Finalized First-Pass Algorithm Dataflow:
To understand this, you will need to understand first:

Topic	**Concept**	**Recap**
Layerwise Relevance Propagation	What the purpose and outputs of **Layerwise Relevance Propagation (LRP)** are.	Redistributes a prediction score backward through the network to assign *relevance scores* to input features.
	What a **Propagation Rule/Function** is.	Defines how to split and redistribute *out-relevance* from outputs back onto inputs at a given operation (e.g., linear, ReLU, softmax).
	How LRP is theoretically run.	Start at the final prediction score, walk backwards through the graph, applying rules at each operation to conserve relevance and push it down toward the input.
	How to orient yourself directionally when discussing LRP, i.e. backward vs. forward, children vs. parents, in-edges vs. out-egdes, in-relevance vs. out-relevance.	**Backward**: means traversing in the opposite direction of forward execution. **Children/Parents**: a node's children are its inputs (previous ops in forward), parents are its consumers (next ops in forward), following the semantics of autograd and .next_functions. **In-/Out-Edges***:* in-edges are inputs flowing into a node; out-edges are outputs flowing to children. **In-Relevance/Out-Relevance**: out-relevance is the relevance score assigned to this node; in-relevance is what it distributes to its chilren.
	What an **LRP Checkpoint** is in our use case.	An injected identity op that signals when to snapshot the current relevance value during traversal.
Autograd	What the Autograd Graph is, how we traverse it, and how it helps us perform LRP.	PyTorch's dynamic graph of operations built during forward. Traversed backward for LRP.
	What an Autograd Node is, what kind of data one stores.	A `grad_fn` object representing an operation. Holds references to children (inputs), saved tensors, metadata, etc

Promises	What a Promise is and what purpose it serves.	A placeholder for relevance that cannot yet be resolved because dependencies (arguments or sibling branches) are incomplete. Lets us avoid caching extra activations while limiting retraversal of the autograd graph.
	What an "**arg Node**" is in the context of Promises.	A descendant autograd node that a promise references as one of its inputs.
	What a Promise is composed of and when/how these members change during traversal.	- Reference to the input out-relevance. - References to arg nodes and the args themselves. - References to the in-relevances meant to go to the arg nodes. - The start node index where relevance is injected. - Stored transform functions for arguments and relevance. - Status flags (ready, complete). - Links to parent and children promises.
	How a Promise branch retrieves arguments and propagates back relevance without requiring extra graph traversal outside of the baseline backward pass.	During descendant traversal, we record the transforms for moving arguments from the arg-node → start-node, and relevance from start-node → arg-node. These stored transforms let the branch pull arguments without extra graph traversal.
	Promise Trees, i.e. when and how Promises nest/link as parents and children.	Nesting that occurs when a promise must spawn child promises during the search for its own arg-nodes.
	What is required for a Promise to be **ready** and **complete**.	Ready once all arguments are satisfied; complete once relevance is redistributed.
	When a Promise branch "**stalls**", i.e. has to wait for another event before the subtree beneath it can be processed.	Happens when a promise waits on unresolved children or sibling branches before propagation can resume.

Pre-Promises	What a Pre-Promise is, what purpose it serves, and when it is applied.	A placeholder for relevance requests created before a full promise exists, helps in preventing Promise deadlocks by pushing past traversal heuristics to continue arg-search.
	How a Pre-Promise interacts with its parent Promises (if any exist).	At first, it is not listed as a child to its parent. Once the parent promise resolves, and all inputs at the Pre-Promise start node land, the Pre-Promise receives all connections to Promises leading into it for full propagation.
Traversal Modes	What “**Normal Traversal Mode** (**NTM**)” is, what heuristics it follows, and why it follows these heuristics.	Standard backward walk; only propagate once all inputs/in-edges have been processed.
	What condition(s) must be true of a Node to be placed in NTM.	Node reached directly by following `.next_functions`.
	What condition(s) must be true of the traversal state for a Node to be retrieved from NTM.	Node is on the NTM stack.
	What “**Promise Traversal Mode (PTM)**” is, what heuristics it follows, and why it follows these heuristics.	Breaks out of NTM to prioritize promise setup and arg-search.
	What condition(s) must be true of a Node to be placed in PTM.	Node has an incoming promise needing arg-search, but hasn’t received all its other inputs.
	What condition(s) must be true of the traversal state for a Node to be retrieved from PTM.	Node is on the PTM stack..
	What “**Promise Fulfillment Mode** (**PFM**)” is, what heuristics it follows, and why it follows these heuristics.	Restarts stalled promise branches once dependencies resolve.

	What condition(s) must be true of a Node to be placed in PFM.	Promise is ready (all deps satisfied) and marked complete.
	What condition(s) must be true of the traversal state for a Node to be retrieved from PFM.	Node is on the Promise Queue, and the Promise saved to it is now complete.

Initialization

Assume we have in- and out-adjacency lists of the Autograd Graph.
Assume we have the output of the model/last hidden states, (named hidden_states).

root := hidden_states.grad_fn

input_tracker := { Node : [None for each parent of Node] for each Node in the Graph }
nodes_pending := { Node : len(in_adj[Node] \ None) for each Node in the Graph }

Note: Using pseudocode for set difference

Note: We say that all of a Node's inputs have landed when nodes_pending[Node] = 0.

normal_stack := [root]
promise_traversal_stack := [ ]
promise_queue := [ ]

Extracting the next node

While any of the 3 Node collections are non-empty and not all LRP Checkpoints have been visited, check:

- If promise_queue is non-empty, dequeue the next node as curnode if either:
    - The Node has an attached Promise and it is complete.
        - Enter **Promise Fulfillment Mode (PFM).**
    - The Node has an attached Pre-Promise and all of its inputs have landed and all of its parents are complete.
        - Enter **Normal Traversal Mode (NTM)** (with Pre-Promise implications).
- Else if promise_traversal_stack is non-empty
    - Pop the top node from it as curnode.
    - Enter **Promise Traversal Mode (PTM).**
- Else pop the top node from normal_stack as curnode

Processing curnode

Check which mode we are in:

- If PFM, get the promised in-relevance from the attached Promise, use this as the out-relevance (propagation input) for curnode.
- If PTM, we create a new **DummyPromise**, a single-branched, pure placeholder Promise whose only goal is to search and retrieve, not add any additional computation or rules other than what it encounters during its search. This DummyPromise, which we will now call a **Pre-Promise** (we will explain the intuition later on for this name) is then sent through and propagated as normal, and we attach it to the metadata of the current Node, i.e. the Node that the Pre-Promise started at.

    - **Note: In Promise Traversal Mode (PTM), even if there is only one input Promise, a new Pre-Promise will still be created. The Pre-Promise will have the input Promises as parents, but it will not be added as a child to these Promises. This is so that the Pre-Promise does not have its backward chain executed when any of its parents complete, as being a Pre-Promise implies that we have not followed the main traversal rule of requiring all inputs to have landed.**
- If NTM, see the following section on **Input Merging**.

Sub-algorithm: Input Merging
Before we can propagate relevance through a Node, recall that we need all the inputs of the Node to have landed. However, for simplicity, propagation functions only accept only one out-relevance object as input (yes I know it's tricky but this is the correct way to define it), either a Tensor or a Promise which acts as a placeholder for relevance that has yet to be calculated.

With inputs that are all Tensors, this is trivial, we will simply add them together, as that is how relevance accumulates, then we will propagate the summed relevance backward.

However, we must consider when there exists multiple Promise inputs, or when there exists both Tensor and Promise inputs.

We then further split Promises into **complete** and **pending/incomplete** Promises.

We now must consider if the current Node has already been traversed in PTM, and thus has a Pre-Promise attached to it or not.

- If the current Node has a Pre-Promise
    - then this implies the following:
        - The Pre-Promise already has the argument that any incomplete Promises at this Node are still looking for, since we prioritize Pre-Promise search, and thus can act as an **arg-node**.
        - There exists one Promise in our inputs that is already a parent to the Pre-Promise, which follows by the condition for entering PTM. Though, we do not know for sure if this Promise is complete or not at this stage.
        - The Pre-Promise already has its fwd/bwd computation chains compiled, but it has only run the fwd() chain, and has yet to propagate relevance via bwd(). We can assume this since this iteration is in NTM, so it is the earliest time at which the current Node has all its inputs landed.
        - Since the graph is a DAG, and due to our traversal heuristic, we know this is exactly the second time we have come by this Node.
    - Now, do:
        - For each of the Pre-Promises existing parents, add the child connection to the Pre-Promise if it does not exist. Then, add the parent's in-relevance to the Pre-Promise's out-relevance, if the parent is complete.

        - For all pending/incomplete Promises, establish parent-child links with the Pre-Promise, if they do not exist already. Then, set the arg-node index of all these pending Promises to the current node's topological index.
        - Request the Pre-Promise's completion, which will handle setting the arguments of any incomplete parents, propagating upwards, then back down as parents complete.
        - If the Pre-Promise remains incomplete after requesting completion, simply add it to the Promise Queue, where we will come back for a third and final time later.
        - If the Promise is complete, we now check the Promise's tail/arg-node and add it to the stack if all of its inputs have now landed.
        - If the tail node is the same as the current node, i.e. the Promise only had to reach forward one Node, then continue with normal propagation for this Node, using the Pre-Promise's now-populated in-relevance as the out-relevance for this Node.
    - This essentially "fans-in" all Promises that could have accumulated at this Node after the initial PTM going through this Node, and restores them to be true parents and children of each other to allow for normal traversal afterwards.
- If there is no Pre-Promise at this node, check:
    - If there are Tensor inputs landing AND incomplete/pending Promises landing, create a new DummyPromise to act as an **Aggregate Promise**, which holds both parent and child connections to the pending Promises, and set the out-relevance of this Aggregate Promise to the sum of all the Tensor inputs.
    - If there are multiple pending Promises landing, do the same but without changing the out-relevance of the new Aggregate Promise.
    - If there are no pending Promises landing, take the sum of all Tensor inputs and complete Promise in-relevances, and set this as the new out-relevance of the current Node, proceed with normal propagation for this Node.

Back to the Main Algorithm

- Now, we have that any inputs that can/must be propagated forward are merged together and passed here. We run the propagation function corresponding to the current Node with this input.
- Whether the output is a Tensor or Promise, we will iterate through each child-output zipped pair and do the following:
    - Add the output to its corresponding slot in the child's input_tracker entry, matching the indexing of the child's in-adjacency list.
        - e.g. If we had A -> C and B -> C, in_adj_list[C] = [A, B], and we just propagated through B, prop(B) would be assigned to input_tracker[C][1], since B holds index 1 in in_adj_list[C]
    - Reduce the number of nodes pending for the child
    - If the child has 0 nodes pending after this, push it to the NTM stack.

- If the child has an open Promise input, and not all inputs landed, push it to the PTM stack.