Solver: Chew Jing Wei

Email Address: jchew029@e.ntu.edu.sg

1a)

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{10^n}{n!}$$

$$= \lim_{n\to\infty}\frac{10^{10}}{10!} * \frac{10*10*10*\dots}{11*12*13*\dots}$$

$$= 0$$

Hence, f is in O(g), f is NOT in Ω(g), and f is NOT in Θ(g).

1b)

0	
1	23
2	
3	
4	15
5	
6	
7	7
8	34
9	29
10	

Insertion:

7 mod 11 = 7. 7 is empty. 7 is hashed to index 7.

23 mod 11 = 1. 1 is empty. 23 is hashed to index 1.

29 mod 11 = 7. 7 is occupied. d = 29 mod 7 + 1 = 2. rehash(j,d) = 7+2 mod 11 = 9. 9 is empty. 29 is hashed to index 9.

15 mod 11 = 4. 4 is empty. 15 is hashed to index 4.

34 mod 11 = 1. 1 is occupied. d = 34 mod 7 + 1 = 7. rehash(j,d) = 1+7 mod 11 = 8. 8 is empty. 34 is hashed to index 8.

Searching:

Note: Those that did not collide when inserting only require 1 key comparison, while those who collided require 1 + number of collisions.

Key values	Key comparisons
7	1
23	1
29	2
15	1
34	2

2a)

Initial array: [10, 3, 5, 7, 4]

5 becomes the pivot: [5, 3, 10, 7, 4]

3 is lower than 5: [5, <u>3</u>, 10, 7, 4], 3 is the last lower than 5 value.

10 is higher than 5: [5, <u>3</u>, 10, 7, 4], 3 is the last lower than 5 value.

7 is higher than 5: [5, <u>3</u>, 10, 7, 4], 3 is the last lower than 5 value.

4 is lower than 5, 4 is placed after 3: [5, 3, <u>4</u>, 7, 10], 4 is the last lower than 5 value.

Finally, 5 is swapped back to the last 'lower than' value, 4.

Output array: **[4, 3, 5, 7, 10]**

2b)

The idea is to heapify the array.

Unlike a normal fixHeap to call, we want to create a minimizing partial order tree in this case.

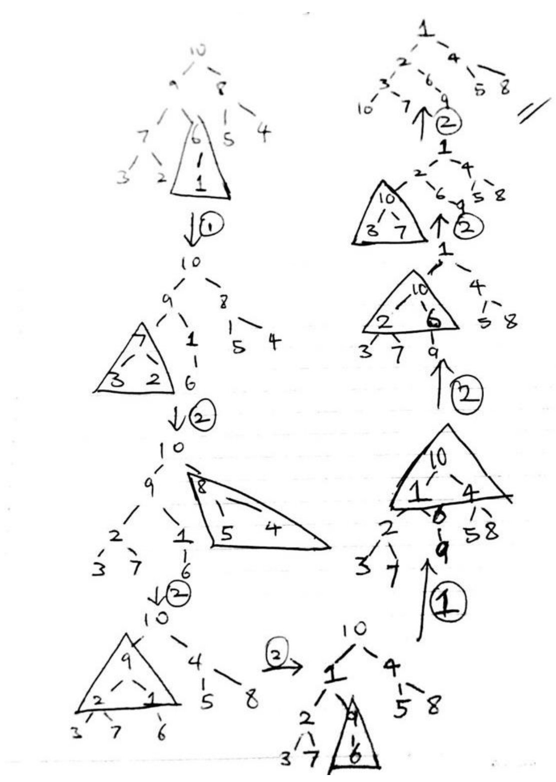
Original array:

10	9	8	7	6	5	4	3	2	1

Heapify calls (on non-leaf subheaps), where the element in bold is the subheap root:

1. [**10**, 9, 8]
2. [**9**, 7, 6]
3. [**8**, 5, 4]
4. [**7**, 3, 2]
5. [**6**, 1]

Since we are doing post-order traversal, the rest of each heapify call is then executed in LIFO order.



Total number of key comparisons = 14

3a)

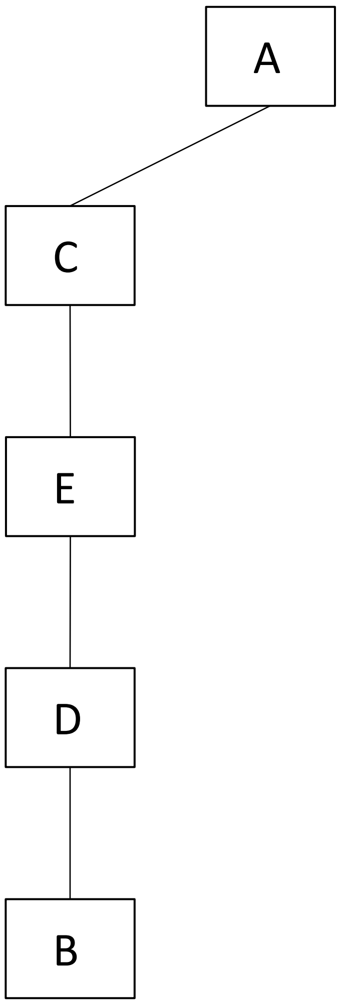
Matrix:

	A	B	C	D	E
A	-	5	1	-	-
B	-	-	-	-	-
C	-	4	-	-	-3
D	-	1	2	-	-
E	-	-	-	-2	-

List:

A	(C, 1)	(B, 5)
B		
C	(E, -3)	(B, 4)
D	(B, 1)	(C, 2)
E	(D, -2)	

3b) (i)



3b) (ii) A list is more efficient in general because of reduced space complexity. For DFS on weighted edges, it is more efficient than a matrix because you can start with sorted adjacency lists, and since DFS picks the first unvisited neighbour of each node, the resultant path generated will be guaranteed to have the minimum cost as well.

3c) (amended on November 2019) (i) (Next node is bolded)

Iteration 0:

	S	D	pi
A	0	0	-
B	0	9999	-
C	0	9999	-
D	0	9999	-
E	0	9999	-

Iteration 1:

	S	D	Pi
A	1	0	-
B	0	5	A
C	0	1	A
D	0	9999	-
E	0	9999	-

Iteration 2:

	S	D	Pi
A	1	0	-
B	0	5	A
C	1	1	A
D	0	9999	-
E	0	-2	C

Iteration 3:

	S	D	Pi
A	1	0	-
B	0	5	A
C	1	1	A
D	0	-4	E
E	1	-2	C

Iteration 4:

	S	D	Pi
A	1	0	-
B	0	-3	D
C	1	1	A
D	1	-4	E
E	1	-2	C

Iteration 5 (end):

	S	D	Pi
A	1	0	-
B	1	-3	D
C	1	1	A
D	1	-4	E
E	1	-2	C

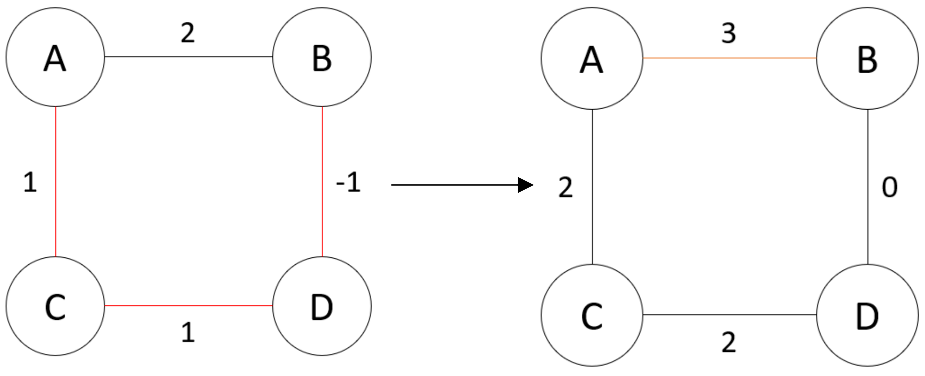
So, the computed shortest paths are:

- (A->B): (A->C)->(C->E)->(E->D)->(D->B)
- (A->C): (A->C)
- (A->D): (A->C)->(C->E)->(E->D)
- (A->E): (A->C)->(C->E)

3c (ii) (amended on November 2019) A->C is not the shortest path. It should instead be a cycle like this: A->C->E->D->C where the total weight will be -2 instead of 1 which it currently is. As such, other routes which are based on A->C are no longer the shortest path either.

3c (iii) It violates the shortest path property because the path formed by Dijkstra's algorithm is no longer the shortest path.

3c (iv) No. The node's weight may increase similarly but the path's weight will increase proportional to the number of nodes involved. One of the counter examples is illustrated below. We may observe that due to the addition (by 1) in every node, the shortest path changes.



4a) Is there a simple path that is longer than length k in a graph?

4b) (https://cs.stackexchange.com/questions/12969/how-to-show-that-problems-are-in-np)

To find any simple path of maximum length, some kind of depth-first search has to be used, which is a non-polynomial solving algorithm. However, to check that a given path is simple and of maximum length only requires a checking function of polynomial time complexity.

4c) The TSP is NP-hard, and the TSP is merely a variant of the LP because the length of a solution in TSP is equal to the maximum length in a graph, which is the goal of LP as well. Therefore, LP is NP-hard as well.

4d) Since LP is both in NP-hard and NP, it is NP-Complete.

Please contact me if you suspect any problems with the solution set, all the best for your exams!