Task 1: Labels (`labels`)

Authored and prepared by: Ho Xu Yang, Damian

Subtask 1

Limits: $N = 2$

There are only 4 possible sequences of length 2: $[1, 1]$, $[1, 2]$, $[2, 1]$ and $[2, 2]$.

$D = [1]$ can be uniquely recovered as $A = [1, 2]$.

$D = [-1]$ can be uniquely recovered as $A = [2, 1]$.

$D = [0]$ can be recovered as either $A = [1, 1]$ or $A = [2, 2]$, hence it is not possible to uniquely recover A.

Time Complexity: $O(1)$

Subtask 2

Limits: $2 \leq N \leq 6$

Run a recursive complete search to try all possible sequences A, and for each sequence check if it is consistent with D. If there are multiple solutions then it is not possible to uniquely recover A.

Time Complexity: $O(N^N)$

Subtask 3

Limits: $2 \leq N \leq 10^3$

To speed up our solution, we note that the only unknown is A_1 since the rest of A can be computed from $A_{i+1} = A_i + D_i$.

Thus, we try every possible value for A_1 (i.e. 1 to N) and compute A, before checking that every element of A lies between 1 and N inclusive.

Time Complexity: $O(N^2)$

Subtask 4

Limits: $-1 \leq D_i \leq 1$

If all $D_i = 1$, we obtain $A = [1, \ldots, N]$.

If all $D_i = -1$, we obtain $A = [N, \ldots, 1]$.

If all $D_i = 0$, then A is a constant sequence and can not be uniquely recovered. This is because we could have a sequence of all 1s, all 2s, etc.

Otherwise, let us denote the largest and smallest elements of A as A_{max} and A_{min} respectively. Thus $1 \leq A_{min} \leq A_{max} \leq N$.

Since we do not have all $D_i = 1$ nor all $D_i = -1$, then $A_{max} - A_{min} < N - 1$.

If $A_{min} > 1$ then we can simply subtract 1 from all elements of A and we will get another valid sequence $A^{'}$ that is also consistent with D.

If $A_{min} = 1$ then $A_{max} < N$, so we can simply add 1 to all elements of A and we will get another valid sequence $A^{'}$ that is also consistent with D.

Hence, it is not possible to uniquely recover A if D is not a constant sequence.

Time Complexity: $O(N)$

Subtask 5

Limits: (No further constraints)

Let us define $D^{'}_k = \sum_{i=1}^{k} D_i = \sum_{i=1}^{k} (A_{i+1} - A_i) = A_{k+1} - A_1$.

From subtask 4, we know that if $A_{max} - A_{min} < N - 1$ then it is not possible to uniquely recover A. Equivalently, we can simply check if $D^{'}_{max} - D^{'}_{min} < N - 1$.

Otherwise, we must have $A_{max} - A_{min} = N - 1$. Thus, $A_{min} = 1$ and $A_{max} = N$. Now, $D^{'}_k = A_{k+1} - A_1 \implies D^{'}_{max} = A_{max} - A_1 \implies A_1 = N - D^{'}_{max}$.

In this manner, A_1 and the rest of A can can be uniquely recovered from D.

Time Complexity: $O(N)$

Task 2: Discharging (`discharging`)

Authored and prepared by: Teow Hua Jun

Subtask 1

Limits: $1 \leq N \leq 3$

The value of N is very small in this subtask, there are only 7 cases of how the N customers can be grouped. Thus, it is possible to brute force through all the possible cases and find the case that will yield the minimum total value.

Subtask 2

Limits:$1 \leq N \leq 1500$, T_i is in non-decreasing order

Firstly, we can change the way we count the total time spent. Instead of considering the amount of time spent for each group, we can consider the amount of time each group costs for itself and the remaining groups.

Note that since the values of T_i is sorted in non-decreasing order, when the customers are divided into contiguous groups the largest T_i value of each group is the last value of each group.

Thus, the cost of the group on all customers where the group is made of contiguous customers from index i to j (1-indexed) is $T_j(N - i + 1)$ as $(N - i + 1)$ is the number of customers that are affected by the charging time of this group.

We will then need to do some dynamic programming. Let $dp(x)$ be the total time cost among all customers by groups that consist of customers from 1 to x, where x is the last customer of the last group.

At each state we have to iterate through all possible lengths of the last group that ends on the x^{th} customer.

Leading to the transition of: $dp(x) = \min(dp(i-1) + T_x(N - i + 1))$ for $i \leq x$

Then our final answer would be $dp(N)$

Time complexity: O(N^2)

Subtask 3

Limits: T_i is in non-decreasing order

The solution to this subtask requires us to speed up the solution of subtask 2. We have to speed up the transition of the dp as for each state we would have to iterate through $O(N)$ states.

Instead we can rearrange the transition of the state to get:

$dp(x) = \min(T_x(N+1) - iT_x + dp(i-1))$ for $i \leq x$

Since $T_x(N+1)$ is independent of i we can pull it out of the expression to get:

$dp(x) = T_x(N+1) + \min(-iT_x + dp(i-1))$ for $i \leq x$

From here, we can use a common dp speed up trick called the convex hull trick to plot different lines ($mx + c$) with the gradients being $-i$ and the constants being $dp(i-1)$, hence at every transition we can substitute T_x as the x-coordinate and find the line that produces the minimum value.

Since, the gradients of the lines and the x-coordinates being queried are monotonic, the lines can be inserted in an armotised time of $O(N)$ and the queries can be resolved in an armotised time of $O(N)$ as well.

Time complexity: O(N)

Subtask 4

Limits: T_i is in non-increasing order

For this subtask we need to use a observation:

Let P_x be the customer with the largest value of T among customers from 1 to x. Note that in the optimal grouping of 1 to x, P_x is in the last group. This observation can be proved through exchange argument and is left as an exercise to the reader.

Using this observation we can see that the first customer needs to be in the last group for the total time to be minimised, hence this means that the optimal grouping is just one big group.

Thus, the answer is just $T_1 N$.

Time complexity: O(N)

Subtask 5

Limits: $0 \leq n \leq 1500$

If we copy the solution from subtask 2 directly, it will fail as the maximum T in each group is no longer the last customer of the group. However, we can remedy this by looping the previous states in reverse and keeping a running maximum to find the maximum value among customer i to x.

$dp(x) = \min(dp(i-1) + \max(T_i, T_{i+1}, ..., T_x) \times (N - i + 1))$ for $i \leq x$

Time complexity: O(N^2)

Subtask 6

Limits: (No further constraints)

Note that it is difficult to apply the convex hull trick to the transition in subtask 5 as we cannot find $\max(T_i, T_{i+1}, ..., T_x)$ very easily. However, we can use the observation in subtask 4 to resolve this issue. Since P_x must be in the last group and has the largest T value, $\max(T_i, T_{i+1}, ..., T_x)$ of the optimal grouping must be equal to the T value of P_x. Thus, we can substitute it into the transition and use the convex hull trick to get an armotised $O(N)$ solution.

$dp(x) = \min(dp(i-1) + T_{P_x}(N - i + 1))$ for $i \leq P_x$

Time complexity: O(N)

Task 3: Progression (`progression`)

Authored and prepared by: Ho Xu Yang, Damian

Introduction

In short, we have an array D of N integers and wish to carry out several operations and queries on it:

Patch operation: Range-add $[L, R]$ with an arithmetic progression of first term S and common difference C

Rewrite operation: Range-set $[L, R]$ with an arithmetic progression of first term S and common difference C

Evaluate query: Range-query $[L, R]$ for the length of the longest subarray that is an arithmetic progression

Subtask 1

Limits: $L = 1, R = N$ for all operations and queries.

Calculate the length of the longest arithmetic progression before any operations are carried out and let this value be X.

Observe that a subarray is an arithmetic progression after a patch operation if and only if it was already an arithmetic progression before the patch operation. To be exact, an arithmetic progression with first term a and common difference d becomes one with first term $a + S$ and common difference $d + C$ after a patch operation.

Thus, the answer to evaluate queries will be X until the first rewrite operation, after which the answer will become N (since the whole array will now always be an arithmetic progression).

Time Complexity: $O(N + Q)$

Subtask 2

Limits: $1 \leq N, Q \leq 10^3$

In this subtask, the limits are small enough to allow us to implement operations directly with for-loops. We then utilise our solution to subtask 1 to answer evaluate queries.

Time Complexity: $O(NQ)$

Subtask 3

Limits: There are no *patch* and *rewrite* operations.

Define a difference array B where $B_i = D_i - D_{i-1}$ for $1 \leq i \leq N$ (With $D_0 = 0$).

Evaluate queries then reduce to finding the length of the longest constant value subarray. Build a segment tree on B and maintain three values at each node — the longest constant value prefix, longest constant value suffix, and longest constant value subarray, storing the length and value of each. We shall allow prefixes and suffixes to potentially span the entire range. However, subarrays shall be strict subarrays — neither prefixes nor suffixes.

At each node (with children *left* and *right*), we calculate the values as such:

1. prefix: either take left.prefix, or left.prefix + right.prefix (if left.prefix.length = left.length and left.prefix.value = right.prefix.value)
2. suffix: either take right.suffix, or left.suffix + right.suffix (if right.suffix.length = right.length and left.suffix.value = right.suffix.value)
3. subarray: take max(left.subarray, right.subarray, left.suffix, right.prefix, left.suffix + right.prefix (if left.suffix.value = right.prefix.value)), taking care to ensure strict subarrays by limiting the length of the prefixes/suffixes

Finally, each evaluate query can be answered by combining the appropriate ranges, and taking max(prefix.length, subarray.length + 1, suffix.length + 1). However, it is possible that our answer might exceed $R - L + 1$ in the case where our suffix spans the entire range and is also a prefix. Hence we limit our answer to $R - L + 1$.

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

Subtask 4

Limits: $L = R$ for all operations.

In this subtask, all operations on D are point-updates. We note that by implementing range-sum query, D_i can be recovered by finding $\sum_{x=1}^{i} B_x$.

A patch operation can be implemented by adding S to B_L and $-S$ to B_{L+1}, taking care to recalculate the values for all affected ranges.

A rewrite operation can be implemented as a patch operation of $S - D_L$.

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

Subtask 5

Limits: There are no *rewrite* operations.

A patch operation can be implemented by

1. Adding S to $[L, L]$
2. Adding C to $[L + 1, R]$ (if $L \neq R$)
3. Adding $-(S + (R - L) \times C)$ to $[R + 1, R + 1]$ (if $R \neq N$)

Range-add updates can be implemented in logarithmic time with lazy propagation.

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

Subtask 6

Limits: (No further constraints)

A rewrite operation can be implemented by

1. Setting $S - D_{L-1}$ to $[L, L]$
2. Setting C to $[L + 1, R]$ (if $L \neq R$)
3. Adjusting B_{R+1} so that D_{R+1} remains unchanged. (if $R \neq N$)

Range-set updates can be similarly implemented in logarithmic time with lazy propagation.

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

Task 4: Arcade (`arcade`)

Authored by: Pang Wen Yuen
Prepared by: Teow Hua Jun

Subtask 1

Limits: $1 \leq N, M, T_i \leq 100$, $1 \leq S \leq 2$

Since the maximum number of hands required is at most 2, it is only necessary to check if it is possible to complete the game with 1 hand. This can be done by processing the instructions in order of time, and at each point checking if the difference in time from the previous instruction is less than or equal to the difference in position from the previous instruction, so that the same hand can reach it.

Time complexity: O($N + M \log M$)

Subtask 2

Limits: $1 \leq N, M, T_i \leq 100$, $1 \leq S \leq 3$

Since the maximum number of hands required is at most 3, we can first check if it is possible to complete the game with 1 hand using the solution outlined in Subtask 1. After which, we can check if the game is completable with 2 hands using the dynamic programming formulation as follows:
$dp(t, a, b)$ = maximum number of buttons that can be pressed such that the hands are on positions a and b at time t.
The transition can be done in $O(1)$ by checking the movements of a and b by 1 from the previous second.

Time complexity: O($(N^2 \max T_i + M \log M)$

Subtask 3

Limits: $1 \leq N, M, T_i \leq 100$, $1 \leq S \leq 4$

Since the maximum number of hands required is at most 4, we can first check the cases of 1 hand and 2 hands using the solution in Subtask 2. The key observation in this subtask involves optimising the DP in Subtask 2 through the fact that during a time tick where an instruction is involved, at least one hand is surely on the button.
As such we can come up with the following DP formulation:
$dp(m, a, b)$ = maximum number of buttons that can be pressed at time T_m such that the three hands are in positions a, b, and A_M. The transition is not $O(1)$, but can be done in amortised

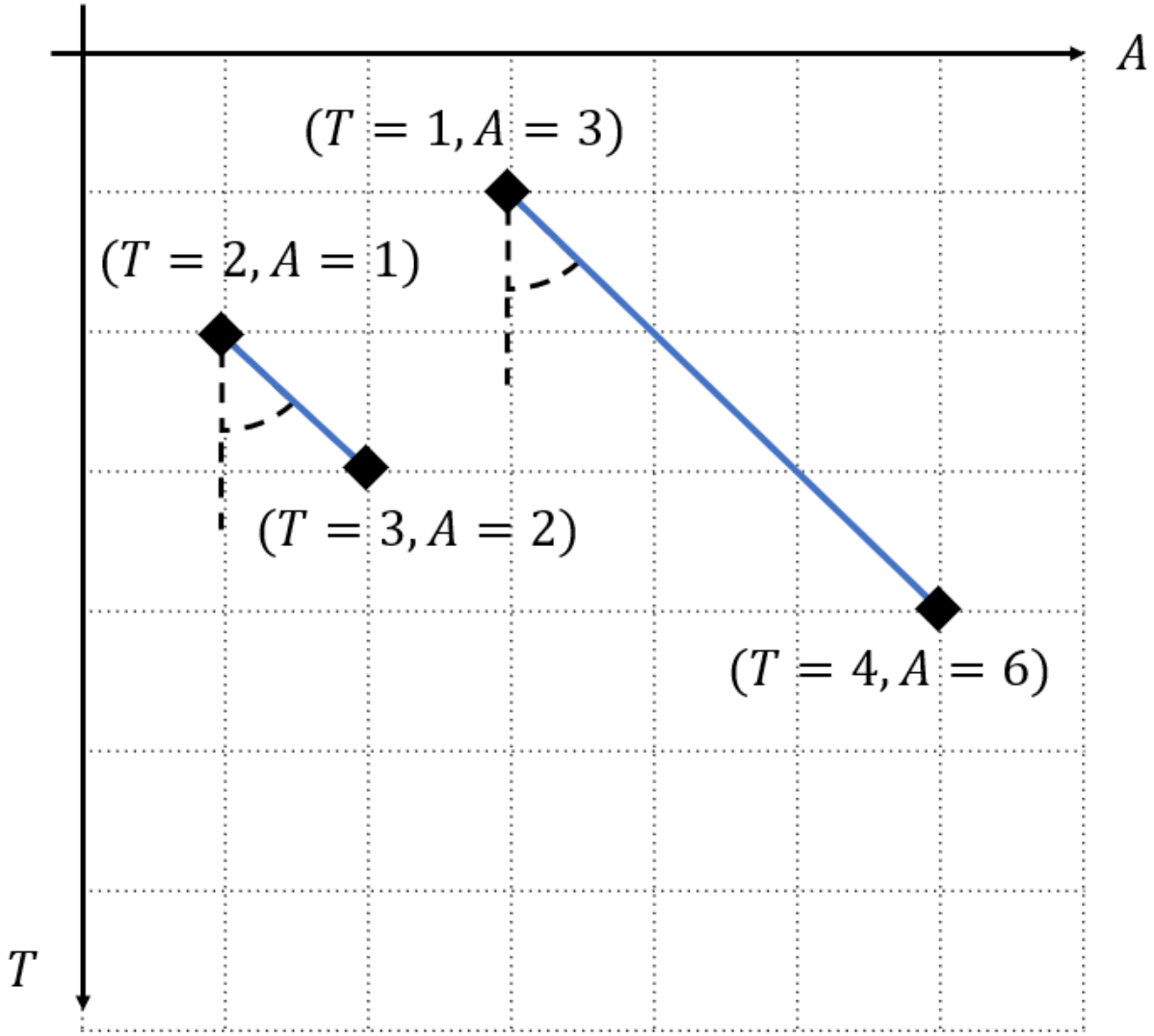
$O(\max T_i^2)$ time, by checking positions of a and b such that the difference between the new and old positions are less than or equal to $T_m - T_{m-1}$.

Time complexity: O($N^2(M + \max T_i) + \max T_i^2 + M \log M$)

Subtask 4

Limits: $0 \leq M \leq 300$

This is the first subtask where there is no limit on the answer, and it requires a completely different approach on the problem. Imagine the instructions as points plotted on a $A_i - T_i$ plane.
The diagram below illustrates the sample case.



Each hand can be represented by a line heading downwards and connecting points, such that no

line exceeds an angle of 45 degrees from the vertical (due to the speed limit of the hands). The problem now becomes: what is the minimum number of lines to connect all points?

This problem can be solved directly via converting it to a Maximum Cardinality Bipartite Matching (MCBM) problem. We can see each point on this grid as having a "parent": which is the directly above it on the same connecting line. For example, (4, 6) will have a parent (1, 3). Each parent can only have one child, and a parent can be connected to a child if $T_{parent} < T_{child}$ and the line connecting them is within 45 degrees of the vertical. Therefore, we can construct a bipartite graph of M nodes on each side, matching parents on the left side to children on the right. The number of nodes that have no matchings is equal to the number of nodes with no parent and equal to the number of lines. Therefore, the minimum number of lines required is equal to $M - MCBM$.

Time complexity: O($M^3 + N$)

Subtask 5

Limits: $0 \leq M \leq 15\,000$

This subtask uses a similar strategy to the one outlined in Subtask 4, but with a slight modification. Instead of viewing the lines as needing to be within 45 degrees of the vertical, we can rotate the grid 45 degrees clockwise. Now all lines must only connect points where one point is to the bottom-left of another. We can view this grid as a DAG, with a having an edge to b if a is to the top-right of b.

The question now becomes: how do we decompose this DAG into the minimum number of chains such that each chain follows the edges on the DAG? By Dilworth's Theorem, the minimum partitioning of a DAG into chains is equal to the size of the maximum antichain, i.e. the maximum size of a subset of nodes that are pairwise unreachable from one another.

In this problem, this is equivalent to the largest set of points where each point is either to the top-left or bottom-right of all other points.This can be seen as finding a line connecting the maximum number of points such that the line can only head bottom-right. The answer can be found with a DP.

First, we sort the points in increasing Y' (the y-coordinate in this rotated grid). Then the DP formulation is as follows:

$dp(i)$ = maximum length of chain that ends at point i

The transition can be done in $O(M)$, as it requires finding the maximum DP value from the set of points that are to the top-left of itself, and extending the chain by 1.

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

Subtask 6

Limits: $0 \leq M \leq 100\,000$

There are several greedy approaches which utilise complex data structures which run in $O(M \log M)$ or $O(M \log^2 M)$ time that do not pass Subtask 7 due to their high constant factors. They will be able to obtain Subtask 6.

Subtask 7

Limits: (No further constraints)

The final solution directly follows from the solution outlined in Subtask 5. Note how the DP in Subtask 5 is actually equivalent to the Longest Increasing Subsequence (LIS) problem. Therefore, if we sort the points by Y', and run LIS on X', finding the LIS trivially will give us the required answer

Time complexity: O($M \log M + N$)

Task 5: Aesthetic (`aesthetic`)

Authored and prepared by: Jeffrey Lee

Introduction

Note that the town is a weighted undirected graph, with its locations as vertices and roads as edges. We will denote the distance between two vertices i and j as $d_{i,j}$.

Subtask 1

Limits: $N, M \leq 100$

There are $\binom{M}{2} = \frac{M(M-1)}{2}$ distinct pairs of roads (i, j), where $i < j$. For each such pair, we can construct the graph (in adjacency-list format) as it would be after j is replicated into i, and then perform Dijkstra's Algorithm in $O(M \log N)$ to determine the resulting $d_{1,N}$. Our final answer will be the highest $d_{1,N}$ found among all pairs.

Time complexity: $O(M^3 \log N)$

Subtask 2

Limits: $N, M \leq 2000$

Notice that if we wanted to extend some road i by picking and replicating another road j into it, choosing a larger W_j would increase $d_{1,N}$ by as much as or more than choosing a smaller W_j. Hence, we need only consider pairs of roads (i, j) in which j has the greatest length out of all roads more aesthetic than i.

We can precompute the largest W_j for all edges i in a total time of $O(M)$, using a sliding maximum from M to 1. Let us call this value the potential extension of edge i, or P_i for short. Subsequently, for each edge i we can again construct the graph in which i is extended by P_i and run Dijkstra's Algorithm. Our final answer is similarly the highest $d_{1,N}$ among all edges i.

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

Subtask 3

Limits: $M = N - 1$

Since the graph is connected and satisfies $M = N - 1$, it is a tree. There thus exists a unique path between locations 1 and N; and increasing the length of any edge i along this path by P_i will also increase $d_{1,N}$ by exactly P_i, while extending any edge i not on the path will not change

$d_{1,N}$. We can run a depth-first search traversal from vertex 1 to determine which edges are on this path. Our answer is then the sum of $d_{1,N}$ and the largest P_i of these edges.

Time complexity: $O(M)$

Subtask 4

Limits: $M = N$

Since the graph is connected and $M = N$, the graph forms a pseudotree and has exactly one cycle. There hence exists either one unique path or exactly two distinct paths between 1 and N, depending on their position with respect to the cycle. Again, these two cases can be distinguished and the edges on the path(s) identified via depth-first search.

If there is only one path from 1 to N, we may reuse the approach of Subtask 3 to find the answer. On the other hand, if there are two paths then extending an edge i will lengthen the path(s) using it by P_i, while leaving the other(s) unaffected. As $d_{1,N}$ will be the length of the shorter of the two paths, we can determine the maximum $d_{1,N}$ by trying each edge and evaluating the resulting path lengths.

Time complexity: $O(M)$

Subtask 5

Limits: $W_i = 1$

Let us call an edge relevant iff there exists at least one shortest path from 1 to N passing through it, and call an edge a bottleneck iff every shortest path from 1 to N passes through it. For this subtask, we realise that $d_{1,N}$ can be increased by 1 if there exists an edge which we can extend to increment all $1 \to N$ shortest paths. In other words, our answer $d'_{1,N}$ is equal to $d_{1,N} + 1$ if there exists a bottleneck which is not edge M, and otherwise $d'_{1,N} = d_{1,N}$.

There are several ways to find bottlenecks. One of them works by first determining $d_{1,i}$ and $d_{N,i}$ for all vertices i in two breadth-first searches. Then, an edge i is relevant iff $d_{1,A_i} + 1 + d_{B_i,N} = d_{1,N}$ (or $d_{1,B_i} + 1 + d_{A_i,N} = d_{1,N}$; without loss of generality, we will assume the former from here onwards). Finally, we have it that an edge i is a bottleneck iff:

- Edge i is relevant, and
- There exists no other relevant edge j such that $d_{1,A_i} = d_{1,A_j}$ and $d_{B_i,N} = d_{B_j,N}$.

This works because i must be the $(d_{1,A_i} + 1)^{th}$ edge in every shortest path to be a bottleneck; and the existence of such a j would require j to displace i from this position in some paths, while the absence of such a j leaves only i to fill this position in all shortest paths.

Time complexity: $O(M)$

Subtask 6

Limits: $0 \leq W_i \leq 10$

We can generalize the ideas introducted in the previous subtask. Let us call a walk from vertex 1 to N of length L or less an L-walk. We say an edge e is L-relevant iff e is used in at least one L-walk, and an edge e is an L-bottleneck iff e is in every L-walk.

For this subtask, we start off with 11 possible values for our answer $d'_{1,N}$: they are the integers $d_{1,N}$ through $d_{1,N} + 10$. Using the abovementioned ideas, we can try these values one by one to determine which of them is the actual answer.

Suppose we had some particular value L, and we were attempting to find and extend an edge such that $d_{1,N}$ would increase to become strictly greater than L. Doing so would be possible iff there exists an edge i such that:

- Edge i is an L-bottleneck, and
- $d_{1,A_i} + W_i + P_i + d_{B_i,N} > L$ and $d_{1,B_i} + W_i + P_i + d_{A_i,N} > L$.

The above holds because extending an edge that satisfies both conditions lengthens all L-walks to above length L; while extending an edge that does not satisfy either condition will respectively leave behind the L-walks not passing through the edge, or fail to extend some of the L-walks through the edge to above length L.

The values of $d_{1,i}$ and $d_{N,i}$ for all vertices i can be determined in two runs of Dijkstra's Algorithm, leaving us with the task of finding L-bottlenecks. First, we can identify all L-relevant edges: these are the edges i which have $d_{1,A_i} + W_i + d_{B_i,N} \leq L$ or $d_{1,B_i} + W_i + d_{A_i,N} \leq L$. We can then construct the subgraph H whose edges are the L-relevant edges of the original graph. Finally, an L-relevant edge i is an L-bottleneck iff:

- In H, deleting edge i disconnects vertices 1 and N.

This condition implies an L-bottleneck because if 1 and N are disconnected without an edge i, it is shown that an L-walk cannot be constructed without i and hence there exist no L-walks which do not pass through i. It is intuitive that all L-bottlenecks satisfy the condition - one proof is as follows:

> Assume for contradiction that an L-bottleneck i exists where deleting i from H does not disconnect 1 and N. Then, there exists a path $a_1 \xrightarrow{g_1} a_2 \xrightarrow{g_2} \ldots \xrightarrow{g_{k-1}} a_k$ from vertex 1 to N in $H - i$, where $a_1 = 1$ and $a_k = N$ (note that this path need not be an L-walk). It is simple to see that there has to exist an L-walk in which g_1 precedes i, and another in which g_{k-1} succeeds i. However, there must then exist two edges g_{j-1} and g_j such that g_{j-1} precedes i but g_j succeeds i: that is, there exist L-walks of the following forms, where each P denotes some path:

- $1 \xrightarrow{P_0} a_{j-1} \xrightarrow{g_{j-1}} a_j \xrightarrow{P_1} A_i \xrightarrow{i} B_i \xrightarrow{P_2} N$
- $1 \xrightarrow{P_3} A_i \xrightarrow{i} B_i \xrightarrow{P_4} a_j \xrightarrow{g_j} a_{j+1} \xrightarrow{P_5} N$

However, then one of $1 \xrightarrow{P_3} A_i \xrightarrow{P_1} a_j \xrightarrow{P_4} B_i \xrightarrow{P_2} N$ or $1 \xrightarrow{P_0} a_{j-1} \xrightarrow{g_{j-1}} a_j \xrightarrow{g_j} a_{j+1} \xrightarrow{P_5} N$ must be short enough to be an L-walk, and thus i is not in every L-walk which is a contradiction.

Our final task is now to determine which edges of H disconnect 1 and N when deleted. This can be done in $O(M)$ by applying Tarjan's Bridge-Finding Algorithm, compressing biconnected components, and then running a depth-first search on the resulting graph.

Time complexity: $O(MW_{max} + M \log M)$

Subtask 7

Limits: (No further constraints)

There are now $10^9 + 1$ possible values of $d'_{1,N}$, up from 11 in the previous subtask. Notice that our trial is monotonic in L, in that it succeeds for all $L < d'_{1,N}$ and fails for all $L \geq d'_{1,N}$. We can therefore determine our answer via binary-search, reducing the number of trials needed from $O(W_{max})$ to $O(\log W_{max})$.

Time complexity: $O(M \log W_{max} + M \log M)$

Alternative solution

There is an alternative solution for this task. We use the same terminology and insight of the previous solution up till subtask 5. We similarly construct the subgraph H containing all relevant edges of G.

Similar to subtask 5, we observe that extending a bottleneck edge is the only possible way to increase our answer $d'_{1,N}$, and by the proof in subtask 6, the set of bottlenecks is precisely the set of edges of H that disconnects vertices 1 and N. Furthermore, if some bottleneck i of H is extended, the shortest distance from 1 to N, say S_i, would become the minimum of:

- $d_{1,N} + P_i$, and
- The length of the shortest path from 1 to N that does not contain i.

The first case arises when edge i is still short enough that it remains on some shortest path, and the second case arises when edge i was extended sufficiently such that another path that does not contain i now becomes the shortest.

Since we have the freedom to pick the edge to extend, our answer $d'_{1,N}$ would be the maximum S_i over all bottlenecks i of H.

As with subtask 2, $d_{1,N} + P_i$ may be determined for every edge i with simple precomputation. It hence remains to determine for every bottleneck i of H the length of the shortest path from 1 to N that does not contain i.

Define the relation $\prec$ on bottlenecks of H where $i \prec j$ means that all paths in H from vertex 1 to edge j contain edge i and all paths in H from edge i to vertex N contain edge j. Observe that $\prec$ is a total ordering, and so we can enumerate the bottlenecks, say $E_1, E_2, \ldots, E_B$, where B is the total number of bottlenecks.

For each bottleneck i, let C_i be a set of candidate lengths of shortest path from 1 to N that do not contain i. We use the following algorithm to populate the sets C_i:

> Initially, $C_i = \{\}$ for all bottlenecks i.
>
> Run Dijkstra's algorithm from vertex 1 to N, but with an additional value Q_v for every vertex v, representing the number of bottlenecks that the shortest path from 1 to v found by the Dijkstra's algorithm contains (note: where there are multiple shortest paths to v the value of Q_v may depend on the tie-breaking mechanism of the Dijkstra's algorithm implementation, but it does not affect the correctness of this solution). This may be done by maintaining the value of Q_v on non-bottleneck edges, and incrementing it on bottleneck edges. However, we additionally do the following when arriving at a visited vertex:
>
> Let e be the incoming edge being processed, v be the vertex that we arrive at (which is already visited previously), u be the other endpoint of the incoming edge ($u \neq v$), and y be the additional value carried in e (which is equal to $Q_u + 1$ if e is a bottleneck, and Q_u otherwise).
>
> - If $y < Q_v$, we insert $d_{1,u} + W_e + d_{v,N}$ to the set C_{E_j} for all $y < j \leq Q_v$.
> - Otherwise we do not do anything special.
>
> Note that in this instance of Dijkstra's algorithm, we do not store whether edges have been visited, so each edge will be processed twice — one in each direction.

We claim that the above algorithm will result in valid C_i for all bottlenecks i. By saying that C_i is valid, we mean that the length of the shortest path from 1 to N that does not contain i is indeed in C_i, and furthermore it must be the minimum value present in C_i.

Before proving the claim, we need to make the following observation (Lemma 1): for any vertex u, every $1 \to u$ shortest path contains bottlenecks $E_1, \ldots, E_k$ and does not contain bottlenecks $E_{k+1}, \ldots E_B$, for some k possibly dependent on the path taken to u. This can be seen by contradiction:

Suppose there is some $k_1 < k_2$ such that some $1 \to u$ shortest path P contains E_{k_2} but not E_{k_1}. Then since E_{k_2} is a bottleneck, all $1 \to N$ shortest paths must contain E_{k_2}. In particular there must be at least one such path Q. Since P is a shortest path, the path from 1 to E_{k_2} that follows Q (which contains E_{k_1}) is no shorter than the path from 1 to E_{k_2} that follows P (which does not contain E_{k_1}). By appending the portion of Q from E_{k_2} to N, we have obtained a path from 1 to N that does not contain E_{k_1} but is at least as short as Q, thus contradicting the assumption that E_{k_1} is a bottleneck.

To prove that C_i does not contain any value less than the length of the $1 \to N$ shortest path that does not contain $i =: E_j$:

Suppose for contradiction that this is not the case. Then the Dijkstra's algorithm above must have visited some vertex v such that all $1 \to N$ shortest paths that contain v also contain i, and furthermore at v we must have an adjacent vertex u such that $y < j \leq Q_v$ (for any incoming value y as defined in the Dijkstra's algorithm above). It is easy to see that either v comes after i in every such path, or v comes before i in every such path (otherwise i cannot be a bottleneck). In the case where v comes after i, then every such path would have passed through i, and hence by Lemma 1 we have $y \geq j$. In the case where v comes before i, then by nature of Dijkstra's algorithm we would set Q_v to some value strictly less than j. In any case the condition $y < j \leq Q_v$ cannot be fulfilled, hence arriving at a contradiction.

We now need to show the other part of the claim — that for every bottleneck i, the length of the shortest path from 1 to N that does not contain i is indeed in C_i. Since we have already proven the first part the claim, it suffices to show that for every bottleneck i, the length of the shortest path from 1 to N that does not contain i is at least as large as some element in C_i.

Consider any shortest path P from vertices 1 to N that does not contain edge $i =: E_j$. Since $Q_1 = 0$ and $Q_N = B$, there must exist an edge i_0 in P such that $Q_{A_{i_0}} < j \leq Q_{B_{i_0}}$, where A_{i_0} and B_{i_0} are the two endpoints of i_0, and A_{i_0} precedes B_{i_0} in P. Furthermore, since P does not contain i, i_0 cannot be i itself.

Observe that i_0 cannot be any other bottleneck either: If i_0 is a bottleneck such that $i \prec i_0$, then by Lemma 1 this is a contradiction, since the shortest path from 1 to A_{i_0} found by the Dijkstra's algorithm contains $Q_{A_{i_0}} < j$ bottleneck edges. On the other hand, if $i_0 \prec i$, observe that $d_{1,A_i} + W_i + d_{B_i,B_{i_0}} \leq d_{1,B_{i_0}}$, where A_i and B_i are the two endpoints of i and A_i precedes B_i in the Dijkstra's algorithm, because $Q_{B_{i_0}} \geq j$ implies by Lemma 1 that the shortest $1 \to B_{i_0}$ path passes through i. By triangle inequality $d_{1,B_{i_0}} \leq d_{1,A_i} + d_{A_i,B_{i_0}}$, and since $W_i \geq 0$, we have $d_{B_i,B_{i_0}} \leq W_i + d_{A_i,B_{i_0}}$. This means that $d_{B_{i_0},B_i} + d_{B_i,N} \leq d_{B_{i_0},A_i} + W_i + d_{B_i,N}$, which means that there is a $B_{i,0} \to N$ path that does not contain i, and this path is at least as short as the shortest $B_{i,0} \to N$ path containing i. This contradicts the fact that i is a bottleneck.

Hence, during the execution of the Dijkstra's algorithm, there will be an iteration where $v = B_{i_0}$ and $u = A_{i_0}$ (where v and u are as defined in the Dijkstra's algorithm above), and $y = Q_u <$

$j \leq Q_v$, and hence $k \coloneqq d_{1,A_{i_0}} + W_{i_0} + d_{B_{i_0},N}$ is in C_i. Finally, observe that k is the length of the shortest path from 1 to N that contains i_0. Thus, since P contains i_0, the length of P must be at least k.

This proves the correctness of the solution.

Maintaining the sets C_i naively takes $O(MB)$ time and space complexity. However, with an appropriate range-minimum data structure, we can compute the array C_i in $O(B + M \log M) = O(M \log M)$. For example, after sorting the ranges by their start index, we could use a priority queue with smallest distance at the top to obtain the array C_i.

Time complexity: $O(M \log M)$