Name: Chloe Hu
Student ID: 21044009
Email: chuap@connect.ust.hk

QUESTION 1:

a)

Creating the Data Structure:
Description:

The proposed data structure is an AVL Tree that orders the array into a balanced tree structure to ensure that in the worst case, the range query is found in O(logn) time. The AVL tree takes up O(n) space as it rearranges the array, not adding extra data. The data structure has logn levels and n elements to insert, thus the running time to create an AVL is O(nlogn) since in the worst case, the value being inserted to the tree is a leaf node every insertion. The algorithm to generate the AVL tree is as follows:

Pseudocode:

```
createAVL(A):
        node* root = null;

        for all A[i]
                root = insert(root, A[i]);

        return root;

insert(node, value):
        if (node = null)
                return value;
        else if (value < root-> value)
                root->left = insert(root->left, value);
        else if (value >= root->value)                  // allow duplicates
                root->right = insert(root->right, value);

        checkBalance(root)          // checks if the tree is balanced and rotates accordingly)
```

Finding the range query:
Description:

To use array C to find the range query, we would first check if the root node is greater than Y. If the root node is greater than Y we would recurse on root->left. Likewise if the node is smaller X we would recurse on root->right. When we reach a number in between X and Y (including Y but not X) we increment the count by 1 and recurse on both root->left and root->right. As a result we

will find the number of occurrences of numbers (X, Y]. The base case is when root = null.

Finding the range query takes O(logn) time due to the balanced structure of the AVL tree where each iteration will decrease the sample space by half.

Pseudocode:

```
count = 0;
findRangeQuery(root, X, Y, count):
        if root = null
                return 0;

        if (root->value > Y)
                return findRangeQuery(root->left, X, Y, count);
        else if (root->value <= X) {
                return findRangeQuery(root->right, X, Y, count);
        else
                count++
                count += findRangeQuery(root->left, X, Y, count);
                count += findRangeQuery(root->right, X, Y, count);
                return count;
```

b)

Creating the Data Structure:
Description:

The proposed data structure is an array that holds the cumulative sums from all numbers smaller than or equal to the current number.

First we initialise array C of size K with all indexes to 0. Then we loop through array A and add the corresponding number to the index of C.

Finally, we will loop through C, cumulatively adding the sums of all numbers smaller than or equal to the current number.

The run time for constructing C is O(K) time since it must initialise an array of size K with all 0's (K time), then loop through A and add the elements of A to C (N time), then cumulatively sum C (K time).

Pseudocode:

```
createCumulativeArray(A[n], C[k], K, N):

    for i = 0 to K
        C[i] = 0

    for i = 1 to N
        C[A[i]] = A[i]

    for i = 2 to K
        C[i] += C[i -1]
```

Finding the sum query:

Description:

To use C to find the sum query we compute C[b] - C[a] since C is an ordered array from 1 to K. This is a single subtraction and would take O(1) time.

Pseudocode;

```
findSumQuery(C[k], a, b){
    Return C[b] - C[a]
```

QUESTION 2:

The devices will create a binary tree of switches to generate any n! permutation. This works due to the fact that each input/output is a single bit of value 0 or 1 represented in the tree by a right/left split. Furthermore, a binary tree has n leaves and a height of logn, as a result the maximum amount of switches needed is nlogn.

The devices will divide the n inputs into two halves of n/2 each until it reaches the base case of n = 2 switches. When it reaches this case it will generate 2! possible permutations.

QUESTION 3:

Description:

To find the largest and second largest elements in the array, we first find the largest element of the array using binary search. At the same time, we add all elements that were compared to the largest to a “compared” array (referred to as C). To find the largest, the number of comparisons carried out is $(n - 1)$ since each number is compared once, except for the maximum which is not compared again.

To find the second largest element in the array, we use C generated from the first call. The elements of C are all the elements compared to the largest value found in the first call of the helper function. As a result, the second largest value must be inside C. Due to the binary search in the algorithm, the size of C is $logn$. Thus, the number of comparisons to find the second largest element is $(logn - 1)$ as there are $logn$ elements and the (second) largest value is not compared again.

In conclusion, the total number of comparisons is as follows:

$$= n - 1 + logn - 1$$
$$= n + logn - 2$$

Pseudocode:

```
// main algorithm, the largest is stored in finalArray[0] and the second largest is stored in
// finalArray[1]

findMax(A):
        finalArray[2]

        max1 = findMaxHelper(A)
        max2 = findMaxHelper(max1[2..logn])        // call the helper function on index 2 to logn

        finalArray[0] = max1[1]
        finalArray[1] = max2[1]
        return finalArray

// Helper function that returns an array where the 0th element is the number of comparisons,
// the 1st element is the largest and the remaining elements are all the elements compared

findMaxHelper(start, end, A):
        if (start = end)          // base case
                C[0] = 1          // increment the count of comparisons
                C[1] = A[i]       // add A[i] to the list of numbers compared with
                return C
```

```
C1= findMax(start, end/2, A)          // binary search on left and right halves
C2= findMax(end/2 + 1, end, A)

if (C1[1] > C2[1])
        C1[0]++                       // increment the count of comparisons
        C1[C1[0]] = C2[1]             // add to compared list
        return C1
else
        C2[0]++
        C2[C2[0]] = C1[1]
        return C2
```

QUESTION 4:

We will prove that Random-Sample produces a subset of size m by an induction on m.
The base case is m = 0, which returns an empty set.
Suppose S is a subset of $m - 1$ size from the set [1, $n - 1$]. Let i be a variable taken from the set [1, $n - 1$]. Accordingly, the probability of i also being in S is $\frac{m-1}{n-1}$.

Now, let S' be the output of the Random-Sample function. The probability that i is in S' is the following:

$$Pr(i\ in\ S') = Pr(i\ in\ S) + Pr(i\ not\ in\ S) \times Pr(i = n)$$

Substituting the the probability of i also being a variable of S as $\frac{m-1}{n-1}$ gives

complement of $Pr(i \text{ in } S)$

$$= \frac{m-1}{n-1} + \left(1 - \frac{m-1}{n-1}\right) \times \frac{1}{n}$$

$$= \frac{m-1}{n-1} + \frac{1}{n} - \frac{m-1}{n(n-1)}$$

$$= \frac{n(m-1) + n - 1 - m + 1}{n(n-1)}$$

$$= \frac{nm - m}{n(n-1)}$$

$$= \frac{m(n-1)}{n(n-1)}$$

$$= \frac{m}{n}$$

In conclusion, the probability of each subset of $[1, n]$ of size m being drawn is uniform, at a chance of m/n.

QUESTION 5:

a)

In randomised quicksort, the pivot is selected randomly, thus the probability of any element being selected is uniform. Accordingly, the chance of the smallest number being picked is equal to the chance of any number being picked.

So:

$$E[X_i] = \frac{1}{n}$$

b)

When the pivot is picked, $n - 1$ comparisons are performed, where all the other elements to the pivot. Depending on the pivot chosen, the one subarray will contain $(i - 1)$ elements, while the other contains $(n - i - 1)$ elements. The " $- 1$" accounts for the pivot itself.

All these combinations of outputs are equally likely as derived from part a) with a probability of $\frac{1}{n}$. Thus we can express $E[T(n)]$ as follows:

comparisons of two subarrays

comparing pivot with n-1 elements

$$E[T(n)] = \left[\sum_{i=1}^{n} X_i \left(T(i-1) + T(n-i) \right) + (n-1) \right]$$

$$= \left[\sum_{i=1}^{n} X_i \left(T(i-1) + T(n-i) \right) + \Theta(n) \right]$$

c)

$$E\left[\sum_{i=1}^{n} X_i \left(T(i-1) + T(n-i) + \Theta(n)\right]\right.$$

$$= \sum_{i=1}^{n} E\left[X_i \left(T(i-1) + T(n-i) + \Theta(n)\right]\right.$$

$$= \frac{1}{n} \sum_{i=1}^{n} \left[E\left[T(i-1) + T(n-i) + \Theta(n)\right]\right]$$

$$= \frac{1}{n} \sum_{i=1}^{n} \left[E\left[T(i-1) + T(n-i)\right]\right] + \Theta(n)$$

$$= \frac{1}{n} \left(\left[E\left[\sum_{i=1}^{n} T(i-1)\right] + E\left[\sum_{i=1}^{n} T(n-i)\right]\right) + \Theta(n)\right.$$

$$= \frac{1}{n} \left(2\, E\left[\sum_{i=1}^{n} T(i-1)\right]\right) + \Theta(n)$$

$$= \frac{2}{n} E\left[\sum_{i=1}^{n} T(i-1)\right] + \Theta(n)$$

$$= \frac{2}{n} E\left[\sum_{i=0}^{n-1} T(i)\right] + \Theta(n)$$

$$= \frac{2}{n} \cdot \sum_{i=0}^{n-1} E[T(i)] + \Theta(n)$$

$$= \frac{2}{n} \cdot \sum_{i=2}^{n-1} E[T(i)] + \Theta(n)$$

d)

To compute the LHS, we will rewrite the sum as an integral and use integration by parts:

$$\int_1^{n-1} \log k \times k \, dk = \frac{k^2}{2}\log k - \int \frac{k^2}{2} \times \frac{1}{k} dk$$

$$= \frac{k^2}{2}\log k - \int \frac{k}{2} dk$$

$$= \frac{k^2}{2}\log k - \frac{k^2}{4}$$

subs in bounds:

$$\frac{(n-1)^2 \log(n-1)}{2} - \frac{(n-1)^2}{4} - \left(\frac{1}{2}\log(1) - \frac{1}{4}\right)$$

$$= \frac{(n-1)^2 \log(n-1)}{2} - \frac{(n-1)^2}{4} + \frac{1}{4}$$

$$\frac{(n-1)^2 \log(n-1)}{2} - \frac{(n-1)^2}{4} + \frac{1}{4} \leq \frac{n^2 \log n}{2} - \frac{n^2}{8}$$

the $x \log x$ variable's have the largest impact:

$$\frac{(n-1)^2 \log(n-1)}{2} \leq \frac{n^2 \log n}{2}$$

Thus $\sum_{k=1}^{n-1} k \log k \leq \frac{(n-1)^2 \log(n-1)}{2} - \frac{(n-1)^2}{4} + \frac{1}{4} \leq \frac{n^2 \log n}{2} - \frac{n^2}{8}$

e)

By induction assume that

$$E[T(i)] \le c i \log i$$

Thus:

$$E[T(n)] \le \frac{2}{n} \sum_{i=2}^{n-1} c i \log i + \Theta(n)$$

$$\le \frac{2c}{n} \sum_{i=2}^{n-1} i \log i + \Theta(n)$$

$$\le \frac{2c}{n} \left(\frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \Theta(n)$$

$$\le c n \log n - \frac{cn}{4} + \Theta(n)$$

$$= \Theta(n \log n)$$