Summary

Known graph algorithms (so far)

- Depth first search (DFS)
  - Input: Any graph
  - Output options: DFS order, dead-end order, spanning tree
  - Applicable when: Problem requires visiting all the things (vertices)
- Breadth first search (BFS)
  - Input: Any graph
  - Output options: BFS order, spanning tree
  - Applicable when: Problem requires visiting all the things (vertices)
- Connected components
  - Input: Any graph
  - Output options: Count or Boolean ("is it connected")
  - Note: Modified DFS/BFS
  - Applicable when: Determining how many "clumps" of vertices there are

Summary

Known graph algorithms (so far)

- Topological sort
  - Input: Directed acyclic graph (DAG)
  - Output: Linear ordering of the vertices
  - Note: We know two algorithms – modified DFS and dec&conq
  - Applicable when: Need to find an order of the vertices
- Minimum spanning tree (MST)
  - Input: Weighted graph
  - Output: Tree
  - Note: We know two algorithms – Prim, Kruskal
  - Applicable when: Want to form the cheapest connected network
- Single-source shortest paths (SSSP)
  - Input: Weighted graph + starting vertex
  - Output options: "Lengths" array, shortest-path tree (aka "prev" array)
  - Note: Dijkstra
  - Applicable when: Looking for shortest path from a particular vertex to all others

Lecture 10

COMP 3760

Dynamic programming

Text chapter 8

Hello, dear old friends: The Fibonacci numbers

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

- Each number is the sum of the previous two:

  fib(0) = 1

  fib(1) = 1

  fib(n) = fib(n-1) + fib(n-2)

- How many can we compute?

DEMO

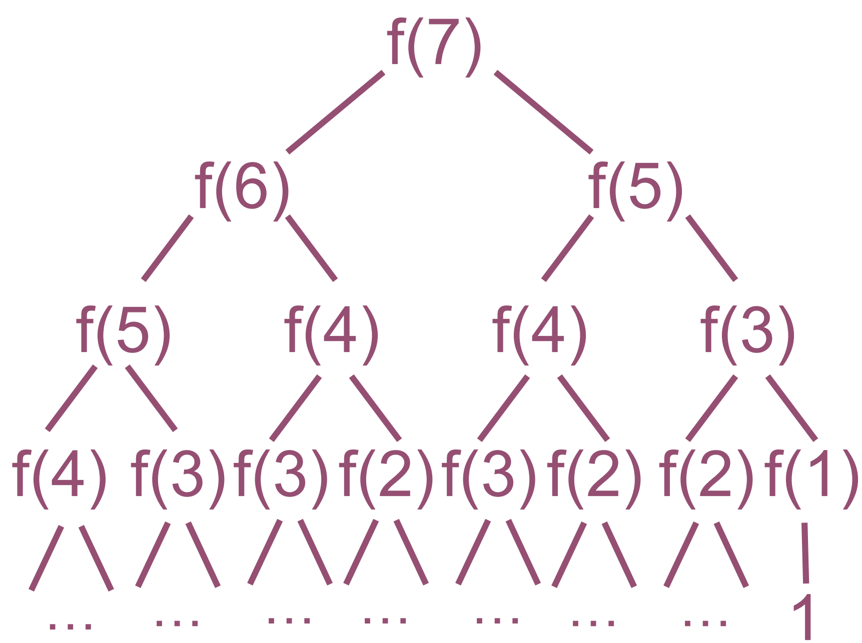
THE CLASSIC RECURSIVE ALGORITHM:

```
fib (n):
    if n < 2
        return n     //i.e. fib(0) is 0 and fib(1) is 1
    else
        return = fib(n-1) + fib(n-2)
```

Fibonacci numbers: Why you so slow?

```
fib (n):
    if n < 2
        return n;
    else
        return = fib(n-1) + fib(n-2)
```

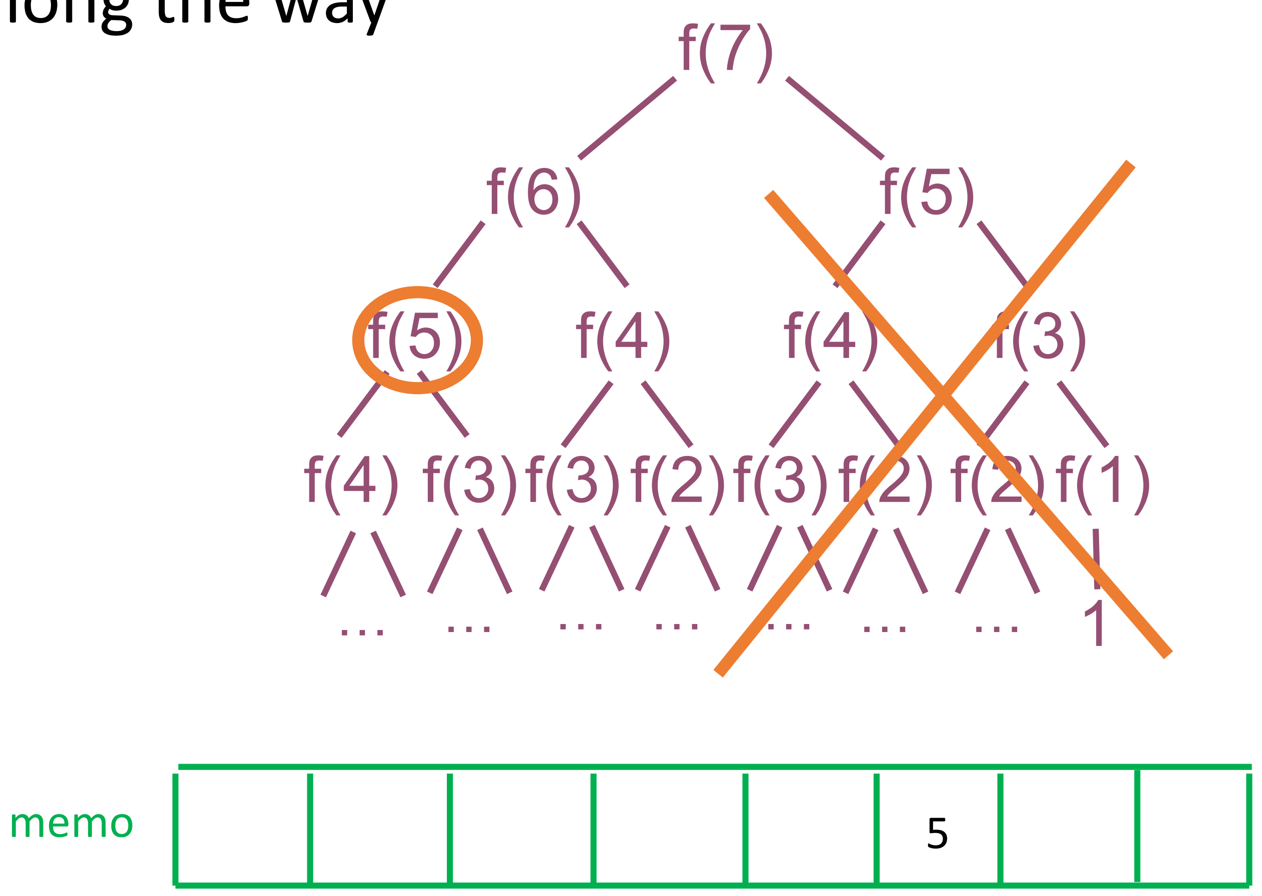
Execution tree:



F(n) takes exponential time to compute.

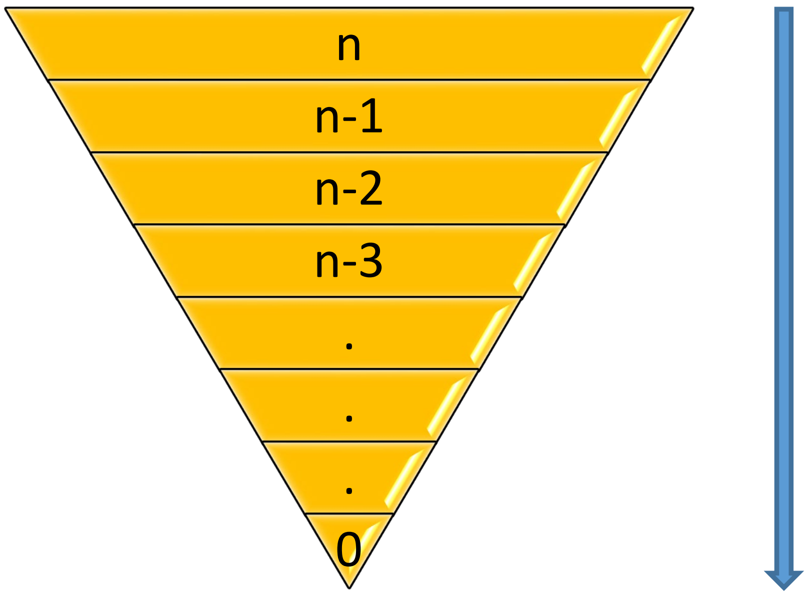
Space-time trade-off

- Augment the algorithm by *remembering* the results you get along the way



Fibs, top-down

```
fib (n) {
    if memo[n] exists, return it
    if n < 2
        return n
    else
        f = fib(n-1) + fib(n-2)
        memo[n] = f
        return f
}
```



top-down (Recursive)

memo

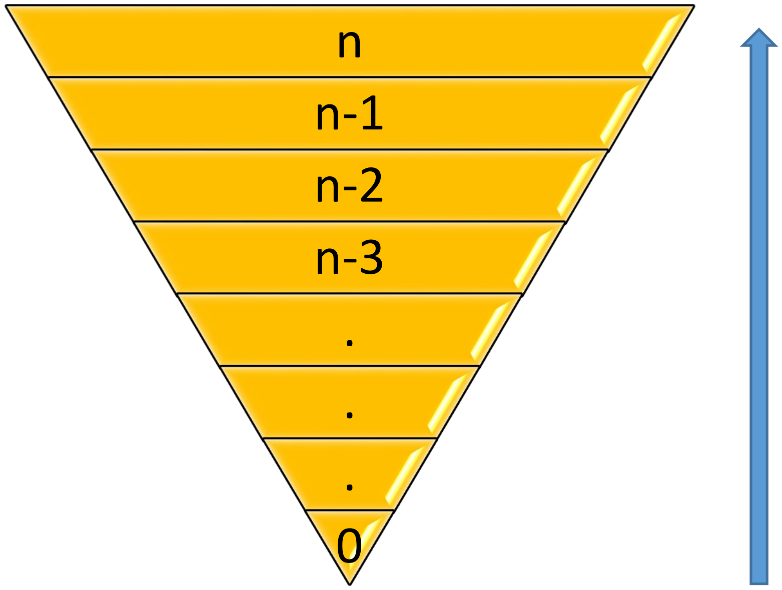
0	**1**	**1**	**. . .**	$\boldsymbol{fib(n\text{-}2)}$	$\boldsymbol{fib(n\text{-}1)}$	$\boldsymbol{fib(n)}$

Efficiency:

- time: O(n)
- space: Needs an array size O(n)

Fibs, bottom-up

```
fib (n) {
    memo[0] = 0;
    memo[1] = 1;
    for i ← 2 to n do
        memo [i] = memo[i-1] + memo[i-2]
    return memo[n]
}
```



bottom-up

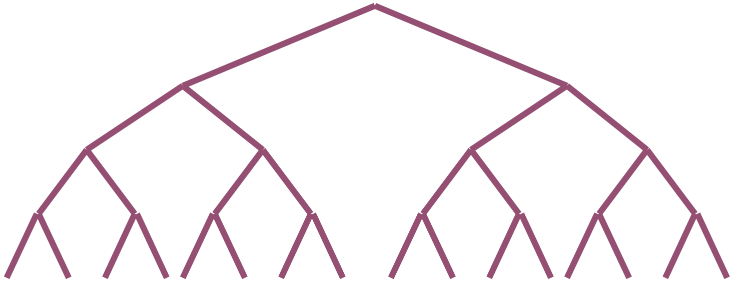
memo							
	0	**1**	**1**	. . .	$fib(n\text{-}2)$	$fib(n\text{-}1)$	$fib(n)$

Efficiency:

- time: O(n)
- space: Needs an array size O(n)

Dynamic programming

- Key point: *remembering* recursively-defined solutions to sub-problems and using them to solve the problem
- Might remind you of divide-and-conquer … but it isn't!
  Store solutions to sub-problems for possible reuse.
- A good idea if many of the sub-problems are repeats



Dynamic programming overview

- **Step 1:**
  - Decompose problem into smaller, equivalent sub-problems
- **Step 2:**
  - Express solution in terms of sub-problems
- **Step 3:**
  - Use table to compute optimal value bottom-up
- **Step 4:**
  - Find optimal solution based on steps 1-3

Dynamic programming examples

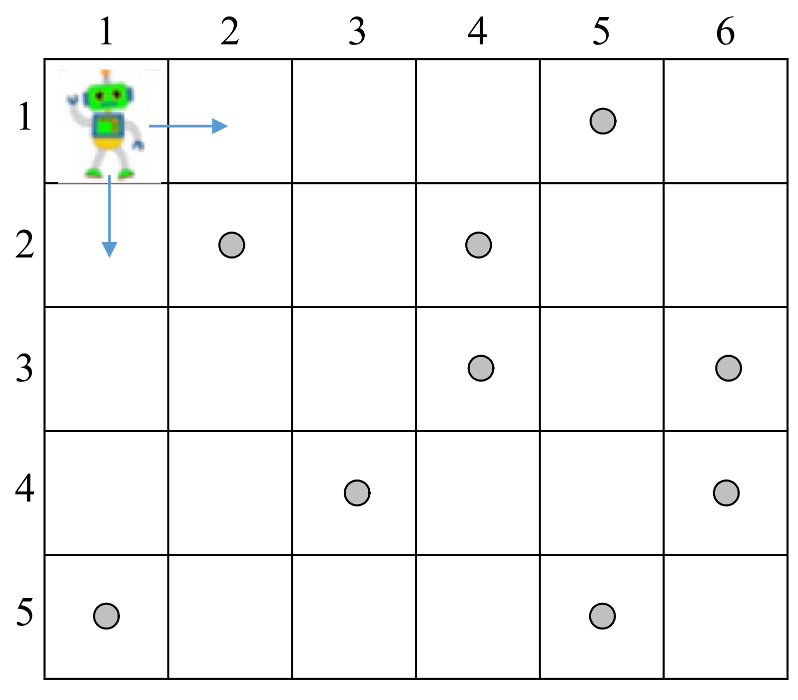
- Fibonacci numbers
- Robot Coin Collecting
- Transitive Closure (Warshall)
- All Pairs Shortest Paths (Floyd)

Dynamic Programming: Coin-collecting Robot

(Chapter 8)

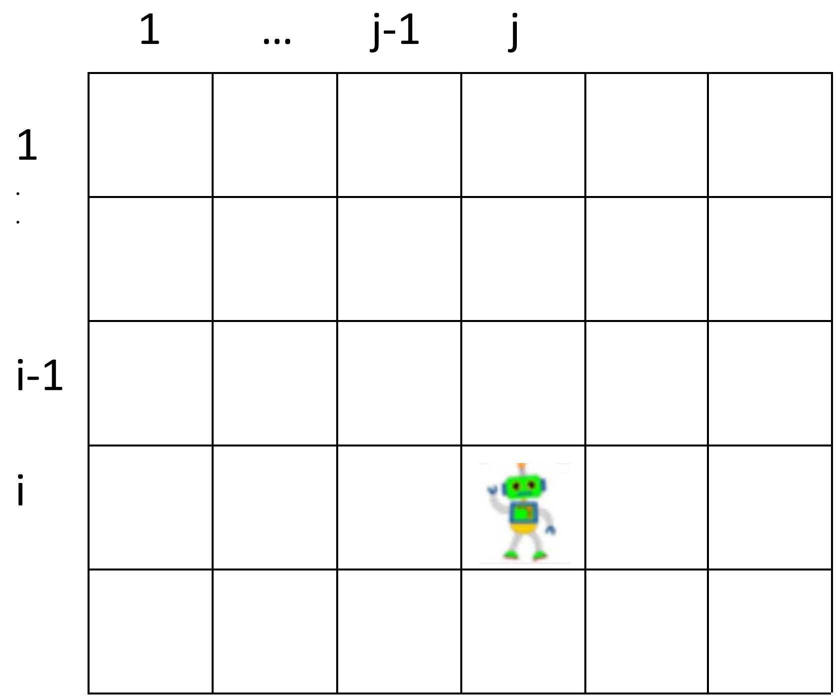
Coin-collecting robot

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. The robot can only move *right* or *down*.



Solution

- Let F(i,j) be the largest number of coins the robot can collect and bring to cell (i,j) in the *ith* row and *j*th column.

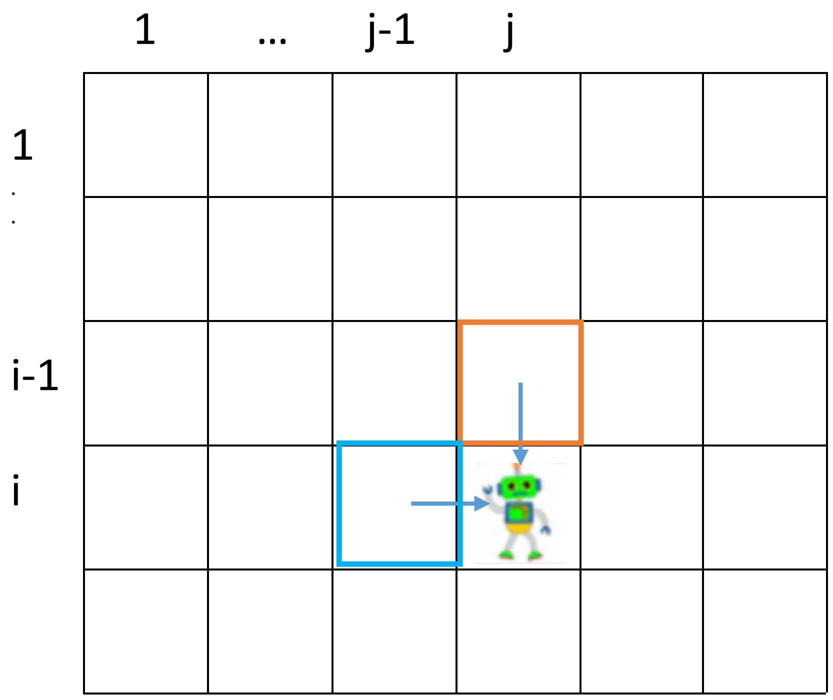


Solution

How many coins could the robot bring to cell (i,j)?

If it comes from the left → F(i, j-1)

If it comes from above → F(i-1, j)

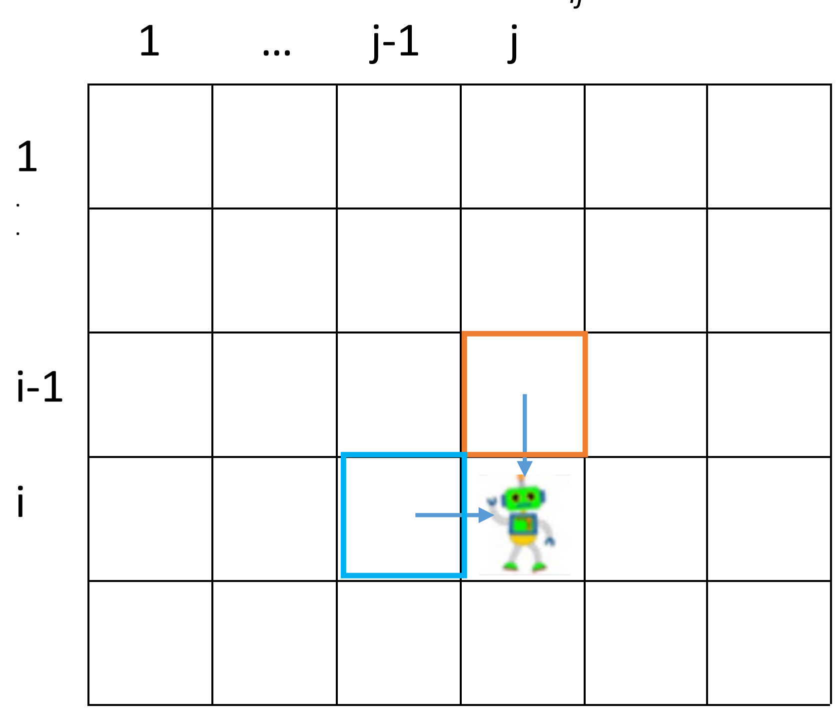


Solution

Recursive definition:

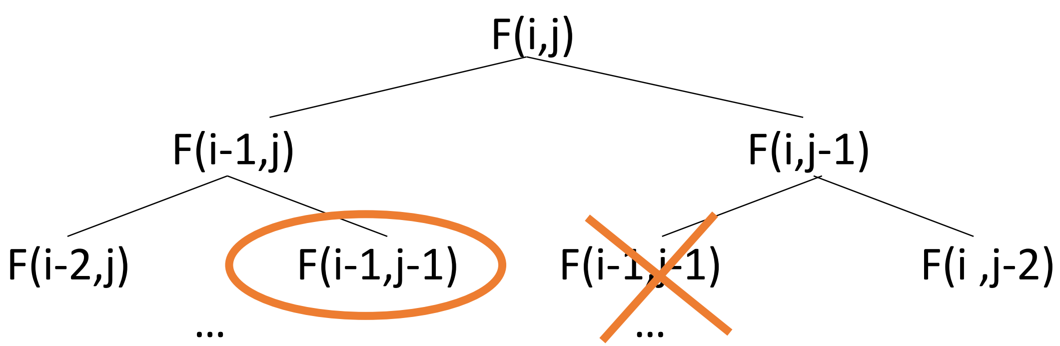
$$F(i, j) = \max\{F(i\text{-}1, j),\ F(i, j\text{-}1)\} + c_{ij} \quad \text{for } 1 \le i \le n,\ 1 \le j \le m$$

where c_{ij} = 1 if there is a coin in cell (i,j), and c_{ij} = 0 otherwise



Solution (cont.)

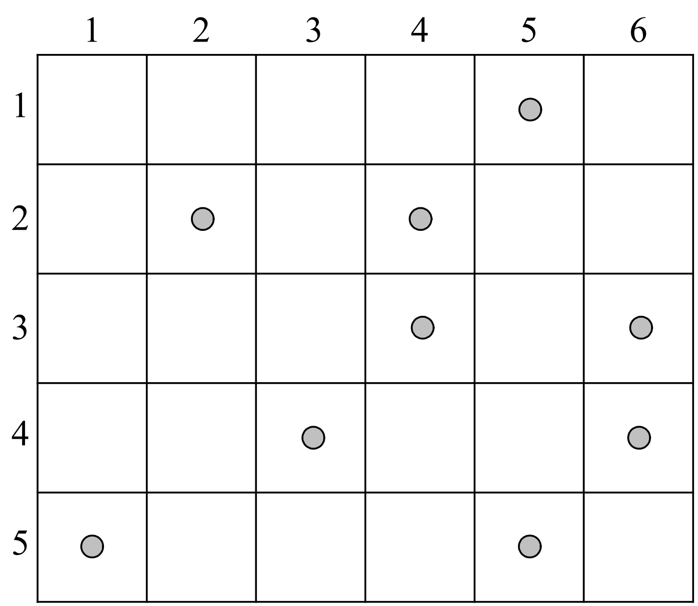
- $F(i, j) = \max\{F(i-1, j),\ F(i, j-1)\} + c_{ij}$
- $F(0, j) = 0$ for $1 \leq j \leq m$ and $F(i, 0) = 0$ for $1 \leq i \leq n$.



Solution (cont.)

Bottom-up calculation

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



	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**

Robot Coin Collection

```
ALGORITHM RobotCoinCollection(C[1..n, 1..m])
  // Robot coin collection using dynamic programming
  // Input: Matrix C[1..n, 1..m] with elements equal to 1 and 0 for
  //        cells with and without coins, respectively.
  // Output: Returns the maximum collectible number of coins
  F[1, 1] ← C[1, 1]
  for j ← 2 to m do
    F[1, j] ← F[1, j - 1] + C[1, j]
  for i ← 2 to n do
    F[i, 1] ← F[i - 1, 1] + C[i, 1]
    for j ← 2 to m do
      F[i, j] ← max(F[i - 1, j], F[i, j - 1]) + C[i, j]
  return F[n, m]
```

Complexity? Θ(*nm*) time, Θ(*nm*) space

Dynamic programming TL/DR

- Understand the problem
- Make a recursive definition of the problem
  - What are the *subproblems*?
  - How are the subproblems *related*?
- Decide how to store the results of subproblems
- Algorithm to calculate/fill in the data structure

Dynamic Programming: Transitive Closure

(Chapter 8)

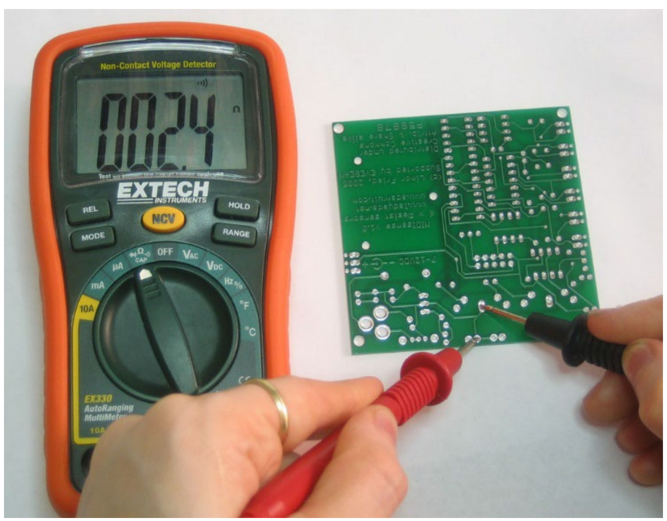
Transitive Closure

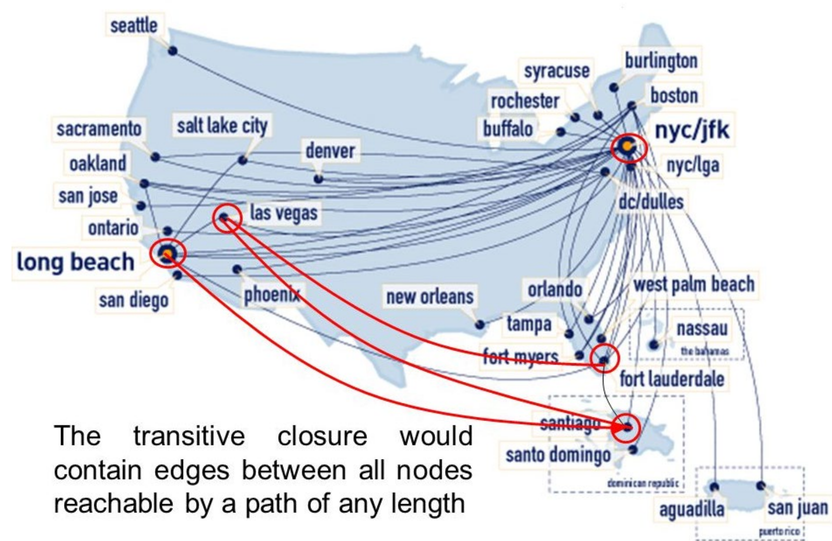
- What nodes are reachable from other nodes?
- Problem:
  - given a directed unweighted graph G with n vertices, find all paths that exist from vertices v_i to v_j, for all $1 \leq (i, j) \leq n$

- Note: this problem is always solved with an adjacency matrix graph representation

Transitive Closure

- Applications:
  - Testing digital circuits, reachability testing

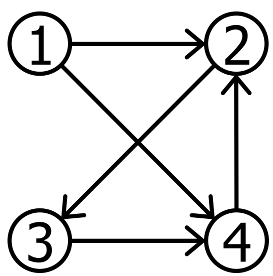




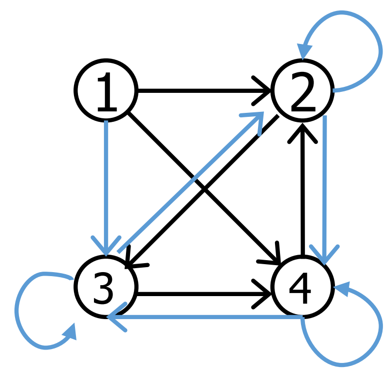
The transitive closure would contain edges between all nodes reachable by a path of any length

Transitive Closure

- Idea of algorithm:
  - Create a new graph where every *edge* represents a *path* in the original



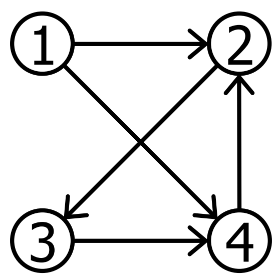
	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0



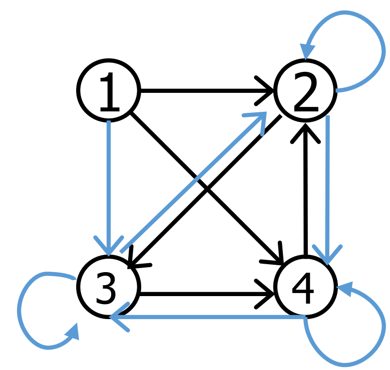
	1	2	3	4
1	0	1	1	1
2	0	1	1	1
3	0	1	1	1
4	0	1	1	1

Transitive Closure

- Idea of algorithm:
  - Create a new graph where every ***edge*** represents a ***path*** in the original



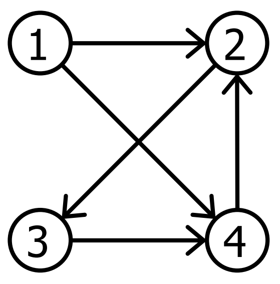
	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0



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

Transitive Closure example

- Consider the graph below, and its corresponding adjacency matrix ...



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

- We call this initial matrix R^0.
  - For convenience here we are using a 1-based array: A[1..n][1..n]

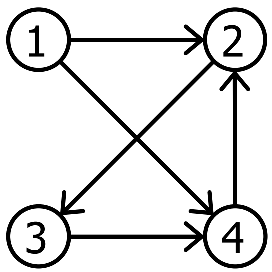
Transitive Closure

Step 1:

- select row 1 and column 1
- for all i,j

    if (i,1) = 1 and (1,j) = 1 then set (i,j) ← 1

In this case there are no changes.



j

i

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

At the end of this step this matrix is known as R^1.

Transitive Closure

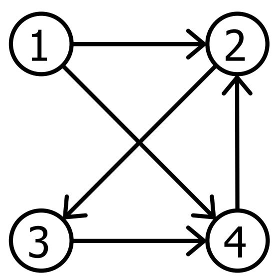
Step 2:

- select row 2 and column 2
- for all i,j

  if (i,2) = 1 and (2,j) = 1 then set (i,j) ← 1

Notice:

(1,2) == (2,3) == 1 → set (1,3) ← 1
(4,2) == (2,3) == 1 → set (4,3) ← 1



	1	2	3	4
1	0	1	**1**	1
2	0	0	1	0
3	0	0	0	1
4	0	1	**1**	0

At the end of this step this matrix is known as R^2.

Transitive Closure

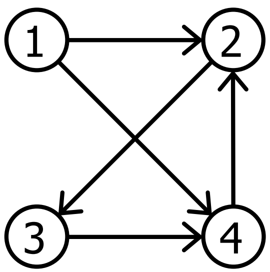
Step 3:

- select row 3 and column 3
- for all i,j

    if (i,3) = 1 and (3,j) = 1 then set (i,j) ← 1

Notice:

(1,3) == (3,4) == 1 → set (1,4) ← 1
(2,3) == (3,4) == 1 → set (2,4) ← 1
(4,3) == (3,4) == 1 → set (4,4) ← 1



j

i		1	2	3	4
	1	0	1	1	**1**
	2	0	0	1	**1**
	3	0	0	0	1
	4	0	1	1	**1**

At the end of this step this matrix is known as R^3.

Transitive Closure

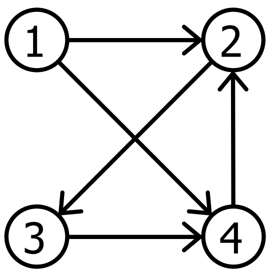
Step 4:

- select row 4 and column 4
- for all i,j

  if (i,4) = 1 and (4,j) = 1 then set (i,j) ← 1

Notice:

(2,4) == (4,2) == 1 → set (2,2) ← 1
(3,4) == (4,2) == 1 → set (3,2) ← 1
(3,4) == (4,3) == 1 → set (3,3) ← 1



j

i		1	2	3	4
	1	0	1	1	1
	2	0	**1**	1	1
	3	0	**1**	**1**	1
	4	0	1	1	1

At the end of this step this matrix is known as R^4. It is the "Transitive Closure on G". The existence of a one in cell (i,j) tells us that there exists a path from i to j in G.

Warshall's algorithm

- Maybe the best thing about this algorithm is its simplicity

```
Warshall(G[1..n, 1..n])
    for k ← 1 to n {
        for i ← 1 to n {
            for j ← 1 to n {
                if ( G[i,k] == G[k,j] == 1 ) {
                    set G[i,j] ← 1
                }
            }
        }
    }
```

Efficiency: ?

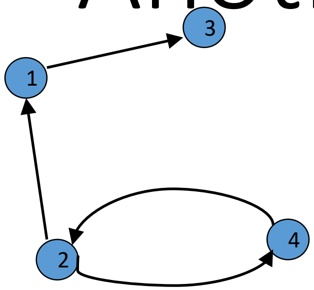
Why is this Dynamic Prog?

- 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

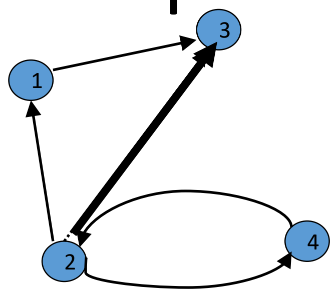
- So: It finds the paths from simpler subproblems

- Also produces the result bottom-up from a matrix recording as you go

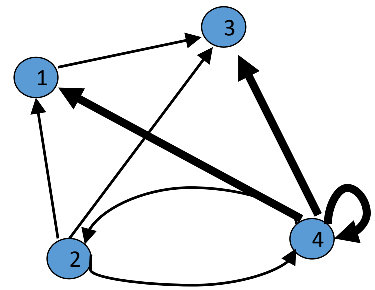
Another Example



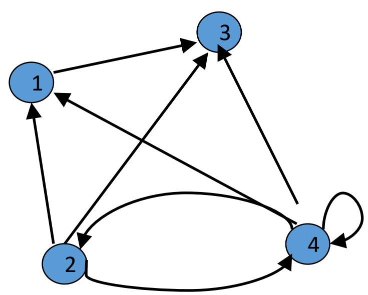
$$R^{(0)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$



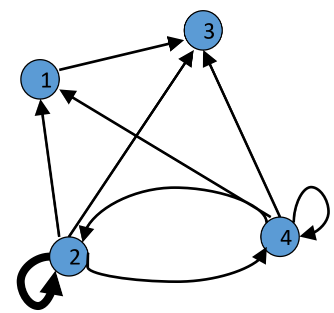
$$R^{(1)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$



$$R^{(2)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix}$$



$$R^{(3)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$



$$R^{(4)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

Dynamic Programming: All-pairs shortest paths

(Chapter 8)

All-pairs shortest paths

- Problem:
  - Given a directed weighted graph G with n vertices, find the shortest path from any vertex v_i to any other vertex v_j, for all $1 \leq (i,j) \leq n$
- Note: this problem is always solved with an adjacency matrix graph representation
- Applications: This problem occurs in lots of applications – notably in computer games, where it is useful to find shortest paths before planning movement.

Floyd's algorithm

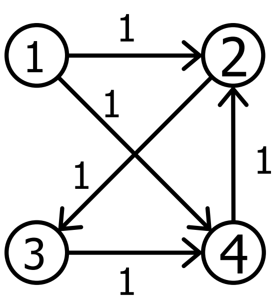
- Like Warshall's algorithm, but different:
  - Add weight (or cost) to each edge in the initial graph
  - When no edge exists the weight is ∞
    - "You can't get there from here" (yet)
  - Set the weights on the diagonal to be 0
    - The shortest path from a vertex to itself should be 0

Floyd's algorithm

- And the real key change:
  - Warshall's algorithm says this:
    - if (i,k) == (k,j) == 1 then set (i,j) $\leftarrow$ 1
    - i.e. If you can get from i to k and from k to j, then now you can get from i to j
  - …but for Floyd's we will say this:
    - if (i,k) + (k,j) < (i,j) then set (i,j) $\leftarrow$ (i,k) + (k,j)
    - i.e. If i-k-j costs less than the (so far) best known path from i to j, then update the best known path"

Floyd’s algorithm

- Initial representation of the graph



j

i

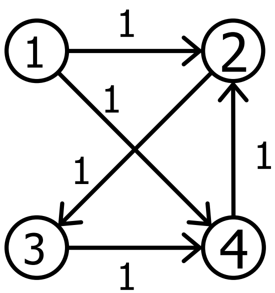
	1	2	3	4
1	0	1	∞	1
2	∞	0	1	∞
3	∞	∞	0	1
4	∞	1	∞	0

Floyd’s Algorithm

Step 1:

- select row 1 and column 1
- for all i,j

  if (i,1) + (1,j) < (i,j) then set (i,j) ← (i,1) + (1,j)



j

i

	1	2	3	4
1	0	1	∞	1
2	∞	0	1	∞
3	∞	∞	0	1
4	∞	1	∞	0

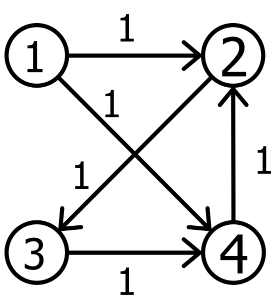
In this case there are no changes.

Floyd's Algorithm

Step 2:

- select row 2 and column 2
- for all i,j

    if (i,2) + (2,j) < (i,j) then set (i,j) ← (i,2) + (2,j)



	1	2	3	4
1	0	1	**2**	1
2	∞	0	1	∞
3	∞	∞	0	1
4	∞	1	**2**	0

Notice:

$(1,2) + (2,3) < \infty \rightarrow$ set $(1,3) \leftarrow 2$

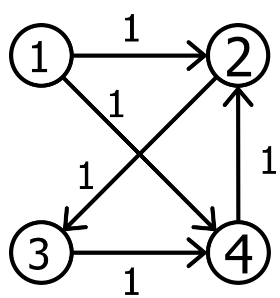
$(4,2) + (2,3) < \infty \rightarrow$ set $(4,3) \leftarrow 2$

Floyd’s Algorithm

Step 3:

- select row 3 and column 3
- for all i,j

  if (i,3) + (3,j) < (i,j) then set (i,j) ← (i,3) + (3,j)



j

i

	1	2	3	4
1	0	1	2	1
2	∞	0	1	**2**
3	∞	∞	0	1
4	∞	1	2	0

There is only one change this time ...

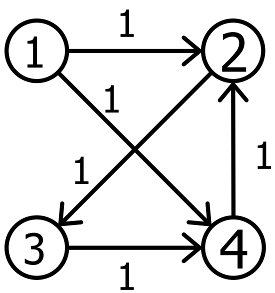
(2,3) + (3,4) < ∞ → set (2,4) ← 2

Floyd’s Algorithm

Step 4:

- select row 4 and column 4
- for all i,j

  if (i,4) + (4,j) < (i,j) then set (i,j) ← (i,4) + (4,j)



j

i

	1	2	3	4
1	0	1	2	1
2	∞	0	1	2
3	∞	**2**	0	1
4	∞	1	2	0

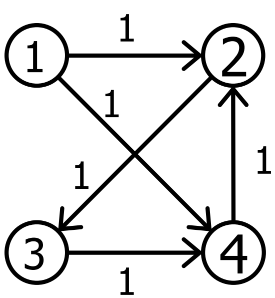
Again, only one change ...

$(3,4) + (4,2) < \infty \rightarrow$ set $(3,2) \leftarrow 2$

Floyd’s Algorithm

This time our solution gives the shortest paths from any i to any j.

We can see that the none of 2,3, or 4 have paths to 1, and the algorithm has discovered two hop paths for 1→3, 2→4, 3→2, and 4→3,

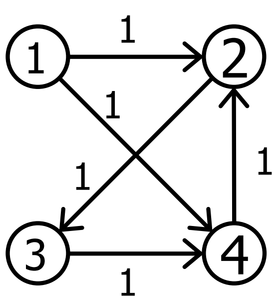


j

i	1	2	3	4
1	0	1	2	1
2	∞	0	1	2
3	∞	2	0	1
4	∞	1	2	0

Floyd’s Algorithm

- The final matrix gives the shortest paths from any i to any j.
- Observations:
  - You can’t get from anywhere to 1
  - The algorithm discovered two-hop paths for $1 \rightarrow 3$, $2 \rightarrow 4$, $3 \rightarrow 2$, and $4 \rightarrow 3$



j

i		1	2	3	4
	1	0	1	2	1
i	2	∞	0	1	2
	3	∞	2	0	1
	4	∞	1	2	0

Floyd’s Algorithm (pseudocode)

```
Floyd(G[1..n, 1..n])
    for k ← 1 to n {
        for i ← 1 to n {
            for j ← 1 to n {
                cost_thru_k ← G[i,k] + G[k,j]
                if ( cost_thru_k < G[i,j] ) {
                    set G[i,j] ← thru_k
                }
            }
        }
    }
```

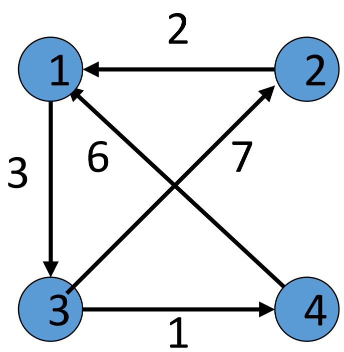
This middle section is referred to as the "Warshall Parameter". We can change it around to solve a variety of problems.

Efficiency: ?

How is this DP?

- (Like Warshall's) the "sub-problem" is that it is finding shortest paths that use vertices 1..k as hopping points
- One new vertex (k) is added into the picture at each step
- After each step, you have a matrix D_k that gives the best (yet) distance through those vertices

Another Example



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

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

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

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

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