Virus infection

We're given a tree with n vertices and exactly $1 \leq k \leq n$ infected vertices initially and we can only infect one new vertex from a currently infected vertex and we want to know the minimal time to infect all the vertices

Here I will propose a solution that works for any value of k

First of all we'll look at some simpler cases that are needed in the generalized version:

Special cases

K = 1

If we only have one infected vertex it is easy to come up with a dp that solves the problem in O(n): When computing the answer for a specific node we'll look at all of my children and calculate their dp values recursively and then save those values in a vector S

It is easy to see that in the optimal answer the order in which we visit those vertices depend only on the values of their dp's, and we'll visit them in decreasing order, because if we have a pair of vertices such that the one with smaller dp come first we could always swap them and the answer would always be less than or equal to the original one. So what we can do is simply sort all of the values in decreasing order and then transform each element of this set: $\sum_{i=0}^{i<|S|} S_i := S_i + i + 1$, because since we have defined their order we'll consume 1 unit to visit the first element, 2 units to visit the second and so on

This way we can calculate the dp for all the values and then find the value of the root and the answer

K = 2

For this case I'll define a couple of terms:

- Active vertices: Those are the vertices that are either initially infected or are part of the shortest path between two initially infected vertices
- Dead vertices: Those are the ones that are not active, which means, those that do not belong to the path of any two initially infected vertices.
- A property of all dead vertices is that each one of those "belong" to some active vertex, which for each dead vertex is exactly the active vertex that is closest to him

Now what we want to do is binary search the value of the answer; for this situation I'll introduce a new variable, which will be the limit of the answer LIM and we want to know for each given LIM whether it is possible to achieve such answer

In this case with K=2 we have that the graph of active vertices, which I'll define as G, is a line and the starting vertices are on the leafs of this graph, which is where we'll start our procedure.

Now suppose that I am at vertex V, which is one of the leafs of G, and that we arrived at him at time $T \geq 0$, also we will need to calculate the set S (already modified and ordered version of it) considering the whole tree as being vertex V and all the dead vertices that belong to V, which I'll define as D_V. Now we have exactly two options of the next move, either we'll move to one the vertices in D_V or we'll move to our only neighbor $U \in G$, which is uniquely defined because V is a leaf of G and therefore has only one neighbor. Now we will consider three different cases depending on the values of $LIM, max(S), T$ and find the best move in each one of those:

- $T + max(S) > LIM$

  In this situation it is clear that we are not able visit all the element of D_V in time LIM so we can simply return this procedure and not mark the vertex V as visited, because we cannot clear its subtree, and if we're ever going to visit all of his subtree we would either need a bigger LIM or have arrived at him with a time smaller than T.

- $T + max(S) < LIM$

  In this case, since the time needed to clear the most expensive element in S is smaller than LIM and we only need to pay 1 unit to visit U it is never a bad decision to first infect U and then infect the rest of my subtree, because the time will never go over LIM inside D_V and we will start the procedure at U with a smaller time. Therefore we can mark V as visited and continue the procedure at U with time $T + 1$ considering now him as a new leaf of the line graph, since V was removed.

- $T + max(S) = LIM$

  This situation is a bit tricky because we can no longer visit U and then continue in D_V , because doing so would add 1 to every element of S and thus increase the answer above LIM, so we are forced to visit the first element of our set S. After that move we achieved a similar configuration which can be analysed as one of the three cases defined above, considering a state with vertex V, time $T + 1$ and set $S' = S \setminus S_0$, where S_0 is the first element of S that we visited in this move. When implementing such step we can actually precalculate what is that prefix that we need to remove and do this step in $O(1)$, by moving to state $U, T + 1 + Removed$, where $Removed$ is the number of elements that we had to remove from the prefix of S for us to be able to move to U.

  > Note that if the graph consists only of V we don't actually move to U because it doesn't exist and we just mark V as completed depending on which of the above case that we fall into and then terminate.

Now we have achieved an algorithm to calculate efficiently how far an initial vertex can go considering the limit we have defined in the binary search, and using it we can achieve a $O(n \cdot log(n))$ solution for this special case because that's the whole process we need, by checking after each iteration of the binary search whether we could visit all the active vertices and therefore the all of the dead, because for us to mark an active vertex we also must be able to visit all of its dead vertices.

General case $1 \leq K \leq n$

Once again, I'll have to define a new term:

- Confusing vertices are those vertices that are not leaves in $Condensed(G)$, where $Condensed(G)$ means creating the virtual or also sometimes called condensation tree of G in which the only active vertices that we want to condense are the initially infected ones, and they have such name because when we reach them it may be possible that we have multiple possible next moves inside of G, and therefore applying the special case of the line wouldn't work for them

Note that we can divide $Condensed(G)$ into a set of disjoint paths, in which we could apply the special case of a line.

So let's take all of those disjoint paths that start at some leaf, let's call it V (which by the construction of $Condensed(G)$ must be an infected vertex) and end at some confused vertex, let's call it C (in case there are no confused vertices we know that graph is a line and then we can solve it using the previous special case for K=2) and apply our algorithm for a line starting V, and then remove V from the graph according to one of those cases.

There are two possibilities that could happen:

- The algorithm starting from V did not mark all of the vertices in the path between V and C (I*)

  Since we need to cover all the vertices in the graph, there must be some infected vertex coming from C that would infect the rest of those vertices, and we can safely assume that the vertices that were filled by V in this iteration of the algorithm are optimal, because they must be filled either by C or V, and since V can already fill it, choosing not to and adding this set of vertices to C would only make the whole process equal or worse than it was when choosing those vertices to be filled by V before. So what we can do is take all the vertices that were not filled by V and update them as dead vertices that belong to C, because since we have defined that V cannot fill them C must do it.

- The other situation is that the algorithm indeed filled all the vertices in the path and arrived at C at time T_V, which we will just save in a set ST_C for now, in which will save the times of all the successful arrivals at C from some V

Now we have some confusing vertices that have had all of their leaves removed are now leaves themselves. Once again we have two possible situations for them, which we deal separately (We will call this confusing vertex C again).

- No leaf that was a child of C managed to arrive at it and C wasn't initially an infected vertex

  In this case, we know that C isn't an infected vertex and none of its children could infect it, so we can simply say that he is a dead vertex of his parent in $Condensed(G)$, for the same reason argued in (I*)

- Some child of C managed to arrive at it or C itself was initially infected

  In this situation we can always say the time T in which this vertex was first infected by some children (or by itself if it was initially infected, in which case $T = 0$), and it is always optimal to take the smallest time because waiting more time to take another child would never be optimal because that first child has already visited all of its dead vertices and if we did not choose to move with him we would simply be wasting a move because the other non-minimal element would come at a later time and therefore we would have the same situation with C as a vertex but with a time $T' > T$, and that is never optimal since we want to minimize the time to visit every vertex.

Now having this time T fixed we once again have two possibilities that could happen

- $T + max(S) > LIM$, where S is calculated now with the modified version including all the dead vertices appended to him by some of his children that were initially present in G. In this case we know for sure that if we're starting at C at time T we cannot fill in all the dead vertices, which means that we cannot achieve an answer starting at C and being infected from below, therefore it there is some answer we can safely say that it must come from above, that is, from its parent on G coming downwards and infecting D_C. So we can transform C and all of his dead children into a big group of dead children and append it to his parent in $Condensed(G)$
- $T + max(S) \leq LIM$. Now we know that we can safely fill D_C using the vertex considered above, and for the argument expressed in (I*) we know that it is always optimal to take him instead of letting some vertex from above come and fill in what I could do, and now we can create a new infected vertex with initial time T.

We can continue recursively with this algorithm until we have only one or two infected vertices, which is a special case that we have already covered and know how to solve.

This time the complete algorithm runs in $O(n \cdot log^2(n))$ because we need to calculate the Sets S because as we've seen the sets D change dynamically depending on some cases of the algorithm, which is itself multiplied by the log given by the binary search.

And that concludes the proof. ■