# Infection – Solution

**Author**: Atanas Dimitrov

**Tester**: Emil Indjev

## Subtask 1 – $O(n^3)$ when $D_i = 0$

This subtasks requires a naive simulation of the procedure. Since all people start their party-tour at $D_i = 0$ we simulate the whole movement day-by-day for all $O(n)$ days. Each person $i$ moves at most $n$ times as they follow the simple path from $S_i$ to $T_i$ for a total of $O(n^2)$ such steps. Checking whether a person becomes infected at the end of some day $j$ can be in $O(1)$ by marking infected vertices. The last important part is finding the next city every person will move to in the case their tour has not ended. This subtask requires doing this in $O(n)$ either by a simple BFS or DFS. Therefore the final time complexity is $O(n^3)$ and implemented in `author_n3.cpp`.

## Subtask 2 – $O(n^2)$ when $D_i = 0$

The only difference between this subtask and the previous is that one can precompute the whole path from $S_i$ to $T_i$ for each person in $O(n^2)$, thus finding the next step in the tour becomes $O(1)$ for a total of $O(n^2)$ since we are still looping over all of the days. This is implemented in `author_n2.cpp`.

## Subtask 3 – $O(n^2)$

We can notice that we only need to simulate days when something is actually happening, i.e. if no person is moving at the current day we can just skip until some person starts their tour. This can be done either in $O(n \log(n))$ by sorting the people in the beginning or naively in $O(n^2)$.

## Subtask 4 – $O(n \log(n))$ when $D_i = 10^6 i$

In this subtask no 2 people ever meet (a very lonely case), thus any uninfected person stays uninfected until they leave. It is clear that the answer is either the distance between $S_i$ and $T_i$ or 0 depending on whether the person began as infected, which can be computed using either binary lifting or heavy-light decomposition in $O(n \log(n))$. A sample solution can be found in `author_nonintersecting.cpp`.

## Subtask 5 – $O(n \log(n))$ when $D_i = 0$ and line-tree

The subtasks for line-trees are where the main difficulty in the task lies. We make the following observation since we will now be trying to find all times an infection happens:

**Observation 1:** A person can only get infected once.

We will denote people as moving left or right along the line (depending on whether $S_i > T_i$ or $S_i < T_i$).

**Observation 2:** When $D_i = 0$, a person either becomes infected at time 0 or gets infected by a person moving in the opposite direction.

Now examine the following case: 2 people having $(S_1, T_1, I_1) = (1, 4, \text{true})$ and $(S_2, T_2, I_2) = (4, 1, \text{false})$. Even though these people follow the same path, with differing directions, they never meet and they do not transfer the infection.

**Observation 3:** When $D_i = 0$ two people $a$ and $b$ can only meet if $S_a \equiv S_b \pmod 2$.

These 3 observations lead us to have the following solution:

We divide the task in 2 unrelated subproblems – the people having $S_i \equiv 0 \pmod 2$ and those having $S_i \equiv 1 \pmod 2$. From this point on we only deal with the case of $S_i \equiv 0 \pmod 2$ as the other case is exactly the same.

For each such case we keep 2 data-structures (we will discuss the exact details later) called interaction chains – one that takes care of the right-moving infected people and the left-moving uninfected people and another that takes care of the right-moving uninfected people and the left-moving infected people. Additionally we will process "events" one-by-one – an event is either some person getting infected and changing their interaction chain or some person leaving their current interaction chain, since they are at their designated destination. We now explain how to implement the interaction chain where infected people are right-moving and uninfected people are left-moving as the implementation of the other chain is symmetrical.

At the high-level we would like to keep a sorted list of the people and their current positions in the line. After a person leaves (or becomes infected and moves into the other interaction chain) they get removed from this list. We now need to take care of the 2 immediate neighbors of the removed person in the list (note they can be arbitrarily far in the line) and check whether they introduce a new potential future infection event. The case in which this happens is: suppose the removed person was at position $P_r$ and had two immediate neighbors: WLOG an infected one at position $P_a < P_r$ and an uninfected one at position $P_b > P_r$, then we need to emit a new event that will happen after $\frac{|P_b - P_a|}{2}$ days, since the 2 people are moving towards each other and closing the gap by 2 positions per day. Inserts in the interaction chain can be done by a similar procedure – when we insert a person we check whether we need to emit new events for them and 1 of their 2 immediate neighbors.

For the concrete implementation of a single interaction chain one can use 2 `std::set`-s – one for the right-moving infected people and the left-moving uninfected people. Since all people move with constant speeds ($\pm 1$ depending on the direction) we can use a global offset to handle those movements. To check whether a future potential infection event needs to be added, one needs to search for the position of the intermediate neighbors in the 2 `std::set`-s separately.

Finally to conclude the analysis we note that each person gets inserted into an interaction chain at most once, gets moved between the 2 chains at most once and exits (as their tour has ended) at most once. Since each of those operations can be implemented in $O(\log(n))$ and we store the events in a `std::priority_queue` we get total time complexity $O(n \log(n))$.

## Subtask 6 – $O(n \log(n))$ and line-tree

The solution to this subtask is mostly the same as subtask 5, but the only change is that we need to consider people getting inserted at arbitrary times.

We will modify the observations we made before to accommodate this change:

**Observation 2 (revisited):** A person $a$ becomes infected either at time $D_a$, at time $D_b$ for some newly inserted and infected person $b$ or by a person moving in the opposite direction.

**Observation 3 (revisited):** Two people $a$ and $b$ can only meet if $S_a + D_a \equiv S_b + D_b \pmod 2$.

Thus the only change we need to make to the solution in subtask 5 is to keep the separated subproblems based on $S_a + D_a \pmod 2$ and to introduce an insertion event (apart from the infection and removal event) that handles a person starting their tour and possibly infecting someone moving in the same direction. This can be done without any asymptotic slowdown in $O(n \log(n))$ and is implemented in `emo_line.cpp`.

## Subtasks 7 and 8 – $O(n \log^2(n))$

Subtask 7 serves the purpose of giving points to slower implementations of the full solution and thus we focus on subtask 8.

Since we have a (relatively) clean solution for a line-tree we would like to find some way to convert the tree into a set of chains. A standard approach to do this is finding the heavy-light decomposition (HLD) of the tree. It decomposes the tree into chains, such that each simple path moves through at most $O(\log(n))$ such chains. Therefore we can split each person's tour into $O(\log(n)$ "subtours" along the path they visit and using the solution from subtask 6 as a black-box simulate what happens in each such "subtour". Note that we need to take care of all those events sequentially and thus keep a global priority queue for all chains. The total number of events is now $O(n \log(n) + n) = O(n \log(n))$ and since each event takes $O(\log(n))$ time the final complexity is $O(n \log^2(n))$.

The full implementation of the problem is done in `author_nlog2.cpp` and `emo_nlog2.cpp`.