# Computer Algorithms

*Lecture 10: Dynamic Programming – Ch 8*

Dr. Manal Helal, Spring 2014.

http://moodle.manalhelal.com

# Lecture Learning Objectives

1. Use a Dynamic Programming algorithm design strategy to solve problems such as optimisation problems, graph problems and optionally optimal binary search trees construction,

# Dynamic Programming

Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems, in which an optimal solution is related to the optimality of the subproblems.

- 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

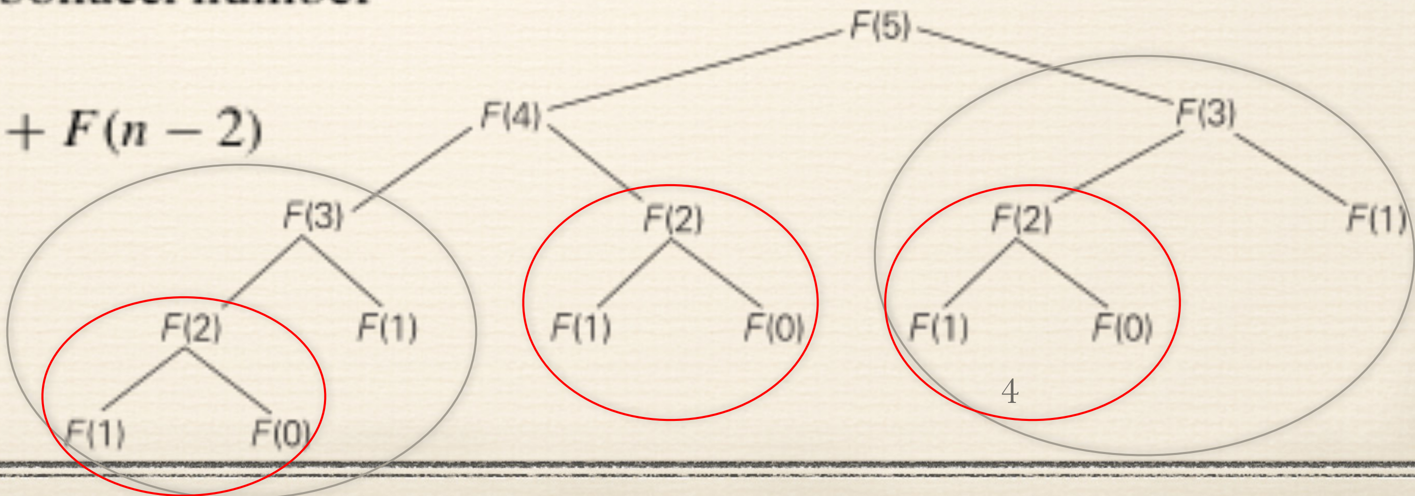
# Example 1: Fibonacci numbers

- Recall definition of Fibonacci numbers:
  $F(n) = F(n\text{-}1) + F(n\text{-}2)$
  $F(0) = 0$
  $F(1) = 1$

- Computing the $n^{th}$ Fibonacci number recursively (top-down):

**ALGORITHM** $F(n)$

//Computes the $n$th Fibonacci number recursively by using its definition
//Input: A nonnegative integer $n$
//Output: The $n$th Fibonacci number
**if** $n \le 1$ **return** $n$
**else return** $F(n-1) + F(n-2)$



# Iterative Fibonacci

**ALGORITHM** *Fib*($n$)

```
//Computes the nth Fibonacci number iteratively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
F[0] ← 0;  F[1] ← 1
for i ← 2 to n do
    F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

# Fibonacci Efficiency

Applying the homogeneous second-order linear recurrence with constant coefficients theorem to our recurrence with the initial conditions given—see Appendix B—we obtain the formula :

$$F(n) = \frac{1}{\sqrt{5}}\phi^n \quad \text{where } \phi = (1+\sqrt{5})/2 \approx 1.61803$$

Constant $\phi$ is known as the golden ratio. The most pleasing ratio of a rectangle's two sides to the human eye.

Therefore the recursive algorithm computes F(n) by recursively adding F(n-1) + F(n-2) for each element from 2 : n, leading to additions $A(n) \in \Theta(\phi^n)$

# Fibonacci Efficiency – Cont'd

The Iterative Algorithm makes $n - 1$ additions , therefore its efficiency is $\Theta(n)$. We can also save space by storing the last two values in the sequence instead of a complete array of n size.

We can also calculate F(n) using the formula:

Using a brute force exponentiation algorithm with efficiency $\Theta(n)$ , or the use Horner's rule for binary exponentiation with efficiency $\Theta(\log n)$

$$F(n) = \frac{1}{\sqrt{5}}\phi^n$$

# Example 2: Coin-row problem

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

E.g.: 5, 1, 2, 10, 6, 2. What is the best selection?

Let F(n) be the maximum amount that can be picked up from the row of n coins. To derive a recurrence for F(n), we partition all the allowed coin selections into two groups:

those without last coin – the max amount is ?

those with the last coin -- the max amount is ?

# DP solution to the coin-row problem

Thus we have the following recurrence

$F(n) = \max\{c_n + F(n\text{-}2),\ F(n\text{-}1)\}$ for $n > 1$,

$F(0) = 0,\ F(1) = c_1$

**ALGORITHM** *CoinRow*($C[1..n]$)

//Applies formula (8.3) bottom up to find the maximum amount of money
//that can be picked up from a coin row without picking two adjacent coins
//Input: Array $C[1..n]$ of positive integers indicating the coin values
//Output: The maximum amount of money that can be picked up
$F[0] \leftarrow 0;\quad F[1] \leftarrow C[1]$
**for** $i \leftarrow 2$ **to** $n$ **do**
    $F[i] \leftarrow \max(C[i] + F[i-2], F[i-1])$
**return** $F[n]$

# DP solution to the coin-row problem (cont.)

$F(n) = \max\{c_n + F(n-2),\ F(n-1)\}$ for $n > 1$,

$F(0) = 0,\quad F(1) = c_1$

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| coins | -- | 5 | 1 | 2 | 10 | 6 | 2 |
| F( ) | 0 | 5 | 5 | 7 | 15 | 15 | 17 |

Max amount: 17

Coins of optimal solution: backTrace, or store as you go: $c_6$, $c_4$, $c_1$.

Time efficiency: $\Theta(n)$

Space efficiency: $\Theta(n)$

Note: All smaller instances were solved.

# Example 3: Change Making

Give change for amount n using the minimum number of coins of denominations $d_1 < d_2 < \ldots < d_m$, where $d_1 = 1$

Let F(n) be the minimum number of coins whose values add up to n; define F (0) = 0; and consider all coins to minimise $F(n - d_j)$ for all j = 1 ... m.
**Example:** n = 6 and denominations $d_1 = 1$, $d_2 = 3$, $d_3 = 4$:

| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| F( ) | 0 | 1 | 2 | 1 | 1 | 2 | 2 |

# DP Change Making

F(0) = 0
F(j) = 1 + MIN (F(i-dj))
where (1 <= j <= m) and (i-dj >= 0) and (0 <= i <= n)

1. j goes from 1 to m because we have m coin denominations (d1 .. dm).
2. (i-dj >= 0) because money values can not be negative so we exclude those (dj) values that yield negative value of (i-dj).
3. (0 <= i <= n) means i can be any values less than or equal to the money amount we need to make change for.
4. Note also that sub problems are overlapping for example F(i-dj) represents one or more values depending on the value of (j) but not all values are calculated every single time from the scratch. Values are saved in F(i) then looked up whenever needed.

# Change Making DP Algorithm

**ALGORITHM** *ChangeMaking*$(D[1..m], n)$

//Applies dynamic programming to find the minimum number of coins
//of denominations $d_1 < d_2 < \cdots < d_m$ where $d_1 = 1$ that add up to a
//given amount $n$
//Input: Positive integer $n$ and array $D[1..m]$ of increasing positive
// integers indicating the coin denominations where $D[1] = 1$
//Output: The minimum number of coins that add up to $n$

$F[0] \leftarrow 0$
**for** $i \leftarrow 1$ **to** $n$ **do**
  $temp \leftarrow \infty;\ j \leftarrow 1$
  **while** $j \le m$ **and** $i \ge D[j]$ **do**
    $temp \leftarrow \min(F[i - D[j]], temp)$
    $j \leftarrow j + 1$
  $F[i] \leftarrow temp + 1$
**return** $F[n]$

Number of Coins: 2
Time efficiency: $\Theta$ (nm)
Space efficiency: $\Theta$(n)

# Tracing for n = 9 and denominations {1, 3, 6, 7}

F(1) = 1 + MIN (F(1-1))
= 1 + MIN (F(0))
= 1 + 0
= 1

F(2) = 1 + MIN (F(2-1))
= 1 + MIN (F(1))
= 1 + MIN (1)
= 2

F(3) = 1 + MIN (F(3-1), F(3-3))
= 1 + MIN (F(2), F(0))
= 1 + MIN (2, 0)
= 1

F(4) = 1 + MIN (F(4-1), F(4-3))
= 1 + MIN (F(3), F(1))
= 1 + MIN (1, 1)
= 2

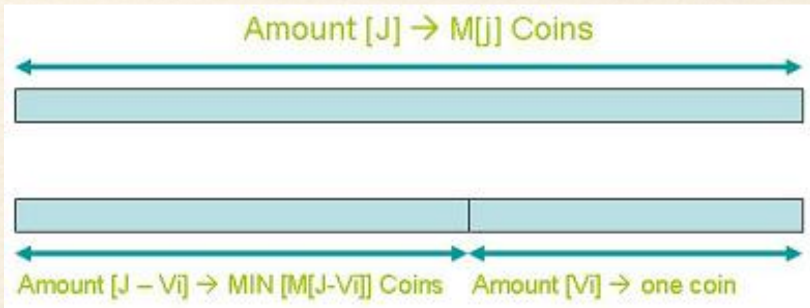
F(5) = 1 + MIN (F(5-1), F(5-3))
= 1 + MIN (F(4), F(2))
= 1 + MIN (2, 2)
= 3

F(6) = 1 + MIN (F(6-1), F(6-3), F(6-6))
= 1 + MIN (F(5), F(3), F(0))
= 1 + MIN (3, 1, 0)
= 1

F(7) = 1 + MIN (F(7-1), F(7-3), F(7-6), F(7-7))
= 1 + MIN (1, 2, 1, 0)
= 1

F(8) = 1 + MIN (F(8-1), F(8-3), F(8-6), F(8-7))
= 1 + MIN (1, 3, 2, 1)
= 2

F(9) = 1 + MIN (F(9-1), F(9-3), F(9-6), F(9-7))
= 1 + MIN (2, 1, 1, 2)
= 2



# Example 4: Coin-Collecting by robot

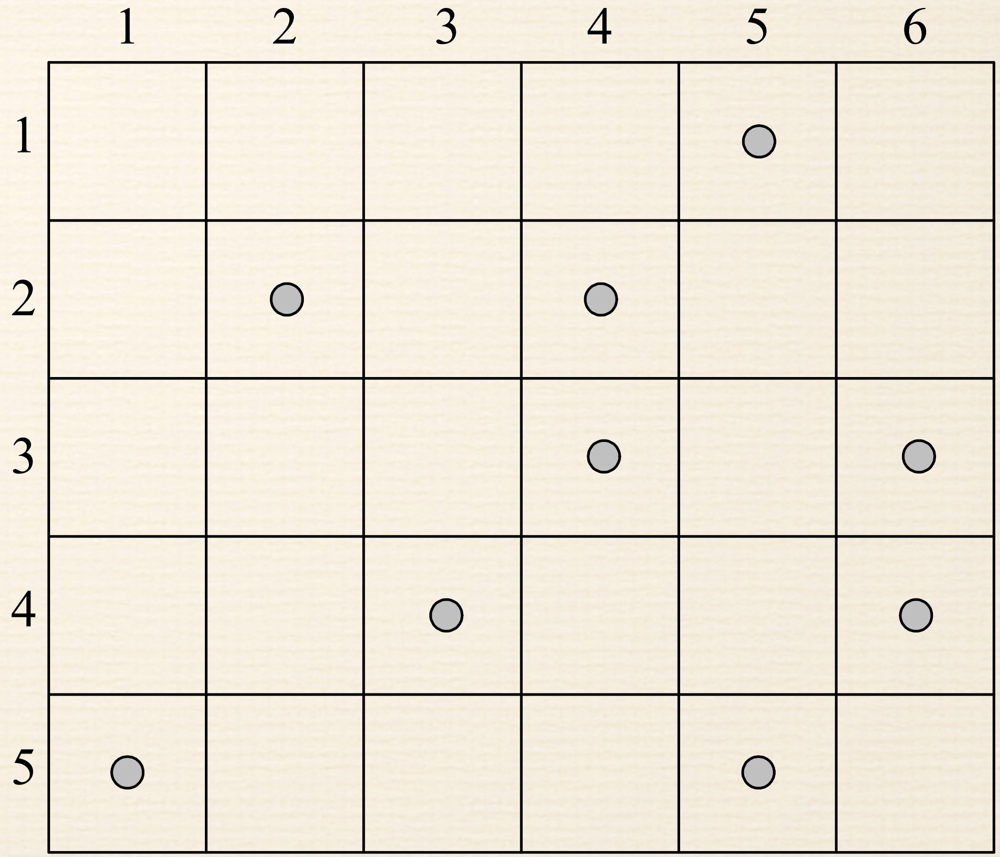
Several coins are placed in cells of an $n \times m$ board. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the right or one cell down from its current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

Let F (i, j ) be the largest number of coins, coming from either F(i-1, j) or F(i, j-1):

$F(i,j)=\max\{F(i-1,j),F(i,j-1)\}+c_{ij}$ for $1 \le i \le n$, $1 \le j \le m$

$F(0,j)=0$ for $1 \le j \le m$

$F(i,0)=0$ for $1 \le i \le n$,



# Coin-Collecting DP Algorithm

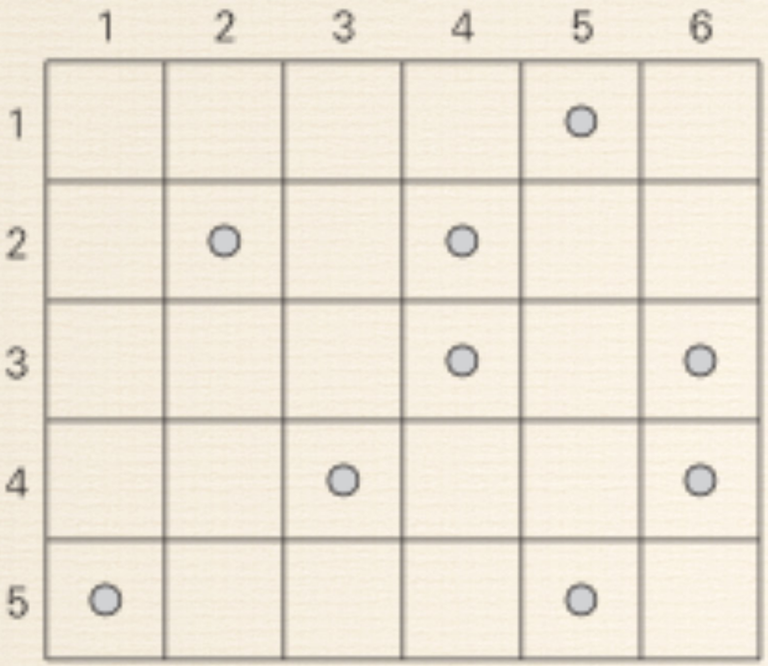
**ALGORITHM** *RobotCoinCollection*($C[1..n, 1..m]$)

//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an $n \times m$ board by starting at (1, 1)
//and moving right and down from upper left to down right corner
//Input: Matrix $C[1..n, 1..m]$ whose elements are equal to 1 and 0
//for cells with and without a coin, respectively
//Output: Largest number of coins the robot can bring to cell $(n, m)$

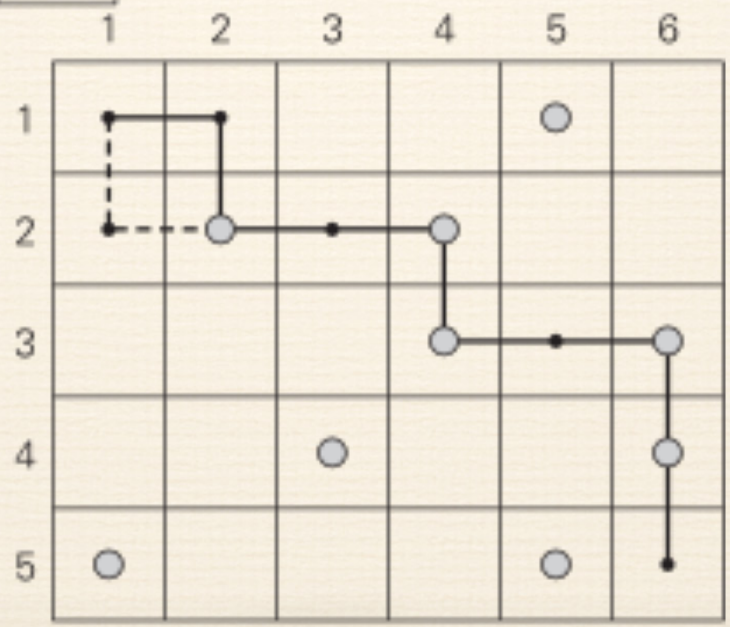
$F[1, 1] \leftarrow C[1, 1]$; **for** $j \leftarrow 2$ **to** $m$ **do** $F[1, j] \leftarrow F[1, j-1] + C[1, j]$
**for** $i \leftarrow 2$ **to** $n$ **do**
    $F[i, 1] \leftarrow F[i-1, 1] + C[i, 1]$
    **for** $j \leftarrow 2$ **to** $m$ **do**
        $F[i, j] \leftarrow \max(F[i-1, j], F[i, j-1]) + C[i, j]$
**return** $F[n, m]$

Time efficiency: $\Theta$ (nm)
Space efficiency: $\Theta$(nm)

# Example 4: Solution

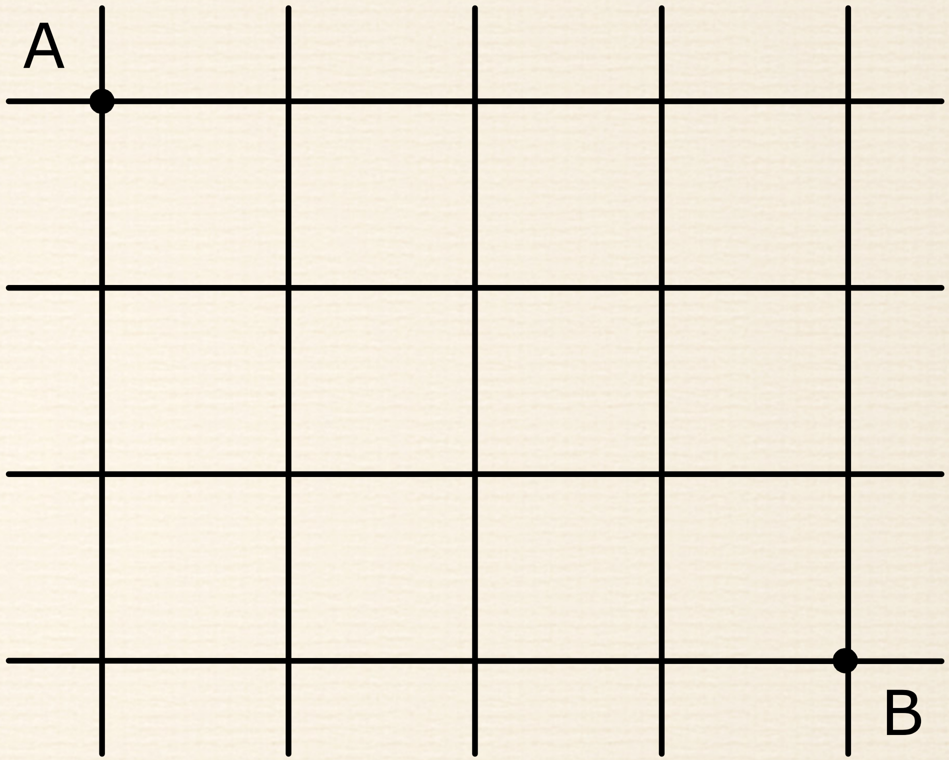


| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 2 | 0 | 1 | 1 | 2 | 2 | 2 |
| 3 | 0 | 1 | 1 | 3 | 3 | 4 |
| 4 | 0 | 1 | 2 | 3 | 3 | 5 |
| 5 | 1 | 1 | 2 | 3 | 4 | **5** |



# Example 5: Path counting

Consider the problem of counting the number of shortest paths from point A to point B in a city with perfectly horizontal streets and vertical avenues



# Other examples of DP algorithms

Computing a binomial coefficient (# 9, Exercises 8.1)

Some difficult discrete optimization problems:
- knapsack (Sec. 8.2)
- traveling salesman

Constructing an optimal binary search tree (Sec. 8.3)

Warshall's algorithm for transitive closure (Sec. 8.4)
Floyd's algorithm for all-pairs shortest paths (Sec. 8.4)

# The 0/1 Knapsack Problem

Given: A set S of n items, with each item i having

- $w_i$ - a positive weight
- $v_i$ - 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

$$\sum_{i \in T} v_i$$

Constraint:

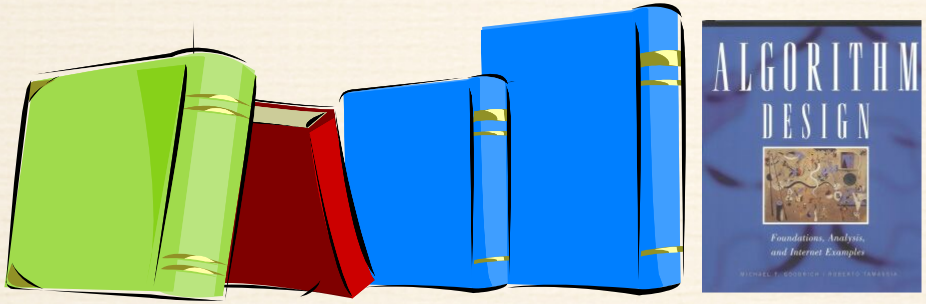
$$\sum_{i \in T} w_i \le W$$

# Example

Given: A set S of n items, with each item i having

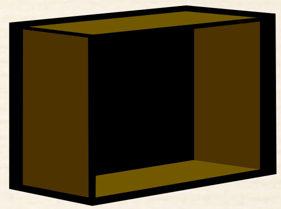
- $b_i$ - a positive "benefit"
- $w_i$ - a positive "weight"

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)

# A 0/1 Knapsack Algorithm, First Attempt

$S_k$: Set of items numbered 1 to k.

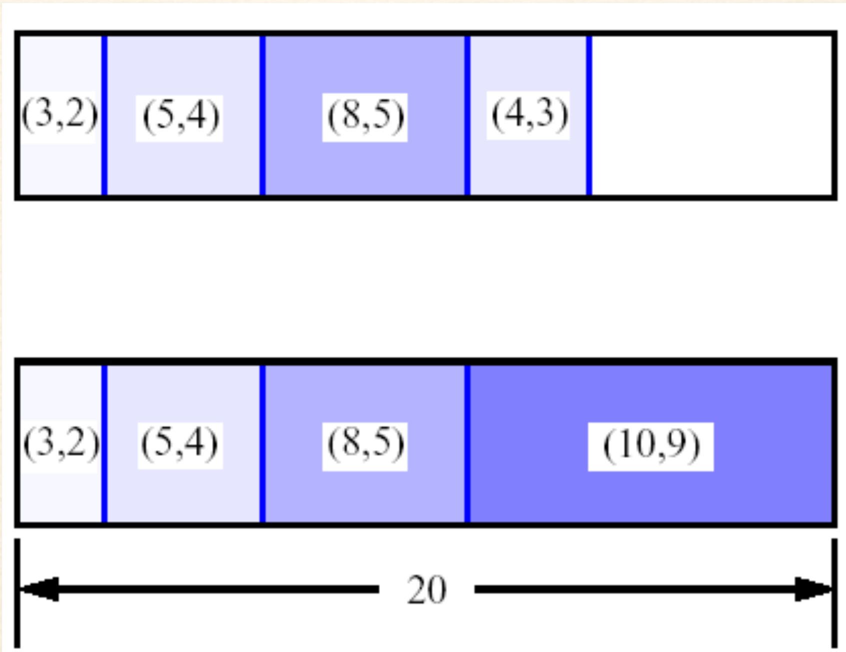
Define $F[k]$ = best selection from $S_k$.

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 $S_5$:




# A 0/1 Knapsack Algorithm, Second Attempt

$S_k$: Set of items numbered 1 to k.

Define F[k,w] to be the best selection from $S_k$ with weight at most w

Good news: this does have subproblem optimality.

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

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

# Knapsack Problem by DP

**Consider** instance defined by first $i$ items and capacity $j$ ($j \leq W$), and value of optimal solution F(i, j) to be the subset of most valuable subset of the first i items that fit into knapsack of capacity j...

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

$$F(0, j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \geq 0.$$

# 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

| k \ w | 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 |

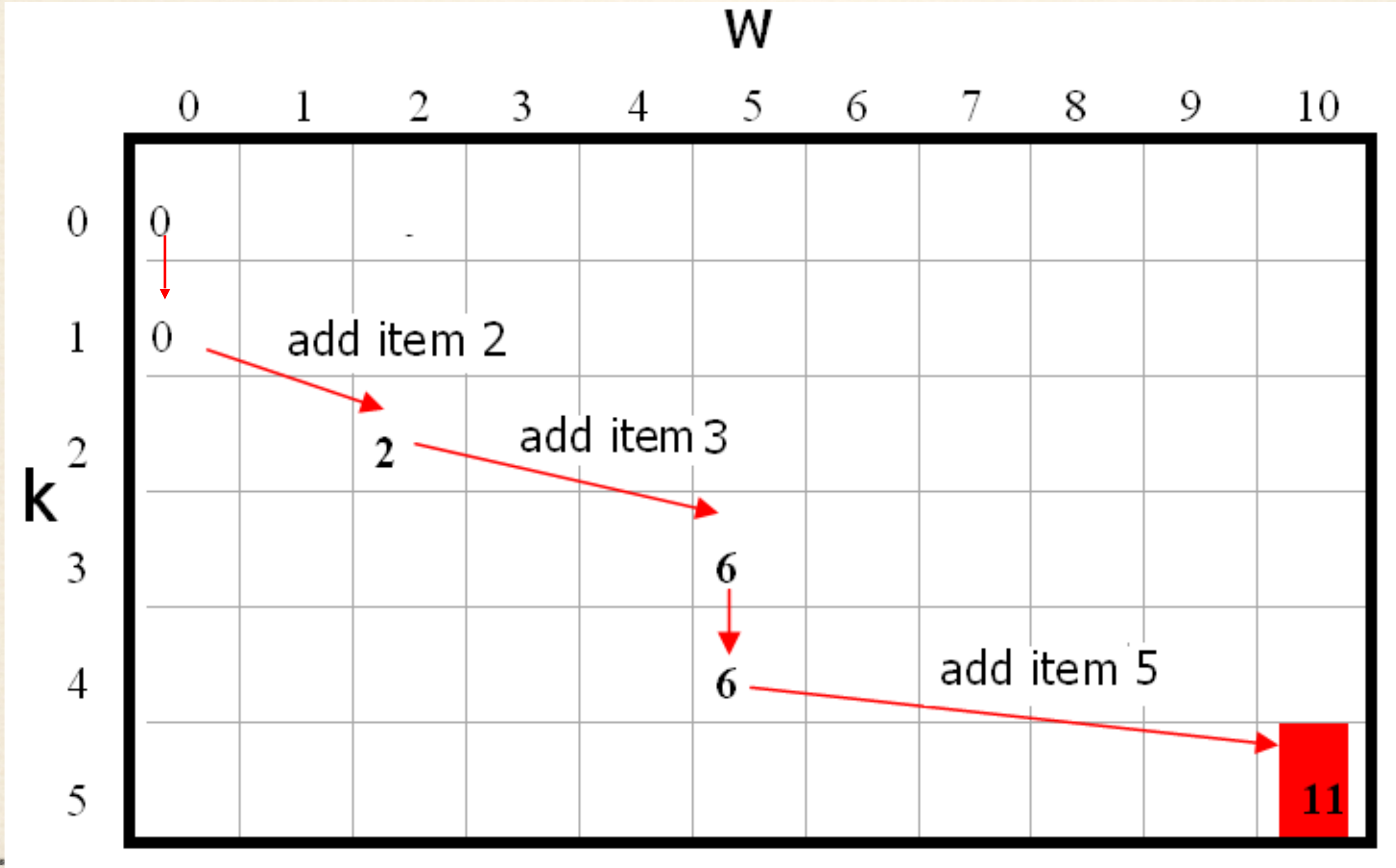
# 0/1 Knapsack Algorithm

Trace back to find the items picked

| k \ W | 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



# DP Knapsack Problem (example 2)

$$F(i, j) = \begin{cases} \max\{F(i-1, j),\ v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \ge 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

capacity $W = 5$

| item | weight | value |
|---|---|---|
| 1 | 2 | \$12 |
| 2 | 1 | \$10 |
| 3 | 3 | \$20 |
| 4 | 2 | \$15 |

| | | 0 | $j-w_i$ | $j$ | $W$ |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| | $i-1$ | 0 | $F(i-1, j-w_i)$ | $F(i-1, j)$ | |
| $w_i, v_i$ | $i$ | 0 | | $F(i, j)$ | |
| | $n$ | 0 | | | goal |

capacity $j$

| items i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | | | | | | |
| 1 | | | | | | |
| 2 | | | | | | |
| 3 | | | | | | |
| 4 | | | | | ? | |

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

# DP Knapsack Problem (example 2)

capacity $W = 5$

| item | weight | value |
|---|---|---|
| 1 | 2 | \$12 |
| 2 | 1 | \$10 |
| 3 | 3 | \$20 |
| 4 | 2 | \$15 |

F(4, 5) = \$37
Items: 4, 2, 1
Time efficiency: $\Theta$ (nW)
Space efficiency: $\Theta$ (nW)
backTrace: O (n)

capacity $j$

| | $i$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $w_1 = 2, v_1 = 12$ | 1 | 0 | 0 | 12 | 12 | 12 | 12 |
| $w_2 = 1, v_2 = 10$ | 2 | 0 | 10 | 12 | 22 | 22 | 22 |
| $w_3 = 3, v_3 = 20$ | 3 | 0 | 10 | 12 | 22 | 30 | 32 |
| $w_4 = 2, v_4 = 15$ | 4 | 0 | 10 | 15 | 25 | 30 | **37** |

>
>
=
>

# Knapsack Problem Bottom-up DP Memory Functions

**ALGORITHM** *MFKnapsack*$(i, j)$

//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer $i$ indicating the number of the first
//      items being considered and a nonnegative integer $j$ indicating
//      the knapsack capacity
//Output: The value of an optimal feasible subset of the first $i$ items
//Note: Uses as global variables input arrays $Weights[1..n]$, $Values[1..n]$,
//and table $F[0..n, 0..W]$ whose entries are initialized with $-1$'s except for
//row 0 and column 0 initialized with 0's
**if** $F[i, j] < 0$
  **if** $j < Weights[i]$
    $value \leftarrow MFKnapsack(i-1, j)$
  **else**
    $value \leftarrow \max(MFKnapsack(i-1, j),$
          $Values[i] + MFKnapsack(i-1, j - Weights[i]))$
  $F[i, j] \leftarrow value$
**return** $F[i, j]$

# Knapsack Problem by DP (example)

capacity $W = 5$

| item | weight | value |
|---|---|---|
| 1 | 2 | \$12 |
| 2 | 1 | \$10 |
| 3 | 3 | \$20 |
| 4 | 2 | \$15 |

F(4, 5) = \$37
Items: 4, 2, 1
Time efficiency: $\Theta$ (nW)
Space efficiency: $\Theta$ (nW)
backTrace: O (n)

capacity $j$

| | $i$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $w_1 = 2,\ v_1 = 12$ | 1 | 0 | 0 | 12 | 12 | 12 | 12 |
| $w_2 = 1,\ v_2 = 10$ | 2 | 0 | — | 12 | 22 | — | 22 |
| $w_3 = 3,\ v_3 = 20$ | 3 | 0 | — | — | 22 | — | 32 |
| $w_4 = 2,\ v_4 = 15$ | 4 | 0 | — | — | — | — | **37** |

# 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]
```

# Longest Common Subsequence

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

Let's give a score M an alignment in this way,

M=sum $s(x_i,y_i)$, where $x_i$ is the i character in the first aligned sequence

$y_i$ is the i character in the second aligned sequence

$s(x_i,y_i)= 1$ if $x_i= y_i$

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

The score for alignment:

```
ababc.
abd.cb
```

M=s(a,a)+s(b,b)+s(a,d)+s(b,.)+s(c,c)+s(.,b)=3

To find the longest common subsequence between sequences $S_1$ and $S_2$ is to find the alignment that maximizes score M.

# Longest Common Subsequence

Subproblem optimality

Consider two sequences

$S_1$: $a_1a_2a_3\ldots a_i$

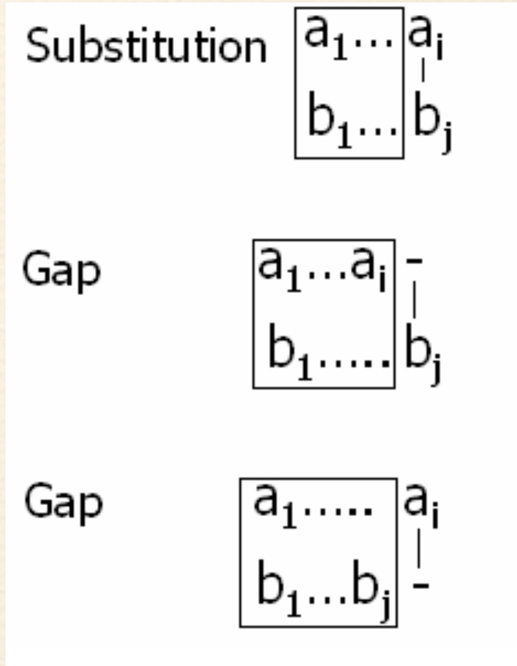
$S_2$: $b_1b_2b_3\ldots b_j$

Let the optimal alignment be

$x_1x_2x_3\ldots x_{n-1}x_n$

$y_1y_2y_3\ldots y_{n-1}y_n$

There are three possible cases for the last pair $(x_n,y_n)$:



# Longest Common Subsequence

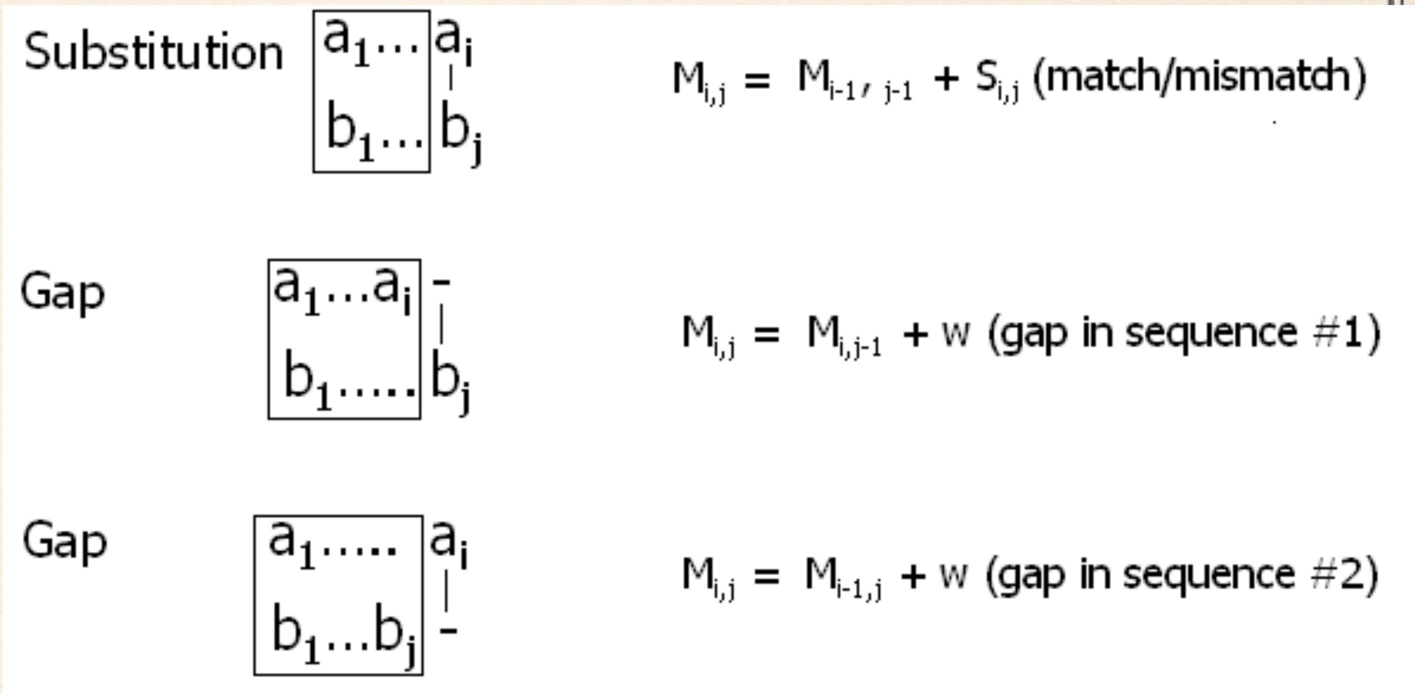
There are three cases for $(x_n, y_n)$ pair:

$S_1$: $a_1 a_2 a_3 \ldots a_i$

$S_2$: $b_1 b_2 b_3 \ldots b_j$

$x_1 x_2 x_3 \ldots x_{n-1} x_n$

$y_1 y_2 y_3 \ldots y_{n-1} y_n$




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

$M_{i,j}$ 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

$$M_{i,j} = \text{MAX}\ \{$$

$$M_{i-1,\ j-1} + S(a_i,b_j)$$

$$M_{i,j-1} + 0$$

$$M_{i-1,j} + 0$$

$$\}$$

$s(a_i,b_j)= 1$ 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 $\boxed{\begin{matrix} a_1 \ldots \\ b_1 \ldots \end{matrix}} \begin{matrix} a_i \\ | \\ b_j \end{matrix}$ $M_{i,j} = M_{i-1, j-1} + S_{i,j}$ (match/mismatch)

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

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

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

| | | 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 | 1 | | | | | | | | | | |
| G | | | | | | | | | | | | |
| A | | | | | | | | | | | | |
| T | | | | | | | | | | | | |
| C | | | | | | | | | | | | |
| G | | | | | | | | | | | | |
| A | | | | | | | | | | | | |

Trace back table B

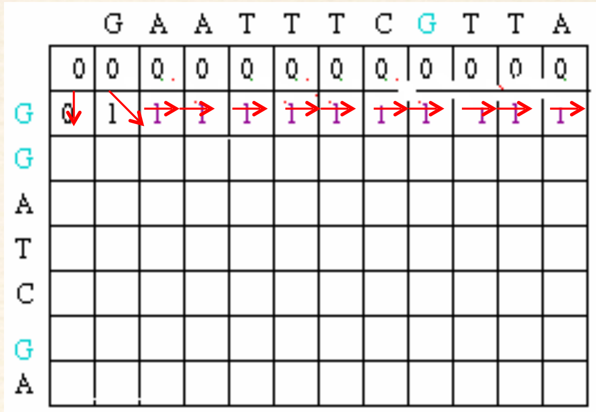
# Longest Common Subsequence

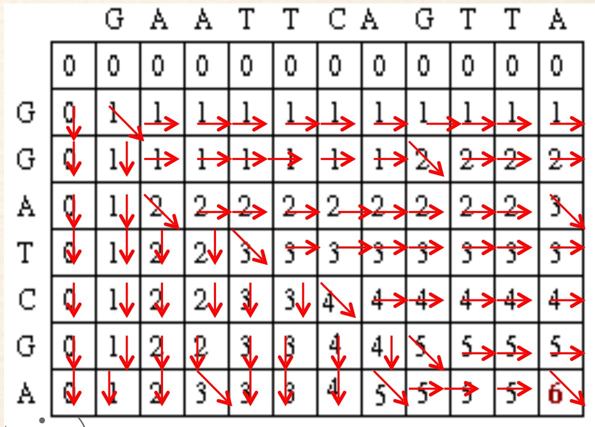
Score matrix M

| | | G | A | A | T | T | T | C | 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 | | | | | | | | | | | | |
| A | | | | | | | | | | | | |
| T | | | | | | | | | | | | |
| C | | | | | | | | | | | | |
| G | | | | | | | | | | | | |
| 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 |

Trace back table B





$M_{7,11}=6$ (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
```

| | | 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 |

:Take one character from each sequence

:Take one character from sequence $S_1$ (columns)

:Take one character from sequence $S_2$ (rows)

```
(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)

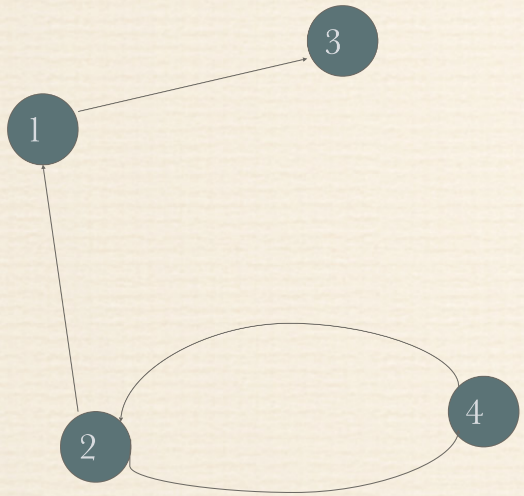
# Longest Common Subsequence

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

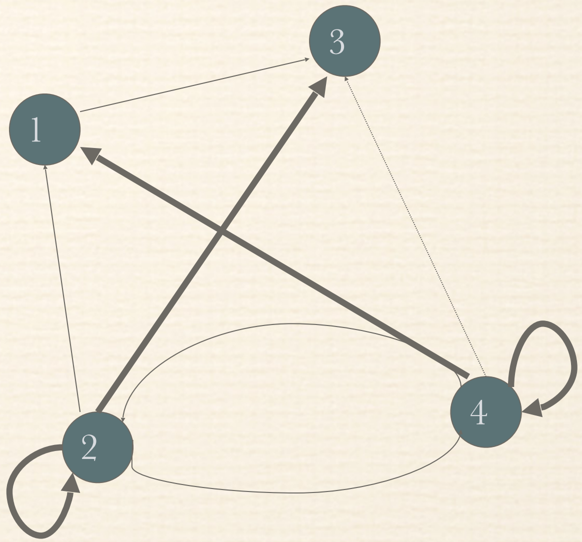
```
M: Score Matrix
B: trace back table (use letter a, b, c for      ↘ )→  ↓
n=A.length()
m=B.length()
// fill in M and B
for (i=0;i<m+1;i++)
    for (j=0;j<n+1;j++)
  if (i==0) || (j==0)
        then M(i,j)=0;
  else if (A[i]==B[j])
       M(i,j)=max {M[i-1,j-1]+1, M[i-1,j], M[i,j-1]}
            {update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

# Warshall's Algorithm: Transitive Closure

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




$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$



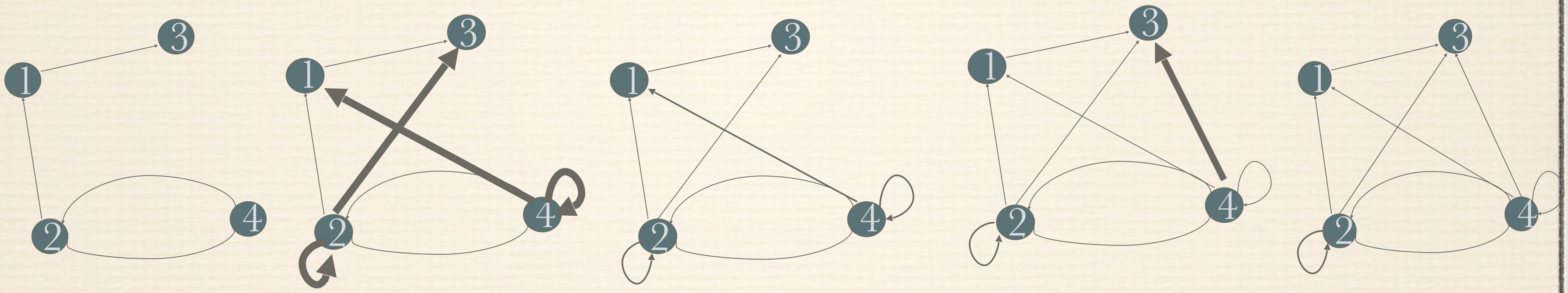

$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix}$$

# Warshall's Algorithm

Constructs transitive closure T as the last matrix in the sequence of n-by-n matrices $R^{(0)}, \ldots, R^{(k)}, \ldots, R^{(n)}$ where

$R^{(k)}[i,j] = 1$ iff there is nontrivial path from i to j with only first k vertices allowed as intermediate

Note that $R^{(0)} = A$ (adjacency matrix), $R^{(n)} = T$ (transitive closure)

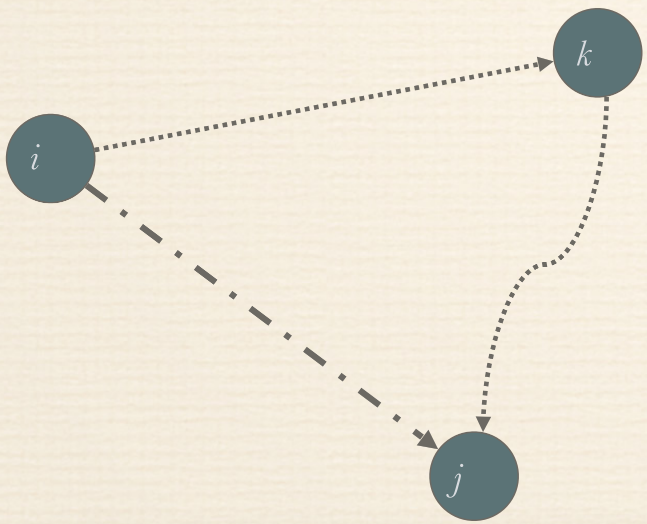



$$R^{(0)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix} \qquad R^{(1)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & 0 & \mathbf{1} \end{matrix} \qquad R^{(2)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{matrix} \qquad R^{(3)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & \mathbf{1} & 1 \end{matrix} \qquad R^{(4)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

# Warshall's Algorithm (recurrence)

On the $k$-th iteration, the algorithm determines for every pair of vertices $i, j$ if a path exists from $i$ and $j$ with just vertices 1,…,$k$ allowed as intermediate

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just 1 ,…,}k\text{-1)} \\ & \text{or} \\ R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j] & \text{(path from } i \text{ to } k \\ & \text{and from } k \text{ to } i \text{ using} \\ & \text{just 1 ,…,}k\text{-1)} \end{cases}$$



# Warshall's Algorithm (matrix generation)

Recurrence relating elements $R^{(k)}$ to elements of $R^{(k-1)}$ is:

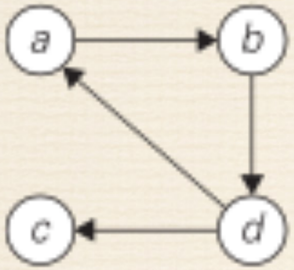
$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

It implies the following rules for generating $R^{(k)}$ from $R^{(k-1)}$:

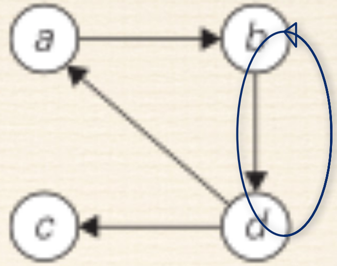
•Rule 1 If an element in row $i$ and column $j$ is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$

•Rule 2 If an element in row $i$ and column $j$ is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row $i$ and column $k$ and the element in its column $j$ and row $k$ are both 1's in $R^{(k-1)}$

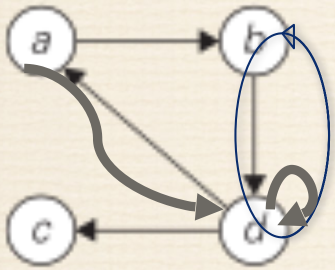
# Warshall's Algorithm (example)



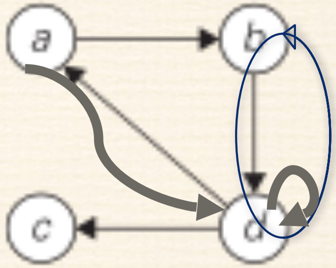
$$R^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$



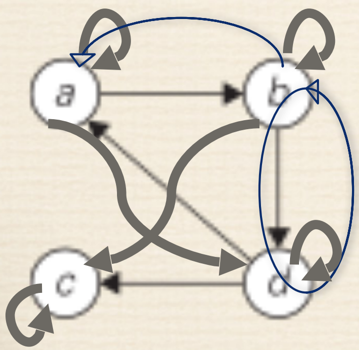
$$R^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & \mathbf{1} & 1 & 0 \end{bmatrix} \end{array}$$



$$R^{(2)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & \mathbf{1} \end{bmatrix} \end{array}$$



$$R^{(3)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$



$$R^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} \mathbf{1} & 1 & \mathbf{1} & 1 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

# Warshall's Algorithm (pseudocode and analysis)

**ALGORITHM** *Warshall*($A[1..n, 1..n]$)
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix $A$ of a digraph with $n$ vertices
//Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$
**for** $k \leftarrow 1$ **to** $n$ **do**
    **for** $i \leftarrow 1$ **to** $n$ **do**
        **for** $j \leftarrow 1$ **to** $n$ **do**
            $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
**return** $R^{(n)}$

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

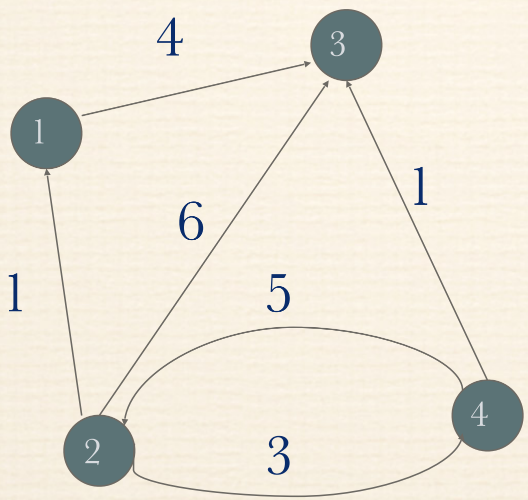
Space efficiency: Matrices can be written over their predecessors

# Floyd's Algorithm: All pairs shortest paths

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

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

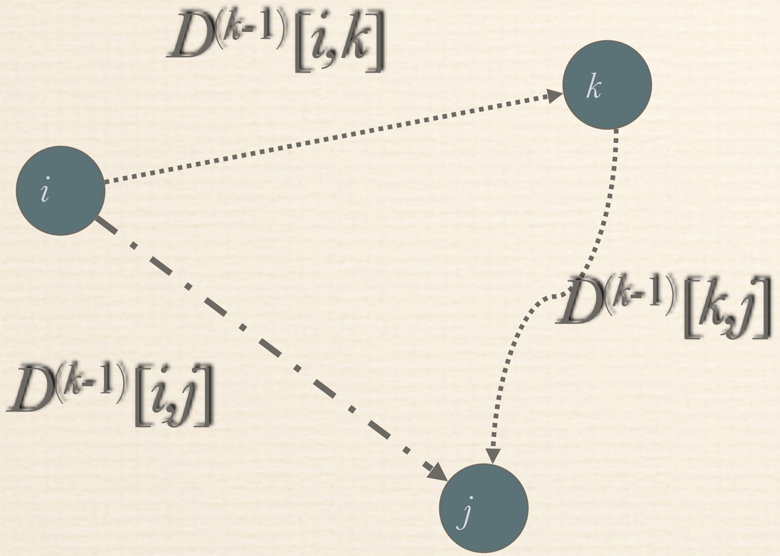
Example:



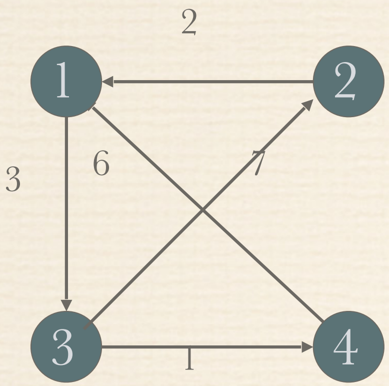
# Floyd's Algorithm (matrix generation)

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

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



# Floyd's Algorithm (example)




$$D^{(0)} = \begin{array}{cccccccc} 0 & \infty & 3 & \infty & 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{array}$$

$$D^{(1)} = \begin{array}{cccc} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{array}$$

$$D^{(2)} = \begin{array}{cccc} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{array}$$

$$D^{(3)} = \begin{array}{cccc} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{array}$$

$$D^{(4)} = \begin{array}{cccc} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{array}$$

# 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
```

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

Space efficiency: Matrices can be written over their predecessors

Note: Shortest paths themselves can be found, too (Problem 10)

# Assignment 3

Consider the change making problem, design the trace back algorithm, or rewrite the algorithm to return the coins indices as well.

# Bonus Assignment 2

Consider the knapsack problem, design the trace back algorithm, or rewrite the algorithm to return the coins indices as well. – 2 marks