Task 1: Cryptography (`crypto`)

Authored and prepared by: Ho Xu Yang, Damian

Subtask 1

Limits: $N = 2$

If $P_1 < P_2$ the answer is 1, otherwise it is 2.

Time Complexity: $O(1)$

Subtask 2

Limits: $1 \leq N \leq 8$

We can enumerate all $N!$ possible permutations of P in lexicographical order (with next_permutation) and count the number of permutations until we get P back.

There are N choices for the first number, $(N-1)$ choices for the second number, and so on until we have 1 choice for the last number, giving us $N \times (N-1) \times \ldots \times 1 = N!$ ways in total.

Time Complexity: $O(N \times N!)$

Subtask 3

Limits: P is either strictly increasing or decreasing.

If P is strictly increasing, it is the lexicographically smallest permutation and the answer is 1.

Otherwise, it is the lexicographically largest permutation and the answer is $N!$.

To calculate $N!$, we can use a for-loop to compute $\prod_{x=1}^{N} x$, taking care to mod the value by 1 000 000 007 after every multiplication to avoid integer overflow.

Time Complexity: $O(N)$

Subtask 4

Limits: $P = [k, 1, \ldots, k-1, k+1, \ldots, N]$ where $1 \leq k \leq N$

Consider $P_1 (= k)$.

The lexicographically smallest permutations will consist of those where $P_1 = 1$, and with $(N-1)$ digits remaining, there are a total of $(N-1)!$ such permutations.

Thus, we have $(N-1)!$ permutations where $P_1 = 1$, $(N-1)!$ permutations where $P_1 = 2$ and so on. In particular, the lexicographical order of a permutation with $P_1 = k$ is between $(k-1) \times (N-1)! + 1$ and $k \times (N-1)!$.

However, since $[1, \ldots, k-1, k+1, \ldots, N]$ is in sorted order, P is the lexicographically smallest permutation with $P_1 = k$.

Hence, the answer is $(k-1) \times (N-1)! + 1$.

Time Complexity: $O(N)$

Subtask 5

Limits: $1 \leq N \leq 3 \times 10^3, 1 \leq P_i \leq N$

Using the idea from subtask 4, we can calculate the lexicographical order of P element by element, by computing its lexicographical order due to P_1, before treating it as a permutation $P_2, \ldots, P_N$ of length $(N-1)$ and repeating the process.

We begin with the set of integers $T = \{1, \ldots, N\}$ and do the following for $i = 1, \ldots, N$:

1. Find the *order* of P_i in T (Number of elements in T that are smaller than or equal to P_i)
2. Add $(order - 1) \times (N-i)!$ to the answer
3. Remove P_i from T

We can implement T with a boolean array of size N, where $T_i = 1$ if i is still in T and 0 otherwise. We can compute the order of P_i via the sum $\sum_{i=1}^{P_i}(T_i = 1)$, and remove P_i from T by setting $T_{P_i} = 0$.

Time Complexity: $O(N^2)$

Subtask 6

Limits: $1 \leq N \leq 3 \times 10^3$

We use a similar approach to Subtask 5, except we initialise T as an integer array that contains the elements of P.

We can compute the order of P_i via the sum $\sum_{i=1}^{N}(T_i \leq P_i)$, and remove P_i from T by changing the value of P_i in T to ∞.

Time Complexity: $O(N^2)$

Subtask 7

Limits: $1 \leq P_i \leq N$

So far, the bottleneck has been the computation of the order of each P_i. (Note that we can precompute the values of $i!$ for all $1 \leq i \leq N$ quickly since $(i+1)! = (i+1) \times i!$)

To do so, we can implement an *order statistic tree* by utilising any data structure that supports range updates and point queries in logarithmic time, such as Fenwick Trees or Segment Trees. We then implement the methods query(X) and update(X) where query(X) denotes the sum of $T_1, \ldots, T_X$ and update(X) sets $T_X = 0$.

Therefore, by using the same logic from subtask 5, we are able to compute the order of P_i via query(P_i) and remove P_i from T via update(P_i) in logarithmic time.

Time Complexity: $O(N \log_2 N)$

Subtask 8

Limits: (No further constraints)

The solution for subtask 7 fails as $\max P_i = 10^9$. However, note that only the relative magnitudes of P_i matter in computing their orders. Thus, we can discretise each P_i into the range $[1, N]$ (by setting the smallest P_i to 1, second smallest P_i to 2, etc.) before using the approach from subtask 7.

Time Complexity: $O(N \log_2 N)$

Task 2: Fuel Station (`fuelstation`)

Authored by: Lim An Jun

Prepared by: Ho Xu Yang, Damian

Preliminaries

We will assume that $X_i \leq X_{i+1}$ by first sorting the input.

Furthermore, we add a $(N+1)^{th}$ fuel station ($\underbrace{\text{D}}_{\text{X}}, \underbrace{0}_{\text{A}}, \underbrace{\text{D}}_{\text{B}}$) to represent our destination.

Finally, we will denote the required minimum possible value of F as F_{min}.

Subtask 1

Limits: $N = 1$

With one fuel station, we can choose to either use it or not.

To use the fuel station, there are the following restrictions on F:

1. F must be large enough to reach the fuel station: $X \leq F$
2. F must be small enough to use the fuel station: $F \leq B$
3. F must be large enough to reach the destination: $D \leq F + A \implies D - A \leq F$

Together, we get $\max(X, D-A) \leq F \leq B$.

If the range is non-empty (i.e. $\max(X, D-A) \leq B$) we get $F_{min} = \max(X, D-A)$.

Otherwise, it is not possible to use the fuel station, and we have $F_{min} = D$ in the case where we travel directly to our destination.

Time Complexity: $O(1)$

Subtask 2

Limits: $B_i = 10^9$

In this subtask, every fuel station can always be used.

We simulate our journey by iterating through the fuel stations; we will have $F_{remaining} = F - X_i + \sum_{X_j < X_i} A_j$ litres of fuel left at station i. (The $\sum_{X_j < X_i} A_j$ term can be computed through a pointer walk).

In order to reach our destination (i.e. fuel station $N+1$), we must have $F_{remaining} \geq 0$ at every fuel station.

Thus, we need $F \geq X_i - \sum_{X_j<X_i} A_j$, i.e. $F_{min} = \max(X_i - \sum_{X_j<X_i} A_j)$.

Time Complexity: $O(N \log_2 N)$

Subtask 3

Limits: $1 \leq D \leq 10^4, 1 \leq N \leq 10^4$

As D is small, we can simply try every value of F. For each value of F, we once again simulate our journey.

However, this time we can only use fuel stations with $F \leq B_i$. Thus, $F_{remaining} = F - X_i + \sum_{X_j<X_i, F \leq B_j} A_j$.

In this manner, F_{min} is simply the smallest value of F that allows us to reach our destination.

Time Complexity: $O(ND + N \log_2 N)$

Subtask 4

Limits: $1 \leq D \leq 10^4$

With $N \gg D$, there are many fuel stations for each value of X_i. Hence, we group the fuel stations according to X_i, sort them by B_i and then calculate a prefix sum on A_i. Each group can be treated as a single fuel station: its value of A depends on the current value of F, and can be found by a pointer walk to find $A_{effective} = \sum_{X_j=X_i, F \leq B_j} A_j$.

We then try every value of F as per subtask 3.

Time Complexity: $O(D^2 + N \log_2 N)$

Subtask 5

Limits: $1 \leq N \leq 16$

We enumerate all possible subsets of fuel stations to use. For a particular subset S, we will set $F = min(B_i)$ in order to ensure that all selected fuel stations are usable.

This time, we will only consider fuel stations that are part of S. Hence, $F_{remaining} = F - X_i + \sum_{X_j<X_i, j \in S} A_j$

Now, we make an *Important Observation*:

Important Observation

Consider a value of $F \in [B_{i-1}+1, B_i]$ (where B is sorted). If we are able to reach our destination with an initial amount of F litres of fuel, we will also be able to reach our destination with $F+1, \ldots, B_i$ litres of fuel.
This is because we will start with more fuel while still having access to the same fuel stations.
This implies that we will be able to reach our destination with F equal to the smallest $B_i \geq F_{min}$.

Therefore, for each value of $F = B_i$ that allows us to reach our destination, we binary search on the range $[B_{i-1}+1, B_i]$ to find the smallest F' that still allows us to reach our destination.

As it turns out, we can skip the binary search altogether (but this observation is not essential).

Important Observation (Cont.)

If we are able to reach our destination with F litres of fuel, then we can also reach our destination with $F - \min(F_{remaining})$ litres of fuel.
This is because we will have $F'_{remaining} = F_{remaining} - \min(F_{remaining}) \geq 0$ at all times.
Furthermore, since $F' \leq F \leq B_i$, we can still use the same fuel stations as before.

Hence, for each value of $F = B_i$, we can obtain $F' = B_i - \min(F_{remaining})$.

Regardless of which approach used, we have F_{min} = smallest F' over all possible subsets.

Time Complexity: $O(2^N \times N)$

Subtask 6

Limits: $1 \leq N \leq 10^4$

To speed up our solution from subtask 5, we note that if we are using a fuel station i, we should also use all fuel stations j where $B_i \leq B_j$. Thus, we will fix our values of F rather than calculating them.

We try $F = B_1, \ldots, B_{N+1}$ (in sorted order of B) and simulate our journey as per subtask 3. If we are able to reach our destination, we then apply the *Important Observation* to obtain F_{min}.

Time Complexity: $O(N^2)$

Subtask 7

Limits: (No further constraints)

To obtain full marks for this question, we need to find a faster way to calculate $F_{remaining}$ for each value of F. We can utilise a range-add update, range-minimum query Segment Tree to

update and query the values of $F_{remaining}$ in logarithmic time.

Recall from subtask 3 that $F_{remaining} = F - X_i + \sum_{X_j < X_i, F \leq B_j} A_j$ for each fuel station i.

We begin with $F = \min(B_i)$, allowing us to use all fuel stations. For each station i, we will initialise $F_{remaining} = F - X_i$. Furthermore, all stations j with $X_j > X_i$ will benefit from A_i, hence we do a range update of A_i on $F_{remaining}$ for all such j (value of j can be precomputed for all i).

If we are able to reach our destination (i.e. range minimum ≥ 0), we apply the *Important Observation* to obtain F_{min}.

Otherwise, we continually increase F to the next smallest value B_j until we find a solution.

We increase all values of $F_{remaining}$ by $B_j - B_i$ (since we start with more fuel). However, if $B_j \neq B_i$, we can no longer use all fuel stations k with $B_k = B_i$. Thus, we will have to range update $F_{remaining}$ by $-A_k$ for all fuel stations with $X > X_i$.

Time Complexity: $O(N \log_2 N)$

Task 3: Relay Marathon (`relaymarathon`)

Authored and prepared by: Sidhant Bansal

Preliminaries

Let us relabel the set of special cities as S and the optimal $\text{start}_1, \text{finish}_1, \text{start}_2$, and finish_2 as x_1, y_1, x_2, and y_2 respectively for shorter notation in the explanation below.

Subtask 1

Limits: $4 \leq K \leq N \leq 50$

Run Floyd-Warshall all pair shortest path algorithm on the graph, which runs in $O(N^3)$ to find shortest path between all pairs and in $O(N^4)$ pick 4 vertices as x_1, y_1, x_2, and y_2.

Time complexity: $O(N^4)$

Subtask 2

Limits: $4 \leq K \leq N \leq 500$

Still run a Floyd-Warshall all pair shortest path algorithm. But now instead of doing the $O(N^4)$ method of picking the optimal quadruple, we will do a $O(N^3)$ solution. Fix every pair of (x_1, y_1). Let α and β be the closest and second closest node respectively in S w.r.t y_1. Now iterate over all $x_2 \in S$, if $\alpha \neq x_2$, then try the candidate answer as (x_1, x_2, y_1, α), otherwise try the candidate answer as (x_1, x_2, y_1, β). Over all these candidate answers take the best one.

Time complexity: $O(N^3)$

Subtask 3

Limits:
- City 1 and City 2 both are **special** and directly connected with one another by an edge that takes 1 second to travel.
- City 1 is NOT connected to any other city except City 2.
- City 2 is NOT connected to any other city except City 1.

Since Node 1 and Node 2 both belong to S and are disconnected from the remaining graph, with distance between them being 1. Therefore $(y_1, y_2) = (1, 2)$. And the problem reduces to finding the closest pair of points in S in the remaining graph, i.e to find (x_1, x_2). This problem is solvable in $O(M \log N)$ via **Method 1** and in $O(M \log^2 N)$ via **Method 2** and **Method 3** as

described below.

Method 1: Run a multi-source Dijkstra with all the vertices in S being the source. Now this Dijkstra should give us $dist[]$ array denoting the shortest distance to each node of the graph and $head[]$ array, where $head[u]$ should denote the closest node in set S w.r.t node u. After obtaining this $head[]$ array. Iterate over all the edges in the graph. If an edge is between (u, v) of weight w, where $head[u] \neq head[v]$, then consider $(head[u], head[v])$ as a valid candidate for (x_1, x_2) with distance between them being equal to $dist[u] + w + dist[v]$. Amongst all the candidates pick the minimum one as (x_1, x_2).

Proof of correctness: Let the optimal (x_1, x_2) be denoted by (a, b). Let the vertices in the shortest path from a to b (including both of them) be denoted by $v_1, v_2, \ldots, v_k$. Now let the heads of $v_1, v_2, \ldots v_k$ be denoted by $h_1, h_2, \ldots h_k$ respectively. If there is any h_i, that is not equal to a and not equal to b. Then (a, h_i) or (h_i, b) is a better pair than (a, b) which is a contradiction. So all elements of $h[]$ must be a or b. Also $h_1 = a$ and $h_k = b$. So there must exist at least one index j, such that $h_j = a$ and $h_{j+1} = b$ where $1 \leq j < k$. And when the edge between v_j and v_{j+1} will be considered, this pair between (a, b) will definitely be considered.

Method 2: Make two initially empty sets, set A and set B. For each element in set S, toss a coin and uniformly at random put it either in set A or set B. Eventually connect all the elements of set A with a dummy source (with weight 0) and all the nodes of set B with a dummy sink (with weight 0). Run a dijkstra from dummy source to dummy sink, and for the optimal route maintain which node in set A and set B it passes through. This can be a candidate answer.

Proof of Correctness: The probability of finding the optimal $(x_1, x_2) \geq \frac{1}{2}$, because probability the optimal pair is split into two different sets is $\frac{1}{2}$. So if we run this $2 \log N$ times and take minimum over all the candidates, then this algorithm will fail with probability $\leq \frac{1}{N}$, i.e almost negligible and most likely ACable.

Method 3: The idea used in **Method 2** can be de-randomised, by the following strategy: In the i_{th} iteration, let A be the set of all the elements of S which when written in binary have their i_{th} bit on, and B be the set of all the remaining elements of S (i.e those which when written in binary have their i_{th} bit off). We will have $\log N$ such iterations.

Proof of Correctness: This should ensure that all possible pairs are partitioned into the two opposite sets at-least once. This is because for any two elements, they must differ by at least one bit in their binary representation and therefore must be in opposite sets at least once.

Time complexity: $O(M \log N)$ or $O(M \log^2 N)$

Subtask 4

Limits: No additional constraints

We have two solutions one is a variant of **Method 2** described above which runs in $O(M \log^2 N)$

and the other one is the expected most efficient solution that runs in $O(M \log N)$. Let us first explain the efficient one.

Efficient Solution: First find the closest pair of elements in S (using **Method 1**) and denote it by (a, b), i.e $D(a, b)$ is minimum amongst all possible pairs in S. Now we can claim that a and b both must belong to the set $\{x_1, x_2, y_1, y_2\}$, i.e
Claim 1: $\{a, b\} \subseteq \{x_1, x_2, y_1, x_2\}$.
Proof: Firstly if both don't belong to $\{x_1, x_2, y_1, y_2\}$, then simply replace (x_1, x_2) with (a, b) to get a better answer. So contradiction. Therefore WLOG let $a = x_1$. Now if b doesn't belong to $\{x_2, y_1, y_2\}$, then replace x_2 with b to get a better answer. Therefore contradiction. Hence b must also belong to $\{x_1, x_2, y_1, y_2\}$.
Now we work out two case and our answer will be the minimum amongst those:

Case 1: $(x_1, x_2) = (a, b)$. Then to find (y_1, y_2) we can simply construct graph $G' = G \setminus \{a, b\}$, i.e the original graph from which a and b are removed. On this graph G' if we use **Method 1** again to find the closest pair of nodes in S, then we will obtain the best (y_1, y_2).

Case 2: $x_1 = a, y_1 = b$. Then we already know (x_1, y_1) and we need to find (x_2, y_2), which can be done by calling two dijkstras from x_1 and y_1 respectively, and then finding optimal (x_2, y_2) in $O(N)$ (Using the technique described of finding optimal (x_2, y_2) for a fixed (x_1, y_1) in **Subtask 2** solution)

Only a constant number of dijkstras are required so it runs in $O(M \log N)$

Inefficient randomized solution: For each node in set S, put it in set A, B, C or D uniformly at random. Then join all the nodes in set A and B with new dummy sources (with weight 0), source_1 and source_2 respectively. Similarly join all the nodes in set C and D with new dummy sinks (with weight 0), sink_1 and sink_2 respectively. Then run a Djikstra from source_1 to sink_1 and another one from source_2 to sink_2. Probability you hit the optimal quadruple is $\geq \frac{1}{32}$. So if we run this algorithm $\frac{32}{\log} N$ times, then this algorithm will fail with probability $\leq \frac{1}{N}$, i.e negligible. So time complexity is $O(M \log^2 N)$.

Time complexity: $O(M \log N)$ or $O(M \log^2 N)$

Note: A solution that runs in $O(M \log^2 N)$ is likely to get AC for **Subtask 3** and **Subtask 4** only if the constant factor of the solution is not that high, i.e if it is written efficiently.

Task 4: Firefighting (`firefighting`)

Authored and prepared by: Bernard Teo Zhi Yi

Preliminaries

We say that a firestation at town i *covers* a town j if the distance between i and j is not more than K kilometres.

Subtask 1

Limits: $1 \leq N \leq 3 \times 10^5$, $1 \leq K \leq 20$, $30 \leq D_i \leq 10^9$

Observe that for any town i, any other town will be at least 30 kilometres away. Since $K \leq 20 < 30$, a fire station from any other town does not cover i. Hence, every town must have its own fire station, regardless of the road network.

Time complexity: $O(N)$

Subtask 2

Limits: $1 \leq N \leq 17$, $1 \leq K \leq 17$, $D_i = 1$

There are 2^N possible subset of towns, and we can test out every one of them. For each subset S of towns, do a breadth-first search from S, stopping once we reach a distance of K (this is a standard multi-source breadth-first search, since all roads have the same length). We then return the minimum-sized S in which the breadth-first search visits all towns.

Time complexity: $O(2^N N)$

Note: Instead of a multi-source breadth-first search, we might do separate single-source depth-first searches for each town i in S. This will incur an additional factor of N, but should still pass this subtask.

Subtask 3

Limits: $1 \leq N \leq 17$, $1 \leq K \leq 10^6$, $1 \leq D_i \leq 10^4$

Similar to subtask 2, we can test every subset of towns. However, we have to now implement Dijkstra's algorithm because the roads may not all have the same lengths.

The limits of this subtask are chosen so that it is unlikely for solutions to run into integer overflow issues when using 32-bit integers.

Time complexity: $O(2^N N \log N)$

Note: Instead of a multi-source Dijkstra's algorithm, we might do separate single-source depth-first searches for each town i in S. The time complexity will then be $O(2^N * N^2)$, but such a solution should still pass this subtask.

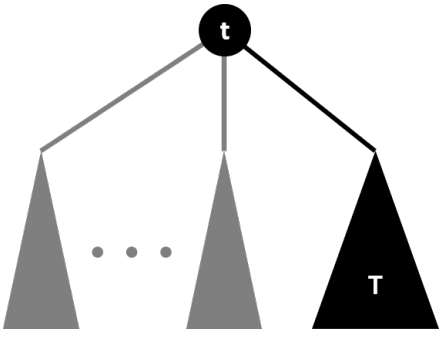
Subtask 4

Limits: $1 \leq N \leq 3 \times 10^5$, $1 \leq K \leq 30$, $20 \leq D_i \leq 10^9$

The limits of this subtask place the following constraint on the graph: if a town does not have its own fire station, then it must have a neighbour (i.e. a town that is reachable using a direct road) with a fire station.

We can root the tree arbitrarily, and do a depth-first search. At every town t, we recursively search each subtree. The return value of searching a subtree T is a pair (x, y), where x is the minimum number of fire stations that need to be selected from that subtree, and to break ties y is the *best* of these three states:

- *positive* — with x fire stations in T, every town in T is covered by some fire station in T; furthermore t is covered by some fire station in T (i.e. the root of T must have a fire station, and the distance between t and the root of T is at most K)
- *zero* — with x fire stations in T, every town in T is covered by some fire station in T; but t is not covered by any fire station in T (i.e. the root of T does not have a fire station, or the distance between t and the root of T is more than K)
- *negative* — with x fire stations in T and another fire station at t, every town in T is covered by some fire station (i.e. the root of T does not have a fire station, and is too far from any fire station in T, but the distance between t and the root of T is at most K)



We define *positive* to be *better* than *zero*, and *zero* to be *better* than *negative*.

Observe that by choosing the minimum x and breaking ties by the *best* y, we have defined a total ordering on the return values, and that if (x_0, y_0) is better than (x_1, y_1), then (x_0, y_0) must be able to lead to a solution (for the whole graph) that has at most as many fire stations as (x_1, y_1).

It is not difficult to see that we can calculate the return value of the subtree rooted at t in $O(\deg(t))$, once we know the return values of all of its subtrees.

Time complexity: $O(N)$

Subtask 5

Limits: $1 \leq N \leq 3 \times 10^3$, $1 \leq K \leq 10^{15}$, $1 \leq D_i \leq 10^9$

Let $C(t)$ be the subset of towns covered by a fire station at town t.

For any town i, let $F_i = \{C(t) \mid t \text{ is a town}, i \in C(t)\}$ be the family of subsets $C(t)$ such that t covers i.

For any town i, observe the following fact: If F_i has a unique maximal element C_0 (i.e. there exists a unique element $C_0 \in F_i$ such that for any $C \in F_i$, $C \subseteq C_0$), then it is optimal to build a fire station at a town t such that $C(t) = C_0$ (if there are multiple such towns, we can choose any one of them).

We make use of the above fact to formulate a greedy algorithm — given any tree T representing the road network, we want to pick a town t such that $T - C(t)$ (i.e. T with all towns in $C(t)$ removed) is "equivalent" to a tree, and furthermore it is optimal to build a fire station at town t.

Root the tree arbitrarily as per subtask 4, and calculate the depth of every vertex (i.e. the distance from each vertex to the chosen root vertex).

Let v be the deepest vertex. Observe that F_v has a unique maximal element C_1, and that there must be a town t_0 that is on the path from v to the root such that $C(t_0) = C_1$. Furthermore, every town in the subtree rooted at t_0 must have been covered by t_0 (since v is chosen to be deepest). Hence, to find t_0, we pick the town of least depth on the path from v to the root such that the distance between v and t_0 is no more than K. We then build a fire station at t_0.

Notice however that $T - C(t_0)$ may be disconnected, so we cannot immediately repeat the algorithm. However, we can solve the connected components in parallel — we pick v to be the deepest vertex across all the components ("deepest" here refers to the distance from the original root; this does not change even though we have multiple components now). However, when traversing the path from v to the root, we allow choosing t_0 from even vertices from the original tree that are already covered previously, as long as the distance between v and t_0 is no more than K. Observe that this still preserves optimality, because even if t_0 was already covered previously, every uncovered descendent of t_0 is covered by t_0, but furthermore t_0 might cover some of its uncovered non-descendants.

In terms of implementation, we first sort all towns from deepest to the root, and mark all of them as unvisited. Then, for each unvisited town i in the sorted list, climb up its chain of ancestors to find the least deep town t that still covers i, build a fire station at t, and then do a depth-first search to mark all towns covered by t as visited. Repeat this process until all towns are visited.

Time complexity: $O(N^2)$

Note: It may be tempting to stop the depth-first search when we arrive at a previously-visited town, in order to reduce the time complexity to $O(N)$. This is incorrect because we will fail to reach any unvisited towns that are beyond a visited town.

Subtask 6

Limits: $1 \leq N \leq 3 \times 10^5$, $1 \leq K \leq 10^{15}$, $1 \leq D_i \leq 10^9$

We can combine the insights from subtasks 4 and 5 to design a depth-first search solution for the final subtask. We need only to extend the information returned from the depth-first search using the fact from subtask 5.

At every town t in the depth-first search, we recursively search each subtree as in subtask 4. The return value of searching a subtree T is a pair (x, y), where x is the minimum number of fire stations that need to be selected from that subtree, and to break ties y is a tagged union containing two possible variants:

- *overflow*(w) where $w \geq 0$ — with x fire stations in T, every town in T is covered by some fire station in T; furthermore any town at most distance w from t (including t itself) is covered by some fire station station in T
- *underflow*(w) where $0 \leq w \leq K$ — with x fire stations in T, every town in T is either covered by some fire station in T or is no more than distance w away from t (so if there is a fire station not in T but at most $(K - w)$ distance away from t, it will be able to cover all towns in T that are not already covered by some fire station in T)

We define any *overflow*(w_1) to be *better* than any *underflow*(w_2) for any w_1 and w_2, that amongst overflows a larger value of w_1 is *better*, and that amongst underflows a smaller value of w_2 is *better*.

Similar to subtask 4, by choosing the minimum x and breaking ties by the *best* y, we have defined a total ordering on the return values, and that if (x_0, y_0) is *better* than (x_1, y_1), then (x_0, y_0) must be able to lead to a solution (for the whole graph) that has at most as many fire stations as (x_1, y_1). The computation of the return value of the subtree rooted at t is similarly linear in the number of roads ending at that town.

Time complexity: $O(N)$