Algorithm Design Techniques: Dynamic Programming-1

“Those who cannot remember the past are condemned to repeat it ”

Reminder: Algorithm Design Methods

- Brute Force
- Greedy
- Divide and Conquer
- Dynamic Programming (DP)
- Backtracking
- ....

- Dynamic Programming is a powerful technique that can be used to solve many optimization problems in polynomial time for which a naive approach would take exponential time.

Warning :

Dynamic programming → planning over time , not computer programming

Basic Approach

- A computational scenario :There are re-occurring subproblems which in turn have their own smaller subproblems.
- Instead of trying to solve those re-appearing subproblems, again and again, dynamic programming suggests solving each of the smaller subproblems only once.

→Record the results in a table from which a solution to the original problem can be obtained.

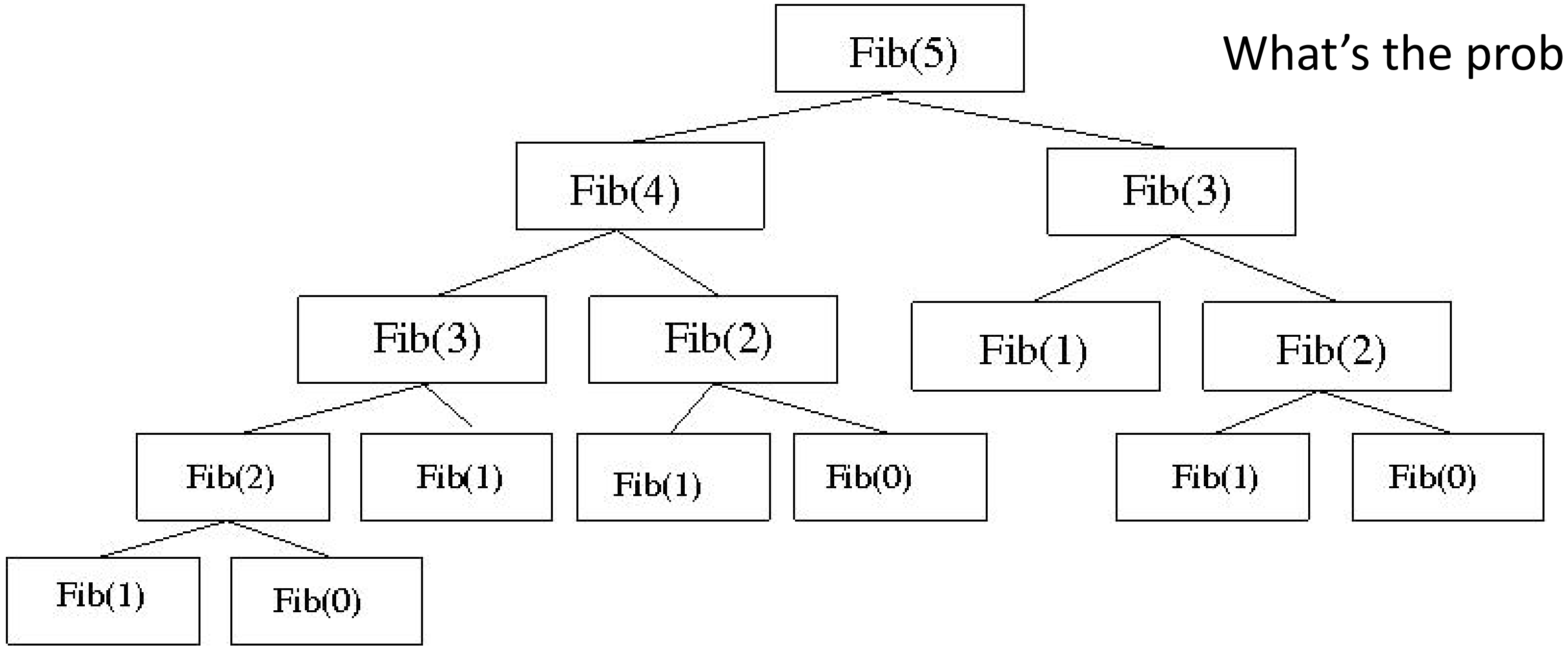
Dynamic Programming Solution Methods

- There are the following two main different ways to solve the problem:
  - Top-down: You start from the top, solving the problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. This is referred to as "Memoization."
  - Bottom-up: You directly start solving the smaller subproblems making your way to the top to derive the final solution of that one big problem. In this process, it is guaranteed that the subproblems are solved before solving the main problem. This can be called Tabulation (table-filling algorithm).
- In reference to iteration vs recursion, bottom-up uses iteration and the top-down uses recursion.

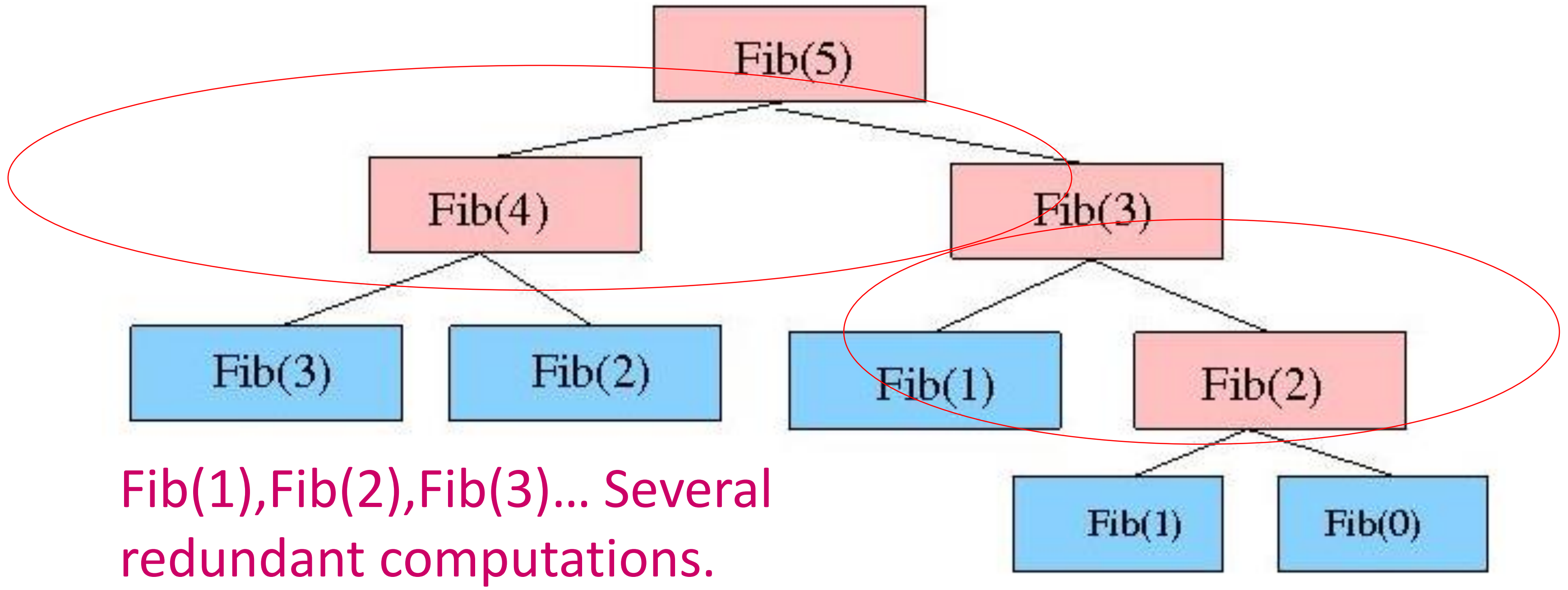
Introduction to Dynamic Programming

- We have looked at several algorithms that involve recursion.
- In some situations, these algorithms solve fairly difficult problems efficiently
  - *BUT* in other cases they are inefficient because they recalculate certain function values many times.
- One example of this is the Recursive Fibonacci algorithm.
- Remember the problem of recursive Fibonacci:
  - *Lots and lots* of calls to Fib(i-1) and Fib(i-2).....are made for i>2
- How to find for example Fib(10) more efficiently ?

Reminder : Fibonbacci Tree



What’s the problem?



Fib(1),Fib(2),Fib(3)… Several redundant computations.

Dynamic Programming : Efficient Fibonacci Function

- First store the values when computed :
  - $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$, $F_8 = 21$, $F_9 = 34$, $F_{10} = 55$…
  - Calculate F_3 by adding F_1 and F_2, then F_4, by adding F_3 and F_4, etc.
- This way, we can calculate any Fibonacci number by simply adding stored values.
- The basic idea of dynamic programming is to avoid making redundant method calls.
  - Instead, store the answers to all necessary method calls in memory and simply look these up whenever needed.
  - This is called Memoization (forming a memo)

  $\rightarrow$ Store the solution of these sub-problems so that we do not have to solve them over and over again

Fibonacci with Memoization

```
//Solution by memoization
 int memo[] = empty
 fib(n) {
   if(memo[n] != empty)
   return memo[n];       //Already solved,return the answer.
   if(n==1 || n==2)
       return 1;
   memo[n] = fib(n-1)+fib(n-2);  //Only fib(n-1) has to be computed !
   return memo[n];
}
```

Fibonacci with Tabulation

```
//Input integer n,output nth fib.number
//Previous values are stored in the array fib[]:Tabulation
fibon(n)
   int fib[n+1]    //Declare the array(Table)
   fib[0] = 0; fib[1] = 1
   for i = 2 to n  do
      fib[i] = fib[i - 1] + fib[i - 2]
   return fib[n]
```

Complexities:

Time Complexity: O(n) , Space Complexity : O(n)

Note that the complexity is just linear, not exponential !

```
//Fibonacci Series using Dynamic Programming
#include<iostream>
using namespace std;
int fib(int n)
{  // Declare an array to store Fibonacci numbers.
  int f[n+1];   int i;
   // 0 th and 1st number of the series are 0 and 1
  f[0] = 0;   f[1] = 1;
  for (i = 2; i <= n; i++)
     // Add the previous 2 numbers in the series and store it
      f[i] = f[i-1] + f[i-2];
  return f[n];
}
 int main ()
{
  int n = 18;
  cout<<"n =  "<< n<<"   "<<"fib n  = "<<fib(n)<<endl;
  return 0;
}
//n =  18   fib n  = 2584
```

Dynamic Programming

- DP is essentially an algorithm design technique for optimization problems: Often minimizing or maximizing.
- Like divide and conquer, DP solves problems by combining solutions to independent subproblems.
- However in DP, subproblems don’t have to be independent.
    - Subproblems may share subproblems
    - However, solution to one subproblem may not affect the solutions to other subproblems of the same problem.

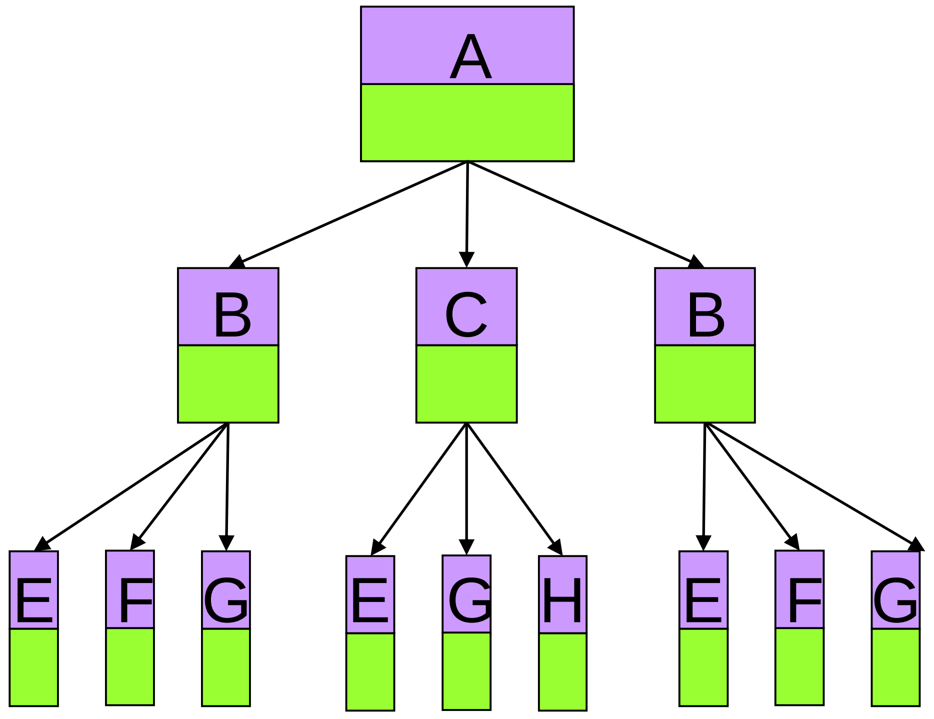
Dynamic Programming : How it Works ?

- Memoization: Store the solution of the subproblems so that we do not have to solve them again and again.
- Dynamic programming and memoization work together. Most of the problems are solved with two components of dynamic programming :
    - Recursion – Solve the subproblems recursively
    - Memoization – Store the solution of these subproblems so that we do not have to solve them again

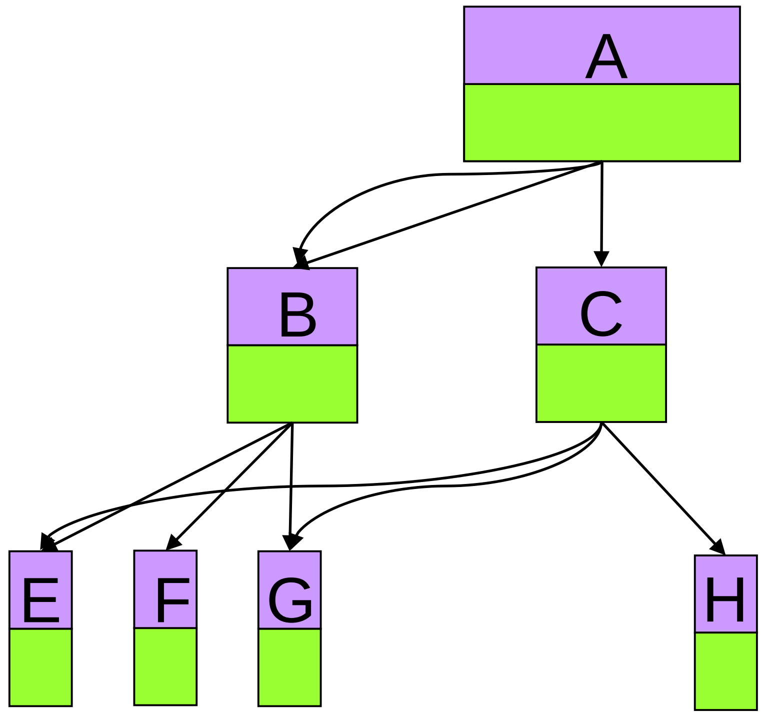
→ DP reduces computations bottom up or top down by

- Solving subproblems
- Storing solution to a subproblem the first time it is solved.
- Looking up the solution when subproblem is encountered again.

Dynamic Programming Solution



Several redundant computations



Actual computations with DP: No redundant repetitions.

Properties of Dynamic Programming

- Suppose we have a problem that we can break it to smaller subproblems having the following properties:
  - Optimal Substructure
    - An optimal solution to a problem contains within it an optimal solution to subproblems.
    - Optimal solution to the entire problem is built from optimal solutions to subproblems.
  - Overlapping Subproblems
    - If a recursive algorithm revisits the same subproblems over and over $\Rightarrow$ the problem has overlapping subproblems
- If the problem has these two properties, then we can attempt to solve that problem using DP.

Steps in Dynamic Programming

1. Characterize the structure of an optimal solution.
2. Define the value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up by storing in a table.
4. Construct an optimal solution from computed values.

Dynamic Programming and Brute Force

Dynamic programming is considered to be a "Careful brute force"

Example:
Dynamic Programming : You want 5 things. You search for them in 100 shops. You save names and locations of the shops and the items you can get from there. If you need those 5 items later, you don't search every time in 100 shops. You check the saved locations and go only to those specific shops
Brute Force : You search 100 shops every time you need the same 5 items.

Dynamic programming and Divide and Conquer

- Comparing to divide-and-conquer:
    - Both partition the problem into sub-problems
    - Divide-and-conquer partitions problems into independent sub-problems.
    - Dynamic programming is applicable when the sub-problems are dependent.

Dynamic programming and Greedy Method

- Comparing to the greedy method:
  - Both are applicable when a problem exhibits optimal substructure
  - In greedy algorithms, we use optimal substructure in a top-down fashion, but, without finding optimal solutions to sub-problems.
  - We first make a choice that looks best at the time-and then solve a resulting sub-problem.
  - In dynamic programming, we use optimal substructure in a bottom-up fashion:
    - We first find optimal solutions to sub-problems and, having solved the sub-problems, we find an optimal solution to the problem.

Illustrative Example : Binomial Coefficients

We know that :

$(x + y)^2 = x^2 + 2xy + y^2$, coefficients : $\underline{1,2}$,1

$(x + y)^3 = x^3 + 3x^2y + 3xy^2 + y^3$, coefficients : 1,$\underline{3,3}$,1

$(x + y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4$, coefficients :1,$\underline{4,6}$,4,1

$(x + y)^5 = x^5 + 5x^4y + 10x^3y^2 + 10x^2y^3 + 5xy^4 + y^5$,coefficients:1,5,10,10,5,1

…………………………………………………………………………… …………………

→The n+1 coefficients can be computed for $(x + y)^n$ using

$C(n, k) = n! / (k! * (n – k)!)$ for each of $k = 0…n$

- The repeated computations of all the factorials is expensive and unnecessary.

Computation of C(n,k) can be simplified .

Dynamic programming solution:

Save the computed values as we go from 1 to n and use the saved values to compute coefficients of the next step.

Simplifying Computations of C(n,k)

- How efficient is to use C(n,k) ?
- This formula is not used for computation because even for small values of n, the values of n! get really large.
- Instead, C(n,k) can be computed by the following definition:

$C(n,k)=C(n-1, k-1)+C(n-1, k)$

$C(n,0)= 1$

$C(n,n)=1$

Recursive Solution for Combination Formula

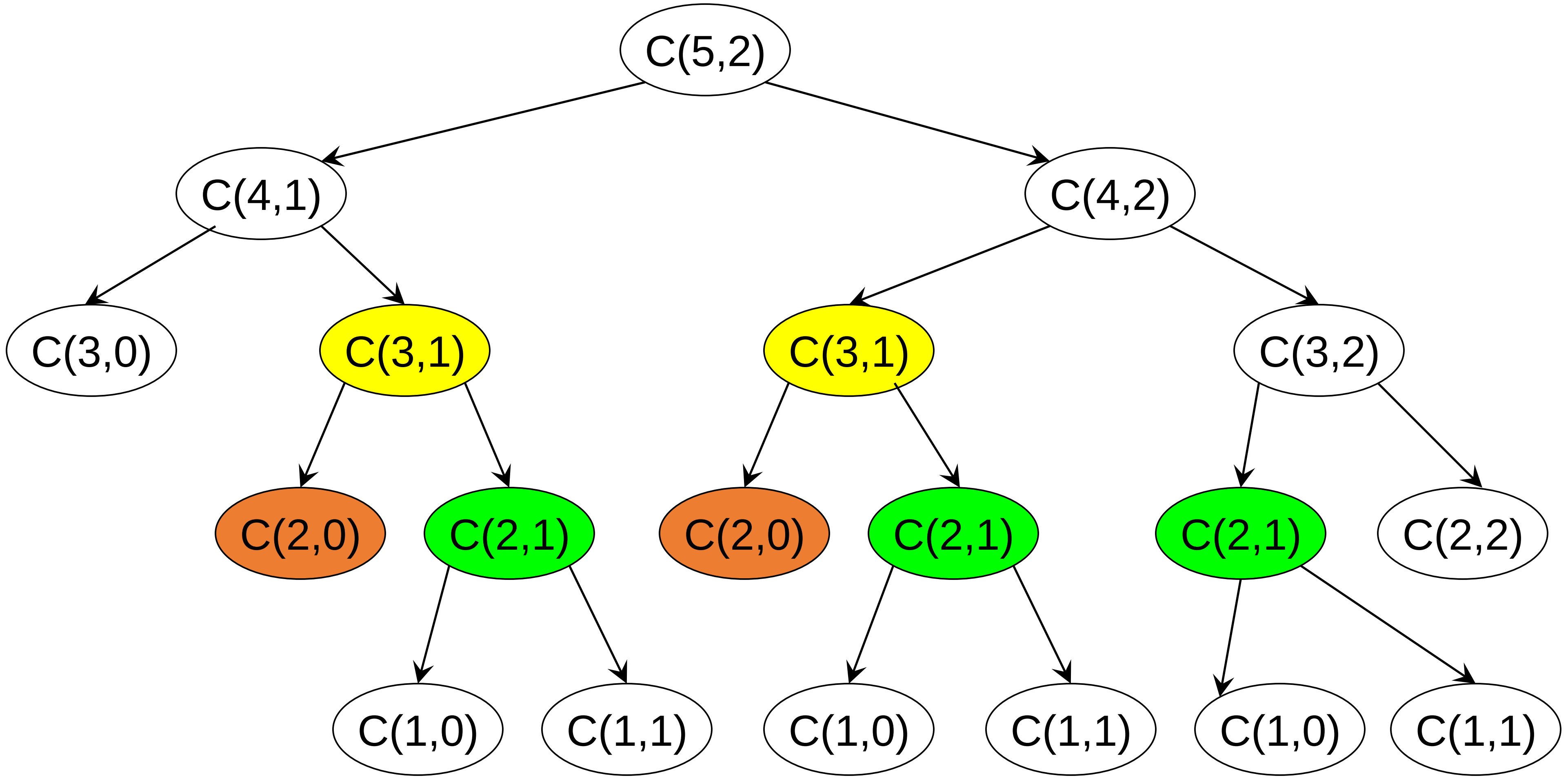
```
long C (int n, int k) {
  if ((k==0) || (k==n))
          return 1;
  else
          return C(n - 1, k) + C(n - 1, k - 1);
}
```

What about complexity ?

Several repeated computations as in Fibonacci!

Binomial Coefficients – RecursionTree for C(5,2)

Notice repeated computations in color. Complexity is exponential !



Solution by Dynamic Programming

Consider the following table result[i][j] of coefficients :

n	c(n,0)	c(n,1)	c(n,2)	c(n,3)	c(n,4)	c(n,5)	c(n,6)
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

Each row depends only on the preceding row

This table is known as Pascal’s Triangle

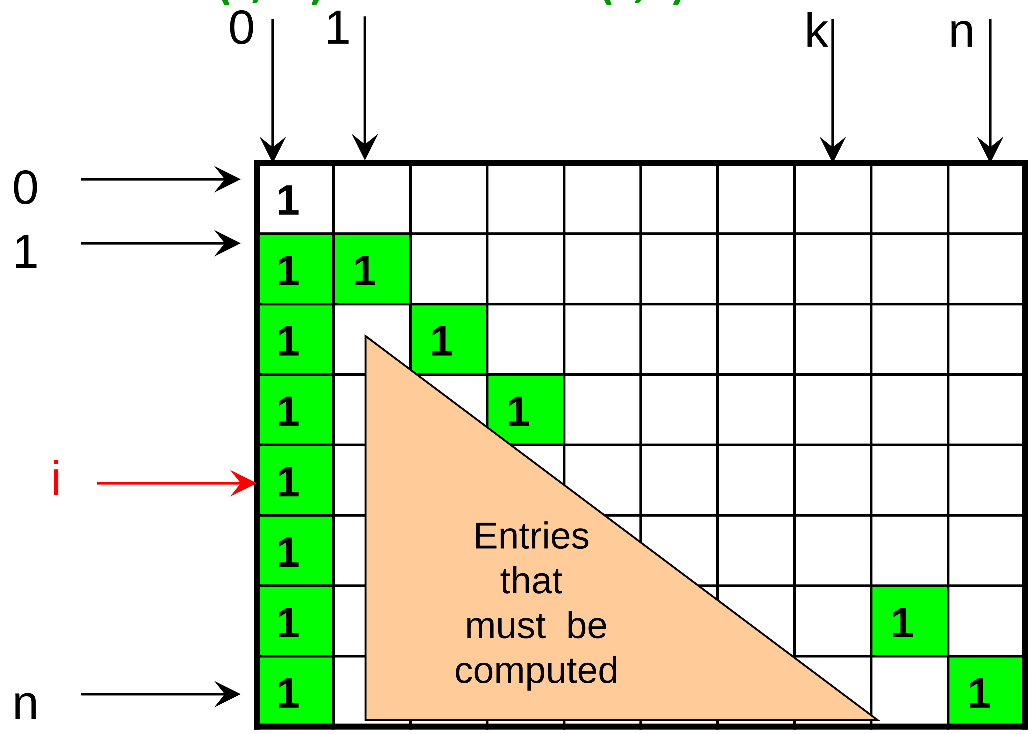
Solution by Dynamic Programming

- We want to eliminate recursion.
- We look at the recursion tree to see in which order the elements appear in the table.
- If we figure out the order, we can replace the recursion with an iterative loop that intentionally fills the table in the right order
- This technique is Dynamic Programming.

Binomial Coefficients – Table Filling Order

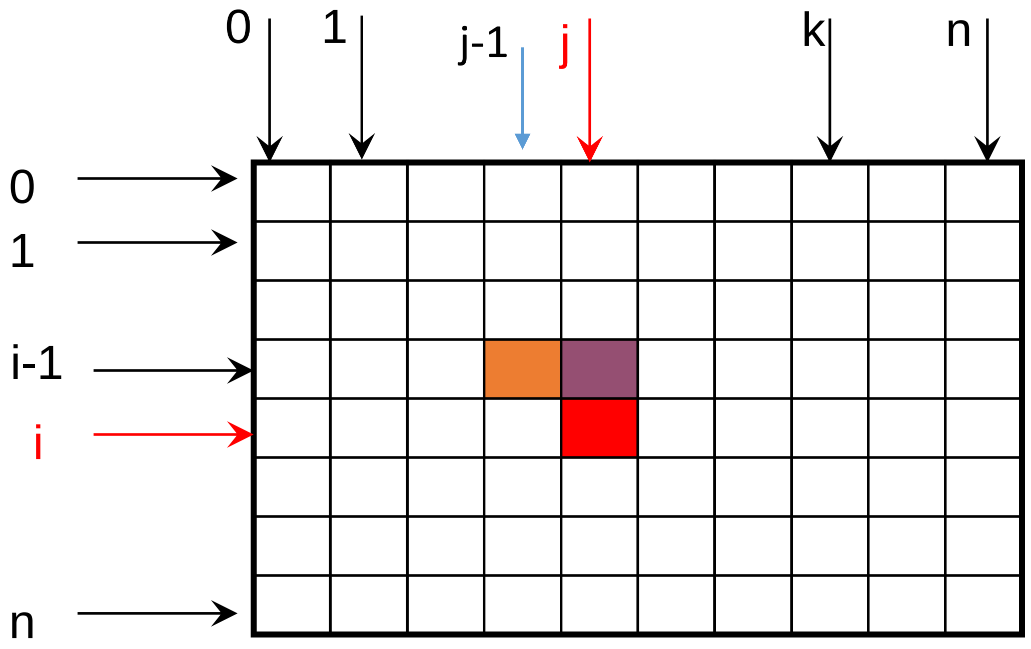
- result[i][j] stores the value of C(i,j)
- Table has n+1 rows and k+1 columns and k<=n.

  →We have to fill the lower triangle of the array.
- Initialization: C(i,0)=1 and C(i,i)=1 for i=1 to n



Binomial Coefficients – Table Filling Order

- result[i][j] stores the value of C(i,j)
- Rest of entries (i,j), for i=2 to n and j = 1 to i-1 are computed using entry (i-1, j-1) and (i-1, j)



C(n,k):Dynamic Programming Solution

```
//Computes coefficient C(n,k) in the range.
long C(int n, int k) {
  long result  [n + 1][n + 1];
  int i, j;
  for (i=0; i<=n; i++) {
      result[i][0]=1;
      result[i][i]=1;
      }
  for (i=2; i<=n; i++)
     for(j=1; j<i; j++)
       result[i][j]=result[i-1][j-1]+result[i-1][j];

  return result[n][k];
}
//C(4,3)→4
```

Complexity:

Time: O(n*n)   (Lower triagle)

Space: O(n*n)

DP Applications: Shortest Path Problems

- Optimal Substructure (Principle of optimality): Suppose that in solving a problem, we have to make a sequence of decisions $D_1, D_2, \ldots, D_n$. If this sequence is optimal, then the last k decisions, $1 < k < n$ must also be optimal.
- Example: The shortest path problem

  If $i, i_1, i_2, \ldots, j$ is a shortest path from i to j, then $i_1, i_2, \ldots, j$ must be a shortest path from i_1 to j

  $\rightarrow$If a problem can be described by a weighted graph, then shortest path problem can be solved by dynamic programming.

All Pairs Shortest Paths

- Given a weighted graph, we want to know the shortest path from any vertex in the graph to all others.
  - The Floyd-Warshall algorithm determines the shortest paths between all pairs of vertices in a graph.

  All Pairs Shortest Paths:
  - Aims to compute the shortest path from each vertex to every other vertex.
  - It Uses Dynamic Programming methodology to solve the problem.

All Pairs Shortest Paths

- Remember that Djkstra's algorithm is used to find shortest path lengths and paths from a single source to all other vertices.
- Can we use Djkstra to solve all pairs shortest paths problem ?
- If |V| is the number of vertices, Dijkstra's runs in ~ $\Theta(|V|^2)$
  - To solve all sources shortest paths problem , we could just call Dijkstra |V| times, passing a different source vertex each time . In that case the complexity is :

    $\Theta(|V| \times |V|^2)$ = ~$\Theta(|V|^3)$

    - Dijkstra's doesn't work with negative-weight edges.
    - Can we reduce the complexity or simplify the solution?

    → Yes,use dynamic programming.

All Pairs Shortest Paths : Floyd-Warshall Algorithm

- *The problem:* find the shortest path between every pair of vertices of a graph

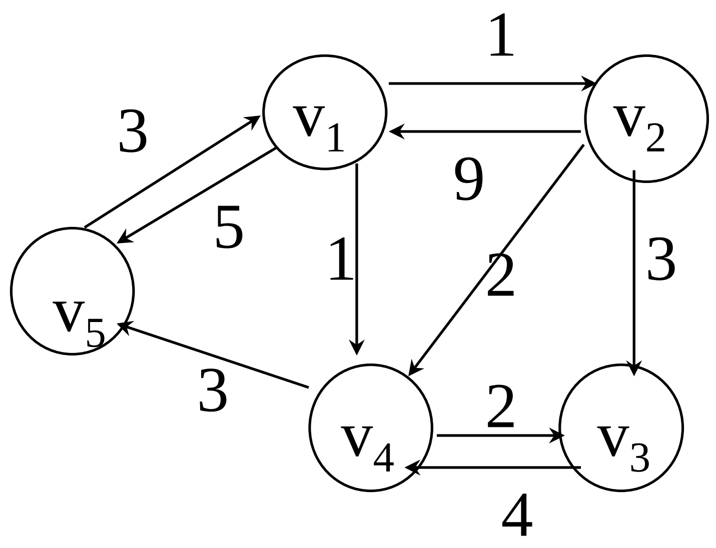
- *The graph* may contain negative edges but no negative cycles(Cycles whose edges sum to a negative value)
  Adjacency matrix representation: Define a weight matrix W:

  W(i,j)= weight of edge (i,j)
  W(i,j)=0 if i=j.
  W(i,j)=∞ if there is no(direct) edge between i and j.

Example : Weight Matrix Representation

$w_{i,j}$: Distance or cost of (i,j), directed graph

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



DP Formulation : The Subproblems

- How can we define the shortest distance $d_{i,j}$ in terms of “smaller” overlapping subproblems?
- One way is to restrict the paths to only include vertices from a restricted subset.
- Initially, the subset is empty. Then, it is incrementally increased until it includes all the vertices.

The subproblems

- Let $D^{(k)}[i,j]$=The weight of a shortest path from v_i to v_j that uses only vertices from $\{v_1,v_2,\dots,v_k\}$ as intermediate vertices in the path.
  - $D^{(0)}$= *W, initial distances*
  - $D^{(k)}$ = Distances at *Intermediary step k*
  - $D^{(n)}$= *T*he goal matrix. Final shortest distances
- How do we compute $D^{(k)}$ from $D^{(k-1)}$ …. ?

→We need to compute : $D^{(0)}$, $D^{(1)}$,……, $D^{(n)}$

These are the solutions to subproblems.

Note that subproblems are overlapping !

The Recursive Definition

- Suppose k is a new node. Is it part of the shortest path ? There are two possibilities:

1. $D^{(k)}[i,j] = D^{(k-1)}[i,j]$ (k has no effect,not part of the min.path)
2. $D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j]$ (k is part of the min.path)

We conclude that the shortest path , including k, must be smaller of 1 or 2:

$$D^{(k)}[i,j] = \min\{ D^{(k-1)}[i,j],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j] \}$$

Floyd-Warshall Algorithm : Explanation

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j with all intermediate vertices in the set {1,2,…,k}. A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases}$$

Floyd-Warshall Algorithm: Explanation

For al k in 0…n ,we have:

$d_{ij}^{(0)} = w_{ij}$ if k=0

(Weight matrix,no intermediate vertices at all)

$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ if $k \geq 1$

(Intermediary steps)

Result: $D^{(n)} = (d_{ij}^{(n)})$

→The matrix $D^{(n)}=(d_{ij}^{(n)})$ gives the final answer:

$d_{ij}^{(n)}=$ Shortest distances for all i,j pairs in V .

Because all intermediate vertices are in the set {1,2,…,n}.

Floyd-Warshall Algorithm : Explanation

Why it is a DP solution ?

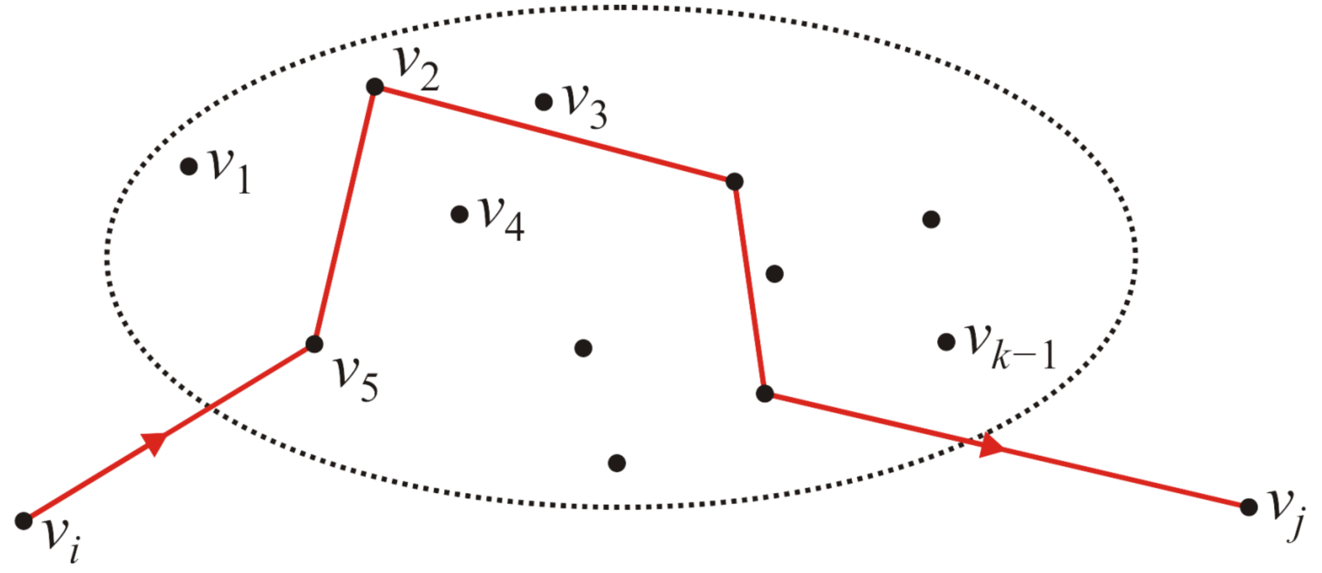
- It Uses optimal substructure of shortest paths: Any subpath of a shortest path is also a shortest path.
- Subproblems are overlapped:Next subpath is built upon the previous subpaths .
- Solution method: Create a 3-dimensional table: (i,j,k)
    - Let $d_{ij}^{(k)}$ –shortest path weight of any path from i to j where all intermediate vertices are from the set {1,2, …, k}, k: 0…n
    - As final solution, we would like to know the values of $d_{ij}^{(n)}$.

The General Step

Define $d_{i,j}^{(k-1)}$ as the shortest distance, but only allowing intermediate visits to vertices in the set: $v_1, v_2, \ldots, v_{k-1}$

- Suppose we have an algorithm that has found these values for all pairs

Can we do better by including v_k now?

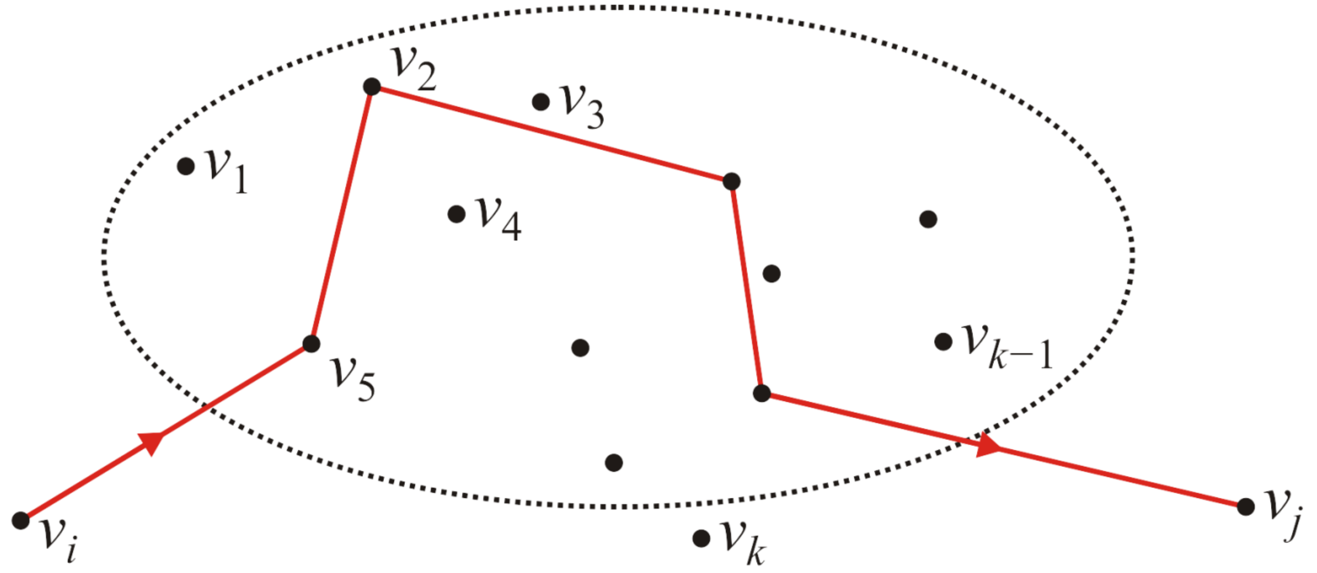


The General Step

How could we find $d_{i,j}^{(k)}$; that is, the shortest path allowing intermediate visits to vertices $v_1, v_2, \ldots, v_{k-1}$?

→We have all shortest paths up to k-1.

can we do better by including v_k now?



The General Step

Does v_k change previous shortest path ?

With v_1, v_2, …, v_{k-1} as intermediates, the shortest paths from v_i to v_j, will include the paths from v_i to v_k and v_k to v_j

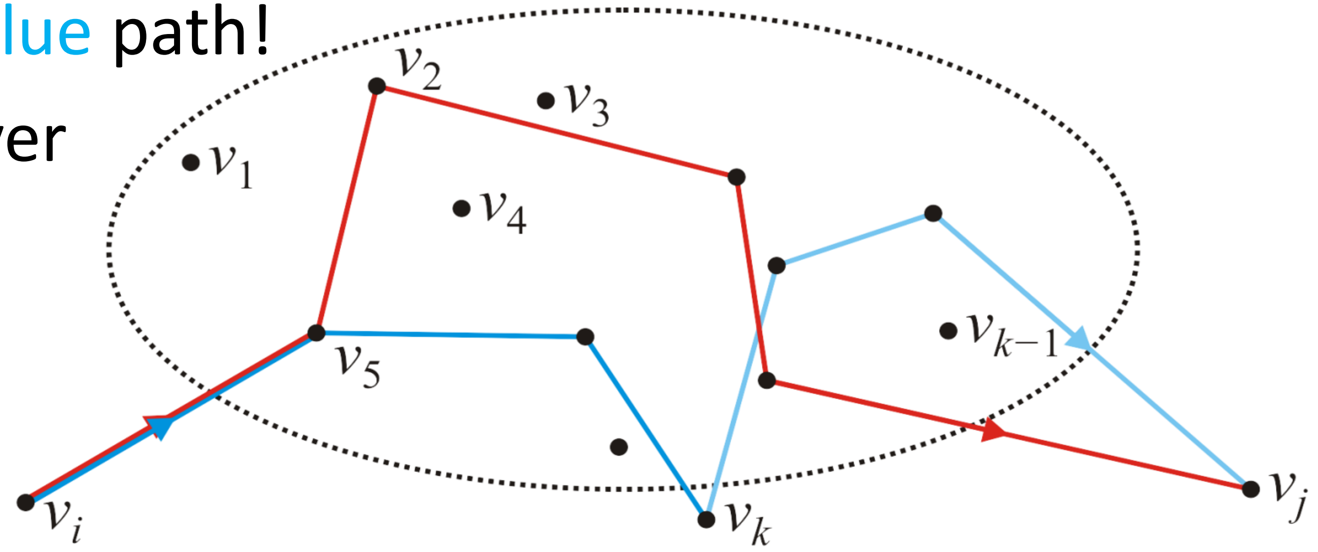
- The only possible shorter path including v_k would be the path from v_i to v_k *and* continuing from there to v_j

Thus, we calculate and choose: $d_{i,j}^{(k)} = \min\left\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right\}$

→Choose red or blue path!

Of course,whichever

is smaller !



The Stages of Floyd-Warshall Algorithm

Matrix based solution :

Solve the problem stage by stage: Find

$D^{(0)}$

$D^{(1)}$

$D^{(2)}$

…

$D^{(n)}$

$D^{(k)}$ contains the shortest paths with all the intermediate vertices from the set $\{1,2\ldots,k\}$.

Why is this a DP solution ?

Each D $^{(i)}$ is stored and used in the next step.

Floyd-Warshal Algorithm :All Pairs Shortest Distances

// At each step, d_{ij} is the (cost of the) shortest path
// from *i* to *j* using intermediate vertices (1..*k*-1).

```
FLOYD-WARSHALL(W, n)
D^(0) ← W
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            do $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$
```

- Time complexity: $O(n^3) = O(|V|^3)$
- Space complexity: $O(n^3)$

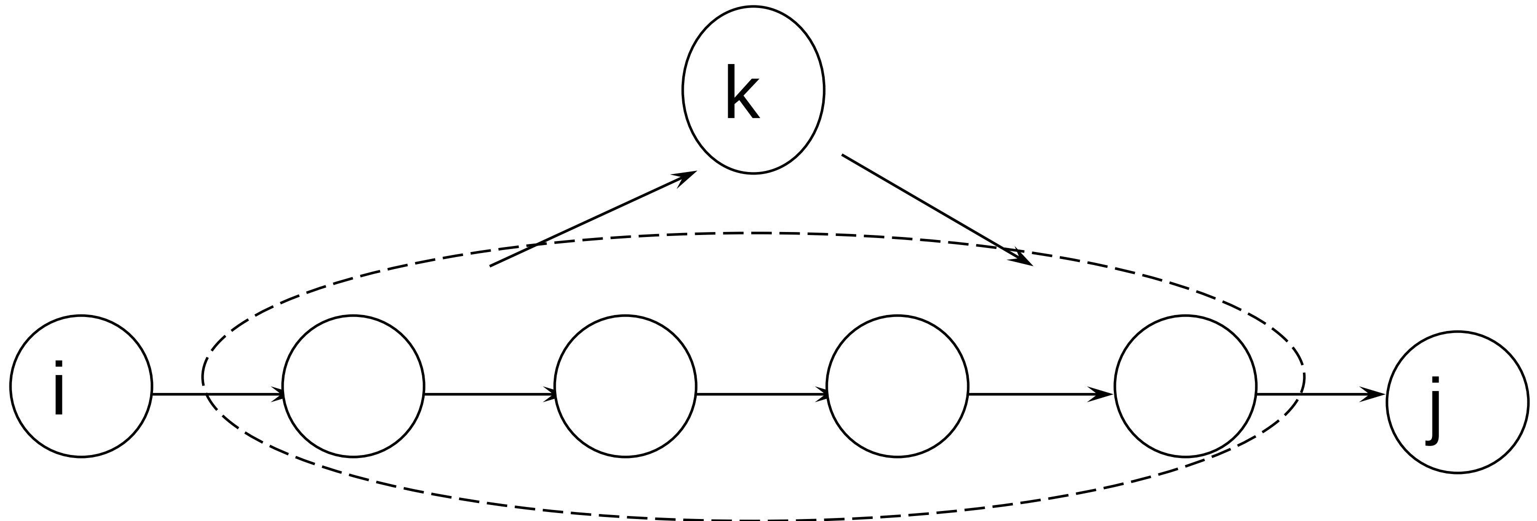
Floyd-Warshall Algorithm:Part of a C++ Function

```
double d[num_vertices][num_vertices];
// Initialize the matrix
……………………………………………
for ( int k = 0; k < num_vertices; ++k ) {
    for ( int i = 0; i < num_vertices; ++i ) {
        for ( int j = 0; j < num_vertices; ++j ) {
            d[i][j] = min( d[i][j], d[i][k] + d[k][j] );
        }
    }
}
```

Tracing Floyd Warshall Algorithm

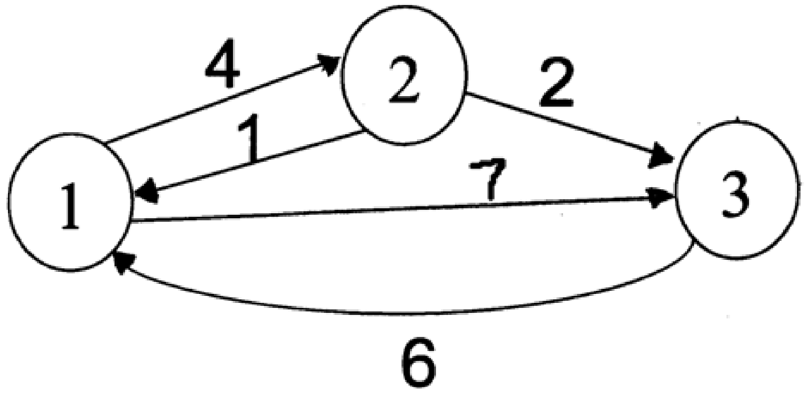
- Let the vertices in a graph be numbered from 1 … n.
- Consider the subset {1,2,…, k} of these n vertices.
- Imagine finding the shortest path from vertex i to vertex j that uses vertices in the set {1,2,…,k} only.
- There are two situations:
  1) k is an intermediate vertex on the shortest path.
  2) k is not an intermediate vertex on the shortest path.

  To decide, we need to perform distance comparisons for each new k.



Tracing Floyd Warshall Algorithm : Example1

$$D^{(0)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & \infty & 0 \end{bmatrix}$$ =W (Original weights).



$$D^{(1)} = \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

k = Vertex 1:
D(3,2) = D(3,1) + D(1,2)
10= 6+4 < ∞

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)})$$

$$D^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

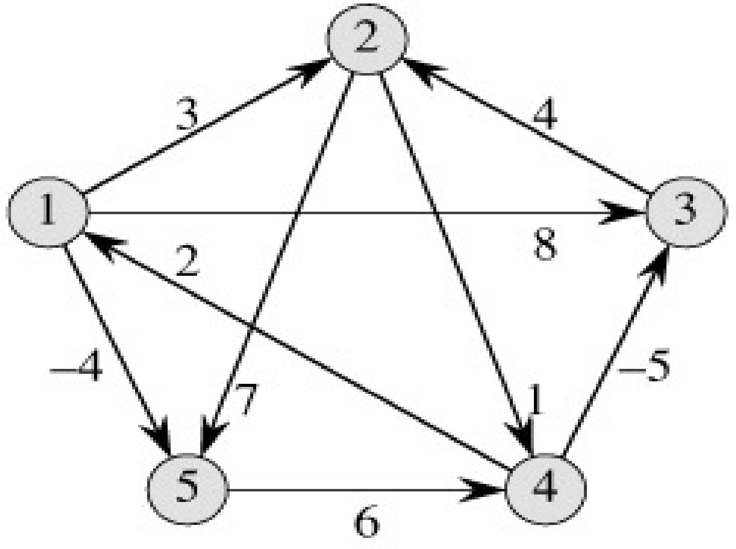
k=Vertex 2:
D(1,3) = D(1,2) + D(2,3)
6=4+2 < 7

$$D^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix}$$

k=Vertex 3:
Nothing changes
→Min distances.

Tracing Floyd Warshall Algorithm : Example2

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \, , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \, . \end{cases}$$

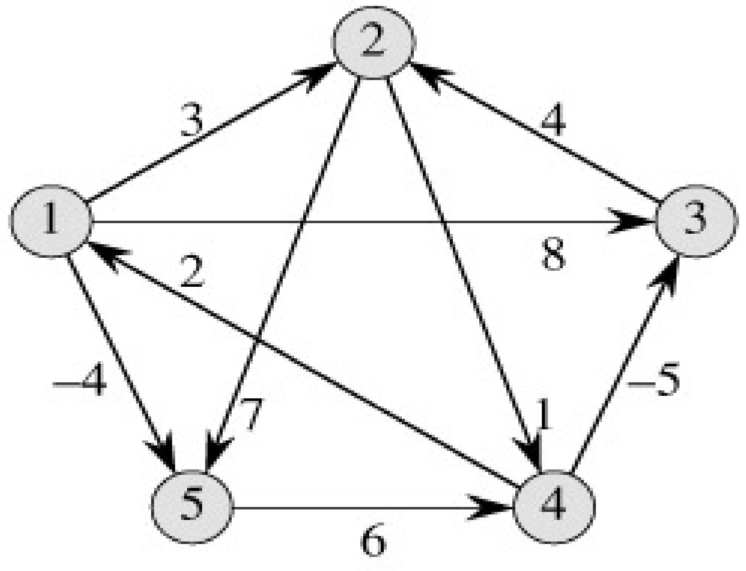


<u>k=0</u>

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Step 1

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 , \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 . \end{cases}$$



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

<u>k=1</u>

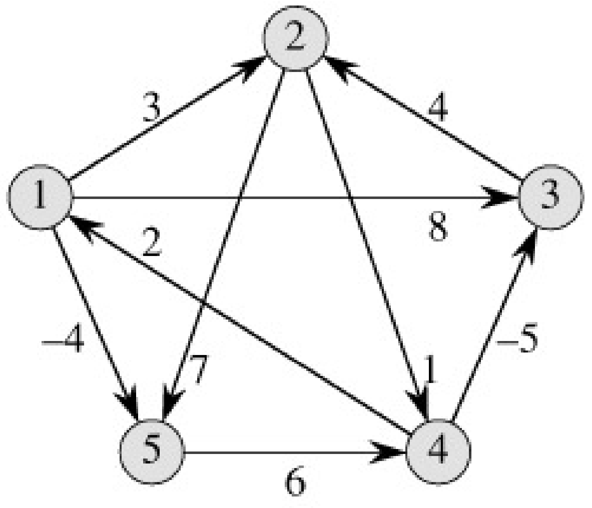
$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \boxed{5} & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$5=\min(\infty ,2+3)$

$-2=\min(\infty,2+(-4))$

Step 2

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$



$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

<u>k=2</u>

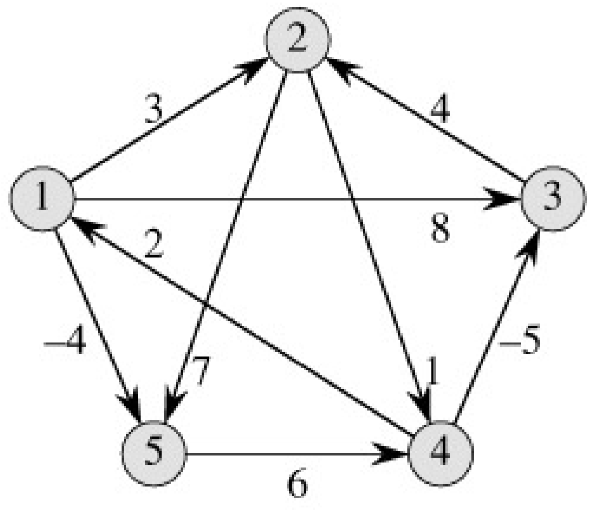
$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \boxed{5} & \boxed{11} \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$5=\min(\infty\ ,4+1)$
$11=\min(\infty\ ,4+7)$

$D^2 = D^1 W$....

Step 3

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$



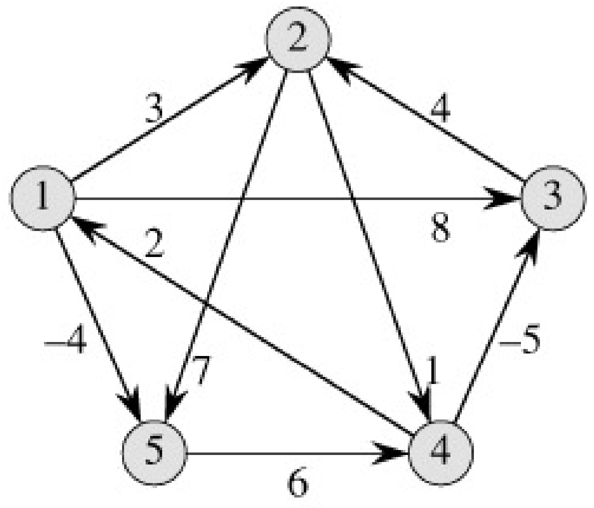
$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & \boxed{-1} & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

-1= min(5,-5+4)

Step 4

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$

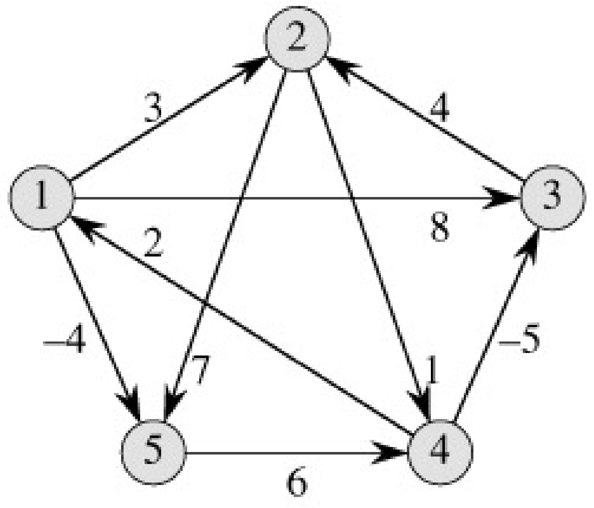


$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Step 5

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$



$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Final shortest distances

Complexity

Running time : 3 nested for loops, n= # vertices
    $O(n^3)$
Memory required : The same
    $O(n^3)$.
If we drop the superscripts and use the same array:
$O(n^2)$

Transitive Closure

- Transitive Closure defines a matrix which shows which pairs of vertices are reachable in a graph.
- If a graph is given, we have to find a vertex j which is reachable from another vertex i, for all vertex pairs (i, j).
- The final matrix is of the Boolean type. When there is a value 1 for vertex i to vertex j, it means that there is at least one path from i to j.
- Note that we are concerned with the existence of paths but not the path lengths.

Transitive Closure of a Directed Graph

Which Vertices of a given directed graph are connected?

- Computing a transitive closure of a directed graph gives complete information about which vertices are connected to which other vertices.

$$t_{ij}^{(0)} = \begin{cases} 0 & i \neq j \text{ and } (i, j) \notin E \\ 1 & i = j \text{ or } (i, j) \in E \end{cases}$$

- Transitive closure definition of a directed graph G=(V,E):

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$$

For i, j , k = 1, 2, …, n , define $t_{ij}^{(k)}=1$, if $\exists$ a path in G from i to j with all intermediate vertices in {1, 2, …, k}; otherwise $t_{ij}^{(k)}=0$

Transitive Closure of a Graph : Explanation

Input: Boolean adjacency matrix: (0-1 matrix)

$\rightarrow$Unweighted adjacency graph *G is given*:

$W[i][j] = 1$, if $(i,j) \in E$,

$W[i][j] = 0$ otherwise.

Output: Transitive closure matrix

$T[i][j] = 1$, if there is a path of any length from i to j in G,

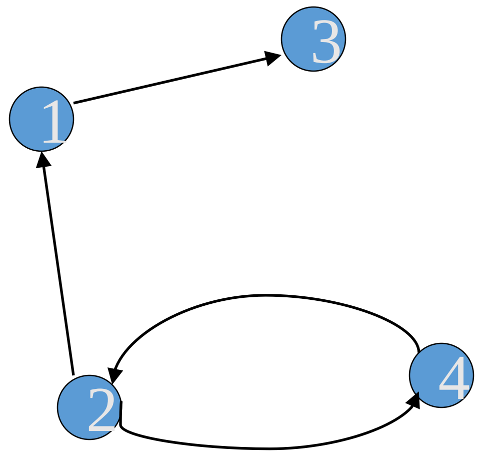
$T[i][j] = 0$ otherwise.

This matrix is also called the “Reachability matrix”

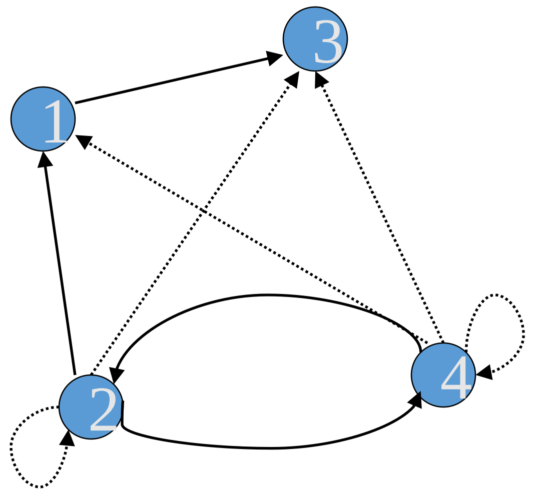
- Algorithm:
  1. Just run Floyd-Warshall with the adjacency matrix G
  2. More efficient: use only Boolean operators

Transitive Closure : Example

- Example Graphs and their transitive closure matrices :



$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$



$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix}$$

Note that there is no connection from 3 to any other node.Why are some entries bold ?

Transitive closure Algorithm -1

//First form the adjacency matrix W= T $^{(0)}$, Then form T$^{(n)}$

```
TRANSITIVE-CLOSURE($G$)
$n \leftarrow |V[G]|$
for $i \leftarrow 1$ to $n$
    do for $j \leftarrow 1$ to $n$
        do if $i = j$ or $(i, j) \in E[G]$
            then $t_{ij}^{(0)} \leftarrow 1$
            else $t_{ij}^{(0)} \leftarrow 0$
for $k \leftarrow 1$ to $n$
    do for $i \leftarrow 1$ to $n$
        do for $j \leftarrow 1$ to $n$
            do $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$
return $T^{(n)}$
```

Transitive Closure Algorithm-2

```
//Assumes that W is given as a Boolean matrix
//Loop indexes start from 1 here
Transitive_Closure(W)
 T ← W    // T=T$^{(0)}$
 for k ←1 to n do // compute T$^{(k)}$
     for i ←1 to n do
         for j ←1 to n do
             T[i][j] ← T[i][j] ∨ (T[i][k] ∧ T[k][j])
 return T
```

Transitive Closure Algorithm: Explanation

- The algorithm is the SAME as the Floyd-Warshall Algorithm, except for when we find a non-infinity path, we simply add an edge to the transitive closure graph.
- In the algorithm , arithmetic operations are replaced by logical operations v and ^
- Reason : Logical operators are more efficient to apply.
- Every round, we use the previous route reached namely, $R^{(k-1)}$

How to generate $R^{(k)}$ from $R^{(k-1}$? There are two possibilities:

$r_{ij}^{(k)} = r_{ij}^{(k-1)}$ or   (No change after adding v_k)

$= (r_{ik}^{(k-1)}$ and $r_{kj}^{(k-1)})$ (A new path is added)

The algorithm makes this choice for every k .

Transitive Closure : Part of a C++ Function

The *transitive closure* is a Boolean Matrix:

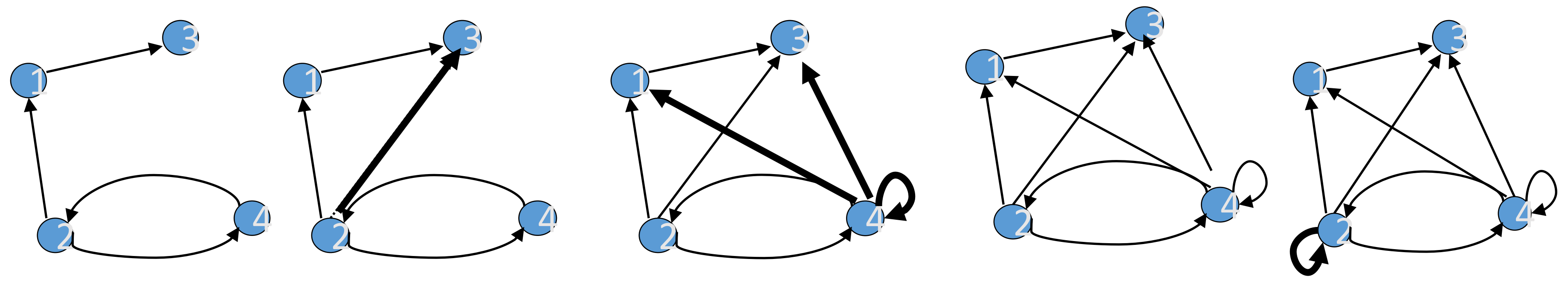
```
bool tc[num_vertices][num_vertices];

// Initialize the matrix tc
//   ...

// Run Floyd-Warshall with Boolean tc matrix
   for ( int k = 0; k < num_vertices; ++k ) {
     for ( int i = 0; i < num_vertices; ++i ) {
        for ( int j = 0; j < num_vertices; ++j ) {
            tc[i][j] = tc[i][j] || (tc[i][k] && tc[k][j]);
        }
    }
}
```

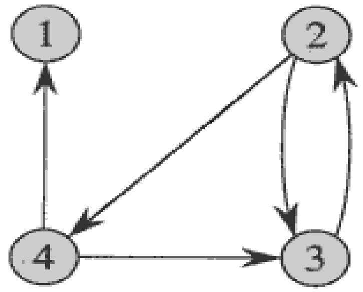
Transitive Closure Algorithm:Example-1

Constructs transitive closure T as the last matrix in the sequence of n-by-n matrices $R^{(0)}, \dots, R^{(k)}, \dots, R^{(n)}$ where
$R^{(k)}[i,j] = 1$ iff there is nontrivial path from i to j with only first k vertices allowed as intermediate
Note that $R^{(0)} = A$ (adjacency matrix), $R^{(n)} = T$ (transitive closure)



$$R^{(0)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix} \qquad R^{(1)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix} \qquad R^{(2)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix} \qquad R^{(3)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix} \qquad R^{(4)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

Transitive closure-Example-2



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Figure 25.5 A directed graph and the matrices $T^{(k)}$ computed by the transitive-closure algorithm.

Computer Applications of Transitive closure

- Computing the transitive closure of a digraph is an important problem in many computer science applications. Some examples:
  - Evaluation of recursive database queries.
  - Analysis of reachability (connectivity) of transition graphs in communication networks.
  - Construction of parsing automata in compilers.