Name: Yan, Zi
Course: CIS 502
Assignment: HW3

Problem 1

(a) The claim is false, which means there can exist at least one stream i satisfying $b_i > rt_i$.

Proof. Suppose there are two streams (b_1, t_1) and (b_2, t_2), where $b_1 \le rt_1$, and we just need to show it is possible that $b_2 > rt_2$.

For first stream, the constraint is satisfied. So we just need to prove that after adding second stream which has $b_2 > rt_2$, the constraint is still satisfied. We need to show $b_1 + b_2 \le r(t_1 + t_2)$. In order to facilitate our proof, we can replace the inequalities with some equations, $b_1 + \delta_1 = rt_1$ and $b_2 - \delta_2 = rt_2$, where $\delta_1, \delta_2 > 0$. Then $b_1 + b_2 \le r(t_1 + t_2)$ will change to $rt_1 - \delta_1 + rt_2 + \delta_2 \le rt_1 + rt_2$. Therefore, we need to show $\delta_2 - \delta_1 \le 0$. So it means if the amount of bits you send more than rt_2 while sending 2nd stream does not exceed the extra amount that you can send while sending 1st stream, the constraint will still be satisfied. In sum, $b_2 > rt_2$ is possible. □

(b) The algorithm is that first we calculate the rate r_i for each time period t_i while sending b_i bits, then put the streams with $r_i < r$ before those with $r_i \ge r$. The outcome result is the schedule.

Proof. We first split all streams into two group, one for their $r_i < r$, and the other for their $r_i \ge r$. Assume there are k streams whose rates are below r, therefore first group of streams are (b_{l_1}, t_{l_1}) to (b_{l_k}, t_{l_k}), where $l_1, ..., l_k \in \{1, ..., n\}$, and $b_{l_i} \le rt_{l_i}$ $(1 \le i \le k)$, and second group of streams are (b_{h_1}, t_{h_1}) to $(b_{h_{n-k}}, t_{h_{n-k}})$, where $h_1, ..., h_{n-k} \in \{1, ..., n\}$, and $b_{h_i} > rt_{h_i}$ $(1 \le i \le (n-k))$. For two groups of streams, the inequalities can be rewritten as $b_{l_i} + \delta_{l_i} = rt_{l_i}$ and $b_{h_j} - \delta_{h_j} = rt_{h_j}$, where $1 \le i \le k$, $1 \le j \le (n-k)$. And $\sum\limits_{j=1}^{n-k} \delta_{h_j} - \sum\limits_{i=1}^{k} \delta_{l_i} \le 0$.

It is obvious that any schedule for the (b_{l_i}, t_{l_i}) streams will not violate the constraint, so we can arrange the those k streams in any order. Then we need to take care of the (b_{h_j}, t_{h_j}) streams. Because we already have $\sum\limits_{j=1}^{n-k} \delta_{h_j} - \sum\limits_{i=1}^{k} \delta_{l_i} \le 0$, any order of arrangement of the rest streams will not break it as well as the constraint.

Consequently, the algorithm will give a valid schedule. □

The runtime consists of calculating the r_i for each stream, which will take $O(n)$ time, and putting all (b_{l_i}, t_{l_i}) before any (b_{h_j}, t_{h_j}), which will use $O(n)$. So the total runtime is $O(n)$.

Problem 2

(a) Use the Interval Scheduling algorithm to find as many disjoint intervals as possible, where the number of intervals is n. But slightly modify the algorithm by taking the finish time of each selected interval as the status_check point and regarding incompatible intervals in Interval Scheduling algorithm as covered processes here . After running the modified algorithm, all status_ check points are out.

Proof. Suppose not. There is a better way to detect by only using $n-1$ status_check. According to the Interval Scheduling algorithm, there can be as many as n disjoint process intervals, and one status_check cannot detect any two disjoint process intervals. Therefore, it contradicts that $n-1$ status_check can suffice to check all the sensitive process intervals. □

The runtime of this algorithm is the same as Interval Scheduling algorithm, namely $O(n \log n)$

(b) The claim is true.

Proof. Because there are k^* disjoint processes, and no status_check can detect any two disjoint processes, any number, which is less than k^*, of status_check is sure not to detect all the sensitive processes. So there must be a set of k^* status_check for all sensitive processes. □

Problem 3

The algorithm is: divide the n cards into two groups, then recursively find the most duplicated cards in each group by dividing again and finding the most duplicated cards in subgroups, finally merge the result. At the merging stage, we should choose the most duplicated cards in each group (if there are equal-size groups, choose any one), search the other group to find and merge any card which is equivalent to the selected cards, and at last choose the group of cards whose amount is larger as the most duplicated cards. After merging all the groups together, if the group of most duplicated cards has more than $n/2$ cards, then return "Yes", otherwise "No".

Proof. Suppose not. Assume that if we choose a group of less duplicated cards while merging, we can still find out that if $n/2$ cards are equivalent to each other. At level i, where $0 \leq i \leq \log n$ and level 0 means all cards are in one group, level $\log n$ means after $\log n$ divisions n groups of one card, we have 2^i groups. We will choose a group of less duplicated cards, which means the percentage $a_{i,k}$ of the cards in group k is fewer than $1/2$, otherwise this group of cards will be most duplicated, if $a_{i,k} \geq 1/2$. Therefore, after any merging, this kind of cards should still occupy fewer than half of the amount in the merged group, otherwise we will choose the less duplicated cards. Based on the assumption, we can say that, after each merging, the selected cards will always occupy fewer than half of the total cards in the merged group, so it is in the group after final merging. The result contradicts that we can find out $n/2$ cards are equivalent. □

We will use $O(n)$ time to scan each group to find the equivalent cards to the selected cards in the other group at merging. So the total runtime is $T(n) = 2T(n/2) + cn$, namely $T(n) = O(n \log n)$

Problem 4

Observation 1: If we sort the lines increasingly by slope, the first line and the last one will always be visible.

Observation 2: Two lines intersected, then the smaller slope one will lie left to the larger slope one.

First of all, we sort all the lines by slope in a increasing order, which takes $O(n \log n)$ time. Then we divide all lines into two groups, and recursively divide groups until we meet the base case, at most 3 lines in a group.

For base case, from **Observation 1**, we know first and third lines will be visible, so if the second line is above the intersection of first and third, then it is visible, otherwise the second is not visible.

For merging case, we have two groups of lines, $\overline{L} = \{L_{l_1}, ..., L_{l_k}\}$, and $\overline{R} = \{L_{r_1}, ..., L_{r_k}\}$, where k is the amount of lines in each sub-solution, and lines with subscripts l_1 to l_k will always have smaller slopes than those with r_1 to r_k, in addition, both will in order of increasing slope. We also need the intersections between any two consecutive lines, where $node_{l_i}$ is the intersection between L_{l_i} and $L_{l_{i+1}}$, $node_{r_i}$ is the intersection between L_{r_i} and $L_{r_{i+1}}$. While merging, we merge two groups of intersections $node_{l_i}$ and $node_{r_i}$ into one group of N_j, where $i \in \{1..(k-1)\}, j \in \{1..2(k-1)\}$, in order of increasing x- coordinate. This will take $O(n)$, like merge sort. We start from left to right along the x-axis, finding the smallest N_s at whose x-coordinate the uppermost line from $\overline{R}$ is above the one from $\overline{L}$, and let the lines be $L_l \in \overline{L}$ and $L_r \in \overline{R}$. And let (x_I, y_I) be the intersection of L_l and L_r. Therefore, x_I will be between the x-coordinates of N_{s-1} and N_s. From **Observation 2**, we know L_l is to the left of L_r. Thus, the merged result is that visible lines are L_{l_1}, ..., L_l, L_r, ..., L_{r_k} and intersections are N_1, ..., N_{l-1}, (x_I, y_I), N_l, ..., $N_{2(k-1)}$. The algorithm is done.

So the total runtime will be $O(n \log n)$, according to the algorithm above.