COMP2230

Algorithms

Lecture 9

Professor Ljiljana Brankovic

Lecture Overview

- Dynamic Programming – Chapter 8, Text

Slides based on:

Levitin. The Design and Analysis of Algorithms
R. Johnsonbaugh & M. Schefer. Algorithms
G. Brassard & P. Bratley. Fundamentals of Algorithms
Slides by Y. Lin and P. Moscato

Dynamic Programming

- Use a "bottom-up" approach
  - answers to small instances build up to larger ones
- Create a table of (partial) answers.
  - The sub-answers are combined to form full answers.
  - Trade some space for either better time or reliability.
- There is an implicit assumption
  - What are we assuming about the sub-answers, to ensure the full answer is correct?

Principal of Optimality

If we have a set of optimal solutions to a group of sub-instances, and we have an optimal method of combining solutions, then we have an optimal solution to an instance made up from the sub-instances.

or

If S is an optimal solution to an instance, then components of S are optimal solutions to sub-instances.

Note that this principle does not always apply – for example, it does not apply to the longest path problem. We <u>need</u> this principle to apply for Dynamic Programming to work.

A Risky bet...

Fred is a notoriously good cards player. He makes you an interesting (though possibly expensive) proposition:

- You and Fred play 10 games per week.
- First to win 100 games is the winner.
  - If Fred wins, you pay Fred $\$100$
  - If you win, Fred pays you $\$1000$
- You estimate Fred has a 60% chance of winning each game, based on previous games.

Should you accept this bet?

Accept?

- If Fred and you are to play a single game then you could easily calculate the expected gain:

$$E(gain) = 0.4 \times 1000 + 0.6 \times (-100) = 340$$ **Accept!**

- Let $P(i, j)$ be the probability of you winning the bet, given that you need to win i more games, and Fred needs to win j more games.

- We need to calculate $P(100,100)$

$$E(gain) = P(100,100) \times 1000 + \left(1 - P(100,100)\right) \times (-100)$$

- This is similar to the calculations performed by casinos around the world, to work out how much to charge per bet.

$$0.4 \times 1000 + 0.6 \times (-x) < 0 \rightarrow x > 667$$

Setting up a recursion

- Let $P(i, j)$ be the probability of you winning the bet, given that you need to win i more games, and Fred needs j more games.

- We need to calculate $P(100{,}100)$

- Probability that Fred wins a single game is $p = 0.6$

- Probability that Fred loses a single game is $q = 1 - p = 0.4$

- Boundary conditions need to be specified…

- $P(0, j) = 1$ for all $0 < j \leq 100$
  - You have already won!

- $P(i, 0) = 0$ for all $0 < i \leq 100$
  - Fred has already won!

- $P(0{,}0)$ can't exist
  - We can't have both winning

Probability formula

- How we relate what has happened with the future ?
- We have the following formula:

$$P(i,j) = \ p\, P(i,j-1) + \ (1-p)P(i-1,j) \qquad i,j \ \geq \ 1$$

```
function P(i,j)
   if i=0 then return 1
   if j=0 then return 0
   return pP(i,j-1) + (1-p)P(i-1,j)
```

How long will this take?

Barometer statement?

$i = 0$ *in the first **if** statement*

Let $C(a,b)$ be the number of times the barometer statement is executed, given inputs a & b.

- $$C(a,b) = \begin{cases} 1 & a = 0 \\ 1 & b = 0 \\ C(a-1,b) + C(a,b-1) + 1 & otherwise \end{cases}$$

How long (cont)

- We can show by induction that

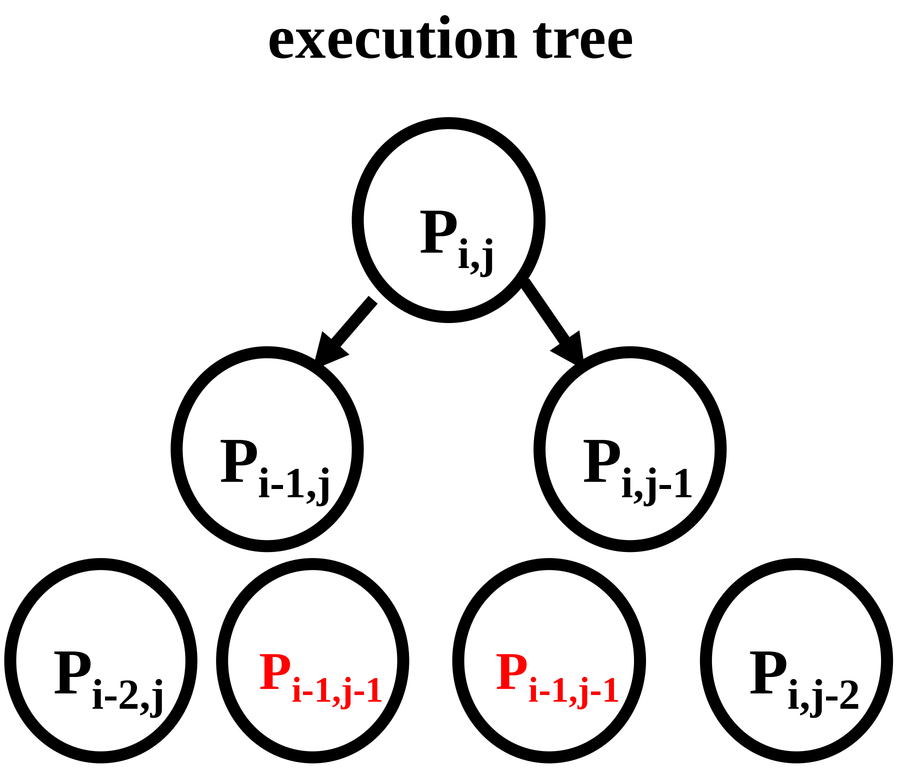
$$C(a,b) = \Omega\binom{2n}{n}$$

where $n = \min(a,b)$

- For $n = 100$, this is approximately 9×10^{58}
- If you execute 10^{10} instructions per second, that's about 2.8×10^{41} years (which is a really long time as 1.4×10^{10} years is the estimated age of the universe)
- Why is it so bad? What can be done?

Solution: Save the work we've done

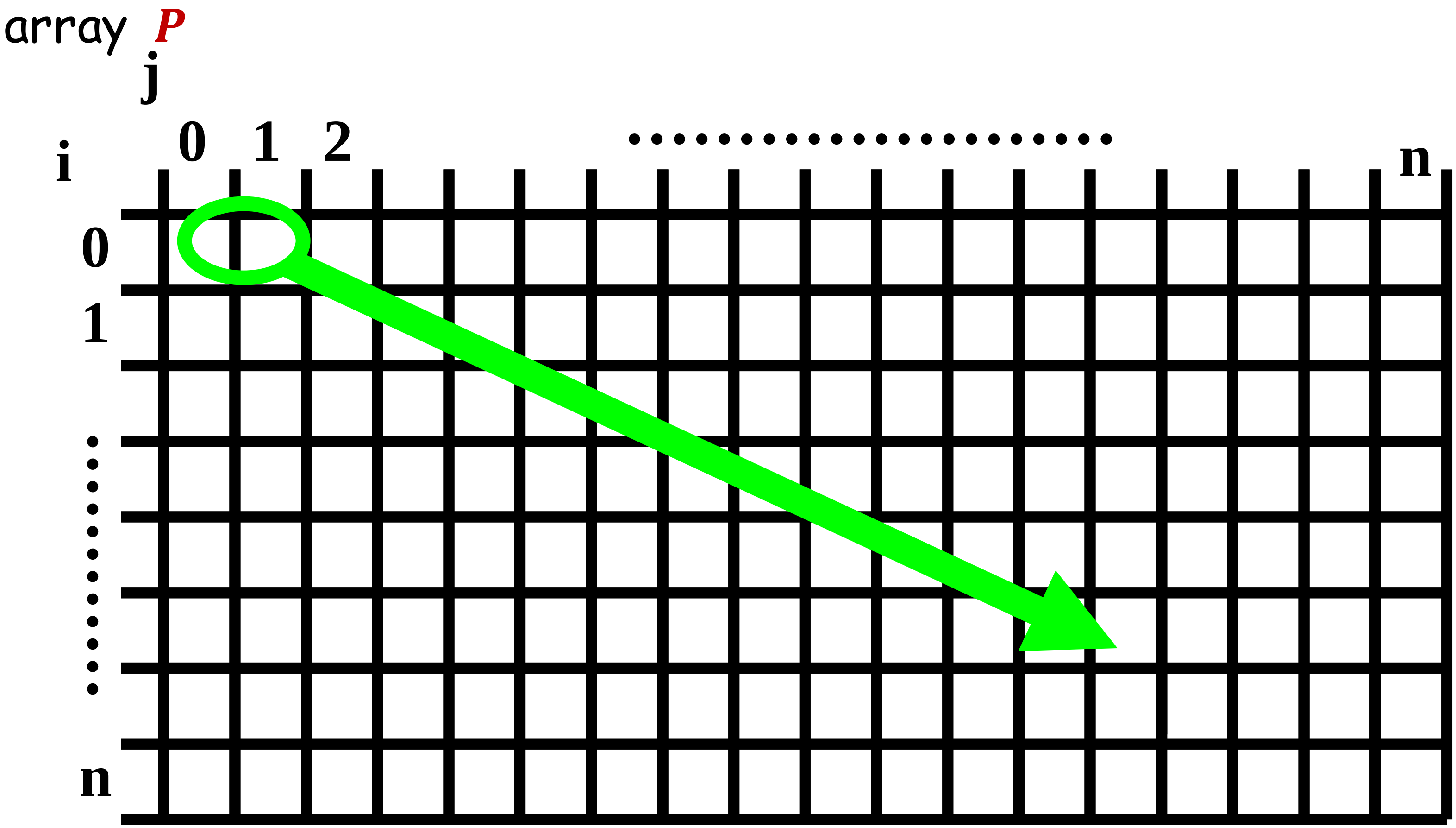
- Recall that D&C gives a top-down approach.
- We should try to work from bottom up and re-use the work we've already done.
  - avoid calculating the same thing twice (or many times)



Dynamic Programming

- Bottom-up approach
  - combines solutions to smaller instances to make solutions to bigger ones
  - big improvements in performance are possible
  - “programming” refers to a matrix (array) storage
  - some storage issues
    - trades big improvement in time complexity against small loss in space
  - needs good book-keeping, keeps track of what is going on.

Calculating $P(i,j)$ using array P



We need a way to fill in array, from $(0,0)$ to (n,n)

Then we fill the array…

- We know all the boundary values: $P(i,0)$ and $P(0,j)$
- So fill them in first.
- How do we get

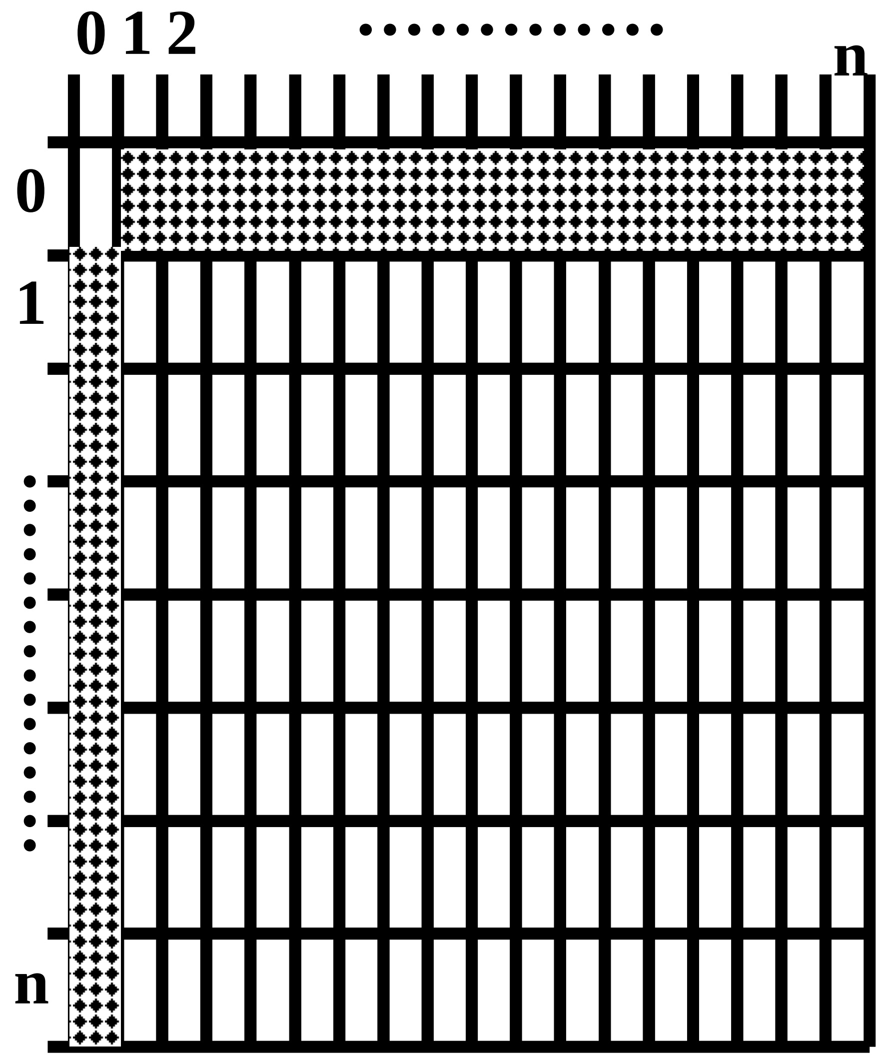
$P(1,1)$?

$P(1,2)$?

$P(1,3)$?

$P(n,n)$?

In our case: $P(100,100)$?



The table

	j = 0	1	2	3	4	5	6	7	8	9	10
i = 0		1	1	1	1	1	1	1	1	1	1
1	0	0.4	0.64	0.784	0.87	0.922	0.953	0.972	0.983	0.99	0.994
2	0	0.16	0.352	0.525	0.663	0.767	0.841	0.894	0.929	0.954	0.97
3	0	0.064	0.179	0.317	0.456	0.58	0.685	0.768	0.833	0.881	0.917
4	0	0.026	0.087	0.179	0.29	0.406	0.517	0.618	0.704	0.775	0.831
5	0	0.01	0.041	0.096	0.174	0.267	0.367	0.467	0.562	0.647	0.721
6	0	0.004	0.019	0.05	0.099	0.166	0.247	0.335	0.426	0.514	0.597
7	0	0.002	0.009	0.025	0.055	0.099	0.158	0.229	0.308	0.39	0.473
8	0	7E-04	0.004	0.012	0.029	0.057	0.098	0.15	0.213	0.284	0.359
9	0	3E-04	0.002	0.006	0.015	0.032	0.058	0.095	0.142	0.199	0.263
10	0	1E-04	7E-04	0.003	0.008	0.018	0.034	0.058	0.092	0.135	0.186

The dynamic programming approach

```
function(a,b,p)
//Calculate P[a,b] with prob. p
n ← max(a,b); array P[0..n,0..n]
for i = 1 to n
    P[i,0] ← 0
for j = 1 to n
    P[0,j] ← 1
for i = 1 to n
    for j = 1 to n
          P[i,j] ← p P[i,j-1] + (1-p) P[i-1,j]
return P[a,b]
```

Set up array with the boundary conditions

Work to complete the array

So, how long now?

- Analysis is straight forward:
  - filling the array takes $\Theta(n^2)$
  - array takes space $\Theta(n^2)$
  - assumes addition is $\Theta(1)$

- Could we make it take less space? $\Theta(n)$?

Benefits

- Improved efficiency
  - as shown, we avoid calculating things more than once
- Improved reliability
  - some problems we have seen can be better solved with dynamic programming
    - consider some of the greedy problems
    - sometimes dynamic programming gives a solution where greedy wouldn't

Making change revisited

<u>Making change problem</u>: For a given amount A and given denominations, make amount A using smallest total number of coins

Example 1:

- denominations: 1, 4, 6
- find change for 8
- greedy: 6, 1, 1
- optimal: 4, 4

Example 2:

- denominations : $1, 1\frac{3}{4}, 4$
- find change for 3
- greedy: $(1\frac{3}{4})$ + 1.... fail
- optimal: 1, 1, 1

Sometimes greedy gives non-optimal solution, and sometimes it completely fails.

Dynamic Programming for change...

What would a bottom-up approach to making change look like?

- Some sort of table of sub-solutions

- Problem reduced by considering reduced sets of denominations; in Example 1:
  - only coins of value 1
  - then coins of value 1 and 4
  - then coins of value 1, 4 and 6 ... .

Dynamic making change

- Imagine we have found an “optimal” solution using only the first $k-1$ denominations (don’t care how yet):

- Coins: $d_1 \ d_2 \ d_3 \ \ldots \ d_{k-1}$
  - we know how many, and what types of coins are needed to give optimal results for all change amounts

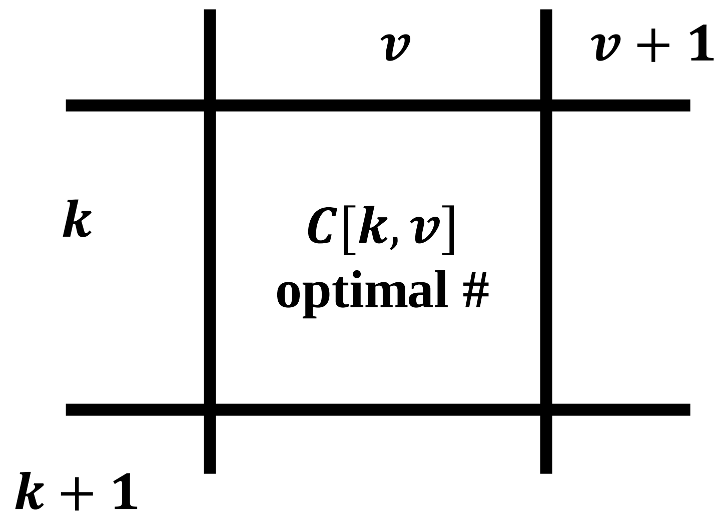
all old denominations
+ *the next one*

- Say we now consider the case $d_1 \ d_2 \ \ldots \ d_{k-1} \ d_k$
  - what choices do we have to update the old solution?

Dynamic Programming for Change – The array

Create a table of answers $C[k, v]$ such that:

- k = number of denominations considered so far
- v = amount of change to make
- $C[k, v]$ = *total number* of coins required to make change value v, if only the first k denominations are used.



We will assume we have a list of all coin denominations, stored as a vector, $d[k]$ or d_k for clarity

To use, or not to use

- We want $\$v$ change.
- Let old solution be $C[k-1, v]$
- The table is full to this point

which option do we choose?

- The optimal solution does not use coin d_k

$$C[k, v] = C[k-1, v]$$

  - solution is the same
  - if it were optimal before, and we don't need to use the coin d_k, it's optimal now

- The optimal solution uses coin d_k
  - $C[k, v] = 1 + C[k, v - d_k]$
  - we need to solve problem of making $\$(v - d_k)$ given the same set of coins
  - If that solution were optimal, we get the optimal solution

$$C[k, v] = min\{C[k-1, v], 1 + C[k, v - d_k]\}$$

What about the boundary conditions?

- How many coins do we need to make $\$0$ in change?
  - note the pattern for $C[k, 0]$

- What if we can't make a certain amount using (as many as possible) coins given?
  - e.g., make $\$1$ using only $\$2$ coins.
  - e.g., make negative amounts

- Use a special symbol, e.g., $+\infty$

A back-of-envelope test

- Try out the ideas before we write the pseudo-code.

Amount:	0	1	2	3	4
$d_1 = 1$	0	1	2	3	4
$d_2 = 4$	0				
$d_3 = 6$	0				

compare $C[k-1, v]$ and $1 + C[k, v - d[k]]$

Dynamic Change

```
function Coins(A, d[1..n])
//Finds minimum # coins to make $A change
//Array d gives coin denominations
//Assume infinite amount  of  coins in each denomination
array C[1..n, 0..n]
for i ← 1 to n
   C[i,0] = 0
for i ← 1 to n
   for j ← 1 to A
      C[i,j] ← if i = 1 & j < d[i]  then +∞
               elseif i = 1 then 1 + C[1, j-d[1]]
               elseif j < d[i] then C[i-1, j]
               else min{C[i-1,j], 1 + C[i,j-d[i]]}
```

Which coins?

- This gives (as greedy did) a solution which only tells us how many coins are needed, but not which ones.

Questions for you to answer:

- How do we interpret the table to find the coins to use?
- What is O( ) of making change?
- What is O( ) of finding coins?

Dynamic Progamming Summary

- Build up solutions from little ones
  - may have to fill up array and waste space/time
  - results may be faster than straight divide & conquer
  - results may be more reliable than straight greedy

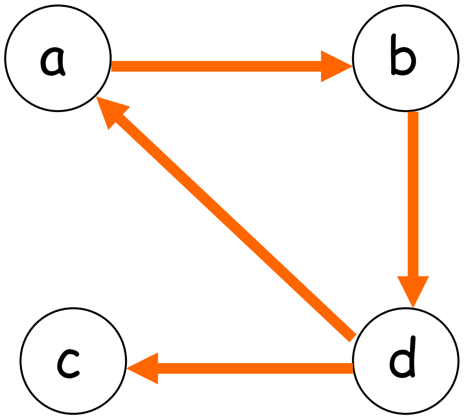
- Based on the Principle of Optimality: *In an optimal sequence of decisions, each subsequence must also be optimal.*

Warshall's Algorithm

- Warshall's algorithm is a dynamic programming algorithm for computing the transitive closure of a directed graph.

- The transitive closure of a directed graph G is an $n \ x \ n$ matrix T of zeros and ones, where $t_{ij} = 1$ if and only if there is a directed nontrivial path from vertex i to vertex j.

Example 3 - Warshall's algorithm



$$A = \begin{array}{c|cccc} & a & b & c & d \\ a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

$$T = \begin{array}{c|cccc} & a & b & c & d \\ a & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

A - adjacency matrix

T - transitive closure

Warshall's Algorithm

- One way to solve this problem would be to run a depth-first search (or breath-first search) from each vertex in digraph G.

- Another way to solve this problem is Warshall's algorithm which starts from adjacency matrix M_0 of the digraph G and then through a series of matrices $M_1, M_2, \ldots, M_n$ constructs the transitive closure of G.

- In the matrix M_k, $m_{ij} = 1$ if and only if there is a directed path from i to j such that no vertex on the path has label greater than k.

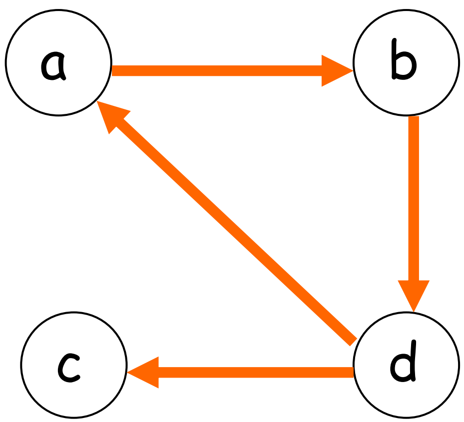
Warshall's Algorithm

The matrix M_k can be obtained directly from M_{k-1} as follows. The element m_{ij} in M_k is equal to one if and only if there is a path from i to j which does not contain any vertices with label greater than k.

This will be the case if one of the following is true:

1. There is a path from i to j which does not contain any vertex with label greater than $k-1$; in this case $m_{ij} = 1$ in M_{k-1}
2. There is a path from i to j that contains vertex k but does not contain any vertex with a label greater than k; then there is a path from i to k and from k to j and thus in M_{k-1} we have $m_{ik} = 1$ and $m_{kj} = 1$.

Example 3 - Warshall's algorithm



$$A=M_0=\begin{array}{cc} & \begin{array}{cccc}a & b & c & d\end{array}\\ \begin{array}{c}a\\ b\\ c\\ d\end{array} & \begin{bmatrix}0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0\\ 1 & 0 & 1 & 0\end{bmatrix}\end{array}$$

$$M_1=\begin{array}{cc} & \begin{array}{cccc}a & b & c & d\end{array}\\ \begin{array}{c}a\\ b\\ c\\ d\end{array} & \begin{bmatrix}0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0\\ 1 & 1 & 1 & 0\end{bmatrix}\end{array}$$

$$M_2=\begin{array}{cc} & \begin{array}{cccc}a & b & c & d\end{array}\\ \begin{array}{c}a\\ b\\ c\\ d\end{array} & \begin{bmatrix}0 & 1 & 0 & 1\\ 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0\\ 1 & 1 & 1 & 1\end{bmatrix}\end{array}$$

$$M_3=\begin{array}{cc} & \begin{array}{cccc}a & b & c & d\end{array}\\ \begin{array}{c}a\\ b\\ c\\ d\end{array} & \begin{bmatrix}0 & 1 & 0 & 1\\ 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0\\ 1 & 1 & 1 & 1\end{bmatrix}\end{array}$$

$$T=M_4=\begin{array}{cc} & \begin{array}{cccc}a & b & c & d\end{array}\\ \begin{array}{c}a\\ b\\ c\\ d\end{array} & \begin{bmatrix}1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1\\ 0 & 0 & 0 & 0\\ 1 & 1 & 1 & 1\end{bmatrix}\end{array}$$

Algorithm 8.5.12 Warshall's Algorithm

This algorithm computes the transitive closure of a diagraph G onvertices $\{1, \ldots, n\}$. The input is the adjacency matrix A of G. The output is the transitive closure of G.

```
Input Parameters: A
Output Parameters: A
transitive_closure(A) {
    n = A.last
    for k = 1 to n
        for i = 1 to n
            for j = 1 to n
                A[i][j] = A[i][j] ∨ (A[i][k] ∧ A[k][j])
}
```

Floyd's Algorithm

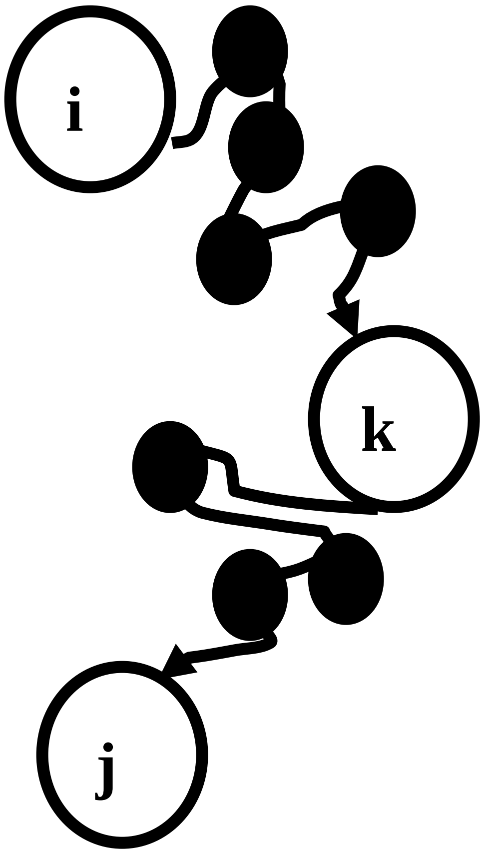
Given a graph G, either directed or undirected, we want to find the shortest path from each node to every other node.

- Brute force: run Dijkstra from every node.
- What is the complexity of this?

- Dynamic programming may be helpful.
  - Does optimality apply?
    - what do we need for optimality to apply?

Shortest paths (cont)

- Consider a single shortest path through three nodes $i - k - j$ (plus other intermediates)
  - Say k is on the shortest path from i to j
  - What can we say about the paths from i to k, and k to j?



i to j via k

We can ask:

- for a given pair of nodes (i, j), is it shorter to go via the current path, or is it better to go via some node k ?

- Assume we have a matrix of lengths $D_{k-1}[1..n, 1..n]$
  - D_{k-1} holds the best path lengths for intermediate nodes $\{1, 2, .., k-1\}$
  - If we now consider node k, how do we update D_{k-1} ?

- We can do this using similar approach as Warshall's algorithm.

via k, or not via k

- If we go via node k, the path will be from $i \longrightarrow k$, and then from $k \longrightarrow j$, using the previous best paths.

$$D_k[i,j] = D_{k-1}[i,k] + D_{k-1}[k,j]$$

- If we don't go via k, then the path length is unchanged

$$D_k[i,j] = D_{k-1}[i,j]$$

  - What would we do if we wanted to know the actual path from i to j?
  - What is D_0?

Working up to the pseudo-code

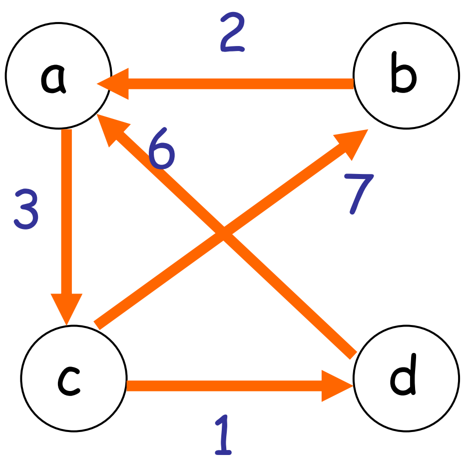
- We need to check all pairs (i, j) and ask is the node k on a shorter path.

So we'll have a nested for loop.

```
for i ← 1 to n
      for j ← 1 to n
            D[i,j] ← min { D[i,j] ,  D[i,k] + D[k,j] }
```

- Note: we can just overwrite D as we go, so we do not need to store all the matrices.

Example 4 – Floyd's algorithm



$$A = D_0 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array} \qquad D_1 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{bmatrix} \end{array}$$

$$D_2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array} \qquad D_3 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \end{array} \qquad D_4 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

Algorithm 8.5.3 Floyd's Algorithm, Version 1

This algorithm computes the length of a shortest path between each pair of vertices in a simple, directed or undirected, weighted graph G, which does not contain a negative cycle. The input is the adjacency matrix A of G. The output is the matrix A whose ij-th entry is the length of a shortest path from vertex i to vertex j.

```
Input Parameter: A
Output Parameter: A
all_paths(A) {
    n = A.last
    for k = 1 to n // compute A(k)
        for i = 1 to n
            for j = 1 to n
                if (A[i][k] + A[k][j] < A[i][j])
                    A[i][j] = A[i][k] + A[k][j]
}
```