香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

T252 – Sage's Shopping Spree

Isaac Chan {snowysecret}

2025-05-10

Background

Problem idea by `snowysecret`

Preparation by `ethening`, `yfng`, `QwertyPi`, `happypotato` (Thanks!) and `snowysecret`

Presented by `snowysecret`

Inspiration: This problem came up in my mind when I was actually going shopping during term break

(2nd easiest problem in the problem set!)

The Problem

You are given a tree with N nodes. Each node i holds a single item of unique size A[i]. Each edge (U[j], V[j]) has a "capacity" W[j].

Collect all N items using one or more **trips**. A trip is defined as follows:

- Pick any start node; basket starts empty.
- Walk along edges as you like (re-visits allowed), but whenever you do so, **every item** already in the basket must have **size ≤ that edge's capacity**.
- At each visited node you may pick up its item (if not taken yet) or skip it.
- End the trip at any node and empty the basket (the collected items are now "home").

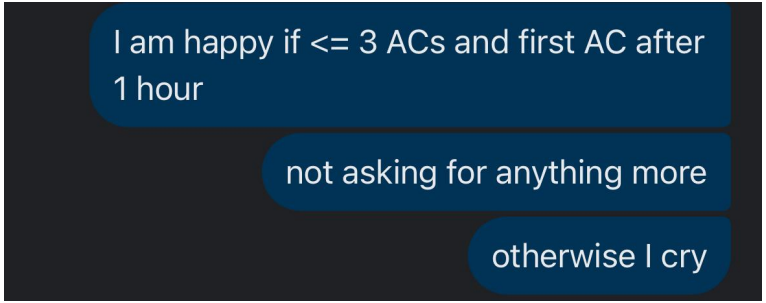
Find the minimum trips needed to collect all items. Output –1 if more than K trips are needed.

Subtasks

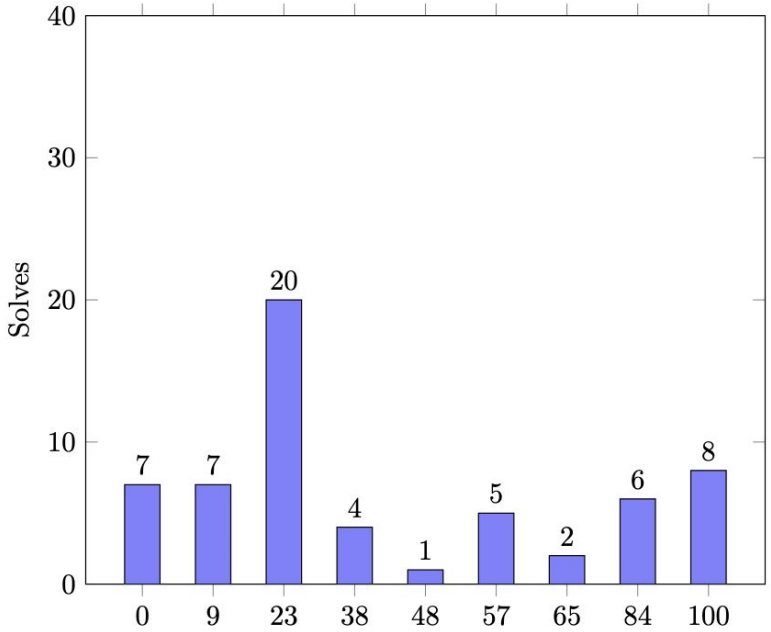
Subtask	Score	Constraints
1	9	$K = 1$, $N = 3$, star
2	14	$K = 1$, $N \leq 200$, star
3	15	$K = 1$, $N \leq 200$
4	19	$K = 1$
5	10	$K = 2$, $N \leq 200$, at least two edges with capacity 1
6	9	$N \leq 200$
7	8	$N \leq 2000$
8	16	No additional constraints

Prediction

- # of 100: 3
- # of 84+: 6
- # of 57+: 10



Statistics



First solved by **dbsculver0412** at **0:34**

Before we begin

- In this solution session I might omit some of the proofs used to validate the solution.
- You are **not expected** to come up with a proof during the contest!
- But they are included in the slides and you can feel free to try them out if you like.

K = 1: Gathering Observations

Subtask	Score	Constraints
1	9	K = 1, N = 3, star
2	14	K = 1, N ≤ 200, star
3	15	K = 1, N ≤ 200
4	19	K = 1
5	10	K = 2, N ≤ 200, at least two edges with capacity 1
6	9	N ≤ 200
7	8	N ≤ 2000
8	16	No additional constraints

Subtask 1 (9%): K = 1, N = 3, The tree is a star

Since here K = 1 and N = 3, one way is try to brute force all 3! = 6 orders of taking the items in a single trip.

- First go to node 0, then node 1, then node 2.
- First go to node 0, then node 2, then node 1.
- ...
- First go to node 2, then node 1, then node 0.

Very careful case handling will allow you to pass this subtask.

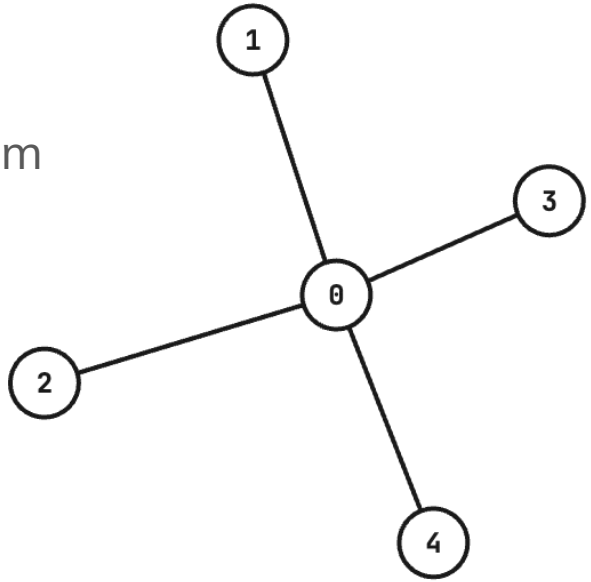
Time complexity: O(N!)

Expected score: 9

Subtask 2 (14%): K = 1, N ≤ 200, The tree is a star

A star has many leaves. Hence we will have to get items from leaves quite often!

Let's analyse what happens after we pick up an item at leaf node i (≥ 1).

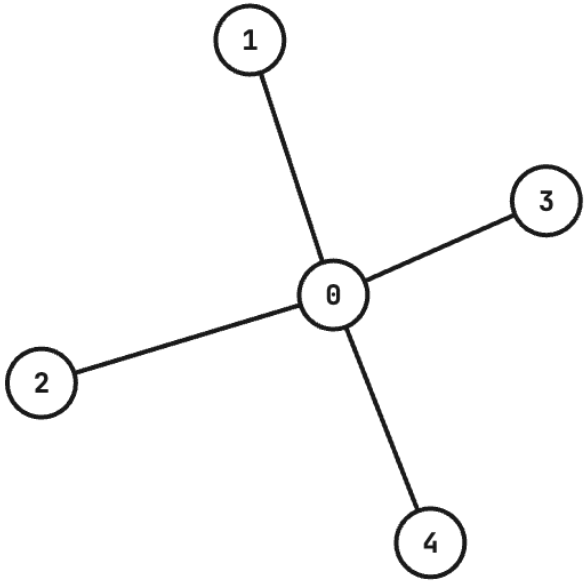


Subtask 2 (14%): K = 1, N ≤ 200, The tree is a star

Let's analyse how we can pick up an item from node i ≥ 1.

There are two cases:

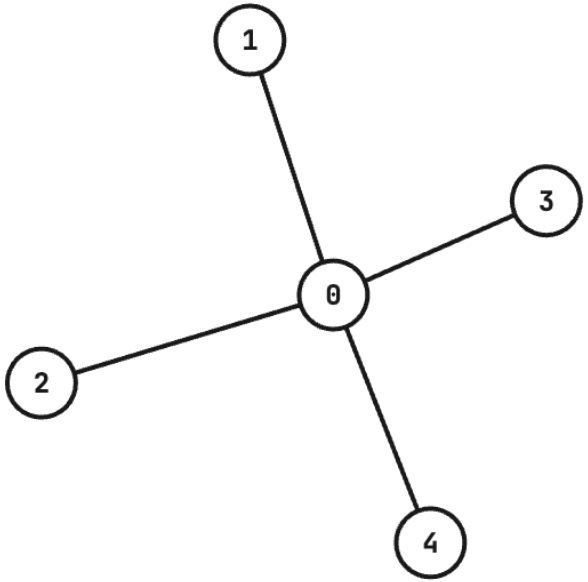
1. Pick up an item from a leaf node.
   Then take it back to node 0.
2. Pick up an item from leaf node.
   Then don't take it back to node 0.



Subtask 2 (14%): K = 1, N ≤ 200, The tree is a star

1. Pick up an item from a leaf node.
   Then take it back to node 0.

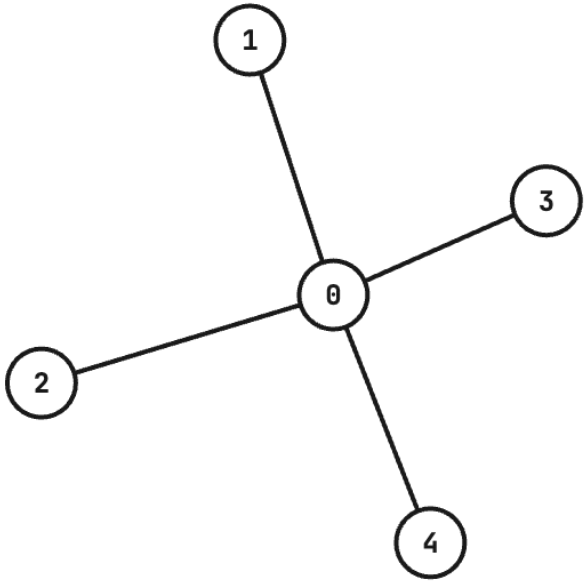
This imposes the constraint **A[i] ≤ W[i–1]**, where
A[i] is the size of node i's item, and
W[i-1] is the capacity of edge (0)–(i).



Subtask 2 (14%): K = 1, N ≤ 200, The tree is a star

2. Pick up an item from leaf node.
   Then don't take it back to node 0.

But the current node is a leaf node, so we have nowhere to go. The only possibility is that we are **ending the trip** at node i.



Observation 1: Recall that each of item sizes 1, 2, ..., N appear exactly once in the nodes.
Hence if we end the trip at node i and successfully collected all N items, we must have brought **either** item with size N – 1 or item with size N in!

Subtask 2 (14%): K = 1, N ≤ 200, The tree is a star

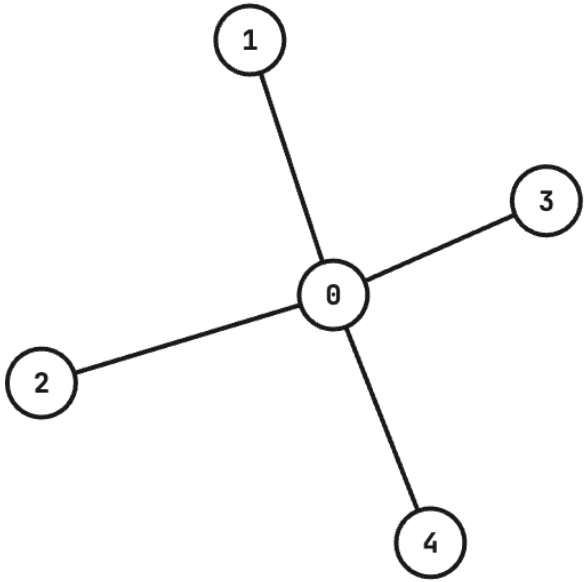
2. Pick up an item from leaf node.
   Then don't take it back to node 0.

Case 1) A[i] = N

- We brought item with size N – 1 in.
- Need W[i-1] ≥ N – 1.

Case 2) A[i] ≠ N

- We brought item with size N in.
- Need W[i-1] = N.

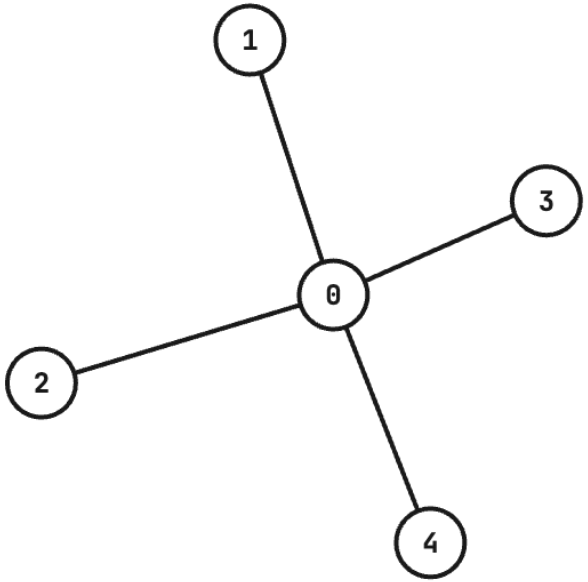


Subtask 2 (14%): K = 1, N ≤ 200, The tree is a star

Let's analyse how we can pick up an item from node i ≥ 1.

There are two cases:

1. Pick up an item from a leaf node.
   Then take it back to node 0.
   ⇒ Need **A[i] ≤ W[i–1]**.
2. Pick up an item from leaf node.
   Then don't take it back to node 0.
   ⇒ Need **A[i] = N, W[i-1] ≥ N - 1**
   ⇒ Or **A[i] < N, W[i-1] = N**



Subtask 2 (14%): K = 1, N ≤ 200, The tree is a star

Rewrite the cases as follows:

Case 1) For the node i with A[i] = N (if i ≥ 1), set it as the ending node of this trip. This requires W[i-1] ≥ N - 1, and A[j] ≤ W[j-1] for all other leaves j.

Case 2) Do not end the trip at the node i with A[i] = N, requiring W[i-1] = N. This requires us to find another node k where W[k-1] = N, and A[j] ≤ W[j-1] for all other leaves j.

Subtask 2 (14%): K = 1, N ≤ 200, The tree is a star

Rewrite the cases as follows:

Case 1) For the node i with A[i] = N (if i ≥ 1), set it as the ending node of this trip. This requires W[i-1] ≥ N - 1, and A[j] ≤ W[j-1] for all other leaves j.

Case 2) Do not end the trip at the node i with A[i] = N, requiring W[i-1] = N. This requires us to find another node k where W[k-1] = N, and A[j] ≤ W[j-1] for all other leaves j. **⇒ This is strictly more constrained than Case 1)!**

Subtask 2 (14%): K = 1, N ≤ 200, The tree is a star

We prefer case 1) over case 2).

Conclusion: For the node i with A[i] = N (if i ≥ 1), set it as the ending node of this trip. This requires W[i-1] ≥ N - 1, and A[j] ≤ W[j-1] for all other leaves j.

Simply checking these conditions is sufficient. In particular, we need not consider the weight of the center node.

Time complexity: O(N)

Expected score: 23

Subtask 3 (15%): K = 1, N ≤ 200

In Subtask 2, we observed that it is the best idea to end the trip at node i, where A[i] = N.

Does this give you any insight on the ordering of nodes within a trip, for a general tree (not just a star)?

Subtask 3 (15%): K = 1, N ≤ 200

In Subtask 2, we observed that it is the best idea to end the trip at node i, where A[i] = N.

Does this give you any insight on the ordering of nodes within a trip, for a general tree (not just a star)?

Let's consider the nodes in **ascending size order (1, 2, ..., N)**! (Intuition: this is the "simplest possible ordering" we can find that ends at size N.)

Subtask 3 (15%): K = 1, N ≤ 200

Observation 2: In a valid trip, it is optimal to visit nodes in ascending size order. In the case where K = 1, that is: visit the node with size 1, then the node with size 2, and so on.

Subtask 3 (15%): K = 1, N ≤ 200

Observation 2: In a valid trip, it is optimal to visit nodes in ascending size order. In the case where K = 1, that is: visit the node with size 1, then the node with size 2, and so on.

Proof: Consider any two nodes in the trip, u and v, where A[u] < A[v].
Regardless of whether we visit u first, then visit v, **OR** visit v first, then visit u, in between visiting the two nodes u and v we will definitely have passed through the entire path between u and v.
Let w be the minimum capacity of an edge on the path between u and v.
Since A[u] < A[v], it is possible that A[u] ≤ w < A[v], but impossible that A[v] ≤ w < A[u]. Hence, u should go **before** v in the trip.

Subtask 3 (15%): K = 1, N ≤ 200

We simply need to check whether the "ascending size" trip works.

Define a function `reachable(x, y)` as follows:

- `reachable(x, y)`: whether node y can go **immediately after** node x in a trip.
- This requires the minimum capacity edge along the path from x to y to have a capacity w ≥ A[x].

reachable(x, y) can be computed using a simple DFS in O(N) time.

Subtask 3 (15%): K = 1, N ≤ 200

Then, simply check whether `reachable(pos[1], pos[2])`, `reachable(pos[2], pos[3])`, ..., `reachable(pos[N-1], pos[N])` are true, where pos[i] denotes the node ID which has an item of size i.

Time complexity: $O(N^2)$ // We allow slower $O(N^3)$ implementations too

Expected score: 38

Subtask 4 (19%): K = 1

We have O(N) calls to `reachable`, which is fine – but each call takes O(N) time, which is too slow.

What tools have we learnt that could speed up `reachable`?
Hint: in `reachable(x, y)` we are checking, for some pair of nodes x and y, whether the **path minimum** on the path from x to y has capacity ≥ A[x]

Subtask 4 (19%): K = 1

We have O(N) calls to `reachable`, which is fine – but each call takes O(N) time, which is too slow.

What tools have we learnt that could speed up `reachable`?
Hint: in `reachable(x, y)` we are checking, for some pair of nodes x and y, whether the **path minimum** on the path from x to y has capacity ≥ A[x]

There are various methods to solve this, such as **Binary Lifting / Kruskal Reconstruction Tree**

Subtask 4 (19%): K = 1

- Binary Lifting approach: The time complexity of reachable is O(log N).
- KRT approach: If you use O(1) methods to compute the LCA of x and y within the KRT, you can obtain the minimum capacity directly in O(1).

Both approaches can pass!

Time complexity: O(N log N)

Expected score: 57

Small N: Flow Modelling

Subtask	Score	Constraints
1	9	K = 1, N = 3, star
2	14	K = 1, N ≤ 200, star
3	15	K = 1, N ≤ 200
4	19	K = 1
5	10	K = 2, N ≤ 200, at least two edges with capacity 1
6	9	N ≤ 200
7	8	N ≤ 2000
8	16	No additional constraints

Subtask 5 & 6 (Total 19%): N ≤ 200

Now that we defined the function `reachable`, let's try building an auxiliary graph, where there is an edge from x to y iff `reachable(x, y)` and x < y.

In the original graph, the problem is to find the minimum number of trips such that all N items would be taken.

What is the equivalent form in this auxiliary graph?

Subtask 5 & 6 (Total 19%): N ≤ 200

Now that we defined the function `reachable`, let's try building an auxiliary graph, where there is an edge from x to y iff `reachable(x, y)` and x < y.

In the original graph, the problem is to find the minimum number of trips such that all N items would be taken.

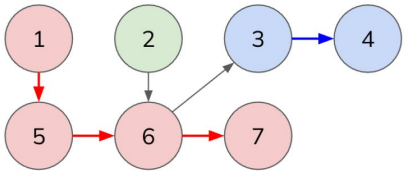
What is the equivalent form in this auxiliary graph? **It is to find the minimum number of paths such that each node lies in exactly one path.** (Does that sound familiar to you?)

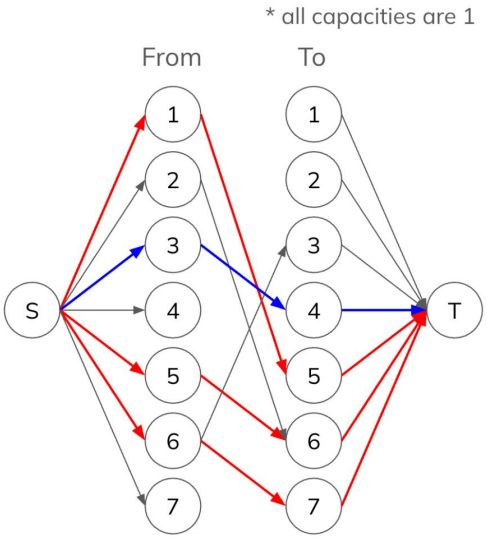
Subtask 5 & 6 (Total 19%): N ≤ 200

... or maybe you have attended the Flow and Graph Matching (II) lecture?

Modeling Problems with Matching

- Create a "from" vertex (out-edge) and "to" vertex (in-edge) for each vertex.
- For each edge u → v, build an edge `u.from` → `v.to`.
- Answer = `|V|` - maximum matching.





Subtask 5 & 6 (Total 19%): N ≤ 200

Indeed, the answer to this problem is just the minimum path cover of the "reachable" auxiliary graph.

Run any reasonable bipartite matching algorithm.

Time complexity: $O(N^3)$

Expected score: 19 (Cumulative: 57, same as just doing K = 1)

Towards Full: Forward Greedy

Subtask	Score	Constraints
1	9	K = 1, N = 3, star
2	14	K = 1, N ≤ 200, star
3	15	K = 1, N ≤ 200
4	19	K = 1
5	10	K = 2, N ≤ 200, at least two edges with capacity 1
6	9	N ≤ 200
7	8	N ≤ 2000
8	16	No additional constraints

Subtask 5 (10%): K = 2, N ≤ 200, two or more W[j] = 1 edges

Here we can use at most two trips, but there are two or more edges with capacity = 1.

Generalisation of Observation 2: In any valid trip, it is optimal to visit nodes in ascending size order.

The edges with capacity 1 can only be used for the item with weight 1. After we bring the item with weight 1 to another node, the edges with capacity 1 cannot be used anymore.

Subtask 5 (10%): K = 2, N ≤ 200, two or more W[j] = 1 edges

Observation 3: In this subtask, after we "send" the item with size 1, the graph is effectively **split into ≥ 3 connected components**. Exactly one of which contains the node with size 1.

We now have 3 connected components, but we are only allowed 2 trips in total. How could this possibly work?

Subtask 5 (10%): K = 2, N ≤ 200, two or more W[j] = 1 edges

Observation 3: In this subtask, after we "send" the item with size 1, the graph is effectively **split into ≥ 3 connected components**. Exactly one of which contains the node with size 1.

We now have 3 connected components, but we are only allowed 2 trips in total. How could this possibly work?

- Exactly one connected component contains **only** the node with size 1
- Then each of the two remaining components corresponds to one trip each
- Handle them separately using K = 1 solution

Time complexity: O(N)

Expected score: 67 (including K = 1)

Subtask 6 & 7 (17%): N ≤ 2000

Can we generalise the "splitting" idea from Subtask 5?

Subtask 6 & 7 (17%): N ≤ 2000

Can we generalise the "splitting" idea from Subtask 5?

Let's try processing the nodes in ascending size order, just like how we did before.

Observation 3: In Subtask 5, after we "send" the item with size 1, the graph is effectively **split into ≥ 3 connected components**. Exactly one of which contains the node with size 1.

Generalisation of Observation 3: In general, after we "send" the item with size X, we effectively **'cut' all the edges with capacity X**.

Subtask 6 & 7 (17%): N ≤ 2000

Generalisation of Observation 3: In general, after we "send" the item with size X, we effectively **'cut' all the edges with capacity X**.

Consequence: Let's suppose we actually cut the edges according to Generalisation of Observation 3. Then, we can only continue the trip by visiting nodes in the same component.

Let's try iterating over the nodes in ascending size order, maintaining the set of existing trips (endpoints) in the process. When it is node u's turn, we try appending it to an existing trip.

Subtask 6 & 7 (17%): N ≤ 2000

Observation 4: There is at most one existing trip to append to.

Why it is true: Let's suppose there are two or more existing trips we can append node u to. Let the final node of these two trips be X and Y respectively, where A[X] < A[Y] < A[u].
By Consequence of Observation 3, X and u are in the same component, and Y and u are in the same component.
⇒ X and Y are in the same component.
⇒ We could have chosen node Y to append to node X instead.
⇒ Hence, there is at most one existing trip to append to.

Subtask 6 & 7 (17%): N ≤ 2000

This motivates the following approach:

```
def Append-While-We-Can(u):
  let the current path tails be t_1, t_2, …, t_k, where all t_i < u
  if (there exists some i such that reachable(t_i, u)):
    append u to the end of path i
  else:
    create new path starting at u
```

Run Append-While-We-Can(u) for nodes u in ascending size order.

Since there is at most one existing trip to append to, the value i on line 3 is uniquely determined.

Subtask 6 & 7 (17%): N ≤ 2000

```
def Append-While-We-Can(u):
  let the current path tails be t_1, t_2, ..., t_k, where all t_i < u
  if (there exists some i such that reachable(t_i, u)):
    append u to the end of path i
  else:
    Create new path starting at u
```

Time complexity: O(NK), if `reachable` takes O(1) time...

or O(NK log N), if `reachable` takes O(log N) time.

Expected score: 84

Full Solution 1: Optimized Forward Greedy

Subtask	Score	Constraints
1	9	K = 1, N = 3, star
2	14	K = 1, N ≤ 200, star
3	15	K = 1, N ≤ 200
4	19	K = 1
5	10	K = 2, N ≤ 200, at least two edges with capacity 1
6	9	N ≤ 200
7	8	N ≤ 2000
8	16	No additional constraints

Subtask 8 (16%): No additional constraints

Our previous solution is too slow. Specifically, for it to even have any hope of passing, we will need to optimize finding the existing trip (if any) that we can append node u to, which is similar to handling tree splits (decompositions).

To handle tree decompositions, we can maintain sets of current components, then perform "small-to-large decomposition" (inverse of small-to-large merging) while maintaining the ending nodes of each trip.

We will not go over the implementation details since there is a better way :)

Full Solution 2: Backward Greedy

Subtask	Score	Constraints
1	9	K = 1, N = 3, star
2	14	K = 1, N ≤ 200, star
3	15	K = 1, N ≤ 200
4	19	K = 1
5	10	K = 2, N ≤ 200, at least two edges with capacity 1
6	9	N ≤ 200
7	8	N ≤ 2000
8	16	No additional constraints

Subtask 8 (16%): No additional constraints

Intuition: 'Split' is complicated, but 'merge' is easier.
Instead of handling 'split', do it backwards and handle 'merge' instead.

'Split': ← Evil implementation
Consider nodes with sizes 1, 2, ..., N in order.
When appending node X to a trip ending at node Y, **disconnect** all edges with capacity ≤ A[Y].
Check for the (unique) trip that node X can append to, if does not exist then add answer by 1.

'Merge': ← Happy implementation
Consider nodes with sizes N, N-1, ..., 1 in order.
When considering node X, **connect** all edges with capacity ≥ A[X].
Check if there is an existing trip in node X's component. If does not exist then add answer by 1.

Subtask 8 (16%): No additional constraints

Backwards Greedy:
Consider nodes with sizes N, N-1, ..., 1 in order.
When considering node X, **connect** all edges with capacity ≥ A[X].
Check if there is an existing trip in node X's component. If does not exist then add answer by 1.

We can simply use a DSU to maintain the connected components, and whether or not there is an existing trip in each connected component.

The model solution is just 35 lines long!

Time complexity: O(N)

Expected score: 100

Conclusion

Conclusion

- Knowledge from different parts of the IOI syllabus would help – particularly, don't neglect flows, it may give you some advantage in certain subtasks / for coming up with ideas

- Thinking from another perspective (i.e. backwards) can give you possibly unexpected insights! Don't be afraid to change your perspective, although it might mean losing some previous "progress"