Dynamic Programming

Stefan D. Bruda

CS 317, Fall 2025

Memoization and dynamic programming

- Recursive implementations can be expensive:

algorithm RecFib(n)**:**
    if $n \leq 1$ **then return** n
    else return RecFib($n-1$) + RecFib($n-2$)

$O(2^n)$ time
$O(1)$+recursion space

Memoization and dynamic programming

- Recursive implementations can be expensive:

algorithm RecFib(n)**:**
  if $n \leq 1$ **then return** n
  else return RecFib($n-1$) + RecFib($n-2$)

$O(2^n)$ time
$O(1)$+recursion space

- Memoization: Remember intermediate results

algorithm MemFib(n)**:**
  if $n = 0 \vee n = 1$ **then return** 1
  else
    if F_n is undefined **then**
      $F_n \leftarrow$ MemFib($n-1$) + MemFib($n-2$)
    return F_n

$O(n)$ time
$O(n)$ (+recursion) space

Memoization and dynamic programming

- Recursive implementations can be expensive:

algorithm RecFib(n)**:**
  if $n \leq 1$ **then return** n
  else return RecFib($n-1$) + RecFib($n-2$)

$O(2^n)$ time
$O(1)$+recursion space

- Memoization: Remember intermediate results

algorithm MemFib(n)**:**
  if $n = 0 \vee n = 1$ **then return** 1
  else
    if F_n is undefined **then**
      $F_n \leftarrow \text{MemFib}(n-1) + \text{MemFib}(n-2)$
    return F_n

$O(n)$ time
$O(n)$ (+recursion) space

- Dynamic programming: Remember intermediate results explicitly

algorithm DynFib(n)**:**
  $F_0 \leftarrow 1$; $F_1 \leftarrow 1$
  for $i = 1$ **to** n **do** $F_n \leftarrow F_{n-1} + F_{n-2}$
  return F_n

$O(n)$ time
$O(n)$ space

Memoization and dynamic programming

- Recursive implementations can be expensive:

algorithm RecFib(n):
    if $n \leq 1$ **then return** n
    else return RecFib($n-1$) + RecFib($n-2$)

$O(2^n)$ time
$O(1)$+recursion space

- Memoization: Remember intermediate results

algorithm MemFib(n):
    if $n = 0 \vee n = 1$ **then return** 1
    else
        if F_n is undefined **then**
            $F_n \leftarrow$ MemFib($n-1$) + MemFib($n-2$)
        return F_n

$O(n)$ time
$O(n)$ (+recursion) space

- Dynamic programming: Remember intermediate results explicitly

algorithm DynFib(n):
    $F_0 \leftarrow 1$; $F_1 \leftarrow 1$
    for $i = 1$ **to** n **do** $F_n \leftarrow F_{n-1} + F_{n-2}$
    return F_n

$O(n)$ time
$O(n)$ space

- Can also consider remembering intermediate results only as needed

algorithm DynFib(n):
    $prev \leftarrow 1$; $curr \leftarrow 1$
    for $i = 1$ **to** n **do**
        $next \leftarrow prev + curr$
        $prev \leftarrow curr$
        $curr \leftarrow next$
    return $curr$

$O(n)$ time
$O(1)$ space

Dynamic programming

- Dynamic programming = recursion without repetition
  1. Formulate the problem recursively
     - Use a bottom-up approach (starting from the base cases)
  2. Build the dynamic programming solution
     1. Identify subproblems
     2. Choose memoization data structure
     3. Identify dependencies and so find evaluation order
- Often but not always applicable to optimization problems
  - But in this case only for problems that satisfy the principle of optimality: An optimal solution to the problem contains optimal solutions to subproblems

0/1 KNAPSACK

- Given $w = \langle w_1, \ldots, w_n \rangle$ and $p = \langle p_1, \ldots, p_n \rangle$, find $x = \langle x_1, \ldots, x_n \rangle$, $x_i \in \{0, 1\}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Bottom-up recursive solution ($O(2^n)$):

algorithm RECKNAPSACK(i, C, n, p, w)**:** (handle the i-th object)

```
if i > n then return (0, ⟨⟩)
else
    (p_−, X_−) ← RECKNAPSACK(i + 1, C, n, p, w)    (do not pick item i)
    if w_i ≤ C then
        (p_+, X_+) ← RECKNAPSACK(i + 1, C − w_i, n, p, w)    (pick item i)
    else
        (p_+, X_+) ← (0, ⟨⟩)    (we cannot pick item i so we set profit to minimum)
    return MAXFST({(p_−, ⟨0⟩ + X_−), (p_+ + w_i, ⟨1⟩ + X_+)})
```

0/1 KNAPSACK

- Given $w = \langle w_1, \ldots, w_n \rangle$ and $p = \langle p_1, \ldots, p_n \rangle$, find $x = \langle x_1, \ldots, x_n \rangle$, $x_i \in \{0, 1\}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Bottom-up recursive solution ($O(2^n)$):

algorithm RECKNAPSACK(i, C, n, p, w): (handle the i-th object)
  if $i > n$ **then return** $(0, \langle\rangle)$
  else
    $(p_-, X_-) \leftarrow$ RECKNAPSACK$(i + 1, C, n, p, w)$ (do not pick item i)
    if $w_i \leq C$ **then**
      $(p_+, X_+) \leftarrow$ RECKNAPSACK$(i + 1, C - w_i, n, p, w)$ (pick item i)
    else
      $(p_+, X_+) \leftarrow (0, \langle\rangle)$ (we cannot pick item i so we set profit to minimum)
    return MAXFST$(\{(p_-, \langle 0 \rangle + X_-), (p_+ + w_i, \langle 1 \rangle + X_+)\})$

- Memoization structure must contain information related to the remaining items and the remaining capacity $\Rightarrow$ table of item $\times$ capacity
  - Increment of capacity smaller than the smallest w_i
- Each subproblem (entry in the table) depends on the "upper" and "upper-left" subproblems
- Table filled in top to bottom, left to right

0/1 KNAPSACK (CONT'D)

- Dynamic programming solution (for maximum profit):

algorithm KNAPSACK(C, n, p, w)**:**

```
for i = 1 to n do  P_{i,0} ← 0
for j = 1 to C do  P_{0,j} ← 0
for i = 1 to n do
    for j = 1 to C do
        if w_i > j then  P_{i,j} ← P_{i-1,j}
        else  P_{i,j} ← max{P_{i-1,j}, p_i + P_{i-1,j-w_i}}
```

- Table P filled top to bottom, left to right

0/1 KNAPSACK (CONT'D)

- Dynamic programming solution (for maximum profit):

algorithm KNAPSACK(C, n, p, w)**:**

```
for i = 1 to n do  P_{i,0} ← 0
for j = 1 to C do  P_{0,j} ← 0
for i = 1 to n do
    for j = 1 to C do
        if w_i > j then  P_{i,j} ← P_{i-1,j}
        else  P_{i,j} ← max{P_{i-1,j}, p_i + P_{i-1,j-w_i}}
```

- Table P filled top to bottom, left to right

- Example: $w = \langle 2, 1, 3, 2 \rangle$, $p = \langle 12, 10, 20, 15 \rangle$, $C = 5$

			c_i					
w_i	p_i	j	0	1	2	3	4	5
0	0	0	0	0	0	0	0	0
2	12	1	0	0	12	12	12	12
1	10	2	0	10	12	22	22	22
3	20	3	0	10	12	22	30	32
2	15	4	0	10	12	25	30	37

0/1 KNAPSACK (CONT'D)

- Obtaining the actual solution:

algorithm KNAPSACKTRACE**:**

```
j ← C
for i = n downto 1 do
    if P_{i,j} = P_{i−1,j} then
        x_i ← 0
    else
        x_i ← 1
        j ← j − w_i
```

			c_i					
w_i	p_i	j	0	1	2	3	4	5
0	0	0	0	0	0	0	0	0
2	12	1	0	0	**12**	12	12	12
1	10	2	0	10	12	**22**	22	22
3	20	3	0	10	12	**22**	30	32
2	15	4	0	10	12	25	30	**37**

Solution: $\langle 1, 1, 0, 1 \rangle$

- Running time: $\Theta(n \times C)$

0/1 KNAPSACK (CONT'D)

- Obtaining the actual solution:

algorithm KNAPSACKTRACE**:**

```
j ← C
for i = n downto 1 do
    if P_{i,j} = P_{i−1,j} then
        x_i ← 0
    else
        x_i ← 1
        j ← j − w_i
```

			c_i					
w_i	p_i	j	0	1	2	3	4	5
0	0	0	0	0	0	0	0	0
2	12	1	0	0	**12**	12	12	12
1	10	2	0	10	12	**22**	22	22
3	20	3	0	10	12	**22**	30	32
2	15	4	0	10	12	25	30	**37**

Solution: $\langle 1, 1, 0, 1 \rangle$

- Running time: $\Theta(n \times C) \rightarrow$ no better than $\Theta(2^n)$!

0/1 KNAPSACK (CONT'D)

- Obtaining the actual solution:

algorithm KNAPSACKTRACE**:**

```
j ← C
for i = n downto 1 do
    if P_{i,j} = P_{i−1,j} then
        x_i ← 0
    else
        x_i ← 1
        j ← j − w_i
```

			c_i					
w_i	p_i	j	0	1	2	3	4	5
0	0	0	0	0	0	0	0	0
2	12	1	0	0	**12**	12	12	12
1	10	2	0	10	12	**22**	22	22
3	20	3	0	10	12	**22**	30	32
2	15	4	0	10	12	25	30	**37**

Solution: $\langle 1, 1, 0, 1 \rangle$

- Running time: $\Theta(n \times C) \rightarrow$ no better than $\Theta(2^n)$!
- Many problems are very similar to 0/1 Knapsack
  - Example (subset sum): Given an array $A_{1\ldots n}$ of positive integers and an integer T, does any subarray of A sums up to T
    - Subproblems: $SS(i, t) =$ TRUE iff some subset of A sums to t
    - Recursive solution:

$$SS(i,t) = \begin{cases} \text{TRUE} & \text{if } t = 0 \\ \text{FALSE} & \text{if } i > n \\ SS(i+1,t) & \text{if } t < A_i \\ SS(i+1,t) \vee SS(i+1,t-A_i) & \text{otherwise} \end{cases}$$

    - Memoization structure: table $S_{1\ldots n,0\ldots T}$
    - Evaluation order: rows bottom to top, arbitrary order in a row

Matrix chain multiplication

- Given $M = M_1 \times M_2 \times \ldots \times M_n$ with the dimensions of the matrices stored in $r_{0\ldots n}$, such that each M_i has r_{i-1} rows and r_i columns, find how to bracket the matrix multiplications to minimize the total number of multiplications
  - Example: $r = \langle 2, 10, 1, 3 \rangle$ that, is $A(2 \times 10) \times B(10 \times 1) \times C(1 \times 3)$
    - $A \times (B \times C)$ needs 90 integer multiplications
    - $(A \times B) \times C$ needs 26 integer multiplications (faster)

Matrix chain multiplication

- Given $M = M_1 \times M_2 \times \ldots \times M_n$ with the dimensions of the matrices stored in $r_{0\ldots n}$, such that each M_i has r_{i-1} rows and r_i columns, find how to bracket the matrix multiplications to minimize the total number of multiplications
  - Example: $r = \langle 2, 10, 1, 3 \rangle$ that, is $A(2 \times 10) \times B(10 \times 1) \times C(1 \times 3)$
    - $A \times (B \times C)$ needs 90 integer multiplications
    - $(A \times B) \times C$ needs 26 integer multiplications (faster)
  - Subproblems: $m_{i,j}$ is the cost of computing $M_i \times \ldots \times M_j$

Matrix chain multiplication

- Given $M = M_1 \times M_2 \times \ldots \times M_n$ with the dimensions of the matrices stored in $r_{0\ldots n}$, such that each M_i has r_{i-1} rows and r_i columns, find how to bracket the matrix multiplications to minimize the total number of multiplications
  - Example: $r = \langle 2, 10, 1, 3 \rangle$ that, is $A(2 \times 10) \times B(10 \times 1) \times C(1 \times 3)$
    - $A \times (B \times C)$ needs 90 integer multiplications
    - $(A \times B) \times C$ needs 26 integer multiplications (faster)
  - Subproblems: $m_{i,j}$ is the cost of computing $M_i \times \ldots \times M_j$
  - Recursive solution:

$$m_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}(m_{i,k} + m_{k+1,j} + r_{i-1} \times r_k \times r_j) & \text{if } i < j \end{cases}$$

  - Memoization structure: table $m_{1\ldots n,1\ldots n}$ to store the result of subproblems
  - Evaluation order: by diagonal top to bottom with arbitrary order within a diagonal

algorithm MatrixChainMult: $O(n^3)$

```
for i = 1 to n do  m_ii ← 0
for r = 1 to n − 1 do
    for i = 1 to n − r do
        j ← i + r
        m_i,j ← min_{i≤k<j}(m_i,k + m_{k+1,j} + r_{i−1} × r_k × r_j)
```

Matrix chain multiplication example

$$\underset{(5\times 3)}{M_1} \times \underset{(3\times 1)}{M_2} \times \underset{(1\times 4)}{M_3} \times \underset{(4\times 6)}{M_4}$$

$r = \langle 5, 3, 1, 4, 6\rangle$

1. $m_{11} = 0$, $m_{22} = 0$, $m_{33} = 0$, $m_{44} = 0$

Matrix chain multiplication example

$$\underset{(5\times 3)}{M_1} \times \underset{(3\times 1)}{M_2} \times \underset{(1\times 4)}{M_3} \times \underset{(4\times 6)}{M_4}$$
$$r = \langle 5, 3, 1, 4, 6\rangle$$

1. $m_{11} = 0$, $m_{22} = 0$, $m_{33} = 0$, $m_{44} = 0$
2. $m_{12} = m_{11} + m_{22} + 5 \times 3 \times 1 = 15$
   $m_{23} = m_{22} + m_{33} + 3 \times 1 \times 4 = 12$
   $m_{34} = m_{33} + m_{44} + 1 \times 4 \times 6 = 24$

Matrix chain multiplication example

$$\underset{(5\times 3)}{M_1} \times \underset{(3\times 1)}{M_2} \times \underset{(1\times 4)}{M_3} \times \underset{(4\times 6)}{M_4}$$
$$r = \langle 5, 3, 1, 4, 6\rangle$$

1. $m_{11} = 0,\ m_{22} = 0,\ m_{33} = 0,\ m_{44} = 0$
2. $m_{12} = m_{11} + m_{22} + 5 \times 3 \times 1 = 15$
   $m_{23} = m_{22} + m_{33} + 3 \times 1 \times 4 = 12$
   $m_{34} = m_{33} + m_{44} + 1 \times 4 \times 6 = 24$
3. $m_{13} = \min\{\ m_{11} + m_{23} + 5 \times 3 \times 4 = 72,$
   $\qquad m_{12} + m_{33} + 5 \times 1 \times 4 = 35\ \}\ (k = 2)$
   $m_{24} = \min\{\ m_{22} + m_{34} + 3 \times 1 \times 6 = 42,$
   $\qquad m_{23} + m_{44} + 3 \times 4 \times 6 = 84\ \}\ (k = 1)$

Matrix chain multiplication example

$$\underset{(5\times 3)}{M_1} \times \underset{(3\times 1)}{M_2} \times \underset{(1\times 4)}{M_3} \times \underset{(4\times 6)}{M_4}$$

$r = \langle 5, 3, 1, 4, 6\rangle$

1. $m_{11} = 0,\ m_{22} = 0,\ m_{33} = 0,\ m_{44} = 0$
2. $m_{12} = m_{11} + m_{22} + 5 \times 3 \times 1 = 15$
   $m_{23} = m_{22} + m_{33} + 3 \times 1 \times 4 = 12$
   $m_{34} = m_{33} + m_{44} + 1 \times 4 \times 6 = 24$
3. $m_{13} = \min\{\ m_{11} + m_{23} + 5 \times 3 \times 4 = 72,$
   $m_{12} + m_{33} + 5 \times 1 \times 4 = 35\ \}\ (k = 2)$
   $m_{24} = \min\{\ m_{22} + m_{34} + 3 \times 1 \times 6 = 42,$
   $m_{23} + m_{44} + 3 \times 4 \times 6 = 84\ \}\ (k = 1)$
4. $m_{14} = \min\{\ m_{11} + m_{24} + 5 \times 3 \times 6 = 132,$
   $m_{12} + m_{34} + 5 \times 1 \times 6 = 69,$
   $m_{13} + m_{44} + 5 \times 4 \times 6 = 155\ \}\ (k = 2)$

Matrix chain multiplication example

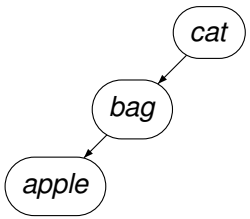
$$\underset{(5\times 3)}{M_1} \times \underset{(3\times 1)}{M_2} \times \underset{(1\times 4)}{M_3} \times \underset{(4\times 6)}{M_4}$$
$$r = \langle 5, 3, 1, 4, 6\rangle$$

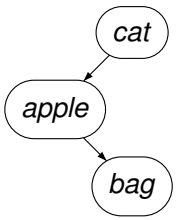
1. $m_{11} = 0,\ m_{22} = 0,\ m_{33} = 0,\ m_{44} = 0$
2. $m_{12} = m_{11} + m_{22} + 5 \times 3 \times 1 = 15$
   $m_{23} = m_{22} + m_{33} + 3 \times 1 \times 4 = 12$
   $m_{34} = m_{33} + m_{44} + 1 \times 4 \times 6 = 24$
3. $m_{13} = \min\{\ m_{11} + m_{23} + 5 \times 3 \times 4 = 72,$
   $m_{12} + m_{33} + 5 \times 1 \times 4 = 35\ \}\ (k = 2)$
   $m_{24} = \min\{\ m_{22} + m_{34} + 3 \times 1 \times 6 = 42,$
   $m_{23} + m_{44} + 3 \times 4 \times 6 = 84\ \}\ (k = 1)$
4. $m_{14} = \min\{\ m_{11} + m_{24} + 5 \times 3 \times 6 = 132,$
   $m_{12} + m_{34} + 5 \times 1 \times 6 = 69,$
   $m_{13} + m_{44} + 5 \times 4 \times 6 = 155\ \}\ (k = 2)$

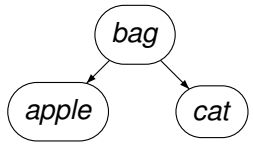
- Optimal multiplication: $(M_1 \times M_2) \times (M_3 \times M_4)$

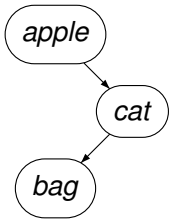
Optimal BST

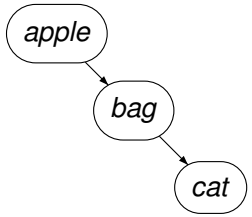
- Given n keywords along with their probabilities $p_1, p_2, \ldots, p_n$, store them in a binary search tree such that the average search time is minimized
  - Example: *cat* (0.1), *bag* (0.2), *apple* (0.7)
  - Sorted: *apple* (0.7), *bag* (0.2), *cat* (0.1)
  - Five different BST:









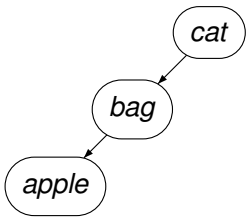


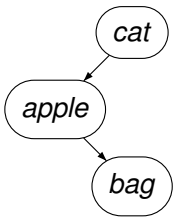
Average search time:

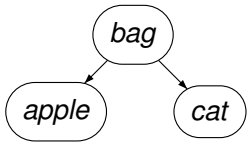
$0.1+2\times0.2+3\times0.7=2.6$ $0.1+2\times0.7+3\times0.2=2.1$ $0.2+2\times0.7+3\times0.1=1.8$ $0.7+2\times0.1+3\times0.2=1.5$ $0.7+2\times0.2+3\times0.1=1.4$

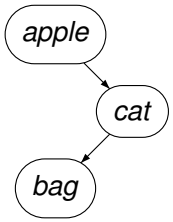
Optimal BST

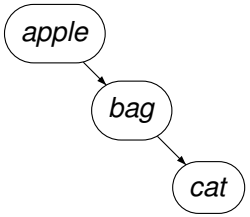
- Given n keywords along with their probabilities $p_1, p_2, \ldots, p_n$, store them in a binary search tree such that the average search time is minimized
  - Example: *cat* (0.1), *bag* (0.2), *apple* (0.7)
  - Sorted: *apple* (0.7), *bag* (0.2), *cat* (0.1)
  - Five different BST:











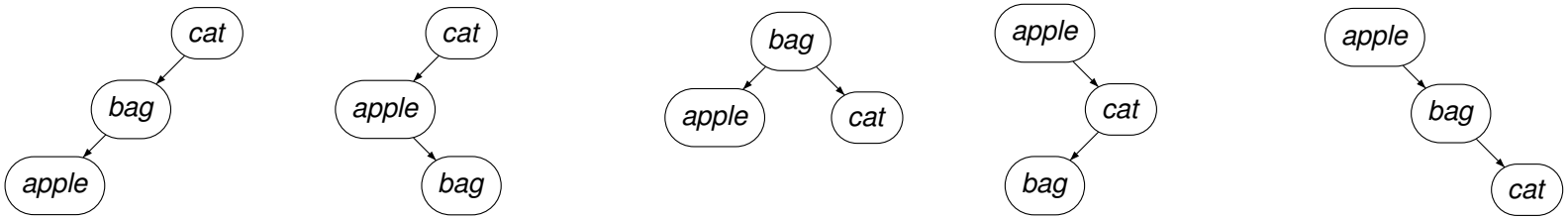
Average search time:

$0.1+2\times0.2+$ $3\times0.7=2.6$	$0.1+2\times0.7+$ $3\times0.2=2.1$	$0.2+2\times0.7+$ $3\times0.1=1.8$	$0.7+2\times0.1+$ $3\times0.2=1.5$	$0.7+2\times0.2+$ $3\times0.1=1.4$

- Subproblems: $A_{i,j}$ is the average search time for a BST with keywords from i to j

Optimal BST

- Given n keywords along with their probabilities $p_1, p_2, \ldots, p_n$, store them in a binary search tree such that the average search time is minimized
  - Example: *cat* (0.1), *bag* (0.2), *apple* (0.7)
  - Sorted: *apple* (0.7), *bag* (0.2), *cat* (0.1)
  - Five different BST:



Average search time:

$0.1+2\times0.2+$ $3\times0.7=2.6$	$0.1+2\times0.7+$ $3\times0.2=2.1$	$0.2+2\times0.7+$ $3\times0.1=1.8$	$0.7+2\times0.1+$ $3\times0.2=1.5$	$0.7+2\times0.2+$ $3\times0.1=1.4$

- Subproblems: $A_{i,j}$ is the average search time for a BST with keywords from i to j
- Recursive solution ($O(n^3)$ with memoization):

$$A_{i,j} = \begin{cases} p_i \text{ (root } i\text{)} & \text{if } i = j \\ 0 \text{ (null)} & \text{if } i > j \\ \min_{i \le k \le j}(A_{i,k-1} + A_{k+1,j} + \sum_{m=i}^{j} p_m) \text{ (root } k\text{)} & \text{if } i < j \end{cases}$$

- Obvious memoization
- Evaluation order: down by diagonal, arbitrary order within diagonal

Optimal BST example

apple (0.7), *bag* (0.2), *cat* (0.1)

1. $A_{11} = 0.7$/root 1, $A_{22} = 0.2$/root 2, $A_{33} = 0.1$/root 3, for all $i > j$: $A_{ij} = 0$/null node

Optimal BST example

apple (0.7), *bag* (0.2), *cat* (0.1)

1. $A_{11} = 0.7$/root 1, $A_{22} = 0.2$/root 2, $A_{33} = 0.1$/root 3,
   for all $i > j$: $A_{ij} = 0$/null node
2. $A_{12} = \min\{\ A_{10} + A_{22} + p_1 + p_2 = 1.1,$
   $A_{11} + A_{32} + p_1 + p_2 = 1.6\ \}$ $(k = 1,$ root 1$)$
   $A_{23} = \min\{\ A_{21} + A_{33} + p_2 + p_3 = 0.4,$
   $A_{22} + A_{43} + p_2 + p_3 = 0.5\ \}$ $(k = 2,$ root 2$)$

Optimal BST example

apple (0.7), *bag* (0.2), *cat* (0.1)

1. $A_{11} = 0.7$/root 1, $A_{22} = 0.2$/root 2, $A_{33} = 0.1$/root 3,
   for all $i > j$: $A_{ij} = 0$/null node
2. $A_{12} = \min\{\ A_{10} + A_{22} + p_1 + p_2 = 1.1,$
   $A_{11} + A_{32} + p_1 + p_2 = 1.6\ \}$ $(k = 1,$ root 1)
   $A_{23} = \min\{\ A_{21} + A_{33} + p_2 + p_3 = 0.4,$
   $A_{22} + A_{43} + p_2 + p_3 = 0.5\ \}$ $(k = 2,$ root 2)
3. $A_{13} = \min\{\ A_{10} + A_{23} + 1 = 1.4,$
   $A_{11} + A_{33} + 1 = 1.8,$
   $A_{12} + A_{43} + 1 = 2.1\ \}$ $(k = 1,$ root 1)

Optimal BST example

apple (0.7), *bag* (0.2), *cat* (0.1)

1. $A_{11} = 0.7$/root 1, $A_{22} = 0.2$/root 2, $A_{33} = 0.1$/root 3,
   for all $i > j$: $A_{ij} = 0$/null node
2. $A_{12} = \min\{\ A_{10} + A_{22} + p_1 + p_2 = 1.1,$
   $A_{11} + A_{32} + p_1 + p_2 = 1.6\ \}$ $(k = 1,$ root 1$)$
   $A_{23} = \min\{\ A_{21} + A_{33} + p_2 + p_3 = 0.4,$
   $A_{22} + A_{43} + p_2 + p_3 = 0.5\ \}$ $(k = 2,$ root 2$)$
3. $A_{13} = \min\{\ A_{10} + A_{23} + 1 = 1.4,$
   $A_{11} + A_{33} + 1 = 1.8,$
   $A_{12} + A_{43} + 1 = 2.1\ \}$ $(k = 1,$ root 1$)$

- Solution:
  - Root 1 (from A_{13}), left null (A_{10}), right contains 2 and 3 (A_{23})
  - Right child root 2 (from A_{23}), left null (A_{21}), right 3 (A_{33})

All-pairs shortest path

- Given a weighted (directed or undirected) graph $G = (V, E)$ wits $|V| = n$ and $|E| = m$, find the shortest path from each vertex to all other vertices
- Floyd's algorithm: Find shortest paths of rank k for increasing k
  - Uses the adjacency matrix $G_{1\ldots n,1\ldots n}$ of G
  - Path of rank k: path that only traverses vertices 1 to k (not counting the source and the destination)
  - Subproblems: $P_k = (A^k_{i,j}, \pi^k_{i,j})_{1 \le i \le n, 1 \le j \le n}$
    - $A^k_{i,j}$ is the cost of the minimum path of rank k from i to j

All-pairs shortest path

- Given a weighted (directed or undirected) graph $G = (V, E)$ wits $|V| = n$ and $|E| = m$, find the shortest path from each vertex to all other vertices
- Floyd's algorithm: Find shortest paths of rank k for increasing k
  - Uses the adjacency matrix $G_{1\ldots n,1\ldots n}$ of G
  - Path of rank k: path that only traverses vertices 1 to k (not counting the source and the destination)
  - Subproblems: $P_k = (A^k_{i,j}, \pi^k_{i,j})_{1 \le i \le n, 1 \le j \le n}$
    - $A^k_{i,j}$ is the cost of the minimum path of rank k from i to j
    - Recursive solution:

$$A^k_{i,j} = \begin{cases} G_{i,j} & \text{if } k = 0 \\ \min\{A^{k-1}_{i,j}, A^{k-1}_{i,k} + A^{k-1}_{k,j}\} & \text{otherwise} \end{cases}$$

All-pairs shortest path

- Given a weighted (directed or undirected) graph $G = (V, E)$ wits $|V| = n$ and $|E| = m$, find the shortest path from each vertex to all other vertices
- Floyd's algorithm: Find shortest paths of rank k for increasing k
  - Uses the adjacency matrix $G_{1\ldots n,1\ldots n}$ of G
  - Path of rank k: path that only traverses vertices 1 to k (not counting the source and the destination)
  - Subproblems: $P_k = (A^k_{i,j}, \pi^k_{i,j})_{1 \le i \le n, 1 \le j \le n}$
    - $A^k_{i,j}$ is the cost of the minimum path of rank k from i to j
    - $\pi^k_{i,j}$ is the predecessor of j in the minimum cost path of rank k from i to j
    - Recursive solution:

$$A^k_{i,j} = \begin{cases} G_{i,j} & \text{if } k = 0 \\ \min\{A^{k-1}_{i,j}, A^{k-1}_{i,k} + A^{k-1}_{k,j}\} & \text{otherwise} \end{cases}$$

$$\pi^k_{i,j} = \begin{cases} i & \text{if } k = 0 \\ \pi^{k-1}_{i,j} & \text{if } A^{k-1}_{i,j} \le A^{k-1}_{i,k} + A^{k-1}_{k,j} \\ k & \text{if } A^{k-1}_{i,j} > A^{k-1}_{i,k} + A^{k-1}_{k,j} \end{cases}$$

Floyd's algorithm (cont'd)

- Memoization: arrays A^k and π^k for cost and predecessor
- Evaluation order: increasing k, arbitrary for i and j

for $i = 1$ **to** n **do**
  for $j = 1$ **to** n **do**
    $A^0_{i,j} \leftarrow G_{ij}$
    $\pi_{i,j} \leftarrow i$

for $k = 1$ **to** n **do** $O(n^3)$
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
      if $A^{k-1}_{i,j} \leq A^{k-1}_{i,k} + A^{k-1}_{k,j}$ **then**
        $A^k_{i,j} \leftarrow A^{k-1}_{i,j}$
      else
        $A^k_{i,j} \leftarrow A^{k-1}_{i,k} + A^{k-1}_{k,j}$
        $\pi_{i,j} \leftarrow k$

  - Optimization:

Floyd's algorithm (cont'd)

- Memoization: arrays A^k and π^k for cost and predecessor
- Evaluation order: increasing k, arbitrary for i and j

for $i = 1$ **to** n **do**
  for $j = 1$ **to** n **do**
    $A^0_{i,j} \leftarrow G_{ij}$
    $\pi_{i,j} \leftarrow i$

for $k = 1$ **to** n **do** $O(n^3)$
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
      if $A^{k-1}_{i,j} \leq A^{k-1}_{i,k} + A^{k-1}_{k,j}$ **then**
        $A^k_{i,j} \leftarrow A^{k-1}_{i,j}$
      else
        $A^k_{i,j} \leftarrow A^{k-1}_{i,k} + A^{k-1}_{k,j}$
        $\pi_{i,j} \leftarrow k$

  - Optimization: A single predecessor array π
    - When computing $\pi^k_{i,j}$ we only need $\pi^{k-1}_{i,j}$ and then we can overwrite it

Floyd's algorithm (cont'd)

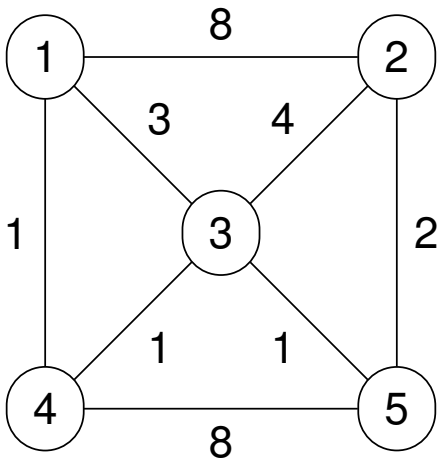
- Memoization: arrays A^k and π^k for cost and predecessor
- Evaluation order: increasing k, arbitrary for i and j

for $i = 1$ **to** n **do**
  for $j = 1$ **to** n **do**
    $A^0_{i,j} \leftarrow G_{ij}$
    $\pi_{i,j} \leftarrow i$

for $k = 1$ **to** n **do** $O(n^3)$
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
      if $A^{k-1}_{i,j} \leq A^{k-1}_{i,k} + A^{k-1}_{k,j}$ **then**
        $A^k_{i,j} \leftarrow A^{k-1}_{i,j}$
      else
        $A^k_{i,j} \leftarrow A^{k-1}_{i,k} + A^{k-1}_{k,j}$
        $\pi_{i,j} \leftarrow k$

  - Optimization: A single predecessor array π
    - When computing $\pi^k_{i,j}$ we only need $\pi^{k-1}_{i,j}$ and then we can overwrite it
  - Further optimization: At any step we only need A^{k-1} and A^k, so we only need two matrices for the cost (current and previous)

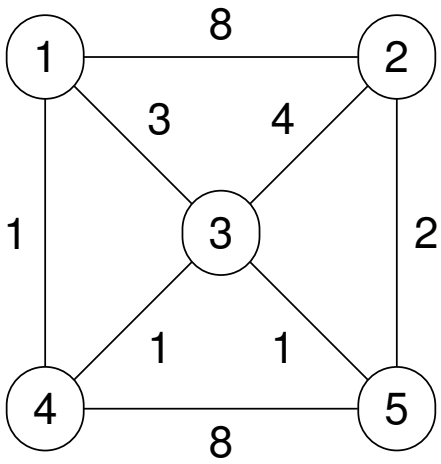
Floyd's algorithm example



$$A^0 = \begin{matrix} 0 & 8 & 3 & 1 & \infty \\ 8 & 0 & 4 & \infty & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & \infty & 1 & 0 & 8 \\ \infty & 2 & 1 & 8 & 0 \end{matrix} \qquad A^1 = \begin{matrix} 0 & 8 & 3 & 1 & \infty \\ 8 & 0 & 4 & 9 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 9 & 1 & 0 & 8 \\ \infty & 2 & 1 & 8 & 0 \end{matrix}$$

$$\pi^0 = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{matrix} \qquad \pi^1 = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 1 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 1 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{matrix}$$

Floyd's algorithm example

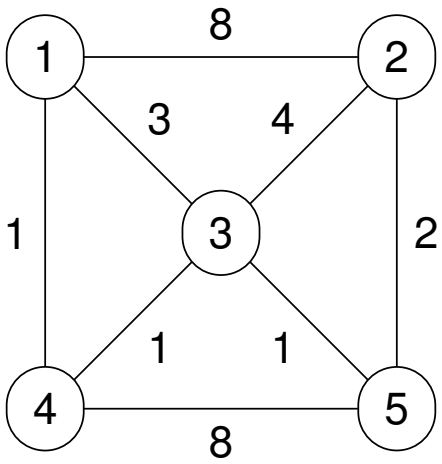


$$A^0 = \begin{matrix} 0 & 8 & 3 & 1 & \infty \\ 8 & 0 & 4 & \infty & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & \infty & 1 & 0 & 8 \\ \infty & 2 & 1 & 8 & 0 \end{matrix} \qquad \pi^0 = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{matrix}$$

$$A^1 = \begin{matrix} 0 & 8 & 3 & 1 & \infty \\ 8 & 0 & 4 & 9 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 9 & 1 & 0 & 8 \\ \infty & 2 & 1 & 8 & 0 \end{matrix} \qquad \pi^1 = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 1 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 1 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{matrix}$$

$$A^2 = \begin{matrix} 0 & 8 & 3 & 1 & 10 \\ 8 & 0 & 4 & 9 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 9 & 1 & 0 & 8 \\ 10 & 2 & 1 & 8 & 0 \end{matrix} \qquad \pi^2 = \begin{matrix} 1 & 1 & 1 & 1 & 2 \\ 2 & 2 & 2 & 1 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 1 & 4 & 4 & 4 \\ 2 & 5 & 5 & 5 & 5 \end{matrix}$$

Floyd's algorithm example



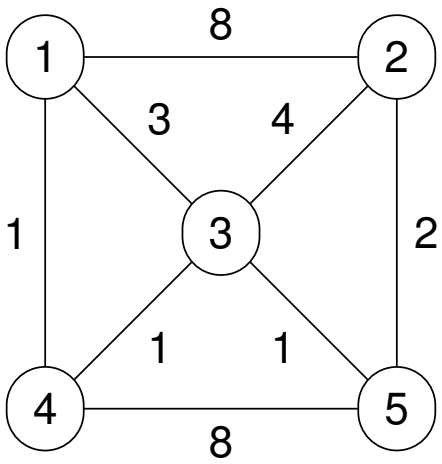
$$A^0 = \begin{matrix} 0 & 8 & 3 & 1 & \infty \\ 8 & 0 & 4 & \infty & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & \infty & 1 & 0 & 8 \\ \infty & 2 & 1 & 8 & 0 \end{matrix} \qquad \pi^0 = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{matrix}$$

$$A^1 = \begin{matrix} 0 & 8 & 3 & 1 & \infty \\ 8 & 0 & 4 & 9 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 9 & 1 & 0 & 8 \\ \infty & 2 & 1 & 8 & 0 \end{matrix} \qquad \pi^1 = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 1 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 1 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{matrix}$$

$$A^2 = \begin{matrix} 0 & 8 & 3 & 1 & 10 \\ 8 & 0 & 4 & 9 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 9 & 1 & 0 & 8 \\ 10 & 2 & 1 & 8 & 0 \end{matrix} \qquad \pi^2 = \begin{matrix} 1 & 1 & 1 & 1 & 2 \\ 2 & 2 & 2 & 1 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 1 & 4 & 4 & 4 \\ 2 & 5 & 5 & 5 & 5 \end{matrix}$$

$$A^3 = \begin{matrix} 0 & 7 & 3 & 1 & 4 \\ 7 & 0 & 4 & 5 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 5 & 1 & 0 & 2 \\ 4 & 2 & 1 & 2 & 0 \end{matrix} \qquad \pi^3 = \begin{matrix} 1 & 3 & 1 & 1 & 3 \\ 3 & 2 & 2 & 3 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 3 & 4 & 4 & 3 \\ 3 & 5 & 5 & 3 & 5 \end{matrix}$$

Floyd's algorithm example



$$A^0 = \begin{matrix} 0 & 8 & 3 & 1 & \infty \\ 8 & 0 & 4 & \infty & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & \infty & 1 & 0 & 8 \\ \infty & 2 & 1 & 8 & 0 \end{matrix} \qquad \pi^0 = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{matrix}$$

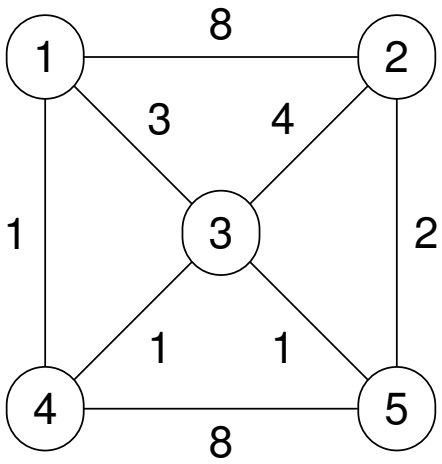
$$A^1 = \begin{matrix} 0 & 8 & 3 & 1 & \infty \\ 8 & 0 & 4 & 9 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 9 & 1 & 0 & 8 \\ \infty & 2 & 1 & 8 & 0 \end{matrix} \qquad \pi^1 = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 1 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 1 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{matrix}$$

$$A^2 = \begin{matrix} 0 & 8 & 3 & 1 & 10 \\ 8 & 0 & 4 & 9 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 9 & 1 & 0 & 8 \\ 10 & 2 & 1 & 8 & 0 \end{matrix} \qquad \pi^2 = \begin{matrix} 1 & 1 & 1 & 1 & 2 \\ 2 & 2 & 2 & 1 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 1 & 4 & 4 & 4 \\ 2 & 5 & 5 & 5 & 5 \end{matrix}$$

$$A^3 = \begin{matrix} 0 & 7 & 3 & 1 & 4 \\ 7 & 0 & 4 & 5 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 5 & 1 & 0 & 2 \\ 4 & 2 & 1 & 2 & 0 \end{matrix} \qquad \pi^3 = \begin{matrix} 1 & 3 & 1 & 1 & 3 \\ 3 & 2 & 2 & 3 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 3 & 4 & 4 & 3 \\ 3 & 5 & 5 & 3 & 5 \end{matrix}$$

$$A^4 = \begin{matrix} 0 & 6 & 2 & 1 & 4 \\ 6 & 0 & 4 & 5 & 2 \\ 2 & 4 & 0 & 1 & 1 \\ 1 & 5 & 1 & 0 & 2 \\ 4 & 2 & 1 & 2 & 0 \end{matrix} \qquad \pi^4 = \begin{matrix} 1 & 4 & 4 & 1 & 3 \\ 4 & 2 & 2 & 3 & 2 \\ 4 & 3 & 3 & 3 & 3 \\ 4 & 3 & 4 & 4 & 3 \\ 3 & 5 & 5 & 3 & 5 \end{matrix}$$

Floyd's algorithm example



$$A^0 = \begin{matrix} 0 & 8 & 3 & 1 & \infty \\ 8 & 0 & 4 & \infty & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & \infty & 1 & 0 & 8 \\ \infty & 2 & 1 & 8 & 0 \end{matrix} \qquad \pi^0 = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{matrix}$$

$$A^1 = \begin{matrix} 0 & 8 & 3 & 1 & \infty \\ 8 & 0 & 4 & 9 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 9 & 1 & 0 & 8 \\ \infty & 2 & 1 & 8 & 0 \end{matrix} \qquad \pi^1 = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 1 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 1 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{matrix}$$

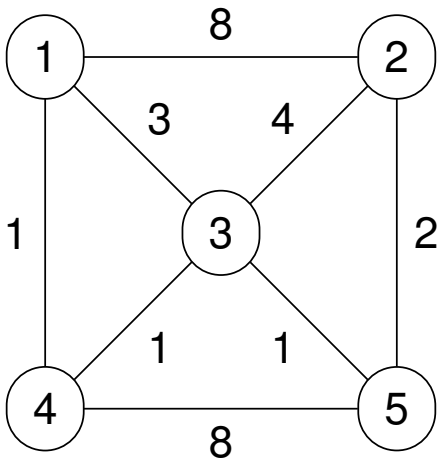
$$A^2 = \begin{matrix} 0 & 8 & 3 & 1 & 10 \\ 8 & 0 & 4 & 9 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 9 & 1 & 0 & 8 \\ 10 & 2 & 1 & 8 & 0 \end{matrix} \qquad \pi^2 = \begin{matrix} 1 & 1 & 1 & 1 & 2 \\ 2 & 2 & 2 & 1 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 1 & 4 & 4 & 4 \\ 2 & 5 & 5 & 5 & 5 \end{matrix}$$

$$A^3 = \begin{matrix} 0 & 7 & 3 & 1 & 4 \\ 7 & 0 & 4 & 5 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 5 & 1 & 0 & 2 \\ 4 & 2 & 1 & 2 & 0 \end{matrix} \qquad \pi^3 = \begin{matrix} 1 & 3 & 1 & 1 & 3 \\ 3 & 2 & 2 & 3 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 3 & 4 & 4 & 3 \\ 3 & 5 & 5 & 3 & 5 \end{matrix}$$

$$A^4 = \begin{matrix} 0 & 6 & 2 & 1 & 4 \\ 6 & 0 & 4 & 5 & 2 \\ 2 & 4 & 0 & 1 & 1 \\ 1 & 5 & 1 & 0 & 2 \\ 4 & 2 & 1 & 2 & 0 \end{matrix} \qquad \pi^4 = \begin{matrix} 1 & 4 & 4 & 1 & 3 \\ 4 & 2 & 2 & 3 & 2 \\ 4 & 3 & 3 & 3 & 3 \\ 4 & 3 & 4 & 4 & 3 \\ 3 & 5 & 5 & 3 & 5 \end{matrix}$$

$$A^5 = \begin{matrix} 0 & 5 & 2 & 1 & 4 \\ 5 & 0 & 3 & 4 & 2 \\ 2 & 3 & 0 & 1 & 1 \\ 1 & 4 & 1 & 0 & 2 \\ 4 & 2 & 1 & 2 & 0 \end{matrix} \qquad \pi^5 = \begin{matrix} 1 & 5 & 4 & 1 & 3 \\ 5 & 2 & 5 & 5 & 2 \\ 4 & 5 & 3 & 3 & 3 \\ 4 & 5 & 4 & 4 & 3 \\ 3 & 5 & 5 & 3 & 5 \end{matrix}$$

Floyd's algorithm example



$$A^0 = \begin{pmatrix} 0 & 8 & 3 & 1 & \infty \\ 8 & 0 & 4 & \infty & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & \infty & 1 & 0 & 8 \\ \infty & 2 & 1 & 8 & 0 \end{pmatrix} \quad \pi^0 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{pmatrix}$$

$$A^1 = \begin{pmatrix} 0 & 8 & 3 & 1 & \infty \\ 8 & 0 & 4 & 9 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 9 & 1 & 0 & 8 \\ \infty & 2 & 1 & 8 & 0 \end{pmatrix} \quad \pi^1 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 1 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 1 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{pmatrix}$$

$$A^2 = \begin{pmatrix} 0 & 8 & 3 & 1 & 10 \\ 8 & 0 & 4 & 9 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 9 & 1 & 0 & 8 \\ 10 & 2 & 1 & 8 & 0 \end{pmatrix} \quad \pi^2 = \begin{pmatrix} 1 & 1 & 1 & 1 & 2 \\ 2 & 2 & 2 & 1 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 1 & 4 & 4 & 4 \\ 2 & 5 & 5 & 5 & 5 \end{pmatrix}$$

$$A^3 = \begin{pmatrix} 0 & 7 & 3 & 1 & 4 \\ 7 & 0 & 4 & 5 & 2 \\ 3 & 4 & 0 & 1 & 1 \\ 1 & 5 & 1 & 0 & 2 \\ 4 & 2 & 1 & 2 & 0 \end{pmatrix} \quad \pi^3 = \begin{pmatrix} 1 & 3 & 1 & 1 & 3 \\ 3 & 2 & 2 & 3 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 3 & 4 & 4 & 3 \\ 3 & 5 & 5 & 3 & 5 \end{pmatrix}$$

$$A^4 = \begin{pmatrix} 0 & 6 & 2 & 1 & 4 \\ 6 & 0 & 4 & 5 & 2 \\ 2 & 4 & 0 & 1 & 1 \\ 1 & 5 & 1 & 0 & 2 \\ 4 & 2 & 1 & 2 & 0 \end{pmatrix} \quad \pi^4 = \begin{pmatrix} 1 & 4 & 4 & 1 & 3 \\ 4 & 2 & 2 & 3 & 2 \\ 4 & 3 & 3 & 3 & 3 \\ 4 & 3 & 4 & 4 & 3 \\ 3 & 5 & 5 & 3 & 5 \end{pmatrix}$$

$$A^5 = \begin{pmatrix} 0 & 5 & 2 & 1 & 4 \\ 5 & 0 & 3 & 4 & 2 \\ 2 & 3 & 0 & 1 & 1 \\ 1 & 4 & 1 & 0 & 2 \\ 4 & 2 & 1 & 2 & 0 \end{pmatrix} \quad \pi^5 = \begin{pmatrix} 1 & 5 & 4 & 1 & 3 \\ 5 & 2 & 5 & 5 & 2 \\ 4 & 5 & 3 & 3 & 3 \\ 4 & 5 & 4 & 4 & 3 \\ 3 & 5 & 5 & 3 & 5 \end{pmatrix}$$

Path $1 \to 2$: $\pi(2,1) = 5, \pi(5,1) = 3, \pi(3,1) = 4, \pi(4,1) = 1 \Rightarrow \langle 1, 4, 3, 5, 2 \rangle$

Path $1 \to 3$: $\pi(3,1) = 4, \pi(4,1) = 1 \Rightarrow \langle 1, 4, 3 \rangle$

Path $3 \to 1$: $\pi(1,3) = 4, \pi(4,3) = 3 \Rightarrow \langle 3, 4, 1 \rangle$ (undirected graph!)

The traveling salesman problem

- Given a weighted directed graph $G = (\{1, 2, \ldots, n\}, E)$ find the Hamiltonian Cycle (traversing all vertices exactly once) of minimum cost
  - Naïve solution: try all permutations, retain the minimal cost ($O(n2^n)$ time)

The traveling salesman problem

- Given a weighted directed graph $G = (\{1, 2, \ldots, n\}, E)$ find the Hamiltonian Cycle (traversing all vertices exactly once) of minimum cost
  - Naïve solution: try all permutations, retain the minimal cost ($O(n2^n)$ time)
- Crux:
  - Start the cycle at vertex 1, let the next vertex be k
  - The path from k to 1 must be an optimal (minimum cost) Hamiltonian path for the graph induced by $V \setminus \{1\}$
- Recursive solution:
  - Let $g(i, S)$ be the length of the shortest path starting at i and going through all the vertices in S back to 1

$$g(i, S) = \begin{cases} \min_{(i,j) \in E}(w(i, j)) & \text{if } S = \emptyset \\ \min_{j \in S}(w((i, j)) + g(j, S \setminus \{j\})) & \text{otherwise} \end{cases}$$

  - Memoization: Table $(g_{i,j})_{i \in \{1,\ldots,n\}, j \in 2^{\{1,\ldots,n\}}}$
  - Order of evaluation: increasing second dimension, do not care for the first
  - Running time: $O(n2^n)$
- When dynamic programming does not improve running time then it should not be chosen

The traveling salesman problem

- Given a weighted directed graph $G = (\{1, 2, \ldots, n\}, E)$ find the Hamiltonian Cycle (traversing all vertices exactly once) of minimum cost
  - Naïve solution: try all permutations, retain the minimal cost ($O(n2^n)$ time)
- Crux:
  - Start the cycle at vertex 1, let the next vertex be k
  - The path from k to 1 must be an optimal (minimum cost) Hamiltonian path for the graph induced by $V \setminus \{1\}$
- Recursive solution:
  - Let $g(i, S)$ be the length of the shortest path starting at i and going through all the vertices in S back to 1

$$g(i, S) = \begin{cases} \min_{(i,j) \in E}(w(i, j)) & \text{if } S = \emptyset \\ \min_{j \in S}(w((i, j)) + g(j, S \setminus \{j\})) & \text{otherwise} \end{cases}$$

  - Memoization: Table $(g_{i,j})_{i \in \{1,\ldots,n\}, j \in 2^{\{1,\ldots,n\}}}$
  - Order of evaluation: increasing second dimension, do not care for the first
  - Running time: $O(n2^n)$
- When dynamic programming does not improve running time then it should not be chosen
- Unknown (million dollar question, literally) whether we can do better than the naïve solution for the traveling salesman and the 0/1 knapsack (and many more problems)