Course: Analysis of Algorithms
Code: CS33104
Branch: MCA -3rd Semester

Lecture 6- Dynamic Programming Strategy

Faculty & Coordinator : Dr. J Sathish Kumar (JSK)

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

Introduction

- Dynamic programming is a technique for solving problems with overlapping sub-problems.
- Typically, these sub-problems arise from a recurrence relating a given problem's solution to solutions of its smaller sub-problems.
- Rather than solving overlapping sub-problems again and again, dynamic programming suggests
  - solving each of the smaller sub-problems only once and
  - recording the results in a table

  from which a solution to the original problem can then be obtained

Introduction

- Powerful algorithm design technique.
- Creeps up when you wouldn't expect, turning seemingly hard (exponential-time) problems into efficiently (polyonomial-time) solvable ones.
- Usually works when the obvious Divide & Conquer algorithm results in an exponential running time.
- Few problems when it doesn't work with greedy, dynamic programming gives solutions optimally.

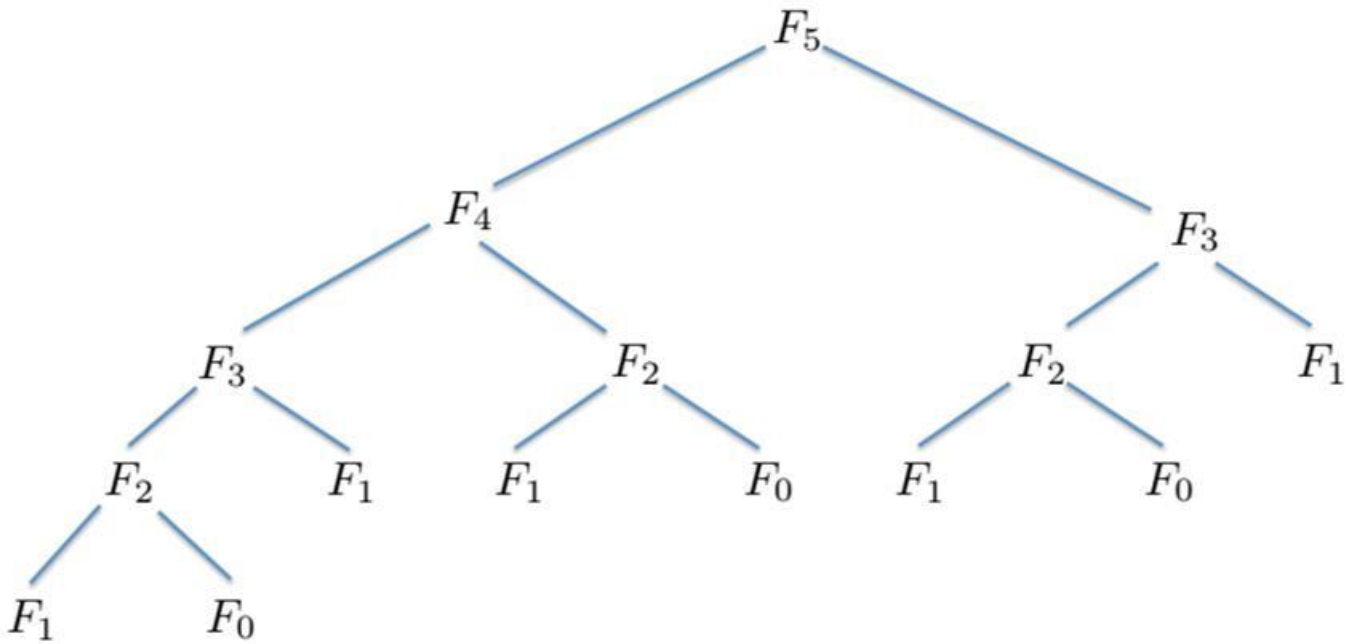
Applications

- Fibonacci Numbers
- Warshall's Algorithm
  - Warshall's Algorithm computing the transitive closure of a directed graph
- Floyd's Algorithm
  - Floyd's algorithm for the all-pairs shortest-paths problem
- Bellman ford algorithm
  - Shortest Path with Negative Edges
- Travelling Sales Man Problem
- The Knapsack Problem
- Matrix-chain multiplication
- Longest Subsequence Problem

Fibonacci Numbers

- Fibonacci Numbers: 0, 1, 1, 2, 3, 5, 8, 13, . . .
- It's the Fibonacci sequence, described by the recursive formula:
  - $F_0 := 0;$
  - $F_1 := 1;$
  - $F_n = F_{n-1} + F_{n-2}$, for all $n \geq 2$.
- The recurrence relation under consideration is that of the **Fibonacci Series,** stated as the
  - as sum of the time to calculate the (n-1)th term + time to calculate (n-2)th term + the time taken to add them together [O(1)].
  - **T(n) = T(n-1) + T(n-2) + O(1)**

Fibonacci Numbers



- N=0, 1 => 1
- N=2 => 3
- N=4 => 9
- N=5 => 15
- N=6 => 25
- N=10 => 177
- N=20 => 21891

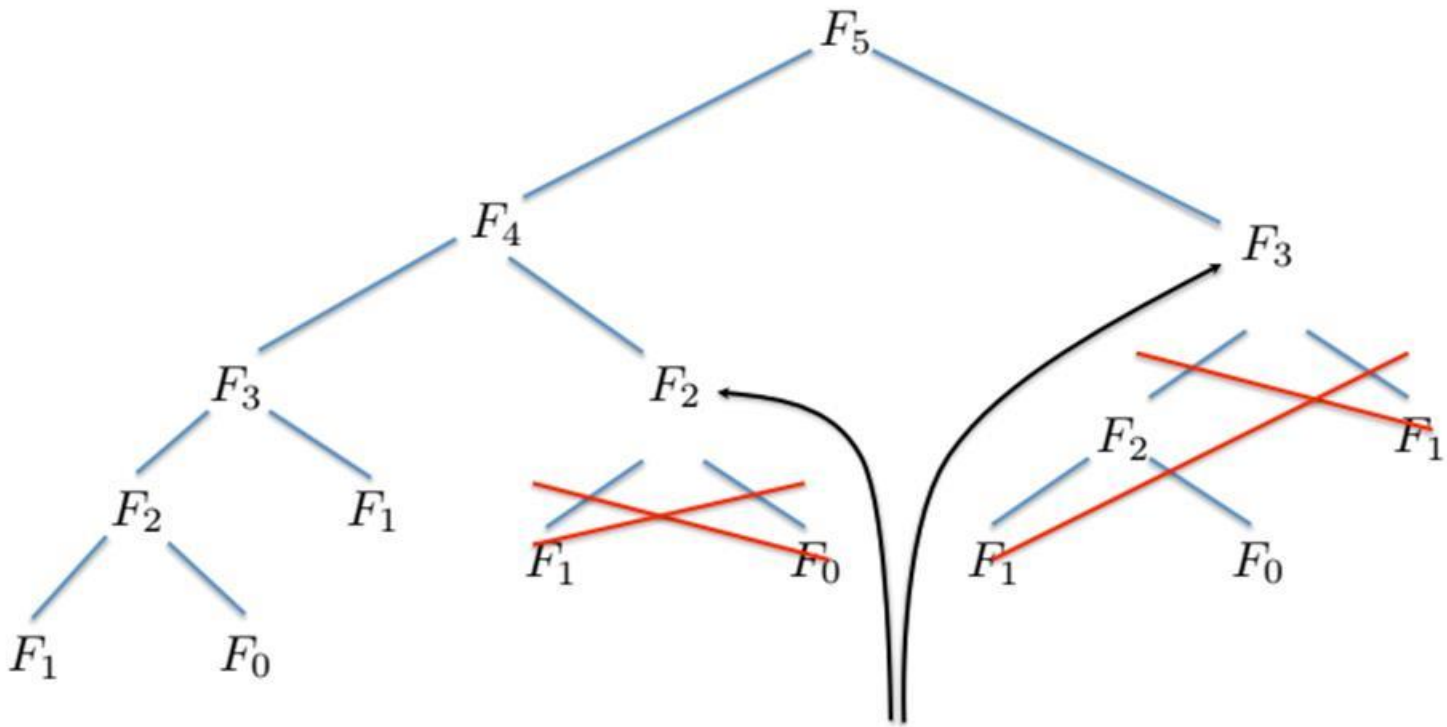
- Exponential Complexity = **$O(2^n)$**

Improved Fibonacci Algorithm

- Never recompute a subproblem $F(k)$, $k \leq n$, if it has been computed before.
- This technique of remembering previously computed values is called **memoization.**

```
memo = { }
fib(n):
        if n in memo: return memo[n]
        else if n = 0: return 0
            else if n = 1: return 1
            else: f = fib(n − 1) + fib(n − 2)
                                   free of charge!
            memo[n] = f
            return f
```

Improved Fibonacci Algorithm



these values are already computed and stored in memo *when runtime processes these nodes of the recursion*

- N=0, 1 => 1
- N=2 => 3
- N=4 => 4
- N=5 => 5
- N=6 => 6

- Complexity = **O(n)**

Improved Fibonacci Algorithm

Runtime, assuming n-bit registers for each entry of memo data structure:

$$T(n) = T(n-1) + c = O(cn),$$

where c is the time needed to add n-bit numbers. So $T(n) = O(n^2)$.

Note: There is also an $O(n \cdot \log n \cdot \log \log n)$- time algorithm for Fibonacci, via different techniques

Dynamic Programming (DP)

- DP ≈ recursion + memoization (i.e. re-use)
- DP ≈ “controlled brute force”
- DP results in an efficient algorithm, if the following conditions hold:
  - the optimal solution can be produced by combining optimal solutions of subproblems;
  - the optimal solution of each subproblem can be produced by combining optimal solutions of sub-subproblems, etc;
  - the total number of subproblems arising recursively is polynomial

Dynamic Programming (DP)

- **Implementation Trick:**
  - Remember (memoize) previously solved “subproblems"; e.g., in Fibonacci, we memoized the solutions to the subproblems *F*0*; F*1*; · · · ; Fn*-1, while unraveling the recursion.
  - if we encounter a subproblem that has already been solved, re-use solution.
- **Runtime** ≈ *No.* of subproblems * time/subproblem

Warshall's Algorithm

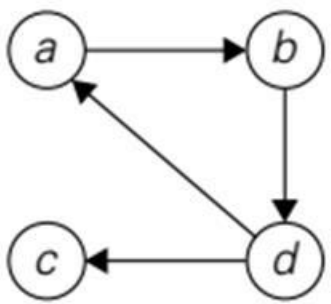
- In software engineering, transitive closure can be used for
  - investigating data flow and control flow dependencies,
  - as well as for inheritance testing of object-oriented software.
- In electronic engineering, it is used for redundancy identification and test generation for digital circuits.

Warshall's Algorithm

- Adjacency matrix $A = \{a_{ij}\}$ of a directed graph is the boolean matrix that has 1 in its i^{th} row and j^{th} column if and only if there is a directed edge from the i^{th} vertex to the j^{th} vertex.
- We may also be interested in a matrix containing the information about the existence of directed paths of arbitrary lengths between vertices of a given graph.
- Such a matrix, called the transitive closure of the digraph, would allow us to determine in constant time whether the j^{th} vertex is reachable from the i^{th} vertex.

Warshall's Algorithm

- **DEFINITION** The ***transitive closure*** of a directed graph with n vertices can be defined as the $n \times n$ boolean matrix $T = \{t_{ij}\}$, in which the element in the i^{th} row and the j^{th} column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the i^{th} vertex to the j^{th} vertex; otherwise, t_{ij} is 0.



(a)

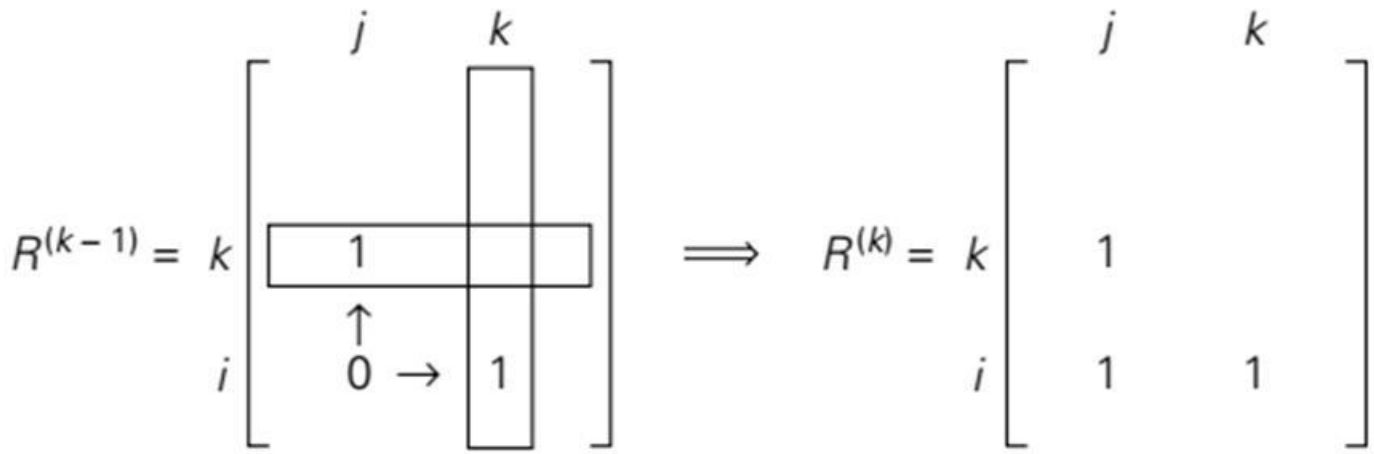
$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

(b)

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

(c)

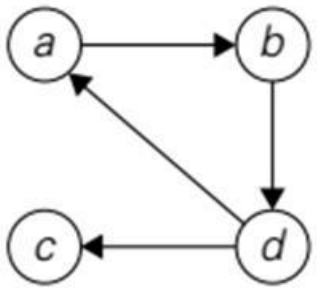
Warshall's Algorithm



- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$.

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \quad \text{or} \quad \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}\right).$$

Warshall's Algorithm



$R^{(0)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

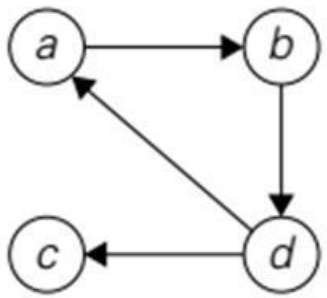
1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$R^{(1)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

Warshall's Algorithm



$R^{(1)} =$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

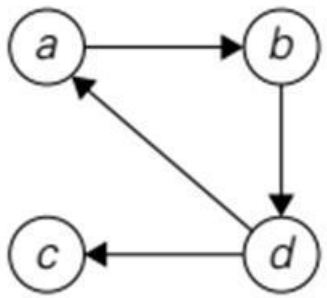
1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$R^{(2)} =$

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

Warshall's Algorithm



$R^{(2)} =$

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

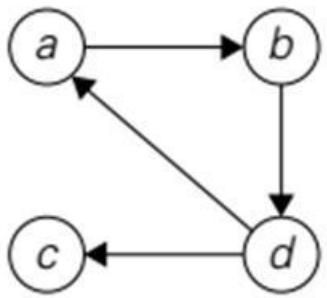
1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$R^{(3)} =$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a, b,* and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

Warshall's Algorithm



$R^{(3)} =$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths);
boxed row and column are used for getting $R^{(4)}$.

$R^{(4)} =$

	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

Warshall's Algorithm

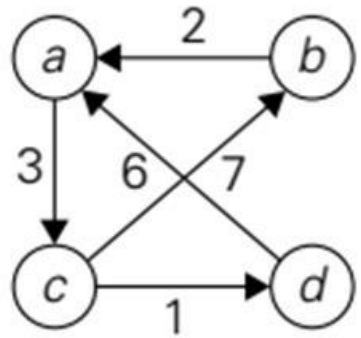
ALGORITHM *Warshall*$(A[1..n, 1..n])$

//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$

$$\Theta(n^3)$$

Floyd's Algorithm

- Given a weighted connected graph (undirected or directed), the ***all-pairs shortest paths problem*** asks to find the distances—i.e., the lengths of the shortest paths— from each vertex to all other vertices.
-



(a)

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

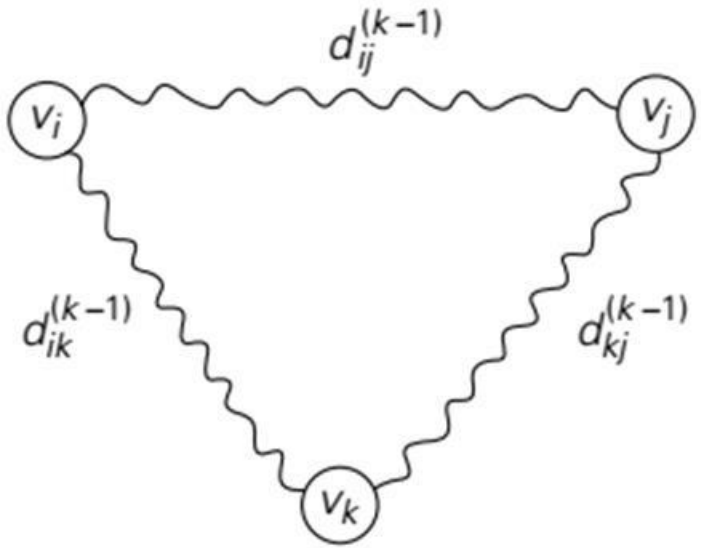
(b)

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

(c)

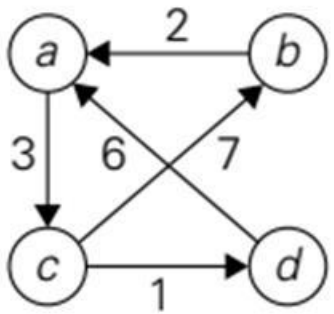
(a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

Floyd's Algorithm



$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1,\ \ d_{ij}^{(0)} = w_{ij}.$$

Floyd's Algorithm



$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

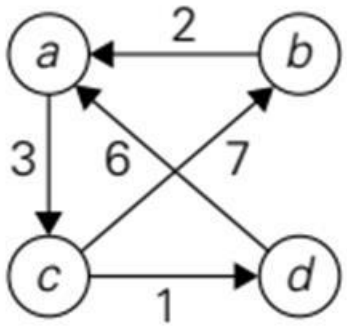
Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

Floyd's Algorithm



$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

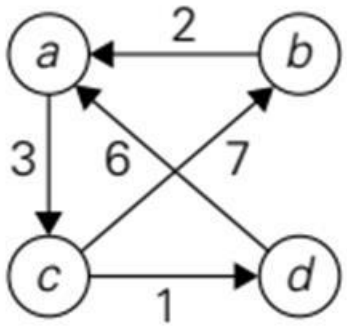
Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and *b* (note a new shortest path from *c* to *a*).

Floyd's Algorithm



$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

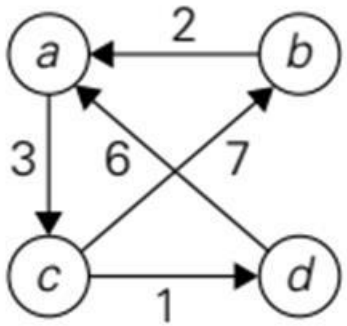
Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and *b* (note a new shortest path from *c* to *a*).

$D^{(3)} =$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

Floyd's Algorithm



$D^{(3)} =$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

Floyd's Algorithm

ALGORITHM *Floyd*$(W[1..n, 1..n])$

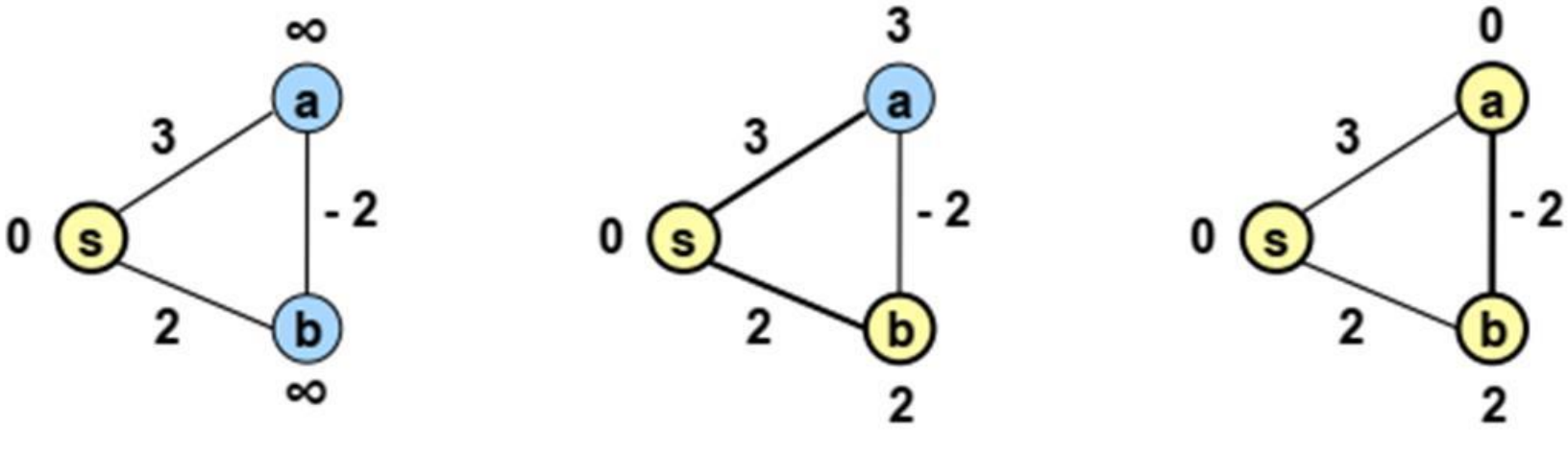
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
$D \leftarrow W$ //is not necessary if W can be overwritten
for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$
return D

$\Theta(n^3)$

Shortest Path with Negative Edges

- Dijkstra's algorithm earlier when discussed, it is assumed with non-negative edge weights.
- Dijkstra's algorithm is based on the assumption that the shortest path to the vertex v in the frontier that is closest to the set of visited vertices, whose distances have been determined, can be determined by considering just the incoming edges of v.
- With negative edge weights, this is not true anymore, because there can be a shorter path that ventures out of the frontier and then comes back to v

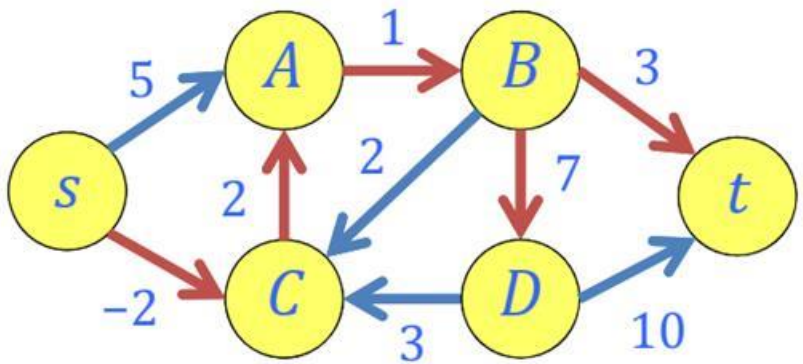
Example of *Dijkstra's property fails with negative edge weights*



Dijkstra's algorithm would visit b then a and leave b with a distance of 2 *instead of the correct distance* 1.
The problem is that when Dijkstra visits b, it fails to consider the possibility of there being a shorter path from a to b (which is impossible with nonnegative edge weights).

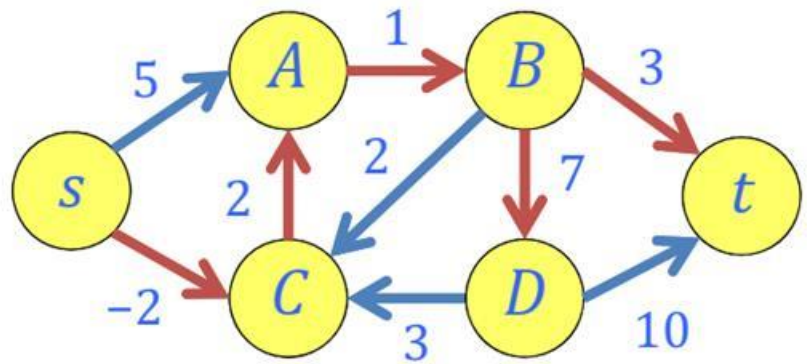
Bellman-Ford algorithm for single-source shortest paths

- $\delta(u, v) = \inf\{w(p) : p \text{ is a path from } u \text{ to } v\}$
- $\delta(u, v) = \infty$ if there's no path from u to v
- $\delta(u, v) = -\infty$ if there's a path from u to v that visits a negative-weight cycle



$\delta(s, t) = -\infty$

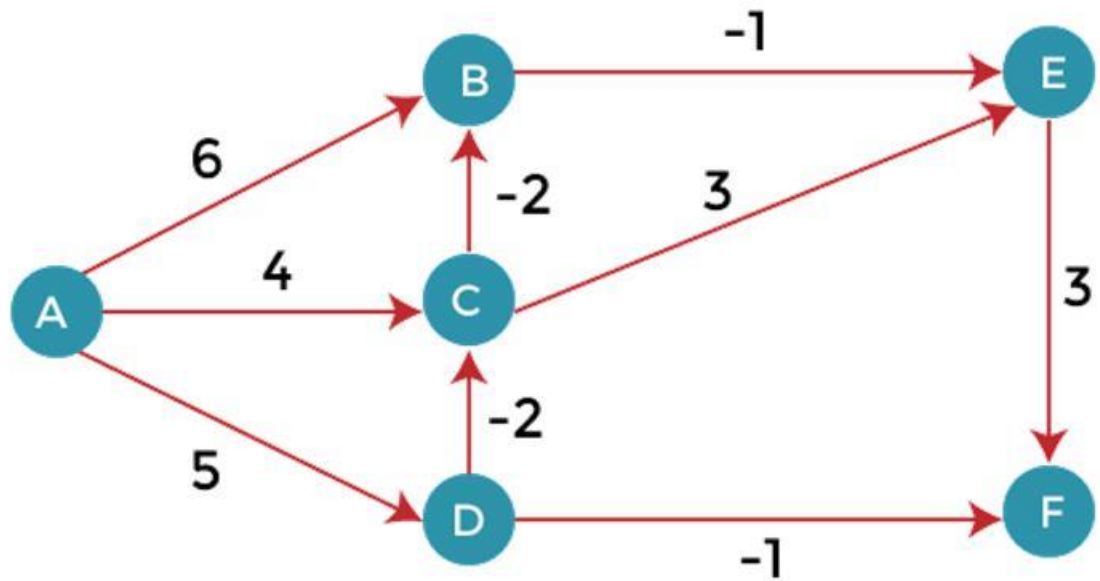
Bellman-Ford algorithm for single-source shortest paths



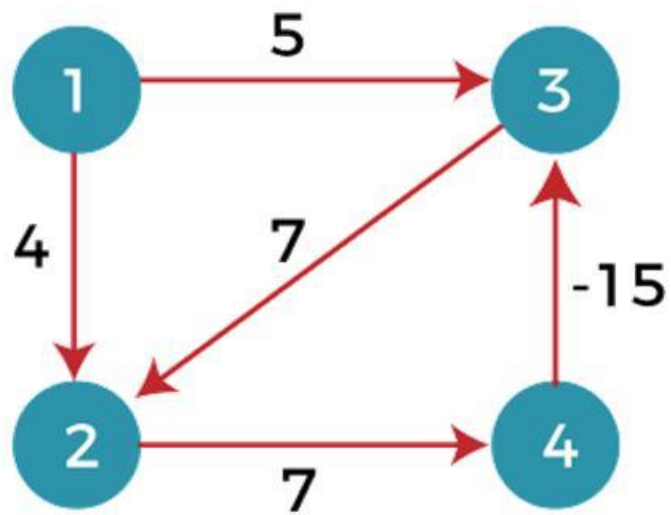
$$\text{if } v.d > u.d + w(u,v):$$
$$v.d = u.d + w(u,v)$$

(s,A), (S,C), (A,B), (C,A), (B,C), (B,D), (D,C), (B,t), (D,t)

Class Exercise #2

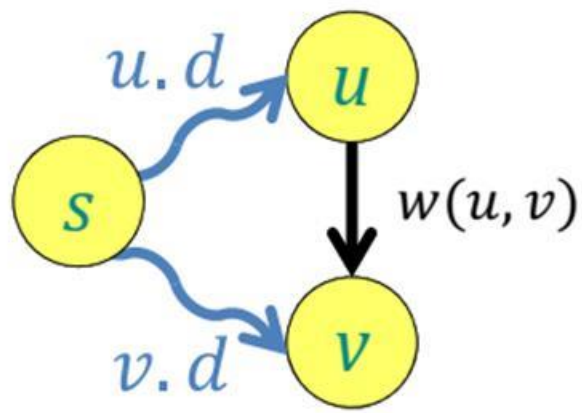


Class Exercise #3

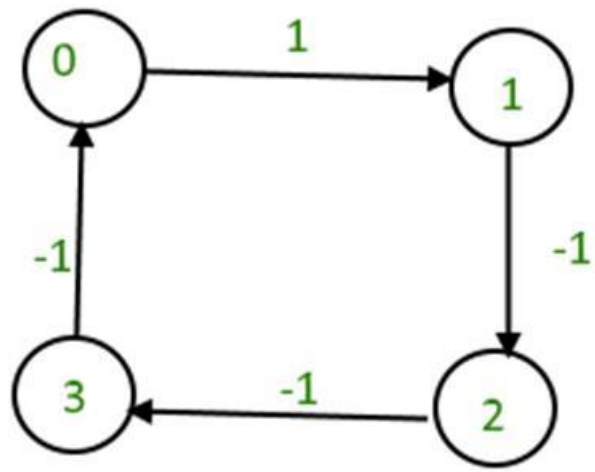


Bellman-Ford Algorithm

```
for v in V:
    v.d = ∞
    v.π = None
s.d = 0
for i from 1 to |V| − 1:
    for (u, v) in E:
        relax(u, v):
            if v.d > u.d + w(u, v):
                v.d = u.d + w(u, v)
                v.π = u
```

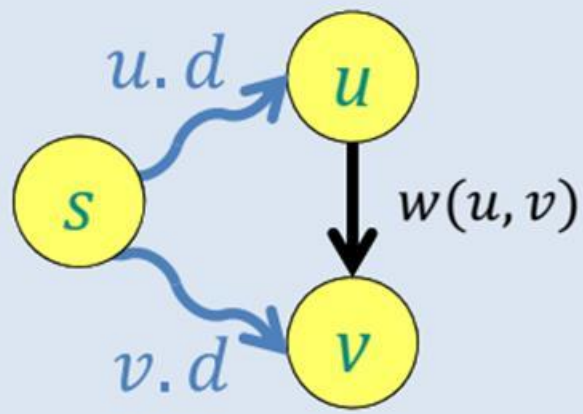


Bellman-Ford Algorithm with Negative-Weight Cycle Detection



Bellman-Ford Algorithm with Negative-Weight Cycle Detection

```
for v in V:
    v.d = ∞
    v.π = None
s.d = 0
for i from 1 to |V| − 1:
    for (u, v) in E:
        relax(u, v)
for (u, v) in E:
    if v.d > u.d + w(u, v):
        report that a negative-weight cycle exists
```

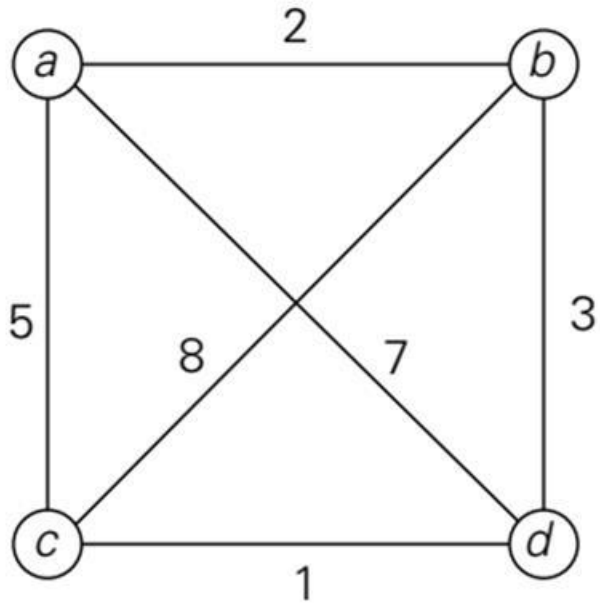


O(VE)

Introduction

- The ***traveling salesman problem (TSP)*** has been intriguing researchers for the last 150 years by its seemingly simple formulation, important applications, and interesting connections to other combinatorial problems.
- The problem asks to find the shortest tour through a given set of *n* cities that visits each city exactly once before returning to the city where it started.
- The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest ***Hamiltonian circuit*** of the graph.

Introduction



Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

Brute Force and Exhaustive Search

- Thus, we can get all the tours by generating all the permutations of n - 1 intermediate cities, compute the tour lengths, and find the shortest among them i.e. $(n - 1)!$ which makes the exhaustive-search approach impractical for all but very small values of n.
- In fact, these problem is the best-known examples of so called ***NP-hard problems***.
- No polynomial-time algorithm is known for any *NP*-hard problem. Hence, approximation algorithms

TSP: Dynamic Programming technique

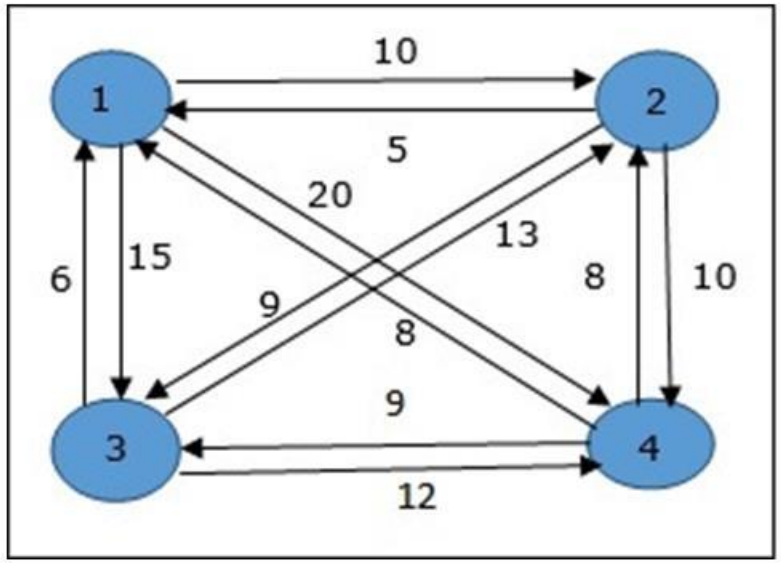
- Suppose we have started at city ***1*** and after visiting some cities now we are in city ***j***. Hence, this is a partial tour. We certainly need to know ***j***, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.
- For a subset of cities ***S ∈ {1, 2, 3, ... , n}*** that includes ***1***, and ***j ∈ S***, let ***C(S, j)*** be the length of the shortest path visiting each node in **S** exactly once, starting at ***1*** and ending at ***j***.
- When ***|S|*** > 1, we define ***C(S, 1)*** = ∝ since the path cannot start and end at **1**.

TSP: Dynamic Programming technique

- Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that

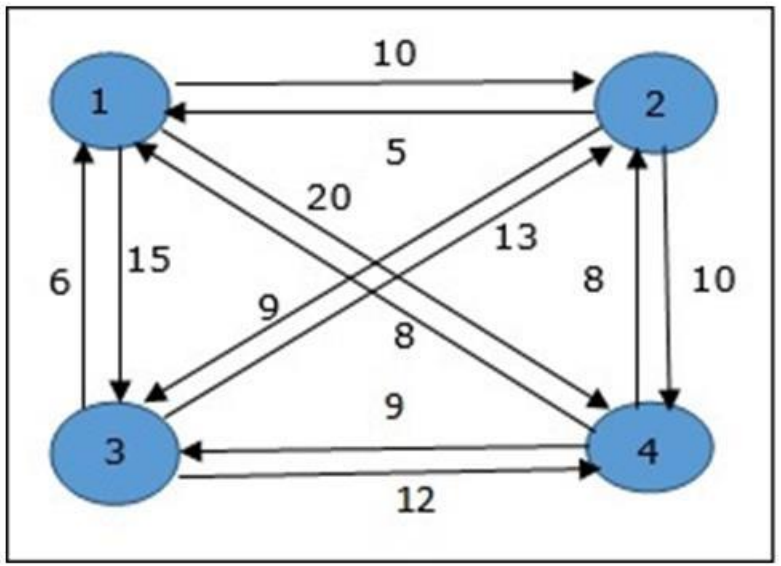
$$C(S, j) = min\, C(S - \{j\}, i) + d(i, j)\, where\, i \in S\, and\, i \neq j$$

TSP: Dynamic Programming technique



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

TSP: Dynamic Programming technique



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

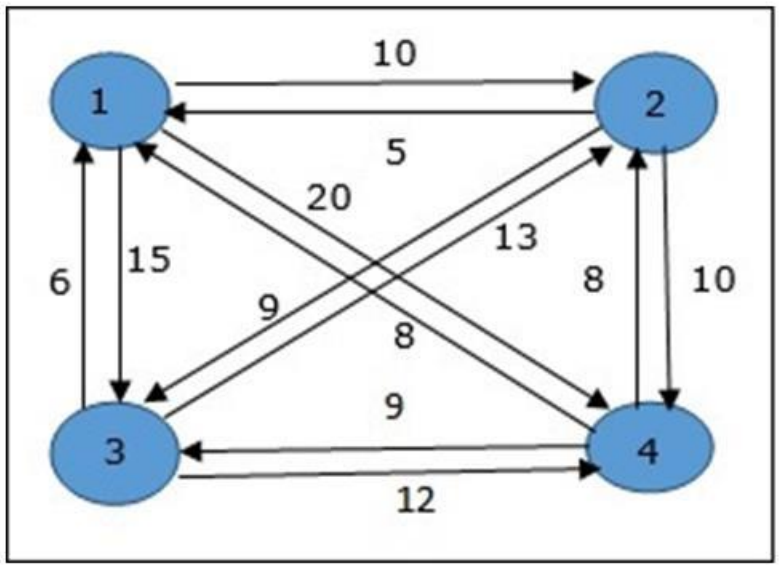
$S = \Phi$

$Cost(2,\Phi,1)=d(2,1)=5$

$Cost(3,\Phi,1)=d(3,1)=6$

$Cost(4,\Phi,1)=d(4,1)=8$

TSP: Dynamic Programming technique



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

S = 1

Cost(i,s)=min{Cost(j,s−(j))+d[i,j]}Cost(i,s)=min{Cost(j,s)−(j))+d[i,j]}

Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1)=9+6=15

Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=18

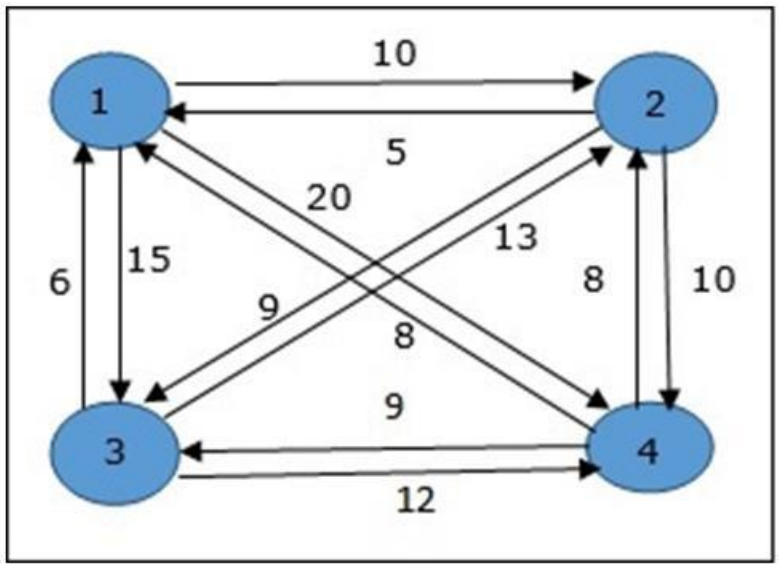
Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18

Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20

Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15

Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13

TSP: Dynamic Programming technique



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

S = 2

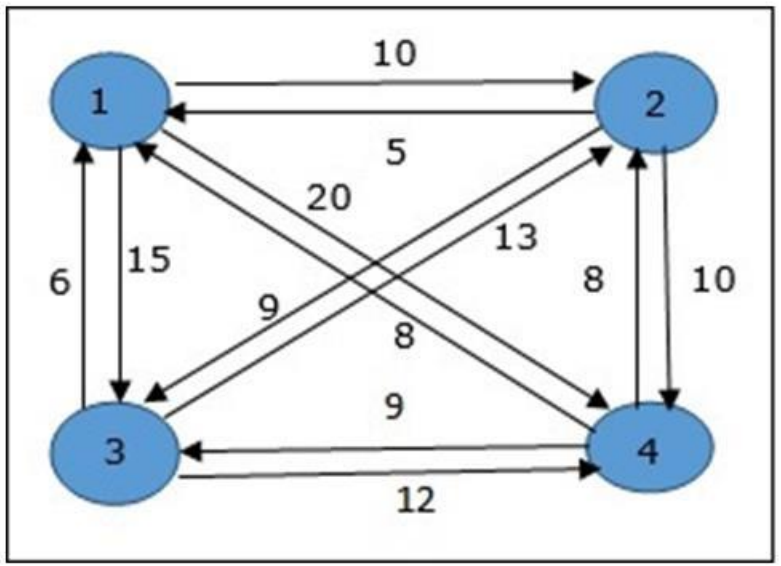
Cost(i,s)=min{Cost(j,s–(j))+d[i,j]}Cost(i,s)=min{Cost(j,s)−(j))+d[i,j]}

Cost(2,{3,4},1)= min(d[2,3]+Cost(3,{4},1)=9+20=29,
d[2,4]+Cost(4,{3},1)=10+15=25)=25

Cost(3,{2,4},1)= min (d[3,2]+Cost(2,{4},1)=13+18=31,
d[3,4]+Cost(4,{2},1)=12+13=25)=25

Cost(4,{2,3},1)= min (d[4,2]+Cost(2,{3},1)=8+15=23,
d[4,3]+Cost(3,{2},1)=9+18=27)=23

TSP: Dynamic Programming technique



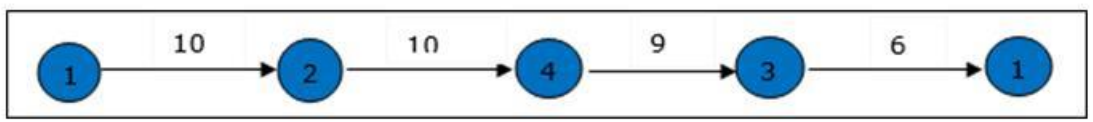
	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

S = 3

Cost(i,s)=min{Cost(j,s–(j))+d[i,j]}Cost(i,s)=min{Cost(j,s)−(j))+d[i,j]}

Cost(1,{2,3,4},1)= min(d[1,2]+Cost(2,{3,4},1)=10+25=35,
d[1,3]+Cost(3,{2,4},1)=15+25=40,
d[1,4]+Cost(4,{2,3},1)=20+23=43)=35

The minimum cost path is 35.
Start from cost **{1, {2, 3, 4}, 1}**, we get the minimum value for **d [1, 2]**. When **s = 3**, select the path from 1 to 2 (cost is 10) then go backwards. When **s = 2**, we get the minimum value for **d [4, 2]**. Select the path from 2 to 4 (cost is 10) then go backwards.



TSP: Dynamic Programming technique

Algorithm: Traveling-Salesman-Problem

C ({1}, 1) = 0
for s = 2 to n do
  for all subsets S ∈ {1, 2, 3, … , n} of size s and containing 1
    C (S, 1) = ∞
  for all j ∈ S and j ≠ 1
    C (S, j) = min {C (S – {j}, i) + d(i, j) for i ∈ S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

There are at the most $2^n.n$ sub-problems and each one takes linear time to solve.
Therefore, the total running time is $O(2^n.n^2)$

The Knapsack Problem

- Given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.
- To design a dynamic programming algorithm, we need to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller sub-instances.

The Knapsack Problem

- Divide all the subsets of the first i items that fit the knapsack of capacity j into two categories
- Those that do not include the i^{th} item and those that do.
  1. Among the subsets that do not include the i^{th} item, the value of an optimal subset is, by definition, $F(i - 1, j)$.
  2. Among the subsets that do include the i^{th} item (hence, $j - w_i \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i - 1$ items that fits into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + F(i - 1, j - w_i)$.

The Knapsack Problem

- Divide all the subsets of the first *i* items that fit the knapsack of capacity *j* into two categories
- Those that do not include the i^{th} item and those that do.
  1. Among the subsets that do not include the i^{th} item, the value of an optimal subset is, by definition, $F(i - 1, j)$.
  2. Among the subsets that do include the i^{th} item (hence, $j - w_i \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i - 1$ items that fits into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + F(i - 1, j - w_i)$.

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

Example

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

capacity $W = 5$.

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

The Knapsack Problem

- Thus, the maximal value is $F(4, 5) = \$37$.
- Since $F(4, 5) > F(3, 5)$, item 4 has to be included in an optimal solution along with an optimal subset for filling 5 - 2 = 3 remaining units of the knapsack capacity.
- The value of the latter is $F(3, 3)$. Since $F(3, 3) = F(2, 3)$, item 3 need not be in an optimal subset.
- Since $F(2, 3) > F(1, 3)$, item 2 is a part of an optimal selection, which leaves element $F(1, 3 - 1)$ to specify its remaining composition.
- Similarly, since $F(1, 2) > F(0, 2)$, item 1 is the final part of the optimal solution {item 1, item 2, item 4}.

The Knapsack Problem

ALGORITHM *MFKnapsack*(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//       items being considered and a nonnegative integer j indicating
//       the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
//and table F[0..n, 0..W] whose entries are initialized with −1's except for
//row 0 and column 0 initialized with 0's
if F[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                    Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← value
return F[i, j]
```

The time needed to find the composition of an optimal solution is in O(n)

Matrix-chain multiplication

- Given a sequence (chain) $(A_1, A_2, \ldots\ldots A_n)$ of n matrices to be multiplied, and we wish to compute the product

$$A_1 A_2 \ldots\ldots A_n$$

- Matrix multiplication is associative, and so all parenthesizations yield the same product.
- For example, if the chain of matrices is (A_1, A_2, A_3, A_4) , then we can fully parenthesize the product in five distinct ways:

$$\begin{aligned}
&(A_1(A_2(A_3A_4))) ,\\
&(A_1((A_2A_3)A_4)) ,\\
&((A_1A_2)(A_3A_4)) ,\\
&((A_1(A_2A_3))A_4) ,\\
&(((A_1A_2)A_3)A_4) .
\end{aligned}$$

Matrix-chain multiplication

- To illustrate the different costs, consider the problem of a chain (A_1, A_2, A_3) of three matrices.
- Suppose that the dimensions of the matrices are 10x100, 100x5, and 5x50, respectively.
- $((A_1A_2)A_3)$
  - (10 . 100 . 5) = 5000 scalar multiplications to compute the 10x5 matrix product A_1A_2,
  - Plus another (10 . 5 . 50) = 2500 scalar multiplications to multiply this matrix by A3
  - Total of 7500 scalar multiplications.
- $(A_1(A_2A_3))$
  - (100 . 5 . 50) = 25,000 scalar multiplications to compute the 100x50 matrix product A2A3
  - Plus another (10 . 100 . 50)= 50,000 scalar multiplications to multiply A1 by this matrix,
  - Total of 75,000 scalar multiplications.
- Thus, computing the product according to the first parenthesization is 10 times faster.

Matrix-chain multiplication

- We state the ***matrix-chain multiplication problem*** as follows:
- Given a chain $(A_1, A_2, \ldots\ldots A_n)$ of n matrices, where for i =1, 2,…., n, matrix A_i has dimension p_{i-1} x p_i, fully parenthesize the product $A_1A_2\ldots\ldots A_n$ in a way that minimizes the number of scalar multiplications.
- Our goal is only to determine an order for multiplying matrices that has the lowest cost.

Matrix-chain multiplication

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \sum_{k=1}^{n-1} P(k) P(n-k) & \text{if } n \geq 2 . \end{cases}$$

Where, P_n is the number of alternative parenthesizations of a sequence of n matrices

Brute Force $\Omega(2^n)$

Assignment #4

- Apply Dynamic Programming Technique to Matrix-chain multiplication problem and verify whether the complexity could be decreased or not? If it could, then by how much will be complexity and detail the proposed solution using pseudo code.
- Apply Dynamic Programming Technique to Longest Subsequence Problem.