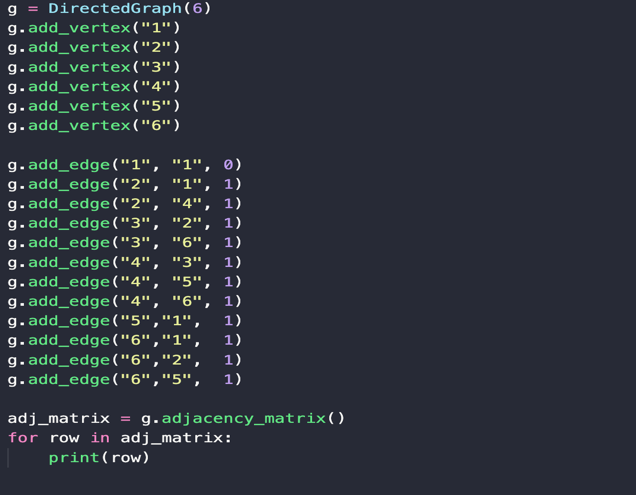
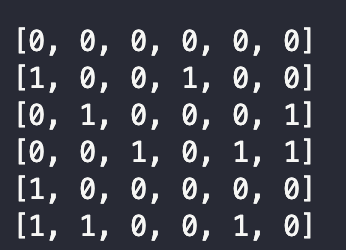
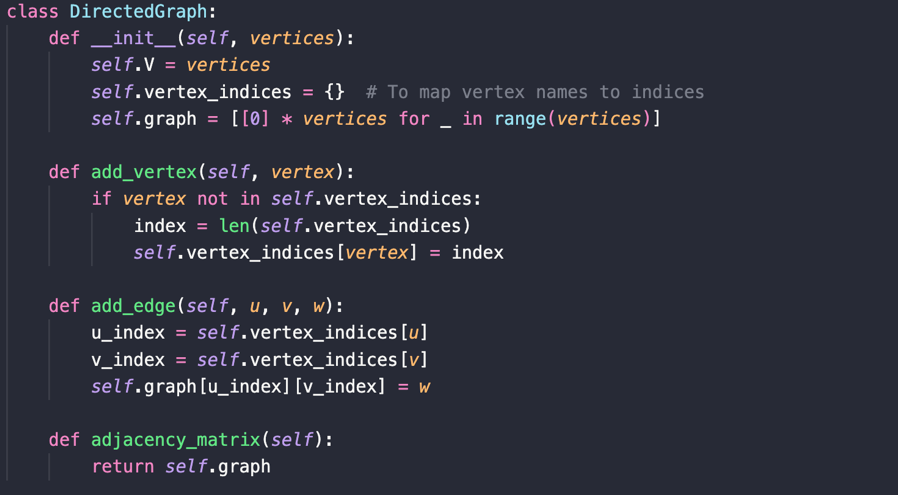


A, Adjacency matrix of G directed graph:

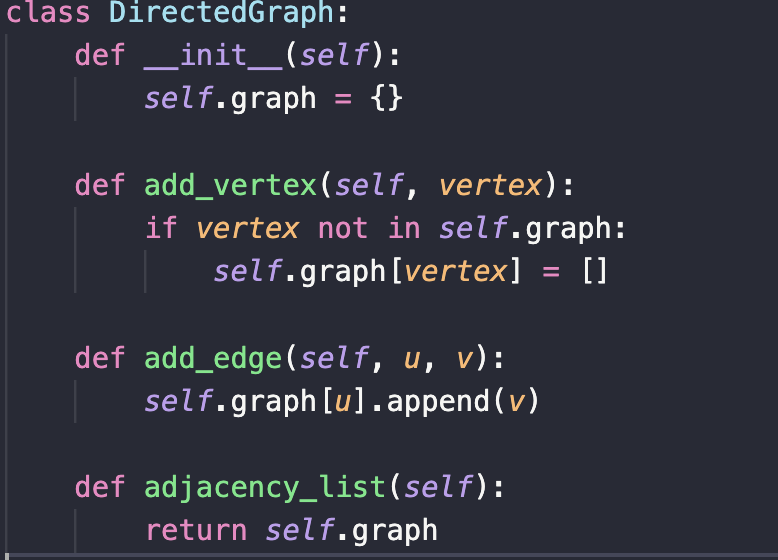
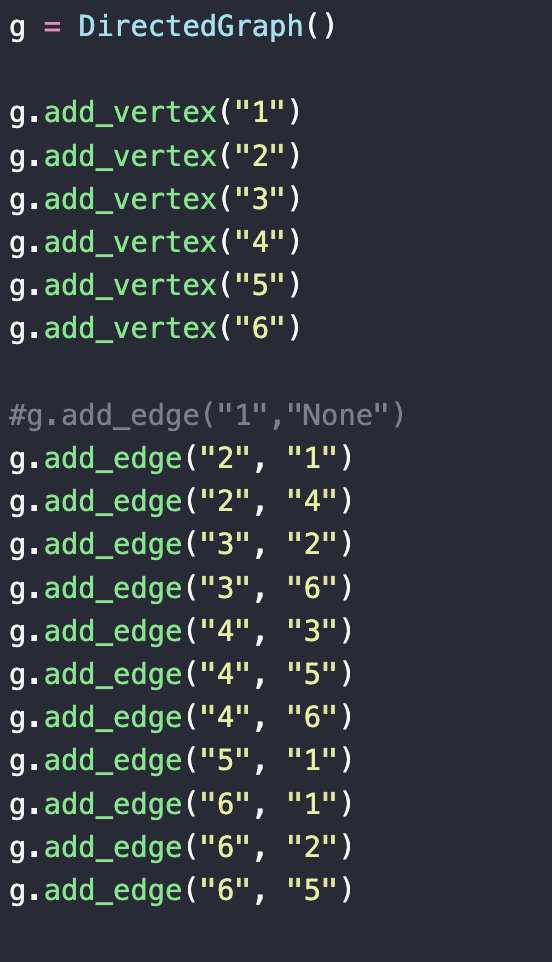
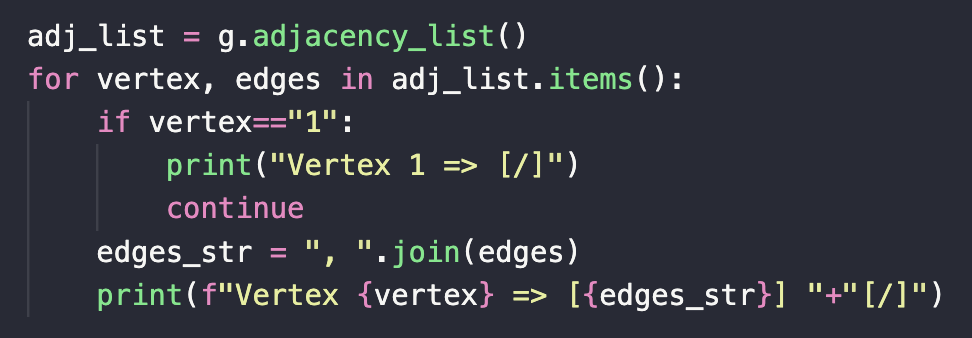
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 1 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 | 0 | 0 |
| 6 | 1 | 1 | 0 | 0 | 1 | 0 |

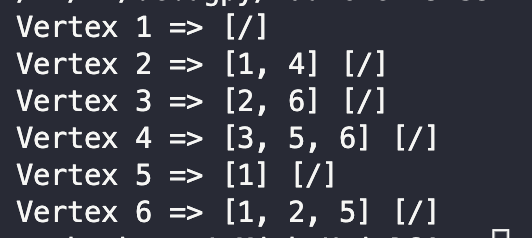
Python program:

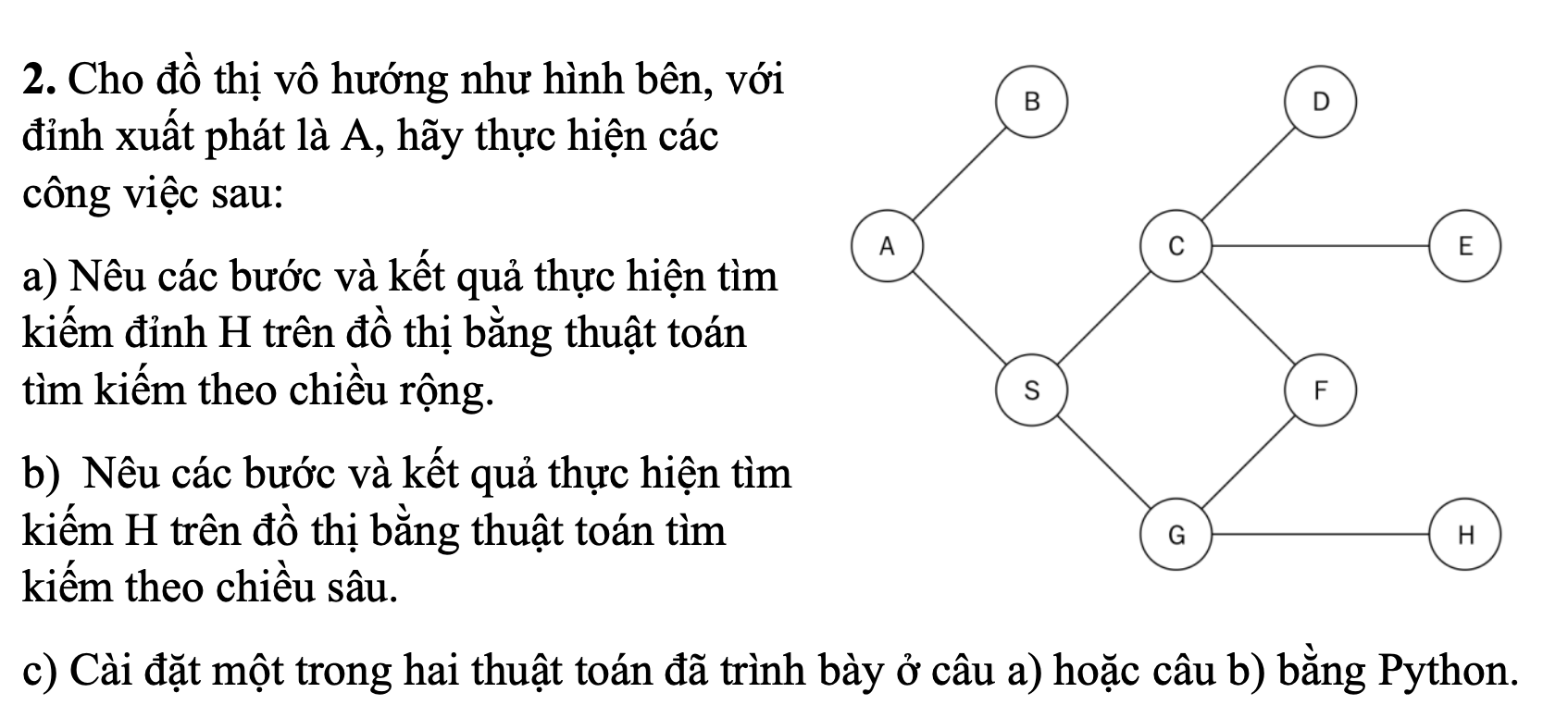


B, Adjacency list of G directed graph:

* Vertex 1 🡪 [/]
* Vertex 2 🡪 [1,4] [/]
* Vertex 3 🡪 [2,6] [/]
* Vertex 4 🡪 [3,5,6] [/]
* Vertex 5 🡪 [1] [/]
* Vertex 6 🡪 [1,2,5] [/]

Python program:





A, Breadth-First Traversal

Algorithm:

* Initialize the queue and the starting vertex is A.
* Put the starting peak in the queue.
* Repeat until the queue is empty.
  + Take the top from the top of the queue (the original top).
  + Check if that peak is peak H, end of search.
  + Otherwise, add all adjacent vertices of the current peak to the queue (if not already visited).
* When the queue is empty and the peak H has not been found yet, the conclusion is that it is not found.

Step 1:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A |  |  |  |  |  |  |  |
| Queue | B | S |  |  |  |  |  |  |

Step 2:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A | B |  |  |  |  |  |  |
| Queue | S |  |  |  |  |  |  |  |

Step 3:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A | B | S |  |  |  |  |  |
| Queue | C | F | G |  |  |  |  |  |

Step 4:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A | B | S | C |  |  |  |  |
| Queue | F | G | D | E |  |  |  |  |

Step 5:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A | B | S | C | F |  |  |  |
| Queue | G | D | E |  |  |  |  |  |

Step 6:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A | B | S | C | F | G |  |  |
| Queue | D | E | H |  |  |  |  |  |

Step 7:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A | B | S | C | F | G | D | E | **H** |
| Queue |  |  |  |  |  |  |  |  |  |

**Found a vertex H in the last step!**

B, Depth-First Traversal

Algorithm:

* Initialize the stack and the starting vertex is A.
* Put the starting peak in the stack.
* Repeat until the stack is empty.
  + Take the top from the top of the stack (the original top).
  + Check if that peak is peak H, end of search.
  + Otherwise, add all adjacent vertices of the current peak to the stack (if not already visited).
* When the stack is empty and the peak H has not been found yet, the conclusion is that it is not found.

Step 1:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A |  |  |  |  |  |  |  |
| Stack | B |  |  |  |  |  |  |  |

Step 2:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A | B |  |  |  |  |  |  |
| Stack | S |  |  |  |  |  |  |  |

Step 3:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A | B | S |  |  |  |  |  |
| Stack | C |  |  |  |  |  |  |  |

Step 4:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A | B | S | C |  |  |  |  |
| Stack | D |  |  |  |  |  |  |  |

Step 5:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A | B | S | C | D |  |  |  |
| Stack | E |  |  |  |  |  |  |  |

Step 6:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A | B | S | C | D | E |  |  |
| Stack | F |  |  |  |  |  |  |  |

Step 7:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A | B | S | C | D | E | F |  |
| Stack | G |  |  |  |  |  |  |  |

Step 8:

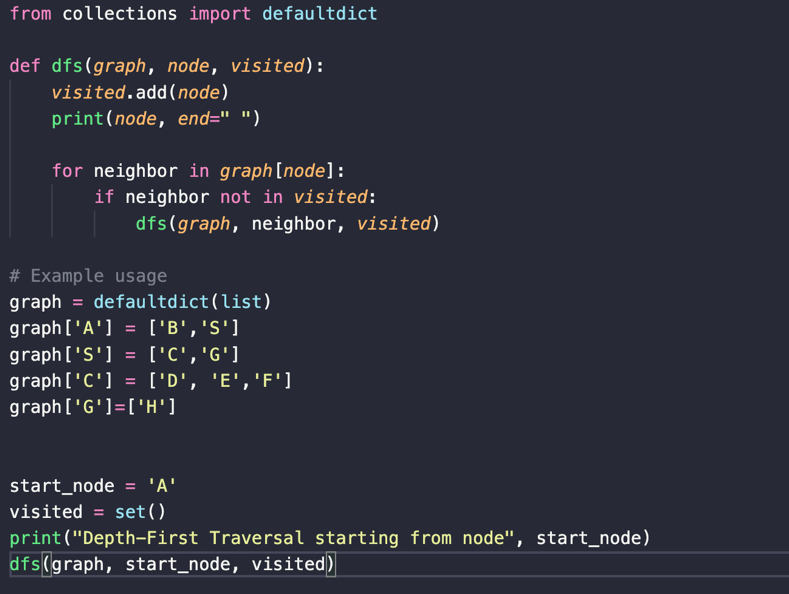
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A | B | S | C | D | E | F | G |
| Stack | H |  |  |  |  |  |  |  |

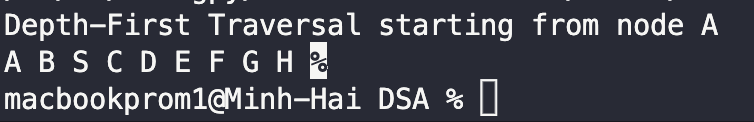
Step 9:

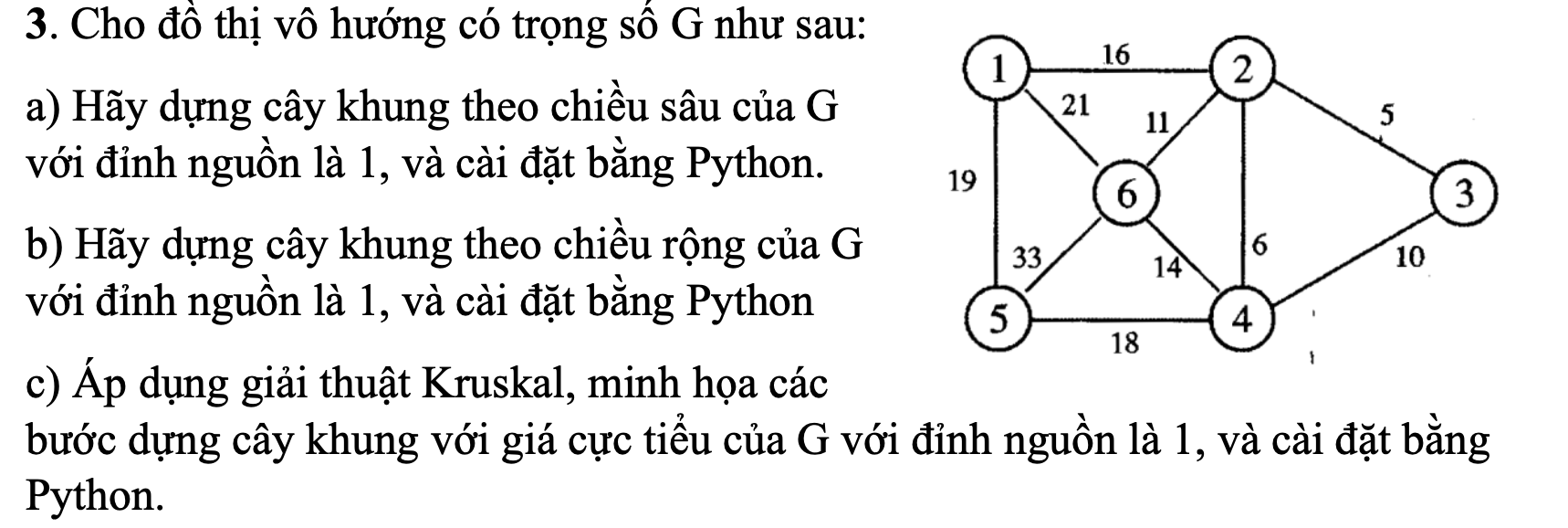
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Visited | A | B | S | C | D | E | F | G | H |
| Stack |  |  |  |  |  |  |  |  |  |

**Found a vertex H in the last step!**

C, Depth-First Traversal

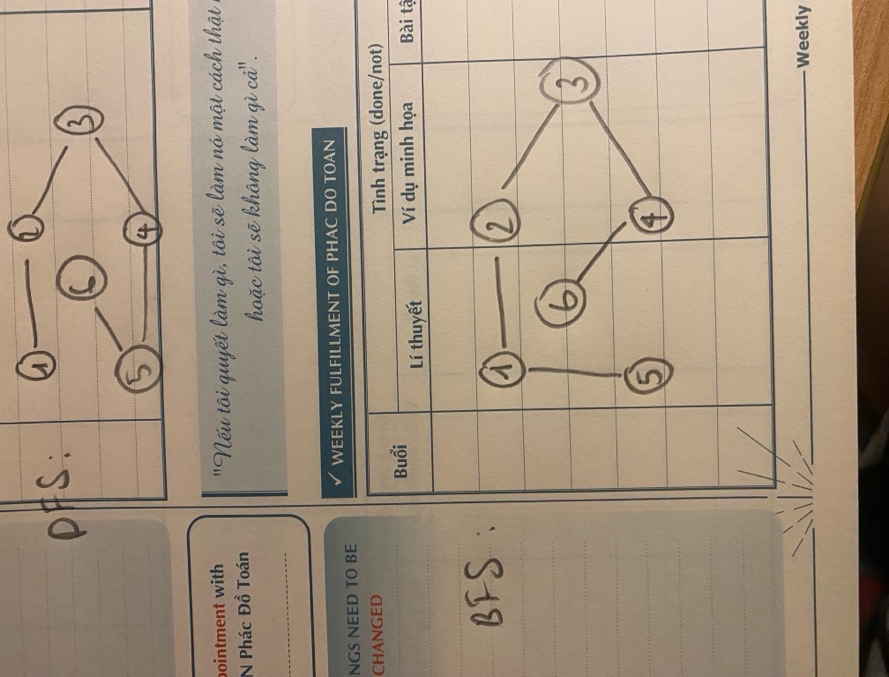
Python programing:





A, DFS Depth-First Spanning Tree of G graph with the source vertex is 1

B, BFS Breadth-First Spanning Tree of G graph with the source vertex is 1

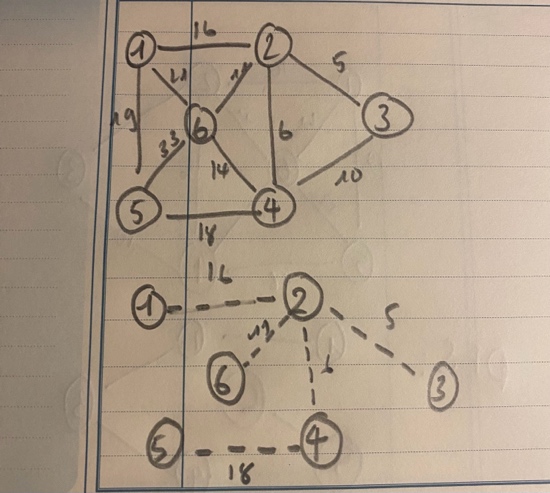


C,

Step 1: Selecting the first edge with the source weight in the spanning tree is 16

Step 2: Selecting edges with weights 5 and 6 in the spanning tree

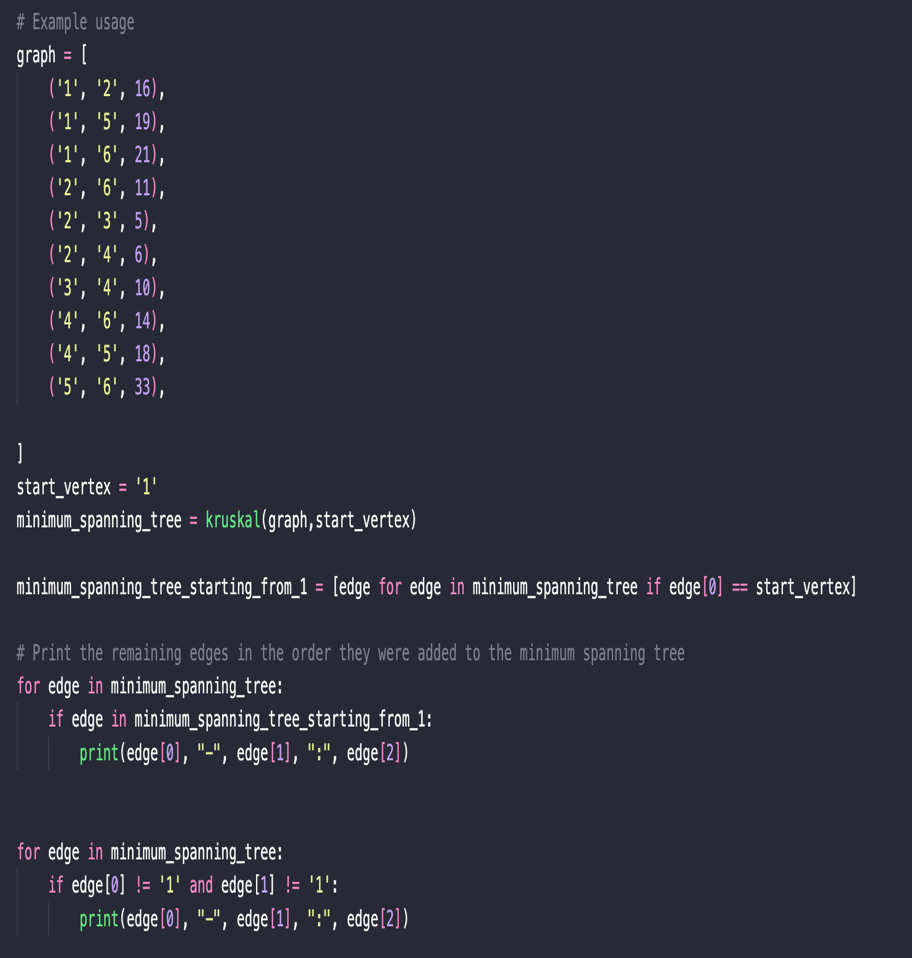
Step 3: Selecting edges with weights 11 and 18 in the spanning tree

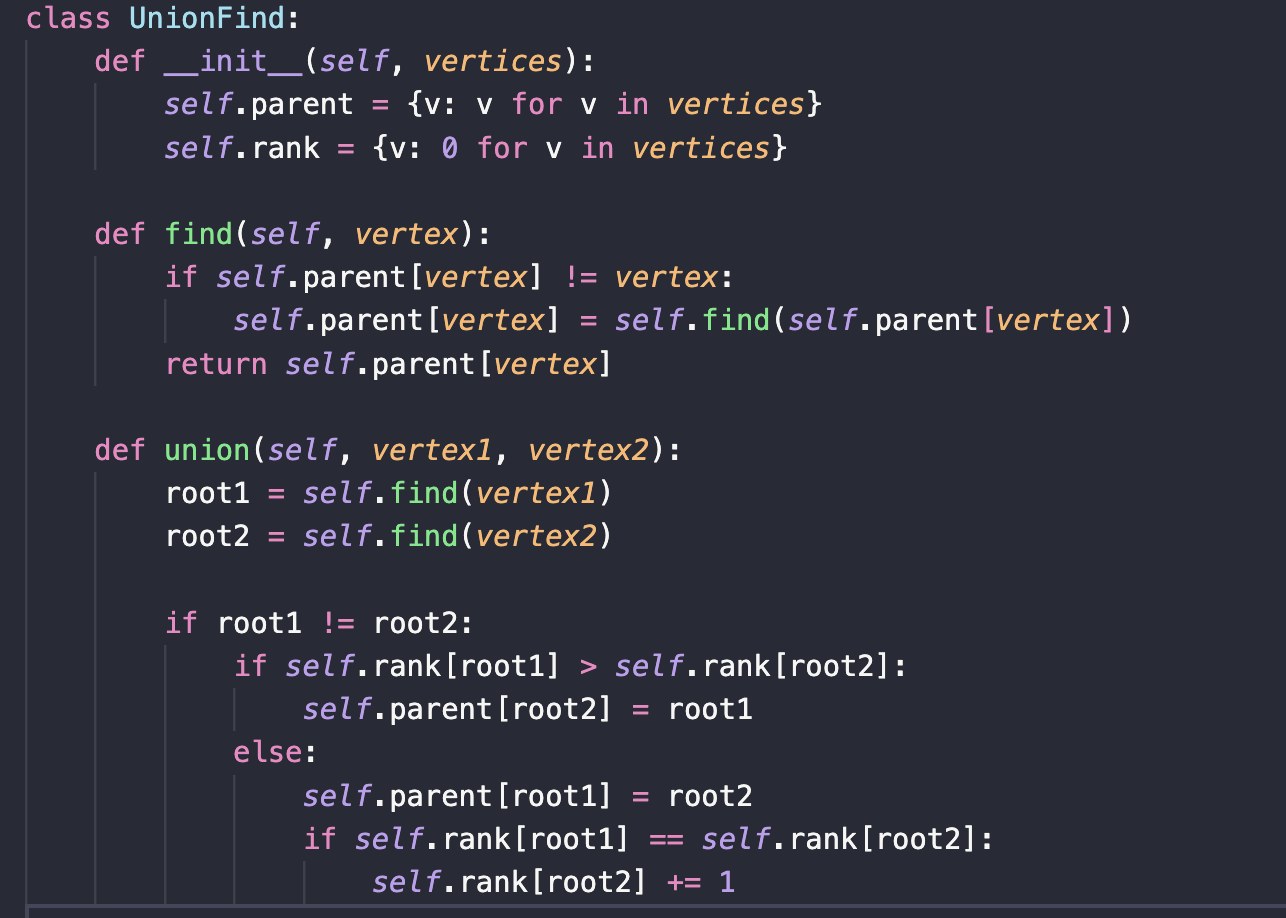


**The final spanning tree**

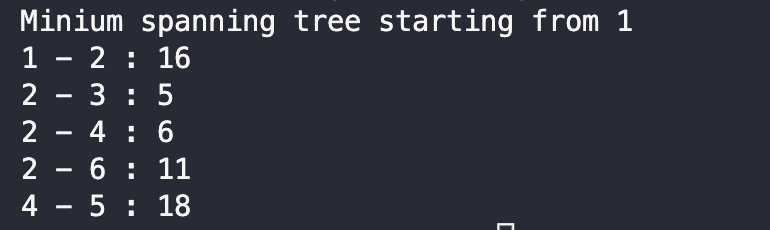
**created using**

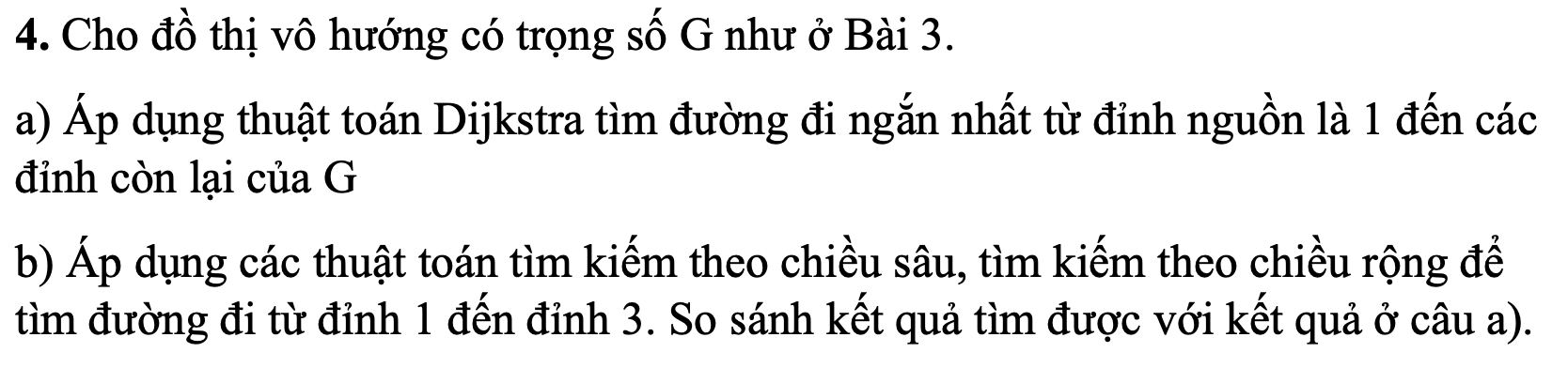
**Kruskal’s algorithm!**

Python programing:

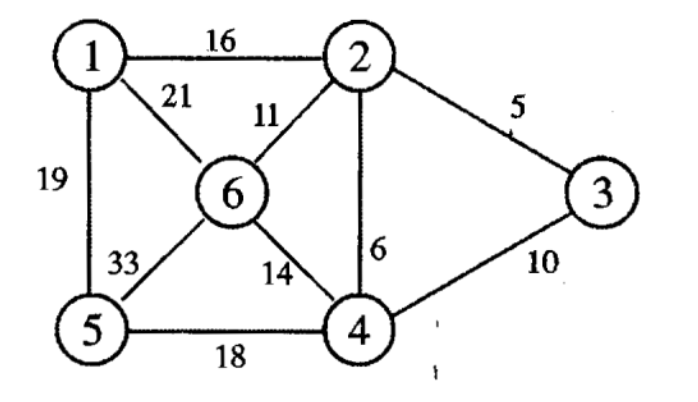








A,



dist = [0, ], visited = [0, 0, 0, 0,0,0]

1)

u = 1, v = 2, dist[2] = dist[1] + (1, 2) = 16 < dist[2] = , dist[2] = 16,

u = 1, v = 5, dist[5] = dist[1] + (1, 5) = 19 < dist[5] = ∞, dist[5] = 19,

u = 1, v = 6, dist[6] = dist[1] + (1, 6) = 21 < dist[6] = ∞, dist[6] = 21,

dist = [0, ], visited = [1, 0, 0, 0,0,0]

2)

u = 2, v = 3, dist[3] = dist[2] + (2,3) = 21< dist[3] = , dist[3] = 21,

u = 2, v = 4, dist[4] = dist[2] + (2,4) = 22< dist[4] = , dist[4] = 22,

u = 2, v = 6, dist[6] = dist[2] + (2,6) = 27> dist[6] =21 , dist[4] = 21,

dist = [0,16,21,22,19,21], visited = [1, 1, 0, 0,0,0]

3)

u = 3, v = 4, dist[4] = dist[3] + (3,4) = 31> dist[4] =22 , dist[3] = 22,

dist = [0,16,21,22,19,21], visited = [1, 1, 1, 0,0,0]

4)

u = 4, v = 6, dist[6] = dist[4] + (4,6) =36 > dist[6] =21 , dist[6] = 21,

u = 4, v = 5, dist[5] = dist[4] + (4,5) =40 > dist[5] =19 , dist[5] = 19,

dist = [0,16,21,22,19,21], visited = [1, 1, 1, 1,1,0]

5)

dist = [0,16,21,22,19,21], visited = [1, 1, 1, 1,1,0]

6)

dist = [0,16,21,22,19,21], visited = [1, 1, 1, 1,1,1]

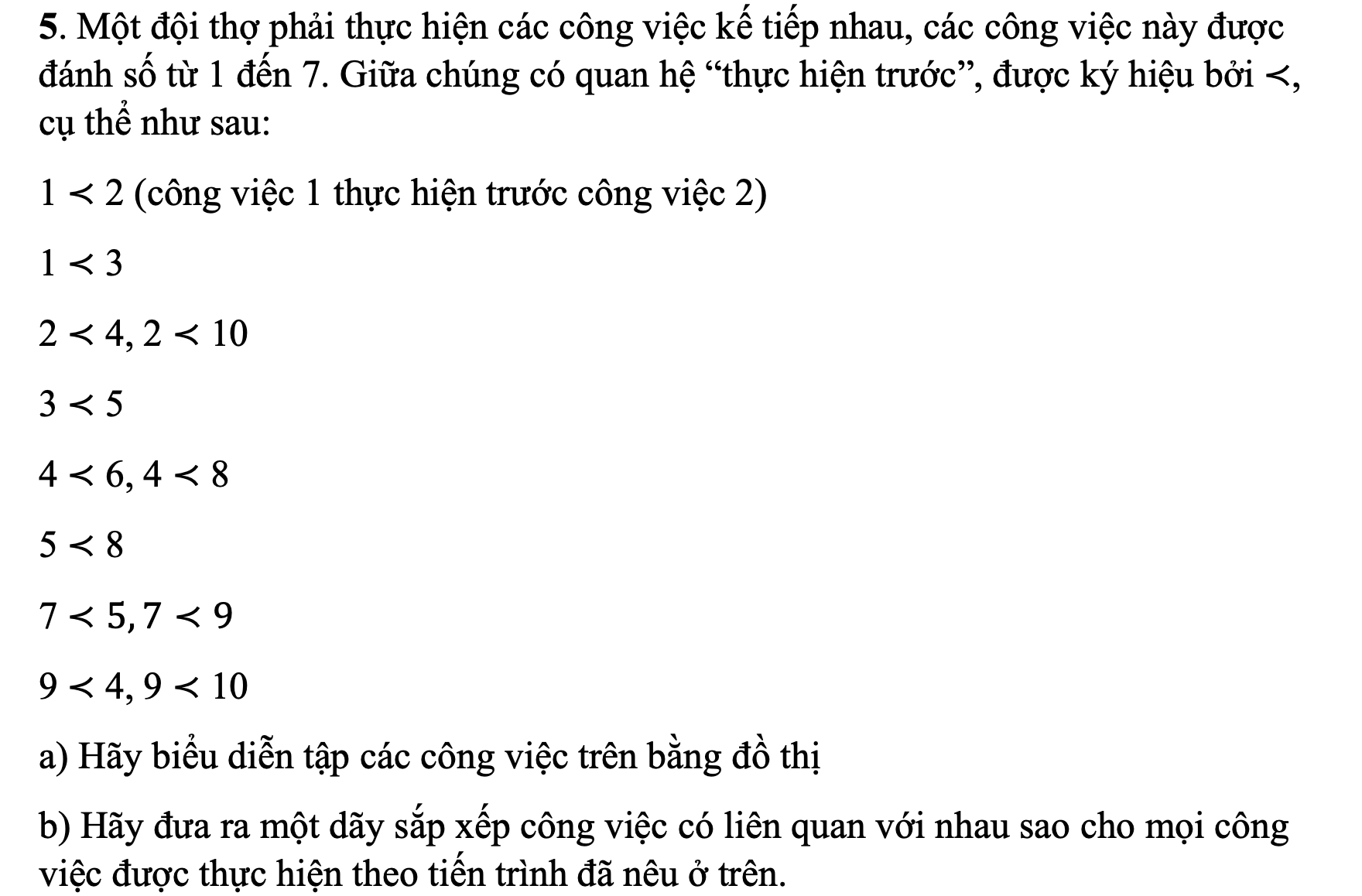
result a shortest path length from 1 vertex to others vertex is [0,16,21,22,19,21]

b,

DFS result: 1 -> 2 -> 3 -> 4 -> 6 -> 5

BFS result: 1 -> 5 -> 6 -> 2 -> 4 -> 3

***performing a DFS or BFS on the graph will produce a spanning tree, but neither of those algorithms takes edge weights into account***



A,

Graph representation:

B, a sequence of work arrangements: 1🡪2🡪3🡪7🡪9🡪4🡪5🡪6🡪8🡪10