Report for CSC3100 Assignment 2

Problem 1 Find the Winner

Language: c++

Originally because I am more familiar with *substr* and some string operations than these in Java. But then I realized that I don't need them at all. (I mean I am not plagiarizing).

Observations:

1. No player giving note means that no one mark himself/herself as a winner, then the interception of marks from all players is an empty set;
2. More than 2 players not giving notes means that A and B (assume it's them) only mark themselves as winner. Then the interception of their marks (thus of all players) is an empty set;
3. Assume there's only one player (A) not giving notes. Assume B didn't leave a note to A. The interception of their marks (thus of all players) is an empty set.
4. Only one player not giving notes and all other players giving notes to him/her will make the player the winner. Thus, there is whether one or no winner.

Possible Solutions:

(1) Check the line of all 0's first, then check if all other people leave notes to him/her ($O(n^3)$);
(2) Use a 2-dimensional vector to store how many notes one player receives and sends, if there's one sending 0 notes while receives *n-1* notes, he/she is the winner ($O(n^2)$).

Algorithm:

1. Check every line (ask everyone) to obtain what notes they send to and use a vector to store the information. Not sending to anyone is equal to letting himself/herself be the winner, which is also stored in a vector;
2. Each time in the iteration, take intercept between the vector from last time and the vector obtained from the current line. Break the loop if the interceptional vector is empty and otherwise go into the next iteration until n^{th} time;
3. After the loop, if the vector is empty, there's no winner. Otherwise, there could be only one winner according to observations above.

Time Complexity Analysis:

n: the number of players

Functions: *findNotes*: O(n), *intercept*: O(m) (m: number of notes each player sends)

Only one for-loop is used in the main function for "turning to every player" ("asking if the player gave other players notes one by one" is done in *findNotes*"). "Turning to another player" is done

right after inputting a new line. Thus, the input won't contribute to time complexity. Besides, the time complexity of *intercept* is no larger than *findNotes* while they are parallel in the iteration. Therefore, the *intercept* does not contribute to time complexity, either.

The time complexity after all, is O(n) from the loop times O(n) from *findNotes* which is $O(n^2)$.

Advantages of the Algorithm:

1. The time complexity is equal to the number of "questions must be asked", which means that the time complexity is the lowest in theory;
2. Only one-dimensional vector is imported and only 2 vectors are used in each iteration. Thus, the memory space may be smaller than other algorithms;
3. I don't see true advantages of mine compared to classmates' …

Problem 2

Language: Java

Observations: (The first 3 points were taught in tutorial)

1. If the current element is greater than or equal to the element at the end of the queue, then the element at the end of the queue will not contribute to the result. Thus, we can discard it;
2. If the element at the beginning of the queue is *k* distance away from the current element, which means the window has passed the element and it will no longer have an effect in the following output. Thus, we discard the one at the beginning of the queue;
3. The data structure is in descending order and the maximum value of the slicing window after each move is at the head of the queue;
4. Notice that both the values and the indices are important. Finding index using value is hard while finding value using index is simple for each data. Thus, the queue stores the index of each data.

Possible Solution:

For every continuous subarray of the input array of *k* length, find the maximum value in it ($O(n^2)$).

Algorithm One: (The one taught in tutorial)

1. Use a double-ended queue as the queue for operations at both ends. The iteration will go on *n* times;
2. If the current element is greater than or equal to the element searched by the index at the end of the queue, pop the queue from the tail;
3. If the index at the beginning of the queue is less than the current index minus *k*, pop the queue from the head;

4. Enqueue the index each time with the current index (we cannot tell whether it is useless at present);
5. If the current index is larger than *k* (the window is built after that), start printing out the result.

Time Complexity Analysis:

No extra functions in the code.

There's only one for-loop iterate n times, which conducts inputting and dealing data at the same time (After analysis, I think whether use another for-loop for input or input in the process makes no difference since there must be that input no matter where to place it.). Although there are 2 while-loops in the main loop, the total number of iterations of the first while-loop and the second while-loop must be less than or equal to *n-k* since there are at most *n-k* elements to be popped, and other parts are constant. The worst case that *k=1*, the time complexity, ignoring the constant part is roughly O(2n-1), which is O(n). Thus, the total time complexity is O(n).

Advantages of Algorithm One:

1. Fully use of operation;
2. Easy to understand.

Algorithm Two: (The Best Solution of LeetCode Problem 259)

(I checked this during the tutorial, which is before the assignment.)

1. Use variables to denote maximum value and its corresponding index;
2. Use 2 pointers of indices to represent the left bound and right bound of the "slicing window", which increment in each iteration to represent the moving of the "window";
3. The index of the maximum value is larger than the left pointer means that the maximum value is still in the "window". In this case, we only check if the newly entered value (pointed by the right pointer) is larger or equal to the maximum value and note it as the maximum value if it is;
4. The index of the maximum value is less than the left pointer means that it leaves from the left of the "window". In this case, iteratively go over every value between the left and right pointer for the maximum value. We can check the left pointer and left pointer if they are equal or only 1 less than the original maximum value (left one cannot be equal) to reduce calculations in certain cases.

Time Complexity Analysis:

The time complexity of the worst case, if all numbers are in descending order and difference between all numbers are at least 2 (means there is possibility that there exists numbers larger than the new left value for the algorithm, which avoids the "short path"), could be O(n-k) times O(k), because the window only moves *n-k* times and its length is *k*. The worst of the worst could

be $k = \frac{n}{2}$, which is O ($\frac{n^2}{2}$). Thus, the time complexity of the worst case is O(n^2), but the average time complexity could be O(n) by avoiding looping if not in extreme cases.

Advantages of the Algorithm Two:

1. Even more intuitive since no popping operations are contained;
2. Less memory space occupied since there's no imports of data structures;
3. It could be even faster since incrementing integer variable and check array value from index is much faster than popping and peeking operations in queue (and it does in test cases of OJ).

Problem 3

Language: Java

Observations:

1. To obtain the minimum number, we start from the largest digit to delete numbers;
2. We delete large numbers in large digits and leave the smaller ones for minimum in largest digits. Therefore, we hope to obtain the digits in an ascending order before we delete *k* numbers from the largest digit to the smallest.

Possible Solution:

Enumeration (O (C_k^n)).

Algorithms: Greedy Algorithm

1. Use a stack for storing numbers. Push in numbers in order of the digit from largest to smallest;
2. Each time before the push in operation, compare the number to be pushed in with the top of the stack number. If the top number is smaller than the current one, pop it out until the number is not bigger than the current one (thus in ascending order) or already popped out *k* numbers;
3. If the number of digits that are deleted is less than *m* after appending all digits into the stack, delete the largest number from behind (the smallest digit to the largest);
4. Use a double-ended queue instead of the stack since its easier to obtain the final solution in correct order.

Time complexity Analysis:

In the data processing part, there is one for-loop as the main loop containing one while-loop and a while-loop outside for fix-up cases. The total iteration time of the sum of 2 while-loops is *m* since only *m* numbers need to be removed while the for-loop iterates *n* times since there are *n*

numbers to be appended. Thus, the total time complexity of this part, ignoring constant parts, is roughly O(n+m).

There is also a for-loop in the output part of O(n-m) time complexity since only *n-m* digits were not deleted and need to be output.

In all, the total time complexity is O(2n), which is O(n).

Advantage of the Algorithm:

The dominant advantage is the low time complexity, representing high efficiency.