Solver: Callista Rossary Chang

1ai) $\Theta(\lg^2(N))$

Notes: The outer for loop will execute $\lceil \log_3(N) \rceil$ times, while the inner loop will execute $\lceil \lg(N) \rceil$ times. Hence, the print statement will execute $\lceil \log_3(N) \rceil \lceil \lg(N) \rceil$ times. In big-Θ notation, this can be represented as $\Theta(\lg^2(N))$.

1aii) $\Theta(2^N)$

Notes: Firstly, craft the recurrence relation for function qaII. It is $T_N = 2T_{N-1} + N$, where $T_0 = 0$. The final answer to the relation is $T_N = 2^{N+1} - N - 2$. For in-depth solutions to the above recurrence relation, see https://math.stackexchange.com/q/239974.

Since the relation may be difficult and time-consuming to solve, and it is only necessary to find the dominant term for the big-Θ notation, the dominant term 2^N can be inferred by drawing similarities to the recurrence relation $T_N = 2T_{N-1}$.

1b)

```
int matrixBinarySearch(int[][] matrix, int n, int key) {
      int first = 0;
      int last = n-1;

      // perform iterative binary search on each matrix row
      for (int i = 0; i < n; i++) {
            while (first <= last) {
                  int mid = (first + last) / 2;

                  if (matrix[i][mid] == key)
                        return 1; // element found
                  else if (matrix[i][mid] < key) // key is larger
                        first = mid + 1; // consider right subarray
                  else // key is smaller
                        last = mid - 1; // consider left subarray
            }
      }
      return 0; // element not found
}
```

Notes: There are different ways to do this question.

1bii) matrixBinarySearch(matrix, 4, 15);

i	Loop 1	Loop 2	Loop 3	Loop 4
0	mid = (0+3)/2 = 1 matrix[0][1] = 4 Since 4 < 15, first = mid+1 = 2	mid = (2+3)/2 = 2 matrix[0][2] = 6 Since 6 < 15, first = mid+1 = 3	mid = (3+3)/2 = 3 matrix[0][3] = 11 Since 11 < 15, first = mid+1 = 4	Since first > last, it exits the while loop.
1	mid = (0+3)/2 = 1 matrix[1][1] = 5 Since 5 < 15, first = mid+1 = 2	mid = (2+3)/2 = 2 matrix[1][2] = 8 Since 8 < 15, first = mid+1 = 3	mid = (3+3)/2 = 3 matrix[1][3] = 14 Since 14 < 15, first = mid+1 = 4	Since first > last, it exits the while loop.
2	mid = (0+3)/2 = 1 matrix[2][1] = 7 Since 7 < 15, first = mid+1 = 2	mid = (2+3)/2 = 2 matrix[2][2] = 12 Since 12 < 15, first = mid+1 = 3	mid = (3+3)/2 = 3 matrix[2][3] = 15 Since 15 == 15, **the key has been found**. The function returns 1.	

1biii) Best case: Ω(1)
Average case: Θ(nlg(n))
Worst case: O(nlg(n))

The best case happens when the element to be found is the first to be found – that is, in the first row and middle column of the matrix (1 key comparison). The worst case happens when the element cannot be found in the matrix (nlg(n) key comparisons).

2a) Number of key comparisons = 13
Number of swaps = 10

Notes:

Iteration	Initial Array	No. of Key Comps	No. of Swaps
1	[7, **5**, 13, 9, 5, 2, 10]	1 (5 <= 7 → swap)	1
2	[5, 7, **13**, 9, 5, 2, 10]	1 (13 > 7 → no swap)	0
3	[5, 7, 13, **9**, 5, 2, 10]	2 (9 <= 13 → swap 9 > 7 → no swap)	1
4	[5, 7, 9, 13, **5**, 2, 10]	4 (5 <= 13 → swap 5 <= 9 → swap 5 <= 7 → swap 5 <= 5 → no swap)	3
5	[5, 5, 7, 9, 13, **2**, 10]	5 (2 <= 13 → swap 2 <= 9 → swap 2 <= 7 → swap 2 <= 5 → swap 2 <= 5 → swap)	5
6	[**2**, 5, 5, 7, 9, 13, 10] * 2 is at the first position.	**Total: 1+1+2+4+5 = 13**	**Total: 1+1+3+5 = 10**

2b) A descending array is a best case input of Mergesort. Consider the input array [8, 7, 6, 5, 4, 3, 2, 1]:

Step	Arrays	Comparisons Made
1	[8] [7] [6] [5] [4] [3] [2] [1]	7 vs. 8 5 vs. 6 3 vs. 4 1 vs. 2 Total: 4 comparisons
2	[7, 8] [5, 6] [3, 4] [1, 2]	7 vs. 5, 7 vs. 6 3 vs. 1, 3 vs. 2 Total: 4 comparisons
3	[5, 6, 7, 8] [1, 2, 3, 4]	5 vs. 1, 5 vs. 2, 5 vs. 3, 5 vs. 4 Total: 4 comparisons
4	[1, 2, 3, 4, 5, 6, 7, 8]	

During merge operations, the minimum value in the left subarray is always larger than the maximum value in the right subarray. Hence, only the minimum value in the left subarray will be compared with the values in the right subarray. When all the values in the right subarray have been compared, the remaining values in the left subarray are appended to the merged array. This leads to the minimum number of key comparisons made.

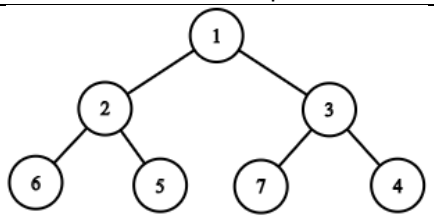
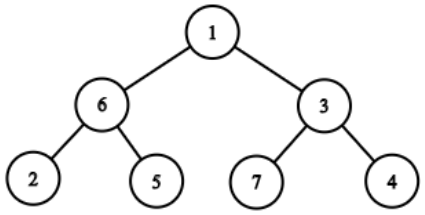
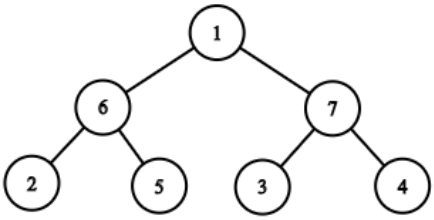
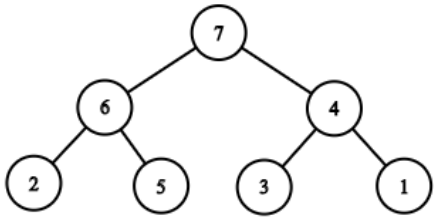
2c) 5 <= x < 10

Notes: The value of last_small must be the leftmost or rightmost index after every call. In other words, the pivots chosen must always either be the smallest or largest element in the (sub)array.

Call	Arrays	Observation
1	Initial array: [10, 5, **3**, 9, x] After swapping pivot: [**3**, 5, 10, 9, x]	For 3 to be the smallest element in the array, x must be larger or equals to 3. **x >= 3**
2	Right subarray: [5, **10**, 9, x] After swapping pivot: [**10**, 5, 9, x]	For 10 to be the largest element in the array, x must be smaller than 10. **x < 10**
3	Left subarray: [x, **5**, 9] After swapping pivot: [**5**, x, 9]	For 5 to be the smallest element in the array, x must be larger or equals to 5. **x >= 5**
4	Right subarray: [**x**, 9]	Since there are only 2 elements in the subarray, the value of x does not matter (x will always either be the largest or smallest element in the array).

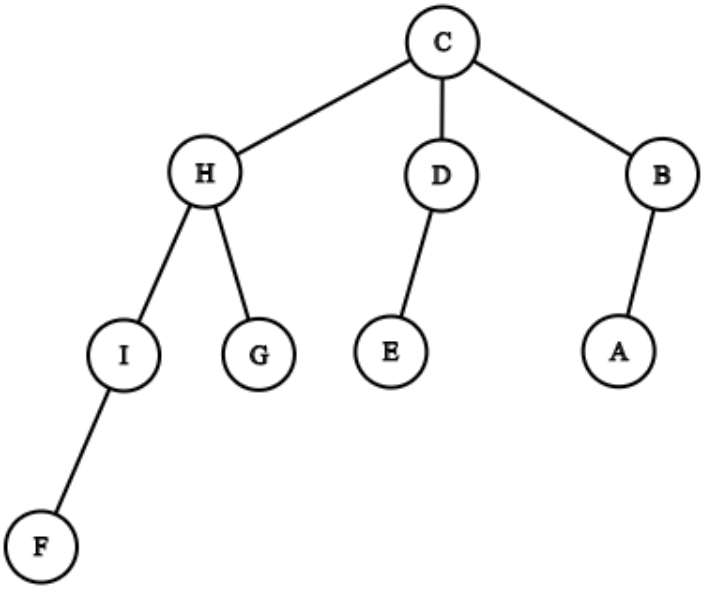
2di) [7, 6, 4, 2, 5, 3, 1]

Notes:

Initial Heap	1st fixHeap call
	
2nd fixHeap call	**3rd fixHeap call**
	

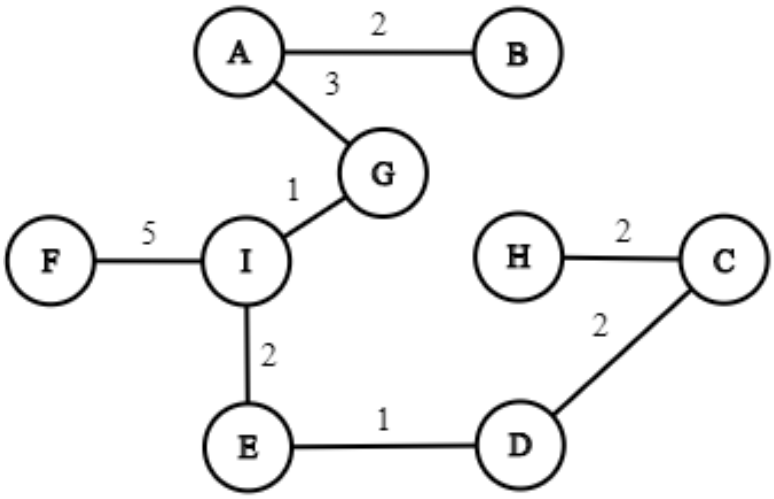
2dii) O(lg(n)). A heap is a nearly complete binary tree. A complete binary tree of k levels has $2^k - 1$ nodes. Hence, the height of a heap with n nodes and k levels is O(lg(n)). Since each call of fixHeap moves down a level, the time complexity of fixHeap is O(lg(n)). *[From slides]*

3a)



BFS on G, where C is the starting vertex and neighbors are visited in reverse alphabetic order

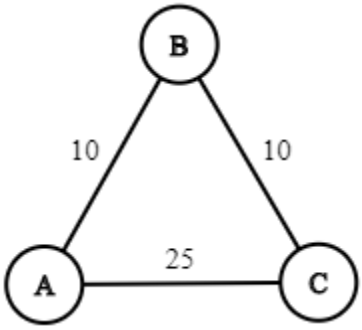
3b)



Prim's algorithm on G, where A is the starting vertex

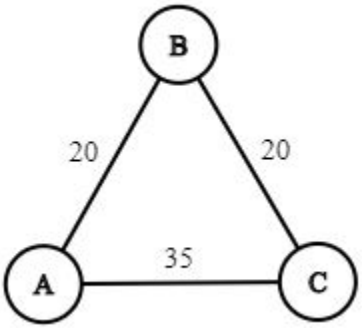
Notes: The order in which the edges are added onto the minimum spanning tree is **AB → AG → GI → IE → ED → DC → CH → IF**.

3c) No. In the following counterexample, let A be the source vertex and C be the target vertex:



Before adding 10 to the weights of every edge

Currently, the path with the lowest cost in G is AB → AC (20 as opposed to 25).



After adding 10 to the weights of every edge

The path with the lowest cost becomes AC (35 as opposed to 40). Hence, path P may not remain the shortest path when the weight of every edge in G is increased by 10.

4ai) Sequence of edges chosen (using the shortest-link algorithm):

1. AD (3)
2. FE (5)
3. EC (7)
4. DB (8) // DC (9) is skipped as it would be the 3rd edge incident on D
5. BF (10)
6. CA (12) // complete the cycle

Total distance = 3+5+7+8+10+12 = 45

Notes: Order the edges by weight in ascending order: 3, 5, 7, 8, 9, 10, 11, 12, 13, 15.

4aii) Sequence of edges chosen (using the nearest neighbor algorithm, starting at node A):

1. AD (3)
2. DB (8)
3. BF (10)
4. FE (5)
5. EC (7)
6. CA (12) // complete the cycle

Total distance = 3+8+10+5+7+12 = 45

4b) Given an undirected graph G and a natural number k, can G be colored with more than k colors such that each node is assigned a different color from all its neighbors?

4c) *Preliminary notes:*

- A **vertex subset** is a subset of vertices in a graph G.
- An **induced subgraph** is a vertex subset together with any edges whose endpoints are both in this subset.
- A **clique** is a vertex subset such that every 2 distinct vertices in the subset are adjacent; that is, an induced subgraph that is a complete graph.
- Hence, the decision problem version of the **clique problem** is – "Given an undirected graph G and an integer k, does G have a vertex subset of at least size k whose induced subgraph is complete?"

https://en.wikipedia.org/wiki/Clique_problem is a great resource for this question, because it includes a visual for the brute force algorithm on a 7-vertex graph where k = 4.

4ci) Number of vertex subsets $= {}^{7}C_{4}$
$= 7! / ((7-4)! * 4!)$
$= 35$

4cii) *"Given an undirected graph G and an integer k, does G have a clique (i.e. complete induced subgraph) of at least size k?"*

```
int bruteForceCliqueProblem(Graph G, int k) {
      // generate induced subgraphs of at least size k
      for (int i = k; i <= G.size(); i++) {
            Graph[] subgraphs = G.getCombinations(k); // given function
            for (s in subgraphs) {
                  // check if the induced subgraph is complete
                  if (s has k(k-1)/2 edges && all vertices in s have k-1
                  edges each) return 1; // clique found
            }
      }
      return 0; // clique not found
}
```

4ciii) $O(n^k k^2)$. This is because there are $O(n^k)$ subgraphs to check, each of which has $O(k^2)$ edges whose presence in G needs to be checked. *[From Wikipedia]*

Notes:

- There are $O(n^k)$ subgraphs to check because when k is constant, $O({}^nC_k) = O(n! / ((n-k)!*k!)) = O(n^k)$. See https://math.stackexchange.com/questions/1265519/ for proof.
- There are $O(k^2)$ edges to check because the number of edges in a complete graph is k(k-1)/2.

– END OF ANSWERS –
All the best for your exams! :>