COMP 3011 Assignment 3

JAHJA Darwin, 16094501d

1.

The probability of a classroom does not enter by a student is $1 - 1/n$.

For each classroom $i \in \{1, 2, ..., n\}$, we can define a indicator random variable,

$$X_i = I\{\text{classroom } i \text{ is empty after all } m \text{ students have gone to the classrooms}\}$$

Therefore,

$$E[X_i] = Pr\{\text{classroom } i \text{ is empty after all } m \text{ students have gone to the classrooms}\} = (1-1/n)^m$$

Then, let X be the number of empty classrooms after all student m has gone to the classroom. By linear of expectation, $E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n}(1 - 1/n)^m = n(1 - 1/n)^m$.

2.

If $k = 4$ and the initial of the counter is 0, in this case, if we decrement the counter, it would do 4 flips and the counter will turn from 0000 to 1111. Then, if we increment the counter, it would do another 4 flips and turn from 1111 to 0000. By performing such sequences of increment and decrement for $n/2$ times ($n/2$ increments + $n/2$ decrements), the total amortized cost will be 4n.

Similarly, for k-bits binary counter starting from 0, a decrement costs k flips, following up by a increment also costs k flips. Therefore, when performing such sequences of increment and decrement, the amortized cost per operation could be $\Theta(k)$.

3.

(a) l is the left boundary of the Z-box. The idea is to maintain an interval $[l, r]$ with max r so that $[l, r]$ is the prefix substring (substring which is also prefix).

(b) Pseudocode of Algorithm

Input: A string R and a string S.

Output: The longest suffix of S which is equal to a prefix of R.

```
let string T = R + "$" + S
lenT = length of string T
Z = array of length lenT, with all elements initialized as 0
left = 0, right = 0
```

```
for k = 1 to lenT:
    if k > right do
        lt = rt = k
        while rt < lenT and T[right] == T[right-left]:
            right = right + 1
        Z[k] = right - left
        right = right - 1

    else
        // Operation inside the box
        k1 = k - left

        // if the value does not stretched till right bound then just copy it
        if Z[k1] < right-k+1 do
            Z[k] = Z[k1]

        else
            // Otherwise try to see if there are more matches
            left = k
            while right < lenT and T[right] == T[right-left] do
                right = right + 1
            Z[k] = right - left
            right = right - 1

    // Check whether it is a suffix
    if k + Z[k] == lenT:
        return R[0:Z[k]], which is the prefix of R equal to longest suffix of S

return NULL, at which there is no suffix of S equal to prefix of R
```

The time complexity of the above algorithm using Z-Box algorithm is $O(|R| + |S|)$, where $|R|$ and $|S|$ are the lengths of string R and S respectively.

4.

Let $\langle p_1, p_2, ..., p_n\rangle$ be the points in P sorted in order of ascending x-coordinates then ascending of y-coordinates.

Then, we need to create a list R containing all possible combinations for a pair of points. We can do this by looping each point linearly and combine them with other points (i.e. $R\{(p_1, p_2), (p_1, p_3), ..., (p_1, p_n)$,$(p_2, p_3), ..., (p_{n-1}, p_n)\}$) so that the pairs of points in R are still in ascending order.

After that, for each pair of points (p_a, p_c) in R, we consider them as 2 diagonally opposite points of a square to calculate the other two points, p_b and p_d where $x_{pb} < x_{pd}$ or $y_{pb} < y_{pd}$ if $x_{pb} = x_{pd}$, and store them in a pair (p_b, p_d). At this stage, we can do binary search in

R to check whether (p_b, p_d) exists in R. If it exists, then put the result in a list S as a set $\{p_a, p_b, p_c, p_d\}$ if this set does not exist in S. Otherwise, continue the loop.

Finally, output S which contains all sets of points that can form a square. Based on the above method, the problem can be solved in $O(|P|^2 \cdot log|P|)$.

Steps:

1. Sort the points in P in order of ascending x-coordinates then ascending of y-coordinates.
2. Let R =$\{(p_1, p_2), (p_1, p_3), ..., (p_1, p_n)$,$(p_2, p_3), ..., (p_{n-1}, p_n)\}$, storing all possible combination of pair of points.
3. For each pair of points in R, take them as 2 diagonally opposite points (p_a, p_c) of a square to calculate (p_b, p_d).

```
centerX = (Xa+Xc)/2; centerY = (Ya+Yc)/2; // Center point
halfX   = (Xa-Xc)/2; halfY   = (Ya-Yc)/2; // Half-diagonal

Xb = centerX-halfY; Yb = centerY+halfX; // Pb
Xd = centerX+halfY; Yd = centerY-halfX; // Pd
```

Then, perform binary search in R to find (p_b, p_d). If found, put $\{p_a, p_b, p_c, p_d\}$ in a list S if not in S. Otherwise, continue the loop.

4. Output S.

To analyze the running time, sorting takes $O(nlogn)$ time. Step 2 takes $O(n^2)$ time while Step 3 takes $O(n^2logn)$ Other operations take $O(1)$ time.

Therefore, by summing up the cost of every line, the time complexity of this algorithm is $O(n^2logn)$, where $n = |P|$.