Problem 1

1. Possible Solutions for the Problem

Brute Force: Check each candidate to see if they meet the winner criteria by iterating over the entire matrix for each candidate. This approach has a time complexity of $O(n^2)$, making it inefficient for large matrices.

Graph Theory: Model the problem as a directed graph where nodes represent candidates, and edges represent the direction of note sending. The winner is a "sink" node, receiving edges from all other nodes but not sending any. Algorithms like Tarjan's or Kosaraju's could identify such nodes but are more complex to implement.

2. Solution Explanation

The solution provided in the program follows an optimized linear search strategy, broken down into two main phases:

Hypothesis Formation: Starting with an assumption that the first candidate is the winner, the algorithm iteratively compares the assumed winner with every other candidate. If the assumed winner has sent a note to any candidate, that candidate becomes the new assumed winner. This phase identifies a potential winner based on outgoing notes.

Validation: After hypothesizing a winner, the algorithm checks two conditions for each candidate against the hypothesized winner: the winner must not have sent them a note, and every other candidate must have sent a note to the winner. If these conditions are not met for any candidate, the algorithm concludes that there is no winner (-1).

3. Advantages of the Chosen Solution

The chosen solution offers several advantages over alternative methods:

Efficiency: By reducing the problem to two linear scans of the matrix, the solution operates with $O(n)$ complexity, which is significantly more efficient than a brute-force $O(n^2)$ approach, especially for large matrices.

Simplicity: The algorithm is straightforward to implement and understand, avoiding the complexity and overhead of graph-theoretical approaches or extensive data structures.

Scalability: Due to its linear time complexity, this solution scales well with increasing problem sizes, maintaining reasonable execution times even as the number of candidates grows.

Problem 2

1. Possible Solutions for the Problem

Brute Force: For each window of size k, iterate through the window to find the tallest house. This approach has a time complexity of O(nk), where n is the number of houses.

Segment Tree: Build a segment tree to find the maximum height in a range efficiently. Though segment trees can find the maximum in O(log n) time, building the tree initially takes O(n) time, making this approach more complex and potentially less efficient for problems where the window moves by one position at a time.

2. Solution Explanation

The program solves the problem using a deque to efficiently manage the sliding window, following these steps:

Initialization: Disable synchronization between C++ streams for faster I/O. Read the number of houses (n) and the window size (k). Input the heights of the houses.

Sliding Window Management: Iterate through each house, using the deque to maintain a list of candidate indices for the tallest house within the current window. The key operations are:

Pruning: Remove indices from the back of the deque if they are outside the current window (i - k).

Maintaining Order: Remove indices from the front if their corresponding house heights are less than the height of the current house, ensuring the deque always has the tallest house at the back.

Updating Window: Add the current index to the front of the deque.

Recording Tallest House: Once the window is fully within the row of houses (i >= k - 1), append the height of the house at the back of the deque to the results.

Output: Print the height of the tallest house visible in each window as the window slides from the start to the end of the row.

3. Advantages of the Chosen Solution

Efficiency: With a time complexity of O(n), it is significantly faster than the brute force approach, especially for large values of n and k.

Memory Usage: The deque only stores indices within the current window, making it memory efficient compared to storing the heights directly or building a segment tree.

Simplicity and Directness: Unlike segment trees, which require a complex setup and understanding of tree data structures, deques are simple to use and understand, making the code easier to read and maintain.

Problem 3

1. Possible Solutions for the Problem

- **Brute Force:** Generate all possible numbers by removing `m` digits and then comparing them to find the smallest one. This approach, while straightforward, has an exponential time complexity and is impractical for large numbers or high values of `m`.

- **Dynamic Programming:** Solve the problem by breaking it down into smaller subproblems, storing solutions to these subproblems to avoid recalculating them. This can be more efficient than the brute force approach but can still be complex to implement and understand, especially for those unfamiliar with dynamic programming.

- **Greedy with Stack:** As implemented in the provided program, this approach uses a stack to maintain a list of digits in a way that ensures the resulting number is the smallest possible after removing `m` digits. This method is both efficient and relatively simple to implement, operating in linear time relative to the length of the number.

2. Solution Explanation

The program uses a monotone stack to solve the problem, outlined in the following steps:

- **Input: R**ead the number as a string (`s`) and the count of digits to remove (`m`).

- **Building the Smallest Number:** Iterate through each digit of the input number. For each digit:
  - If the current digit is smaller than the digit on the top of the stack and we still have digits to remove (`m > 0`), pop the top of the stack (remove the larger digit) and decrement `m`.
  - Push the current digit onto the stack.

- **Removing Extra Digits: If** there are still digits to remove after processing the entire number (meaning `m > 0`), remove the top elements of the stack, which corresponds to removing the largest remaining digits at the end of the number.

- **Output the Result: T**ransfer the digits from the stack to an output array in reverse order, since the stack stores them in reverse. Then, iterate through the array to print the digits, skipping leading zeros.

3. Advantages of the Chosen Solution

- **Efficiency:** The algorithm has a linear time complexity, O(n), where n is the number of digits in the input number. This makes it much faster than brute force and more straightforward than dynamic programming for large inputs.

- **Simplicity: T**he logic behind using a stack to maintain the desired order of digits is easy to understand and implement, making this approach accessible even to those with a basic understanding of data structures.