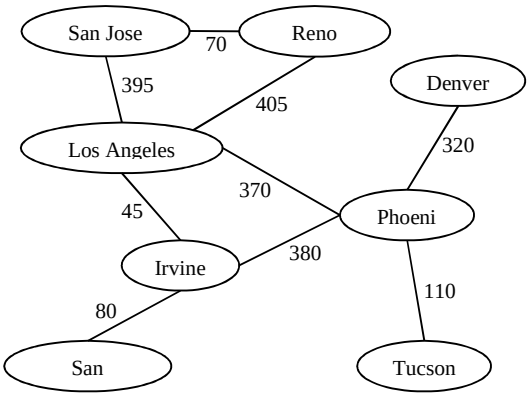
Level 4: Graph Theory

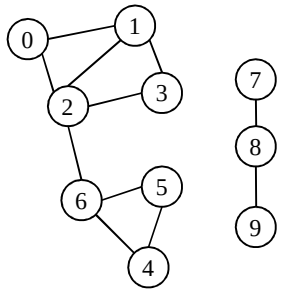
Consider the simple map below:



It contains cities such as *Los Angeles*, *Irvine* and roads between them. The lines are the roads and the numbers on the lines are the length of the roads. A graph is a mathematical object that can model these simple maps.

1. Representation of Graphs

A *graph* is a collection of vertices and edges. An *edge* is a connection between two *vertices*. Vertices can be considered as cities and edges can be considered as roads in a map. In the graph, adjacent vertices of a vertex *k*, are called **neighbors** of *k*. In the example above, *Irvine* has *San Diego*, *Phoenix* and *Los Angeles* as neighbors.



The vertices of the graph above are {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

We can represent a graph in matrix form in our programs. For example, the graph above can be represented as follows:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	1	1	0	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	0	0
2	1	1	0	1	0	0	1	0	0	0
3	0	1	1	0	0	0	0	0	0	0
4	0	0	0	0	0	1	1	0	0	0
5	0	0	0	0	1	0	1	0	0	0
6	0	0	1	0	1	1	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	1	0	1
9	0	0	0	0	0	0	0	0	1	0

In the matrix, the location (i, j) is 1 if there is an edge between vertices i and j where i is the row and j is the column. 0 represents no edge between vertices i and j. For example, there is a road between 7 and 8 hence location (7, 8) is 1. Notice that (i, j) and (j, i) are same since the edges are two-way. This matrix is called **adjacency matrix**.

Exercise 1: Write a program that outputs the neighbors of a given vertex *k*. The input has three integers *n*, *m* and *k* in the first line. *n* is the number of vertices (numbered from 0 to *n* – 1) and *m* is the number of edges. Then there will be *m* lines that has two integers, *a* and *b* which means there is an edge between vertices *a* and *b*. In the output file, write the number of neighbors of *k* in the first line then its neighbors from smaller to larger.

Sample: The graph above is given as the input as follows:

neighbor.in:	*neighbor.out*:

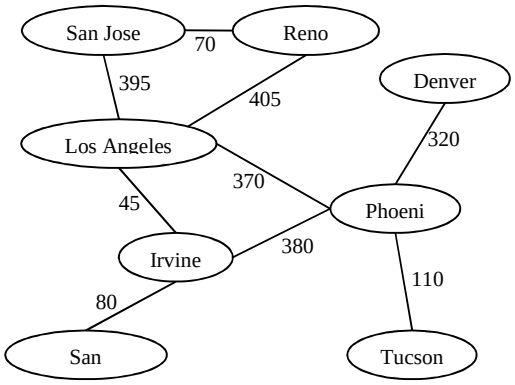
`10 11 6`
`9 8`
`7 8`
`6 5`
`6 2`
`0 1`
`2 0`
`3 1`
`1 2`
`3 2`
`4 5`
`4 6`
	`3`
	`2`
	`4`
	`5`

In this graph, vertex 6 has neighbors of 2, 4 and 5.

2. Basic Concepts in Graphs

It is not necessary for a graph to have weights on the edges such as the graph above. The number on an edge of a graph is called the *weight* of that edge like the length of the road. If a graph has weights on its edges, it is called *weighted graph*. For instance, maps are weighted graph.

Question: Through which cities we can go from San Diego to San Jose in the following map?

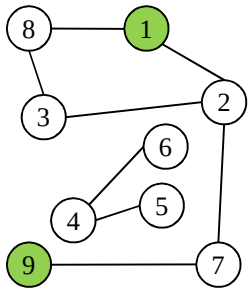


A *path* from a city to another city is a list of cities, in which successive cities are connected by road on the map. In the map above, *San Diego – Irvine – Los Angeles – San Jose* is a path from *San Diego* to *San Jose*. *San Diego – Irvine – Phoenix – Irvine – Los Angeles – San Jose* is also a path however it is not a *simple path*. A city cannot be used more than once in a simple path like *Irvine* in this path.

Question: What is the distance between *San Diego* and *San Jose* following the path *San Diego – Irvine – Los Angeles – San Jose* in the map given above?

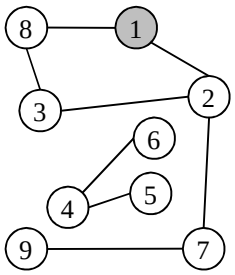
A path in a graph is similarly a list of vertices, in which successive vertices are connected with an edge in the graph.

Question: In the following graph, find a path from 1 to 9.

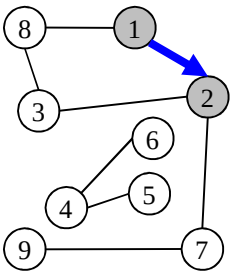


Question: How can you find a path between two vertices in a given graph?

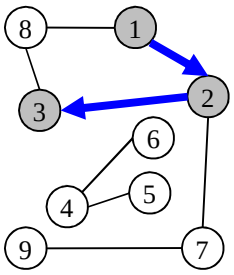
DFS will search the graph as follows:



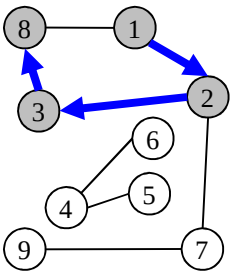
(1)



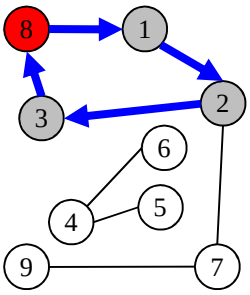
(2)



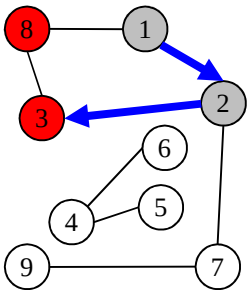
(3)



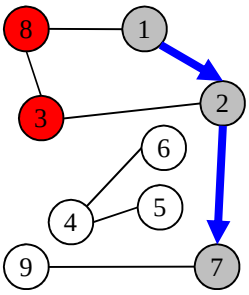
(4)



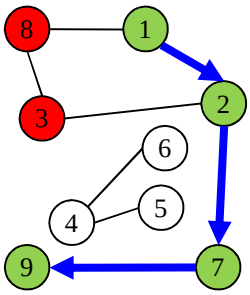
(5) Stuck at 8 – return



(6) Stuck at 3 – return



(7)

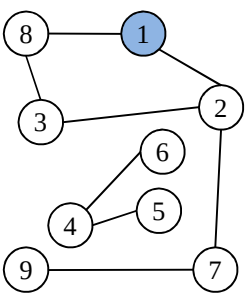


(8)

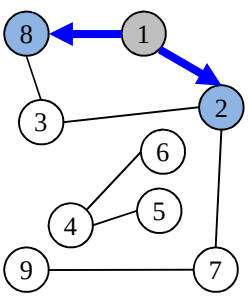
The search proceeds as follows:

```
1 (start)
|---> 2
      |---> 3
      |     |---> 8 ⇒ stuck – return
      |     |---> stuck – return
      |---> 7
            |---> 9 (finished)
```

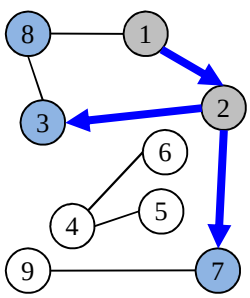
BFS will search the graph as follows (blue vertices are in the queue):



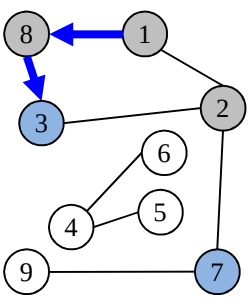
(1)



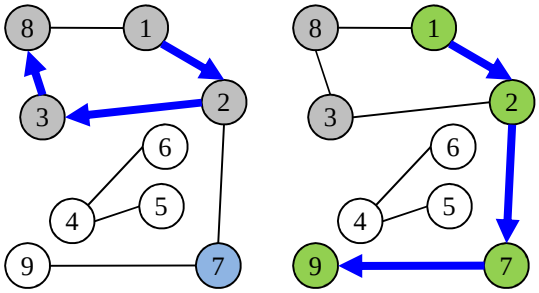
(2)



(3)



(4)



(5) (6)

The search proceeds as follows:

Step	*Queue*
Initially	
1	1
2	2 8
3	8 3 7
4	3 7
5	7
6	

Exercise 2: Write a program that finds a path between two vertices *x* and *y* in a given graph using DFS. The input has four integers *n*, *m*, *x* and *y* in the first line. *n* is the number of vertices (numbered from 1 to *n*) and *m* is the number of edges. Then there will be *m* lines that has two integers, *a* and *b* which means there is an edge between vertices *a* and *b*. In the output file, write the length of the path (the number of vertices in the path) in the first line then the path from *x* to *y*.

Sample: The graph above is given as the input as follows:

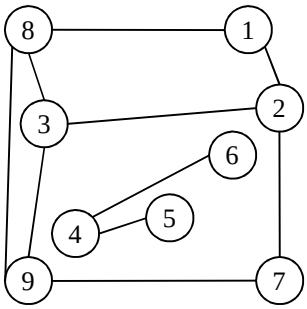
pathdfs.in:	*pathdfs.out*:

`9 8 1 9`
`4 5`
`5 6`
`9 7`
`7 2`
`3 8`
`2 3`
`1 8`
`1 2`
	`4`
	`1`
	`2`
	`7`
	`9`

Exercise 3: Solve *Exercise 2* using BFS. The input file is *pathbfs.in* and the output file is *pathbfs.out*.

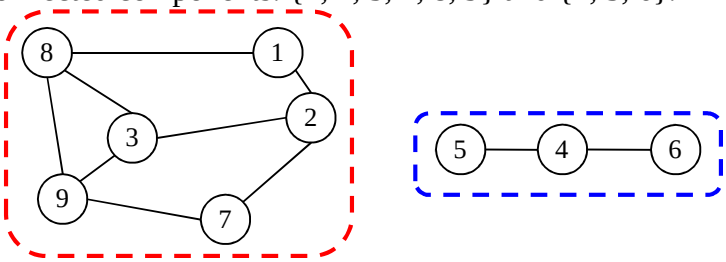
Connectivity

Question: In the graph below, find a path from 6 to 1?



Question: How can you determine if there is a path between two vertices?

A graph is *connected* if there is a path from a vertex to all vertices in the graph. A graph which is not connected is composed of *connected components*. For example, the graph below has two connected components: {1, 2, 3, 7, 8, 9} and {4, 5, 6}.



Question: How can you determine if there is a graph is connected?

Question: How can you find the connected components of a graph?

Exercise 4: Write a program that finds the connected components of a given graph using DFS. The input has two integers *n* and *m* in the first line. *n* is the number of vertices (numbered from 1 to *n*) and *m* is the number of edges. Then there will be *m* lines that has two integers, *a* and *b* which means there is an edge between vertices *a* and *b*. In the output file the first line is the number of components. Then each line provides the vertices in each component.

Sample: The graph above is given as the input as follows:

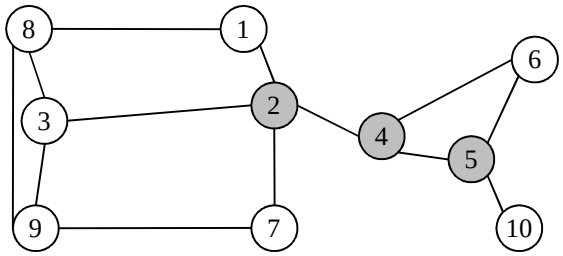
compdfs.in:	*compdfs.out*:

`9 10`
`4 5`
`5 6`
`9 7`
	`2`
	`1 2 3 7 8 9`
	`4 5 6`

```
7 2
3 8
2 3
1 8
1 2
8 9
9 3
```

Exercise 5: Solve *Exercise 4* using BFS. The input file is *compbfs.in* and the output file is *compbfs.out*.

A connected graph may have a *critical vertex k* such that there is no path between two arbitrary vertices *a* and *b* when k is removed from the graph. For example, in the following graph {2, 4, 5} are critical vertices of the graph. When 5 is removed from the graph, 10 has no connection to the graph. Similarly, there will be no connection from 1 to 5 when 4 is removed.



Exercise 6: Write a program that finds critical vertices in a graph. The input has two integers *n* and *m* in the first line. *n* is the number of vertices (numbered from 1 to *n*) and *m* is the number of edges. Then there will be *m* lines, each has two integers corresponding to an edge. In the output file the first line is the number of critical vertices, then the list of critical vertices in ascending order.

Sample: The graph above is given as the input as follows:

critical.in:

```
10 13
1 8
1 2
2 3
2 7
2 4
3 8
3 9
4 5
4 6
5 6
5 10
7 9
8 9
```

critical.out:

```
3
2
4
5
```

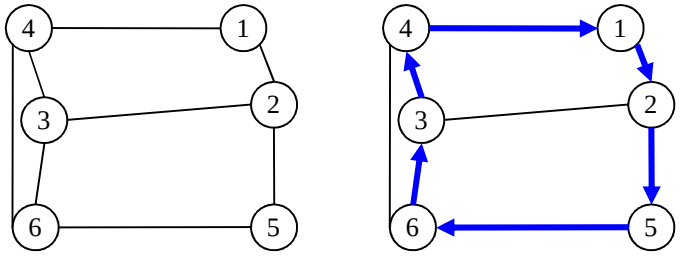
A graph is *biconnected* if there are at least two disjoint paths (paths that have no common vertices) between any two vertices.

Exercise 7: Write a program that checks if a graph is biconnected or not. The input has two integers *n* and *m* in the first line. *n* is the number of vertices numbered from 1 to *n*, and *m* is the number of edges. Then there will be *m* lines, each has two integers corresponding to an edge. In the output file, 'yes' means biconnected graph otherwise 'no'.

A *cycle* is a path that has the same ending vertex with its starting vertex. A cycle has to be a simple path except its starting and ending vertices.

Exercise 8: Write a program that finds a cycle that travels all the vertices exactly once in a given graph. The input has two integers *n* and *m* in the first line. *n* is the number of vertices (numbered from 1 to *n*) and *m* is the number of edges. Then there will be *m* lines, each has two integers corresponding to an edge. In the output file there will be *n* lines that corresponds to the order of the vertices in the cycle. The first vertex in the cycle is 1.

For example, the following graph in (a) has 1 – 2 – 5 – 6 – 3 – 4 as the satisfying cycle as illustrated in (b).



(a) (b)

Sample: The graph above is given as the input as follows:

hcycle.in:	*hcycle.out*:
6 8 1 6 1 4 2 3 5 2 6 5 3 6 3 4 6 4	1 2 5 6 3 4

Question: How must be the graph in order to have a cycle in the above question?

Transitive Closure

There is another way to check the connectivity in a graph. Connectivity has a transitive property among vertices; if *x* is connected to *y* and *y* is connected to *z* then *x* is connected to *z*. The idea is to generate the *transitive closure* matrix of the graph. For all vertex pairs (*i*, *j*), we can check an intermediate vertex *k* such that if *i* is connected to *k* and *k* is connected to *j*, *i* and *j* are connected.

Given the adjacency matrix `mat` of a graph G, the following code will generate the transitive closure matrix of G on the same matrix. In the code, *n* is the number of vertices, and vertices are numbered from 0 to *n* – 1.

```
for (k=0; k < n; k++)
  for (i=0; i < n; i++)
    if (mat[i][k] == 1)       // if there is a path from i to k
      for (j=0; j < n; j++)
        if (mat[k][j] == 1) mat[i][j]=1;
```

NOTE: Be careful about the order of the loops in the code. If it is changed, the code will not work properly.

Question: How can we also use this code to check if there is a path from a vertex to another one?

Question: How can we also use this code to check if a graph is connected or not?

Question: How can we also use this code to output connected components in a graph?

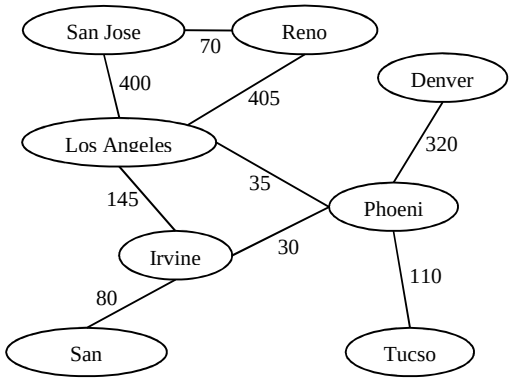
Exercise 9: Solve *Exercise 4* using transitive closure. The input file is *tclosure.in* and the output file is *tclosure.out*.

Shortest Path

We can find the distance between two vertices by summing all the edge weights on the given path in a weighted graph.

Shortest path between two cities in a map is the path that has shortest distance between two cities. Similarly, in a weighted graph, shortest path between two vertices is the path that has shortest distance.

Question: What is the shortest path from *San Diego* to *San Jose* in the following map? What is the shortest distance of these cities?



We can use the idea in transitive closure algorithm, to calculate all the shortest path between any two vertex pair; `mat[x][y]` will give us the length of shortest path between *x* and *y*. Observe that if there is a path between *i* and *k* with length *l*(*i*,*k*) and there is a path between *k* and *j* with length *l*(*k*,*j*) then there is a path between *i* and *j* with length *l*(*i*,*j*) = *l*(*i*,*k*) + *l*(*k*,*j*). If this path is shorter than a previously found path then we update it.

Given the adjacency matrix `mat` of a graph G where `mat[x][y]` indicates the length of the road between *x* and *y*, the following code will generate the all shortest path matrix of G on the same matrix. In the code, *n* is the number of vertices, and vertices are numbered from 0 to *n* – 1.

```
for (k=0; k < n; k++)
  for (i=0; i < n; i++)
    if (mat[i][k] > 0)        // if there is a path from i to k
      for (j=0; j < n; j++)
        if (mat[k][j] > 0)    // if there is a path from k to j
          // if there is no path between i and j, or it is longer
          if (mat[i][j] == 0 || (mat[i][j] > mat[i][k]+mat[k][j])
            mat[i][j]=mat[i][k]+mat[k][j];
```

Exercise 10: Write a program that finds the shortest path between two vertices *a* and *b* in a given graph. The input has four integers *n*, *m*, *a*, and *b* in the first line. *n* is the number of vertices (numbered from 1 to *n*) and *m* is the number of edges. Then there will be *m* lines, each has three integers corresponding to an edge and its length. In the output file the first line has one integer; the length of the path.

Sample: The graph above is given as the input as follows:

allspath.in:	*allspath.out*:
9 14 1 3 1 6 3 1 8 5 1 4 17 1 5 10 6 7 14 6 3 37 2 7 12 2 9 8 3 9 12 4 8 9 4 5 2 4 9 9 5 8 4 7 9 7	32

3. Trees

A **tree** is an undirected graph that is connected but has no cycle (with more than two vertices). For example:

(a) (b)

Figure (a) is not a tree but figure (b) is a tree. In figure (a) 1 – 2 – 3 – 4 – 1 is a cycle.

NOTE: Observe that there is one unique path between any two vertices in a tree.

Question: Why?

Question: How can we determine if there is a cycle in a given graph?

Question: How can we determine if a given graph is tree?

We can represent trees in an adjacency matrix as in graphs.

Exercise 11: Write a program that checks if a given graph is a tree or not. The input has two integers *n* and *m* in the first line. *n* is the number of vertices (numbered from 1 to *n*) and *m* is the number of edges. Then there will be *m* lines, each has two integers corresponding to an edge. In the output file is one line either "`TREE`" or "`NOT TREE`".

Sample: The tree above is given as the input as follows:

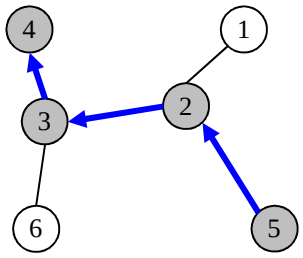
tree.in:	*tree.out*:
5 6 1 2 2 3 3 4 3 6 2 5	TREE

Question: How many edges does a tree have if it has *n* vertices? Why?

Exercise 12: Write a program that outputs the path between two vertices *x* and *y* in a given tree. The input four two integers *n*, *m*, *x* and *y* in the first line. *n* is the number of vertices (numbered from 1 to *n*) and *m* is the number of edges. Then there will be *m* lines, each has two integers corresponding to an edge. In the output file the first line is the number of vertices in the path, then the list of vertices in the path from *x* to *y*.

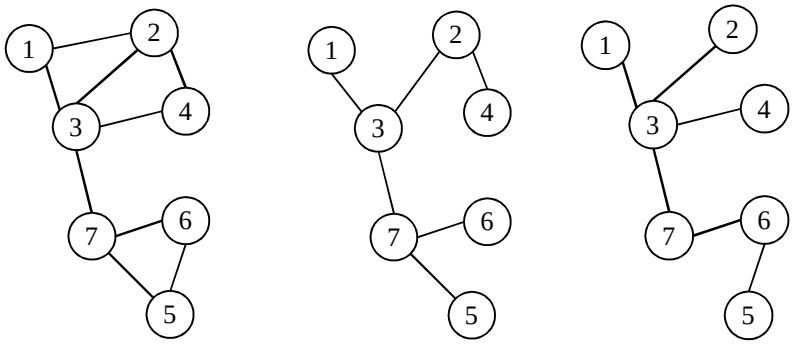
Sample: The tree above is given as the input as follows:

treepath.in:	*treepath.out*:
5 6 5 4 1 2 2 3 3 4 3 6 2 5	4 5 2 3 4

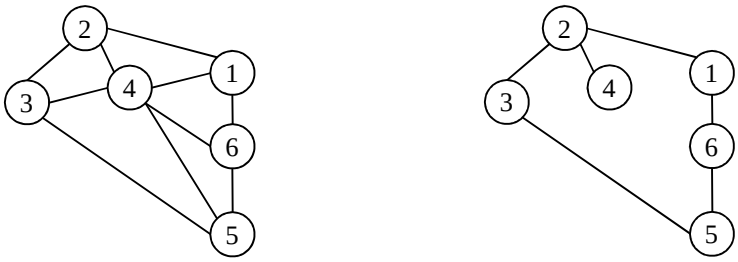


Spanning Tree

A **spanning tree** of a graph is a subgraph of it, contains all the vertices, and forms a tree.



Example: Above, the trees in the middle and on the right are examples of spanning trees of the graph on the left.



Question: In the figures above, is the structure on the right, a spanning tree of the graph on the left? Why / why not?

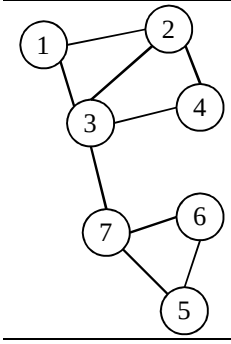
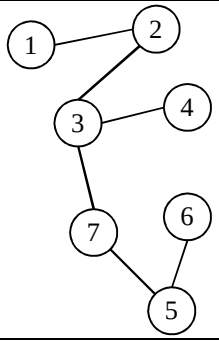
Question: How do you write a program that checks if a graph G_2 is a spanning tree of a graph G_1 ?

Question: How do you find a spanning tree of a given graph?

Question: If graph is not connected, does it have a spanning tree?

Exercise 13: Write a program that finds a spanning tree of a given graph using DFS. The input has two integers *n* and *m* in the first line. *n* is the number of vertices (numbered from 1 to *n*) and *m* is the number of edges. Then there will be *m* lines, each has two integers corresponding to an edge. In the output file is n – 1 lines where each line indicates an edge.

Sample: The tree below is given as the input as follows:

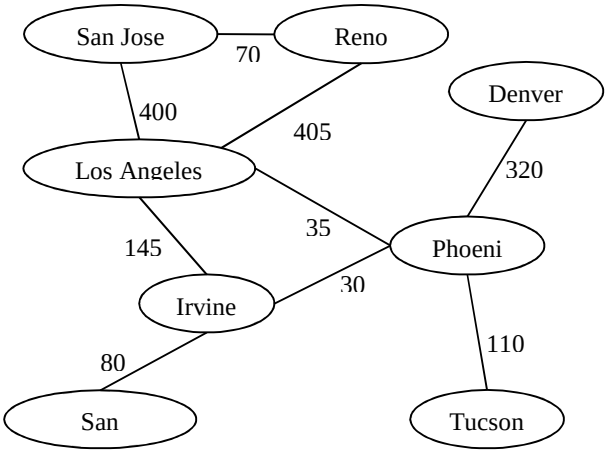
spandfs.in:	*spandfs.out*:
7 9 1 2 2 3 1 3 2 4 3 4 3 7 7 6 7 5 6 5	1 2 2 3 3 4 3 7 7 5 5 6
	

Exercise 14: Solve *Exercise 13* using BFS. The input file is *spanbfs.in* and the output file is *spanbfs.out*.

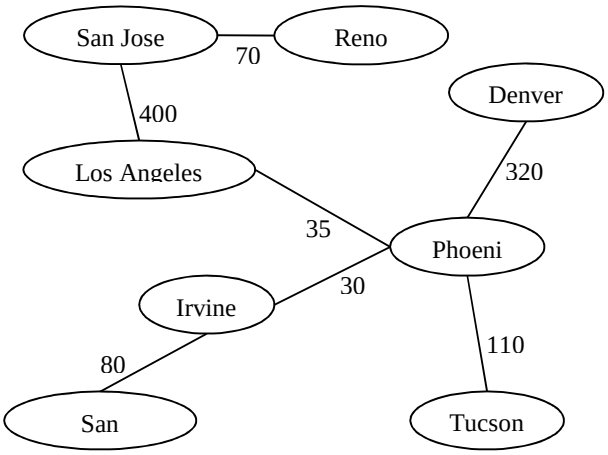
Observe that the output may be different from DFS in this problem. Because, spanning tree is not unique, and DFS and BFS differs in terms of search strategy.

Minimum Spanning Tree

Question: Find a spanning tree of the following graph.



The *minimum spanning tree* (MST) is the spanning tree of a graph with minimum total edge weights. For example,

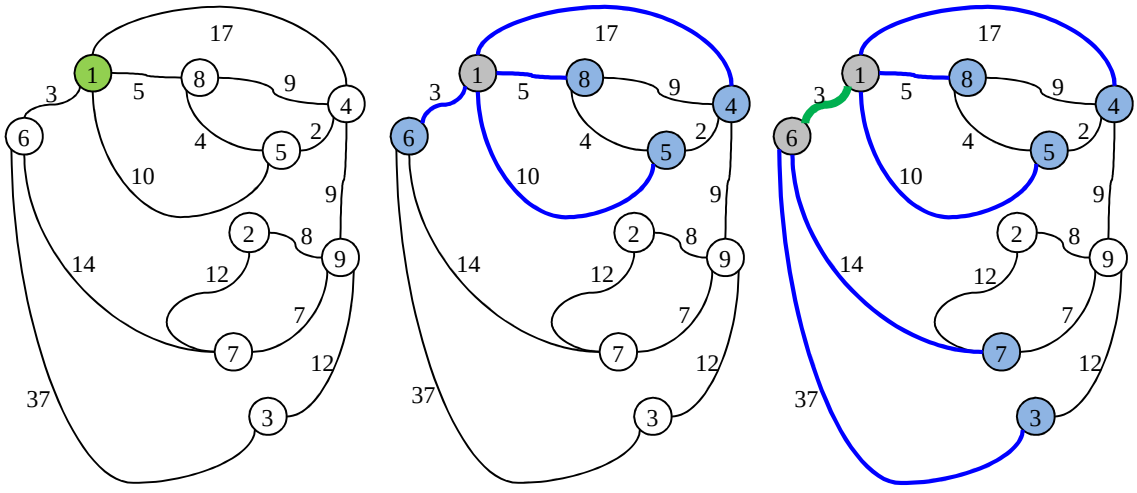


The spanning tree above is the MST of the graph in the previous question. The sum of the edge weights is 400 + 70 + 35 + 320 + 30 + 80 + 110 = 1045.

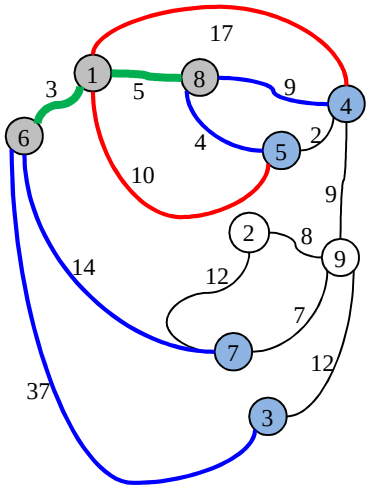
In order to find the MST in a graph, we can use the following simple algorithm.

1. We will have an array that keeps the expanded edge weights during the search; hence initially all the elements of array are ∞ but the starting vertex is 0. The algorithm can start from any vertex.
2. Starting from vertex *a*, we mark it as used, and extend its neighbors which means to record their edge weights.
3. In the next step, we get the vertex with minimum edge weight, and keep the edge in the MST. Then continue expanding its neighbors as well as updating their edge weights.

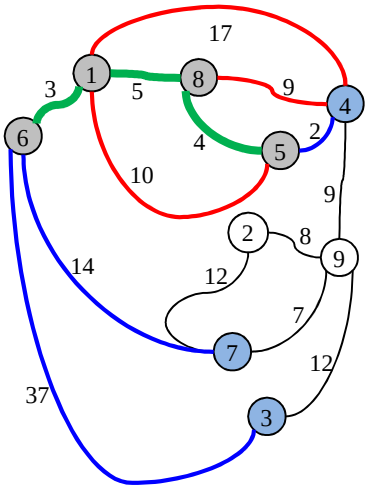
Let's investigate the algorithm on the following graph to find its MST. In the following figure, grey vertices are marked as used, blue vertices are the expanded ones. The blue edges are the current edges in the MST, and the red ones are excluded ones.



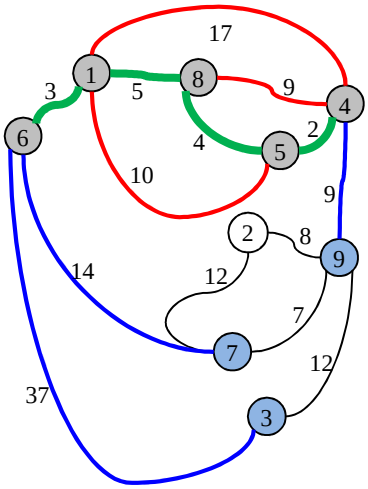
(0)



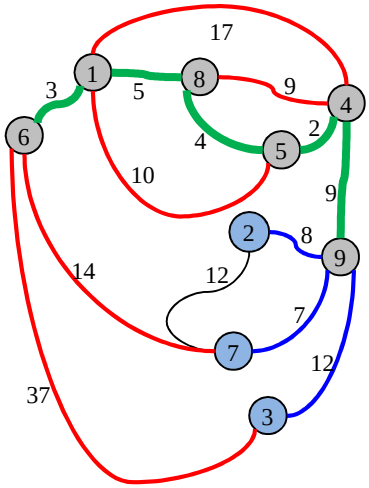
(1) Get 1 and expand it



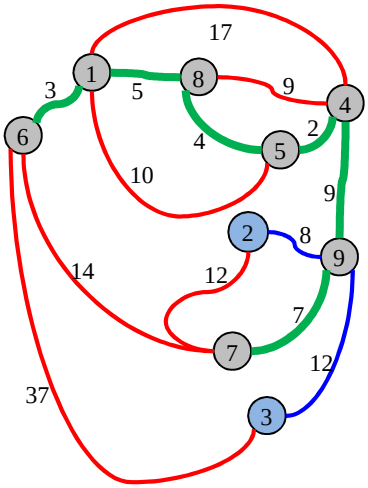
(2) Get 6 and expand it



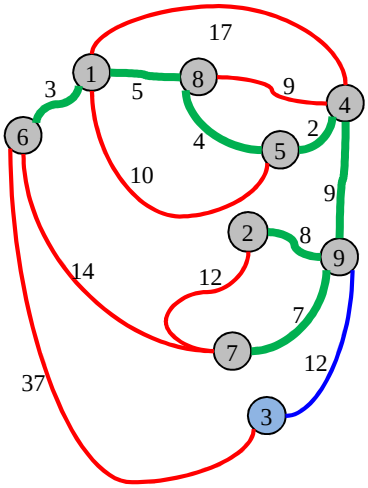
(3) Get 8 and expand it



(4) Get 5 and expand it



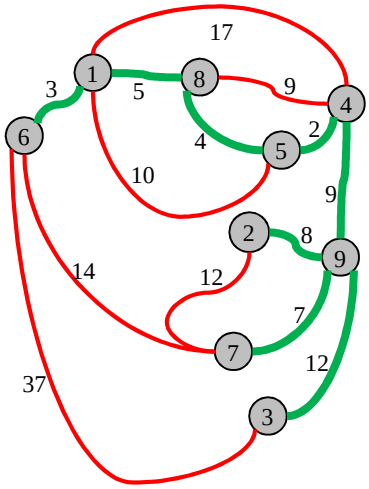
(5) Get 4 and expand it



(6) Get 9 and expand it

(7) Get 7 and expand it

(8) Get 2 and expand it



(9) Get 3 and finish

The following table demonstrates the progress of MST algorithm; how edge weights and spanning tree changes. Grey vertices are the used ones, blues are the expanded ones, and the pinks are the updated ones. `from` array records the previous vertex that reaches to the vertex. For instance, in the first step, we reach vertices 4, 5, 6 and 8 from vertex 1.

Step	*Weights*									*From*								
Initially	0	∞	∞	∞	∞	∞	∞	∞	∞	0	0	0	0	0	0	0	0	0
1	0	∞	∞	17	10	3	∞	5	∞	0	0	0	1	1	1	0	1	0
2	0	∞	37	17	10	3	14	5	∞	0	0	6	1	1	1	6	1	0
3	0	∞	37	9	4	3	14	5	∞	0	0	6	8	8	1	6	1	0
4	0	∞	37	2	4	3	14	5	∞	0	0	6	5	8	1	6	1	0
5	0	∞	37	2	4	3	14	5	9	0	0	6	5	8	1	6	1	4
6	0	8	12	2	4	3	7	5	9	0	9	9	5	8	1	9	1	4
7	0	8	12	2	4	3	7	5	9	0	9	9	5	8	1	9	1	4
8	0	8	12	2	4	3	7	5	9	0	9	9	5	8	1	9	1	4
9	0	8	12	2	4	3	7	5	9	0	9	9	5	8	1	9	1	4

At the end, the MST is recorded in in `from` array. The edges are in the form of (*i*, from[*i*]). For instance (1, 9) is in the tree since 1 is reached from 9. The total weight of MST is the sum of the `weights` array.

Exercise 15: Write a program that finds MST of a given graph. The input has two integers *n* and *m* in the first line. *n* is the number of vertices (numbered from 1 to *n*) and *m* is the number of edges. Then there will be *m* lines, each has three integers corresponding to an edge and its length. In the output file the first line has one integer; the total weight of MST. Then n – 1 lines that lists the edges in MST.

Sample: The graph above is given as the input as follows:

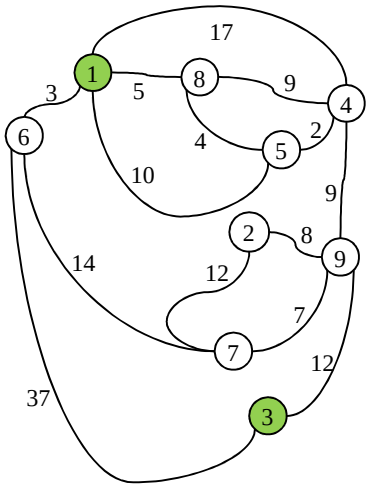
minspan.in:	*minspan.out*:

`9 14`
`1 6 3`
`1 8 5`
`1 4 17`
`1 5 10`
`6 7 14`
`6 3 37`
`2 7 12`
`2 9 8`
`3 9 12`
`4 8 9`
`4 5 2`
`4 9 9`
`5 8 4`
`7 9 7`
	`50`
	`2 9`
	`3 9`
	`4 5`
	`5 8`
	`6 1`
	`7 9`
	`8 1`
	`9 4`

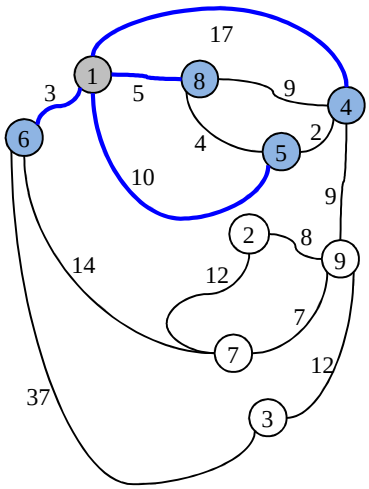
In order to find the shortest path between two vertices *a* and *b*, we can use an algorithm similar to MST. In fact the algorithm provides shortest paths from *a* to all vertices.

4. We will have an array that keeps the distances from vertex *a*; hence initially all the distances are ∞ but the distance of *a* is 0.
5. Starting from *a*, we mark it as used, and extend its neighbors which means to record their distances.
6. In the next step, we get the vertex with minimum total distance, and keep the edge in the shortest path. Then continue expanding its neighbors as well as updating their minimum distances from *a*.

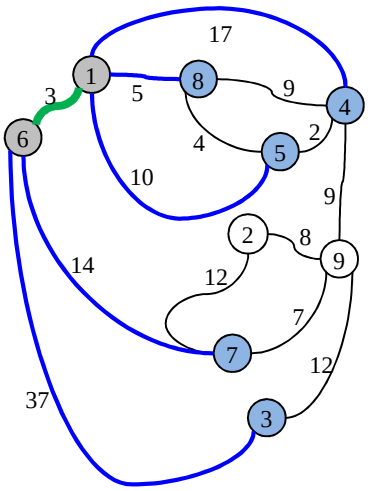
Let's investigate the algorithm on the following graph to find the shortest path between 1 and 3. In the following figure, grey vertices are marked as used, blue vertices are the expanded ones. The blue edges are the current edges in the shortest path from 1 to the corresponding vertex, and the red ones are excluded from shortest paths. Here, we presume the graph is connected.



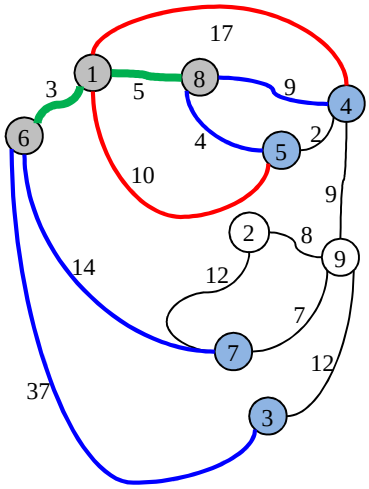
(0)



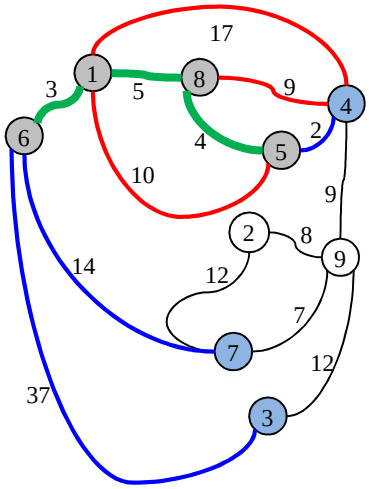
(1) Get 1 and expand it



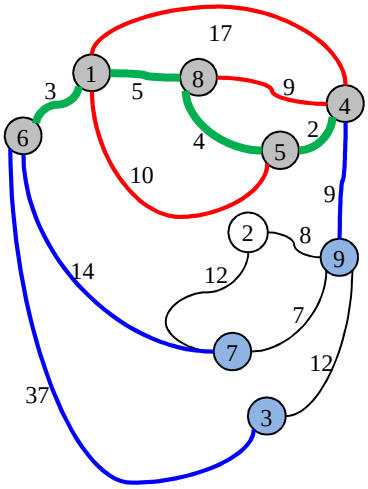
(2) Get 6 and expand it



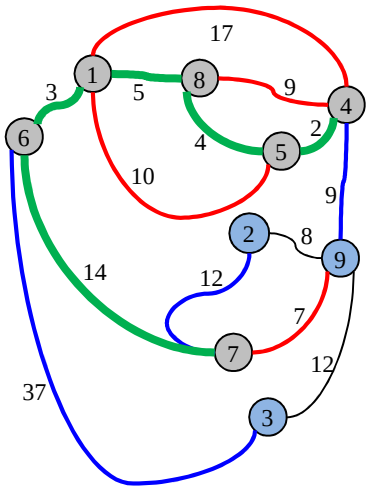
(3) Get 8 and expand it



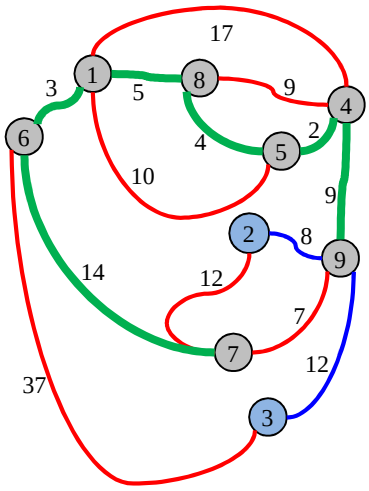
(4) Get 5 and expand it



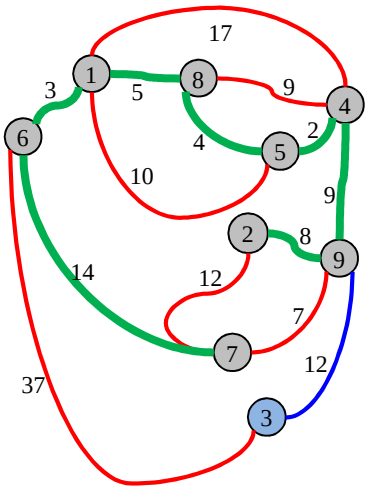
(5) Get 4 and expand it



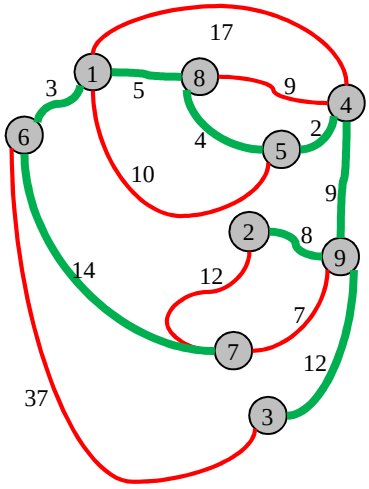
(6) Get 7 and expand it



(7) Get 9 and expand it



(8) Get 2 and expand it



(9) Get 3 and finish

The following table demonstrates the progress of shortest path algorithm; how distances and paths change. Grey vertices are the used ones, blues are the expanded ones, and the pinks are the updated ones. `path` array records the previous vertex that reaches to the vertex. For instance, in the first step, we reach vertices 4, 5, 6 and 8 from vertex 1.

Step	*Distances*									*Paths*								
Initially	0	∞	∞	∞	∞	∞	∞	∞	∞	0	0	0	0	0	0	0	0	0
1	0	∞	∞	17	10	3	∞	5	∞	0	0	0	1	1	1	0	1	0
2	0	∞	40	17	10	3	17	5	∞	0	0	6	1	1	1	6	1	0
3	0	∞	40	14	9	3	17	5	∞	0	0	6	8	8	1	6	1	0
4	0	∞	40	11	9	3	17	5	∞	0	0	6	5	8	1	6	1	0
5	0	∞	40	11	9	3	17	5	20	0	0	6	5	8	1	6	1	4
6	0	29	40	11	9	3	17	5	20	0	7	6	5	8	1	6	1	4
7	0	28	32	11	9	3	17	5	20	0	9	9	5	8	1	6	1	4
8	0	28	32	11	9	3	17	5	20	0	9	9	5	8	1	6	1	4
9	0	28	32	11	9	3	17	5	20	0	9	9	5	8	1	6	1	4

At the end, in order to output the path, we backtrack to the vertex 1, starting from position 3 in `path` array. 3 is the final vertex and we came to 3 from 9, and we came to 9 form 4, and so on. Finally the path is 1 – 8 – 5 – 4 – 9 – 3.

Question: Notice that when you backtrack the `path` array, you get the path in reverse order. How can we output the path in correct order?

Exercise 16: Write a program that finds the shortest path between two vertices *a* and *b* in a given connected graph. The input has four integers *n*, *m*, *a*, and *b* in the first line. *n* is the number of vertices (numbered from 1 to *n*) and *m* is the number of edges. Then there will be *m* lines, each has three integers corresponding to an edge and its length. In the output file the first line has two integers; the length of the path and the number of vertices in the path. Then the list of vertices in the path is given starting from *a*.

Sample: The graph above is given as the input as follows:

spath.in:	*spath.out*:

`9 14 1 3`
`1 6 3`
`1 8 5`
`1 4 17`
`1 5 10`
`6 7 14`
`6 3 37`
`2 7 12`
`2 9 8`
`3 9 12`
`4 8 9`
`4 5 2`
`4 9 9`
`5 8 4`
`7 9 7`
	`32 6`
	`1`
	`8`
	`5`
	`4`
	`9`
	`3`