H: Handing Over

Problem author: Mike de Vries

Problem: n people with wingspans $w_1, \ldots, w_n$ are seated around a table. People i and j can hand each other condiments if $w_i + w_j \geq \min(|i-j|, n-|i-j|)$. Answer q queries of the form: how many handovers are needed to pass from i to j?

H: Handing Over

Problem author: Mike de Vries

Problem: n people with wingspans $w_1, \ldots, w_n$ are seated around a table. People i and j can hand each other condiments if $w_i + w_j \geq \min(|i-j|, n-|i-j|)$. Answer q queries of the form: how many handovers are needed to pass from i to j?

Strategy: Pick one of the two possible directions to handover. Then repeatedly handover to the person within reach that can reach the farthest in the desired direction, until the destination can be reached.

H: Handing Over

Problem author: Mike de Vries

Problem: n people with wingspans $w_1, \ldots, w_n$ are seated around a table. People i and j can hand each other condiments if $w_i + w_j \geq \min(|i-j|, n-|i-j|)$. Answer q queries of the form: how many handovers are needed to pass from i to j?

Strategy: Pick one of the two possible directions to handover. Then repeatedly handover to the person within reach that can reach the farthest in the desired direction, until the destination can be reached.

Edge case: For the first handover, it can be optimal to handover in the opposite direction if there is a person with a very large wingspan. The final handover can also be in the opposite direction.

H: Handing Over

Problem author: Mike de Vries

Problem: n people with wingspans $w_1, \ldots, w_n$ are seated around a table. People i and j can hand each other condiments if $w_i + w_j \geq \min(|i-j|, n-|i-j|)$. Answer q queries of the form: how many handovers are needed to pass from i to j?

Strategy: Pick one of the two possible directions to handover. Then repeatedly handover to the person within reach that can reach the farthest in the desired direction, until the destination can be reached.

Edge case: For the first handover, it can be optimal to handover in the opposite direction if there is a person with a very large wingspan. The final handover can also be in the opposite direction.

Solution: First determine for each person and each of the two directions the optimal first handover, and then apply binary lifting on this.

H: Handing Over

Problem author: Mike de Vries

Problem: n people with wingspans $w_1, \ldots, w_n$ are seated around a table. People i and j can hand each other condiments if $w_i + w_j \geq \min(|i-j|, n-|i-j|)$. Answer q queries of the form: how many handovers are needed to pass from i to j?

Strategy: Pick one of the two possible directions to handover. Then repeatedly handover to the person within reach that can reach the farthest in the desired direction, until the destination can be reached.

Edge case: For the first handover, it can be optimal to handover in the opposite direction if there is a person with a very large wingspan. The final handover can also be in the opposite direction.

Solution: First determine for each person and each of the two directions the optimal first handover, and then apply binary lifting on this.

Todo: How do we determine the optimal first handovers? And how do we apply (and what even is) binary lifting?

H: Handing Over

Problem author: Mike de Vries

Step 1: We must determine for each person and each of the two directions the optimal first handover.

H: Handing Over

Problem author: Mike de Vries

Step 1: We must determine for each person and each of the two directions the optimal first handover.

Approach: Duplicate the array of wingspans and go right to left. Keep a stack (variable sized array) with each person whose reach is not a subset of another person's reach, sorted by index.

H: Handing Over

Problem author: Mike de Vries

Step 1: We must determine for each person and each of the two directions the optimal first handover.

Approach: Duplicate the array of wingspans and go right to left. Keep a stack (variable sized array) with each person whose reach is not a subset of another person's reach, sorted by index.

For the people in the original half of the array, the top of the stack gives the optimal first handover for the case that we want to move left, but the first handover is to the right. Then with binary search, you can find the optimal first handover for the case that we want to move right, and the first handover is to the right.

H: Handing Over

Problem author: Mike de Vries

Step 1: We must determine for each person and each of the two directions the optimal first handover.

Approach: Duplicate the array of wingspans and go right to left. Keep a stack (variable sized array) with each person whose reach is not a subset of another person's reach, sorted by index.

For the people in the original half of the array, the top of the stack gives the optimal first handover for the case that we want to move left, but the first handover is to the right. Then with binary search, you can find the optimal first handover for the case that we want to move right, and the first handover is to the right.

Then remove all people from the top of the stack that are obsoleted by the new person, and add the new person to the top if they are not obsolete themselves.

H: Handing Over

Problem author: Mike de Vries

Step 1: We must determine for each person and each of the two directions the optimal first handover.

Approach: Duplicate the array of wingspans and go right to left. Keep a stack (variable sized array) with each person whose reach is not a subset of another person's reach, sorted by index.

For the people in the original half of the array, the top of the stack gives the optimal first handover for the case that we want to move left, but the first handover is to the right. Then with binary search, you can find the optimal first handover for the case that we want to move right, and the first handover is to the right.

Then remove all people from the top of the stack that are obsoleted by the new person, and add the new person to the top if they are not obsolete themselves.

Finally, reverse the array and repeat the process to get the optimal first handovers in the opposite directions.

H: Handing Over

Problem author: Mike de Vries

Step 1: We must determine for each person and each of the two directions the optimal first handover.

Approach: Duplicate the array of wingspans and go right to left. Keep a stack (variable sized array) with each person whose reach is not a subset of another person's reach, sorted by index.

For the people in the original half of the array, the top of the stack gives the optimal first handover for the case that we want to move left, but the first handover is to the right. Then with binary search, you can find the optimal first handover for the case that we want to move right, and the first handover is to the right.

Then remove all people from the top of the stack that are obsoleted by the new person, and add the new person to the top if they are not obsolete themselves.

Finally, reverse the array and repeat the process to get the optimal first handovers in the opposite directions.

Performance: This step takes $\mathcal{O}(n \log n)$ time.

Step 2: Binary lifting.

Step 2: Binary lifting.

Setup: We know for each person the optimal first move in each direction. If we know for each person the optimal first 2^k moves in each direction, we can combine two such sequences to form the optimal first 2^{k+1} moves in each direction.

H: Handing Over

Problem author: Mike de Vries

Step 2: Binary lifting.

Setup: We know for each person the optimal first move in each direction. If we know for each person the optimal first 2^k moves in each direction, we can combine two such sequences to form the optimal first 2^{k+1} moves in each direction.

Performance: This costs $\mathcal{O}(n \log n)$ time and memory to determine and store.

H: Handing Over

Problem author: Mike de Vries

Step 2: Binary lifting.

Setup: We know for each person the optimal first move in each direction. If we know for each person the optimal first 2^k moves in each direction, we can combine two such sequences to form the optimal first 2^{k+1} moves in each direction.

Performance: This costs $\mathcal{O}(n \log n)$ time and memory to determine and store.

Queries: To go from i to j, we consider both directions. We repeatedly pick the largest k for which the optimal first 2^k moves does not overshoot the target.

Step 2: Binary lifting.

Setup: We know for each person the optimal first move in each direction. If we know for each person the optimal first 2^k moves in each direction, we can combine two such sequences to form the optimal first 2^{k+1} moves in each direction.

Performance: This costs $\mathcal{O}(n \log n)$ time and memory to determine and store.

Queries: To go from i to j, we consider both directions. We repeatedly pick the largest k for which the optimal first 2^k moves does not overshoot the target.

Performance: Each query is answered in $\mathcal{O}(\log n)$ time.

H: Handing Over

Problem author: Mike de Vries

Step 2: Binary lifting.

Setup: We know for each person the optimal first move in each direction. If we know for each person the optimal first 2^k moves in each direction, we can combine two such sequences to form the optimal first 2^{k+1} moves in each direction.

Performance: This costs $\mathcal{O}(n \log n)$ time and memory to determine and store.

Queries: To go from i to j, we consider both directions. We repeatedly pick the largest k for which the optimal first 2^k moves does not overshoot the target.

Performance: Each query is answered in $\mathcal{O}(\log n)$ time.

Running time: The complete algorithm requires $\mathcal{O}((n + q) \log n)$ time and $\mathcal{O}(n \log n)$ memory.

H: Handing Over

Problem author: Mike de Vries

Step 2: Binary lifting.

Setup: We know for each person the optimal first move in each direction. If we know for each person the optimal first 2^k moves in each direction, we can combine two such sequences to form the optimal first 2^{k+1} moves in each direction.

Performance: This costs $\mathcal{O}(n \log n)$ time and memory to determine and store.

Queries: To go from i to j, we consider both directions. We repeatedly pick the largest k for which the optimal first 2^k moves does not overshoot the target.

Performance: Each query is answered in $\mathcal{O}(\log n)$ time.

Running time: The complete algorithm requires $\mathcal{O}((n + q) \log n)$ time and $\mathcal{O}(n \log n)$ memory.

Statistics: 24 submissions, 0 accepted, 17 unknown