CS211FZ: Data Structures and Algorithms II
11- Dynamic Programming
Flyod Warshall Algorithm

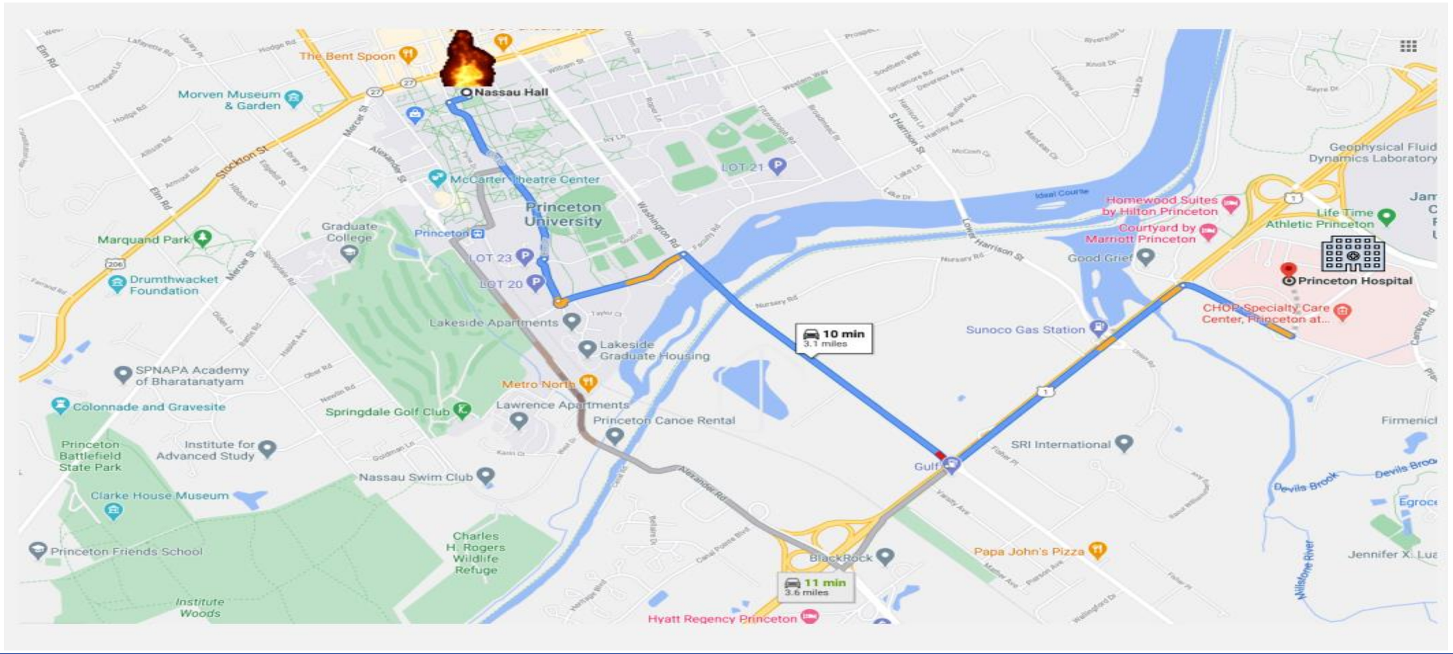
LECTURER: MAHWISH KUNDI & IRFAN AHMAD

MAHWISH.KUNDI@MU.IE , IRFAN.AHMAD@MU.IE

Introduction

Algorithms for Optimization Problems

- Designed to find the best solution among a set of possible solutions.
- For example:
  - Finding the shortest route or maximizing profits.



Algorithms for Optimization Problems

- Commonly used algorithms for solving optimization problems include:
  - Dynamic Programing
  - Greedy Algorithms

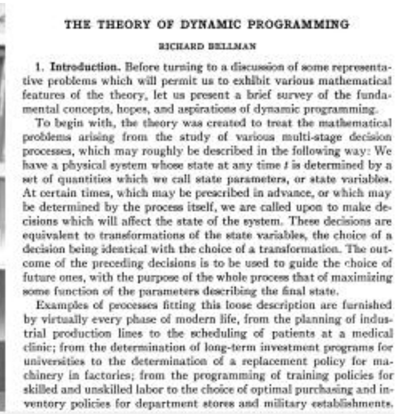
Dynamic Programming Vs Greedy Algorithm

- Both are employed to find best solution among various possibilities.

Dynamic programing	Greedy algorithms
• Divides the complex problem into small subproblems.	• Divides the complex problem into small subproblems.
• Make a choice at each step.	• Make a choice at each step.
• Choice depends on knowing optimal solutions to subproblems. Solve subproblems first.	• Make the choice before solving the subproblems.
• Solve bottom-up. More optimal but slower.	• Solve top-down. Less optimal but quicker.

Dynamic Programming - Definition

- Richard Bellman introduced the idea of dynamic programming in the 1950s.
- Algorithm design paradigm.
  - Divide a complex problem into a number of simpler overlapping subproblems.
  - Solve each subproblem only once.
  - Store the results of each solved sub-problem for later reuse.
  - Use the stored answers to find the overall solution.



THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time *t* is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

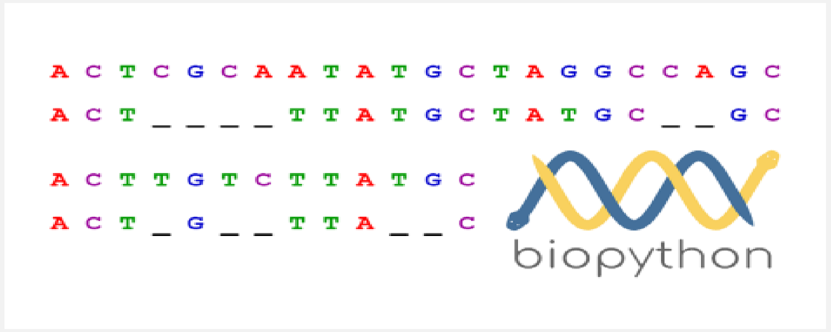
Richard Bellman, *46

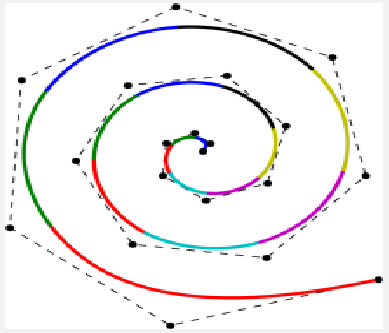
Dynamic Programming - Application Areas

- Operations research: multistage decision processes, control theory, optimization, ...
- Computer science: AI, compilers, systems, graphics, databases, robotics, theory, ….
- Economics.
- Bioinformatics.
- Information theory.
- Tech job interviews.

Dynamic Programming - Algorithms

- Some famous algorithms:
  - System R algorithm for optimal join order in relational databases.
  - Needleman–Wunsch/Smith–Waterman for sequence alignment.
  - Cocke–Kasami–Younger for parsing context-free grammars.
  - Bellman–Ford–Moore for shortest path.
  - De Boor for evaluating spline curves.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.


A C T C G C A A T A T G C T A G G C C A G C
A C T _ _ _ _ T T A T G C T A T G C _ _ G C
A C T T G T C T T A T G C
A C T _ G _ _ T T A _ _ C
biopython



Dynamic Programming - Some Books

PRINCETON LANDMARKS IN MATHEMATICS
Richard Bellman
Dynamic Programming

Huaguang Zhang, Derong Liu, Yanhong Luo, Ding Wang
Adaptive Dynamic Programming for Control
Algorithms and Stability
Springer

Dynamic Programming: Sequential Decision Making
Williams, Kenneth

WILEY
Markov Decision Processes
Discrete Stochastic Dynamic Programming
MARTIN L. PUTERMAN
WILEY SERIES IN PROBABILITY AND STATISTICS

Dynamic Programming
Foundations and Principles
Second Edition
Moshe Sniedovich

SECOND EDITION
Approximate Dynamic Programming
Solving the Curses of Dimensionality
Warren B. Powell
WILEY

DYNAMIC PROGRAMMING FOR CODING INTERVIEWS
A BOTTOM-UP APPROACH TO PROBLEM SOLVING
MEENAKSHI & KAMAL RAWAT

COMPUTER SCIENCE
An Interdisciplinary Approach
ROBERT SEDGEWICK
KEVIN WAYNE

pp. 284–289

Applied Dynamic Programming for Optimization of Dynamical Systems
Rush D. Robinett III, David G. Wilson, G. Richard Eisler, John E. Hurtado
siam

Introduction to Dynamic Programming
Leon Cooper and Mary W Cooper
Pergamon Press

WILEY
Approximate Dynamic Programming
Solving the Curses of Dimensionality
Warren B. Powell

Dynamic Programming of Economic Decisions

Dynamic Programming with Management Applications (Operational Research)
Hastings, N.A.J.

HANDBOOK OF LEARNING AND APPROXIMATE DYNAMIC PROGRAMMING
JENNIE SI, ANDREW G. BARTO, WARREN B. POWELL, DONALD WUNSCH II

Dynamic Programming and Optimal Control
Dimitri P. Bertsekas

The Art and Theory of Dynamic Programming

DYNAMIC PROGRAMMING
Models and Applications
Eric V. Denardo

Marlin Wolf Ulmer
Approximate Dynamic Programming for Dynamic Vehicle Routing
Springer

SPRINGER BRIEFS IN QUANTITATIVE FINANCE
Pablo Azcue, Nora Muler
Stochastic Optimization in Insurance
A Dynamic Programming Approach
Springer

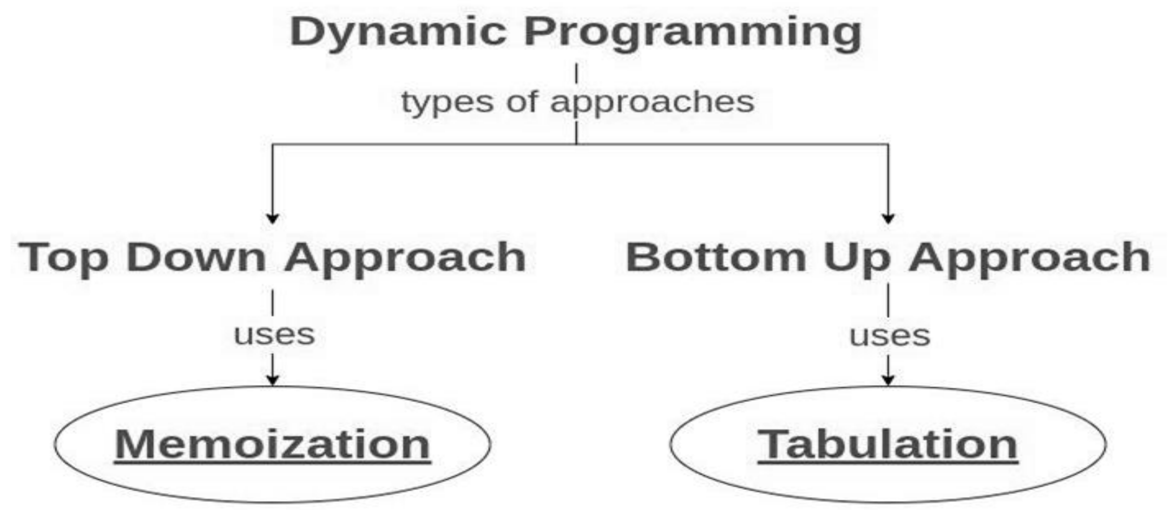
Dynamic Programming and Markov Processes
Howard

ITERATIVE DYNAMIC PROGRAMMING
REIN LUUS
CHAPMAN & HALL/CRC

NEURO-DYNAMIC PROGRAMMING
DIMITRI P. BERTSEKAS
JOHN N. TSITSIKLIS

Dynamic Programming - Techniques to Solve Problems

- Dynamic programming is divided into two main approaches:



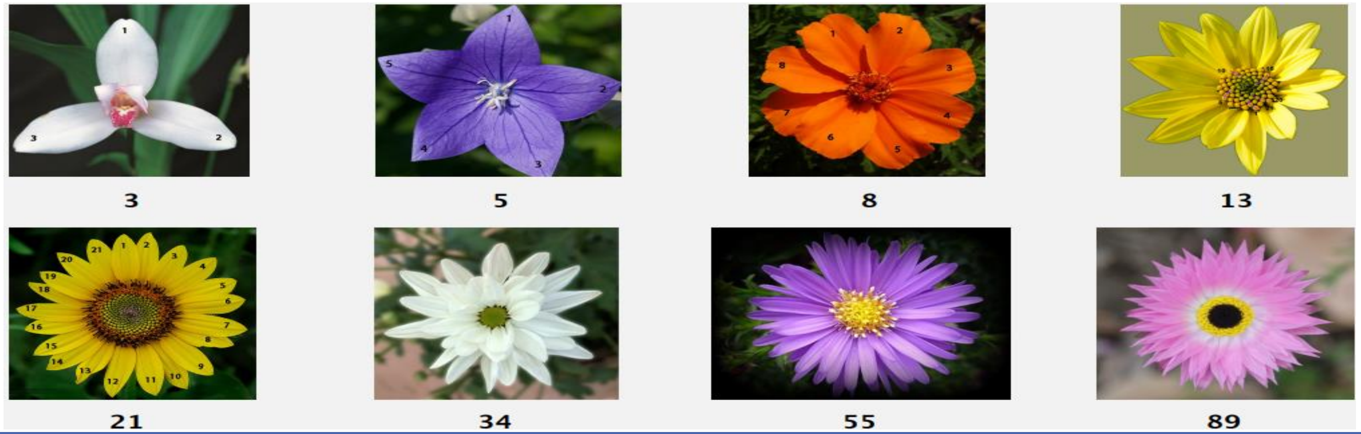
- Both help in solving complex problems more efficiently by storing and reusing solutions of overlapping subproblems, but they differ in the way they work.

Example - Fibonacci numbers

- Fibonacci numbers. 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

$$F_i = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ F_{i-1} + F_{i-2} & \text{if } i > 1 \end{cases}$$

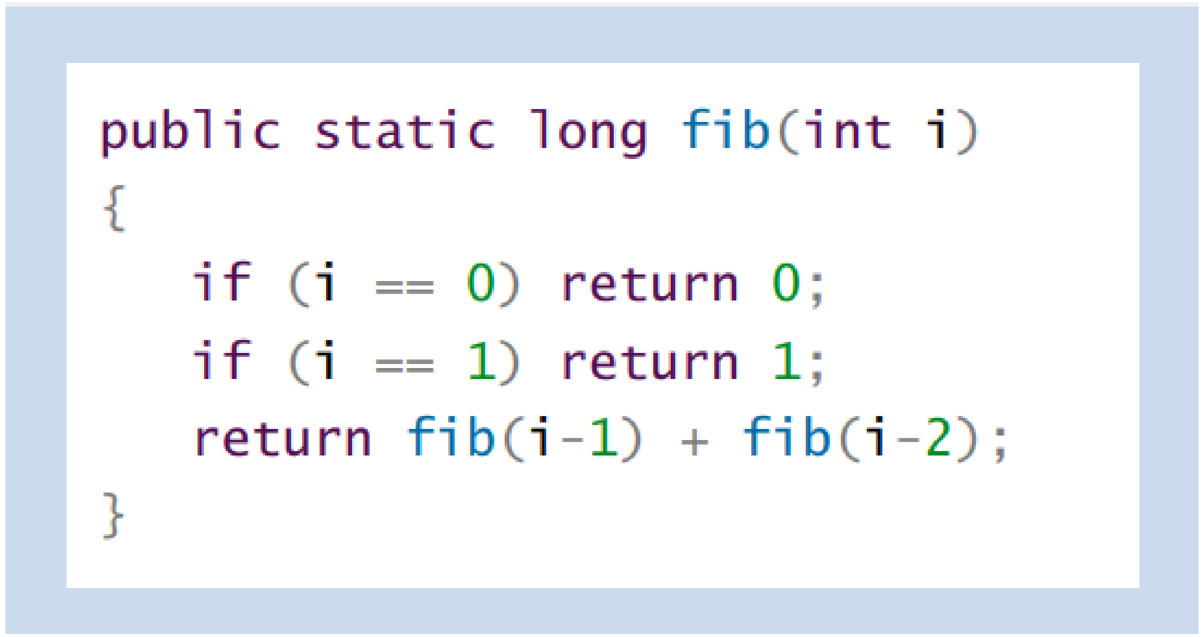
Leonardo Fibonacci



Flower Patels follow Fibonacci numbers.

Fibonacci numbers: naïve recursive approach

- Goal: Given n, compute the nth Fibonacci number F_n.
- Naïve recursive approach:



```
public static long fib(int i)
{
   if (i == 0) return 0;
   if (i == 1) return 1;
   return fib(i-1) + fib(i-2);
}
```

Fibonacci numbers: naïve recursive approach

- How long to compute fib(80) using the naïve recursive algorithm?

  A. Less than 1 second.

  B. About 1 minute.

  C. More than 1 hour.

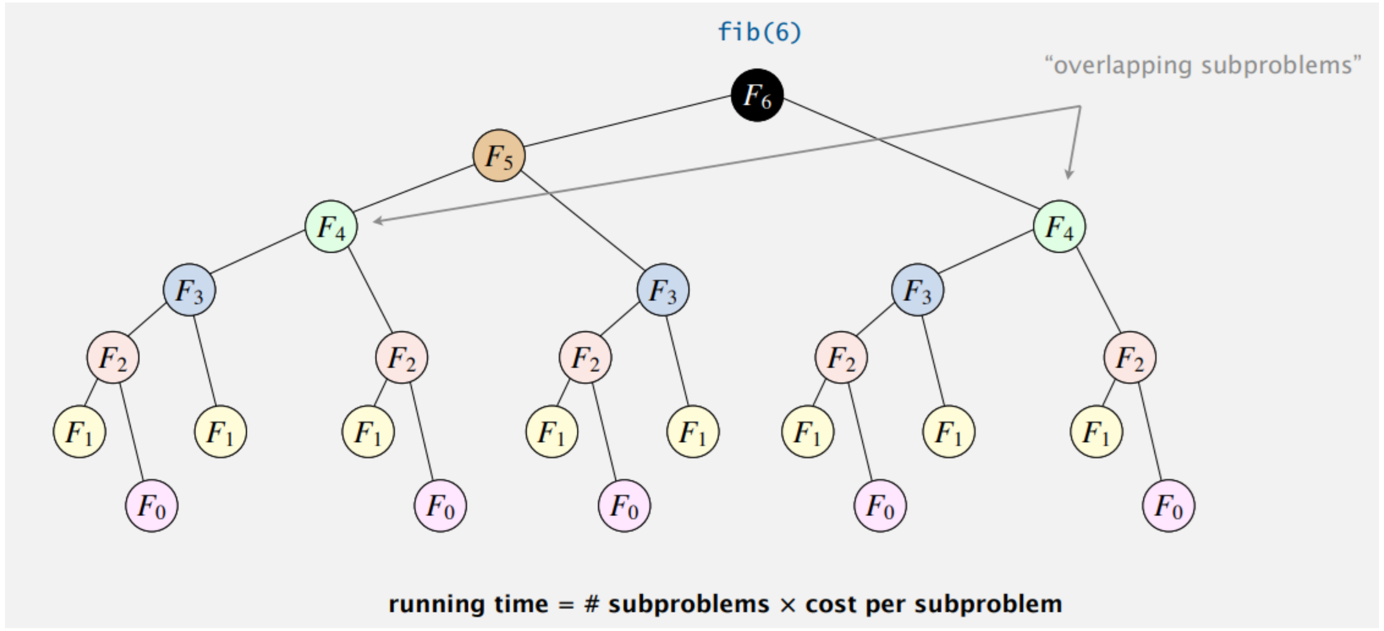
  D. Overflows a 64-bit long integer.

C. More than 1 hour. Computing **fib(80)** using the naive recursive algorithm would take an extremely long time, typically much more than one hour, due to the exponential increase in the number of recursive calls.

Fibonacci numbers: naïve recursive approach

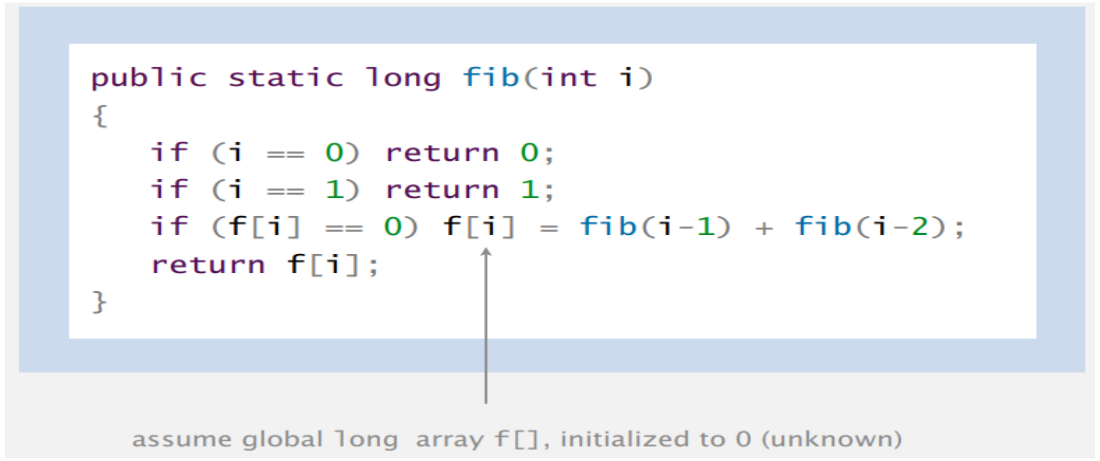
- **Exponential waste.** Same overlapping subproblems are solved repeatedly.

- Ex. To compute fib(6):
  - fib(5) is called 1 time.
  - fib(4) is called 2 times.
  - fib(3) is called 3 times.
  - fib(2) is called 5 times.
  - fib(1) is called = 8 times.


fib(6)
"overlapping subproblems"
F_6
F_5
F_4
F_3
F_2
F_1
F_0
running time = # subproblems × cost per subproblem

Fibonacci numbers: top-down dynamic programming

- Memoization.
  - Top-down dynamic programming is also known as **memoization** because it avoids duplicating work by remembering the results of function calls.
  - Maintain an array (or symbol table) to remember all computed values.
  - If value to compute is known, just return it; otherwise, compute it; remember it; and return it.



```
public static long fib(int i)
{
   if (i == 0) return 0;
   if (i == 1) return 1;
   if (f[i] == 0) f[i] = fib(i-1) + fib(i-2);
   return f[i];
}
```

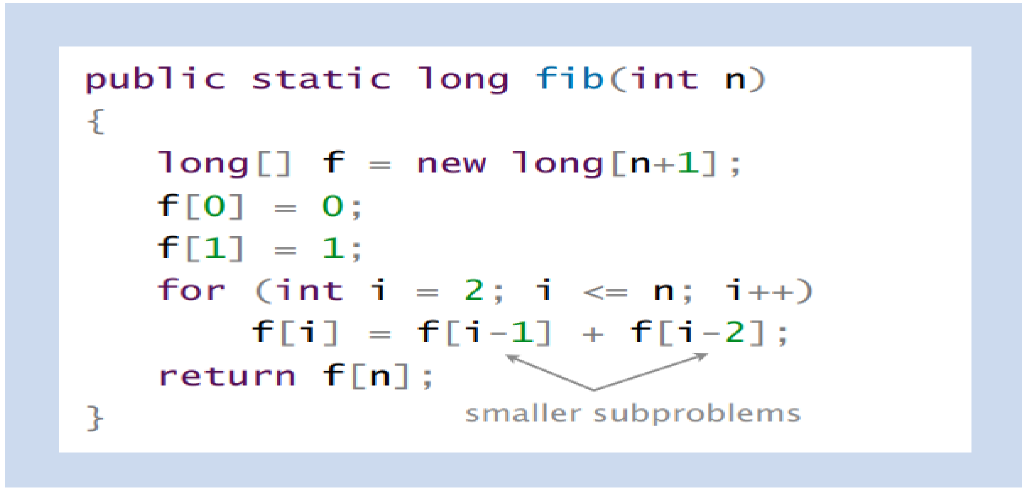
assume global long array f[], initialized to 0 (unknown)

Fibonacci numbers: Top-down dynamic programming

- Explanation of the code.
  - Declares static array f for caching. Assume global long array f[], initialized to 0.
  - Handles base cases for n = 0 and n = 1.
  - Computes and caches Fibonacci values recursively.
  - Returns cached value if already computed.
- Impact:
  - Solves each subproblem F_i only once; $\Theta(n)$ time to compute F_n.

Fibonacci numbers: Bottom-up dynamic programming

- Tabulation.
  - Build computation from the “bottom up.”
  - Solve small subproblems and save solutions.
  - Use those solutions to solve larger subproblems.



```
public static long fib(int n)
{
   long[] f = new long[n+1];
   f[0] = 0;
   f[1] = 1;
   for (int i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];
   return f[n];
}
```

Fibonacci numbers: Bottom-up dynamic programming

- Explanation of the code.
  - Creates array f of size n+1.
  - Sets f[0] = 0 and f[1] = 1.
  - Loops from 2 to n, calculating f[i].
  - Returns the value f[n].
- For example, to solve f[2] = f[1]+f[0], we already have solution of f[1] and f[0].
- Impact:
  - Solves each subproblem F_i only once; $\Theta(n)$ time to compute F_n. No recursion.
- When the ordering of the subproblems is clear, and space is available to store all the solutions, bottom-up dynamic programming is a very effective approach.

Fibonacci numbers: bottom-up dynamic programming

When the ordering of the subproblems is clear, and space is available to store all the solutions, bottom-up dynamic programming is a very effective approach.

Dynamic programming - Summary

- Divide a complex problem into a number of simpler overlapping subproblems.
  - [define n + 1 subproblems, where subproblem i is computing the ith Fibonacci number]
- Define a recurrence relation to solve larger subproblems from smaller subproblems.
  - [easy to solve subproblem i if we know solutions to subproblems i − 1 and i − 2]
- Store solutions to each of these subproblems, solving each subproblem only once.
  - [use an array, storing subproblem i in f[i]]
- Use stored solutions to solve the original problem.
  - [subproblem n is original problem]

Floyd-Warshall Algorithm

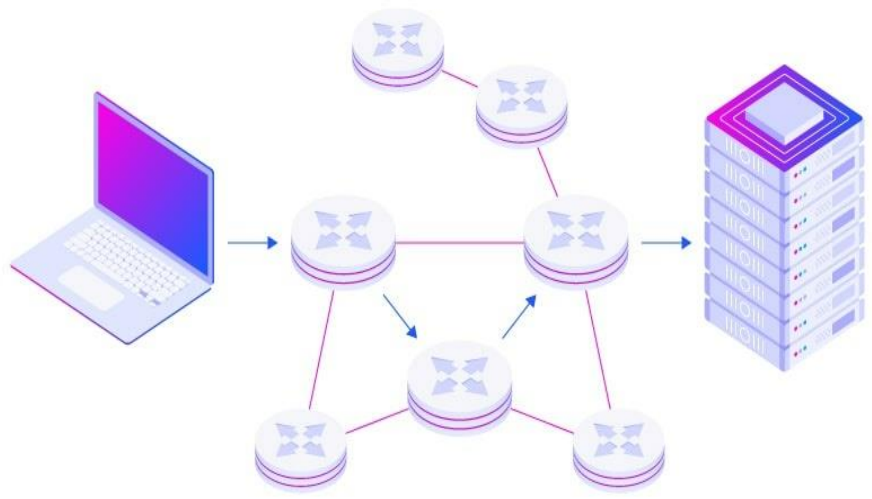
Introduction

- Floyd-Warshall algorithm is used for finding the shortest path between all the pairs of vertices in a weighted graph.
- This algorithm works for both the directed and undirected weighted graphs.
- Floyd-Warhshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.

Application- Network routing

- The Floyd-Warshall algorithm is used in computer networks to determine the shortest paths between all pairs of nodes.

- This is essential for efficient data transmission and routing packets through the network.

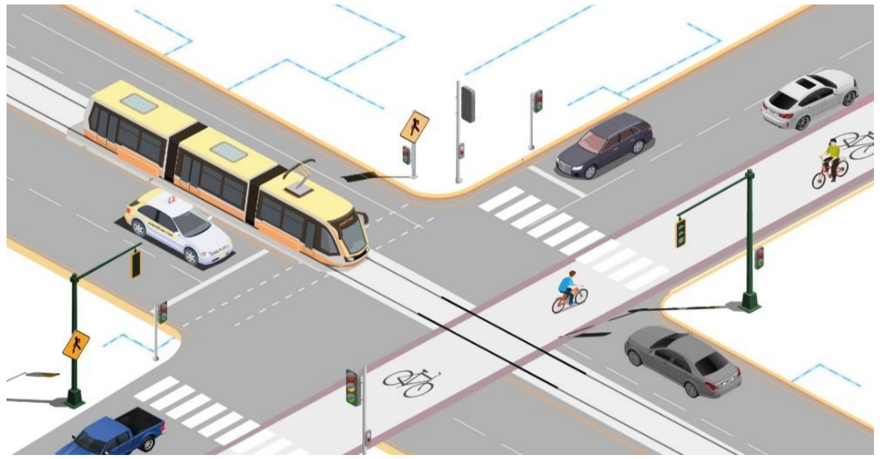


Application- Transportation and logistics

- The Floyd-Warshall algorithm can be used to find the shortest paths between all pairs of locations in transportation systems, such as road networks or airline routes,
- This helps in optimizing routes for vehicles or flights, minimizing travel time, and reducing fuel consumption.

Application- Traffic management

- Urban planners and traffic engineers use Floyd-Warshall to analyze traffic flow and optimize signal timings at intersections.
- By finding the shortest paths between different points in a city, they can develop strategies to alleviate congestion and improve overall traffic efficiency.



Application- Robotics

- Path planning algorithms in robotics often utilize the Floyd-Warshall algorithm to find the shortest paths between various points in a robot's environment.
- This is crucial for autonomous robots navigating complex terrains or environments with obstacles.

Floyd-Warshall- Advantages

- It helps to find the shortest path in a weighted graph with positive or negative edge weights.
- A single execution of the algorithm is sufficient to find the lengths of the shortest paths between all pairs of vertices.
- It is easy to modify the algorithm and use it to reconstruct the paths.

Floyd-Warshall- Disadvantages

- It can find the shortest path only when there are no negative cycles.
- It does not return the details of the paths.

Floyd-Warshall- Steps

Step 1 − Construct an adjacency matrix **A** with all the costs of edges present in the graph. If there is no path between two vertices, mark the value as ∞.

Step 2 − Derive another adjacency matrix $\mathbf{A_1}$ from **A** keeping the first row and first column of the original adjacency matrix intact in $\mathbf{A_1}$. And for the remaining values, say $\mathbf{A_1[i,j]}$,

if **A[i,j]>A[i,k]+A[k,j]** then replace $\mathbf{A_1[i,j]}$ with **A[i,k]+A[k,j]**.
Otherwise, do not change the values. Here, in this step, **k = 1** (first vertex acting as pivot).

Step 3 − Repeat **Step 2** for all the vertices in the graph by changing the **k** value for every pivot vertex until the final matrix is achieved.

Step 4 − The final adjacency matrix obtained is the final solution with all the shortest paths.

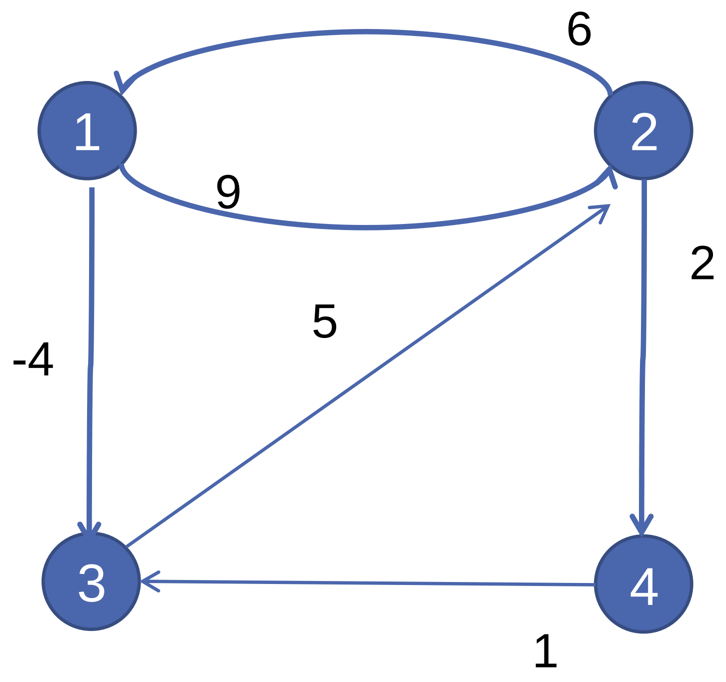
Floyd-Warshall- Pseudo code

```
Floyd-Warshall (dist)
  n = number of vertices in graph
  // Initialize distance matrix with original edge weights
  for i = 0 to n-1
    for j = 0 to n-1
      if i == j
       dist[i][j] = 0  // Distance from a vertex to itself is always 0
      else  if dist[i][j] == 0           //no direct edge
// Set to infinity if no direct edge (assuming 0 means no edge)
       dist[i][j] = infinity

 // Consider all vertices as intermediate vertices
for k = 0 to n-1
  for i = 0 to n-1
    for j = 0 to n-1
      if dist[i][j]> dist[i][k] + dist[k][j]
        dist[i][j] = dist[i][k] + dist[k][j]
 return dist
```

Example

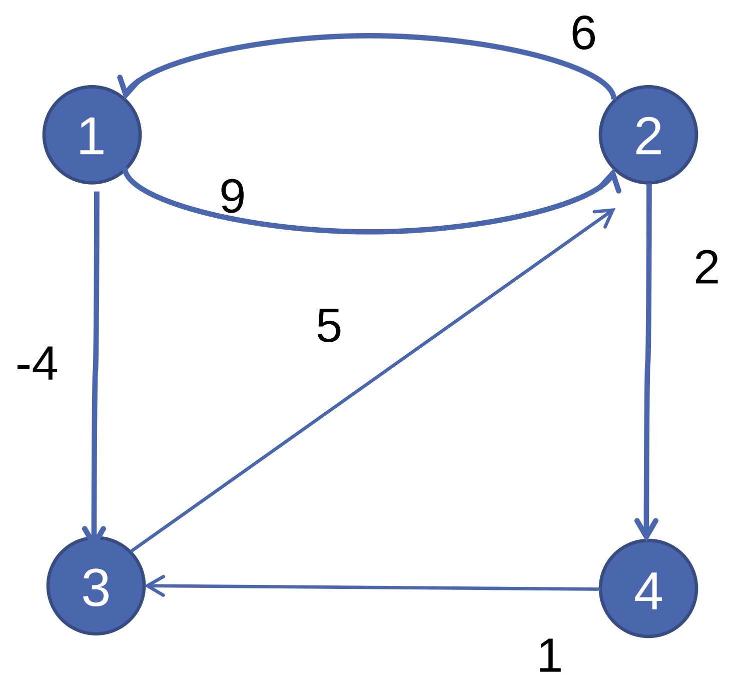
- Let the given graph be:



Create D0- Steps

- Create a matrix D^0 of dimension n*n (n is the number of vertices).
- The row and the column are represented by i and j respectively. (i and j are the vertices of the graph.)
- Each cell D0[i][j] is filled with the distance from the ith vertex to the jth vertex.
  - If $i=j$, set $D0[i][j]=0$ (Distance from any vertex to itself is 0).
  - If there is a direct edge from vertex *i* to vertex *j*, set $D0[i][j]$ to the weight of that edge.
  - If there is no direct path from vertex *i* to vertex *j*, set $D0[i][j]=\infty$

Create D0



Fill each cell with the distance between ith and jth vertex

$$D^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & \infty & 2 \\ 3 & \infty & 5 & 0 & \infty \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

Matrix D^0

Steps for creating- D1,D2,D3,D4

- The elements in the first column and the first row of Dk will remain the same as in Dk−1.
- The diagonal elements of Dk will be set to zero.

- Update the Remaining Elements:

Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex (1).

For each cell $Dk[i][j]$, update it using the formula:

If $Dk-1[i][j]>Dk-1[i][1]+Dk-1[1][j]$

then update $Dk[i][j]$ as:

$Dk[i][j]=Dk-1[i][1]+Dk-1[1][j]$

If the condition is not true , keep the element the same:

$Dk[i][j]=Dk-1[i][j]$

Create D1 from D0

- Keep the values of row1 and column1 as D0.
- Keep the value of diagonal to 0

$$D0=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & \infty & 2 \\ 3 & \infty & 5 & 0 & \infty \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

$$D1=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & & \\ 3 & \infty & & 0 & \\ 4 & \infty & & & 0 \end{array}$$

Create D1- D0[2,3]

For D0[2,3] through vertex 1, we have

D0[2,3]   D0[2,1]+D0[1,3]

Value of D0[2,3] = ∞

Value of D0[2,1]+D0[1,3], 6 + (-4)=2

Since   ∞> 2

Therefore, refill D0[2,3] =2

$$D0=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & \infty & 2 \\ 3 & \infty & 5 & 0 & \infty \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

$$D1=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & 2 & \\ 3 & \infty & & 0 & \\ 4 & \infty & & & 0 \end{array}$$

Create D1- D0[2,4]

For D0[2,4] through vertex 1, we have

D0[2,4] D0[2,1]+D0[1,4]

Value of D0[2,4]= 2

Value D0[2,1]+D0[1,4]=   6+ ∞

Thus D0[2,4] < D0[2,1]+D0[1,4]

i.e   2 < 6+ ∞

Therefore, keep the value of D0[2,4] same i.e no change
So D0[2,4]= 2

If the condition does not meet, keep the element the same.

$$D0=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & \infty & 2 \\ 3 & \infty & 5 & 0 & \infty \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

$$D1=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & 2 & 2 \\ 3 & \infty & & 0 & \\ 4 & \infty & & & 0 \end{array}$$

Create D1- D0[3,2]

For D0[3,2] through vertex 1, we have

D0[3,2] D0[3,1]+ D0[1,2]

Value of D0[3,2]=5

Value of D0[3,1]+D0[1,2]= ∞+9

Thus 5 < ∞+9

Therefore, keep the value of D0[3,2] same 5 i.e no change

Using this formula you can find remaining D0[3,4], D0[4,2],D0[4,3] etc

$$D0=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & \infty & 2 \\ 3 & \infty & 5 & 0 & \infty \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

$$D1=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & 2 & 2 \\ 3 & \infty & 5 & 0 & \\ 4 & \infty & & & 0 \end{array}$$

Final- D0 to D1

$$D0=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & \infty & 2 \\ 3 & \infty & 5 & 0 & \infty \\ 4 & \infty & \infty & 1 & 0 \end{array} \longrightarrow D1=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & 2 & 2 \\ 3 & \infty & 5 & 0 & \infty \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

Create D2 from D1

- D2 is created using D1.
- The elements in the second column and the second row will remain same as of D1.
- In this step, k is the second vertex (i.e. vertex 2).

$$D1=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & 2 & 2 \\ 3 & \infty & 5 & 0 & \infty \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

$$D2=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & & \\ 2 & 6 & 0 & 2 & 2 \\ 3 & & 5 & 0 & \\ 4 & & \infty & & 0 \end{array}$$

Create D2- D1[1,3]

For D1[1,3] through vertex 2, we have

D1[1,3]   D1[1,2]+D1[2,3]

Value of D1[1,3] = -4

Value of D1[1,2]+D1[2,3]= 9+2=11

Since -4 <11

Therefore, keep the value of D1[1,3] same

D1[1,3] =-4

$$D1= \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & 2 & 2 \\ 3 & \infty & 5 & 0 & \infty \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

$$D2= \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \\ 2 & 6 & 0 & 2 & 2 \\ 3 & & 5 & 0 & \\ 4 & & \infty & & 0 \end{array}$$

Create D2- D1[1,4]

For D1[1,4] through vertex 2

D1[1,4]   D1[1,2]+D1[2,4]

Value of D1[1,4] = ∞

Value of D1[1,2]+D1[2,4], 9+2= 11

Since ∞ > 9+2=11

Therefore, refill D1[1,4] =11

$$D1=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & 2 & 2 \\ 3 & \infty & 5 & 0 & \infty \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

$$D2=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & 11 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & & 5 & 0 & \\ 4 & & \infty & & 0 \end{array}$$

Create D2- D1[3,1]

For D1[3,1] through vertex 2

D1[3,1]   D1[3,2]+D1[2,1]

Value of D1[3,1] = ∞

Value of D1[3,2]+D1[2,1], 5+6= 11

Thus ∞ > 5+6=11

Therefore, refill D1[3,1] =11

Using this formula you can find remaining D1[3,4],D1[4,1] and D1[4,3].

$$D1=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & 2 & 2 \\ 3 & \infty & 5 & 0 & \infty \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

$$D2=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & 11 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & \\ 4 & & \infty & & 0 \end{array}$$

Final- D1 to D2

$$D1= \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & \infty \\ 2 & 6 & 0 & 2 & 2 \\ 3 & \infty & 5 & 0 & \infty \\ 4 & \infty & \infty & 1 & 0 \end{array} \longrightarrow D2= \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & 11 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & 7 \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

Create D3

- Keep the values of row3 and column3 as of D2.
- Keep the value of diagonal 0
- In this step, k is the third vertex (i.e. vertex 3).

$$D2= \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & 11 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & 7 \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

$$D3= \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & -4 & \\ 2 & & 0 & 2 & \\ 3 & 11 & 5 & 0 & 7 \\ 4 & & & 1 & 0 \end{array}$$

Create D3 - D2[1,2]

For D2[1,2] through vertex 3, we have

D2[1,2]   D2[1,3]+D2[3,2]

Value of D2[1,2] = 9

Value of D2[1,3]+D2[3,2] = -4+5=1

Thus 9> 1

Therefore, refill D2[1,2] =1

$$D2= \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & 11 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & 7 \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

$$D3= \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & -4 & \\ 2 & & 0 & 2 & \\ 3 & 11 & 5 & 0 & 7 \\ 4 & & & 1 & 0 \end{array}$$

Create D3- D2[1,4]

For D2[1,4] through vertex 3, we have

D2[1,4] D2[1,3]+D2[3,4]

Value of D2[1,4] = 11

Value of D2[1,3]+D2[3,4] = -4+7=3

Thus 11> -4+7=3

refill D2[1,4] =3

$$D2=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & 11 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & 7 \\ 4 & \infty & \infty & 1 & 0 \end{array}$$

$$D3=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & -4 & 3 \\ 2 & & 0 & 2 & \\ 3 & 11 & 5 & 0 & 7 \\ 4 & & & 1 & 0 \end{array}$$

Using this formula you can find remaining D2[2,1],D2[2,4] D2[4,1] and D2[4,2].

Final-D2 to D3

$$D2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 9 & -4 & 11 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & 7 \\ 4 & \infty & \infty & 1 & 0 \end{array} \longrightarrow D3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & -4 & 3 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & 7 \\ 4 & 12 & 6 & 1 & 0 \end{array}$$

Create D^4

- Keep the values of row4 and column4 as D3.
- Keep the value of diagonal 0

$$D3=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & -4 & 3 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & 7 \\ 4 & 12 & 6 & 1 & 0 \end{array}$$

$$D4=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & & 3 \\ 2 & & 0 & & 2 \\ 3 & & & 0 & 7 \\ 4 & 12 & 6 & 1 & 0 \end{array}$$

Create D4 -D3[1,2]

For D3[1,2] through vertex 4, we have

D3[1,2]   D3[1,4]+D3[4,2]

Value of D3[1,2] = 1

Value of D3[1,4]+D3[4,2] = 3+6=9

Hence, 1 < 3+6=9

Therefore, keep the value of D3[1,2] same

D3[1,2] =1

$$D3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & -4 & 3 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & 7 \\ 4 & 12 & 6 & 1 & 0 \end{array}$$

$$D4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & & 3 \\ 2 & & 0 & & 2 \\ 3 & & & 0 & 7 \\ 4 & 12 & 6 & 1 & 0 \end{array}$$

Create D4 -D3[1,3]

For D3[1,3] through vertex 3, we have

D3[1,3] D3[1,4]+D3[4,3]

Value of D3[1,3] = -4

Value of D3[1,4]+D3[4,3] = 3+1=4

Since -4< 3+1= 4

Therefore, keep the value of D1[1,3] same .

D3[1,3] =-4

Using this formula you can find remaining D3[2,1],D3[2,3] D3[3,1] and D3[3,2].

$$D3=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & -4 & 3 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & 7 \\ 4 & 12 & 6 & 1 & 0 \end{array}$$

$$D4=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & -4 & 3 \\ 2 & & 0 & & 2 \\ 3 & & & 0 & 7 \\ 4 & 12 & 6 & 1 & 0 \end{array}$$

Final-D3 to D4

$$D3=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & -4 & 3 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & 7 \\ 4 & 12 & 6 & 1 & 0 \end{array} \longrightarrow D4=\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & -4 & 3 \\ 2 & 6 & 0 & 2 & 2 \\ 3 & 11 & 5 & 0 & 7 \\ 4 & 12 & 6 & 1 & 0 \end{array}$$

D4 give us the shortest path

Time complexity

- The **time complexity** of the Floyd-Warshall algorithm is **O(n³)**, where 'n' is the number of vertices in the graph. There are three loops in the algorithm that iteratively update the shortest path distances between every pair of vertices. Since each of these loops runs n times, the total number of iterations is : $n \times n \times n = n^3$

  - **Outer Loop (k)**: This loop runs n times, iterating over each possible intermediate vertex.
  - **Middle Loop (i)**: For each iteration of the outer loop, this loop runs n times, iterating over each source vertex.
  - **Inner Loop (j)**: For each iteration of the middle loop, this loop runs n times, iterating over each destination vertex.

- The **space complexity** of the algorithm is **O(n²)**.