Dynamic Programming

(Chapter 8)

Dynamic Programming

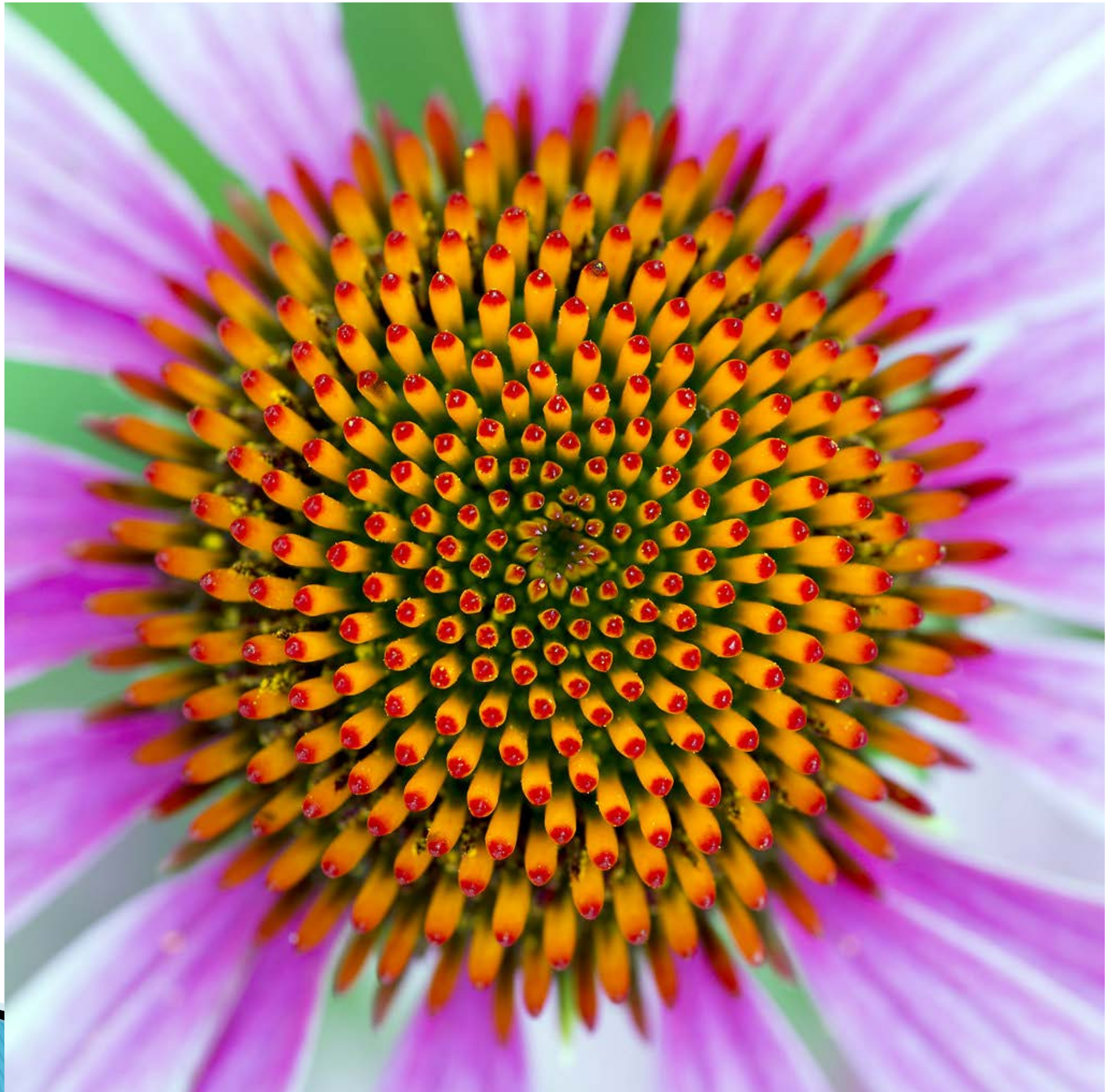
- Fibonacci numbers
- Robot Coin Collecting
- Transitive Closure (Warshall)
- All Pairs Shortest Path (Floyd)
- Knapsack Problem

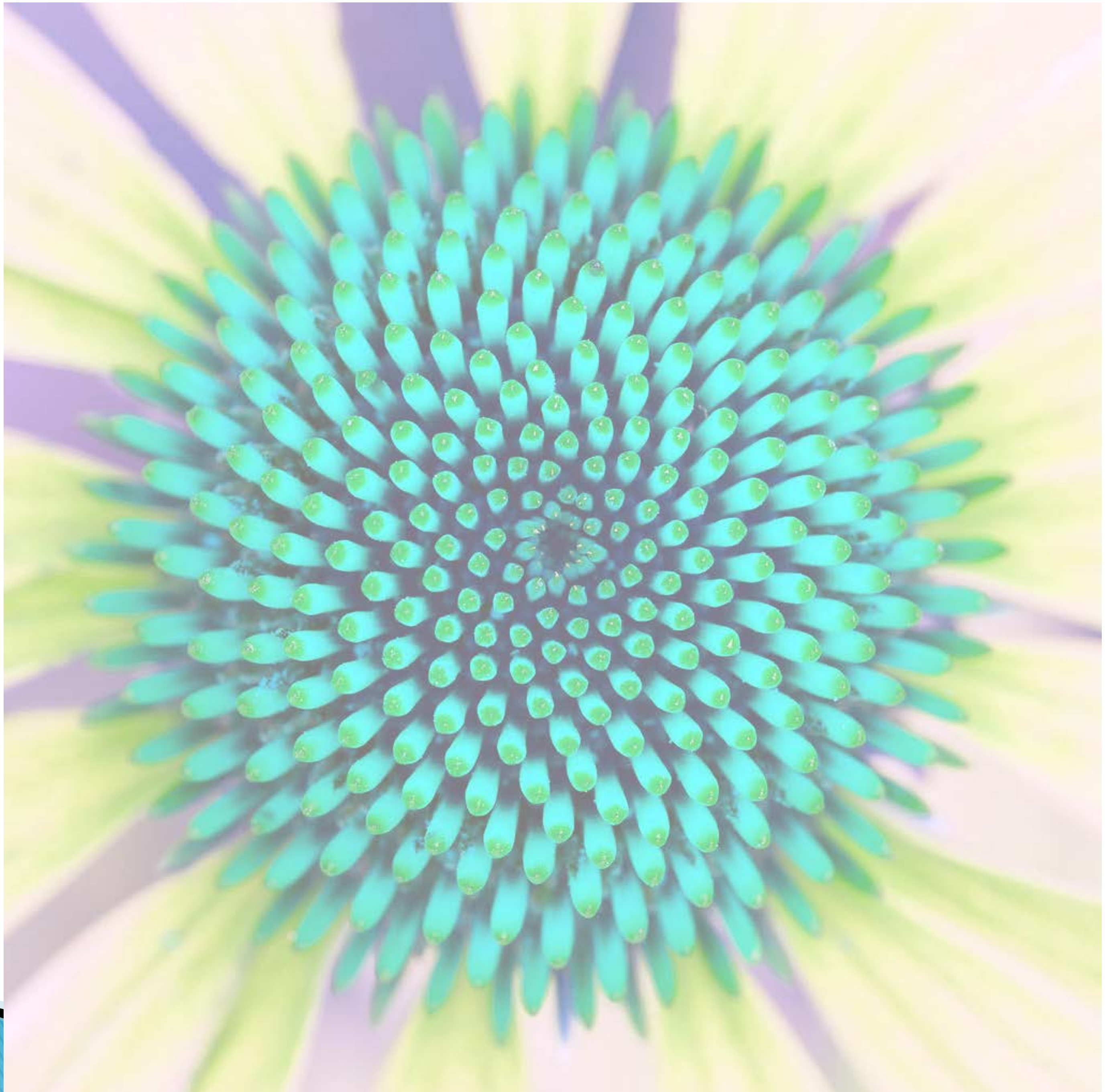
Dynamic Programming

- Dynamic Programming is a general algorithm design technique for solving optimization problems

- Invented by American mathematician Richard Bellman in the1950s

- The key point: *remembering* recursively-defined solutions to subproblems and using them to solve the problem





Fibonacci numbers

- Fibonacci numbers:

  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`
  where each number is the sum of the preceding two.

  fib(0) = 0
  fib(1) = 1
  fib(n) = fib(n−1) + fib(n−2)

Fibonacci numbers (Divide & Conquer)

```
fib (n) {
   if n < 2
      f = n;
   else
      f = fib(n−1) + fib(n−2)
   return f
}
```

Execution tree:

fib(n)

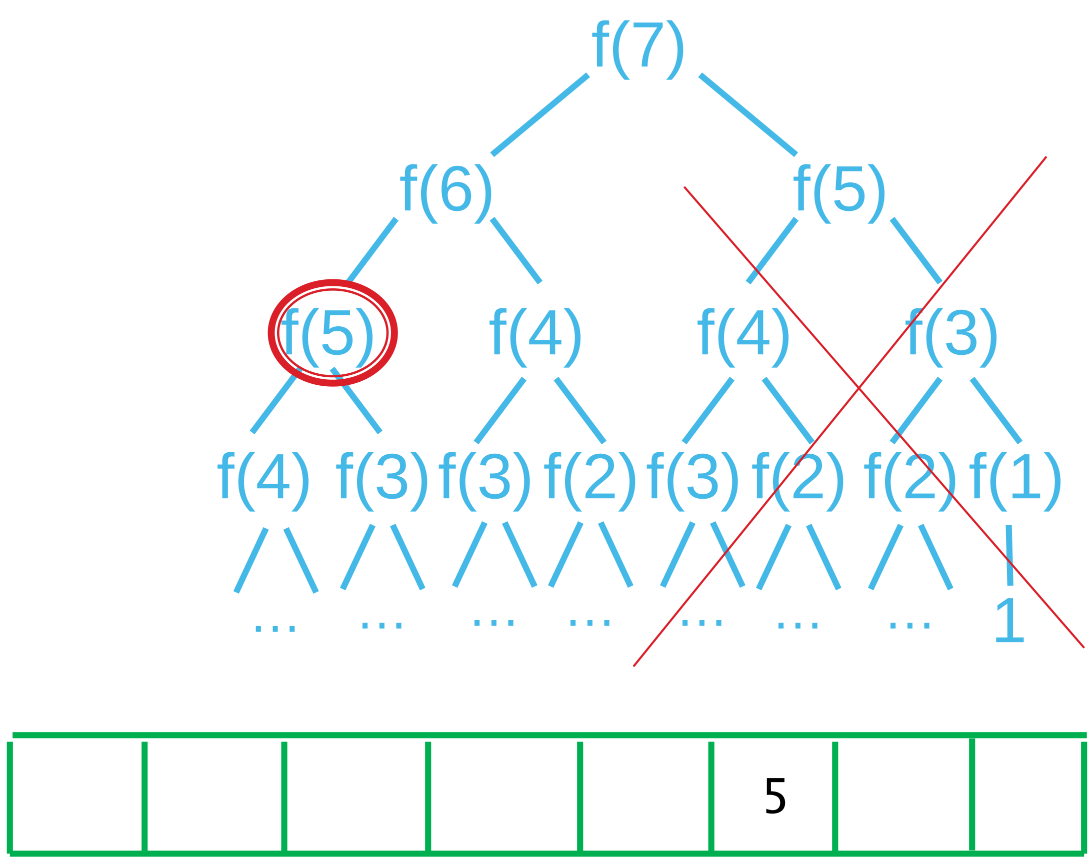
fib(n−1) + fib(n−2)

fib(n−2) + fib(n−3) fib(n−3) + fib(n−4)

… …

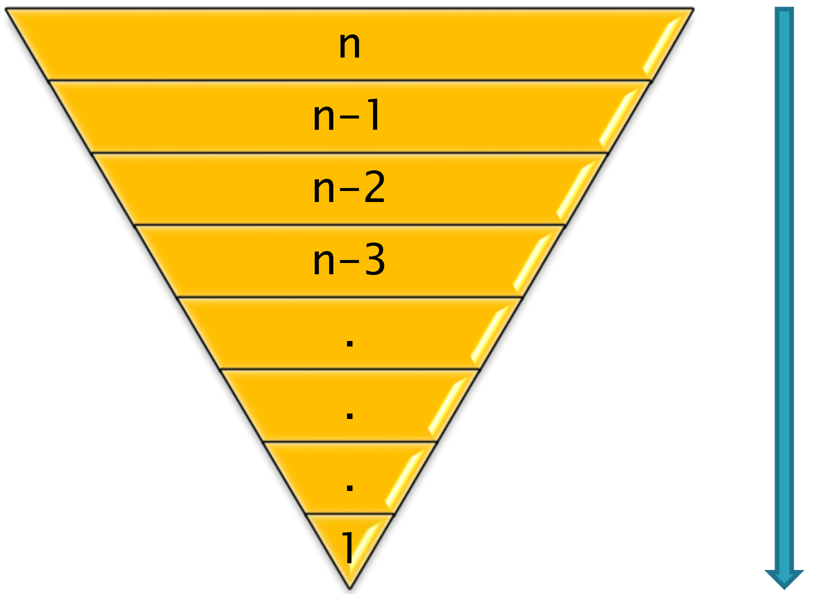
F(n) takes exponential time to compute.

Fibonacci numbers



DP, (top-down)

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



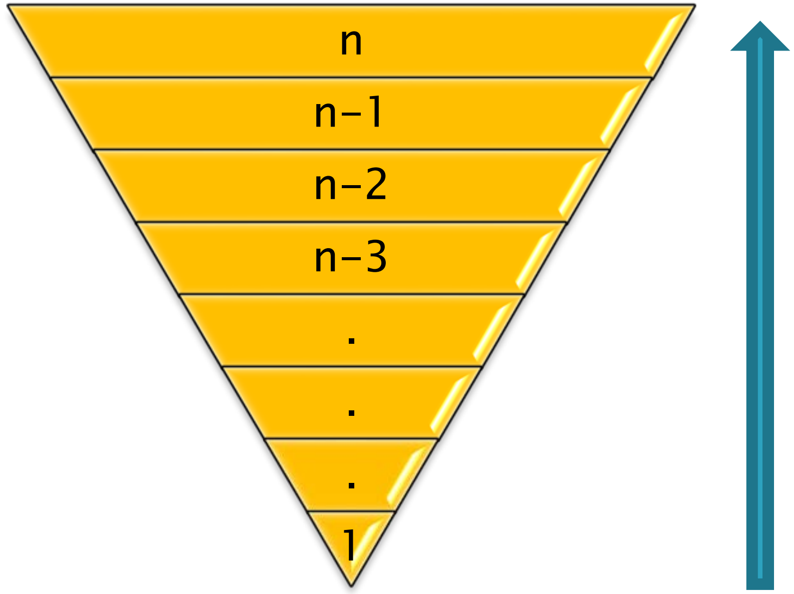
top-down (Recursive)

memo							
	0	**1**	**1**	**. . .**	***fib*(*n*-2)**	***fib*(*n*-1)**	***fib*(*n*)**

Efficiency:

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

DP, (bottom-up)



bottom-up

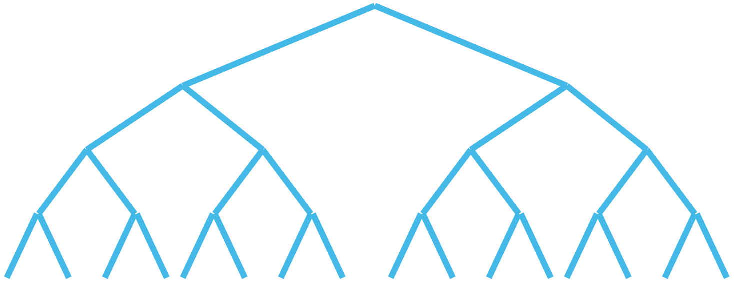
```
fib (n) {
   memo[0] = 0;
   memo[1] = 1;
   for i ← 2 to n do
      memo [i] = memo[i-1] + memo[i-2]
   return memo[n]
}
```

memo							
memo	**0**	**1**	**1**	. . .	*fib*(*n*-2)	*fib*(*n*-1)	*fib*(*n*)

Efficiency:
- time: O(n)
- space: Needs an extra array

Dynamic Programming

- Exactly the same as divide-and-conquer … but store the solutions to sub-problems for possible reuse.
- A good idea if many of the sub-problems are the same as one another.

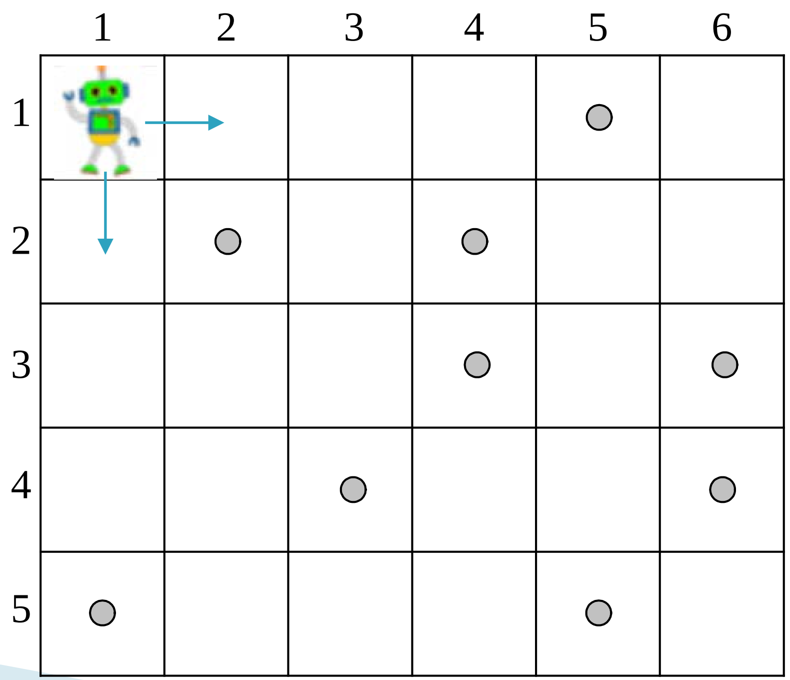


Dynamic Programming

- Fibonacci numbers
- Robot Coin Collecting
- Transitive Closure (Warshall)
- All Pairs Shortest Path (Floyd)
- Knapsack Problem

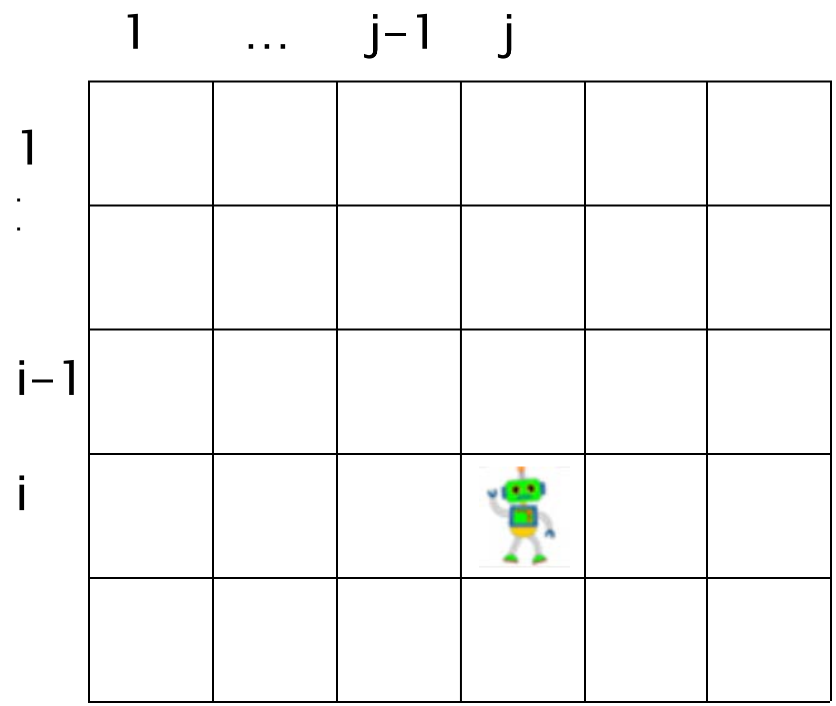
Robot Coin Collecting

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.



Solution

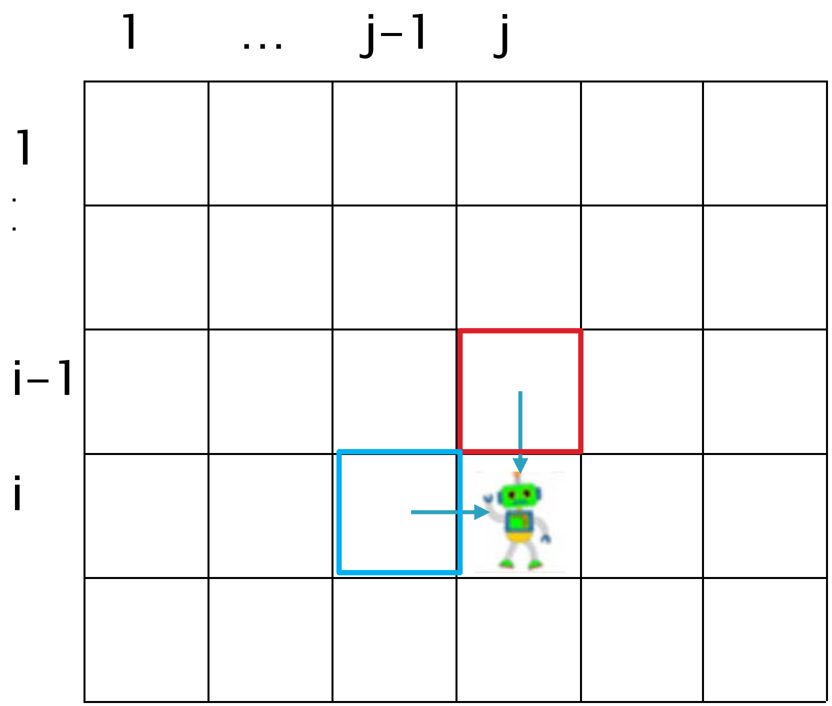
- Let F(i,j) be the largest number of coins the robot can collect and bring to cell (i,j) in the *ith* row and *j*th column.



Solution

The largest number of coins that can be brought to cell (*i*,*j*):

from the left neighbor ? F(i, j-1)
from the neighbor above? F(i-1, j)

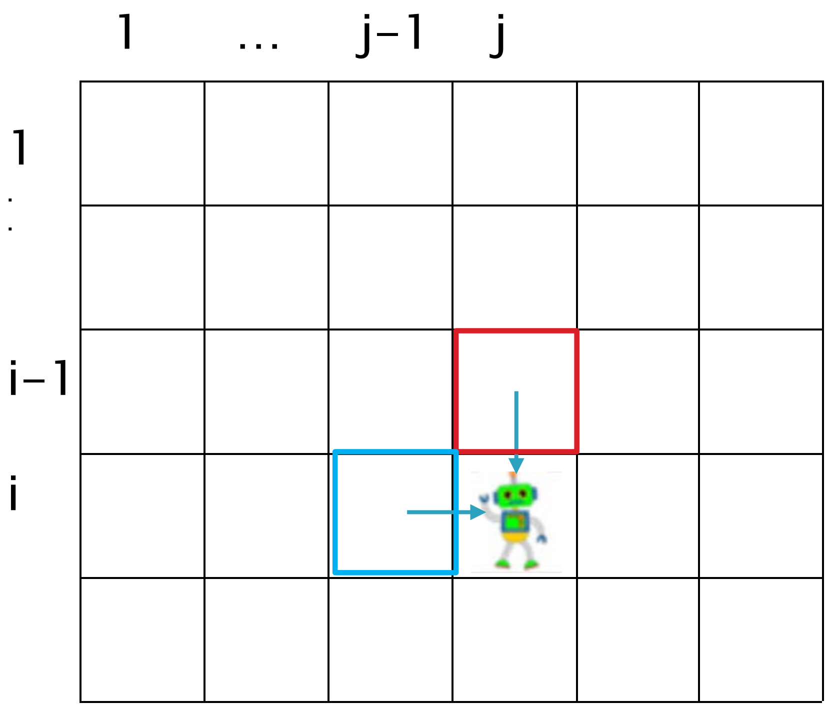


Solution

The recurrence:

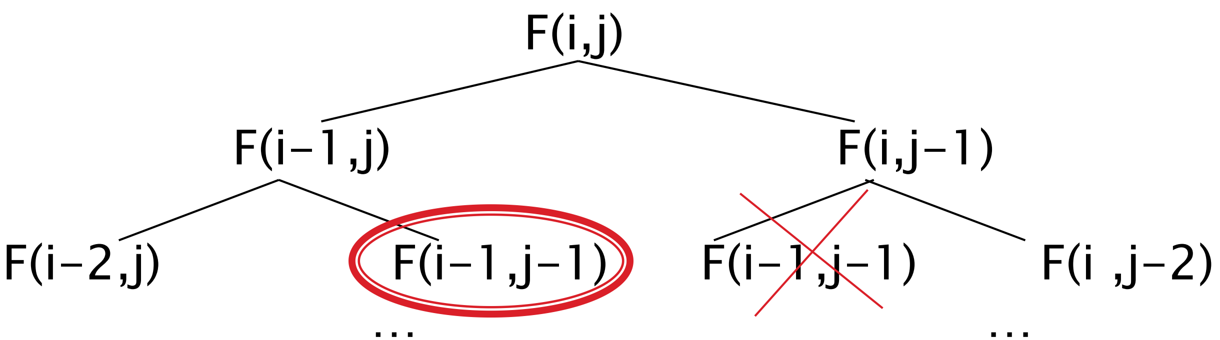
$$F(i, j) = \max\{F(i-1, j),\ F(i, j-1)\} + c_{ij} \quad \text{for } 1 \leq i \leq n,\ 1 \leq j \leq m$$

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



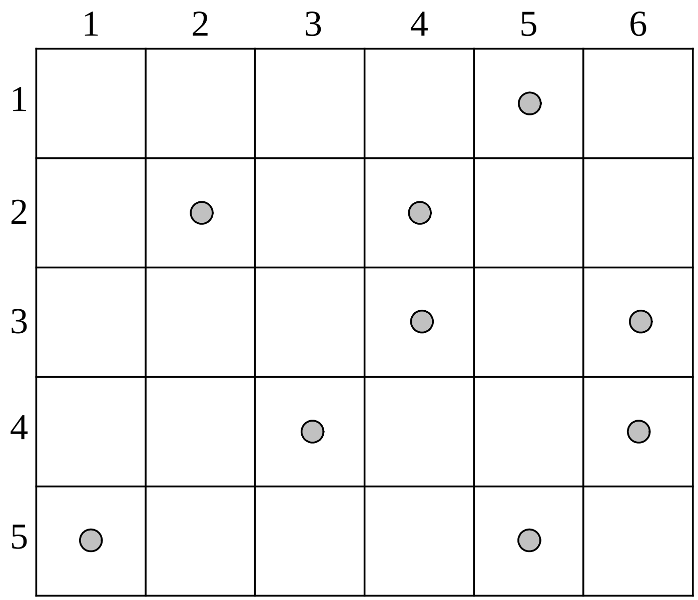
Solution (cont.)

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



Solution (cont.)

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



C

	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

F

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? $\Theta(nm)$ time, $\Theta(nm)$ space

DP Algos: General Principles

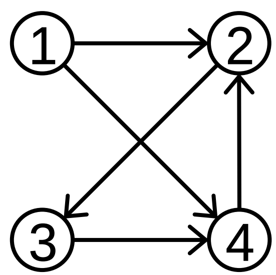
- Step 1:
  - Decompose problem into simpler sub-problems
- Step 2:
  - Express solution in terms of sub-problems
- Step 3:
  - Use table to compute optimal value bottom-up
- Step 4:
  - Find optimal solution based on steps 1-3

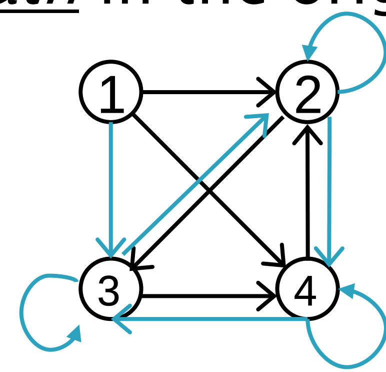
Dynamic Programming

- Fibonacci numbers
- Robot Coin Collecting
- Transitive Closure (Warshall)
- All Pairs Shortest Path (Floyd)
- Knapsack Problem

Transitive Closure

- Idea:
  - Start with a graph, create a new graph where every *edge* is obtained from a *path* in the original



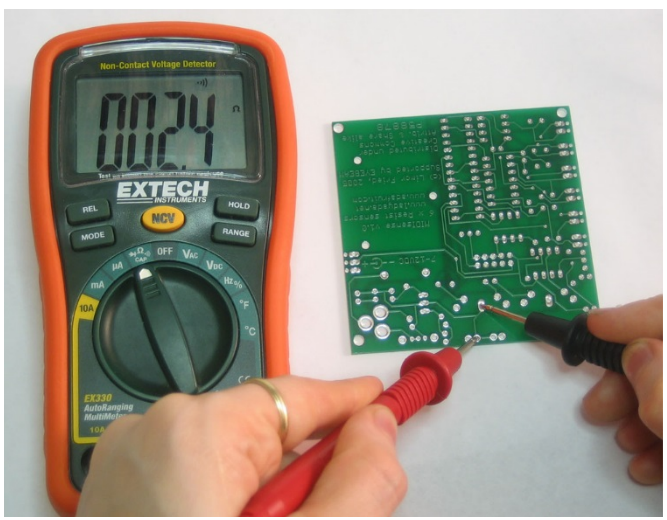


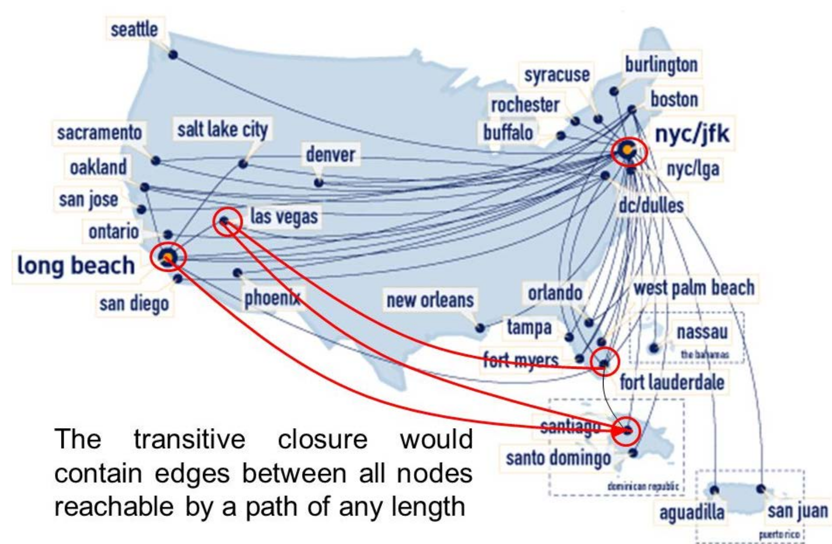
	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0

	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

- Applications:
  - Testing digital circuits, reachability testing





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

Transitive Closure

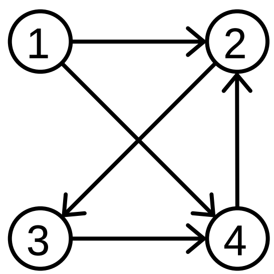
Problem:

- given a directed unweighted graph G with n vertices, find all vertices v_i that have paths to any other vertex v_j, for all $1 \le (i,j) \le n$

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

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. We will define it as A[1..n][1..n]

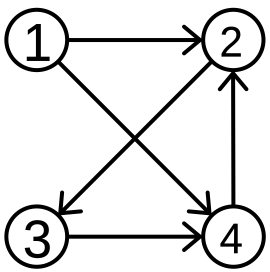
Transitive Closure

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

		1	2	3	4
	1	0	1	0	1
i	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

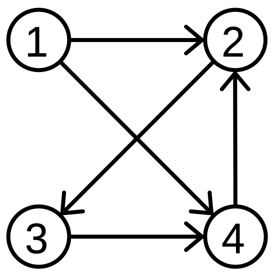
Step 2:

- 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



j

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

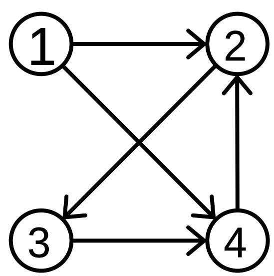
Step 3:

- 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:

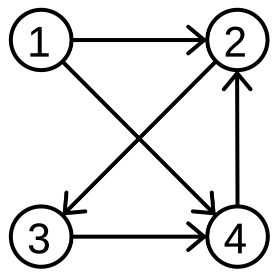
– select row 4 and column 4

– 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 (pseudocode)

- The best part about this algorithm is its simplicity
- Look at the simple pseudocode:

```
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: $O(n^3)$

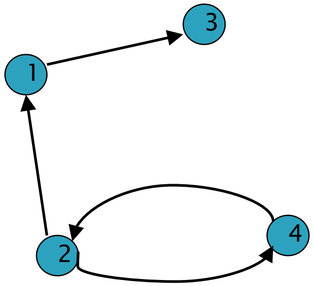
Why is this Dynamic Prog?

- 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

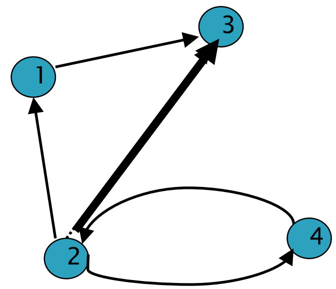
- So: It finds the paths from simpler subproblems

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

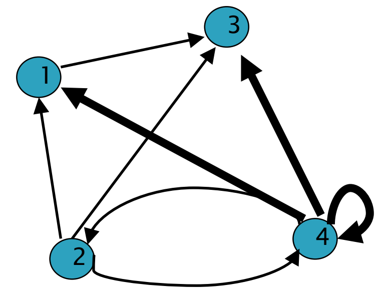
Another Example



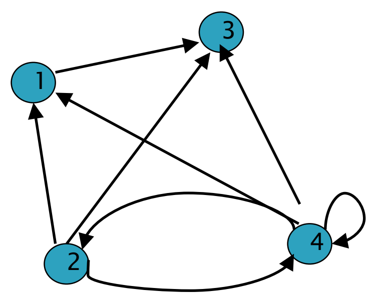
$R^{(0)}$
$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$



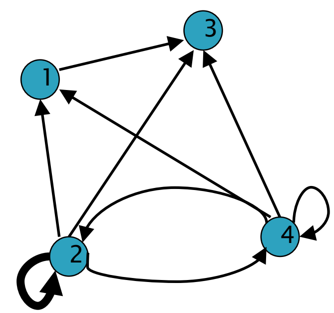
$R^{(1)}$
$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$



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



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



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

Dynamic Programming

- Fibonacci numbers
- Robot Coin Collecting
- Transitive Closure (Warshall)
- All Pairs Shortest Path (Floyd)
- Knapsack Problem

All Pairs Shortest Path Problem

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

Application: 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

Floyd's algorithm is a dynamic programming solution to APSP.

It is a variation on Warshall's algorithm.

We call it a DP algorithm because it incrementally finds the shortest paths by finding shortest paths using only vertices from 1..k. At each step, you find a matrix D^k that gives the distance through those vertices.

Floyd's Algorithm

We will start by considering Warshall's algorithm, with the following changes:

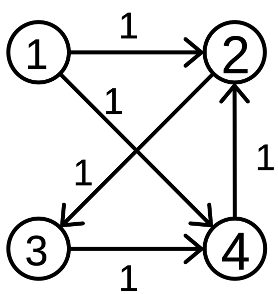
- we will add edge weights of w to each edge in the initial graph
- when no edge exists we will set the weight to be ∞ in the adj matrix
- we will set the weights on the diagonal to be 0, as the shortest path from a vertex to itself should be 0
- we will change the "Warshall Parameter" from ...

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

  ... to ...

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

Floyd's Algorithm



j

i

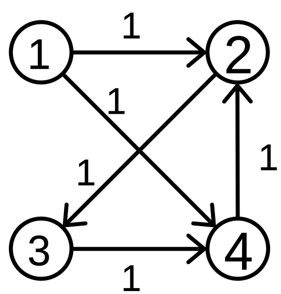
	1	2	3	4
1	0	1	∞	1
2	∞	0	1	∞
3	∞	∞	0	1
4	∞	1	∞	0

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

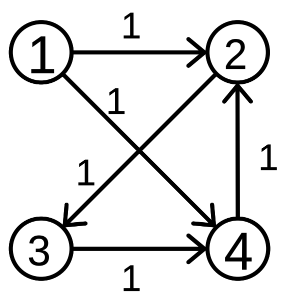
In this case there are no changes.

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)



j

i

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

Notice:

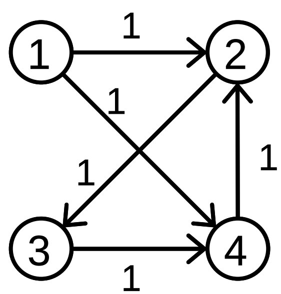
(1,2) + (2,3) < ∞ → set (1,3) ← 2
(4,2) + (2,3) < ∞ → set (4,3) ← 2

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 …

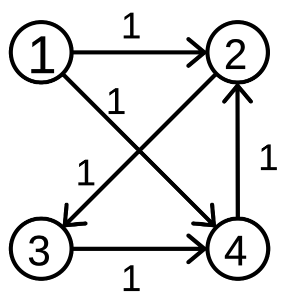
(2,3) + (3,4) < ∞ → set (2,4) ← 2

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)



i		j			
		1	2	3	4
	1	0	1	2	1
	2	∞	0	1	2
	3	∞	**2**	0	1
	4	∞	1	2	0

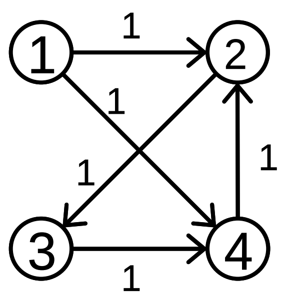
Again, only one change …

(3,4) + (4,2) < ∞ → set (3,2) ← 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

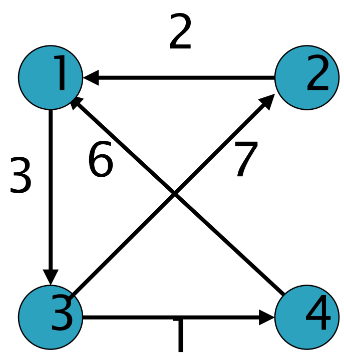
Floyd's Algorithm (pseudocode)

- this algorithm is known as Floyd's Algorithm, and it solves APSP
  - The efficiency here is clearly $O(n^3)$

```
Floyd(G[1..n, 1..n])
   for k ← 1 to n {
       for i ← 1 to n {
           for j ← 1 to n {
               thru_k ← G[i,k] + G[k,j]
               if ( 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 related problems.

Another Example



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

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

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

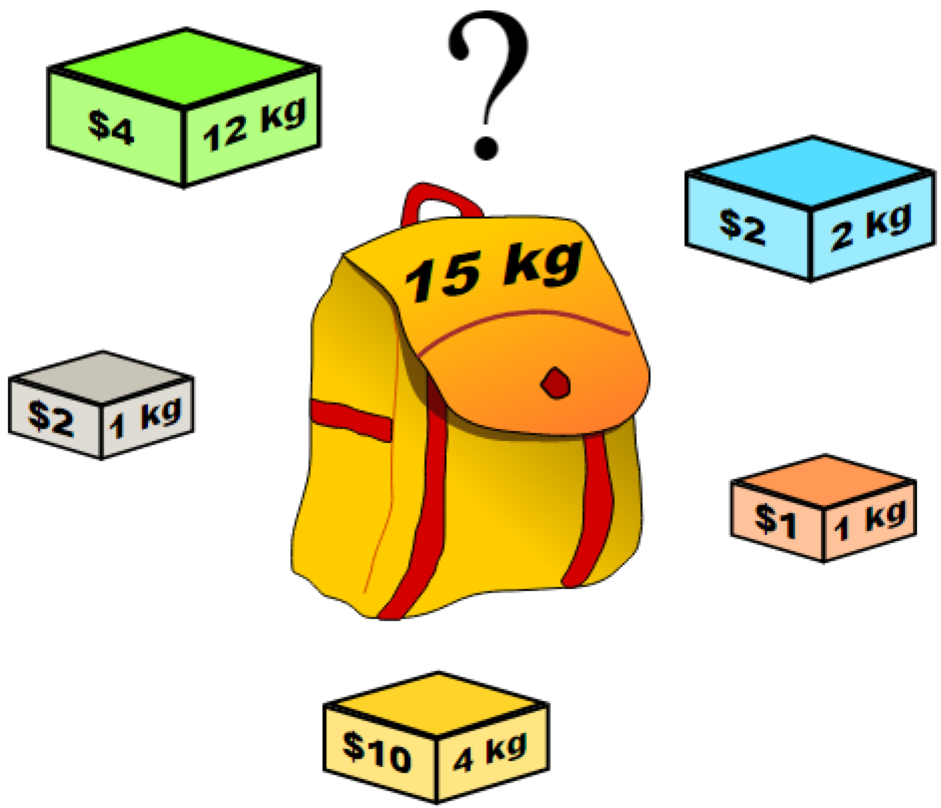
$$D^{(3)} = \begin{matrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{matrix}$$

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

Dynamic Programming

- Fibonacci numbers
- Robot Coin Collecting
- Transitive Closure (Warshall)
- All Pairs Shortest Path (Floyd)
- Knapsack Problem

Knapsack Problem



Knapsack Problem

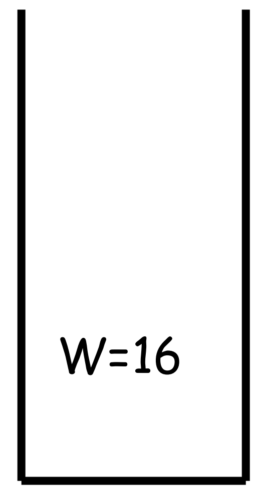
- Input:
  - weights: $w_1 \quad w_2 \; \dots \; w_n$
  - values: $v_1 \quad v_2 \; \dots \; v_n$
  - a knapsack of capacity W

- Goal:
  - Find most valuable subset of the items that fit into the knapsack

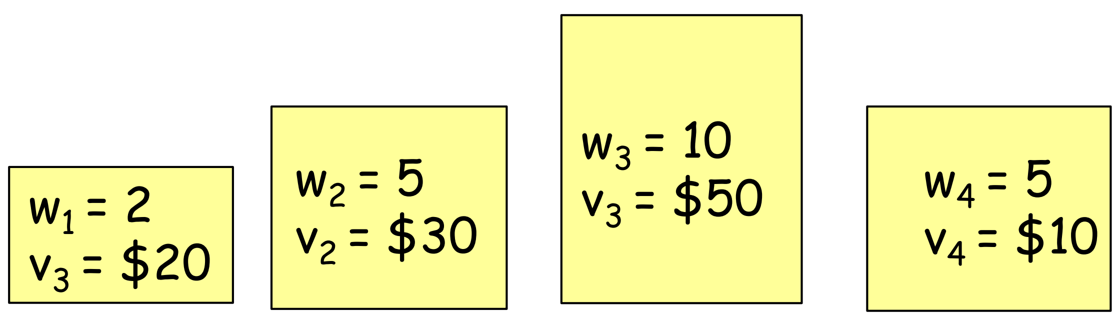
Knapsack Problem

Example: Knapsack capacity W=16

item	weight	value
1	2	\$20
2	5	\$30
3	10	\$50
4	5	\$10



knapsack



Knapsack Problem (Brute Force)

- Generate all possible subsets of the n items
- Compute total weight of each subset
- Identify feasible subsets
- Find the subset of the largest value

Efficiency?

Need to generate *all subsets*. For n items, there are 2^n subsets. So this is a $O(2^n)$ algorithm.

We would like something more efficient…

DP Solution to Knapsack

- Step 1: Identifying sub-problems
- $V[i, w]$= max value for items 1..i with max weight w

$$V[i, w]$$

$$V[i-1, w] \qquad\qquad v_i + V[i-1, w-w_i])$$

DP Solution to Knapsack

- Step 2: Recursive definition of optimal sol.
  - Initial values:

$$V[0, w] = 0 \qquad \text{for } 0 \leq w \leq W,$$

  - Recursive step:

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w-w_i])$$

$$\text{for } 1 \leq i \leq n, 0 \leq w \leq W.$$

Example

Input data:

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

So there are 4 elements

Max weight W=5

Example (2)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

for w = 0 to W

V[0,w] = 0

Example (3)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

for i = 1 to n

    V[i,0] = 0

Example (4)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

$i=1$

$v_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

Can not fit first item with max weight 1…

Because the first item has a weight of 2.

Example (5)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

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

$i=1$

$v_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

Now you can fit it

Because the weight is less than 2, the max weight in this column

Example (6)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

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

$i=1$

$v_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

You only have one item.. And it weighs less than 3…

Example (7)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

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

$i=1$

$v_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

You only have one item.. And it weighs less than 3…

Example (8)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

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

$i=1$

$v_i=3$

$w_i=2$

$w=5$

$w-w_i=3$

You only have one item.. And it weighs less than 3…

Example (9)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

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

$i=2$

$v_i=4$

$w_i=3$

$w=1$

$w-w_i=-2$

Neither item 1 or 2 weighs less than 1.

So you can not put anything in the bag.

Example (10)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

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

$i=2$

$v_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

$w_i > w$

So you can not add item 2 to the sack. Copy weight from above.

Example (11)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

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

$i=2$

$v_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

Now $w_i <= w$ so item 2 can be part of the solution

Also: The value of 2 is greater than the solved subproblem to the left… so putting 2 in the bag is the best solution.

Example (12)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

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

$i=2$

$v_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

Same

Example (13)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

$i=2$

$v_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

Now… you can fit item 2 in addition to item 1…

Example (14)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	**0**	**3**	**4**		
4	0					

$i=3$

$v_i=5$

$w_i=4$

$w= 1..3$

Item 3 is too big for the first colums…

Example (15)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

$i=3$

$v_i=5$

$w_i=4$

$w= 4$

$w- w_i=0$

Can fit in column 4… value is bigger than that to the left

Example (16)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

$i=3$

$v_i=5$

$w_i=4$

$w= 5$

$w- w_i=1$

Solution with {1,2} is better… so don't replace it

Example (17)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	**0**	**3**	**4**	**5**	

$i=4$

$v_i=6$

$w_i=5$

$w= 1..4$

Weight = 5… can just copy first 4 columns

Example (18)

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	**7**

$i=4$

$b_i=6$

$w_i=5$

$w= 5$

$w- w_i=0$

Adding 5 still not the optimal choice.

Knapsack DP Algorithm

KnapSack(v, w, n, W)
{
  for ($w = 0$ to W) $V[0, w] = 0$;
  for ($i = 1$ to n)
    for ($w = 0$ to W)
      if ($w[i] \leq w$)
        $V[i, w] = \max\{V[i-1, w], v[i] + V[i-1, w - w[i]]\}$;
      else
        $V[i, w] = V[i-1, w]$;
  return $V[n, W]$;
}

- Running time?
  - Loop to n… nested loop to W
  - O(nW)

What does the algo tell us?

- Tracing through the example… we learned that the maximum value you can fit in the bag is 7

- But what items are included?
  - Can extend the algorithm to trace back and find out

How to find actual Knapsack Items

- All of the information we need is in the table.
- $V[n,W]$ is the maximal value of items that can be placed in the Knapsack.
- Let i=n and k=W

  if $V[i,k] \neq V[i-1,k]$ then

  mark the i^{th} item as in the knapsack

  $i = i-1$, $k = k-w_i$

  else

  $i = i-1$ // Assume the i^{th} item is <u>not</u> in the knapsack

  // Could it be in the optimally packed knapsack?

Finding the Items

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=4

k= 5

v_i=6

w_i=5

$V[i,k] = 7$

$V[i-1,k] = 7$

i=n, k=W

while i,k > 0

  if $V[i,k] \neq V[i-1,k]$ then

    mark the i^{th} item as in the knapsack

    $i = i-1,\ k = k-w_i$

  else

    $i = i-1$

Finding the Items (2)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=4

k= 5

v_i=6

w_i=5

$V[i,k] = 7$

$V[i\text{–}1,k] = 7$

i=n, k=W

while i,k > 0

    if $V[i,k] \neq V[i\text{–}1,k]$ then

        mark the i^{th} item as in the knapsack

        $i = i\text{–}1,\ k = k\text{-}w_i$

    else

        $i = i\text{–}1$

Finding the Items (3)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=3$
$k=5$
$v_i=5$
$w_i=4$
$V[i,k] = 7$
$V[i-1,k] = 7$

i=n, k=W
while i,k > 0
    if $V[i,k] \neq V[i-1,k]$ then
        mark the i^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Finding the Items (4)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=2

k= 5

v_i=4

w_i=3

$V[i,k] = 7$

$V[i-1,k] = 3$

$k - w_i = 2$

i=n, k=W
while i,k > 0
    if $V[i,k] \neq V[i-1,k]$ then
        mark the i^{th} item as in the knapsack
        $i = i-1,\ k = k-w_i$
    else
        $i = i-1$

Finding the Items (5)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=1

k= 2

v_i=3

w_i=2

$V[i,k] = 3$

$V[i-1,k] = 0$

$k - w_i = 0$

i=n, k=W

while i,k > 0

    if $V[i,k] \neq V[i-1,k]$ then

        mark the i^{th} item as in the knapsack

        $i = i-1,\ k = k-w_i$

    else

        $i = i-1$

Finding the Items (6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=0

k= 0

The optimal knapsack should contain {1, 2}

i=n, k=W

while i,k > 0

  if $V[i,k] \neq V[i-1,k]$ then

    mark the i^{th} item as in the knapsack

    $i = i-1$, $k = k-w_i$

  else

    $i = i-1$

Another Example

Example: Knapsack of capacity $W = 5$

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

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

capacity j

	0	1	2	3	4	5
0						
1						
2						
3						
4						

Another Solution

Example: Knapsack of capacity $W = 5$

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

capacity j

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

$w_1 = 2,\ v_1 = 12$

$w_2 = 1,\ v_2 = 10$

$w_3 = 3,\ v_3 = 20$

$w_4 = 2,\ v_4 = 15$

Dynamic Programming

- Fibonacci numbers
- Robot Coin Collecting
- Transitive Closure (Warshall)
- All Pair Shortest Path (Floyd)
- Knapsack Problem