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

T251 - Moving Marvellous Marbles

Bosco Wang {happypotato}, Daniel Hsieh {QwertyPi}

2025-05-10

Background

Problem idea by `QwertyPi`

Preparation by `ethening`, `happypotato`, `QwertyPi` (Thanks!)

Presented by `happypotato`, `QwertyPi`

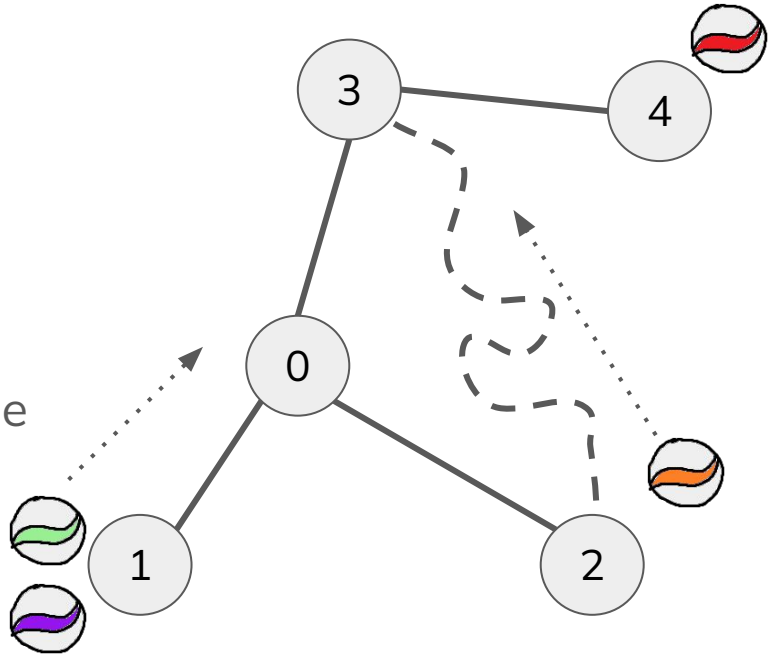
The Problem

Given a tree of N nodes

Initial: A[i] marbles at node i
Target: B[i] marbles at node i

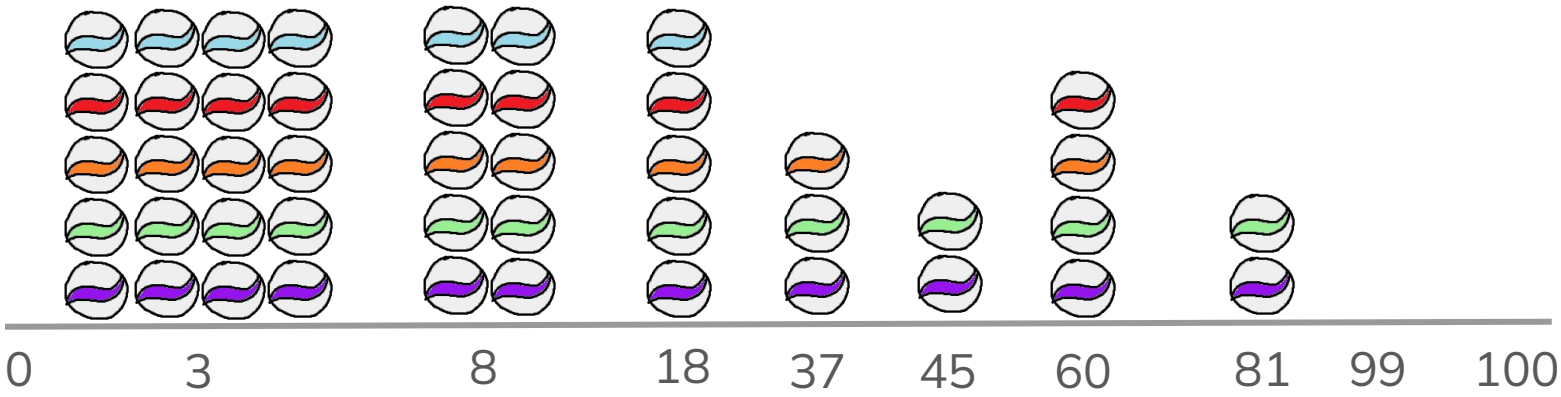
Each second, move a marble through an edge
You can add at most **K (= 0 or 1)** edge

Minimise total time required

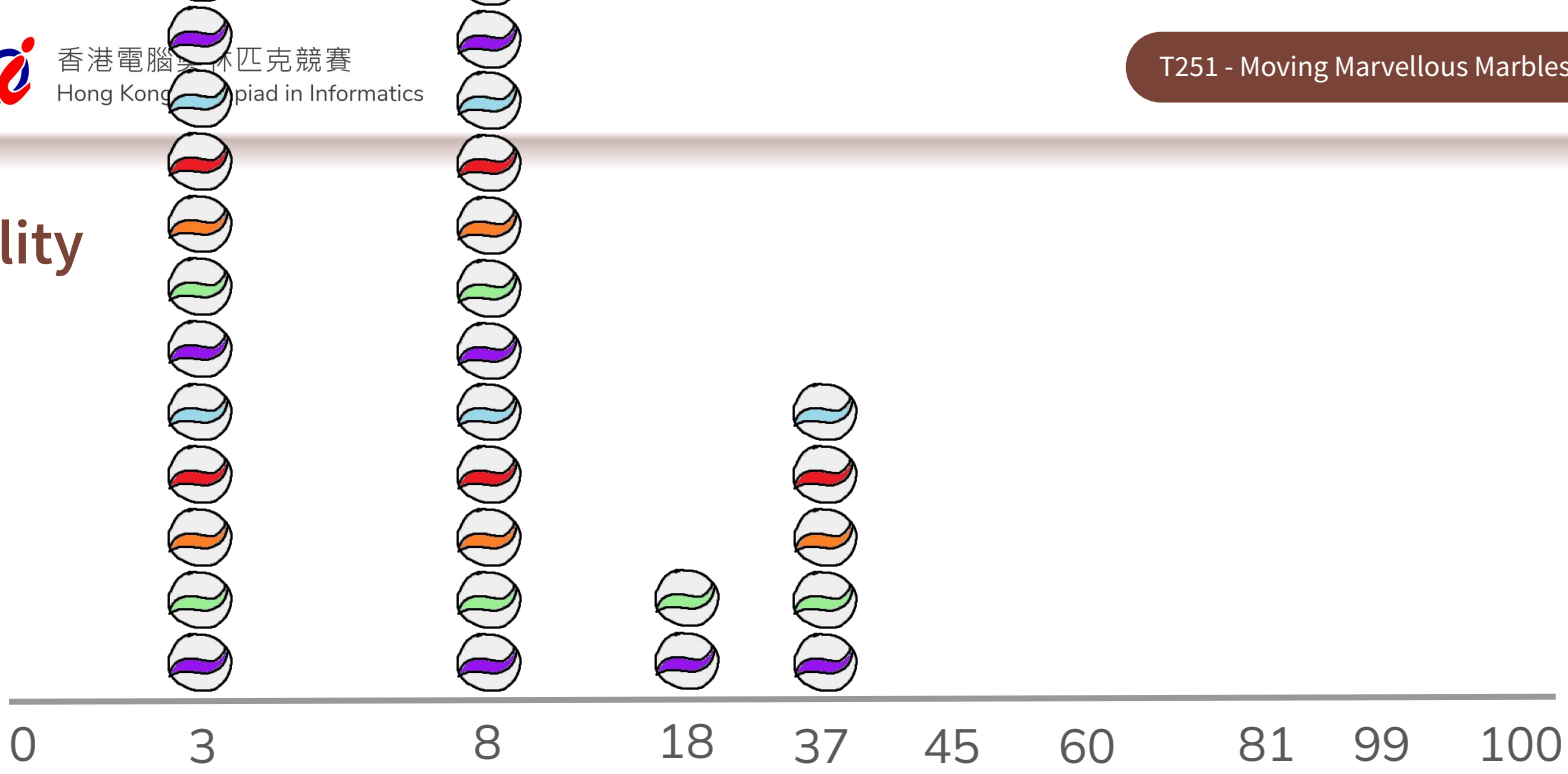


Subtask	Score	K	N	Special Constraints
1	3	K = 0	N ≤ 50000	Chain
2	5			
3	10	K = 1	N ≤ 30	Sum A[i] ≤ 30
4	19		N ≤ 300	
5	8		N ≤ 1500	Chain
6	15			
7	21		N ≤ 5000	
8	18		N ≤ 20000	Complete Binary Tree
9	1		N ≤ 50000	

Dream



Reality

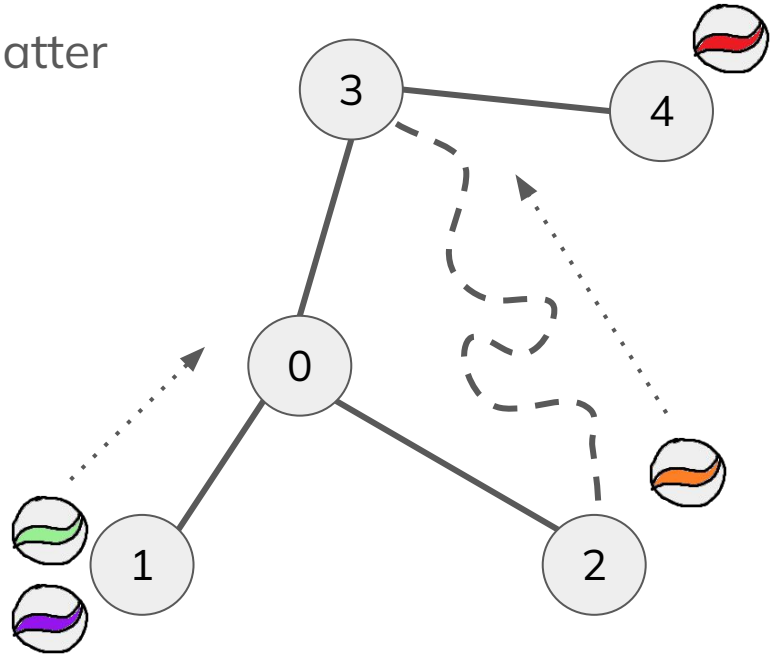


Simplified Problem

Actually, A[i] and B[i] on their own doesn't matter
Only their **differences** matters :)

Initial: C[i] := A[i] - B[i]
Target: C[i] = 0

Move a marble from node u to node v
Pick edge (u, v), do C[u] -= 1 and C[v] += 1



Subtask 1 (3%, K = 0, Chain)

Consider “fixing” the nodes one by one from node 0 to node N-1.

It is not hard to see that greedy works here.

- If you want more marbles in node u, take the nearest ones to the right, and if you want less marbles in node u, throw them to node u+1.

You can simulate this using two-pointers or by maintaining a “buffer”, which will be sum(C[i]) = sum(A[i] - B[i]) for all prefix i. This “buffer” will represent the number of nodes you have left/are missing.

Expected Score: 3

“Flow” Argument

“... maintaining a “buffer”, which will be sum(A[i] - B[i]) for all prefix i. This “buffer” will represent the number of nodes you have left/are missing.”

Contextually, this “buffer” is actually referring to how many times **an edge has been used** to transport marbles.

- Think about **“contribution by edge”**
- Rather than thinking the number of edges a marble has to travel across.

Can this be extended more generally?

“Flow” Argument

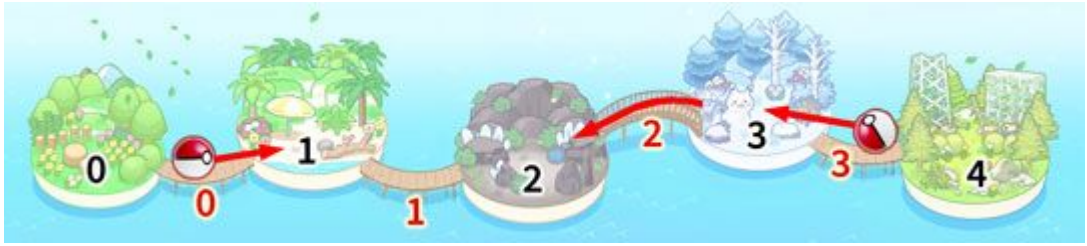
Observation 1. Given a tree, we can calculate the “flow”, i.e., the number of marbles passing through each edge. Why?

- It is easy to observe this is the lower bound.
- We can also inductively show that we can “consume” 1 unit of flow for each movement of marbles.

Generalising from Subtask 1, this value would be **sum(C[v]: v is in the subtree of u)**. The answer is exactly the sum of absolute value of flow.

Note that we should consider ***directed* flow**, + means same direction while - means opposite direction.

“Flow” Argument



In this example, instead of thinking:

“The marble at node 0 must move along 1 edge,
the marble at node 1 must move along 2 edges”,

think of it as:

“Edge 0 has to be used **-1 times** from 1 to 0, **(i.e. it is used to move from 0 to 1)**
edge 1 has to be used 0 times from 2 to 1,
edge 2 has to be used 1 times from 3 to 2,
edge 3 has to be used 1 times from 4 to 3”.

Subtask 2 (5%, K = 0)

Directly calculate the “sum of absolute value of flow” mentioned.

Expected Score: 8

Flow Argument

Observation 2. If we add an edge to a tree, only the **edges on the cycle** formed by the new edge will be affected. The flow of all the other edges are unchanged.

- This can be proved by thinking carefully about the definition of the “flow” described above; it is doing a subtree sum, so only the path will be affected.
- More intuitively, by building a tunnel (u, v), you are trying to skip the original path from u to v to save time => it should make sense that only the original edges on the path from u to v are affected.

Subtask 3 (10%, $K = 1$, $N \leq 30$, $\text{sum}(A_i) \leq 30$)

N is so small, so we can loop over all the $O(N^2)$ possibilities of extra edge.

Observation 3. If we fix the "flow" of the extra edge as F, then it would induce a "flow" F on all the other edges on the cycle in the **opposite** direction.

Why? Conservation of flow (that is, marbles cannot disappear or be created)!

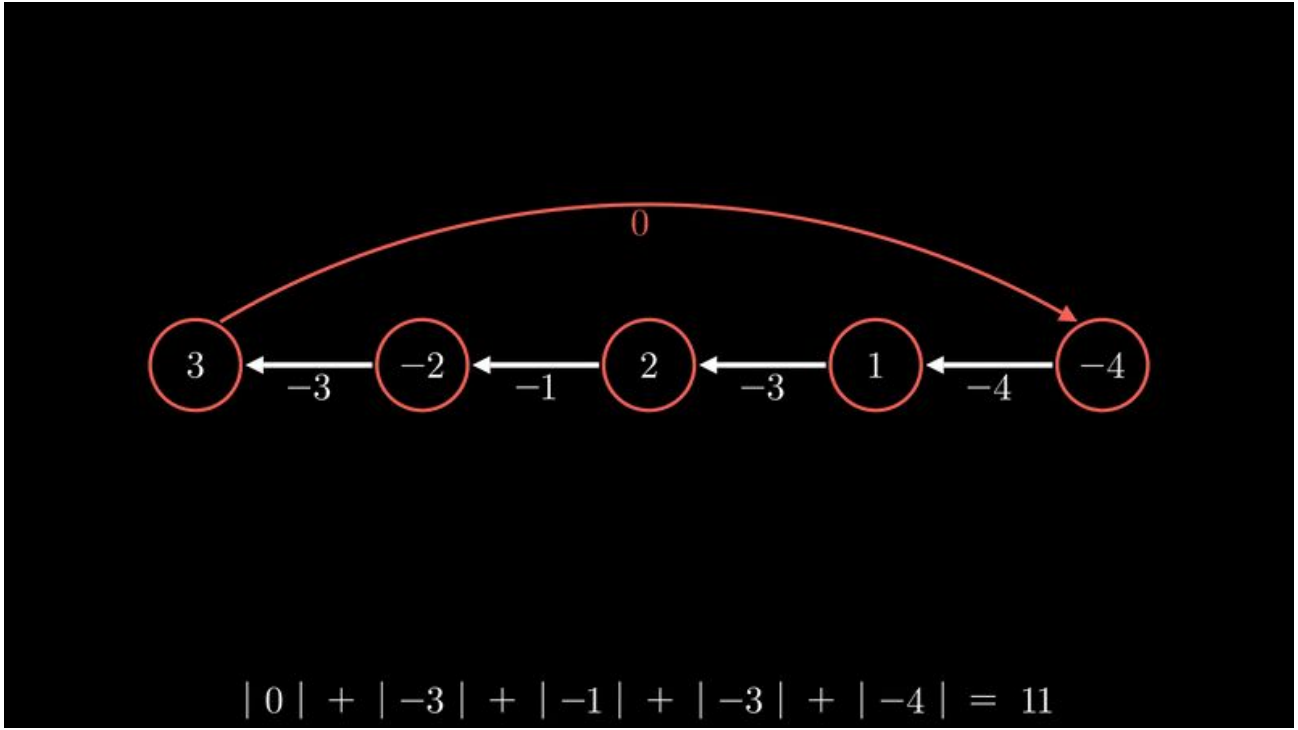
Subtask 3 (10%, K = 1, N ≤ 30, sum(A_i) ≤ 30)

Observation 3. If we fix the "flow" of the extra edge as F, then it would induce a "flow" F on all the other edges on the cycle in the **opposite** direction.

To put this into context, assume you added a tunnel from u to v.

- When you move **exactly** F marbles from u to v using the added tunnel, you will use every edge along the path from u to v (on the tree) F less times.
- Note that by saying "using F times", we are talking about a specific direction to move the marbles; the *actual* times an edge has been used would be the absolute value of what we were describing.

Visualisation of observation



Subtask 3 (10%, K = 1, N ≤ 30, sum(A_i) ≤ 30)

For this subtask, you are also allowed to brute over the flow of the extra edge (denote it as F) as F ≤ sum(Ai) ≤ 30.
Why? Each marble will only pass through each edge **at most once**, so the number of times an edge is used must be less than the number of marbles.

This yields an O(N^3 sum(A_i)) algorithm - brute force over all pairs of edges and F, then calculate the final value by calculating the sum of absolute values of all edge flow. Don't forget to add F to your answer when brute forcing.

Expected score: 18

Subtask 4 (19%, K = 1, N ≤ 300)

Observation 3. If we fix the "flow" of the extra edge as F, then it would induce a "flow" F on all the other edges on the cycle in the **opposite** direction.

Is it possible to not brute force over all possible flow F?

A quick recap on what we want to solve:
Given an array c_1, c_2, c_3, ..., c_k (flows on the path), find F that minimises
$|F| + |c_1 - F| + |c_2 - F| + |c_3 - F| + ... + |c_k - F|$

Subtask 4 (19%, K = 1, N ≤ 300)

Further observe that the optimal F is **equal to** one of c_i.

Therefore, you can sort all c_i and check for all F in O(length) time. This can be done by doing partial sum in both directions.

Time complexity: $O(N^3)$
Expected Score: 37

Alternatively...

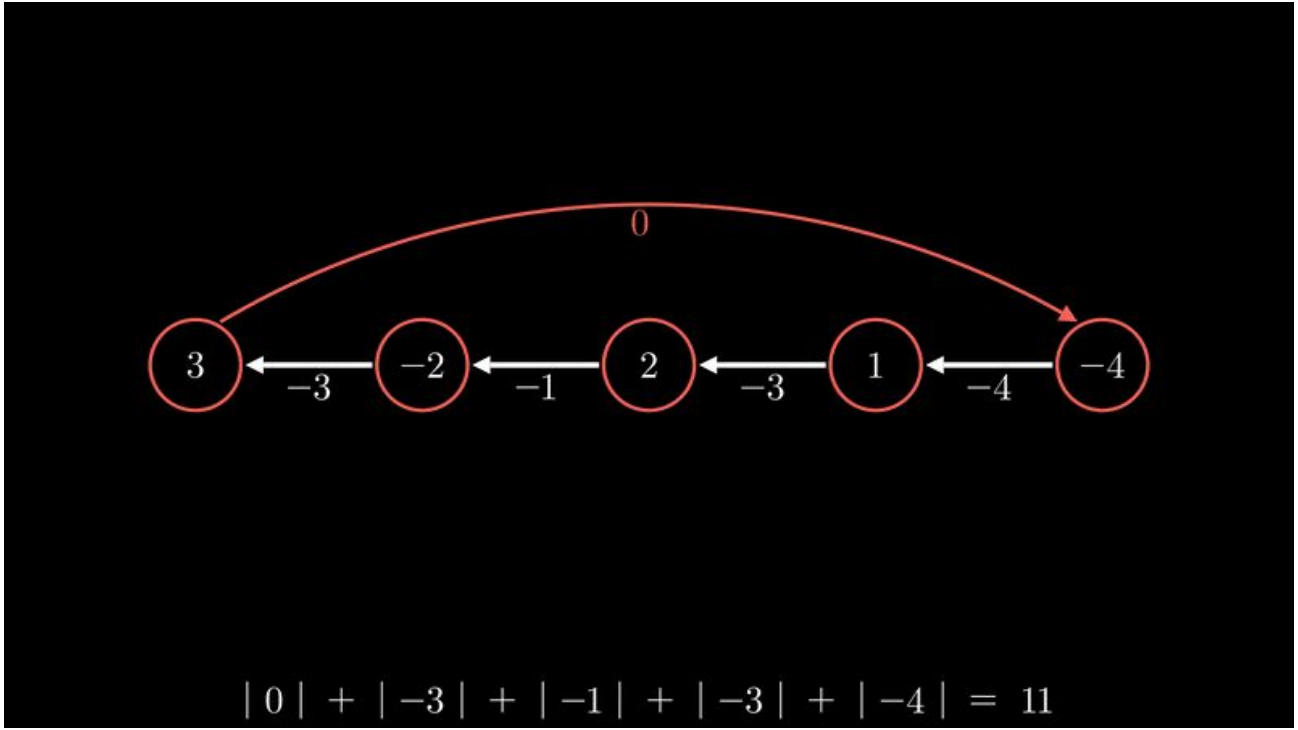
Subtask 4 (19%, K = 1, N ≤ 300)

Let's make more observations!

Observation 4. The optimal flow of the extra edge is exactly the **median** of the flows on the edges **in the cycle**, as counted in the same direction.

More specifically, on a path with k edges, the optimal flow is the floor(k/2)-th (1-based) one in ascending order.

Visualisation of observation



Proof of observation

Observation 4. On a path with k edges, the optimal flow of the extra edge is the **floor(k/2)**-th (1-based) one in ascending order.

This can proven by considering a slight change in flow of the extra edge.
Consider incrementing the flow from F to F+1. How does this affect the flows on the path?

- If the flow is < -F, the change of cost is -1.
- If the flow is ≥ -F, the change of cost is 1.

Let the number of edges with flow < -F be X, ≥ -F be Y. Then the **rate of change of cost** from F to F+1 would be (-X + Y + 1).
Furthermore, notice that as F increases, X is **non-increasing** and Y is **non-decreasing**.
Therefore (-X + Y + 1) is non-decreasing as F increases, which means that the value is maximum if -X + Y + 1 = 0. Hence X = Y + 1 and it would be (approximately) the median.

Proof of observation

More generally, you can also think of it in terms of *convex functions*.

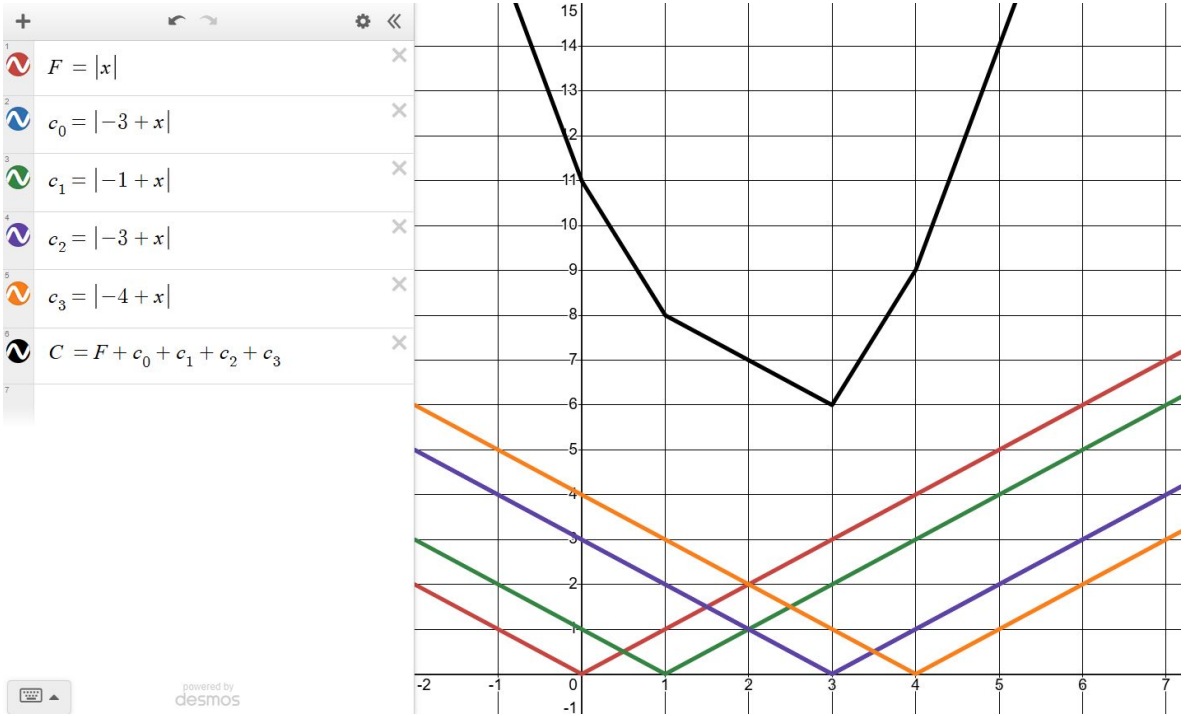
- A *convex function* is a function where the **rate of change** is **non-decreasing** at all times. In the case of absolute functions $f(x) = |x-c|$, the rate of change is -1 when $x < c$ and 1 when $x > c$.

It can be proven that the **sum of convex functions is convex**.

- The proof is actually not too hard: the result of summing up multiple non-decreasing sequences **pointwise** is still non-decreasing.

Therefore, this function attains minimum when the rate of change is 0 (or more precisely, immediately between when it is positive and negative).

Proof of observation



Subtask 4 (19%, K = 1, N ≤ 300)

Back to subtask 4.

A simple way to do this is to loop through all the $O(N^2)$ paths in the tree, then calculate the answer for each of them using the median in $O(N)$.

Time complexity: $O(N^3)$

Expected Score: 37

Subtask 5 ~ 6 (8% + 15%, K = 1, N ≤ 1500)

We can no longer calculate the answer for each path separately.

Therefore, an idea is to calculate the answer **on the go**, i.e. do DFS starting from each node, and add or remove flow to maintain the gain (and median):

```
max_gain = 0
function DFS(v):
    max_gain = max(max_gain, gain())
    For each child u of v:
        add_flow(v, u)
        DFS(u)
        remove_flow(v, u)
```

Subtask 5 ~ 6 (8% + 15%, K = 1, N ≤ 1500)

Subtask 5: The given tree is a chain. Therefore you can directly build a segment tree over it to maintain the median in $O(N^2 \log N)$ time.

Expected score: 8 (Cumulative: 45)

Subtask 6: In fact, you can just maintain the **rolling median** (with undo) using two sets or segment tree in $O(N^2 \log N)$ time.

Expected score: 60

Subtask 7 (21%, K = 1, N ≤ 5000)

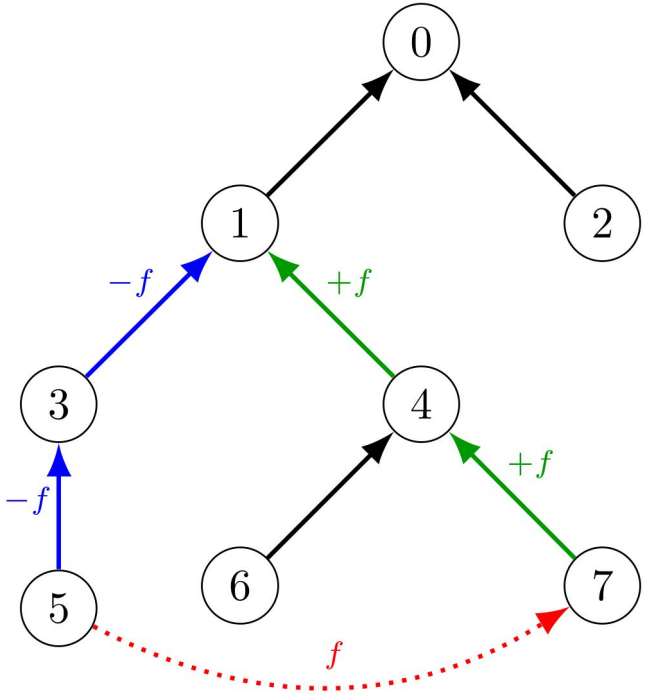
Do we really need to maintain the median? No!

Brute over the flow of the added edge, then do a path dp to find the “added edge” with minimum cost. The “effect” of the flow from u to v would be -f from u to lca(u, v) then +f from lca(u, v) to v.

- Use the following DP state: dp[node][haven’t done anything / going up (+f) / going down (-f) / merged together]

Time Complexity: $O(N^2)$
Expected Score: 81

Subtask 7 (21%, K = 1, N ≤ 5000)



State for each node:

dp[u][0]: haven't done anything

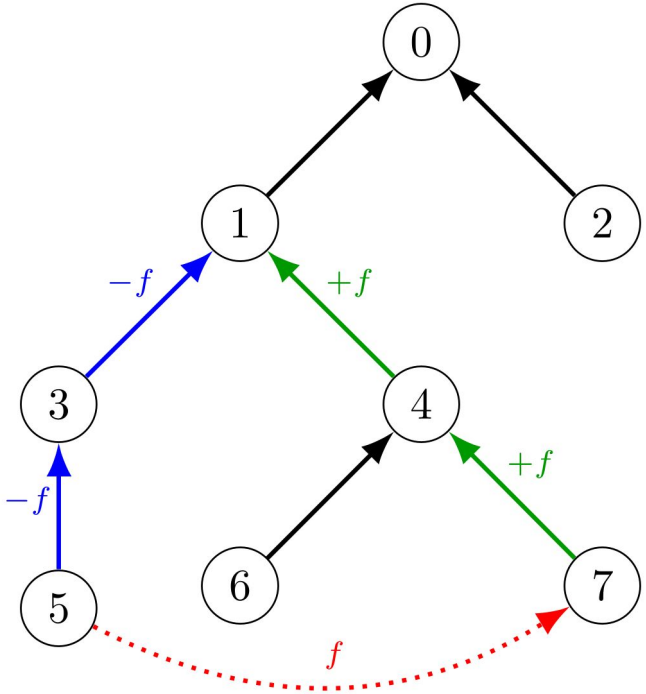
dp[u][1]: doing -f

dp[u][2]: doing +f

dp[u][3]: merged

Assign the flow of each edge to its child, and assign flow 0 to the node

Subtask 7 (21%, K = 1, N ≤ 5000)

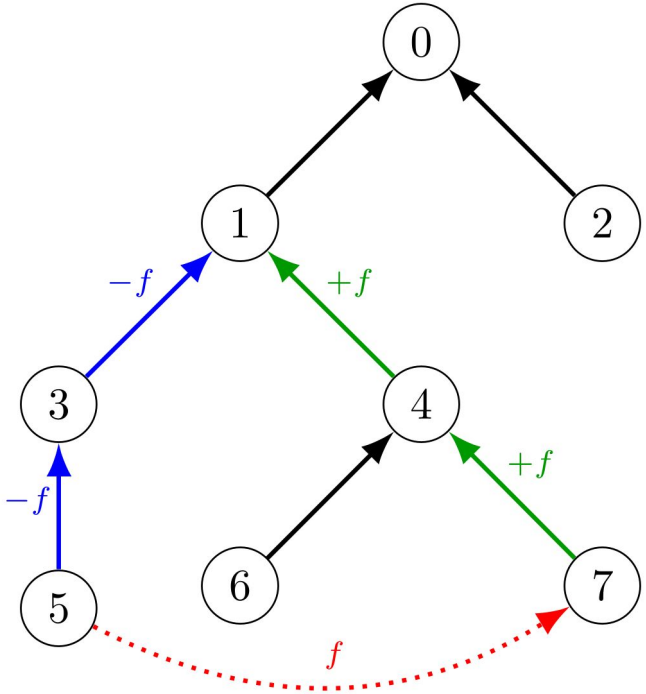


(haven't done anything / doing -f /
doing +f / merged)

Transitions (0 - haven't done anything):

dp[u][0] = sum(dp[v][0) (for all children v of u)
+ abs(flow[u])

Subtask 7 (21%, K = 1, N ≤ 5000)

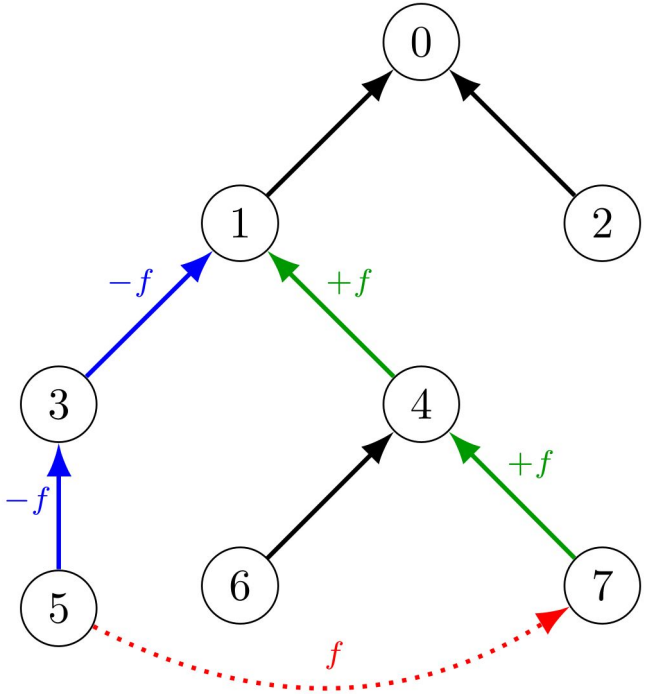


(haven't done anything / doing -f /
doing +f / merged)

Transitions (1 - doing -f):

dp[u][1] = sum(dp[v][0)
+ min(min(dp[v][1] - dp[v][0]), 0)
+ abs(flow[u] - f)

Subtask 7 (21%, K = 1, N ≤ 5000)

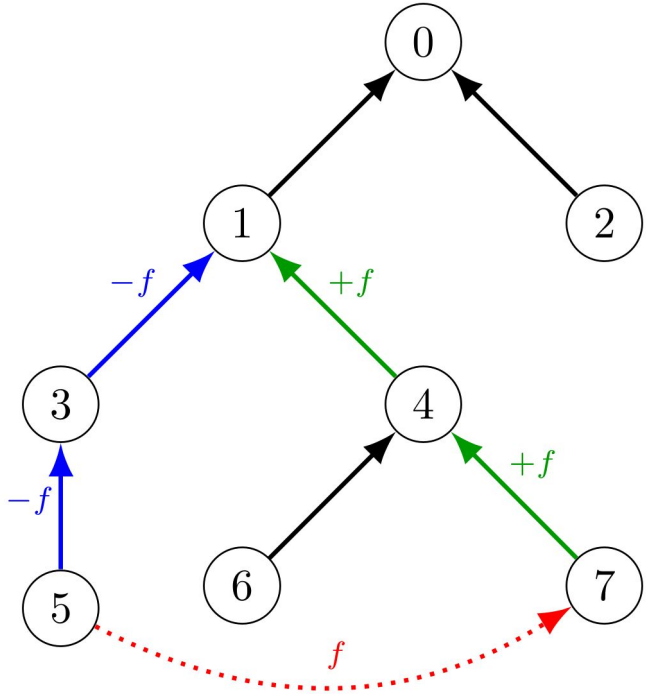


(haven't done anything / doing -f /
doing +f / merged)

Transitions (2 - doing +f):

dp[u][2] = sum(dp[v][0)
+ min(min(dp[v][2] - dp[v][0]), 0)
+ abs(flow[u] + f)

Subtask 7 (21%, K = 1, N ≤ 5000)

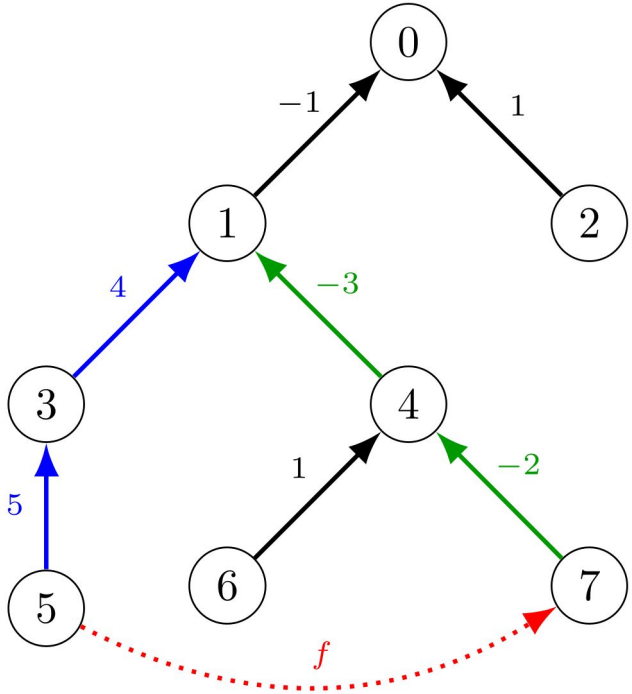


(haven't done anything / doing -f /
doing +f / merged)

Transitions (3 - merged):

```
dp[u][2] = sum(dp[v][0)
           + min(
               (dp[v1][1] - dp[v1][0]) + (dp[v2][2]
               - dp[v2][0]), v1 != v2
               dp[v][3] - dp[v][0]
           )
           + abs(flow[u])
```

Subtask 7 (21%, K = 1, N ≤ 5000)



Example: f = 3

dp[5][1] = 2 (abs(5 - 3))

dp[3][1] = 3 (abs(4 - 3) + dp[5][1])

dp[7][2] = 1 (abs(-2 + 3))

dp[4][2] = 2 (abs(-3 + 3) + dp[4][2] + dp[6][0])

dp[1][3] = 6 (abs(-1) + dp[3][1] + dp[4][2])

dp[0][3] = 7 (dp[1][3] + dp[2][0])

Answer with f = 3: 7 + 3 = 10

Subtask 7 (21%, K = 1, N ≤ 5000)

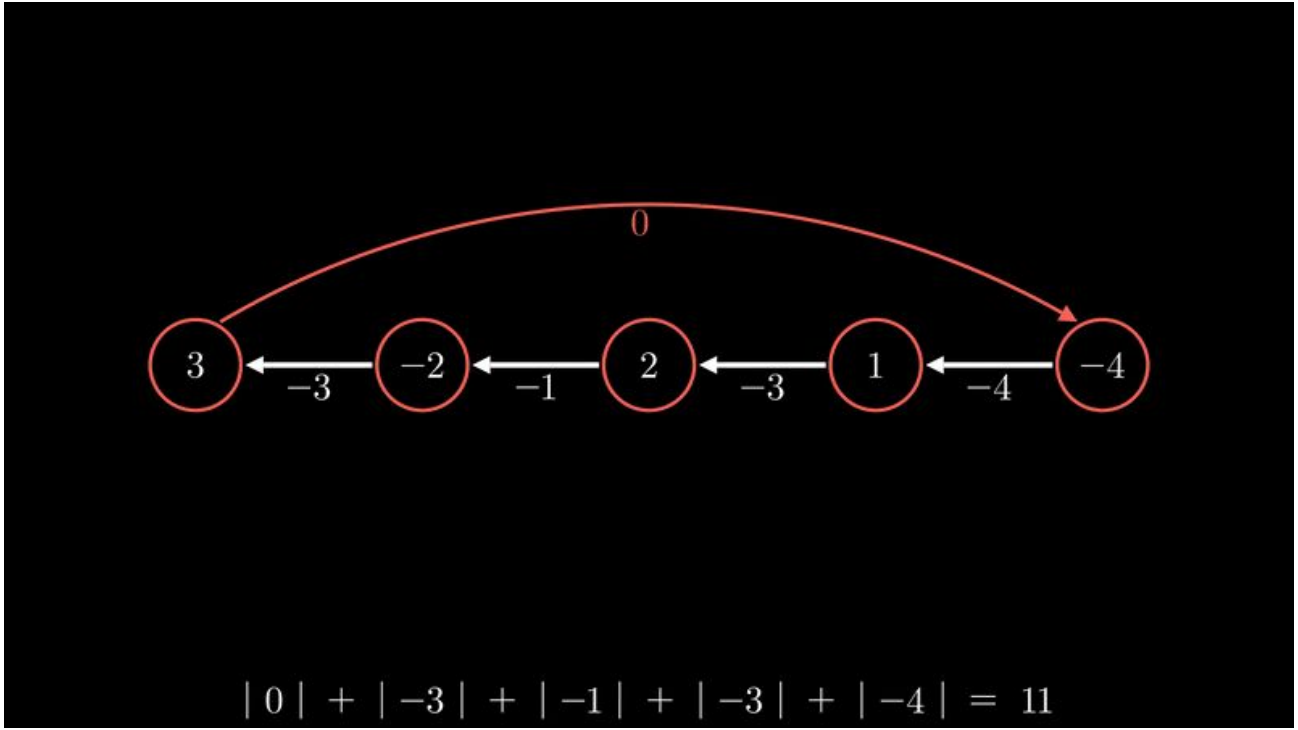
Alternatively... Make more observations!

Observation 5. Optimally, after adding the extra edge, **at least one** of the original edges will have flow 0.
This comes directly from the fact that the extra edge should be the **median** of its path.

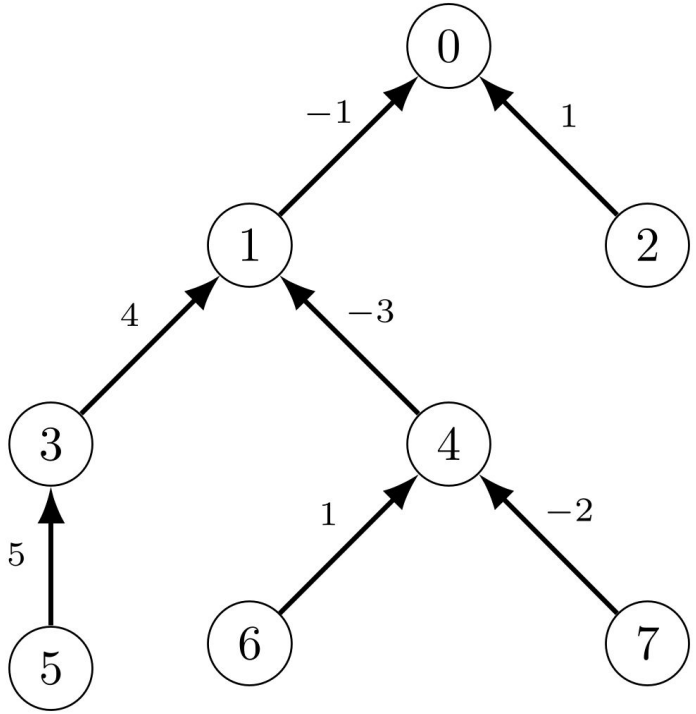
Brute over which edge has flow 0 at the end, then just run a DFS on both sides and find a path with maximum reduced cost.

Time Complexity: $O(N^2)$
Expected Score: 81

Subtask 7 (21%, K = 1, N ≤ 5000)



Subtask 7 (21%, K = 1, N ≤ 5000)

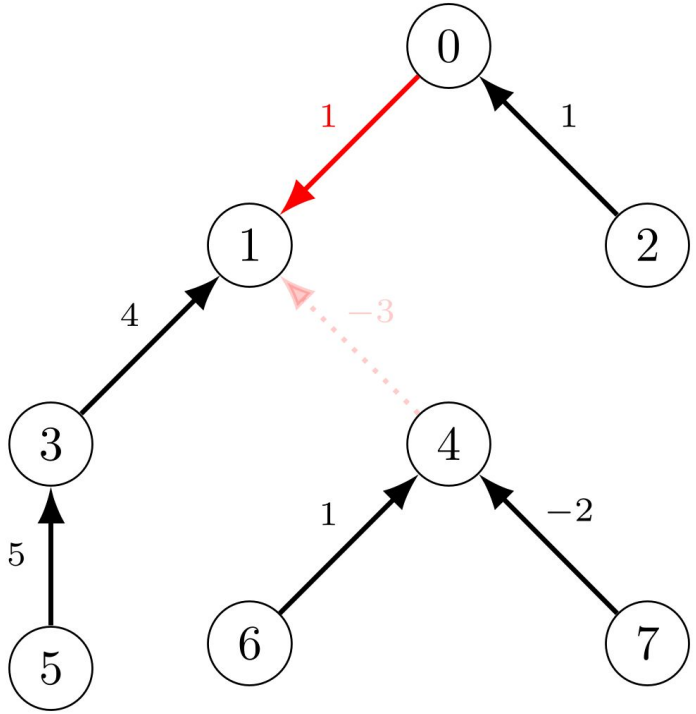


Consider the following example.

Original total sum = 17

Brute over the edge counted as zero, say edge 4 -> 1:

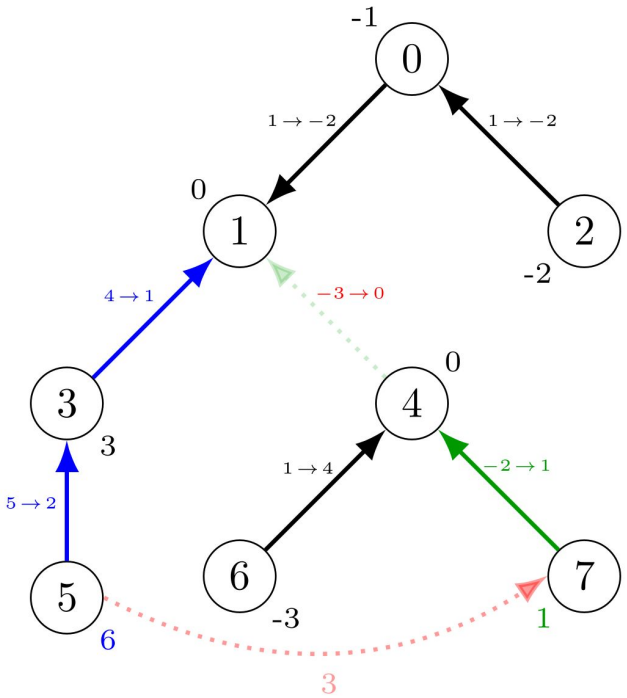
Subtask 7 (21%, K = 1, N ≤ 5000)



Brute over the edge counted as zero, say edge 4 -> 1:

“Erase” the edge, and redirect the edges (in practice, this can be done implicitly)

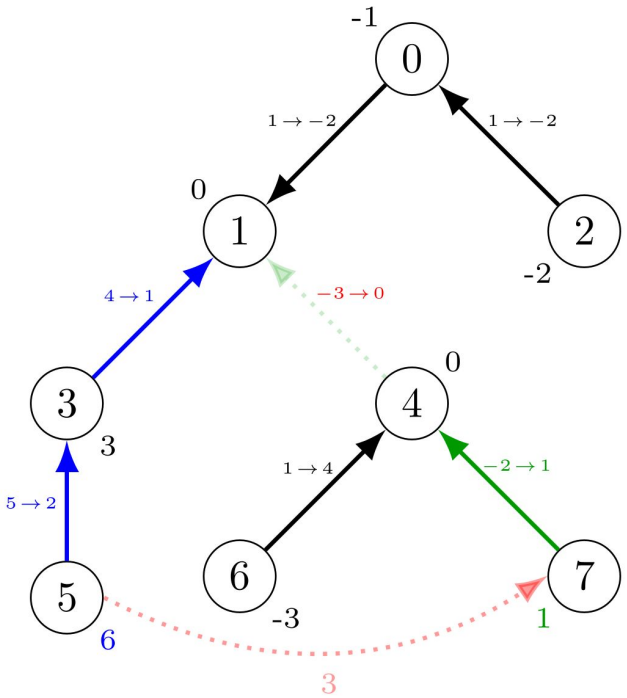
Subtask 7 (21%, K = 1, N ≤ 5000)



Since the newly added edge would be from the 1 side to the 4 side, the edges from 1 should -3 and the edges from 4 should +3

The value next to each node is the “reduced cost” after picking that node as an endpoint, calculated by sum of
abs(flow[u]) - abs(flow[u] +/- f)
along path

Subtask 7 (21%, K = 1, N ≤ 5000)



In this example:

reduced_cost[5] = (|4| - |1|) + (|5| - |2|) = 6
reduced_cost[7] = (|-2| - |1|) = 1

Answer = total sum - reduced cost
= 17 - 7 = 10

Subtask 7 (21%, K = 1, N ≤ 5000)

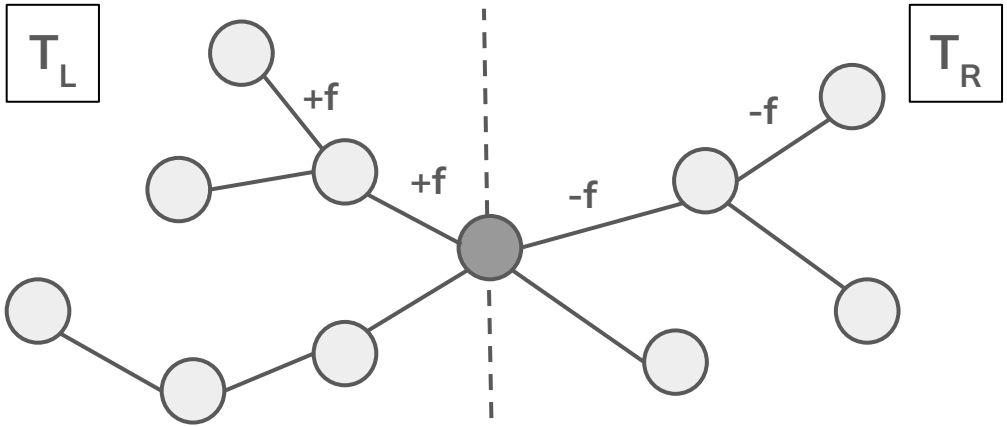
```
# finds maximum reduced cost
def dfs(u, par, flow):
    res = 0
    for v, w in adj[u] and not equal to par:
        res = max(res, dfs(v, u, flow) + (abs(w) - abs(w + flow)))
    return res

# brute force over all edges
deduct = 0
for par, child, flow in edges:
    chmax(deduct, dfs(child, par, -flow) + dfs(par, child, flow))
ans = sum(abs(flow in edges)) - deduct
```

Further than $O(N^2)$

Sadly, “ordinary” centroid decomposition does not work

We enumerate all the $O(N^2)$ paths in before. Can we do it faster?

It seems reasonable for us to split the tree into several **(two)** subtrees:



Sign of the flow: w.r.t. to the “root”

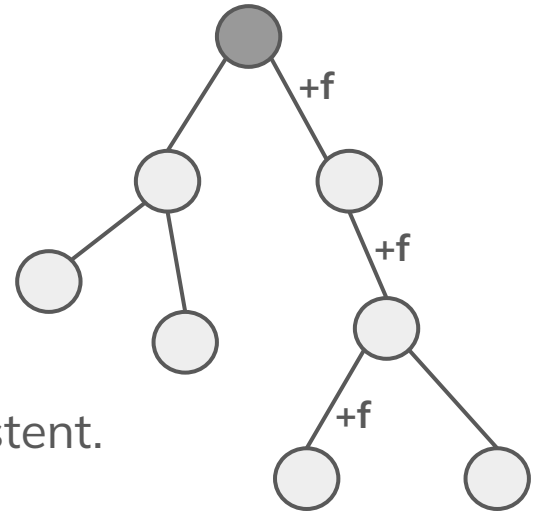
Target: solve for **paths between T_L to T_R**, then recur into two subtrees!

Bold Assumption

Given a subtree T of V nodes, and a list of F = O(V) different flows.

Assumption: We can evaluate minimum delta for the flows in **O(V polylog V)**.

Minimum delta is calculated by inducing a flow,
from any node to the root.



For an edge, if the original flow is f_0, the delta (change) is

$$\Delta = |f_0 - f| - |f_0|$$

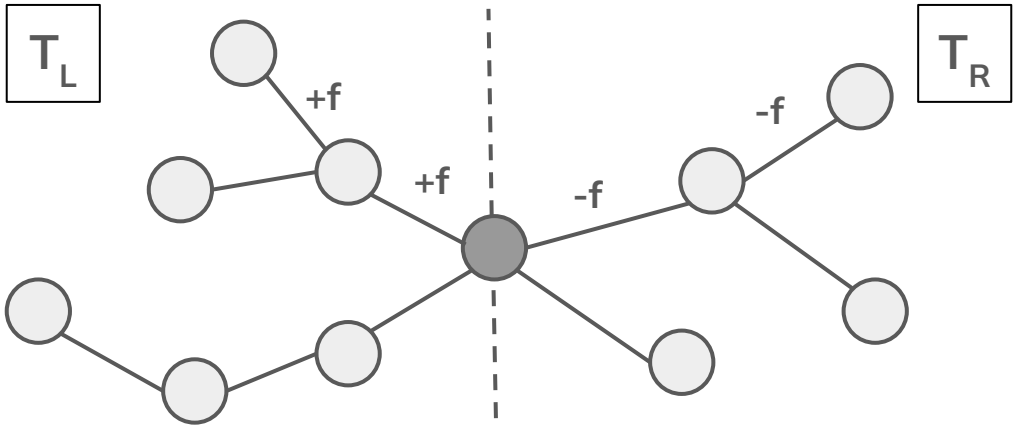
Note: The signs doesn't matter, as long as they are consistent.

Tree Decomposition

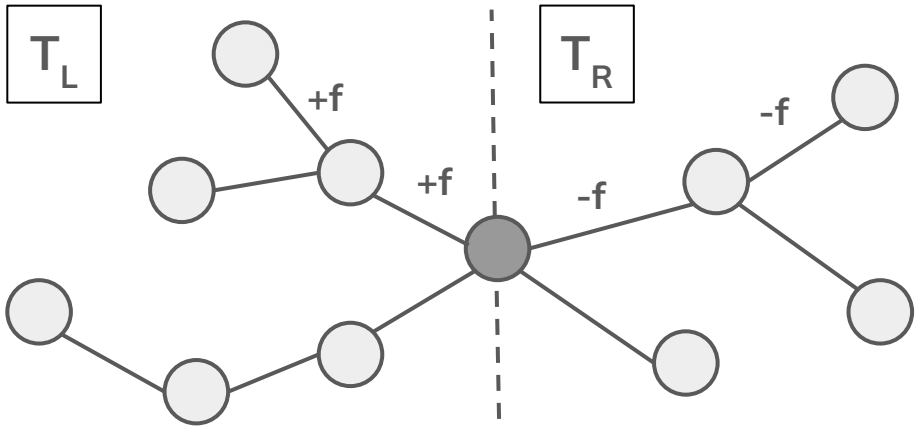
Note we can consider only (discrete) flows that already appears in the tree.

We can calculate the **deltas for the discrete flows** in left and right subtrees.

Merge them by simply adding the respective deltas!



Tree Decomposition

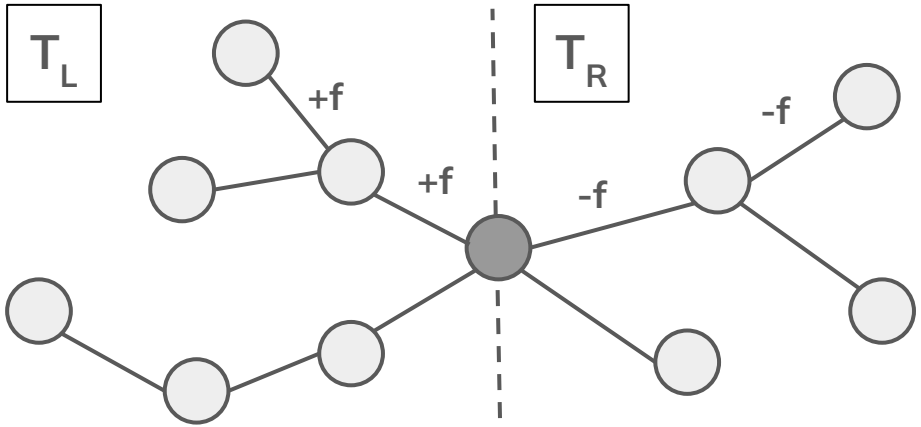


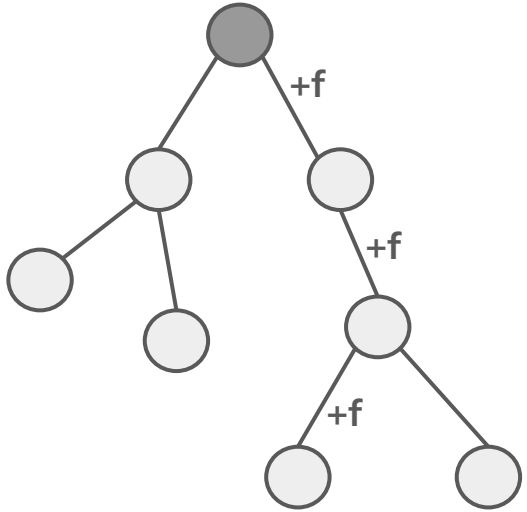

```
function solve(Tree T):
    T_L, T_R = split_tree(T)
    F = possible_flows(T)
    delta_L = min_delta(T_L, F)
    delta_R = min_delta(T_R, -F)
    return min(
        min{delta_L[i] + delta_R[i] + |F[i]|},   # path from T_L to T_R
        solve(T_L), solve(T_R)                   # path in T_L or T_R only
    )
```

Two Steps Away

Two sub-problems ahead of full solution:

1. `split_tree(T)`: How to split the tree “evenly”?
2. `min_delta(T, F)`: How to calculate deltas for flows “efficiently”?

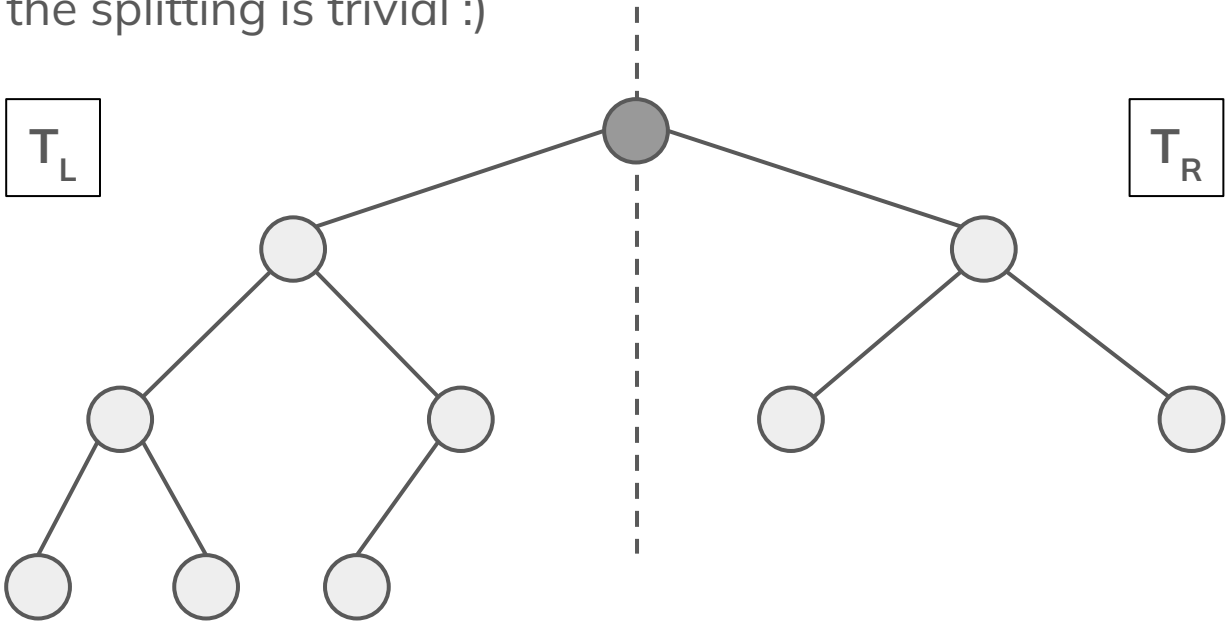




Step 1. Tree Splitting

Subtask 8 restricts the graph to be a complete binary tree.

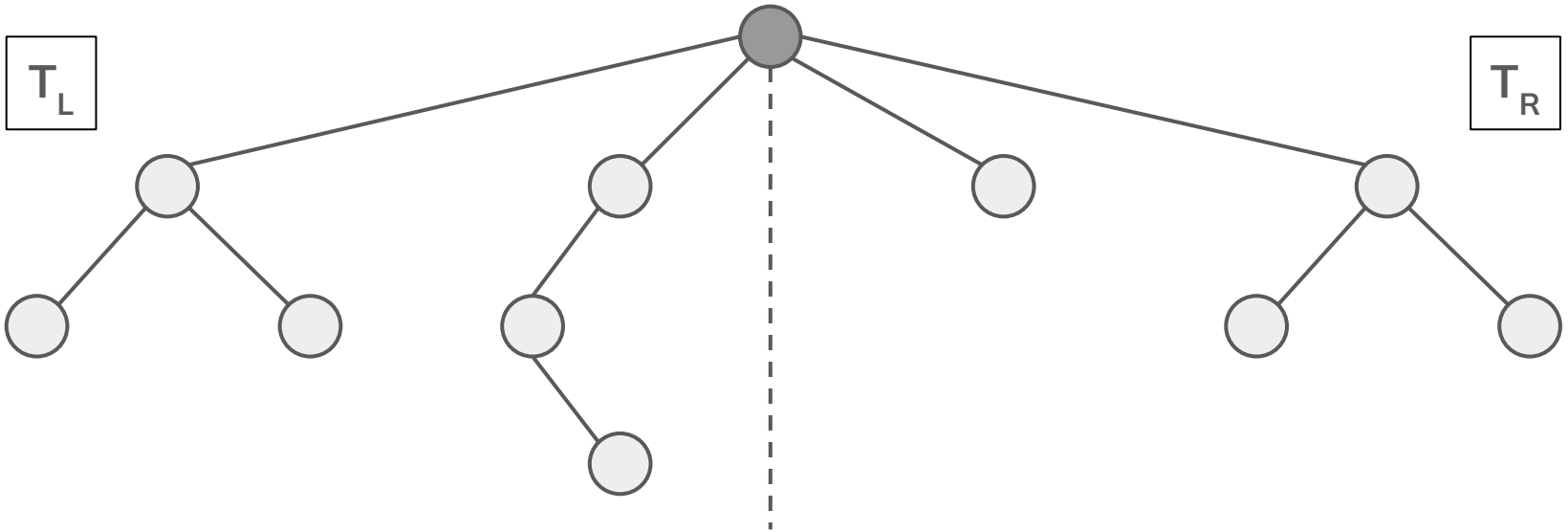
Therefore, the splitting is trivial :)



Step 1. Tree Splitting

Subtree Size ≤ N / 2

How to handle the general case? Find the **centroid**, then use any method to add subtrees to left / right, such that size of T_L and T_R does not exceed **2N / 3**:



Step 1. Tree Splitting

Why **2N / 3**? Suppose the subtree sizes of the centroid are

$$S_1 \le S_2 \le \ldots \le S_k, \text{ where } 1 \le S_i \le N / 2.$$

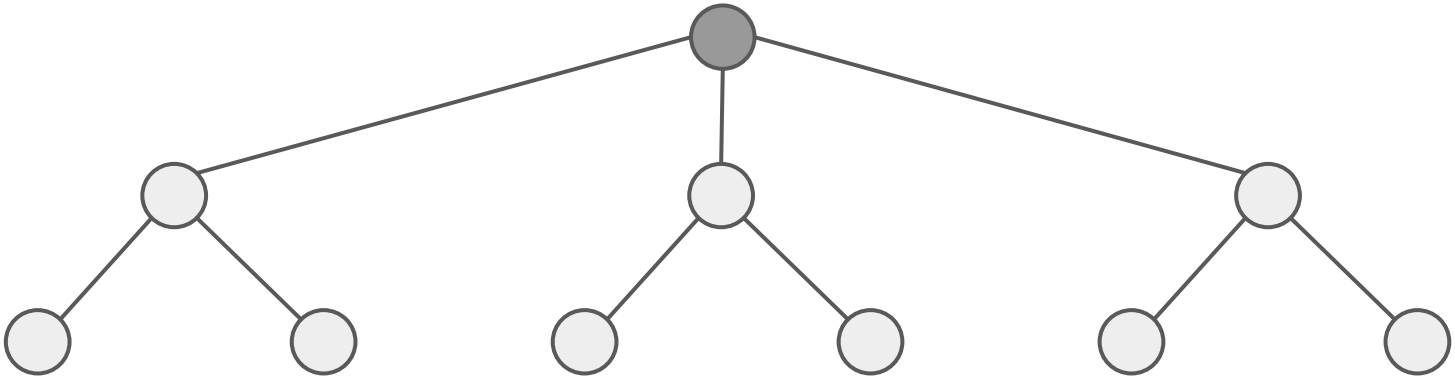
Suppose $\mathbf{S_1 + S_2 + \ldots + S_x < N / 3}$ but $\mathbf{S_1 + S_2 + \ldots + S_x + S_{x+1} > 2N / 3}$.

- Simple math tells us that $S_{x+1} > N / 3$.
- If $x + 1 < k$, then $N / 3 > S_k \ge S_{x+1} > N / 3$, contradiction.
- If $x + 1 = k$, then $S_k > 2N / 3$, also contradiction.

Therefore, add subtrees greedily to left side leads to a [N / 3, 2N / 3] splitting.

Step 1. Tree Splitting

In fact, **[N / 3, 2N / 3]** splitting is the best possible bound:



That being said, we can split tree kind of “evenly”.

Step 1 (alt). Tree Splitting

There is another way to split the tree into two (almost)-equal halves without heuristics on subtree groupings.

Normally in centroid decomposition, we find the centroid **node** in each iterations and consider all paths passing the node.
Alternatively, we can find the centroid **edge**, and consider all paths passing the edge.

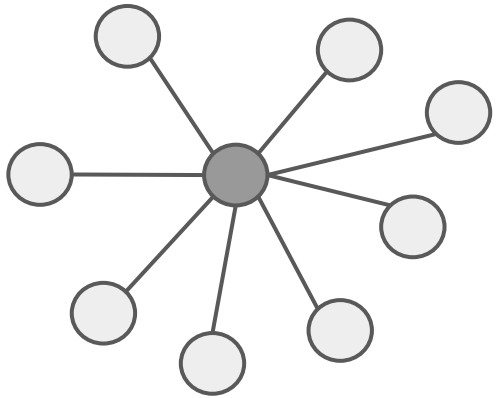
- There are exactly two subtrees connect to an edge, convenient!
- Define the centroid edge as the edge that **minimizes subtree sizes difference**.

Step 1 (alt). Tree Splitting

Find the centroid edge, and consider all paths passing the edge.

There is a slight problem with this approach:

- Ideally, the subtree sizes difference is 0. Then you can achieve good time complexity like centroid node.
- Practically:

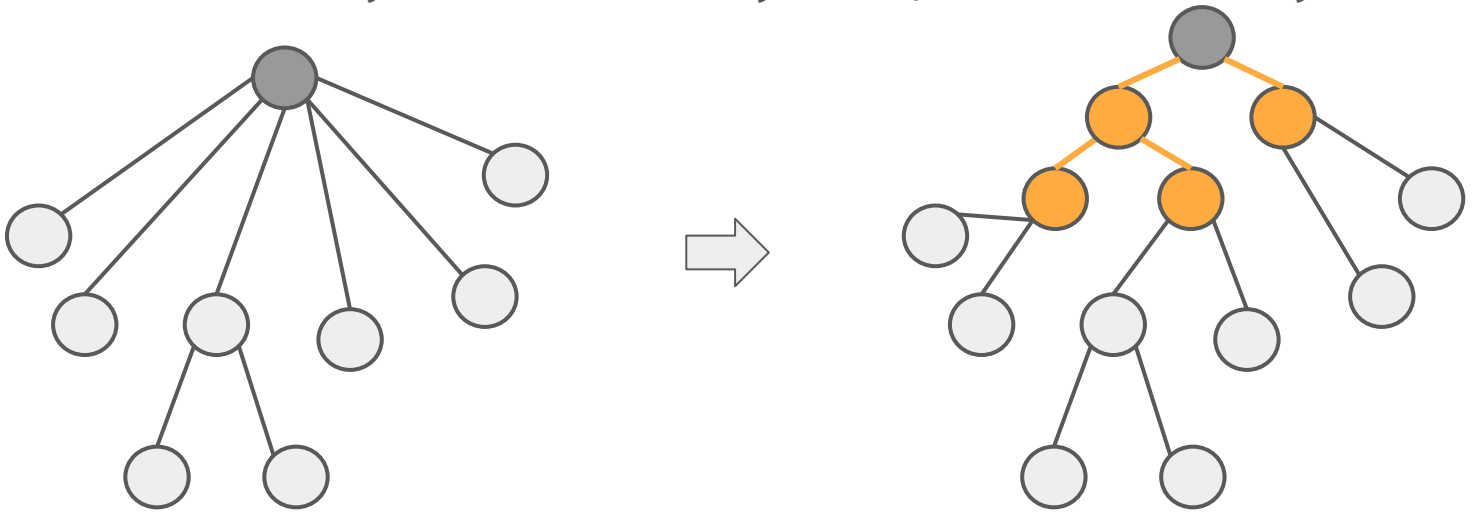


No matter how you choose the edge, the unbalance split makes the problem size **reduce very slowly**.

Step 1 (alt). Tree Splitting

A way to solve this issue is to transform the graph first -> making high-degree nodes into low-degree nodes by adding dummy nodes.

We can **binarize** any tree into a binary tree. (Subtask 8 already did it for you)



Some kind of segment-tree -style dummy node

Step 1 (alt). Tree Splitting

We can **binarize** any tree into a binary tree (Subtask 8 already did it for you)

- At most O(N) additional node and edges are added.
- In a binary tree, at worst you can find an edge that both components after split have <= 2N / 3 edges.

Proof:

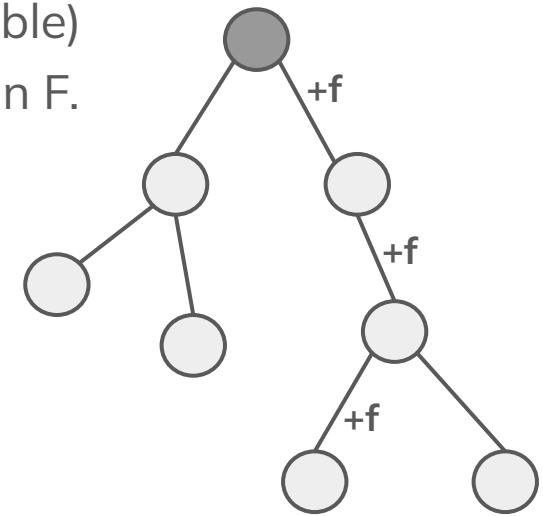
- Let S(v) be number of edges in subtree of v.
- Suppose for each node x, S(left(x)) >= S(right(x)).
- Keep going down left child until S(left(x)) + 1 <= N / 3.
- **N / 3 <= S(x)** = S(left(x)) + 1 + S(right(x)) + 1 <= 2 * [S(left(x)) + 1] **<= 2N / 3**
- You can split the edge between x and its parent.

Step 2. Minimum Delta

Given a subtree T of V nodes, and a list of F = O(V) different flows.

Edge Contribution: $\Delta = |f_0 - f| - |f_0|$ (f_0 original, f variable)

To find: Minimum Δ sum from root to any node, for f in F.



If you somehow get to this point but not yet got $O(N^2)$ solution, you can "naively" calculate in $O(V^2)$:

Time complexity

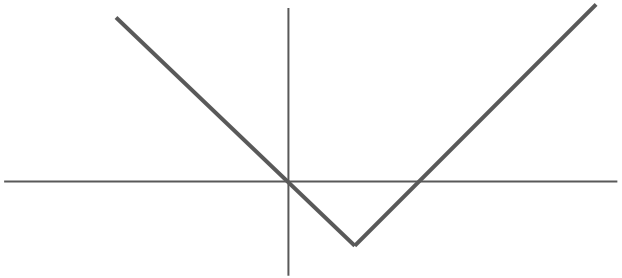
$$f(N) = f(N / 3) + f(2N / 3) + O(N^2)$$

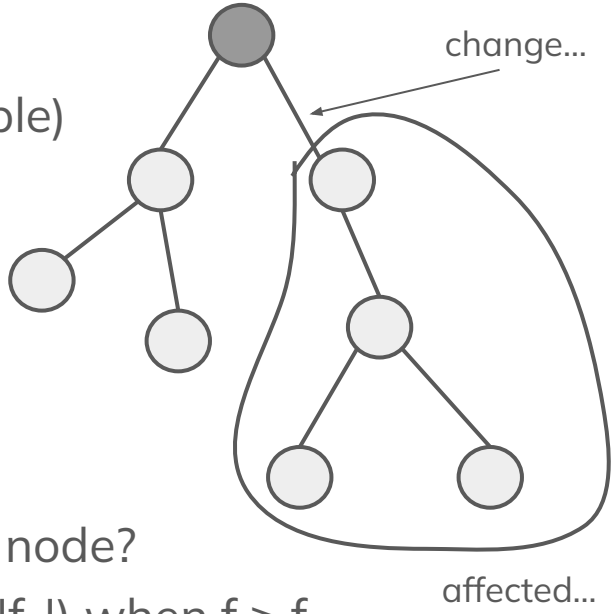
Actually results in $f(N) = O(N^2)$!

Step 2. Minimum Delta

Edge Contribution: $\Delta = |f_0 - f| - |f_0|$ (f_0 original, f variable)

Note that $\Delta = |f_0 - f| - |f_0|$ is piecewise-linear:





Can we directly maintain the linear functions at each node?

Yes, noting $\Delta = f + (-f_0 - |f_0|)$ when $f < f_0$, and $-f + (f_0 - |f_0|)$ when $f \geq f_0$.

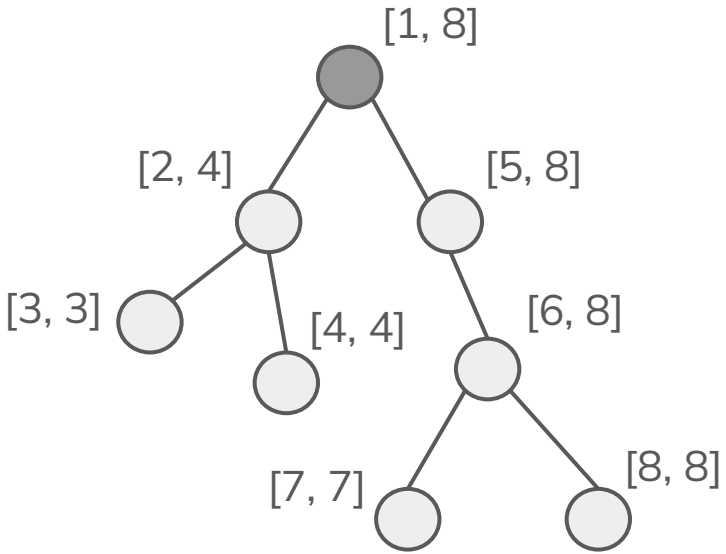
Therefore, we can simply change the edge function when f increase over f_0.

Effect? **Subtree add of linear function!**

Step 2-1. Minimum Delta - Euler Tour

Tempting to flatten the tree with **Euler Tour** :)

subtree add → range add



Fact: In Subtask 8 (Complete Binary Tree), you can skip this flattening part as well.

After flattening the tree, we obtain an even simpler problem:

Step 2-2. Minimum Delta - “Line Container”

There is an array A of N linear functions. Support the following operations:

```
function range_add(int L, int R, line l):
    for i from L to R:
        A[i].m += l.m, A[i].c += l.c

function query_min(int x):
    return min{A[i].m * x + A[i].c}
```

`line {m: int, c: int}` represents the line $mx + c$

Given that both `range_add` and `query_min` are called O(N) times.
Note: `query_min` can be called in increasing order of `x`.

Step 2-2. Minimum Delta - Kinetic Tournament Tree

```
function range_add(int L, int R, line l)
function query_min(int x) # increasing x
```

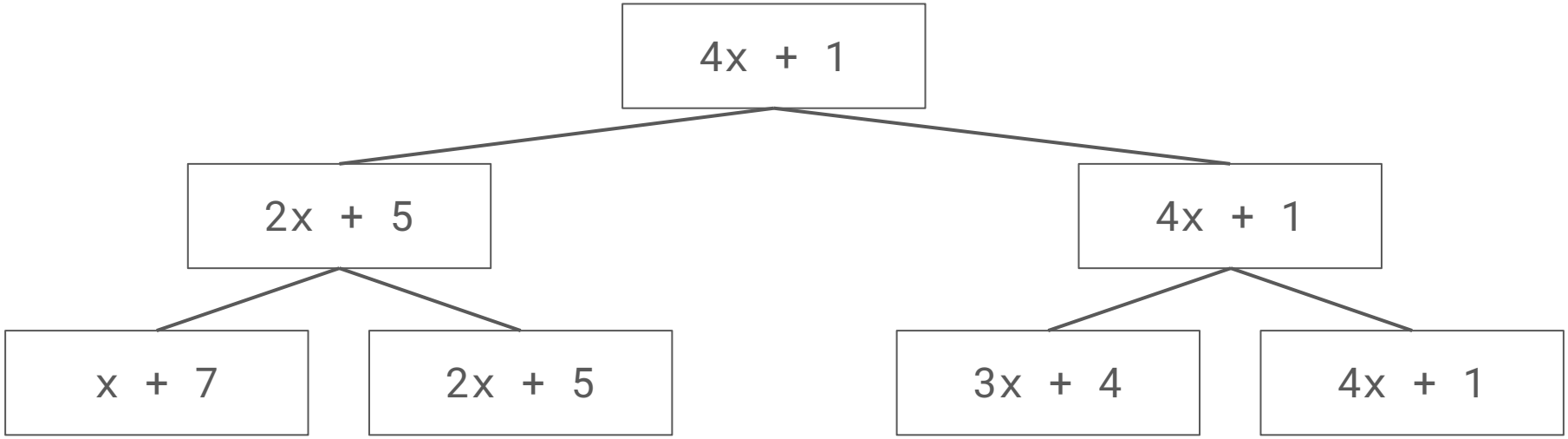
How to maintain? *Segment (Tournament) Tree!*

At every node for range `[L, R]`, we maintain the "winner" for the current `x`. That is, the line `l` such that `l.m * x + l.c` is minimised.

Then we can simply return the line stored at root when `query_min` called.

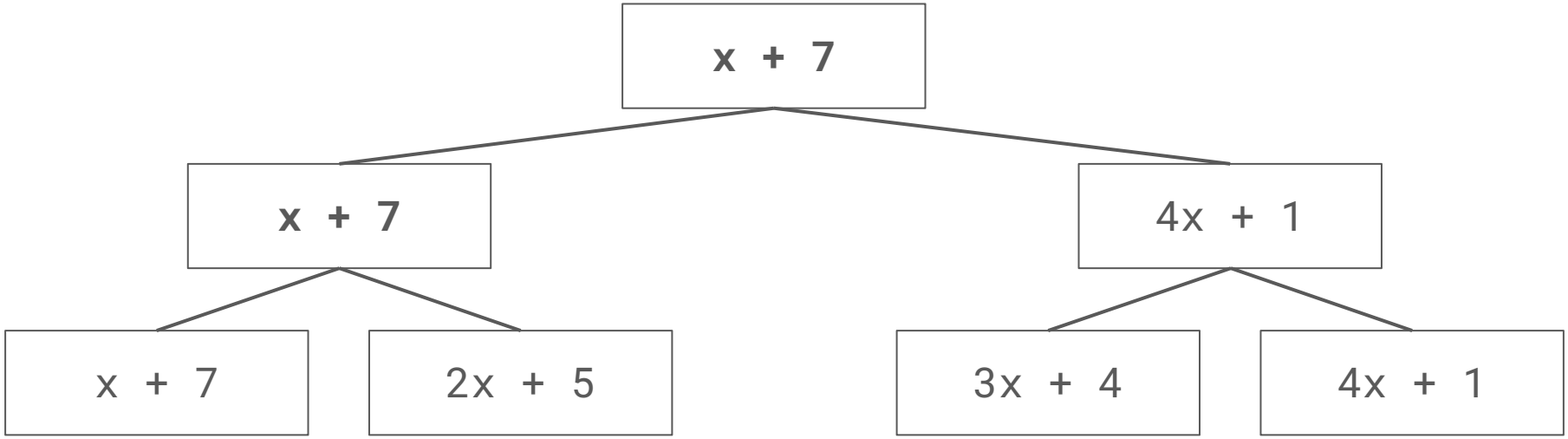
Step 2-2. Minimum Delta - Kinetic Tournament Tree

```
query_min(x = 1)
```



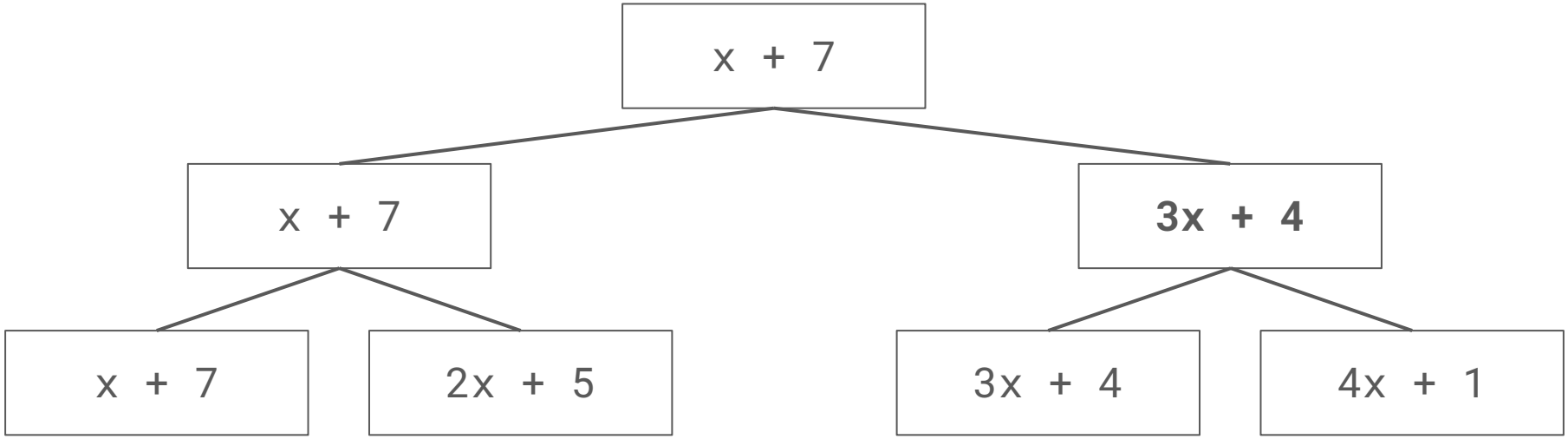
Step 2-2. Minimum Delta - Kinetic Tournament Tree

```
query_min(x = 2)
```



Step 2-2. Minimum Delta - Kinetic Tournament Tree

```
query_min(x = 3)
```



Step 2-2. Minimum Delta - Kinetic Tournament Tree

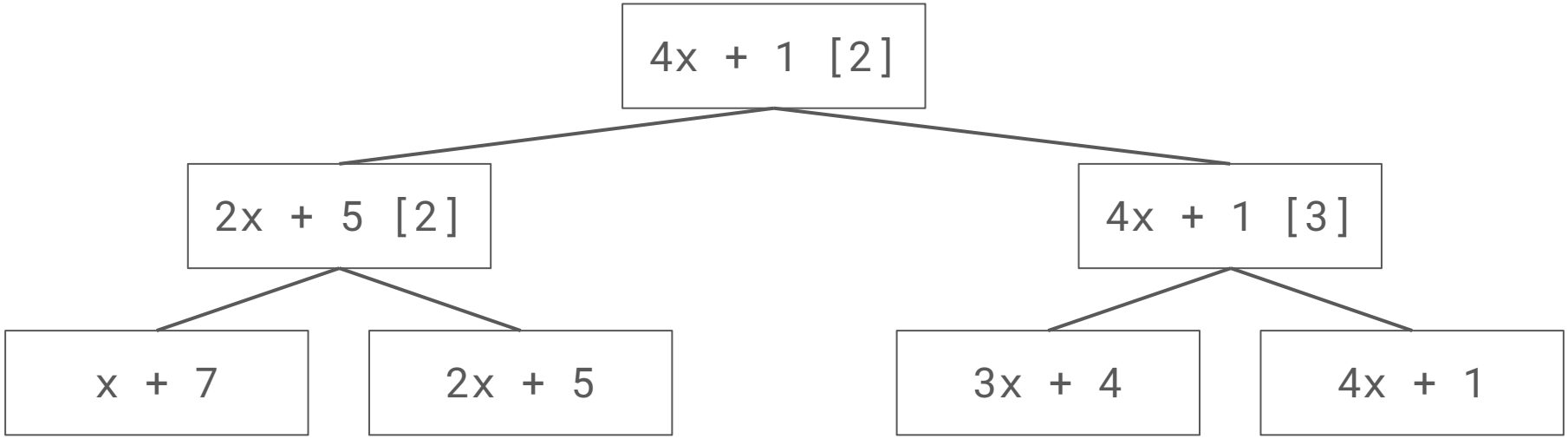
Observation. The winner between two lines l_1 and l_2 is changed at most once.

Therefore, we can store a **threshold** value at each node, which is min of:

- Children's threshold value
- Ceil(intersection of l_1 and l_2) (If slope of l_1 and l_2 equal, then infinity)

If the current x is greater than the threshold at the current node, then recur downwards, and re-compute the winner and the threshold. Otherwise, you can simply break.

Step 2-2. Minimum Delta - Kinetic Tournament Tree



Step 2-2. Minimum Delta - Kinetic Tournament Tree

The rest is the same as normal segment tree with lazy propagation :)

In fact, the time complexity is $O(N \log^2 N)$. Some heuristic arguments:

- Each `range_add` operation affects O(log N) segtree nodes.
- It takes us O(log N) time to resolve a “fight” between two lines, which then the threshold of that node is set to infinity.

Full Solution

Step 1: Tree Decomposition - O(log N)

Step 2: Euler Tour + Kinetic Tournament Tree - O(N $\log^2$ N)

Final Time Complexity: O(N $\log^3$ N) :)

Expected Score: 99 or 100 (depends on your implementation)