}$
  - Problem: How to parenthesize?
- n matrices $A_1, A_2, \ldots, A_n$ with size

  $p_0 \times p_1, p_1 \times p_2, p_2 \times p_3, \ldots, p_{n-1} \times p_n$
  To determine the multiplication order such that # of scalar multiplications is minimized.
- To compute $A_i \times A_{i+1}$, we need $p_{i-1}p_ip_{i+1}$ scalar multiplications.

Matrix-chain multiplication

Example

- B is 3 × 100, C is 100 × 5, D is 5 × 5
- (B*C)*D takes 1500 + 75 = 1575 ops
- B*(C*D) takes 1500 + 2500 = 4000 ops

Example

- A_1: 3 × 5, A_2: 5 × 4, A_3: 4 × 2, A_4: 2 × 5 (n=4)
- $((A_1 \times A_2) \times A_3) \times A_4$, # of scalar multiplications:
  $3 * 5 * 4 + 3 * 4 * 2 + 3 * 2 * 5 = 114$
- $(A_1 \times (A_2 \times A_3)) \times A_4$, # of scalar multiplications:
  $3 * 5 * 2 + 5 * 4 * 2 + 3 * 2 * 5 = 100$
- $(A_1 \times A_2) \times (A_3 \times A_4)$, # of scalar multiplications:
  $3 * 5 * 4 + 3 * 4 * 5 + 4 * 2 * 5 = 160$

Enumeration Approach

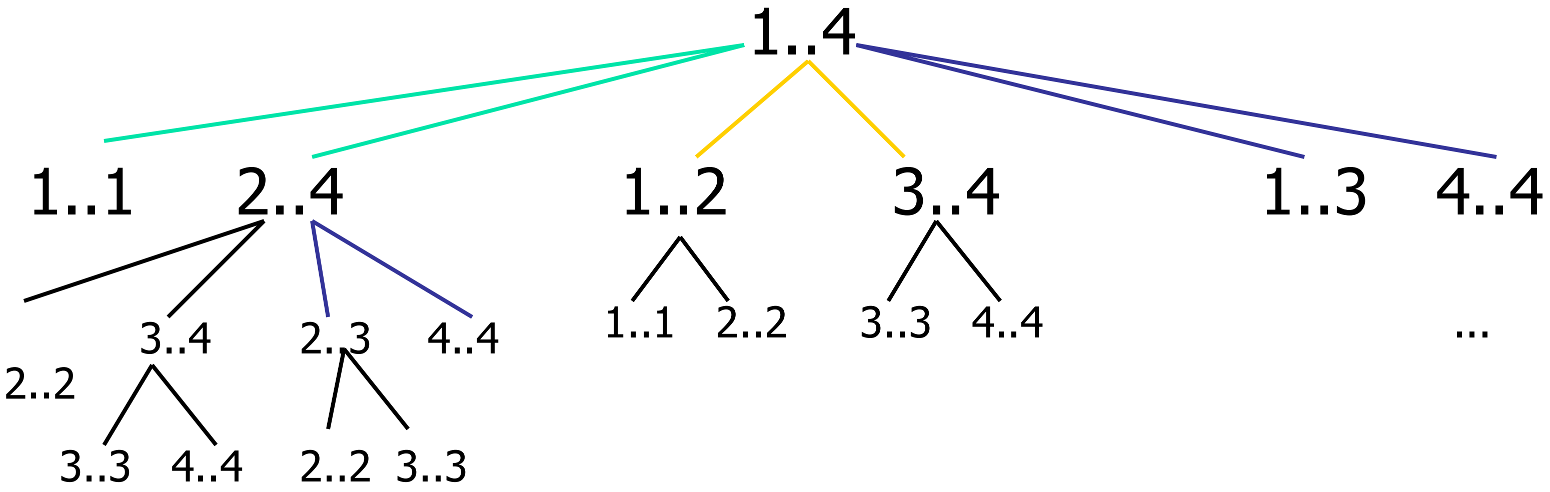
- **Matrix Chain-Product Alg.:**
  - Try all possible ways to parenthesize $A=A_0*A_1*\ldots*A_{n-1}$
  - Calculate number of ops for each one
  - Pick the one that is best
- Running time:
  - The number of parenthesizations is equal to the number of binary trees with n nodes
  - This is **exponential**!
  - It is called the Catalan number, and it is almost 4^n.
  - This is a terrible algorithm!

Subproblem Overlap

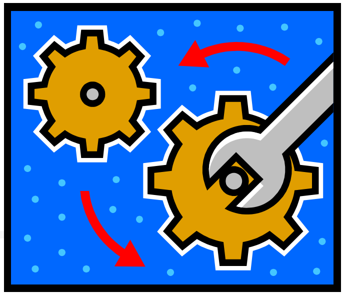
$A_1A_2A_3A_4 \leftarrow (A_1)(A_2A_3A_4)$

or $(A_1A_2)(A_3A_4)$

or $(A_1A_2A_3)(A_4)$



Characterizing Equation



- The global optimal has to be defined in terms of optimal subproblems, depending on where the final multiplication is at.
- Let us consider all possible places for that final multiplication:
  - Recall that A_i is a $d_i \times d_{i+1}$ dimensional matrix.
  - So, a characterizing equation for $N_{i,j}$ is the following:

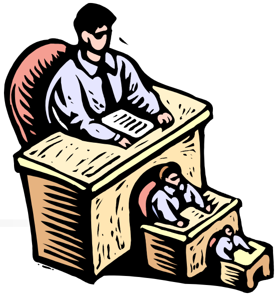
$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

- Note that subproblems are not independent–the **subproblems overlap**.

Subproblem Overlap

```
Algorithm RecursiveMatrixChain(S, i, j):
    Input: sequence S of n matrices to be multiplied
    Output: number of operations in an optimal parenthesization of S
    if i=j
        then return 0
    for k ← i to j do
        $N_{i, j} \leftarrow \min\{N_{i,j}, RecursiveMatrixChain(S, i ,k) + RecursiveMatrixChain(S, k+1,j) + d_i d_{k+1} d_{j+1}\}$
    return $N_{i,j}$
```

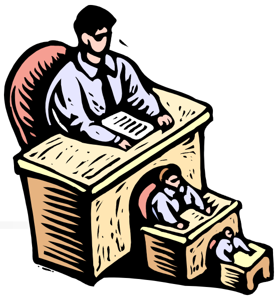
Dynamic Programming Algorithm



- Since subproblems overlap, we don't use recursion.
- Instead, we construct optimal subproblems "bottom-up."
- $N_{i,i}$'s are easy, so start with them
- Then do problems of "length" 2,3,… subproblems, and so on.
- Running time: $O(n^3)$

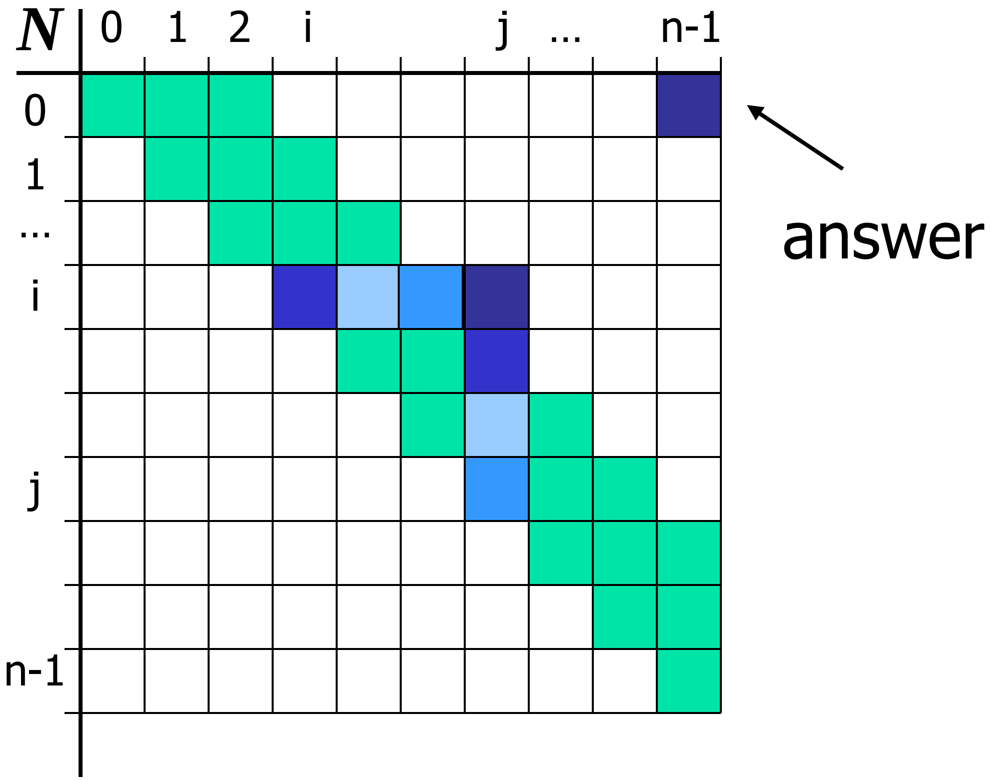
```
Algorithm matrixChain(S):
    Input: sequence S of n matrices to be multiplied
    Output: number of operations in an optimal
        parenthesization of S
    for i ← 1 to n – 1 do
        $N_{i,i} \leftarrow 0$
    for b ← 1 to n – 1 do
        { b = j – i is the length of the problem }
        for i ← 0 to n – b - 1  do
            j ← i + b
            $N_{i,j} \leftarrow +\infty$
            for k ← i to j – 1 do
                $N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i\, d_{k+1}\, d_{j+1}\}$
    return $N_{0,n-1}$
```

Dynamic Programming Algorithm Visualization



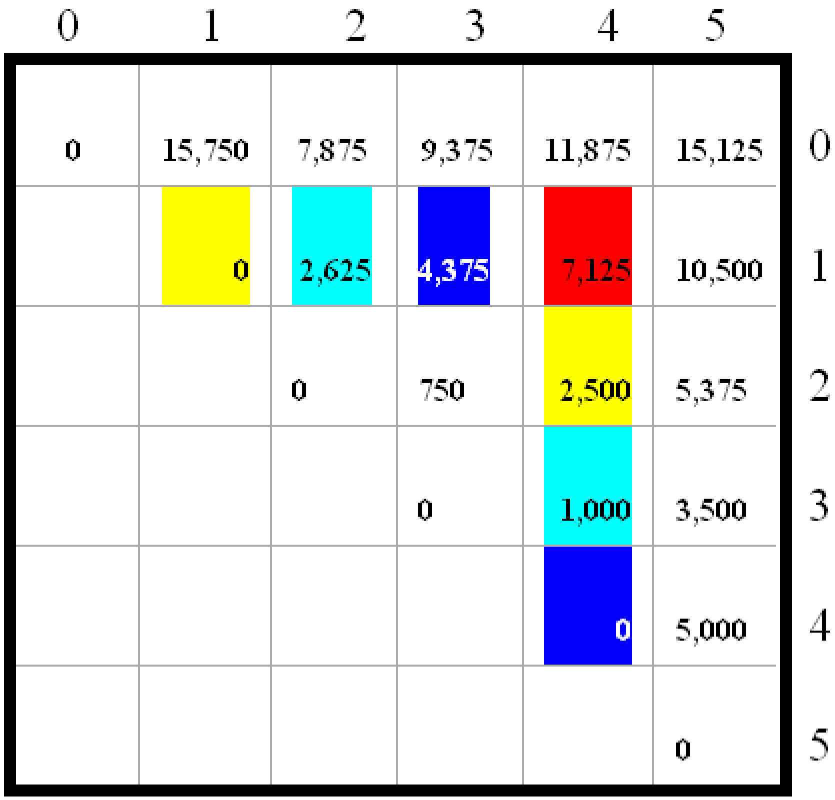
- The bottom-up construction fills in the N array by diagonals
- $N_{i,j}$ gets values from previous entries in i-th row and j-th column
- Filling in each entry in the N table takes O(n) time.
- Total run time: $O(n^3)$
- Getting actual parenthesization can be done by remembering “k” for each N entry

$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$



Dynamic Programming Algorithm Visualization

- A_0: 30 X 35; A_1: 35 X15; A_2: 15X5;
  A_3: 5X10; A_4: 10X20; A_5: 20 X 25



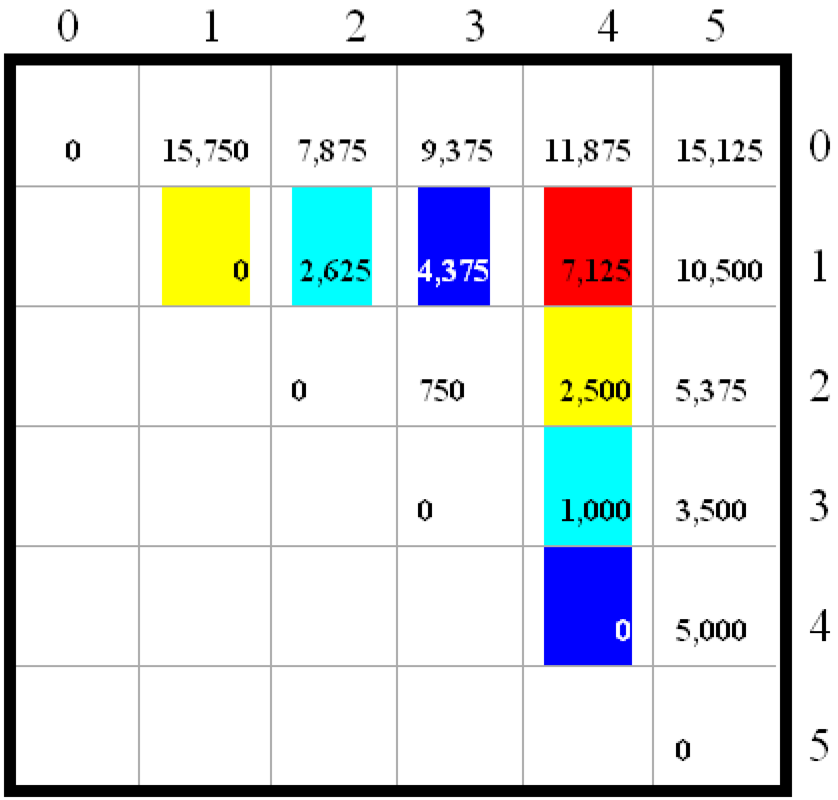
0	1	2	3	4	5	
0	15,750	7,875	9,375	11,875	15,125	0
	0	2,625	4,375	7,125	10,500	1
		0	750	2,500	5,375	2
			0	1,000	3,500	3
				0	5,000	4
					0	5

$$N_{i,j} = \min_{i \le k < j}\{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$$

$$\begin{aligned}
&N_{1,4} = \min\{ \\
&N_{1,1} + N_{2,4} + d_1 d_2 d_5 = 0 + 2500 + 35*15*20 = 13000, \\
&N_{1,2} + N_{3,4} + d_1 d_3 d_5 = 2625 + 1000 + 35*5*20 = 7125, \\
&N_{1,3} + N_{4,4} + d_1 d_4 d_5 = 4375 + 0 + 35*10*20 = 11375 \\
&\} \\
&= 7125
\end{aligned}$$

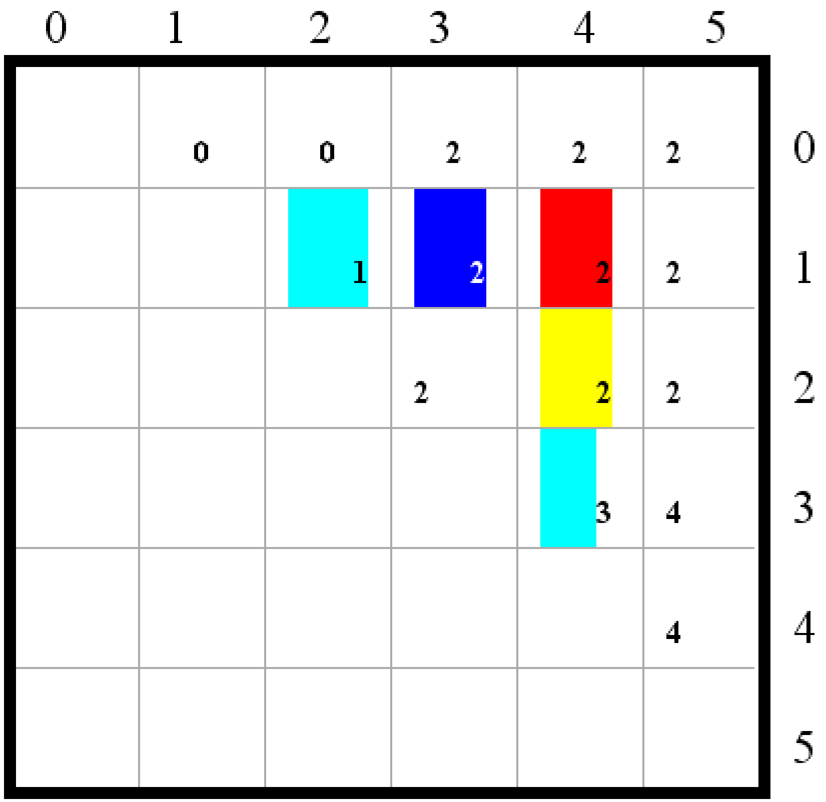
Dynamic Programming Algorithm Visualization

cost



0	1	2	3	4	5	
0	15,750	7,875	9,375	11,875	15,125	0
	0	2,625	4,375	7,125	10,500	1
		0	750	2,500	5,375	2
			0	1,000	3,500	3
				0	5,000	4
					0	5

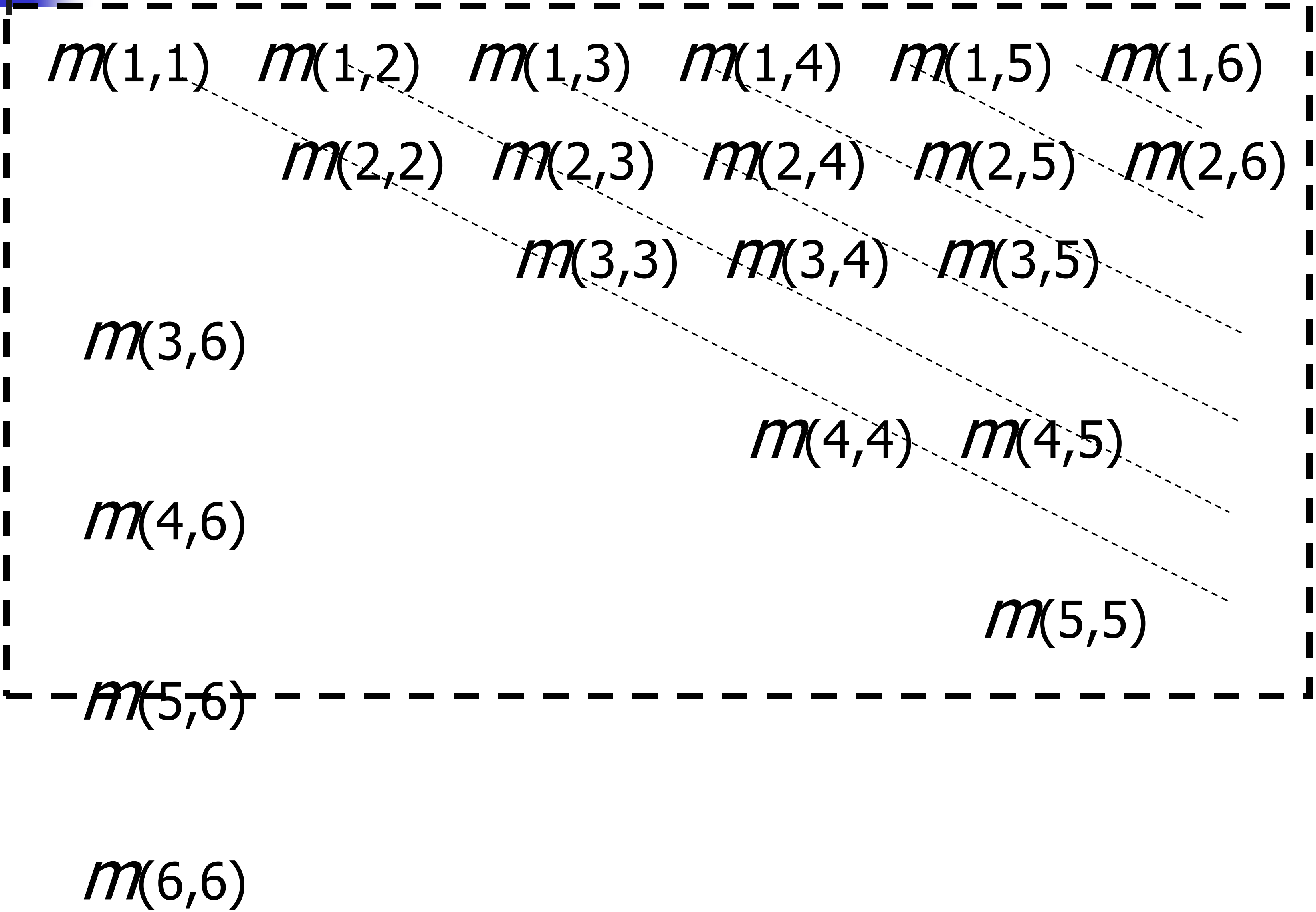
index for trace back



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

$(A_0*(A_1*A_2))*((A_3*A_4)*A_5)$

MCM DP—order of matrix computations



$m(1,1)$ $m(1,2)$ $m(1,3)$ $m(1,4)$ $m(1,5)$ $m(1,6)$

$m(2,2)$ $m(2,3)$ $m(2,4)$ $m(2,5)$ $m(2,6)$

$m(3,3)$ $m(3,4)$ $m(3,5)$

$m(3,6)$

$m(4,4)$ $m(4,5)$

$m(4,6)$

$m(5,5)$

$m(5,6)$

$m(6,6)$

MCM DP—order of matrix computations

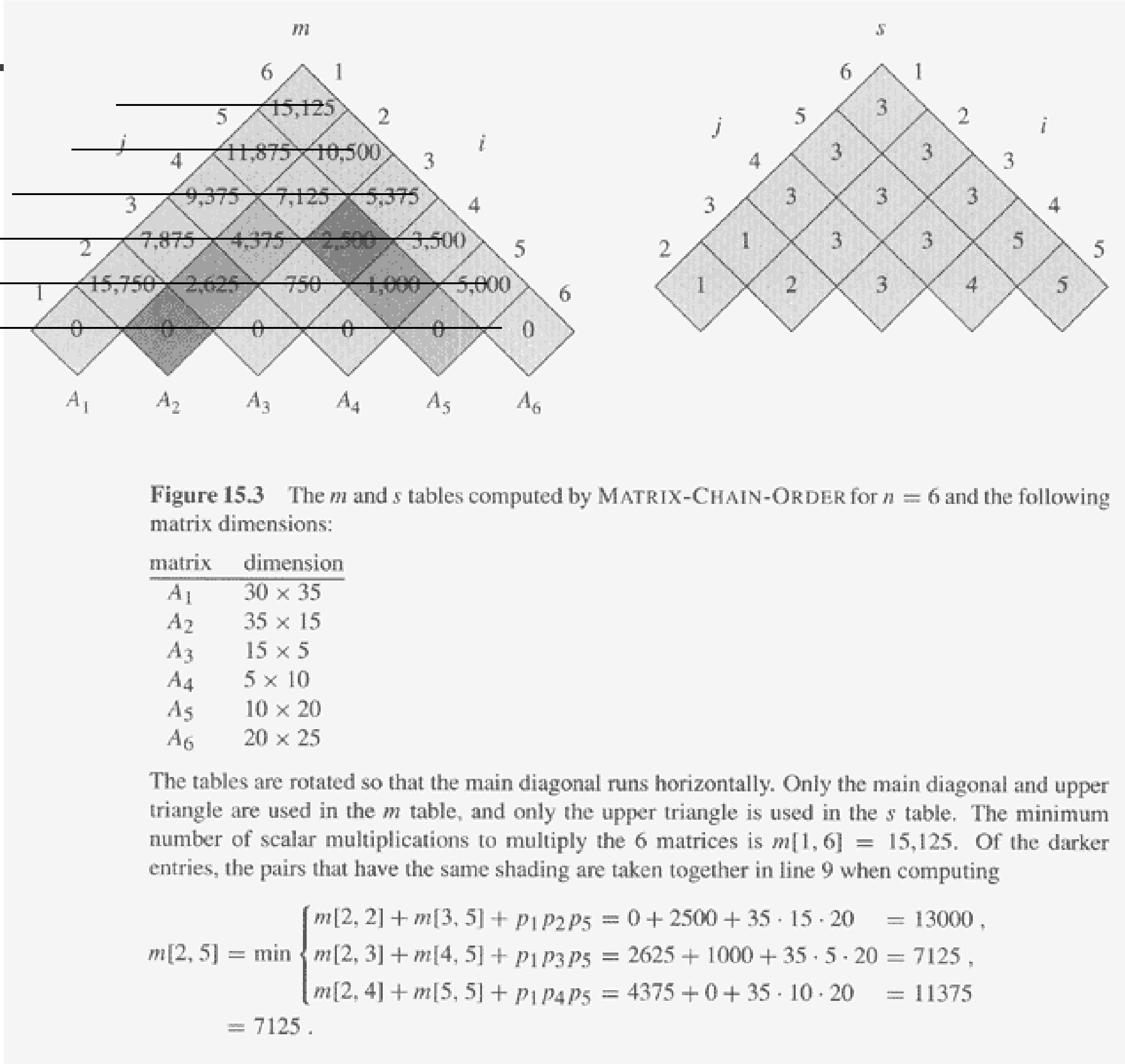


Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the m table, and only the upper triangle is used in the s table. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1,6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

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

0/1 Knapsack Problem

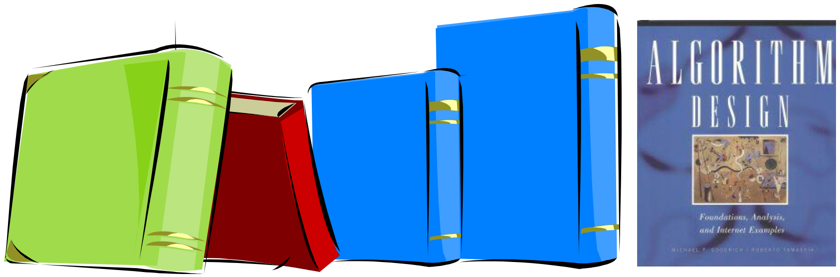
The 0/1 Knapsack Problem

- Given: A set S of n items, with each item i having
  - w_i - a positive weight
  - b_i - a positive benefit
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are **not** allowed to take fractional amounts, then this is the **0/1 knapsack problem**.
  - In this case, we let T denote the set of items we take
  - Objective: maximize $\sum_{i \in T} b_i$
  - Constraint: $\sum_{i \in T} w_i \leq W$

Example

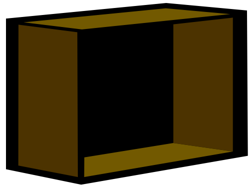
- Given: A set S of n items, with each item i having
  - b_i - a positive “benefit”
  - w_i - a positive “weight”
- Goal: Choose items with maximum total benefit but with weight at most W.

Items:



	1	2	3	4	5
Weight:	4 in	2 in	2 in	6 in	2 in
Benefit:	$20	$3	$6	$25	$80

“knapsack”



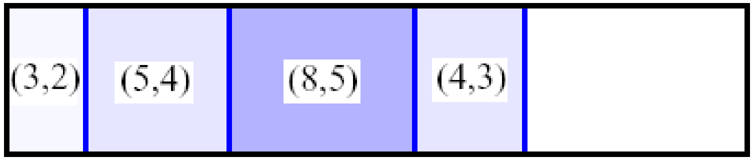
box of width 9 in

Solution:
- item 5 ($80, 2 in)
- item 3 ($6, 2in)
- item 1 ($20, 4in)

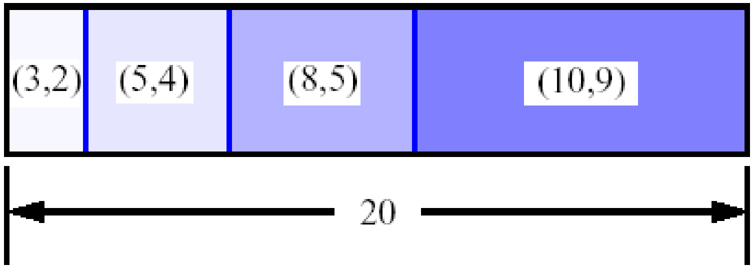
0/1 Knapsack Algorithm, First Attempt

- S_k: Set of items numbered 1 to k.
- Define B[k] = best selection from S_k.
- Problem: does not have subproblem optimality:
  - Consider set S={(3,2),(5,4),(8,5),(4,3),(10,9)} of (benefit, weight) pairs and total weight W = 20

Best for S_4:



Best for S_5:



0/1 Knapsack Algorithm, Second Attempt

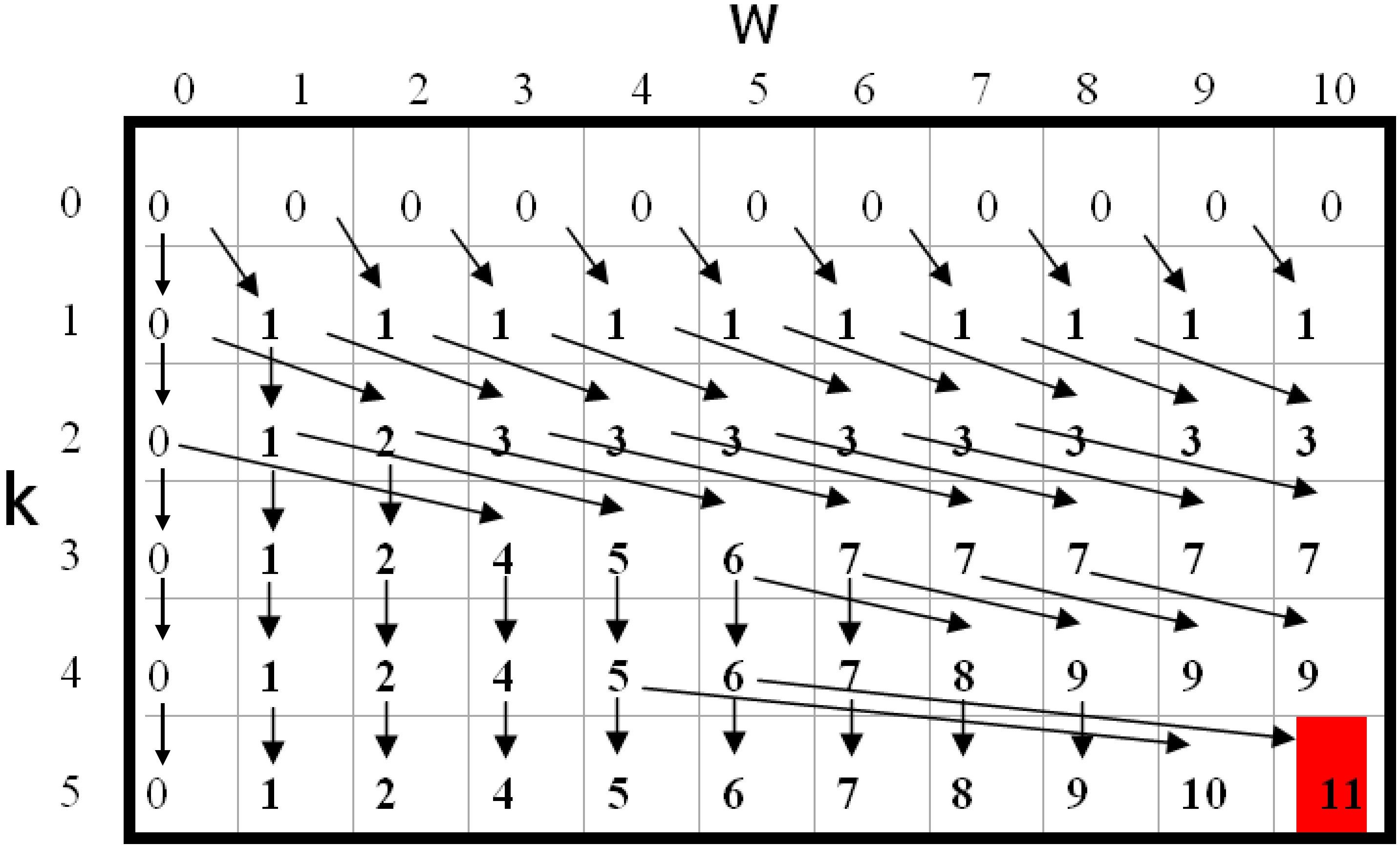
- S_k: Set of items numbered 1 to k.
- Define B[k,w] to be the best selection from S_k with weight at most w
- Good news: this does have subproblem optimality.

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

- I.e., the best subset of S_k with weight at most w is either
  - the best subset of S_{k-1} with weight at most w or
  - the best subset of S_{k-1} with weight at most w–w_k plus item k

0/1 Knapsack Algorithm

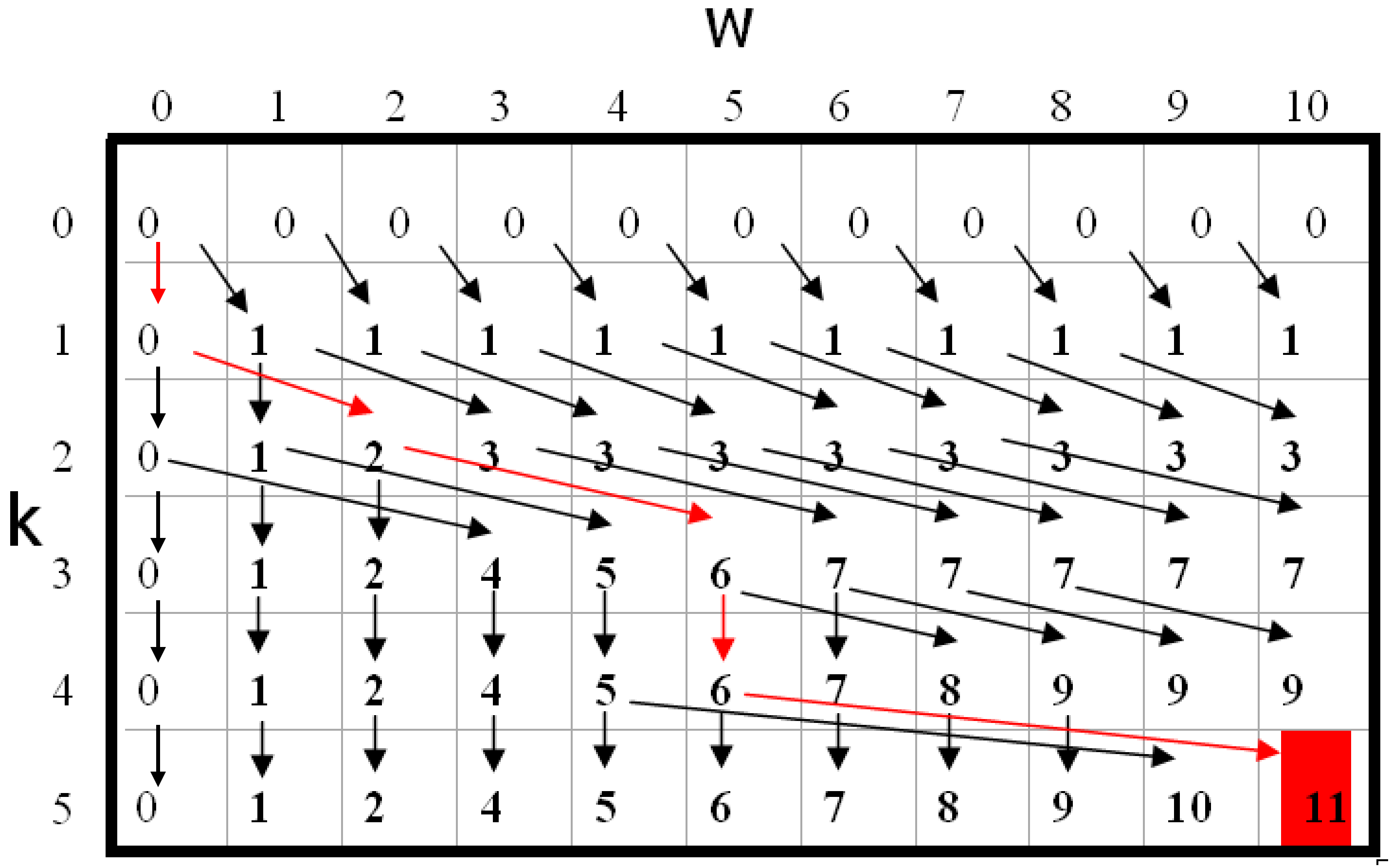
- Consider set S={(1,1),(2,2),(4,3),(2,2),(5,5)} of (benefit, weight) pairs and total weight W = 10



	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1
2	0	1	2	3	3	3	3	3	3	3	3
3	0	1	2	4	5	6	7	7	7	7	7
4	0	1	2	4	5	6	7	8	9	9	9
5	0	1	2	4	5	6	7	8	9	10	11

0/1 Knapsack Algorithm

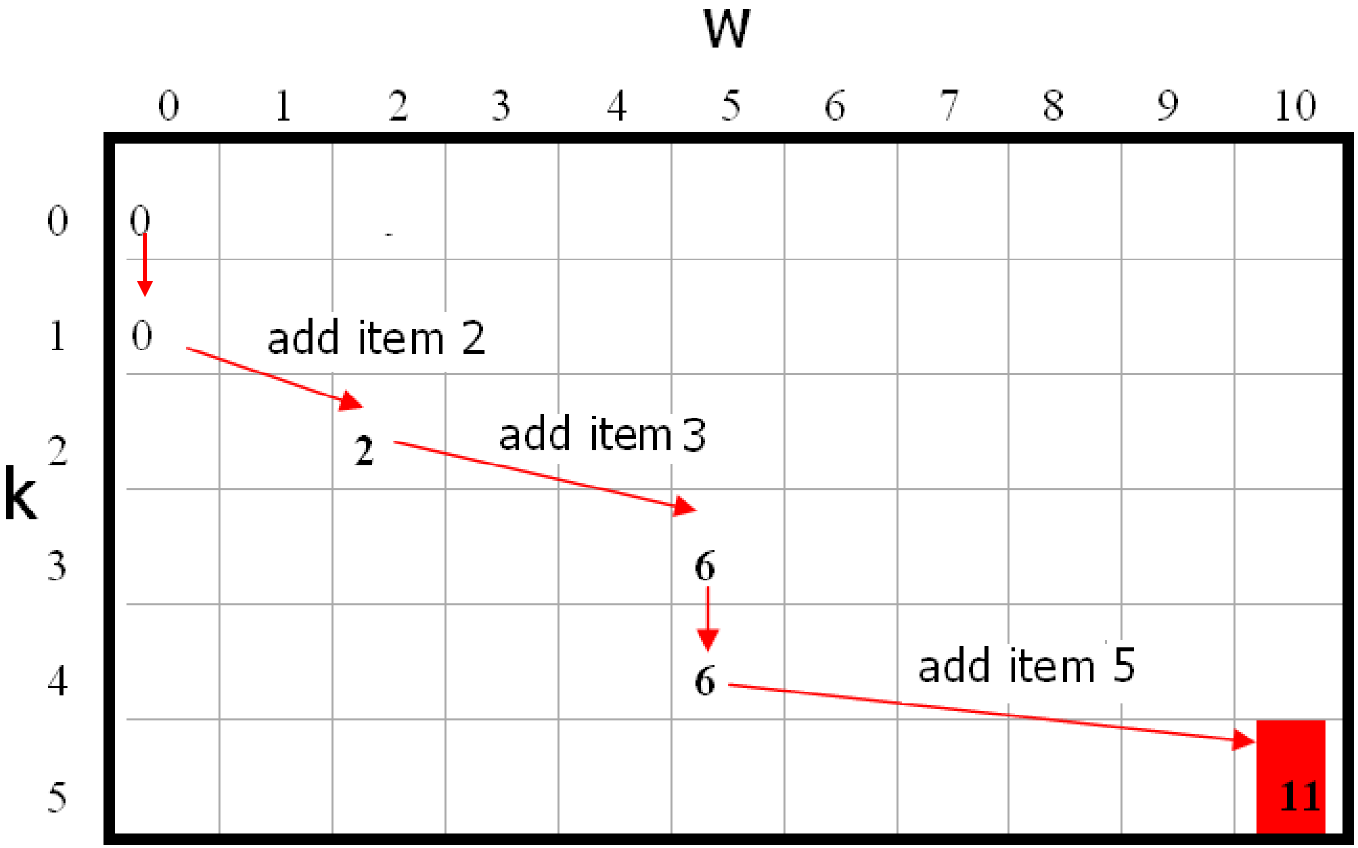
- Trace back to find the items picked



	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1
2	0	1	2	3	3	3	3	3	3	3	3
3	0	1	2	4	5	6	7	7	7	7	7
4	0	1	2	4	5	6	7	8	9	9	9
5	0	1	2	4	5	6	7	8	9	10	11

0/1 Knapsack Algorithm

- Each diagonal arrow corresponds to adding one item into the bag
- Pick items 2,3,5
- {(2,2),(4,3),(5,5)} are what you will take away



0/1 Knapsack Algorithm

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

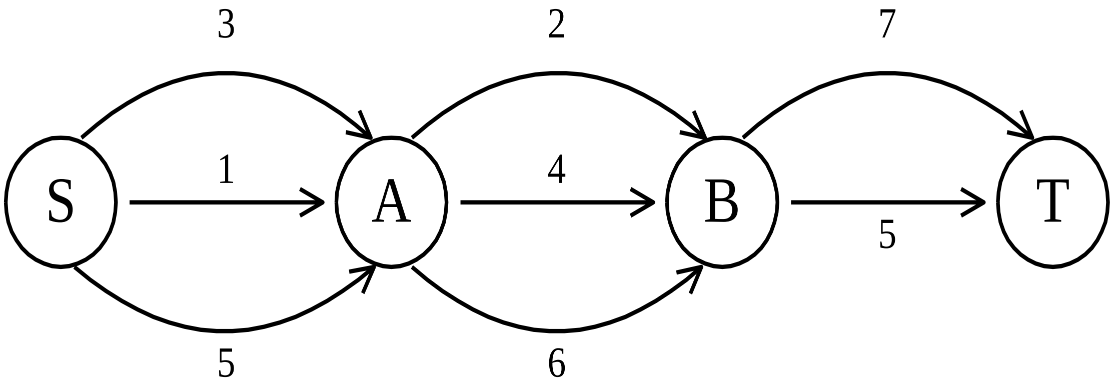
- Recall the definition of B[k,w]
- Since B[k,w] is defined in terms of B[k–1,*], we can use two arrays of instead of a matrix
- Running time: O(nW).
- Not a polynomial-time algorithm since W may be large
- This is a pseudo-polynomial time algorithm

```
Algorithm 01Knapsack(S, W):
    Input: set S of n items with benefit b_i
           and weight w_i; maximum weight W
    Output: benefit of best subset of S with
           weight at most W
    let A and B be arrays of length W + 1
    for w ← 0 to W do
        B[w] ← 0
    for k ← 1 to n do
        copy array B into array A
        for w ← w_k to W do
            if A[w−w_k] + b_k > A[w] then
                B[w] ← A[w−w_k] + b_k
    return B[W]
```

All-Pair Shortest Path

The shortest path

- To find a shortest path in a multi-stage graph



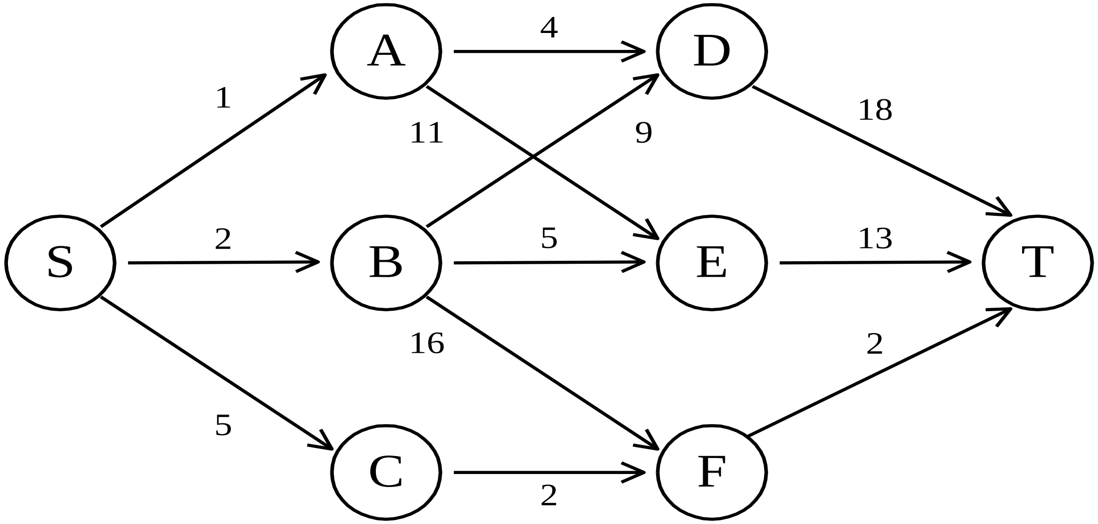
- Apply the greedy method :

  the shortest path from S to T :

  1 + 2 + 5 = 8

The shortest path in multistage graphs

- e.g.

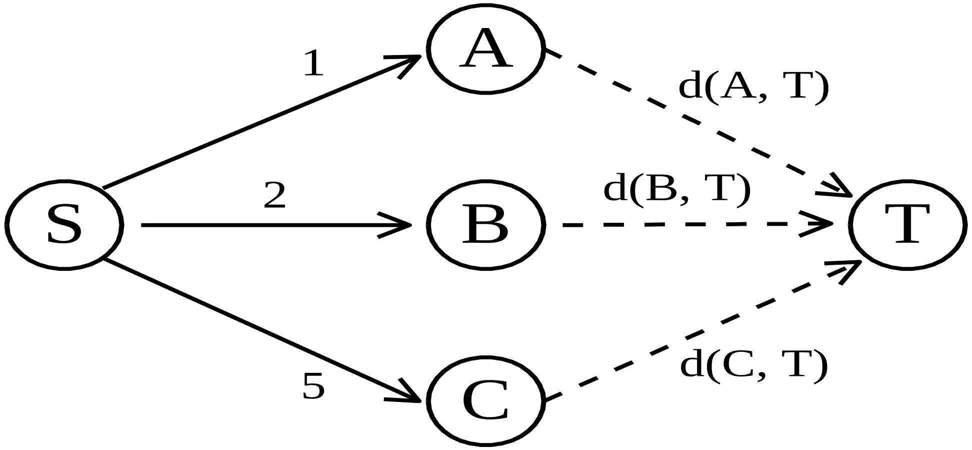


- The greedy method can not be applied to this case: (S, A, D, T) 1+4+18 = 23.
- The real shortest path is:

  (S, C, F, T) 5+2+2 = 9.

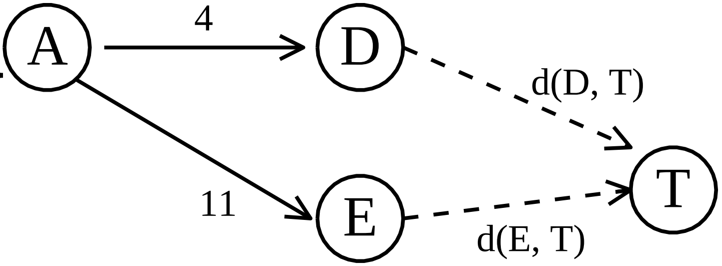
Dynamic programming approach

- Dynamic programming approach (forward approach):



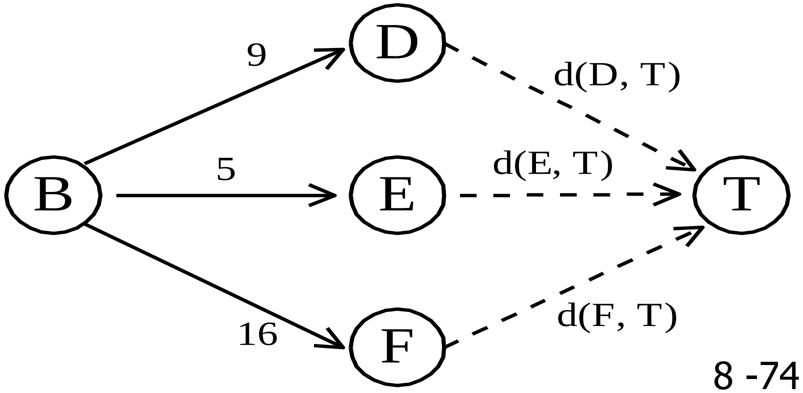
- d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}

- d(A,T) = min{4+d(D,T), 11+d(E,T)}
= min{4+18, 11+13} = 22.



Dynamic programming

- $d(B, T) = \min\{9+d(D, T), 5+d(E, T), 16+d(F, T)\}$
  $= \min\{9+18, 5+13, 16+2\} = 18.$
- $d(C, T) = \min\{ 2+d(F, T) \} = 2+2 = 4$
- $d(S, T) = \min\{1+d(A, T), 2+d(B, T), 5+d(C, T)\}$
  $= \min\{1+22, 2+18, 5+4\} = 9.$
- The above way of reasoning is called backward reasoning.



Backward approach

- d(S, A) = 1
  d(S, B) = 2
  d(S, C) = 5
- d(S,D)=min{d(S, A)+d(A, D),d(S, B)+d(B, D)}
  = min{ 1+4, 2+9 } = 5
  d(S,E)=min{d(S, A)+d(A, E),d(S, B)+d(B, E)}
  = min{ 1+11, 2+5 } = 7
  d(S,F)=min{d(S, A)+d(A, F),d(S, B)+d(B, F)}
  = min{ 2+16, 5+2 } = 7

Backward approach

- $d(S,T) = \min\{d(S, D)+d(D, T), d(S,E)+ d(E,T), d(S, F)+d(F, T)\}$

  $= \min\{ 5+18, 7+13, 7+2 \}$

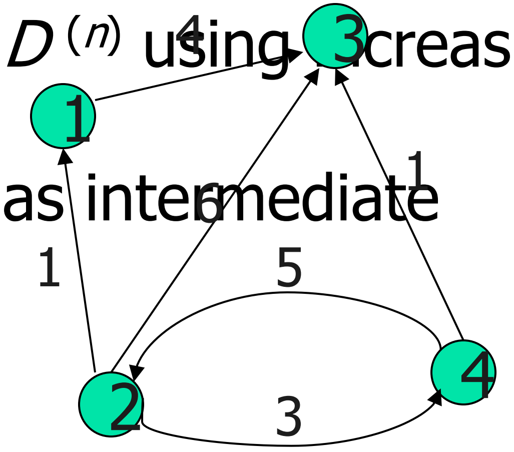
  $= 9$

Floyd's Algorithm:
All pairs shortest paths

Problem: In a weighted (di)graph, find shortest paths between every pair of vertices

Same idea: construct solution through series of matrices $D^{(0)}$, …, $D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

Example:

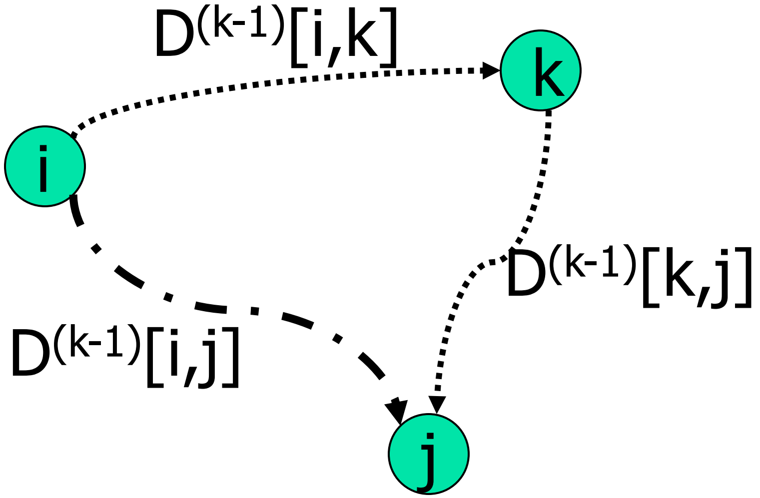


$$\begin{matrix} 0 & \infty & 4 & \infty \\ 1 & 0 & 4 & 3 \\ \infty & \infty & 0 & \infty \\ 6 & 5 & 1 & 0 \end{matrix}$$

Floyd's Algorithm (matrix generation)

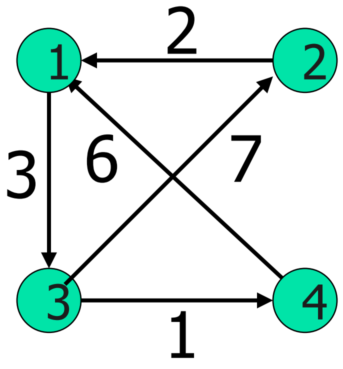
On the k-th iteration, the algorithm determines shortest paths between every pair of vertices i, j that use only vertices among 1,…,k as intermediate

$$D^{(k)}[i,j] = \min \{D^{(k-1)}[i,j],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$



Initial condition?

Floyd’s Algorithm (example)



$D^{(0)}$ =

0 ∞ 3
∞ 2 0
∞ ∞
∞ 7 0 1
6 ∞ ∞
0

$D^{(1)}$ =

0 ∞ 3 ∞
2 0 **5**
∞
∞ 7 0 1
6 ∞ **9** 0

$D^{(2)}$ =

0 ∞ 3 ∞
2 0 5 ∞
9 7 0 1
6 ∞ 9 0

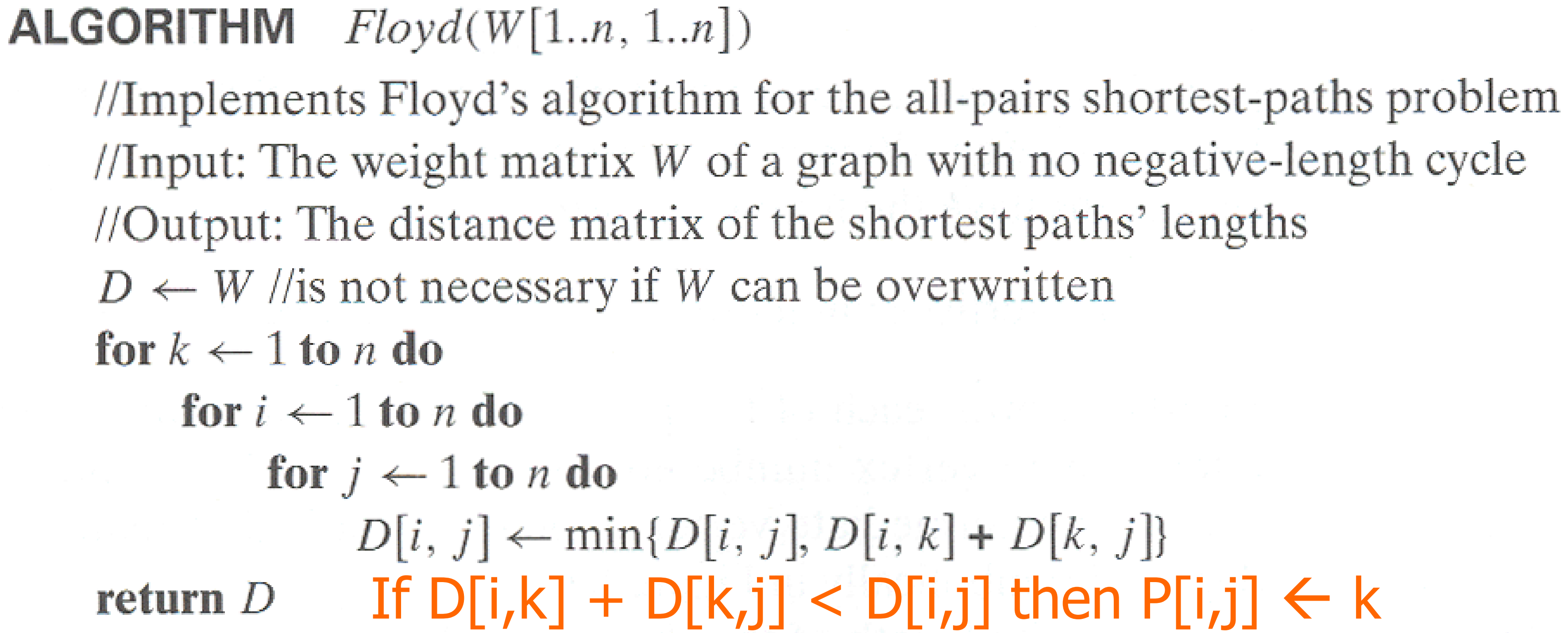
$D^{(3)}$ =

0 **10** 3 **4**
2 0 5 **6**
9 7 0 1
6 **16** 9 0

$D^{(4)}$ =

0 10 3 4
2 0 5 6
7 7 0 1
6 16 9 0

Floyd’s Algorithm (pseudocode and analysis)



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

If D[i,k] + D[k,j] < D[i,j] then P[i,j] ← k

Since the superscripts k or k-1 make no difference to D[i,k] and D[k,j].

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

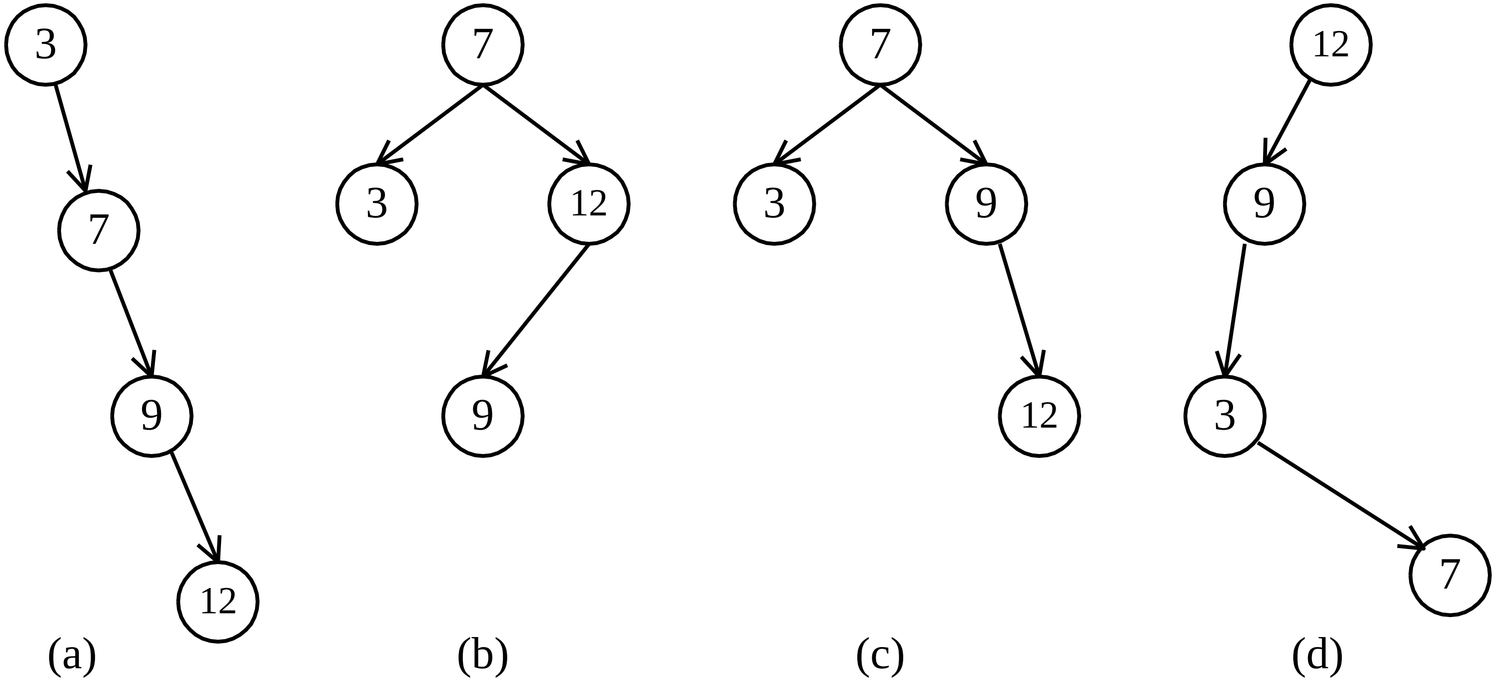
Space efficiency: Matrices can be written over their predecessors

Note: Works on graphs with negative edges but without negative cycles. Shortest paths themselves can be found, too. How?

Optimal Binary Search Tree

Optimal binary search trees

- e.g. binary search trees for 3, 7, 9, 12;

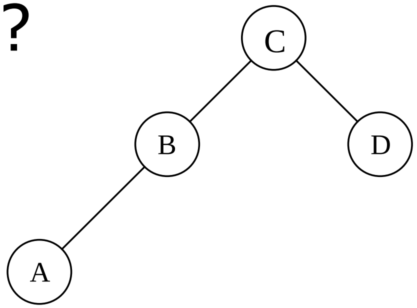


Optimal Binary Search Trees

Problem: Given n keys $a_1 < \ldots < a_n$ and probabilities $p_1, \ldots, p_n$ searching for them, find a BST with a minimum average number of comparisons in successful search.

Since total number of BSTs with n nodes is given by $C(2n,n)/(n+1)$, which grows exponentially, brute force is hopeless.

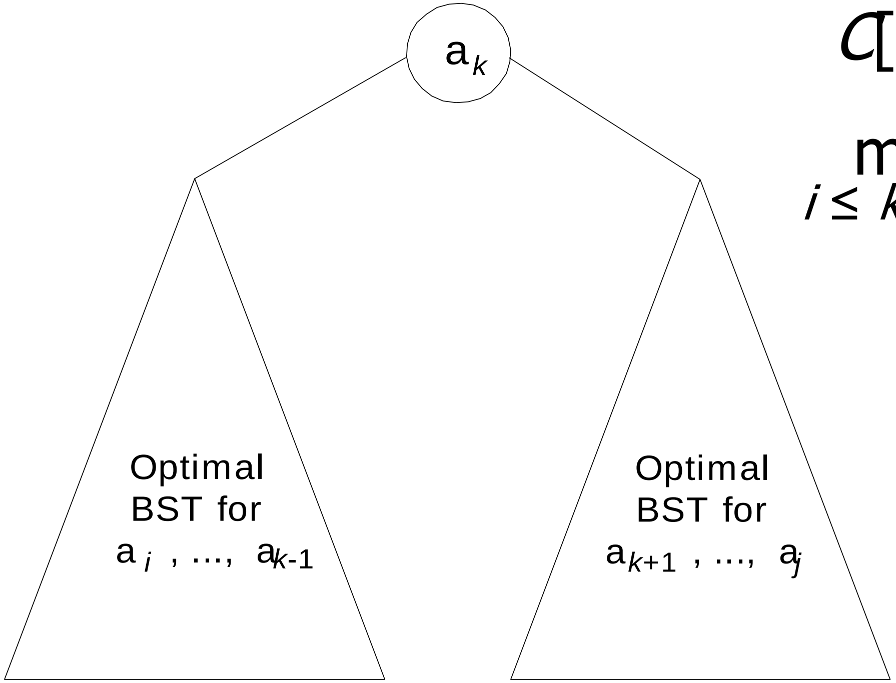
Example: What is an optimal BST for keys A, B, C, and D with search probabilities 0.1, 0.2, 0.4, and 0.3, respectively?



Average # of comparisons $= 1*0.4 + 2*(0.2+0.3) + 3*0.1 = 1.7$

DP for Optimal BST Problem

Let C[i,j] be minimum average number of comparisons made in tree T[i,j], optimal BST for keys $a_i < \ldots < a_j$, where $1 \le i \le j \le n$. Consider optimal BST among all BSTs with some a_k ($i \le k \le j$) as their root; T[i,j] is the best among them.



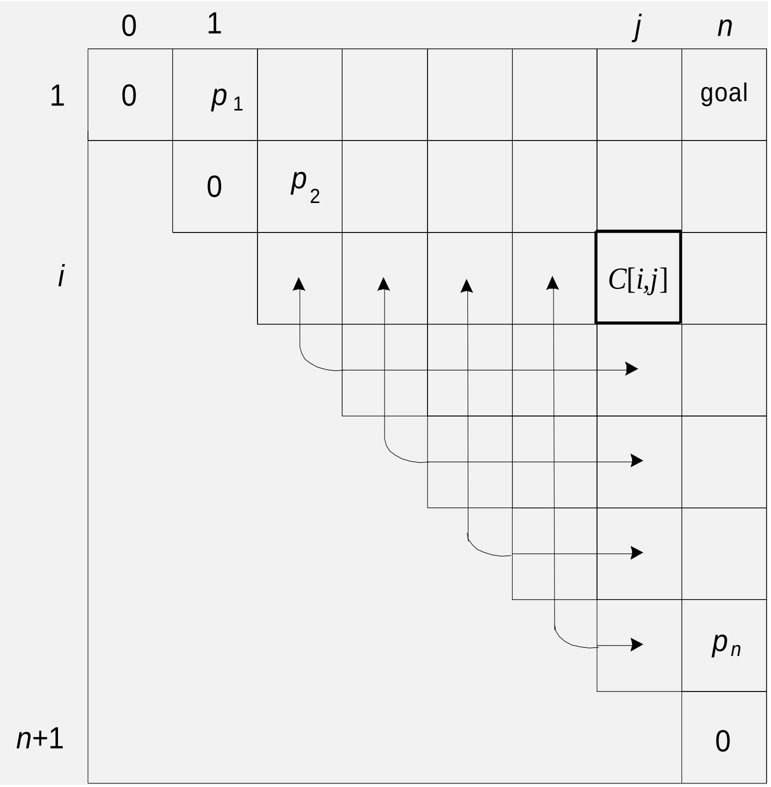
$$C[i,j] = \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \,(\text{level } a_s \text{ in } T[i,k-1] + 1) + \sum_{s=k+1}^{j} p_s \,(\text{level } a_s \text{ in } T[k+1,j] + 1)\}$$

DP for Optimal BST Problem

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

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

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



Example:				
key	*A*	*B*	*C*	*D*
probability	0.1	0.2	0.4	0.3

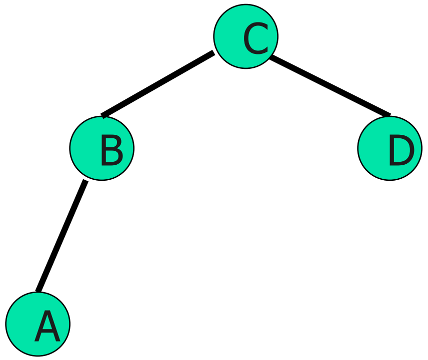
The tables below are filled diagonal by diagonal: the left one is filled using the recurrence

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

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

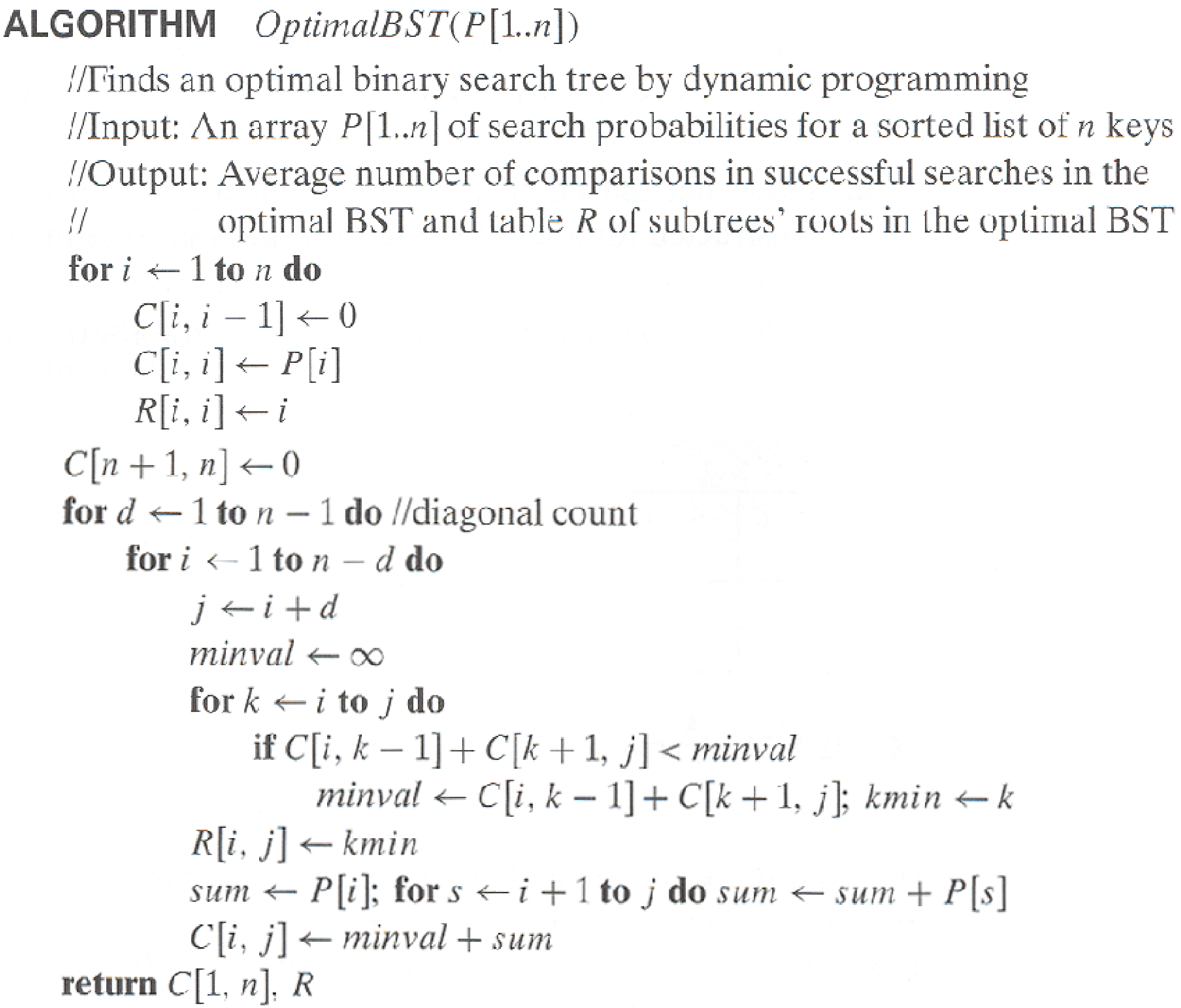
i \ *j*	0	1	2	3	4
1	**0**	**.1**	**.4**	**1.1**	**1.7**
2		**0**	**.2**	**.8**	**1.4**
3			**0**	**.4**	**1.0**
4				**0**	**.3**
5					**0**

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



optimal BST

Optimal Binary Search Trees



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

Problem

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

Space efficiency: $\Theta(n^2)$

Method can be expanded to include unsuccessful searches

Town Matching Problem

Dynamic Programming

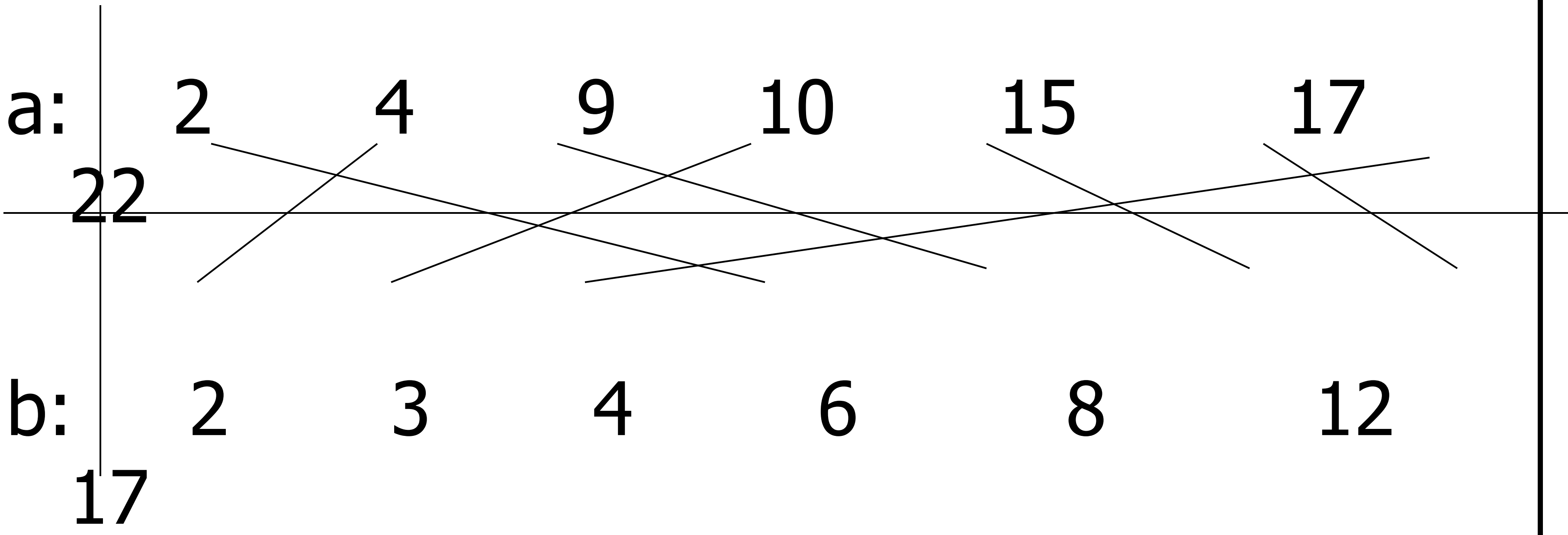
- Problem:
  - The Palmia country is divided by a river into the north and south bank. There are N towns on both the north and south bank. Each town on the north bank has its unique friend town on the south bank. No two towns have the same friend. Each pair of friend towns would like to have a ship line connecting them. They applied for permission to the government. Because it is often foggy on the river the government decided to prohibit intersection of ship lines (if two lines intersect there is a high probability of ship crash).

Dynamic Programming

- Problem (in mathematical terms)
  - There exists a sequences $a_{1,}$, a_2,… a_n and $b_{1,}$,b_2,… b_n such that
    $a_s < a_j$ if s<j
    $b_s < b_j$ if s<j
    for each $s_,$ there exists a s' such that a_s and $b_{s'}$ connected.
  - Find the **maximum** value m such that there exist $x_1 < x_2 < x_3 < \ldots x_{m.}$ m<=n such that
  - $b_{x0'} < b_{x1'} < b_{x2'} < \ldots. b_{xm'}$

Dynamic Programming

- Example Problem

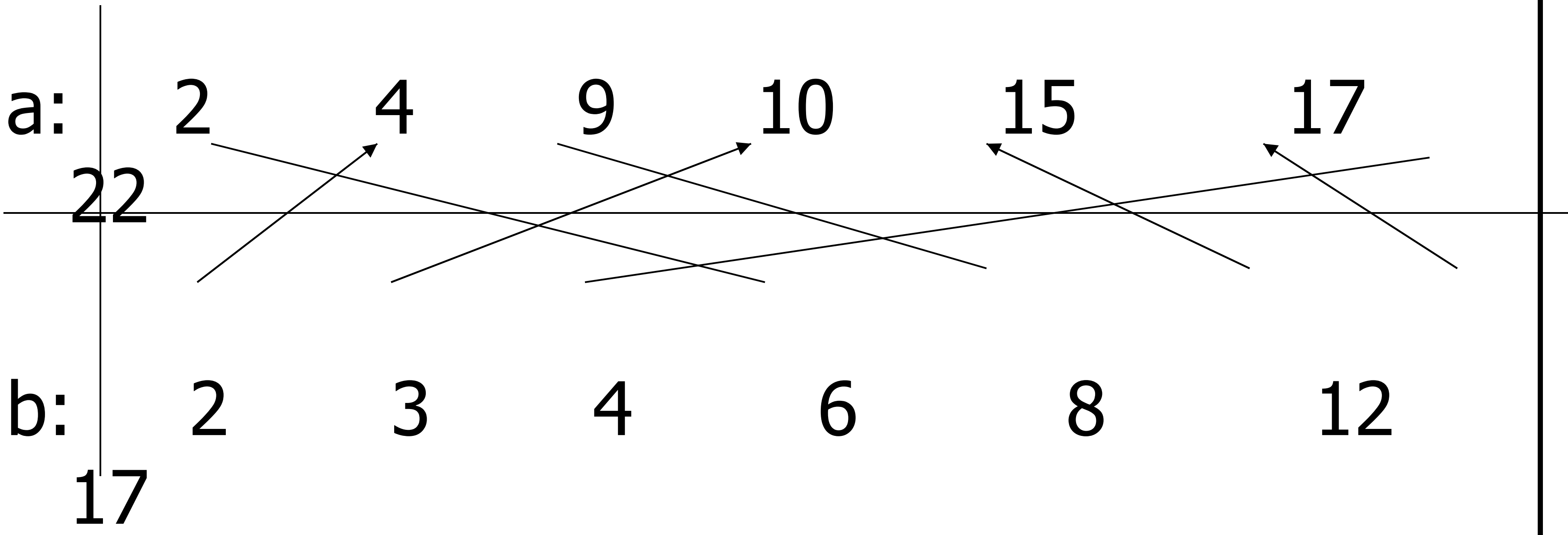


a: 2 4 9 10 15 17 22

b: 2 3 4 6 8 12 17

Dynamic Programming

- Example Problem



a: 2 4 9 10 15 17 22

b: 2 3 4 6 8 12 17

Computing Binomial Coefficient

Binomial Coefficient by DP

Binomial coefficients are coefficients of the binomial formula:

$(a + b)^n = C(n,0)a^nb^0 + . . . + C(n,k)a^{n-k}b^k + . . . + C(n,n)a^0b^n$

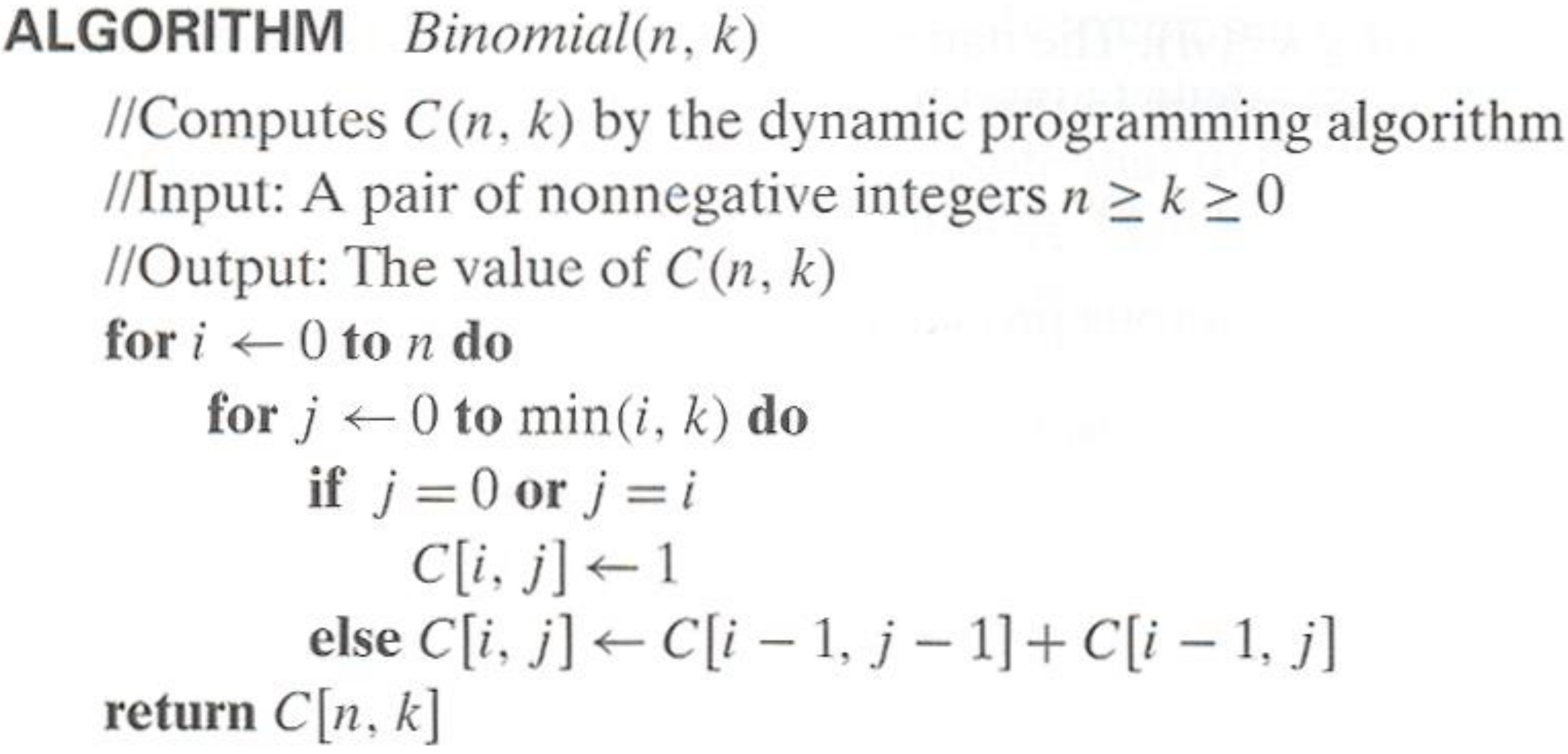
Recurrence: $C(n,k) = C(n-1,k) + C(n-1,k-1)$ for $n > k > 0$

$C(n,0) = 1$, $C(n,n) = 1$ for $n \geq 0$

Value of $C(n,k)$ can be computed by filling a table:

	0	1	2	. . .	k-1	k
0	1					
1	1	1				
.						
.						
.						
n-1					C(n-1,k-1)	C(n-1,k)
n						C(n,k)

Computing C(n,k): pseudocode



```
ALGORITHM Binomial(n, k)
    //Computes C(n, k) by the dynamic programming algorithm
    //Input: A pair of nonnegative integers $n \ge k \ge 0$
    //Output: The value of C(n, k)
    for $i \leftarrow 0$ to $n$ do
        for $j \leftarrow 0$ to $\min(i, k)$ do
            if $j = 0$ or $j = i$
                $C[i, j] \leftarrow 1$
            else $C[i, j] \leftarrow C[i-1, j-1] + C[i-1, j]$
    return $C[n, k]$
```

Time efficiency: $\Theta(nk)$

Space efficiency: $\Theta(nk)$

Dynamic Programming

- **Simple Brute Force Algorithm (Analysis)**
- Pick all possible sets of routes. …. O($2^{n)}$ possible routes
- Check if selection of routes is valid… O(n) time to check
- In total will take O(2^n * n) time.
- EXPONENTIAL TIME = SLOW!!!!!

Dynamic Programming

- Elegant Solution
- Let c(i,j) be the maximum possible routes using the first *i* sequences in a, and j sequences in b.
- We notice that c(I,j) is connected to c(i-1,j), c(i,j-1) and c(i-1,j-1).

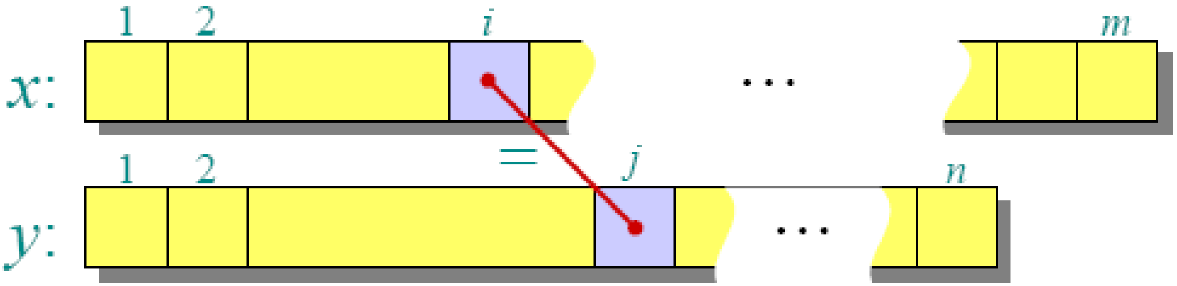
Dynamic Programming

Recursive formulation

Theorem.

$$c[i,j] = \begin{cases} c[i-1,j-1] + 1 & \text{if } a[i] = b[j], \\ \max\{c[i-1,j], c[i,j-1]\} & \end{cases}$$

Proof. Case $a[i] = b[j]$:

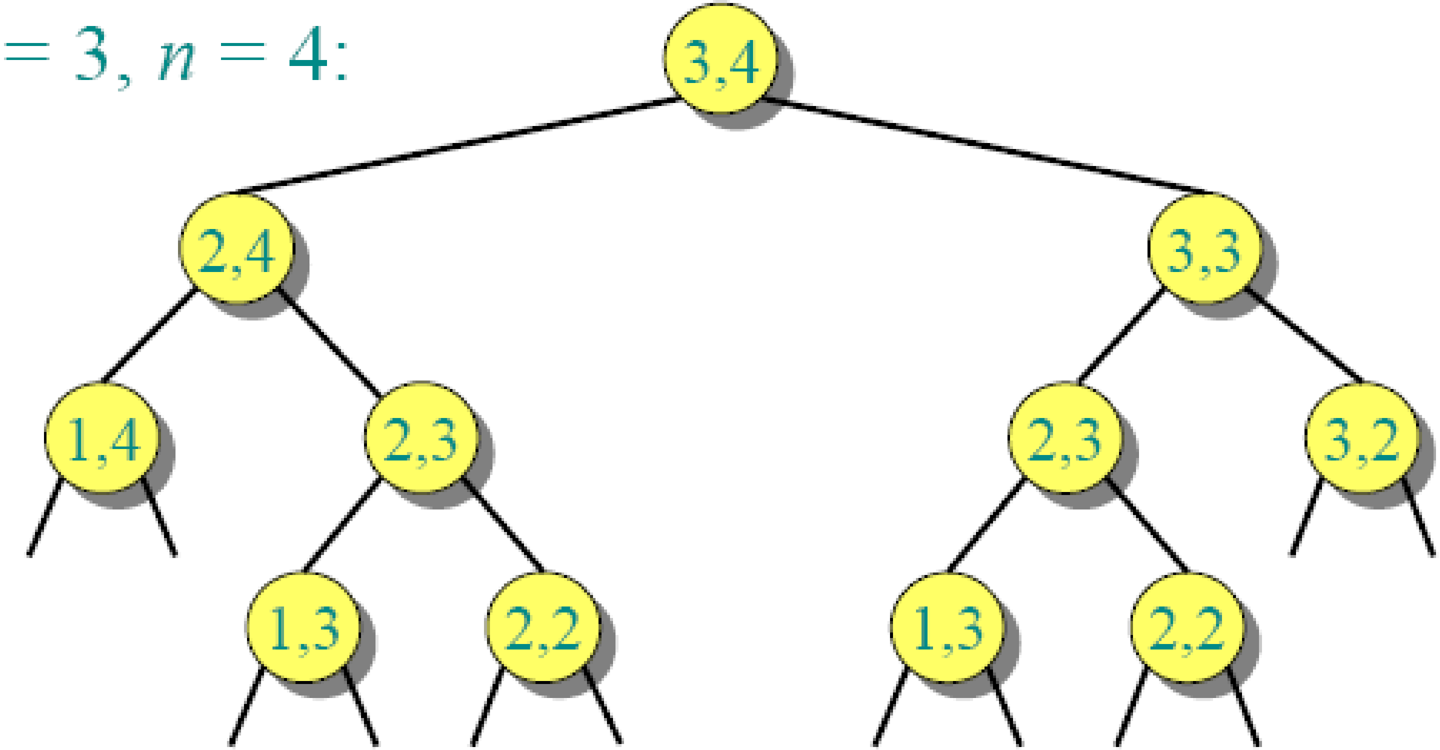


We notice that if a[i] = b[j] (connected), then we can use them in our solution
optimal solution becomes c[i-1,j-1]+1(because we just used a new route).
In all other cases, we just consider the case where we do not use b[j] (i.e c[i,j
or we do not use a[i] (i.e c[i-1,j]

Dynamic Programming

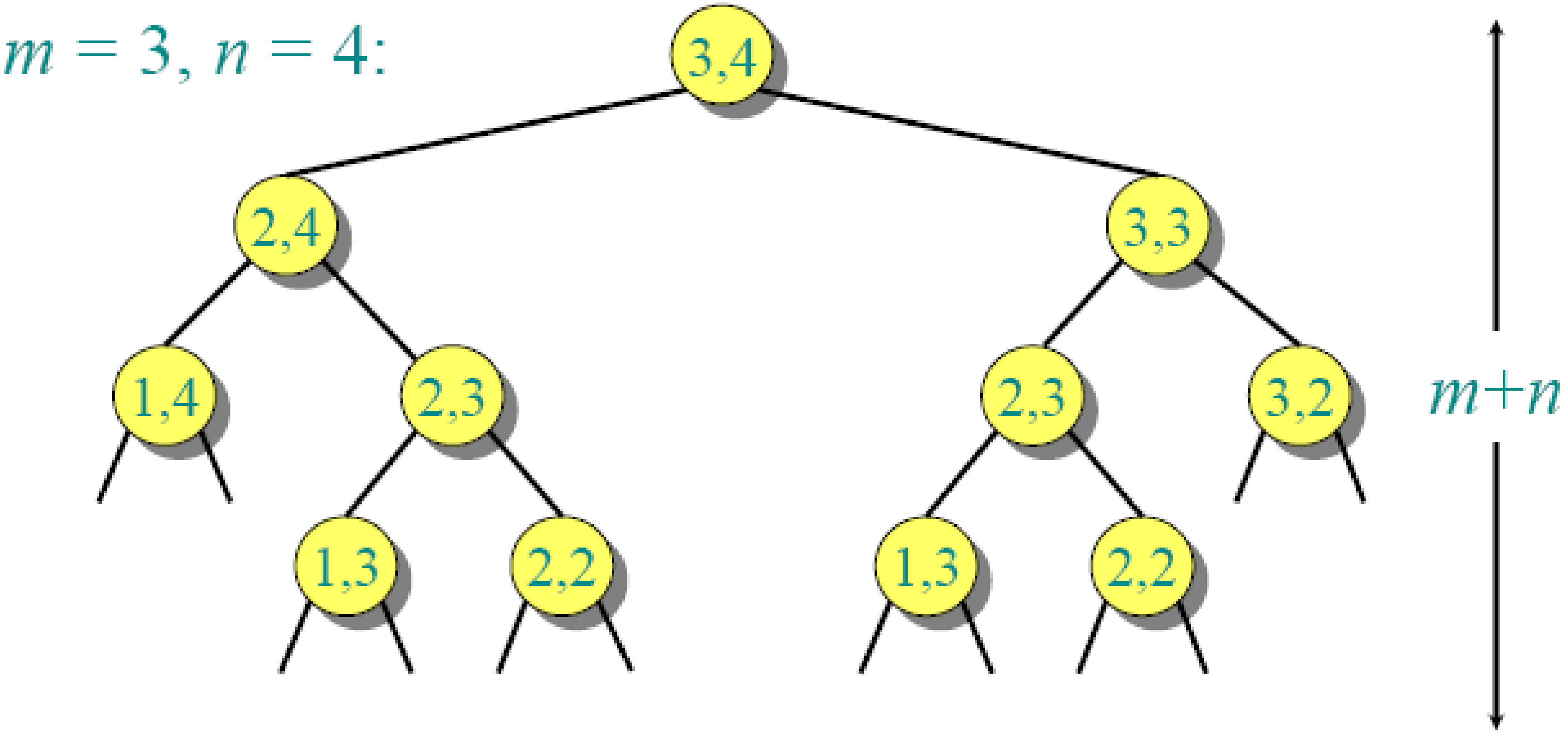
Recursion tree

$m = 3, n = 4$:



Dynamic Programming

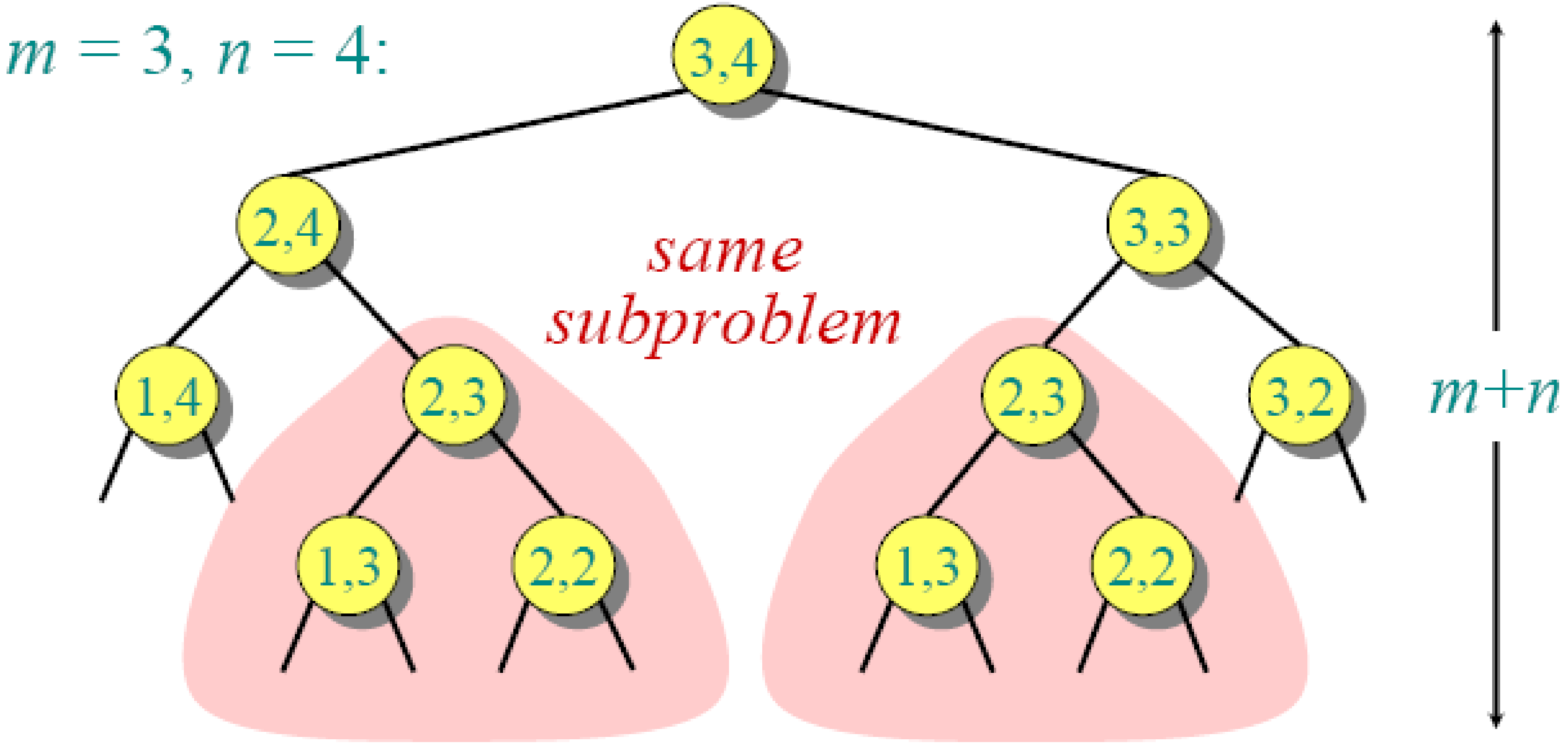
Recursion tree



Height $= m + n \Rightarrow$ work potentially exponential.

Dynamic Programming

Recursion tree



Height = $m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved!

DP Problems

- Fibonacci Number
- Assembly Line
- Matrix Chain
- Subset Sum
- 0-1 Knapsack
- Coin Changer
- All-pair Shortest Paths
- Longest Common Subsequence
- Longest Increasing Subsequence <= O(nlogn)
- Optimal BST <= O(n^2)
- Edit Distance
- Maximum Sum Rectangle
- .. etc