- All problems are worth 15 points. Unless states otherwise, subproblems are weighted equally.
- Please write down name and student ID in all pages.

Problem 1

Solve the following subproblems related to asymptotic notations.

(a) Given $g(n) = 1 + c^{-1} + c^{-2} + \cdots + c^{-n}$, provide interval of c to satisfy following statements:

- (1 point) $g(n) = o(n)$
  $c > 1$ (or $c > 1$ and $c = 0$)
- (1 point) $g(n) = \Theta(n)$
  $c = 1$
- (1 point) $g(n) = \omega(n)$
  $0 < c < 1$ or $c < 1$

Grading Criteria:

- (+1 points) Correct answer.

(b) Determine whether each statement is True or False. If the statement is false, provide a counter-example.

- (3 points) if $f(n) = \Theta(g(n))$ and $g(n) = O(h(n))$, then $f(n) = \Theta(h(n))$.
  False.
  Counter-example: Consider the case when $f(n) = n$, $g(n) = n$, and $h(n) = n^2$.
  Then $f(n) = \Theta(h(n))$ becomes $n = \Theta(n^2)$, which does not satisfy.

- (3 points) if $f(n) = O(g(n))$ and $f(n) = \Omega(h(n))$, then $h(f(n)) = O(g(f(n)))$.
  True.
  Justification: When $f(n) = O(g(n))$, $f(n) \leq c_1 g(n)$, and when $f(n) = \Omega(h(n))$, $f(n) \geq c_2 h(n)$.
  Since $f(n), g(n), h(n)$ is increasing function, $c_2 h(n) \leq f(n) \leq c_1 g(n)$ and $c_2 h(f(n)) \leq f(f(n)) \leq c_1 g(f(n))$ holds true.
- (3 points) if $f(n) = \Omega(g(n))$, then $2^{f(n)} = \Omega(2^{g(n)})$.
  False.
  Counter-example: Let $f(n) = n/2$, $g(n) = n$, $2^{f(n)} = 2^{n/2}$, and $2^{g(n)} = 2^n$.
  Then $2^{n/2} \neq \Omega(2^n)$.

- (3 points) if $\lim_{n\to\infty} f(n)/g(n) = \infty$ or $\lim_{n\to\infty} f(n)/g(n) = c \leq 0$, then $f(n) = \Omega(g(n))$.
  False.
  Counter-example: Let $f(n) = n^2$ and $g(n) = n^3$.
  Then $\lim_{n\to\infty} f(n)/g(n) = 0$, $f(n) \neq \Omega(g(n))$.

Grading Criteria:

- (+1 points) Correct answer.
- (+2 points) Correct counter-example.

Problem 2

Solve the recurrence relation and find a **tight asymptotic bound** using the following method.

(a) (5 points) Use a recursion tree. Assume that the number of depth of a tree is $\log_4 n + 1$.

$$
\begin{aligned}
T_1(n) &= T_1(n/4) + T_1(2n/5) + n^2 \\
T_1(n) &= n^2 + (89/400)n^2 + (89/400)^2 n^2 + \cdots + (89/400)^{\log_4 n - 1} n^2 + n^{log_4(89/400)} \times T(1) \\
&= \sum_{i=0}^{log_4 n - 1} (89/400)^i n^2 + (89/400)^{log_4 n} \times T(1) \\
&= \frac{1 - n^{log_4(89/400)}}{1 - (89/400)} \times n^2 + n^{log_4(89/400)} \times T(1) \\
&= n^2 + n^{log_4(89/400)} \times T(1) = \Theta(n^2)
\end{aligned}
$$

(b) (5 points) Use a substitution method. Provide guess and induction hypothesis then prove it.

$$T_2(n) = 7T_2(n/7) + n$$

Guess: $T(n) = \Theta(n \log n)$
I.H.: $T(k) \leq ck \log k for k < n$.

$$
\begin{aligned}
&\text{Let } k = n/7 \\
T_2(n) &= 7T_2(n/7) + n \\
&\leq 7(c(n/7)\log(n/7)) + n \\
&\leq cn(\log n - \log 7) + n \\
&\leq cn \log n - c \log 7 \times n + n \\
&\leq cn \log n - (c-1)n,\ c > 1 \\
T_2(n) &= \Theta(n \log n)
\end{aligned}
$$

(c) (5 points) Use a master theorem. Notice that $S(m) = T(2^m) = T(n)$.

$$T_3(n) = T_3(\sqrt{n}) + n$$

$$
\begin{aligned}
&\text{Let } 2^m = n. \\
T_3(2^m) &= T_3(2^{m/2}) + 2^m \\
S(m) &= S(m/2) + 2^m \\
f(m) &= m = m^{log_2 1} = \Omega(1)\ \text{, Case 3} \\
f(m/2) &\leq cf(m)\ \text{, e.g. } c = 1/2 \\
S(m) &= \Theta(2^m) \\
T_3(n) &= \Theta(n)
\end{aligned}
$$

Grading Criteria:

(a) (+2 points) For expanding into geometric series or show tree with level sum
(−1 points) Missing sum of level in a tree.
(−1 points) Error on ratio of geometric series.
(+2 points) For solving the geometric series.
(+1 points) Correct answer.

(b) (+0.5 points) For guess and induction hypothesis.
(+1 points) For correct substitution.
(+2 points) For proving the guess / make into assumption form.
(+1 points) Missing range of constant c.
(+0.5 points) For correct answer.

(c) (+2 points) For the substitution (Substitute and substitute back).
(+1 points) Correct $f(m)$ calculation and case.
(+1 points) Correct regularity test.
(+1 points) Correct answer.

Problem 3

There are some cases which we should view each operations bitwise. For example, addition of two n-bit integers takes time $\mathcal{O}(n)$. This way, the best known time complexity multiplying two n-bit integers is $\mathcal{O}(n \log n)$.

(a) (2 points) Prove that you need at most $2n$-bits to store the result of the multiplication of two n-bit integers.

Multiplying two largest integers of n-bits becomes $(2^n - 1)^2 = 2^{2n} - 2^{n+1} + 1 < 2^{2n}$, so $2n$ bits is enough.

Grading Criteria:
(2 points) Correct proof.
(1 point) Minor mistake.
(0 points) Incorrect proof.

(b) (5 points) Given an n-bit integer a and an m-bit non-negative integer p, give a tightest asymptotic bound you can get using m and n as the time complexity to compute a^p. Explain.

By using the powering method discussed in the lecture, we can prove that time complexity of $\mathcal{O}\left(n2^m\left(m + \log n\right)\right)$ can be achieved.

Let the time spent to compute the n-bit number raised to the power of p, where p is an m-bit number, be $T(m, n)$. Then we may write $T(0, n) = \mathcal{O}(1)$ and $T(m, n) \leq T(m - 1, n) + c \cdot 2^{m-1} n \log(2^{m-1} n)$ for some non-negative constant c. Therefore, $T(m, n) \leq c \cdot \sum_{i=0}^{m-1} 2^i n \log(2^i n) + \mathcal{O}(1)$.

We should elaborate more on $\sum_{i=0}^{m-1} 2^i n \log(2^i n) = n \log 2 \sum_{i=0}^{m-1} i 2^i + n \log n \sum_{i=0}^{m-1} 2^i$. By induction or any other method, it is easy to prove that $\sum_{i=0}^{m-1} i 2^i = m2^m - 2^{m+1} + 2$ and $\sum_{i=0}^{m-1} 2^i = 2^m - 1$. Thus, we may conclude that $T(m, n) = \mathcal{O}\left(n2^m\left(m + \log n\right)\right)$.

Grading Criteria:
(+3 points) Correct proof.
(+2 point) Used a powering method or an equivalently fast method.
(0 points) Correct time complexity without proof.
(+0 points) Incorrect proof.
(−1 point) per minor mistake.

(c) (8 points) Given k n-bit integers $a_1, \ldots, a_k$ and an m-bit non-negative integer p, give a tightest asymptotic bound you can get using m, n, and k as the time complexity to compute $\prod_{i=1}^{k} a_i^p$, the multiplication of a_i^ps. Explain.

The process is split into two steps. First, compute $A = \prod_{i=0}^{k} a_i$, and second, compute A^p. As A will have kn bits, by (b) the second operation will take time $\mathcal{O}\left(nk2^m\left(m + \log nk\right)\right)$.

Let $T(s; l)$ be the time spent to compute $\prod_{i=l}^{l+s-1} a_i$ in the following manner: If $s = 1$, then return a_l. Otherwise if $s > 1$, compute $T\left(\left\lfloor \frac{s}{2} \right\rfloor; l\right) = \prod_{i=l}^{l+\lfloor s/2 \rfloor - 1} a_i$ and $T\left(\left\lceil \frac{s}{2} \right\rceil; l + \left\lfloor \frac{s}{2} \right\rfloor\right) = \prod_{i=l+\lfloor s/2 \rfloor}^{l+s-1} a_i$ first, then multiply the two. By this we can obtain a recurrence relation $T(s) \leq 2T(s/2) + c \cdot \frac{ns}{2} \log \frac{ns}{2}$ for some non-negative constant c. Then $T(k) \leq c \sum_{i=0}^{\lceil \log k \rceil - 1} \frac{nk}{2} \log \frac{nk}{2^i} + \mathcal{O}(1)$, and simple algebra leads to the result $T(k) = \mathcal{O}\left(nk \log k \log nk\right)$.

Thus, by combining two procedures, you get a total time complexity of $\mathcal{O}(nk \log k \log nk + nk2^m\left(m + \log nk\right))$.

Grading Criteria:
(+4 points) Correct proof.
(+2 point) First computed $\prod a_i$ and then powered the result.
(+2 point) Multiplied a_i in a tournament-like manner.
(0 points) Correct time complexity without proof.
(+0 points) Incorrect proof.
(−1 point) per minor mistake.

Problem 4

The problem discusses about a variant of quicksort. In the original randomized quicksort, you randomly pick one pivot. In this problem, however, you are going to choose two random pivots. The algorithm is as follows, and calls `ModQuick`$(\texttt{A}, 0, n)$ initially, where n is the length of the list `A`. Also, assume that `A` uses zero-based index.

Algorithm 1 Modified Quicksort `ModQuick`$(\texttt{A[]}, l, r)$

```
A[] : An input list of numbers with length n
if r − l < 8 then
    Sort A using an insertion sort in a range from l inclusive to r exclusive, and terminate.
Randomly select two indices l ≤ i < j < r, in constant time.
if A[i] > A[j] then
    swap(A[i],A[j])
swap(A[i], A[l]); swap(A[j], A[r − 1])
c ← l + 1; p ← l + 1
while c < r − 1 do
    if A[c] < A[l] then
        swap(A[p], A[c])
        p ← p + 1
    c ← c + 1
swap(A[l], A[p − 1])
c ← p; s ← p − 1
while c < r − 1 do
    if A[c] < A[r − 1] then
        swap(A[p], A[c])
        p ← p + 1
    c ← c + 1
swap(A[p], A[r − 1])
ModQuick(A, l, s); ModQuick(A, s + 1, p); ModQuick(A, p + 1, r)
```

(a) (2 points) Explain the correctness of the algorithm.

The algorithm first places two pivots in the right place, and then moves on to three smaller sorting tasks.

Grading Criteria:
(2 points) Correct explanation.
(1 point) Minor mistake.
(0 points) Invalid explanation.

(b) (5 points) What is the running time of `ModQuick` in the worst case? Give a tight asymptotic bound with explanation.

$\mathcal{O}\left(n^2\right)$ occurs when the random procedure always chooses two smallest elements of `A`. Now let's prove that the running time of the algorithm can't exceed $\mathcal{O}\left(n^2\right)$.

Let n be a length of A we're currently observing. There are $\mathcal{O}\left(n^2\right)$ ways to choose a pair of distinct indices of A, and observe that the algorithm does a comparison at most constant time for each of such pairs. For pivots i and j, we compare $A[i]$ and $A[j]$ once, and compare all the others between l and r with $A[i]$ and $A[j]$. Therefore, the total number of comparisons is $\mathcal{O}\left(n^2\right)$, which bounds the worst-case time complexity to $\mathcal{O}\left(n^2\right)$.

Grading Criteria:
(+4 points) Correct proof.
(+1 point) Correctly constructed a worst case scenario.

(0 points) Correct time complexity without proof.
(+0 points) Incorrect proof.
(−0 points) Didn't provide an upper bound.
(−1 point) per minor mistake.

(c) (8 points) Assume that each possible pair (i, j) which $l \leq i < j < r$ in the algorithm is chosen with equal probability. In other words, each pair is chosen with a probability of $\frac{2}{(r-l)(r-l-1)}$. What is the expected running time of `ModQuick`? Give a tight asymptotic bound with explanation.

Let $T(n)$ be the expected time to sort the list A of length n. Then we can get the following relation for $T(n)$:

$$\begin{aligned} T(n) &= \frac{2}{n(n-1)} \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} (T(i) + T(j-i-1) + T(n-j-1) + \Theta(n)) \\ &= \frac{6}{n(n-1)} \sum_{i=0}^{n-2} (n-i-1)T(i) + \Theta(n) \\ &= \frac{6}{n} \sum_{i=0}^{n-2} T(i) - \frac{6}{n(n-1)} \sum_{i=0}^{n-2} iT(i) + \Theta(n) \end{aligned}$$

Let $Q(n)$ be an equation $n(n-1)T(n) = 6(n-1)\sum_{i=0}^{n-2} T(i) - 6\sum_{i=0}^{n-2} iT(i) + \Theta(n^3)$, which is $n(n-1)$ times the last obtained equation. Subtract $Q(n)$ from $Q(n-1)$, then we get $n(n-1)T(n) - (n-1)(n-2)T(n-1) = 6\sum_{i=0}^{n-2} T(i) + \Theta\left(n^2\right)$. Let this equation be $R(n)$, and subtracting it from $R(n-1)$ gives

$$\begin{aligned} n(n-1)T(n) - 2(n-1)(n-2)T(n-1) + (n-2)(n-3)T(n-2) &= 6T(n-2) + \Theta\left(n\right), \text{ or,} \\ n(n-1)T(n) - 2(n-1)(n-2)T(n-1) + (n^2-5n)T(n-2) &= \Theta\left(n\right) \end{aligned}$$

Now we can use the substitution method to prove that $T(n) = \mathcal{O}(n \log n)$. By substituting $T(n) = cn \log n$ for some constant c, the left side of the equation becomes:

$$\begin{aligned} &n^3(\log n - 2\log(n-1) + \log(n-2)) - n^2\left(\log n - 8\log(n-1) + 7\log(n-2)\right) \\ &- 10n(\log(n-1) - \log(n-2)) + 4\log(n-1) \end{aligned}$$

Using the intermediate value theorem, one can prove that $n^2(\log n - 2\log(n-1) + \log(n-2))$, $n(\log n - 8\log(n-1) + 7\log(n-2))$, and $\log(n-1) - \log(n-2)$ are all $\mathcal{O}(1)$. Therefore, the left hand side of the equation is indeed $\mathcal{O}(n)$, and we can conclude that $T(n) = \mathcal{O}(n \log n)$. Additionally, as this algorithm is a comparison-based sorting algorithm, we may also conclude that $T(n) = \Theta(n \log n)$.

Grading Criteria:
(+6 points) Correct proof.
(+2 point) Correctly constructed an equation for the expected time of the algorithm.
(0 points) Correct time complexity without proof.
(+0 points) Incorrect proof.
(−1 point) per minor mistake.

Problem 5

This problem discusses the `RAND-SELECT` and `SELECT` algorithm.

(a) (2 points) Suppose we use RAND-SELECT to select the maximum element of the array $A = [7, 2, 3, 9, 5, 8, 4, 1, 6]$. Describe EACH a sequence of partitions that results in a worst-case and best-case performance of RAND-SELECT.

Best-case: 9 Worst-case: 1,2,3,4,5,6,7,8,9

Grading Criteria:
(+1 points)Got best-case correct.
(+1 points)Got worst-case correct.
(−0 points) Instead of saying the exact sequence, mentions that the worst-case is when pivot is selected from sr

(b) (3 points) Recall the `SELECT` algorithm discussed in the course. Provide a tight asymptotic bound on the `SELECT` algorithm, and explain its recurrence relation.

At least $3\lfloor n/10 \rfloor$ elements are less than or equal to the median of medians, while at least $3\lfloor n/10 \rfloor$ elements are greater than or equal to the median of medians. In the worst case, the search proceeds recursively on a subarray containing $7n/10$ elements. Therefore, the recurrence relation can be expressed as $T(n) \leq T(n/5) + T(7n/10) + O(n)$. By employing the substitution method, this results in a time complexity of $O(n)$.

Grading Criteria:
(+1 points) Explained how the recursive search occurs on a subarray of 7n/10 elements
(−1 points) Explanation is incoherent and is drawings without sufficient explanations.
(+1 points) Got the recurrence relation correct.
(+1 points) Got the tight asymptotic bound as O(n).

(c) (6 points) The `SELECT` algorithm from (b) has the input elements be divided into groups of 5. Prove that `SELECT` algorithm maintains the same time complexity as (b) if the input elements are divided into groups of $g > 3$, where g is an odd integer.

At least $\frac{(g+1)/2}{\lfloor n/2g \rfloor}$ elements are less than or equal to the median of medians, while at least $\frac{(g+1)/2}{\lfloor n/2g \rfloor}$ elements are greater than or equal to the median of medians. In the worst case, the search proceeds recursively on a subarray containing $\frac{(3g-1)}{4g}n$ elements. Therefore, the recurrence relation can be expressed as $T(n) \leq T(n/g) + T(\frac{(3g-1)}{4g}n) + O(n)$.

By employing the substitution method of $T(k) \leq c_1 k$ for $k < n$, we obtain $T(n) \leq \frac{c_1 n}{g} + \frac{c_1(3g-1)}{4g}n + c_2 n \leq \frac{3c_1(g+1)}{4g}n + c_2 n$. Since $\frac{3(g+1)}{4g} < 1$ if $g > 3$, we can choose c_1 thats large enough than c_2 to achieve $\frac{3c_1(g+1)}{4g}n + c_2 n < c_1 n$. This results in a time complexity of $O(n)$.
.

Grading Criteria:
(+2 points) Got the recurrence relation correct.
(−1 points) No mention of how recurrence relation was calculated.
(−1 points) Got one of the terms wrong for the recurrence relation.
(−0 points) Explained how to get the recurrence relation in parallel to problem (b)
(+3 points) Proved the tight asymptotic bound through solving the recurrence relation,
(−2 points) No proof and only contains the name of the method of proof.
(−1 points) Numeric calculation error while proving asymptotic bound
(−1 points) Attempted proof has excessive leaps in reasoning / insufficient explanation..
(+1 points) Got the tight asymptotic bound as O(n)
(−0 points) Said the asymptotic bound is same as (b), and (b) is O(n).

(d) (4 points) Let A be an array that contains n distinct integers, where n is an odd integer. Consider an algorithm `FIND-QUARTILE(r)` that finds and returns set $B = [a_k | \frac{n+1}{2} - r <= k <= \frac{n+1}{2} + r]$ of size $2r+1$. Construct a pseudocode for `FIND-QUARTILE(r)` whose running time complexity is same of that of (b). Hint: When writing your pseudocode, you may use `SELECT(A,i)` from (b) and (c) which returns the ith largest element in A.

Algorithm 2 Find Quartile(A,r)

$B \leftarrow$ Empty array
$n \leftarrow$ size of A
$Front \leftarrow SELECT(A, \frac{n+1}{2} - r)$
$Back \leftarrow SELECT(A, \frac{n+1}{2} + r)$
for each element i in A **do**
  if Front<= i <=Back **then**
    Add i to B.
return B.

Grading Criteria:
(+4 points) Correct pseudocode solution that satisfies all the conditions.
(−2 points) The code doesn't run in O(n).
(−1 points) Assumes SELECT() returns the index.
(−1 points) Difficult to consider as pseudocode.
(−1 points) No return value due to infinite loop, but the intended return value is right.
(−1 points) No definition on variable.
(−2 points) Used external algorithm without clarity of what it returns.
(−2 points) Other wrong return value other than the specified above.(Such as due to assuming A is initially sorted)

Problem 6

A topological sort of a directed graph is a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex v, u comes before v in the ordering. Precisely, a topological sort is a graph traversal in which each node v is visited only after all its dependencies are visited.

(a) (5 points) Is it true that every directed graph can represent one or more linear orderings through topological sorting? If true, what is the reason? If false, provide a counterexample and explain how it contradicts the given logic.

False. Topological sorting is only applicable to directed acyclic graphs (DAGs), meaning graphs that are both directed and contain no cycles.

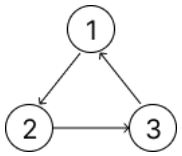

Figure 1: Counterexample for 6-(a)

Grading Criteria:

(+1 points) Correct answer (False).
(+2 points) Correct counterexample.
(+2 points) Correct explanation.

(b) (10 points) The following pseudocode performs topological sorting on a directed graph based on depth-first search (DFS). Complete the contents of the function $VISIT$. Additionally, if you answered False in (a), please include a section in the $VISIT$ function that checks whether topological sorting can be accomplished. The overall running time should be $O(|V| + |E|)$

Algorithm 3 Topological Sorting

```
L ← Empty list that will contain the stored nodes
while exists nodes without a permanent mark do        ▷ Check if all nodes are visited.
    select an unmarked node n                          ▷ Select unvisited node
    VISIT(n)

function VISIT(node n)
    if n has a permanent mark then
        return
    if n has a temporary mark then
        stop    (graph has at least one cycle)

    mark n with a temporary mark

    for node m with an edge from n to m do
        VISIT(m)

    mark n with a permanent mark
    add n to head of L
```

Grading Criteria:

(+10 points) A pseudocode that operates correctly while satisfying all given conditions.
(−0 points) The sorted order is in reverse.
(−2 points) Do not verify/specify whether topological sorting is possible.
(−2 points) Cycle detection is not performed, or an error occurs.
(−2 points) Works correctly only when the selected unvisited node is a source or a sink.
(−4 points) Do not clearly display or store the sorted result.
(−2 points) Although the sorted result is not clearly saved, it can be derived through pre/post.
(−4 points) Do not meet the DFS requirements.
(−3 points) Wrong time complexity.
(−1 points) Error occurs only for the first node.
(−3 points) Infinite loop due to not using the mark.
(−5 points) Contain a fatal error.
(−5 points) Excessive leaps in reasoning / insufficient explanation.
(−8 points) Difficult to consider as pseudocode.

Problem 7

Given a directed graph G with V vertices and E edges, you are tasked with completing the pseudocodes. The solutions should have a time complexity of $O(V + E)$. G^R is the reverse graph of G. All parameters are passed via *call by reference*. Write your answer with clear grammar, ensuring it aligns with the provided base code as much as possible.

(a) (7 points) Complete the pseudocode to calculate the number of strongly connected components (SCCs). **The answer MUST NOT exceed 13 lines.**

Algorithm 4 Finding Number of Strongly Connected Components (SCCs) 1

```
function DFS(cur_v, graph, visited, oplist (optional))
    visited[cur_v] ← True                          ▷ Mark the current vertex as visited
    for each adj_v in graph[cur_v].adjacent do     ▷ Iterate over adjacent vertices
        if visited[adj_v] = False then             ▷ If adjacent vertex has not been visited
            DFS(adj_v, graph, visited, oplist)     ▷ Recursively perform DFS
    if oplist is provided then
        oplist.append(cur_v)                       ▷ Append current vertex to the list if provided

function CAL_NUM_SCCs1(G, G^R)
    visited ← False for all v ∈ G.vertices         ▷ Initialize visited vertices
    oplist ← []                                    ▷ Empty list
    num_scc ← 0                                    ▷ Initialize number of SCCs
    ———————————— Write your answer here ————————————
    for each v ∈ G.vertices do                     ▷ Perform DFS to store vertices
        if visited[v] = False then                 ▷ in the order of finishing time
            DFS(v, G^R, visited, oplist)

    visited ← False for all v ∈ G.vertices         ▷ Reset visited vertices
    while oplist is not empty do                   ▷ Process vertices in reverse finishing order
        if visited[oplist.last] = False then
            DFS(oplist.last, G, visited)           ▷ Mark the vertices in the same SCC
            num_scc ← num_scc + 1                  ▷ Increment the SCC count
        oplist.pop()                               ▷ Remove the vertex after processing
    ———————————— Write your answer here ————————————
    return num_scc
```

Grading Criteria:

(7 points) The answer works correctly.
(4 points) The answer includes a minor error. (*e.g.*, Not resetting *visited*, omitting the condition for while loop, not popping *oplist*, not fully visiting to fill out *oplist*, traversing *oplist* in ascending order, *while all vertices are visited* ⇐ skipping while loop, or using the same G for two DFSs.)
(0 points) The answer includes many minor errors.
(0 points) The answer includes a fatal error. (*e.g.*, one DFS, calling an uncertain function, time complexity over $O(V + E)$, or consistently $num\_scc \geq V$ or $num\_scc \leq 1$)
(0 points) The answer exceeds 13 lines.

(b) (8 points) Complete the pseudocode to calculate the number of strongly connected components (SCCs). **The answer MUST NOT exceed 13 lines.**

Algorithm 5 Finding Number of Strongly Connected Components (SCCs) 2

```
function DFS(cur_v, graph, visited, oplist, pretime, time, num_scc)
    visited[cur_v] ← True                          ▷ Mark the current vertex as visited
    time ← time + 1                                ▷ Increment the time counter
    pretime[cur_v] ← time                          ▷ Set the discovery (pre-time) of the current vertex
    ret ← time                                     ▷ Initialize the return value with the current time
    ———————————— Write your answer here ————————————
    oplist.append(cur_v)
    for each adj_v in graph[cur_v].adjacent do
        if pretime[adj_v] = 0 then
            ret = min(ret, DFS(adj_v, graph, visited, oplist, pretime, time, num_scc)
        else if visited[adj_v] = True then
            ret = min(ret, pretime[adj_v])
    ———————————— Write your answer here ————————————
    if ret = pretime[cur_v] then                   ▷ If cur_v is root of an SCC,
        while oplist is not empty do               ▷ Process vertices in the current SCC
            v ← oplist.last                        ▷ Retrieve the last vertex from oplist
            oplist.pop()                           ▷ and remove the vertex from oplist
            visited[v] ← False                     ▷ Mark the vertex as processed
            if cur_v = v then                      ▷ Stop when the root vertex is reached
                break
        num_scc ← num_scc + 1                      ▷ Increment the SCC count
    return ret

function CAL_NUM_SCCs2(G, G^R)
    visited ← False for all v ∈ G.vertices         ▷ Initialize visited vertices
    oplist ← []                                    ▷ Empty list
    pretime ← 0 for all v ∈ G.vertices             ▷ Initialize pre-time, i.e., time of discovery
    time ← 0                                       ▷ Initialize time counter
    num_scc ← 0                                    ▷ Initialize SCC count

    for each v in G.vertices do                    ▷ Iterate over adjacent vertices
        if pretime[v] = 0 then                     ▷ If adjacent vertex has not been visited
            DFS(v, G, visited, oplist, pretime, time, num_scc)   ▷ Recursively perform DFS
    return num_scc
```

Grading Criteria:

(8 points) The answer works correctly.
(4 points) The answer contains a minor error. (updating *oplist* in post order, or writing incorrect if statement)
(0 points) The answer contains a fatal error. (Not updating *oplist*, Not finding the minimum pre-time in the same SCC, or omitting at least one if statement.)
(0 points) The answer exceeds 13 lines.

Problem 8

Grading Criteria:

(a) (1) (+2 points) vertex : c_i
(+4 points) $l(u, v) = -\log r_{i,j}$
(2)

Algorithm 6 Bellman-Ford Algorithm (G, l, s)

```
1: Input :
2:     G : directed graph (V, E)
3:     l(u, v) : edge lengths of (u, v) ∈ E
4:     s: vertex ∈ V
5: Output :
6:     dist(u): distance from s to u
7:     prev(u) : previous vertex of u in shortest path to u      (+1 points)
8:     or seq(u) : shortest path from s to u      (+1 points)
9:
10: function UPDATE((u, v) ∈ E)
11:     if dist(v) > dist(u) + l(u, v) then      (+2 points)
12:         prev(v) = u      (+1 points)
        or
13:     if dist(v) > dist(u) + l(u, v) then      (+2 points)
14:         seq(v) = seq(u).append(v)      (+1 points)
15:     dist(v) = min{dist(v), dist(u) + l(u, v)}
16:
17: function SHORTEST-PATHS(G, l, s)
18:     dist(s) = 0
19:     for all u ∈ V − {s} do
20:         dist(u) = ∞
21:     repeat |V| − 1 times :
22:         for all e ∈ E do UPDATE(e)
```

(−2 points) If you modify existing code.
(−1 points) Wrong code line.
(−1 points) Code line is not specified.

(b) If $r_{1,i} \cdot r_{i,j} \cdots r_{l,m} \cdot r_{m,1} > 1$, $-\log(r_{1,i} \cdot r_{i,j} \cdots r_{l,m} \cdot r_{m,1}) = (-\log r_{1,i}) + (-\log r_{i,j}) + \cdots + (-\log r_{l,m}) + (-\log r_{m,1}) < 0$. Therefore, if there is a negative cycle in the graph, you can find such a sequence by going to that cycle, doing a few rounds and coming back. Therefore, you have to repeat UPDATE(e) for all $e \in E$ one more (+2 points) and check the any distance of a vertex is changed.(+2 points). If there is a change, there is such a sequence (+1 points).

(+2 points) If you use brute force algorithm (search all cycles).
(+1 points) If you only mentioned negative cycle detection.
(−2 points) If you repeat more than one (not efficient).
(−2 points) If you detect distance increasing, not decreasing.