Dynamic programming

- General problem-solving technique
- Typically applied to optimization problems.
- Solves problems by solving smaller subproblems using optimal substructure.
- Applicable in certain situations where there is a correct but inefficient recursive solution.
- Avoids repeated solution of redundant subproblems: each subproblem is only solved once. This is the fundamental difference between dynamic programming and divide-and-conquer.
- Requires indexing of subproblems.

NOTE: This is difficult material. Readings:

- [GT]: Chapter 12
- [Kleinberg and Tardos], Chapter 6
- [CLRS] Chapter 15

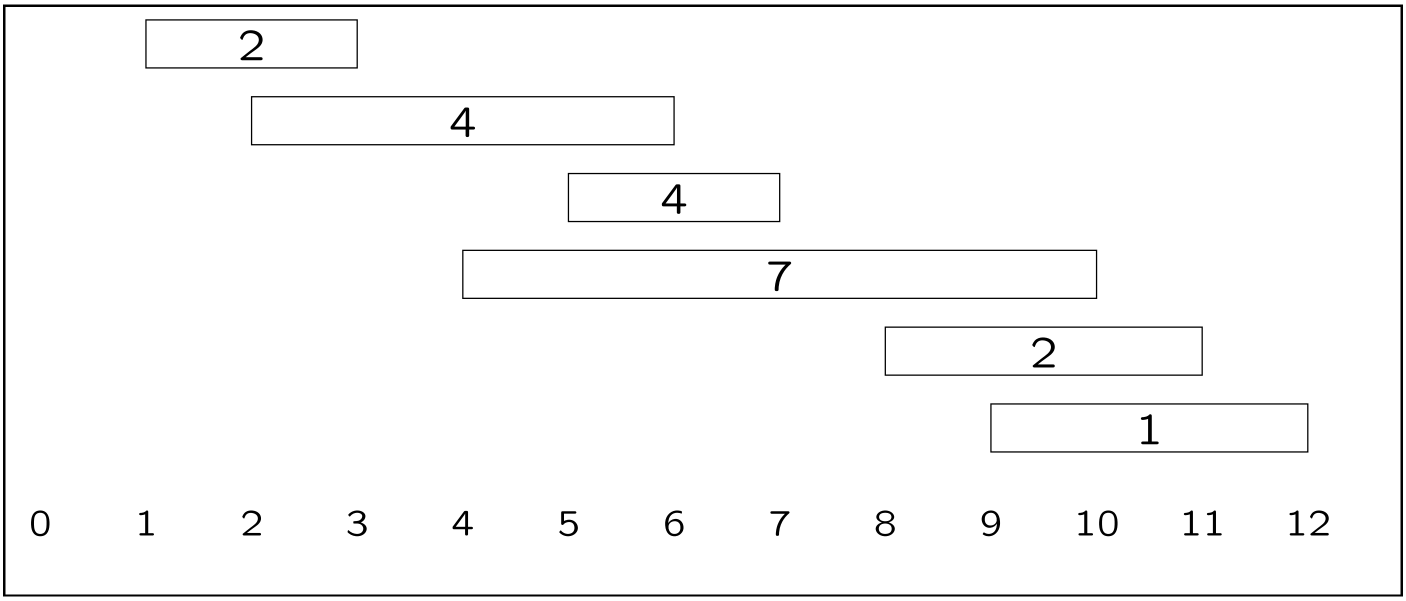
<u>Problem:</u> Job scheduling (Weighted interval scheduling)

- Input: Collection of n Jobs (intervals) represented by Start Time, Finish Time, and Value: $(s(i), f(i), v(i))$.

- Problem: Find a non-overlapping set of intervals that maximizes the total value.

- Example:

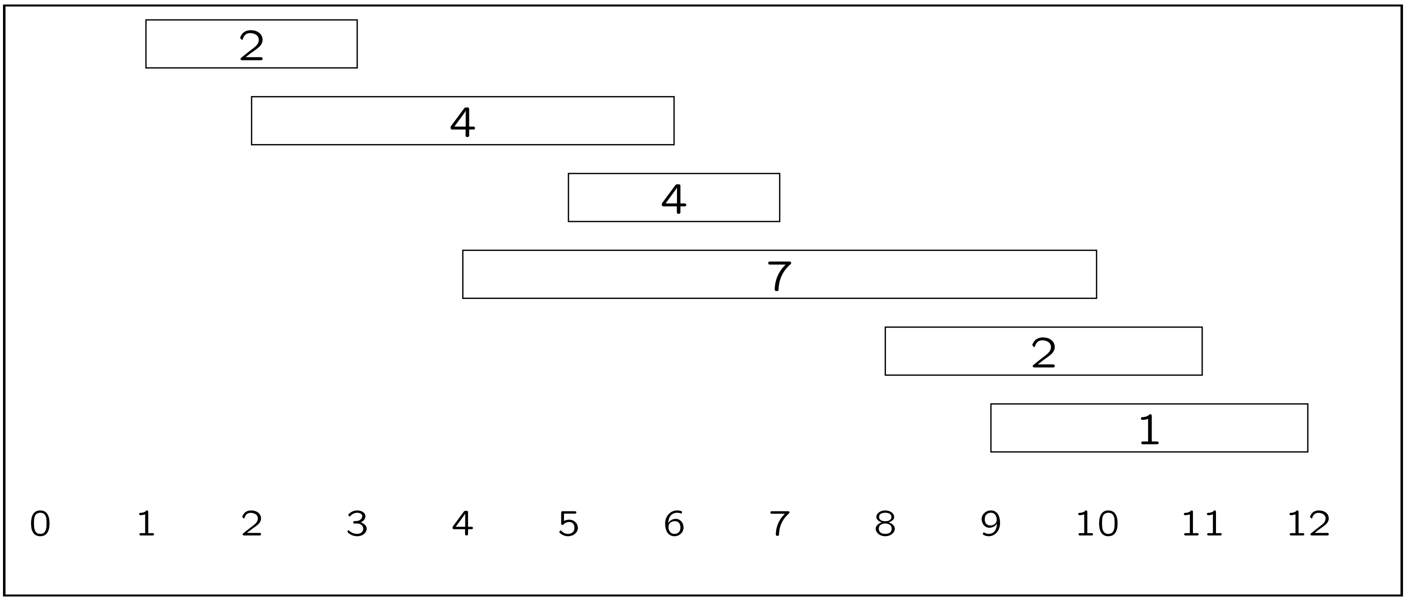
i	$s(i)$	$f(i)$	$v(i)$
1	1	3	2
2	2	6	4
3	5	7	4
4	4	10	7
5	8	11	2
6	9	12	1



Simple recursive algorithm

- Assume intervals are sorted by finishing time
- For each i, let $p(i)$ be the highest-numbered interval to the left of interval i that doesn't overlap it. (See next slide)
- For each i, let OPT(i) be value of the best solution.
- "Either the optimal solution contains the last interval or it doesn't"
  - If it does: optimal value is $v(n)$ plus the value of the optimal collection from $1, \ldots, p(n)$
  - If it doesn't: optimal value is the value of the optimal collection from $1, \ldots, n-1$
- The same principle holds for all j. So:

$$\mathrm{OPT}(j) = \max\left(v(j) + \mathrm{OPT}(p(j)), \mathrm{OPT}(j-1)\right)$$



i	$p(i)$
1	0
2	0
3	1
4	1
5	3
6	3

Pseudocode for simple recursive algorithm

```
int OPT(j)
   begin //OPT
      if j = 0 then return(0);
      else return max(v(j)+OPT(p(j)), OPT(j-1));
   end //OPT
```

- Preceding algorithm is correct, but very inefficient
- Source of inefficiency: Same value of OPT() recomputed multiple times.

Memoizing the recursion: Compute each value only once

- Declare an array $M[1..n]$

- Each entry can contain an integer or "undefined"

- Initialize all entries to "undefined"

```
int Mem_OPT(j)
  begin //Mem_OPT
    if j = 0 then return(0);
    else
      if M[j] = "undefined" then
        M[j] = max(v(j)+Mem_OPT(p(j)), Mem_OPT(j-1));
      return (M[j]);
  end //Mem_OPT
```

- Analysis
  - For every pair of recursive calls, an entry of M gets filled in.
  - Hence, $O(n)$ calls.
- So we have an efficient algorithm.
- But it still has a flaw:
  - Memoized algorithm computes the cost of an optimal interval set, but not the intervals themselves.
  - How can we fix this?

Computing the Optimal Set Of Intervals

Once we have computed the array M:

```
OutputSolution(j)
begin //OutputSolution
   if j = 0 return;
   if v[j] + M[p(j)] >= M[j-1] then
      output(j);
      OutputSolution(p(j));
   else
      OutputSolution(j-1);
end //OutputSolution
```

Bottom-up (Iterative) Solution to Weighted Interval Scheduling Problem

```
IterativeComputeOPT
begin // IterativeComputeOPT
   M[0] = 0;
   for j = 1 to n do
      M[j] = max(v(j)+M[p(j)],M[j-1]);
end // IterativeComputeOPT
```

Recommended Exercise:

1. Trace through this code on example, compute $M[i]$ for each i
2. Trace through this code on previous slide on example, compute intervals in optimum set

Principles of Dynamic Programming

- Can be applied when there is a set of subproblems derived from the original subproblem such that:
  - There are only a polynomial number of subproblems
  - The solution to the original problem can be easily computed from the solution to the subproblems.
    * For example, when the original problem *is* one of the subproblems. . .
  - There is a natural “ordering” on the subproblems (from smallest to largest).
  - There is an easily computed recurrence that can be used to compute the solution to a subproblem from some collection of smaller subproblems.

Truck loading problem

- Truck has weight limit of W.
- We have n boxes: box i has weight w_i.
- We want to carry the maximum weight possible, subject to the weight restriction.
- (Also known as subset-sum problem, 0/1 bin packing problem, etc.)
- Note: greedy heuristics don't give optimum solutions:
  - Largest box first: fails on $(W + 1)/2$, $W/2$, $W/2$.
  - Smallest box first: fails on 1, $W/2$, $W/2$.

Solving the Truck loading problem

- We can get smaller problems by making the maximum capacity and the number of boxes smaller.

- Let $M(i, r)$ be the value of the best way to load the first i boxes using maximum capacity r.

- If we optimally load i boxes using maximum capacity r either we include box i or we don't.

  If we include box i: $w_i + M(i-1, r-w_i)$

  If we do not include box i: $M(i-1, r)$.

  So,

  $M(i, r) = \max\left(w_i + M(i-1, r-w_i), M(i-1, r)\right)$

- Note that if $w_i > r$, we can't use box i, so only the second choice is available.

- Care required with boundary cases. What are $M(i, 0)$, $M(0, j)$?

Bottom-up solution to Truck-Loading Problem

```
OptTruckLoad()
begin //OptTruckLoad
  for i = 1 to n
    M[i][0] = 0;
  for j = 1 to W
    M[0][j] = 0;
  for i = 1 to n
    for r = 1 to W
      if (w[i] > r)
        M[i][r] = M[i-1][r])
      else
        M[i][r] = max(w[i]+M[i-1][r-w[i]], M[i-1][r]);
end //OptTruckLoad
```

Analysis

- Running time: $O(n \cdot W)$.
- Space requirement: $O(n \cdot W)$.

Computing the Optimal Set of Boxes

Once we have computed the array M, call `OutputSolution(n,W)`:

```
OutputSolution(i,r)
begin //OutputSolution
   if i = 0 return;
   if (w[i] <= r) and
     (w[i] + M[i-1][r-w[i]] >= M[i-1][r]) then
      output(i);
      OutputSolution(i-1,r-w[i]);
   else
      OutputSolution(i-1,r);
end //OutputSolution
```

0/1 Knapsack Problem

- Thief has a knapsack with limited capacity, and has to decide what items to steal.
- The are n items: item i has weight w_i, value v_i.
- Knapsack can handle a total weight of at most W.
- Thief wants to steal items with maximum total value, subject to the weight restriction.
- Thief cannot take a "fractional item." For each item, the thief either takes all of it or none of it.

Note on 0/1 Knapsack Problem:

- In "Fractional Knapsack Problem" where fractional items can be taken, greedy heuristic works: order items according to value per unit weight.

- This does not work for 0/1 Knapsack Problem, because we can only take whole items.

  Example: $W = 100$
  $w_1 = 20$, $v_i = 80$
  $w_2 = 90$, $v_2 = 90$.

Solving the 0/1 Knapsack Problem

- Very similar to truck loading problem.
- Let $M(i, r)$ be the value of the best way to load the first i items, using a knapsack with maximum capacity r.
- If we optimally load i items using maximum capacity r either we include item i or we don't. So:

  $M(i, r) = \max(v_i + M(i-1, r-w_i), M(i-1, r));$
- Note that if $w_i > r$, we can't use item i, so only the second choice is available.
- Leads to solution that runs in $O(n \cdot W)$ time.

Dynamic programming

- General problem-solving technique
- Typically applied to optimization problems.
- Solves problems by solving smaller subproblems using optimal substructure.
- Applicable in certain situations where there is a correct but inefficient recursive solution.
- Avoids repeated solution of redundant subproblems: each subproblem is only solved once. This is the fundamental difference between dynamic programming and divide-and-conquer.
- Requires indexing of subproblems.

NOTE: This is difficult material. Readings:

- [GT]: Chapter 12
- [Kleinberg and Tardos], Chapter 6
- [CLRS] Chapter 15

General approach to developing a dynamic programming algorithm: 4 steps (from [CLRS])

1. Characterize the structure of an optimal solution:
   - Goal
   - Base cases
   - Strategy for computing an optimal solution to a problem from optimal solutions to smaller problems
2. Recursively define the value of an optimal solution
3. Develop an algorithm to compute the value of an optimal solution in bottom-up fashion
4. Modify the algorithm to construct an optimal solution from computed information.

Optimal matrix chain multiplication

Some facts about matrix multiplication:

1. Multiplying a $p \times q$ matrix by a $q \times r$ matrix requires $p \cdot q \cdot r$ multiplications. (Because the product will be $p \times r$, and the computation of each entry requires q scalar multiplications).
2. Matrix multiplication is associative:
$$(A \times B) \times C = A \times (B \times C)$$
3. The multiplication order may effect the efficiency.

A:	$p \times q$	$A \times B$:	$p \times r$
B:	$q \times r$	$B \times C$:	$q \times s$
C:	$r \times s$		

$(A \times B) \times C$: Number of scalar multiplications is:

$$p \cdot q \cdot r + p \cdot r \cdot s$$

$A \times (B \times C)$: Number of scalar multiplications is:

$$q \cdot r \cdot s + p \cdot q \cdot s$$

For example, suppose A is 40×2, B is 2×100, and C is 100×50. Then

$(A \times B) \times C$: Cost is

$$40 \cdot 2 \cdot 100 + 40 \cdot 100 \cdot 50 = 8,000 + 200,000 = 208,000$$

$A \times (B \times C)$: Cost is

$$2 \cdot 100 \cdot 50 + 40 \cdot 2 \cdot 50 = 10,000 + 4,000 = 14,000$$

So $A \times (B \times C)$ is the more efficient grouping

General problem:

Given: n matrices: $A_1, \ldots, A_n$.
Matrix A_i is $d_{i-1} \times d_i$.

What is the most efficient way of grouping (i.e.,parenthesizing) to compute $A_1 \times \cdots \times A_n$?

"Most efficient" means "fewest scalar multiplications"

Example:

A_1 : 10×15	d_0 $=10$
A_2 : 15×5	d_1 $=15$
A_3 : 5×60	d_2 $=5$
A_4 : 60×100	d_3 $=60$
A_5 : 100×20	d_4 $=100$
A_6 : 20×40	d_5 $=20$
A_7 : 40×47	d_6 $=40$
	d_7 $=47$

As we will see, for this set of data, the optimal grouping is:

$$(A_1 \times A_2) \times ((((A_3 \times A_4) \times A_5) \times A_6) \times A_7)$$

Total cost of multiplying with this grouping: 56,500 scalar multiplications

(Step 1: Characterize optimal substructure)

Define

$M(i, j) =$ the number of multiplications required to compute the product $A_i \times \cdots \times A_j$ using the best possible grouping

Goal: $M(1, n)$

Base cases: $M(i, i) = 0$ for all i

We need to develop a strategy for computing an optimal grouping for multiplying the chain $A_i \times \cdots \times A_j$ from optimal groupings for smaller chains. . .

To compute $A_i \times \cdots \times A_j$:

- Choose some k with $i \leq k < j$
- Compute using the top-level grouping $(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$, computing both subchains optimally. This requires three steps:
    1. Compute the subchain $A_i \times \cdots \times A_k$. The cost is $M(i,k)$. The resulting matrix is $d_{i-1} \times d_k$.
    2. Compute the subchain $A_{k+1} \times \cdots \times A_j$. The cost is $M(k+1,j)$. The resulting matrix is $d_k \times d_j$.
    3. Perform the final multiply. The cost is $d_{i-1}d_kd_j$, because we are multiplying a $d_{i-1} \times d_k$ matrix by a $d_k \times d_j$ matrix.

So for a particular choice of k, the total cost is:

$$M(i,k) + M(k+1,j) + d_{i-1}d_kd_j$$

The optimal strategy for computing $A_i \times \cdots \times A_j$ requires determining the best k. Hence

$$M(i,j) = \min_{i \leq k \leq j-1} \left(M(i,k) + M(k+1,j) + d_{i-1}d_kd_j\right)$$

Illustration: Consider our example

A_1 : 10×15
A_2 : 15×5
A_3 : 5×60
A_4 : 60×100
A_5 : 100×20
A_6 : 20×40
A_7 : 40×47

$d_0 = 10$
$d_1 = 15$
$d_2 = 5$
$d_3 = 60$
$d_4 = 100$
$d_5 = 20$
$d_6 = 40$
$d_7 = 47$

Consider the computation of $M(3,6)$, the cost of the best strategy for the chain $A_3 \times A_4 \times A_5 \times A_6$. Suppose we have already computed the following values:

$$M[3,3] = 0 \qquad M[4,6] = 168000$$
$$M[3,4] = 30000 \qquad M[5,6] = 80000$$
$$M[3,5] = 40000 \qquad M[6,6] = 0$$

There are 3 possible choices for k:

k	Grouping	Cost
3	$(A_3) \times (A_4 \times A_5 \times A_6)$	$0 + 168000 + 5 \cdot 60 \cdot 40 = 180000$
4	$(A_3 \times A_4) \times (A_5 \times A_6)$	$30000 + 80000 + 5 \cdot 100 \cdot 40 = 130000$
5	$(A_3 \times A_4 \times A_5) \times (A_6)$	$40000 + 0 + 5 \cdot 20 \cdot 40 = 44000$

So the best choice is $k = 5$, the best grouping is $(A_3 \times A_4 \times A_5) \times (A_6)$, and $M(3,6) = 44000$.

(<u>Step 2</u>: Develop recursive solution)

As we have just seen:

$$M(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k \le j-1} \left(M(i,k) + M(k+1,j) + d_{i-1} d_k d_j \right) & \text{if } i < j \end{cases}$$

So the following recursive solution would work (top-level call: `M(1,n)`)

```
function M(i,j)
   begin { M }
      if (i = j) then return(0);
      min = +∞;
      for k = i to j-1 do
         x = M(i,k) + M(k+1,j) + d[i-1] * d[k] * d[j];
         if x < min then min = x;
      end { for };
      return(min);
   end { M }
```

But this does much redundant work. For example

`M[1,n]` requires `M[2,n]`, `M[3,n]`, `M[4,n]`, `M[5,n]`, . . .
  `M[2,n]` requires `M[3,n]`, `M[4,n]`, `M[5,n]`, . . .
    `M[3,n]` requires `M[4,n]`, `M[5,n]`, . . .
      `M[4,n]` requires `M[5,n]`,. . .

In fact, the work done by the above program is $\Omega(2^n)$.
(See [CLRS])

(Step 3: Compute optimal costs efficiently)

Observations:

- There are only a relatively small number of values of $M(i,j)$ (In fact, there are exactly $\binom{n}{2}$ of them.)
- We can store the values of M in a table, and compute each value exactly once.
- Order for filling the table: increasing order of chain length

```
procedure MatrixChainCost(d,n)
begin { MatrixChainCost }
   for i := 1 to n do
      M[i,i] = 0;
   end { for };
   for len = 2 to n do
      for i := 1 to n - len + 1 do
         j = i + len - 1;
         M[i,j] = +∞;
         for k = i to j - 1 do
            x = M[i,k] + M[k+1,j] + d[i-1] * d[k] * d[j];
            if x < M[i,j] then
               M[i,j] = x;
            endif
         end { for };
      end { for };
   end { for };
   return(M);
end { MatrixChainCost }
```

Work: $O(n^3)$ Space: $O(n^2)$

(Step 4: Compute optimal solution)

Previous solution computed the cost of the optimal grouping, but it did not compute the actual optimal grouping.

To compute the optimal grouping, we compute a second table, `S[i,j]`. The value `S[i,j]` tells us the value of k such that the optimal top-level grouping for computing $A_i \times \cdots \times A_j$ is

$$(A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$$

```
procedure MatrixChainOrder(d,n)
begin { MatrixChainOrder }
  for i := 1 to n do
    M[i,i] = 0;
  end { for };
  for len = 2 to n do
    for i := 1 to n - len + 1 do
      j = i + len - 1;
      M[i,j] = +∞;
      for k = i to j - 1 do
        x = M[i,k] + M[k+1,j] + d[i-1] * d[k] * d[j];
        if x < M[i,j] then
          M[i,j] = x;
          S[i,j] = k; ⇐
        endif
      end { for };
    end { for };
  end { for };
  return(M,S); ⇐
end { MatrixChainOrder }
```

Example:

A_1 : 10×15
A_2 : 15×5
A_3 : 5×60
A_4 : 60×100
A_5 : 100×20
A_6 : 20×40
A_7 : 40×47

$d_0 = 10$
$d_1 = 15$
$d_2 = 5$
$d_3 = 60$
$d_4 = 100$
$d_5 = 20$
$d_6 = 40$
$d_7 = 47$

j

	1	2	3	4	5	6	7	i
	0 —	750 1	3750 2	35750 2	41750 2	46750 2	56500 2	1
		0 —	4500 2	37500 2	41500 2	47000 2	56925 2	2
			0 —	30000 3	40000 4	44000 5	53400 6	3
				0 —	120000 4	168000 5	214000 5	4
					0 —	80000 5	131600 5	5
						0 —	37600 6	6
							0 —	7

Optimal grouping is:

$$(A_1 \times A_2) \times ((((A_3 \times A_4) \times A_5) \times A_6) \times A_7)$$

Cost of optimal grouping: 56,500 scalar multiplications

Optimal binary search trees

Given: A set of values to be stored as keys in a binary search tree, and the frequency of access of each value.

Problem: Compute a binary search tree that minimizes the weighted lookup cost.

Weighted lookup cost in a binary tree with n nodes is:

$$\sum_{i=1}^{n} p_i c_i,$$

where

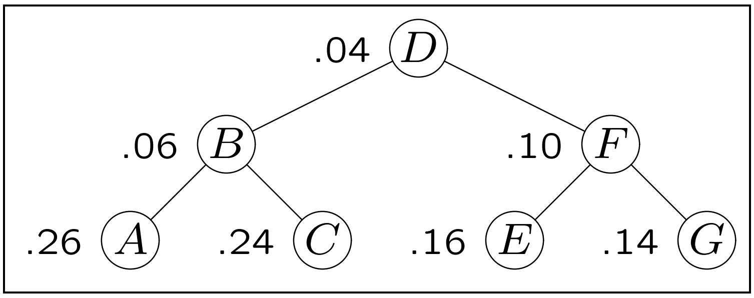
p_i = probability (frequency) of accessing node i

c_i = cost of accessing node i
$= 1 + \text{depth}(\text{node } i)$

Example: Suppose we have the following data values and frequency values:

i	Data	p_i
1	A	.26
2	B	.06
3	C	.24
4	D	.04
5	E	.16
6	F	.10
7	G	.14

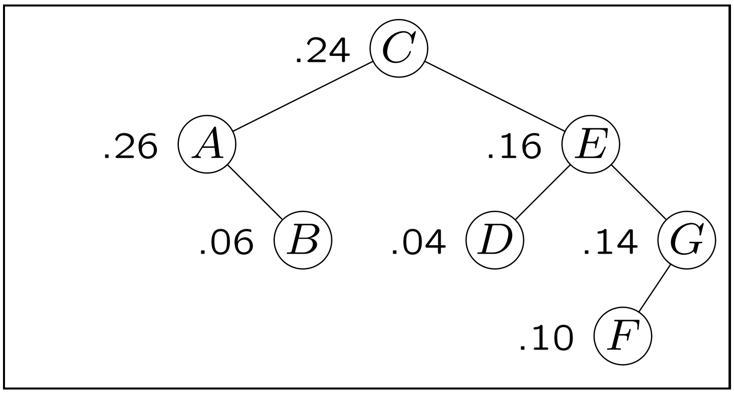
One possible binary search tree:



Weighted lookup cost is 2.76, because ...

i	Node	p_i	c_i	p_ic_i
1	A	.26	3	.78
2	B	.06	2	.12
3	C	.24	3	.72
4	D	.04	1	.04
5	E	.16	3	.48
6	F	.10	2	.20
7	G	.14	3	.42
				2.76

A better binary tree with same keys, same frequency values:



Weighted lookup cost is 2.20:

i	Node	p_i	c_i	$p_i c_i$
1	A	.26	2	.52
2	B	.06	3	.18
3	C	.24	1	.24
4	D	.04	3	.12
5	E	.16	2	.32
6	F	.10	4	.40
7	G	.14	3	.42
				2.20

General problem: Given a set of data values and a set of frequency values, construct a binary search tree of smallest weighted lookup cost.

Let $K_1, \ldots, K_n$ be the keys (in sorted order).
$p_1, \ldots . p_n$ be the corresponding frequency values

Note: We are assuming all searches are successful (i.e., every search request is for one of the n keys $K_1, \ldots, K_n$.) The generalization to allowing unsuccessful searches is discussed in [CLRS].

(Step 1: Characterize optimal substructure)

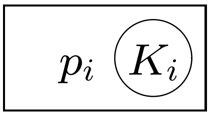
Finding a binary search tree with lowest weighted lookup cost on a given set of keys:

Let $E(i, j) =$ the weighted lookup cost of the binary search tree with lowest weighted lookup cost on the keys $K_i, \ldots, K_j$.

<u>Goal:</u> $E(1, n)$

<u>Base cases:</u>

1. For any i, $E(i, i) = p_i$.

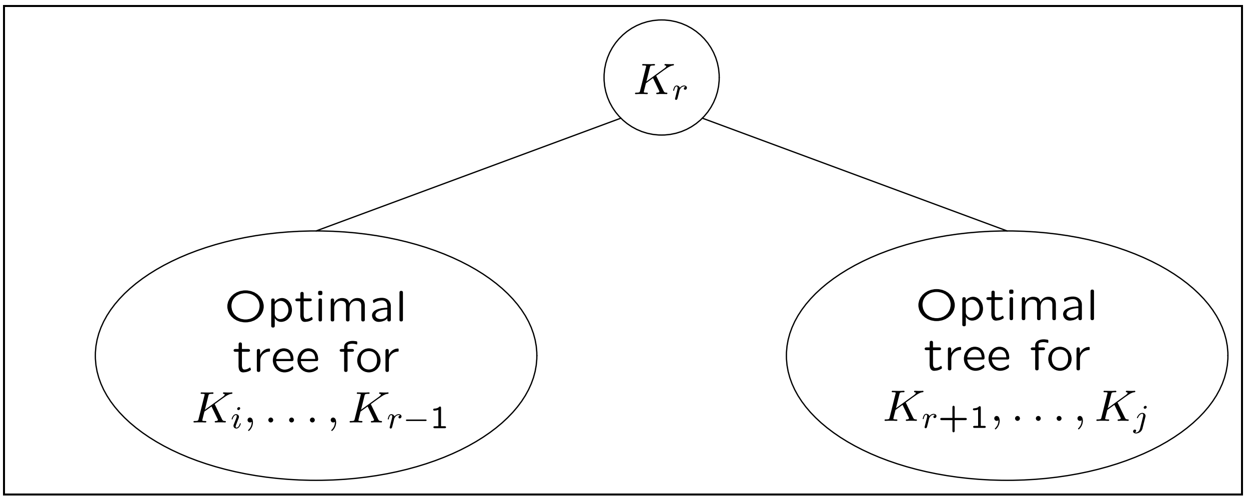


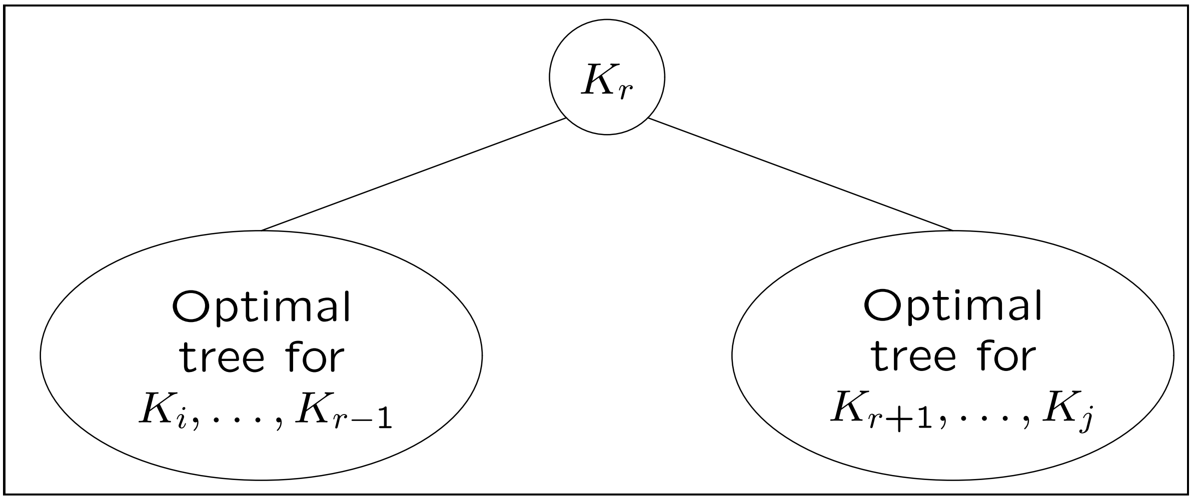
2. For any i, $E(i, i-1) = 0$. (Empty tree)

We need to develop a strategy for constructing the optimal binary on the set of keys $K_i, \ldots, K_n$ from the optimal binary search trees on smaller set of keys.

To build the optimal binary tree on the set of keys $K_i, \ldots, K_j$:

- Choose some r with $i \leq r \leq j$, and make K_r the root.
- The left subtree will be the optimal binary tree on the keys $K_i, \ldots, K_{r-1}$. Note that if $r = i$, this is an empty tree.
- The right subtree will be the optimal binary tree on the keys $K_{r+1}, \ldots, K_j$. Note that if $r = j$, this is an empty tree.





The cost of the tree can be computed as follows:

- The weighted cost of the optimal tree on $K_i, \ldots, K_{r-1}$ is $E(i, r-1)$. When we make this tree a subtree of the tree rooted at K_r, we push each node in the subtree down one level, increasing the cost of each node by 1. So the total weighted cost of the nodes $K_i, \ldots, K_{r-1}$ is

$$E(i, r-1) + p_i + p_{i+1} + \ldots + p_{r-1}.$$

- Similarly, the total weighted cost of the nodes $K_{r+1}, \ldots, K_j$ is

$$E(r+1, j) + p_{r+1} + \ldots + p_j.$$

- The weighted cost of the root node is $1 \cdot p_r = p_r$.

Hence the weighted cost of the tree is:

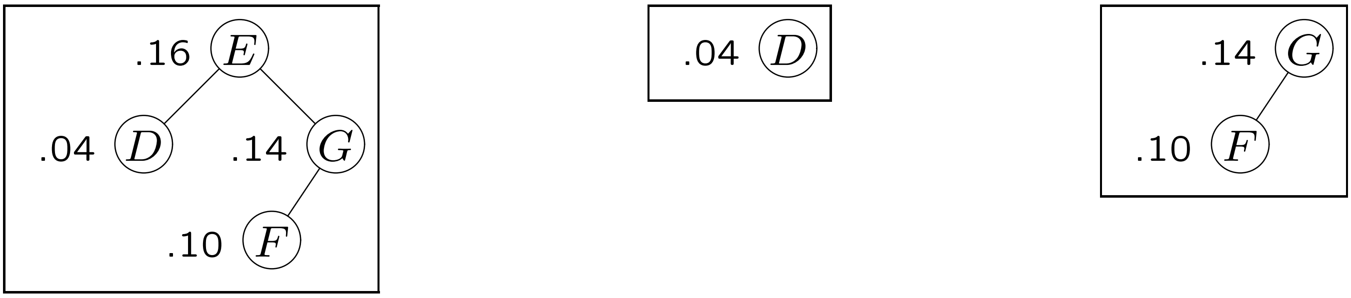
$$\begin{aligned} & E(i, r-1) + E(r+1, j) + p_i + p_{i+1} + \ldots + p_j \\ = \quad & E(i, r-1) + E(r+1, j) + W(i, j), \end{aligned}$$

where

$$W(i, j) = p_i + p_{i+1} + \ldots + p_j$$

is the sum of the frequencies of the keys $K_i, \ldots, K_j$.

Illustration



Left subtree has cost (.04)(1)

Right subtree has cost $(.14)(1) + (.10)(2)$

Entire tree has cost

$$(.04)(2) + (.14)(2) + (.10)(3) + (.16)(1),$$

which can be rewritten as:

$$\begin{pmatrix}\text{cost of}\\ \text{left}\\ \text{subtree}\end{pmatrix} + \begin{pmatrix}\text{cost of}\\ \text{right}\\ \text{subtree}\end{pmatrix} + (.04 + .14 + .10 + .16)\,,$$

or

$$\begin{pmatrix}\text{cost of}\\ \text{left}\\ \text{subtree}\end{pmatrix} + \begin{pmatrix}\text{cost of}\\ \text{right}\\ \text{subtree}\end{pmatrix} + \begin{pmatrix}\text{sum of}\\ \text{frequencies}\end{pmatrix}$$

As we have just seen, for a particular choice of the root note K_r, the weighted lookup cost for the tree on the keys $K_i, \ldots, K_j$ is

$$E(i, r-1) + E(r+1, j) + W(i, j).$$

The optimal weighted tree for $K_i, \ldots, K_j$ requires determining the best key K_r to use as the root. Hence

$$E(i, j) = \min_{i \leq r \leq j} \left(E(i, r-1) + E(r+1, j) + W(i, j) \right).$$

Illustration: Consider our example

i	Data	p_i
1	A	.26
2	B	.06
3	C	.24
4	D	.04
5	E	.16
6	F	.10
7	G	.14

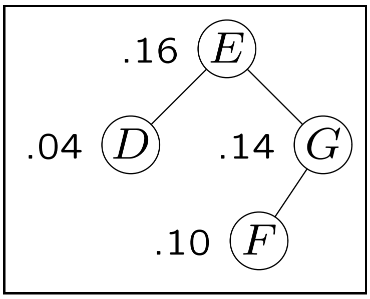
Consider the computation of $E(4,7)$, the cost of the best binary tree for the keys D,E,F,G. Suppose we have already computed the following values:

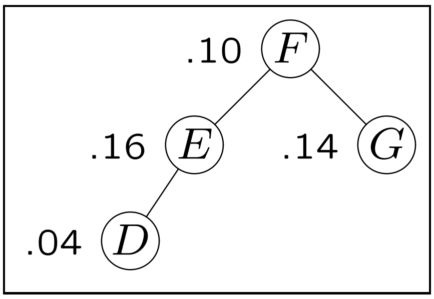
$$\begin{array}{lcl lcl lcl} E[4,3] &=& 0 & E[5,7] &=& 0.70 & W[4,7] &=& 0.44 \\ E[4,4] &=& 0.04 & E[6,7] &=& 0.34 \\ E[4,5] &=& 0.24 & E[7,7] &=& 0.14 \\ E[4,6] &=& 0.44 & E[8,7] &=& 0 \end{array}$$

There are 4 possible choices for r:

r	Cost
4	$0 + 0.70 + 0.44 = 1.14$
5	$0.04 + 0.34 + 0.44 = 0.82$
6	$0.24 + 0.14 + 0.44 = 0.82$
7	$0.44 + 0.00 + 0.44 = 0.88$

So the best choice is $r = 5$ or $r = 6$, $E(4,7) = 0.82$ and the best tree(s) are:





(Step 2: Develop recursive solution)

As we have just seen:

$$E(i,j) = \begin{cases} 0 & \text{if } j < i \\ p_i & \text{if } j = i \\ \text{min}_{i \le r \le j} \left(E(i, r-1) + E(r+1, j) + W(i,j) \right) & \text{if } j > i \end{cases}$$

Just as in the case of matrix chain multiplication, this can be used to derive a recursive solution. (Exercise: do this!!) But the resulting solution is not very efficient.

(Step 3: Compute optimal costs efficiently)

Observations:

- There are only a relatively small number of values of $E(i,j)$ (In fact, there are only $O(n^2)$ of them.)
- We can store the values of E in a table, and compute each value exactly once.
- Order for filling the table: increasing order of tree size

```
procedure OptimalTreeCost(d,n)
begin { OptimalTreeCost }
  for i := 1 to n do
    E[i,i-1] = 0;
    W[i,i-1] = 0;
  end { for };
  for size = 1 to n do
    for i := 1 to n - size + 1 do
      j = i + size - 1;
      E[i,j] = +∞;
      W[i,j] = W[i,j-1] + p[j];
      for r = i to j do
        x = E[i,r-1] + E[r+1,j] + W[i,j];
        if x < E[i,j] then
          E[i,j] = x;
        endif
      end { for };
    end { for };
  end { for };
  return(E);
end { OptimalTreeCost }
```

Work: $O(n^3)$ Space: $O(n^2)$

(Step 4: Compute optimal solution)

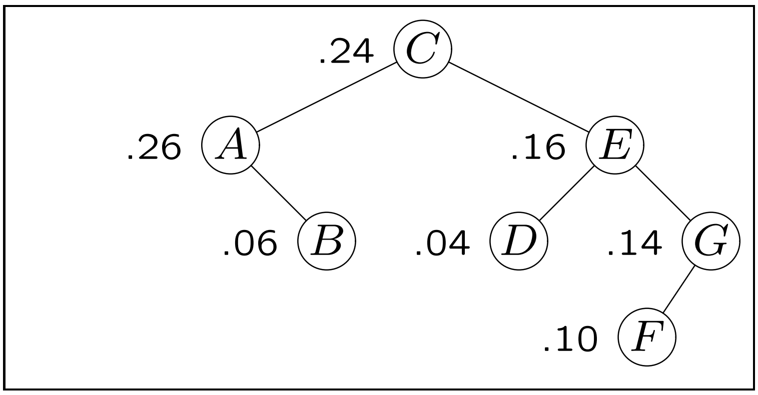
To compute the optimal tree (in addition to its weighted lookup cost), we compute a second table, `root[i,j]`. The value `root[i,j]` tells us the value of r that is the optimal root of the tree consisting of keys $i, \ldots, j$.

```
procedure OptimalTree(d,n)
begin { OptimalTree }
  for i := 1 to n do
    E[i,i-1] = 0;
    W[i,i-1] = 0;
  end { for };
  for size = 1 to n do
    for i := 1 to n - size + 1 do
      j = i + size - 1;
      E[i,j] = +∞;
      W[i,j] = W[i,j-1] + p[j];
      for r = i to j do
        x = E[i,r-1] + E[r+1,j] + W[i,j];
        if x < E[i,j] then
          E[i,j] = x;
          root[i,j] = r; ⇐
        endif
      end { for };
    end { for };
  end { for };
  return(E, root); ⇐
end { OptimalTree }
```

Example:

i	Data	p_i
1	A	.26
2	B	.06
3	C	.24
4	D	.04
5	E	.16
6	F	.10
7	G	.14

i \ j	0	1	2	3	4	5	6	7
1	0 —	0.26 1	0.38 1	0.92 1	1.02 3	1.38 3	1.68 3	2.20 3
2		0 —	0.06 2	0.36 3	0.44 3	0.80 3	1.10 3	1.52 5
3			0 —	0.24 3	0.32 3	0.68 3	0.96 5	1.34 5
4				0 —	0.04 4	0.24 5	0.44 5	0.82 5
5					0 —	0.16 5	0.36 5	0.70 6
6						0 —	0.10 6	0.34 7
7							0 —	0.14 7
8								0 —



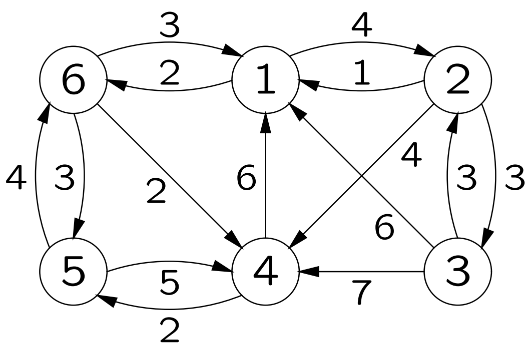
Cost is 2.20.

All-pairs shortest-path problem (Floyd's algorithm)

Given: A weighted graph or digraph G

Output: For every pair of vertices v and w, the shortest path from v to w.

Example



$D_{i,j}$	1	2	3	4	5	6
1	0	4	7	4	5	2
2	1	0	3	4	6	3
3	4	3	0	7	9	6
4	6	10	13	0	2	6
5	7	11	14	5	0	4
6	3	7	10	2	3	0

$D_{i,j}$ = length of shortest path from i to j

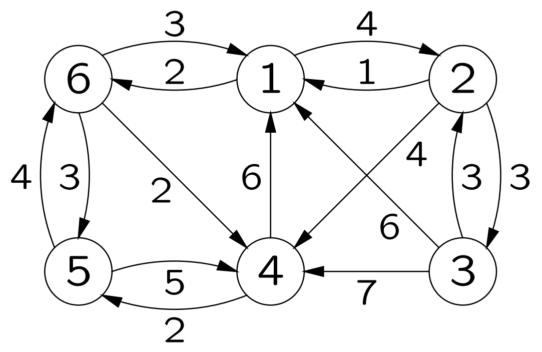
We could solve this problem by running Dijkstra's algorithm n times.

Floyd's algorithm solves the problem in $O(n^3)$ time, with $O(1)$ additional space.

<u>Graph representation:</u> The graph is represented as an adjacency matrix, $W_{i,j}$:

$$W_{i,j} = \begin{cases} \text{weight of the edge from } i \text{ to } j \\ \qquad \text{if the edge from } i \text{ to } j \text{ exists} \\ \infty \quad \text{if } i \neq j \text{ and there is no edge from } i \text{ to } j \\ 0 \quad \text{if } i = j \end{cases}$$

<u>Example</u>



$W_{i,j}$	1	2	3	4	5	6
1	0	4	∞	∞	∞	2
2	1	0	3	4	∞	∞
3	6	3	0	7	∞	∞
4	6	∞	∞	0	2	∞
5	∞	∞	∞	5	0	4
6	3	∞	∞	2	3	0

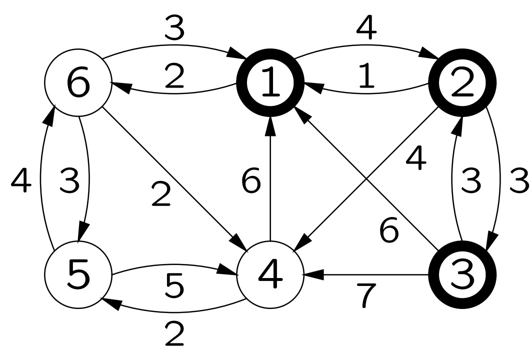
<u>Note:</u> Vertex i is denoted by circle labeled i. We will sometimes refer to this vertex as v_i to make it clear that it is a vertex.

(Step 1: Characterize Optimal Substructure

Define

$D_{i,j}^{(k)} =$ The length of the shortest path from v_i to v_j that uses only vertices in $\{v_1 \ldots v_k\}$ as intermediate vertices.

Example

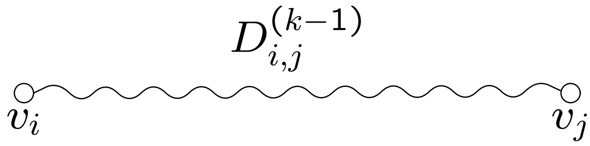


$D_{i,j}^{(3)}$	1	2	3	4	5	6
1	0	4	7	8	∞	2
2	1	0	3	4	∞	3
3	4	3	0	7	∞	6
4	6	10	13	0	2	8
5	∞	∞	∞	5	0	4
6	3	7	10	2	3	0

- $D_{i,j}^{(n)} = D_{i,j}$ (goal)
- $D_{i,j}^{(0)} = W_{i,j}$ (base cases)
- Need a strategy for computing $D_{i,j}^{(k)}$ from values of $D_{i,j}^{(k-1)}$

$D_{i,j}^{(k)}$ is the length of the shortest path from v_i to v_j that only visits vertices in $\{v_1, \ldots, v_k\}$. There are two possible cases:

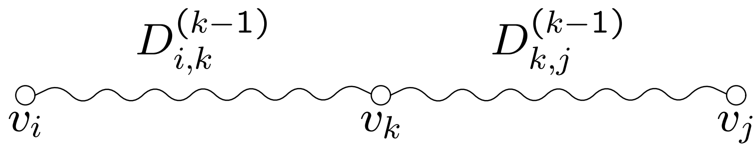
1. This path does not visit v_k.



   In this case:

$$D_{i,j}^{(k)} = D_{i,j}^{(k-1)}.$$

2. This path does visit v_k.



   In this case:

$$D_{i,j}^{(k)} = D_{i,k}^{(k-1)} + D_{k,j}^{(k-1)}$$

Hence

$$D_{i,j}^{(k)} = \min\left(D_{i,j}^{(k-1)}, D_{i,k}^{(k-1)} + D_{k,j}^{(k-1)}\right)$$

(Step 2: Develop recursive solution)

As we have just seen:

1. $D_{i,j}^{(n)} = D_{i,j}$ (goal)
2. $D_{i,j}^{(0)} = W_{i,j}$ (base case)
3. $D_{i,j}^{(k)} = \min\left(D_{i,j}^{(k-1)}, D_{i,k}^{(k-1)} + D_{k,j}^{(k-1)}\right)$
   (recurrence relation)

This can be used to derive a recursive solution. But the bottom-up dynamic programming solution is better . . .

(Step 3: Compute optimal costs (shortest distances) efficiently)

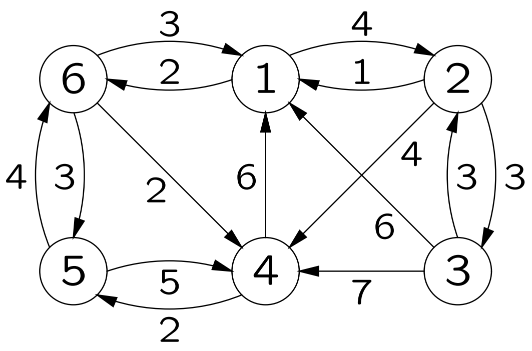
First version: use a triply dimensioned array $D[1..n, 1..n, 0..n]$, and store $D_{i,j}^{(k)}$ in $D[i, j, k]$:

```
procedure Floyd1(W[1..n,1..n]);
  array D[1..n,1..n,0..n]
begin {Floyd1}
  for i = 1 to n do
    for j = 1 to n do
      D[i,j,0] = W[i,j];;
    end { for };
  end { for };
  for k = 1 to n do
    for i = 1 to n do
      for j = 1 to n do
        D[i,j,k] = min(D[i,j,k-1], D[i,k,k-1] + D[k,j,k-1]);
      end { for };
    end { for };
  end { for };
end {Floyd1}
```

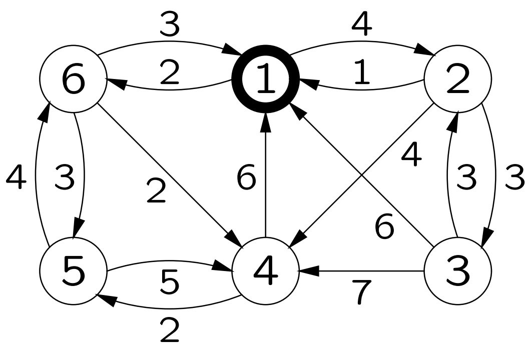
$O(n^3)$ time, $O(n^3)$ space.

We can improve the space requirement. But first, an example.

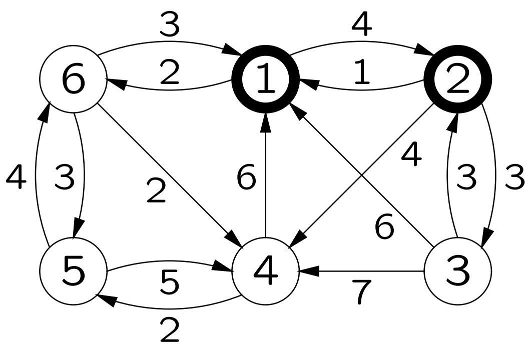
Complete Example



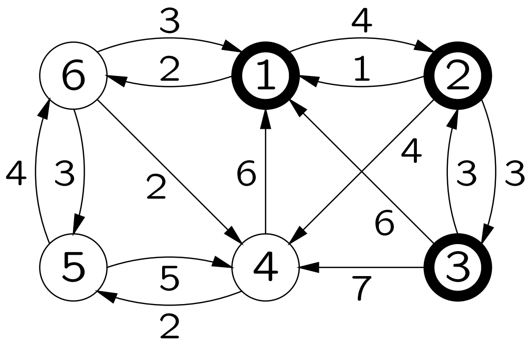
$D^{(0)}_{i,j}$	1	2	3	4	5	6
1	0	4	∞	∞	∞	2
2	1	0	3	4	∞	∞
3	6	3	0	7	∞	∞
4	6	∞	∞	0	2	∞
5	∞	∞	∞	5	0	4
6	3	∞	∞	2	3	0



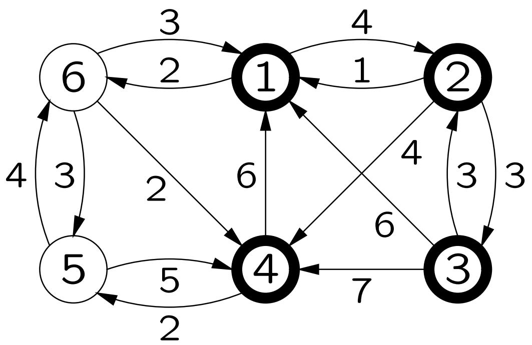
$D^{(1)}_{i,j}$	1	2	3	4	5	6
1	0	4	∞	∞	∞	2
2	1	0	3	4	∞	3
3	6	3	0	7	∞	8
4	6	10	∞	0	2	8
5	∞	∞	∞	5	0	4
6	3	7	∞	2	3	0



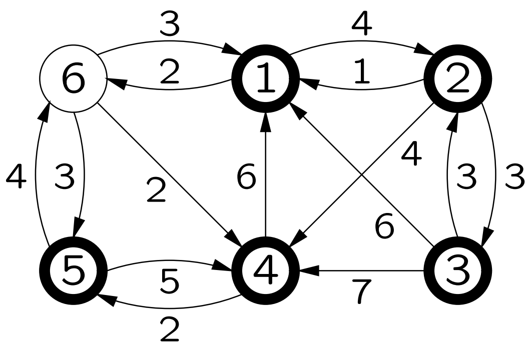
$D^{(2)}_{i,j}$	1	2	3	4	5	6
1	0	4	7	8	∞	2
2	1	0	3	4	∞	3
3	4	3	0	7	∞	6
4	6	10	13	0	2	8
5	∞	∞	∞	5	0	4
6	3	7	10	2	3	0



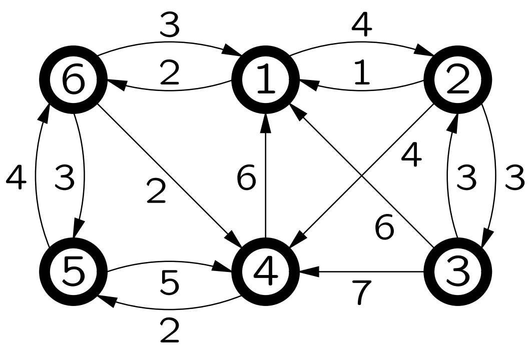
$D^{(3)}_{i,j}$	1	2	3	4	5	6
1	0	4	7	8	∞	2
2	1	0	3	4	∞	3
3	4	3	0	7	∞	6
4	6	10	13	0	2	8
5	∞	∞	∞	5	0	4
6	3	7	10	2	3	0



$D^{(4)}_{i,j}$	1	2	3	4	5	6
1	0	4	7	8	10	2
2	1	0	3	4	6	3
3	4	3	0	7	9	6
4	6	10	13	0	2	8
5	11	15	18	5	0	4
6	3	7	10	2	3	0



$D^{(5)}_{i,j}$	1	2	3	4	5	6
1	0	4	7	8	10	2
2	1	0	3	4	6	3
3	4	3	0	7	9	6
4	6	10	13	0	2	6
5	11	15	18	5	0	4
6	3	7	10	2	3	0



$D^{(6)}_{i,j}$	1	2	3	4	5	6
1	0	4	7	4	5	2
2	1	0	3	4	6	3
3	4	3	0	7	9	6
4	6	10	13	0	2	6
5	7	11	14	5	0	4
6	3	7	10	2	3	0

Improving the space usage in Floyd's algorithm.

Observation 1:

When computing $D_{i,j}^{(k)}$, we only need the values $D_{i,j}^{(k-1)}$, $D_{i,k}^{(k-1)}$, $D_{k,j}^{(k-1)}$. So we can get by with 2 $n \times n$ arrays, reducing space usage to $\Theta(n^2)$.

Observation 2: (Even better) . . .

When computing $D_{i,j}^{(k)}$, the computation depends on only three values:

1. $D_{i,j}^{(k-1)}$ (never used again)
2. $D_{i,k}^{(k-1)}$ $(= D_{i,k}^{(k)})$
3. $D_{k,j}^{(k-1)}$ $(= D_{k,j}^{(k)})$

So we can use one $n \times n$ array D, and update in place

Improved Floyd's algorithm:

```
procedure Floyd1(W[1..n,1..n]);
  array D[1..n,1..n]
begin {Floyd}
  for i = 1 to n do
    for j = 1 to n do
      D[i,j] = W[i,j];;
    end { for };
  end { for };
  for k = 1 to n do
    for i = 1 to n do
      for j = 1 to n do
        D[i,j] = min(D[i,j], D[i,k] + D[k,j]);
      end { for };
    end { for };
  end { for };
end {Floyd}
```

$O(n^3)$ time, $O(n^2)$ space.

(Step 4: Develop Optimal solution)—Encode shortest path

`next[i,j]` holds first vertex on shortest path from i to j, provided such a path exists.

Improved Floyd's algorithm:

```
procedure Floyd1(W[1..n,1..n]);
  array D[1..n,1..n]
begin {Floyd}
  for i = 1 to n do
    for j = 1 to n do
      D[i,j] = W[i,j];
      next[i,j] = j;
    end { for };
  end { for };
   for k = 1 to n do
    for i = 1 to n do
      for j = 1 to n do
        if D[i,k] + D[k,j] ¡ D[i,j] then
          D[i,j] = D[i,k] + D[k,j];
          next[i,j] = next[i,k];
        endif
      end { for };
    end { for };
  end { for };
end {Floyd}
```

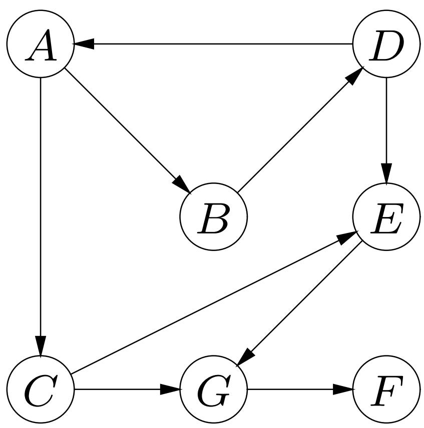
(Other solutions discussed in [CLRS], section 25.2)

Related problem: Transitive closure in a directed graph

Vertex w is reachable from vertex v if there is a path (containing at least one edge) from v to w.

Transitive closure problem: Given a graph, determine for all pairs of vertices v and w whether w is reachable from v.

Example



F is reachable from A

B is not reachable from C

Representation of Problem

Assume vertices are numbered: $v_1, \ldots, v_n$.

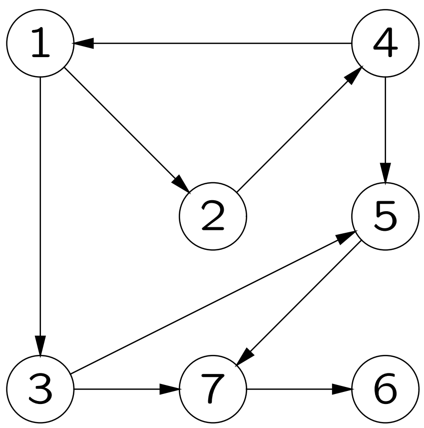
Input: Adjacency matrix A:

$$A_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

Output: Reachability matrix R:

$$R_{i,j} = \begin{cases} 1 & \text{if there is a nontrivial path from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

Example



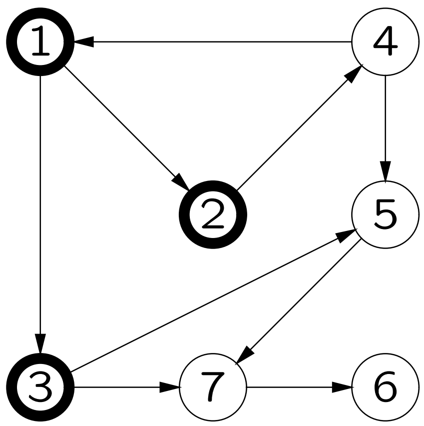
$A_{i,j}$	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	0	0	0	1	0	0	0
3	0	0	0	0	1	0	1
4	1	0	0	0	1	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

$R_{i,j}$	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1
3	0	0	0	0	1	1	1
4	1	1	1	1	1	1	1
5	0	0	0	0	0	1	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

<u>Warshall's algorithm</u> for computing transitive closure: very similar to Floyd's algorithm. Define

$$R_{i,j}^{(k)} = \begin{cases} 1 & \text{if there is a nontrivial path from } v_i \text{ to } v_j \\ & \text{using only vertices in } \{v_1, \ldots, v_k\} \text{ as intermediate} \\ & \text{vertices} \\ 0 & \text{otherwise} \end{cases}$$

<u>Example</u>

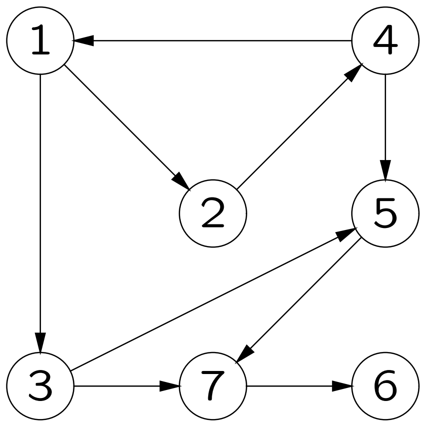


$R_{i,j}^{(3)}$	1	2	3	4	5	6	7
1	0	1	1	1	1	0	1
2	0	0	0	1	0	0	0
3	0	0	0	0	1	0	1
4	1	1	1	1	1	0	1
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

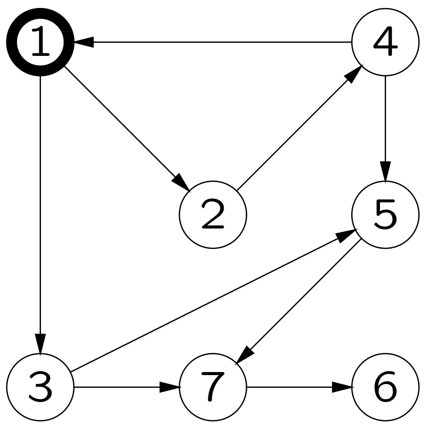
<u>Observations:</u>

1. $R_{i,j}^{(0)} = A_{i,j}$ (initial values)
2. $R_{i,j}^{(n)} = R_{i,j}$ (final values)
3. $R_{i,j}^{(k)} = R_{i,j}^{(k-1)} \vee \left(R_{i,k}^{(k-1)} \wedge R_{k,j}^{(k-1)} \right)$ (recurrence relation)

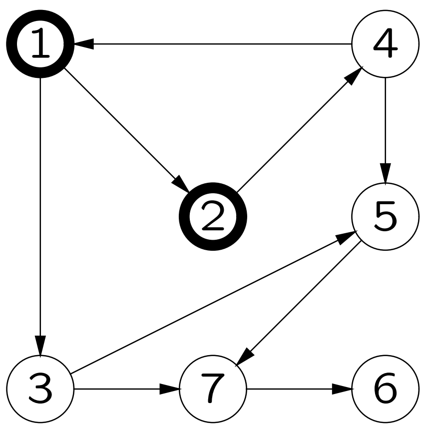
Complete Example



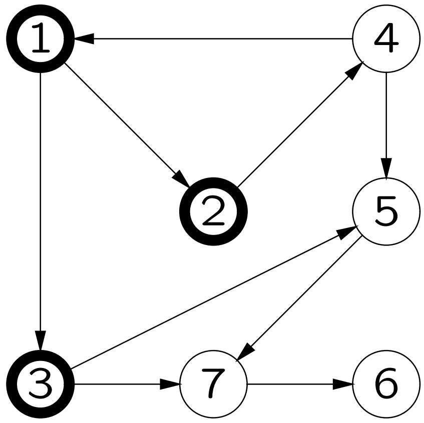
$R^{(0)}_{i,j}$	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	0	0	0	1	0	0	0
3	0	0	0	0	1	0	1
4	1	0	0	0	1	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0



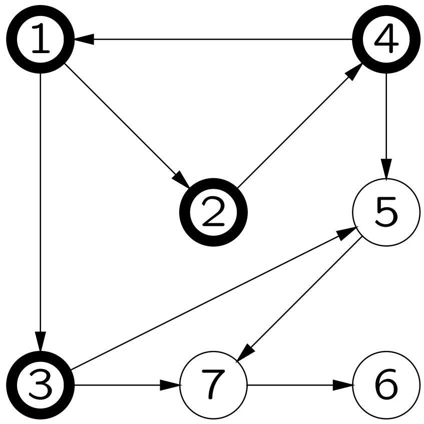
$R^{(1)}_{i,j}$	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	0	0	0	1	0	0	0
3	0	0	0	0	1	0	1
4	1	1	1	0	1	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0



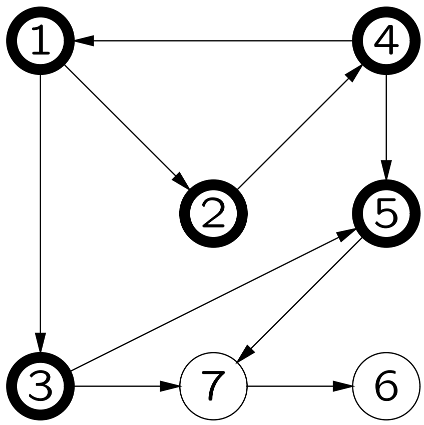
$R^{(2)}_{i,j}$	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	1	0	0	0
3	0	0	0	0	1	0	1
4	1	1	1	1	1	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0



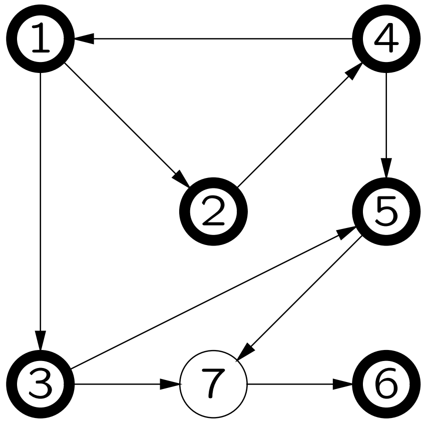
$R_{i,j}^{(3)}$	1	2	3	4	5	6	7
1	0	1	1	1	1	0	1
2	0	0	0	1	0	0	0
3	0	0	0	0	1	0	1
4	1	1	1	1	1	0	1
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0



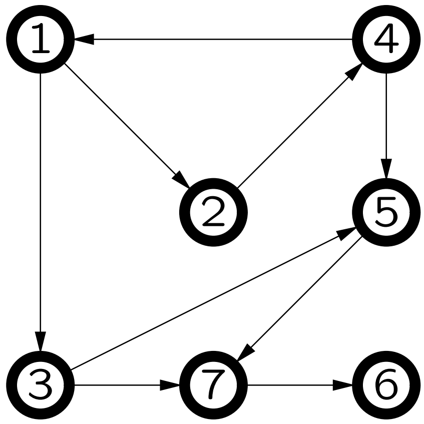
$R_{i,j}^{(4)}$	1	2	3	4	5	6	7
1	1	1	1	1	1	0	1
2	1	1	1	1	1	0	1
3	0	0	0	0	1	0	1
4	1	1	1	1	1	0	1
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0



$R_{i,j}^{(5)}$	1	2	3	4	5	6	7
1	1	1	1	1	1	0	1
2	1	1	1	1	1	0	1
3	0	0	0	0	1	0	1
4	1	1	1	1	1	0	1
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0



$R_{i,j}^{(6)}$	1	2	3	4	5	6	7
1	1	1	1	1	1	0	1
2	1	1	1	1	1	0	1
3	0	0	0	0	1	0	1
4	1	1	1	1	1	0	1
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0



$R_{i,j}^{(7)}$	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1
3	0	0	0	0	1	1	1
4	1	1	1	1	1	1	1
5	0	0	0	0	0	1	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

Code for Warshall's algorithm: same space-saving tricks as Floyd's algorithm

```
begin {Warshall}
  R := M;
  for k = 1 to n do
    for i = 1 to n do
      for j = 1 to n do
        if R[i,k] = 1 and R[k,j] = 1 then
          R[i,j] = 1;
        end { if };
      end { for };
    end { for };
  end { for };
end {Warshall}
```

Alternative codes for Warshall's algorithm:

```
begin {Warshall}
  R := M;
  for k = 1 to n do
    for i = 1 to n do
      if R[i,k] = 1
        for j = 1 to n do
          if R[k,j] = 1 then
            R[i,j] = 1;
          end { if };
        end { for };
      end { if };
    end { for };
  end { for };
end {Warshall}
```

```
begin {Warshall}
  R = M;
  for k = 1 to n do
    for i = 1 to n do
      if R[i,k] = 1 then
        for j = 1 to n do
          R[i,j] = R[i,j] ∨ R[k,j];
        end { for };
      end { if };
    end { for };
  end { for };
end {Warshall}
```

The last implementation may be faster because bit operations can be grouped, performed as logical operations on words.