Data Structures and Algorithms  
Week 5 problem sheet

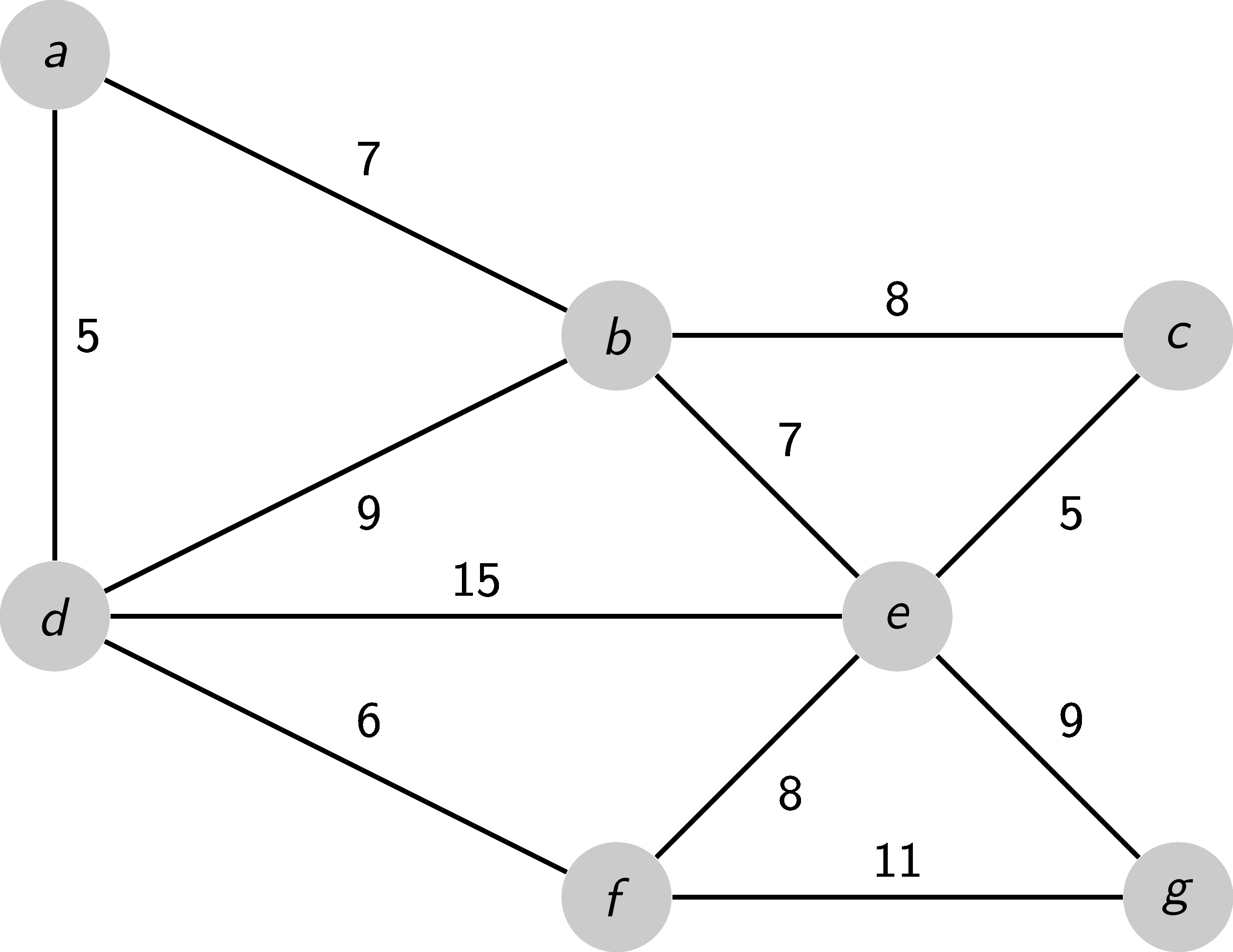
## A. Hash tables

1. What is a perfect hash function? Explain why it is almost always impossible to have a perfect hash function.

* A perfect hash function is defined as one which maps each input to a unique array position. However, in many cases, the number of possible inputs is theoretically infinite (e.g. “all possible strings”), or far larger than the array available to store them in. So a perfect hash function is therefore impossible.
* (If we have a very *small* number of items to store, then a perfect hash function does become more plausible; but for a small number of items, there is probably not much advantage in choosing a hash table over, say, just an array.)

## B. Graph algorithms

Use the example weighted, undirected graph below to answer the following questions:



1. Suppose we start from vertex – in what order are the vertices of graph visited if we do a depth-first search (DFS)?

* **Solution**
* We assume that neighbours are visited counter-clockwise from the “top” (since we don’t have a concept of “left” or “right” here).
* The order is: b, a, d, f, g, e, c.

1. Suppose we start from vertex – in what order are the vertices of graph visited if we do a breadth-first search (BFS)?

* **Solution**
* Within a “layer”, we assume that neighbours are visited counter-clockwise from the “top” (since we don’t have a concept of “left” or “right” here).
* The order will be: b, a, d, e, f, g, c.

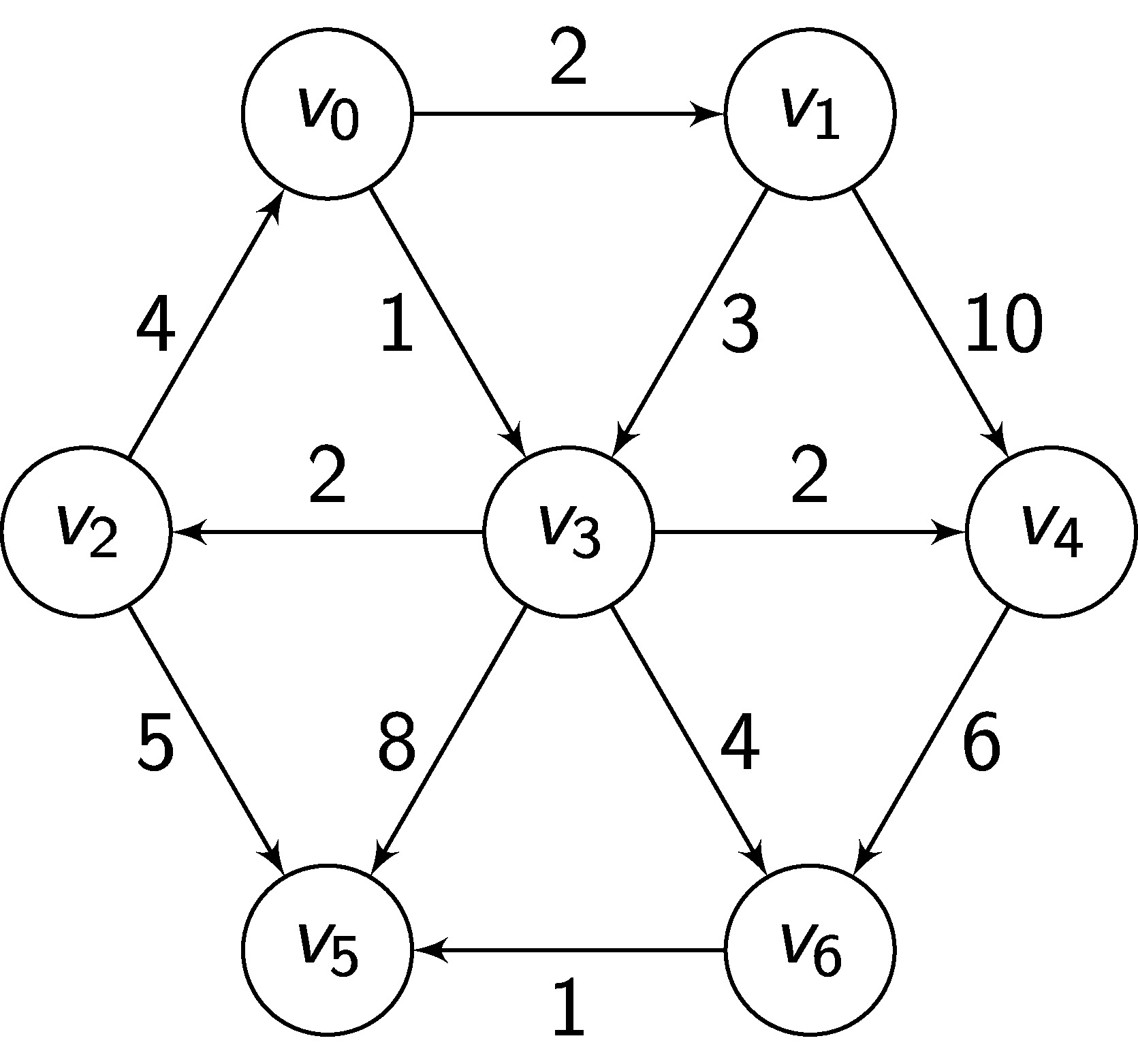
1. Suppose we wish to find the shortest path from vertex to vertex in graph . If we ignore the edge weights, what would your answers be (hint: there is more than one possible answer)?

* **Solution**
* Possible routes are:
  + a, d, f, g
  + a, d, e, g
  + a, b, e, g

1. Suppose we wish to find the shortest path from vertex to vertex in graph . If we calculate a shortest path *using* the edge weights, what would your answer be?

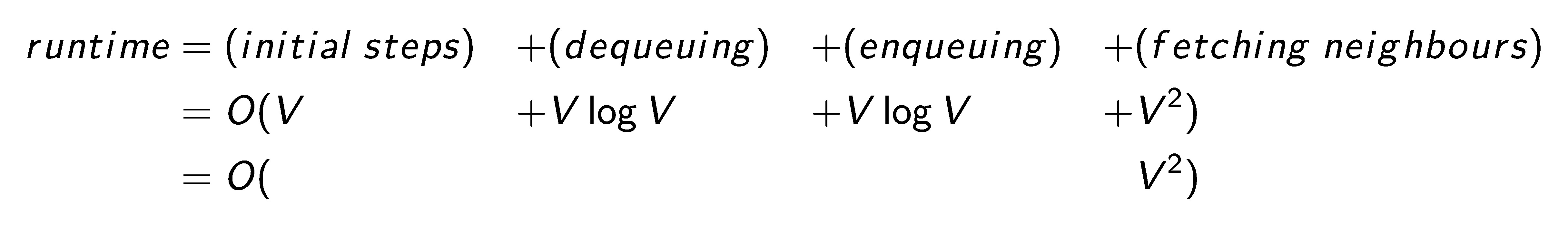
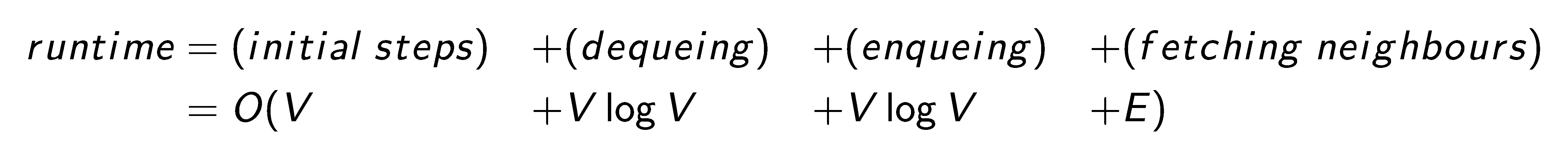
* **Solution**
* The shortest route is:
  + a, d, f, g
* The totel weight is 5 (a to d), plus 6 (d to f), plus 11 (f to g), for a total of 22.

1. On paper, show the steps followed by Prim’s method for calculating a **minimum spanning tree** for the graph shown below. Start your tree from vertex . You could write some code using PrimMST.java to check your answer.

* 
* For this question, it is suggested you check your answer by writing code using   PrimMST.java.

1. PrimMST can be implemented using either an **adjacency matrix** or an **adjacency list** to represent a weighted graph. What difference does the choice of implementation make to the big O complexity of the algorithm? Why?

* Recall the pseudocode for Prim’s algorithm:

* 1: inital step:  
   2: - for all vertices v ∈ V:  
   3: colour[v] = white  
   4: key[v] = ∞  
   5: π[v] = undefined  
   6:  
   7: pick some starting vertex s ∈ V, enqueue it into Q  
   8: key[s] = 0  
   9: colour[s] = grey  
   10:  
   11: while Q is not empty:  
   12: w ← extract-min from Q  
   13: for each v ∈ neighbours(w):  
   14: if colour[v] is white:  
   15: colour[v] = grey  
   16: key[v] = edgeWeight(w,v)  
   17: π[v] = w  
   18: enqueue v into Q using key[v] as priority  
   19: else if colour[v] is grey:  
   20: if key[v] > edgeWeight(w,v):  
   21: key[v] = edgeWeight(w,v)  
   22: π[v] = w  
   23: colour[w] = black
* For an adjacency matrix implementation, we can analyse the runtime of the algorithm as follows.
* - The *initial steps* (lines 1–9) are run times.
* - At some point in the algorithm, every node in the graph will be *dequeued* (line 12). Dequeueing from a Java priority queue takes time (see the [API documentation](https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html)), so the total cost of dequeueing operations will be .
* - Note the code for *enqeueing* a node (line 18). How many times will this happen? Not every time around the while loop, because line 18 is in an if statement.  
  What we do know is that at some point in the algorithm, every node in the graph will be enqeued. So the number of times it happens is ; and since the costs of enqueueing is , that means the total cost across the whole algorithm of enqueueing operations will be .
* - We also have to fetch the list of neighbours (line 13) for a node. For an adjacency matrix representation, the cost of doing so is ; it is done once for every iteration of the while loop, so the total cost is .
* So, the running time will be
* 
* For an adjacency *list* representation, we analyse the runtime as follows.
* - The *initial steps* (lines 1–9) are still run times.
* - At some point in the algorithm, every node in the graph will still be *dequeued* (line 12). As before, the total cost of dequeueing operations will be .
* - The cost for enqeueing nodes likewise remains the same, .
* - However, fetching the list of neighbours for a node (line 13) will be different. The worst-case cost of doing so for for an adjacency list is , where is the number of edges in the graph.  
  Does this mean the cost of fetching neighbours is , since the while loop runs times? Not exactly. Different nodes will have different numbers of edges, so the cost will vary – they can’t *all* have -many edges. What we do know is, every edge will be retrieved *once*. So the *total* cost of retreiving edges, across the whole algorithm, will just be .  
  Similarly, any operations in the “for each neighbour” loop will run times.  
  So the total cost of these operations will be .
* That means the grand total cost of the algorithm for an adjacency list representation is:
* 
* Which of these terms will predominate? We don’t know: it depends on how *dense* the graph is – what proportion of nodes have edges between them.  
  In a very *dense* graph, the number of edges will be close to – every node is connected to every node – so , and the running time won’t differ from an adjacency matrix representation.
* However, in a very *sparse* (but still connected) graph, the number of edges can be as low as – just one edge between every node.
* In such a case, the term would predominate, and the total running time for the algorithm would be .
* So the answer is, we need to know how dense the graph is to know which terms will predominate; depending on the density, the runtime can be anywhere from to .

1. Test your answer to the previous question by implementing an adjacency list version of the algorithm. You can use PrimMST.java as the basis of your second solution. Run some experiments to test the run times of these two algorithms. What do you notice?

* For this question, it is suggested you check your answer by writing code using   PrimMST.java.