**Floyd-Warshall Algorithm**

Floyd-Warshall Algorithm is an algorithm 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. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

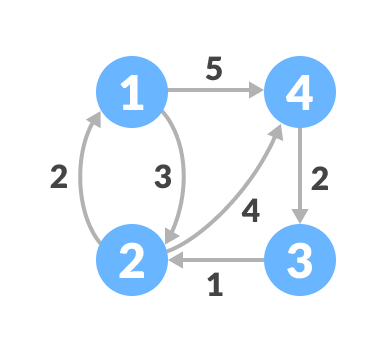
A weighted graph is a graph in which each edge has a numerical value associated with it.

Floyd-Warhshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.

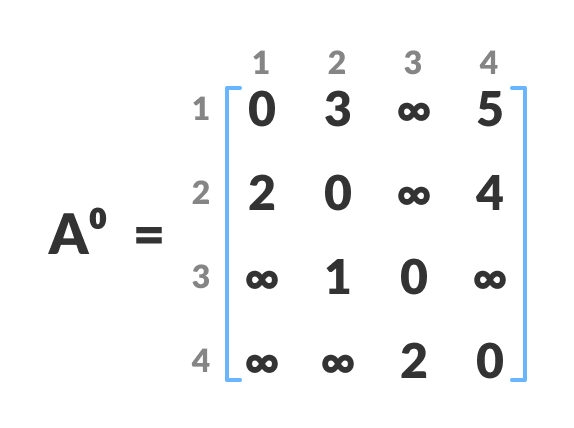
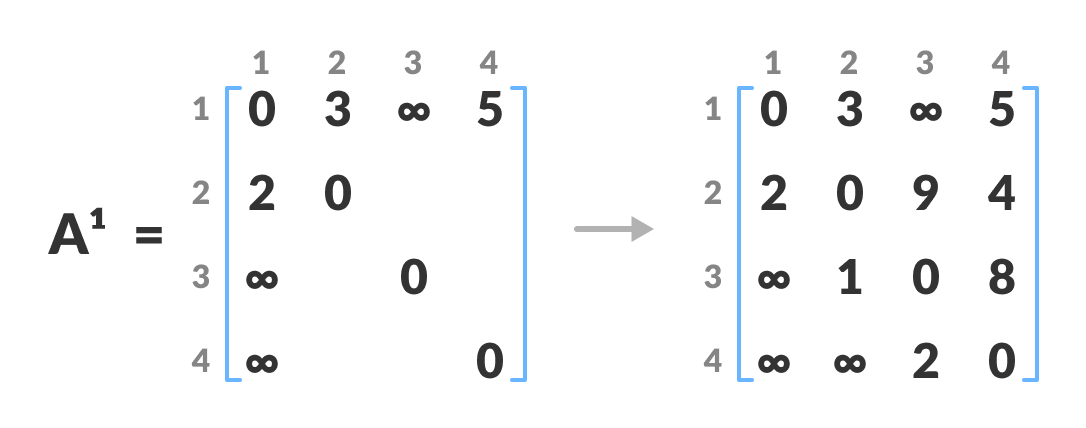
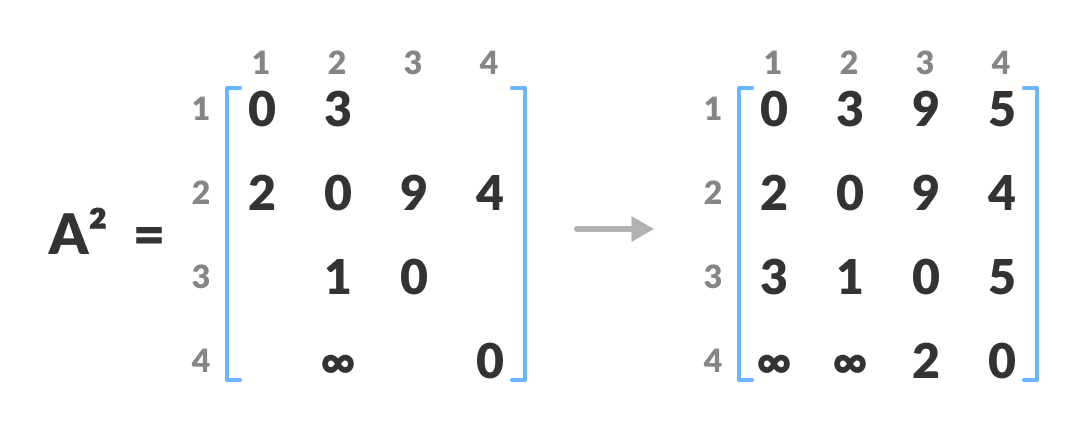
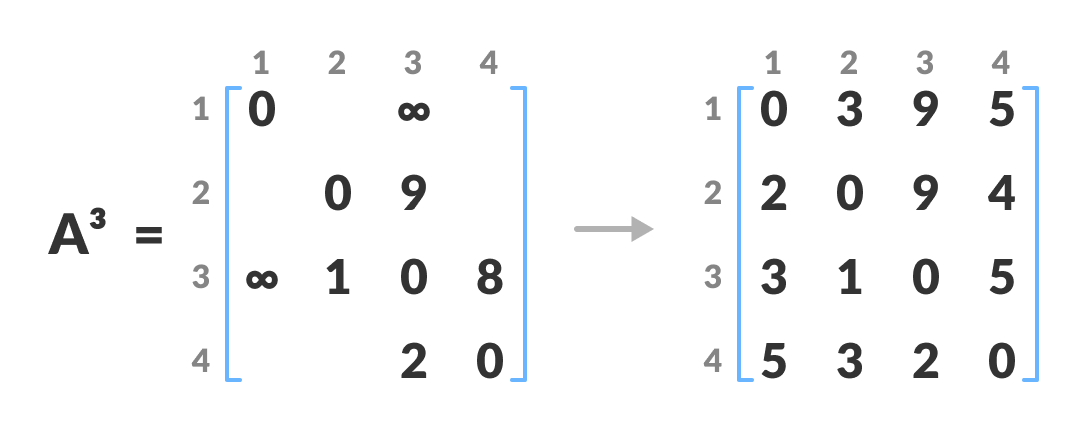
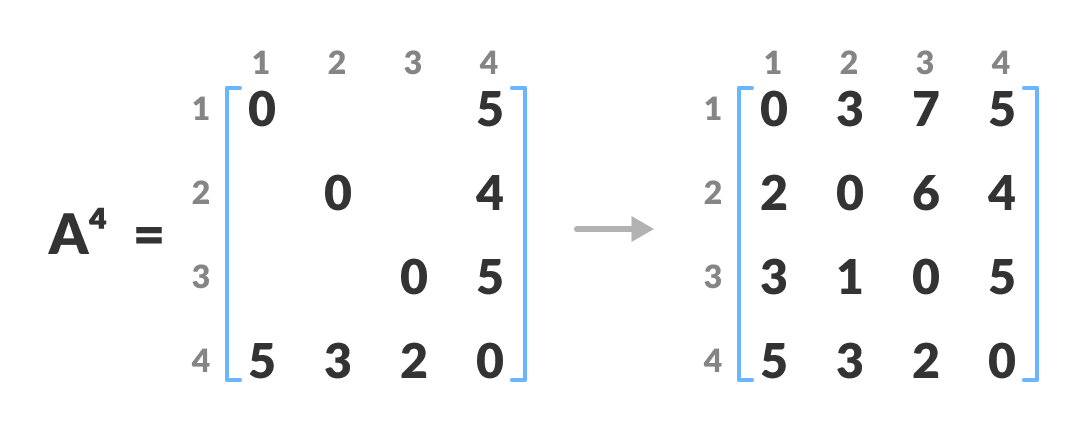
This algorithm follows the [dynamic programming](https://www.programiz.com/dsa/dynamic-programming) approach to find the shortest paths.

**How Floyd-Warshall Algorithm Works?**

Let the given graph be:

Initial graph

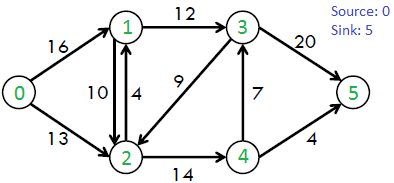
Follow the steps below to find the shortest path between all the pairs of vertices.

1. Create a matrix A0 of dimension n\*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.  
     
   Each cell A[i][j] is filled with the distance from the ith vertex to the jth vertex. If there is no path from ith vertex to jth vertex, the cell is left as infinity.Fill each cell with the distance between ith and jth vertex
2. Now, create a matrix A1 using matrix A0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.  
     
   Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).  
     
   That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].  
     
   In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.Calculate the distance from the source vertex to destination vertex through this vertex k  
   For example: For A1[2, 4], the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since 4 < 7, A0[2, 4] is filled with 4.
3. Similarly, A2 is created using A1. The elements in the second column and the second row are left as they are.  
     
   In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in step 2.Calculate the distance from the source vertex to destination vertex through this vertex 2
4. Similarly, A3 and A4 is also created.Calculate the distance from the source vertex to destination vertex through this vertex 3 Calculate the distance from the source vertex to destination vertex through this vertex 4
5. A4 gives the shortest path between each pair of vertices.

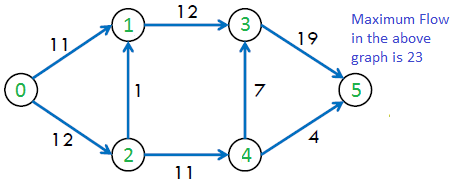
Given a graph which represents a flow network where every edge has a capacity. Also given two vertices *source* ‘s’ and *sink* ‘t’ in the graph, find the maximum possible flow from s to t with following constraints:

* Flow on an edge doesn’t exceed the given capacity of the edge.
* Incoming flow is equal to outgoing flow for every vertex except s and t.

For example, consider the following graph from CLRS book.



The maximum possible flow in the above graph is 23.



Recommended Practice

[Find the Maximum Flow](https://practice.geeksforgeeks.org/problems/find-the-maximum-flow2126/1/)

[Try It!](https://practice.geeksforgeeks.org/problems/find-the-maximum-flow2126/1/)

***Ford-Fulkerson Algorithm***

*The following is simple idea of Ford-Fulkerson algorithm:*

1. *Start with initial flow as 0.*
2. *While there is a augmenting path from source to sink.*
   * *Add this path-flow to flow.*
3. *Return flow.*

**Time Complexity:** Time complexity of the above algorithm is O(max\_flow \* E). We run a loop while there is an augmenting path. In worst case, we may add 1 unit flow in every iteration. Therefore the time complexity becomes O(max\_flow \* E).

**How to implement the above simple algorithm?**

Let us first define the concept of Residual Graph which is needed for understanding the implementation.

***Residual Graph*** of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called ***residual capacity*** which is equal to original capacity of the edge minus current flow. Residual capacity is basically the current capacity of the edge.

Let us now talk about implementation details. Residual capacity is 0 if there is no edge between two vertices of residual graph. We can initialize the residual graph as original graph as there is no initial flow and initially residual capacity is equal to original capacity. To find an augmenting path, we can either do a BFS or DFS of the residual graph. We have used BFS in below implementation. Using BFS, we can find out if there is a path from source to sink. BFS also builds parent[] array. Using the parent[] array, we traverse through the found path and find possible flow through this path by finding minimum residual capacity along the path. We later add the found path flow to overall flow.

The important thing is, we need to update residual capacities in the residual graph. We subtract path flow from all edges along the path and we add path flow along the reverse edges We need to add path flow along reverse edges because may later need to send flow in reverse direction