Variants of Segment Tree

Ethen Yuen {ethening}

2025-06-28

Beyond Basic RMQ

You *(HKOI Training Camp 2025 attendees)* should be familiar with the basic usage of segment trees, Range Min/Max Query and basic range operations.

But the power of Segment Tree goes much further!

- Most of the Data Structure tasks you'll see in your competitive programming life are basically **different Segment Tree applications.**
- It is a very flexible data structure with a lot to learn about!

Lecture Goals

Today's Goal:

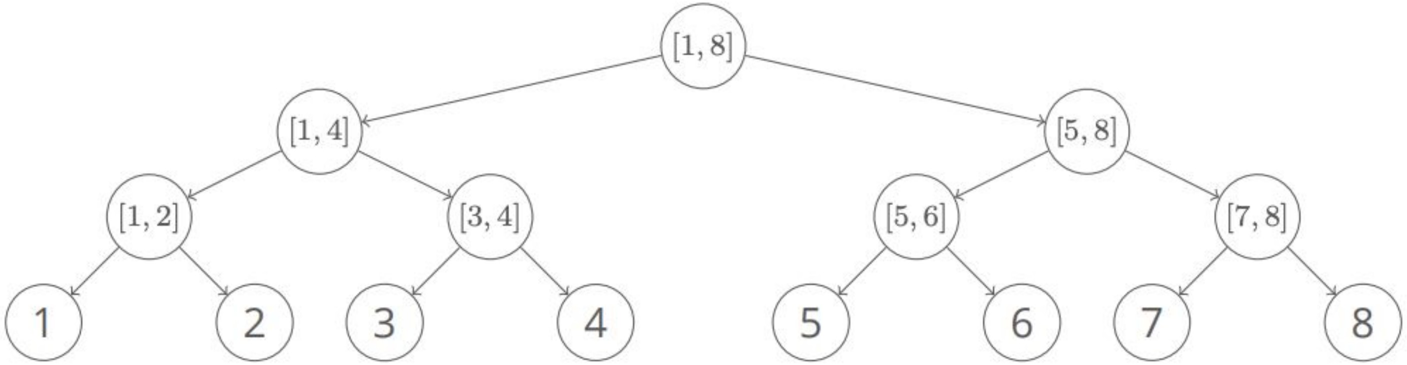
- Explore variants of segment trees
- From these variants, you will gain a deep understanding of core segment tree mechanics
- Understand segment tree as a framework, and learn to think structurally
  - What should a node represent?
  - How do operations flow?
  - How do we design merge operations?

Lecture Outline

Recap: Segment Tree Core

Underlying philosophy: Divide and Conquer

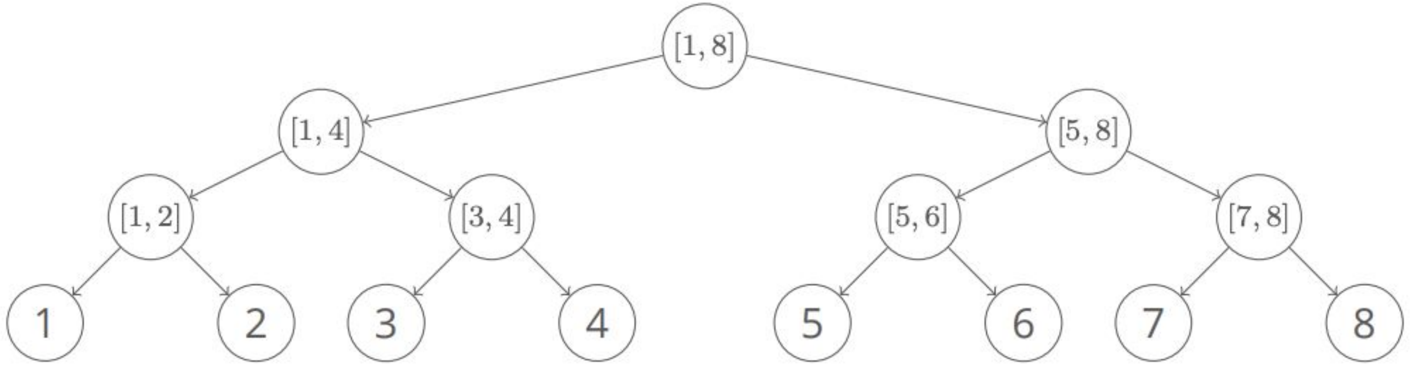
- A node for range [L, R] typically has children for [L, M] and [M+1, R].
- Key operations: **build**, **point update**, **range query**.
- What a node stores: Things like sum, min_val, max_val **for its range.**



Recap: Segment Tree Core

pull_up(node, L, R):

- combine results from children to update the parent.
- e.g. node.max = max(left.max, right.max)



Recap: Standard Lazy Propagation

Why do we need it? For efficient range updates.

How does it work? We use a **tag[]** array to store pending operations for nodes.

push_down(node, L, R)

1. It applies the parent's tag to its children's actual values.
2. It also applies the parent's tag to the children's tags (if tags can accumulate).
3. Then, it clears the parent's tag.

Recap: Standard Lazy Propagation

Why do we need it? For efficient range updates.
How does it work? We use a **tag[]** array to store pending operations for nodes.

When do we call **push_down**? Usually, right before we recurse to children if we need their **up-to-date actual values** for the current operation.

- A common example: Range Add, Range Sum.

We'll look at "Persistent Tags" later, which provides an alternative to range update without push_down.

Recap: Standard Lazy Propagation

Conceptual Question

- In standard lazy propagation (e.g., range add, range sum), when is it **absolutely necessary to call push_down** on a node during a query operation?
    - Can you describe a scenario in a query where push_down might not be strictly needed for a particular node, even if it has a tag?

Recap: Standard Lazy Propagation

Conceptual Question

- In standard lazy propagation (e.g., range add, range sum), when is it **absolutely necessary to call push_down** on a node during a query operation?

Answer

- We push_down whenever we need to **recur down to the children**, because we require the children to be perfectly up-to-date.
- **Subtle implementation-dependent point:**
  Is the stored tag already applied to the current node? Or does the stored tag indicated an operation yet-to-be-applied to the current node?
  - For the first way, you can omit push_down if you just return current node value.

Dynamic Node Allocation (動態開點)

Motivation

- Suppose N can go up to 1e9 or even 1e18.
- But, you only have a relatively small number of actual points or updates (e.g. Q is 1e5).
- The usual tree[4*N] array approach?
  - Not going to work, we use far too much memory!

Example Task

- A company has employee IDs from 1 to 1e9. Only 1e5 actual employees.
- We need to support: update salary for an ID, query max salary in an ID range.

Dynamic Node Allocation (動態開點)

Concept

- Let's only create tree nodes when we actually **need** them.
- Nodes are allocated "on the fly" as we traverse the tree for an update or query that encounters a path not yet created.

Dynamic Node Allocation - Implementation

How do we implement it?

- Nodes are often managed using pointers or indices of a pre-allocated pool of nodes. (or use pointers and new)

```
struct Node { T val; int l = 0; int r = 0; };
Node nodes[MAX_NODES]; int node_count = 0;
```

- We'll have a new_node() function that gets us the next available node from our pool.

Dynamic Node Allocation - Implementation

How do we implement it?

- In our update or query functions, when we try to go to a child:
- Typically, we use 0 to represent a null pointer, if the child index is 0, we know that the node isn't created yet, and we will do so.
- e.g., when we need to access the left child, we always do the following first.

```
if (nodes[id].l == 0) nodes[id].l = new_node();
```

Dynamic Node Allocation

- Complexity:
  - Space: O(Q log C), where C is the full coordinate range. This is a lot better than O(C) with normal segment tree.
  - Time: Still O(log C) per operation.

Dynamic Node Allocation

Conceptual Question

- Coordinate compression (using discretization) can be used as well. When might dynamic node allocation be preferable to coordinate compression?

Dynamic Node Allocation

Conceptual Question

- Coordinate compression (using discretization) can be used as well. When might dynamic node allocation be preferable to coordinate compression?

Answer

- When online query processing is needed. e.g. if you don’t know all the employees ID beforehand to do discretization.

Segment Tree on Value Range (值域線段樹)

- You may find this familiar and not special.
  - It should be, we are just giving it a fancier name
  - Giving names to concepts helps clarify them and helps your thinking.

Concept

- Instead of the covering array indices, what if the segment tree range represents the actual values in our data?
- A leaf node for a value v would store information about occurrences of v (like its count).
- An internal node for a value range [val_L, val_R] would store aggregated information for all values within that specific range.

Segment Tree on Value Range - Applications

Typical Applications

- Counting all elements in a specific value range within the array:
    - A query like query(root, V_MIN, V_MAX, query_val_L, query_val_R) would give us the count.
- Finding the Kth smallest element:
    - Support updating element and dynamic K query. If you don't have a BBST for this.
    - Each node stores the count of numbers in its value range.
    - To find the Kth smallest, we can do binary search on segment tree: if K <= count(left_child), we go left. Otherwise, we go right with K adjusted by count(left_child).

Often combined with Dynamic Node Allocation if the maximum possible value (V_MAX) is very large.

Segment Tree on Value Range - Applications

Additional Applications

- Many query problem that involves two ranges / dimensions can be changed to 1D problem on value range with time axis with sweep line trick (offline)
- **Counting points in 2D rectangle**
  - Given N points on a 2D plane. Answer Q query of different rectangle region: how many points are **within each region**.
    - Maintain segment tree on value range of y-axis (top to bottom). Each node [l, r] represents, how many points appears in (x, y) where l <= y <= r, and x <= T (T = current time)
  - We go from left to right increasing T, like a sweep line.
  - For each query (ql, qr, qu, qd), we only need to store down the results at time ql of query(qu, qd) and the same query at time qr, then subtract to get the answer.

Segment Tree on Value Range - Applications

Additional Applications

- Many query problem that involves two ranges / dimensions can be changed to 1D problem on value range with time axis with sweep line trick (offline)
- **Range MEX (Minimum excluded)**
    - Offline sort the query (ql, qr) with ascending qr.
    - Each leaf node maintains the **last time that value appeared until time T**.
        - Each range value node maintain **min** of two children
    - From index 1 to N:
        - update value node a[i] with time i.
        - handle **query qr = i,** binary search on segment tree to find leftmost node with **time < ql**

Segment Tree on Value Range - Applications

Additional Applications

- Some tasks however, require **persistent segment tree** for the 2nd dimension.
- **Range Kth smallest element** (DS(IV) lecture).
    - Query(l, r, k): find the Kth smallest element in array range [l, r]
    - Manage segment tree on value range, storing count of the element in the range. From index 1 to N sequentially, add the element one by one to the seg tree.
    - Binary search descent on the (l-1)-th tree and r-th tree simultaneously, using their differences to decide go left or right.

Segment Tree on Value Range

Conceptual Question

- Why **sometimes sweep line trick works**? e.g. counting points in 2D rectangle, range MEX. But sometimes we must rely on **persistent segment tree**? e.g. range Kth smallest element?

Segment Tree on Value Range

Conceptual Question

- Why **sometimes sweep line trick works**? e.g. counting points in 2D rectangle, range MEX. But sometimes we must rely on **persistent segment tree**? e.g. range Kth smallest element?

Answer

- **Range Mex**: It works happily because the 2nd dimension is always 0..X
- **Counting Point in 2D rect: Subtractable**. The answer [L, R] can be "subtract" or derived from info about [1, R] and [1, L-1].
- **Range Kth smallest**: Non-subtractable. Kth_min(1, R) and Kth_min(1, L-1) don't help you find Kth_min(L, R). You need the whole state.

Persistent Tags (標記永久化)

- This is sometimes called "Push-free segment tree".

Motivation

- It's an alternative way to handle range updates, different from standard lazy propagation's **push_down** step.
- It can sometimes simplify the update logic.
- A key feature is that it avoids modifying child nodes just to propagate tags. This can be cleaner or even beneficial in certain contexts (e.g. persistence)

Persistent Tags - Core Concept

Core Concept

- Tags are applied to a node and they **stay there**.
    - They are "permanent" for that node.
    - The tag on a node affects the entire range that node represents.
- When we do a query, we need to find the **"true" value**.
- To do this, we accumulate the effects of tags **from ALL ancestor nodes on the path from the root**, PLUS the tag on the current node itself.

Persistent Tags - Application

- Let’s demo this technique on the following task
  - Range max query(l, r): find max(a[l], a[l + 1], ..., a[r]).
  - Range apply max(l, r, x):
    - a[l] := max(a[l], x)
    - a[l + 1] := max(a[l + 1], x)
    - ...
    - a[r] := max(a[r], x)
- Normally in lazy propagation, you will maintain the lazy tags in nodes and push down during query or update.

Persistent Tags - Update

update(id, cur_L, cur_R, update_L, update_R, val)

- Typically, node.value stores an aggregate (like sum) based on its children, before any updates. Node.tag stores the updates.
- If the current node's range is fully contained within the update range
  - **We aggregate the tag directly: nodes[id].tag = max(nodes[id].tag, val)**
  - Then we return.
  - Importantly, nodes[id].value (the aggregate from children) is **NOT changed** here based on this new tag.
- Else (partial overlap / no overlap):
  - We recurse to the children.
  - **No pull_up**!

Persistent Tags - Query

query(id, cur_L, cur_R, query_L, query_R)

- If the current node's range is fully contained in the query range
    - Directly return **max(nodes[id].value, nodes[id].tag)**
- Else (partial overlap / no overlap):
    - For no overlap, we **return -inf**.
    - For partial overlap, we recurse on the children.
    - Suppose the answer is ans_L and ans_R.
    - We combine the results from the children **and the tag**.
        - Return max(ans_L, ans_R, nodes[id].tag)

Persistent Tags

How does this compare to standard lazy propagation?

- No explicit push_down function.
- The query logic itself handles incorporating all relevant tag effects from the path.
- Benefits:
  - Smaller constant
  - Easier to code

When **can** we use the persistent tags tricks then? When **can’t** we use it?

Persistent Tags

When **can** we use the persistent tags tricks then?

- The lazy info can be quickly aggregated together
  - Which is also the basis to lazy propagation for range updates
- **The update operations must be order-independent (commutative)**
  - Suppose the tags are X, Y, and Z. They are applied in this order.
  - X applied on range [1, 4]
  - Y applied on range [1, 8]
  - Z applied on range [2, 2]
  - When query for the value of position 2, with the query code, Z will be applied first, then X, then Y.
    - In order for this to work, Z(Y(X(value))) needs to be equal to Y(X(Z(value))).
    - You should be able to see why they need to be commutative because any order is possible

Persistent Tags

Conceptual Question

- Sometimes the operation is not inherently commutative. For example, range assign(l, r, v), and range query max(l, r). Could we still use persistent tags to handle this?

Persistent Tags

Conceptual Question

- Sometimes the operation is not inherently commutative. For example, range assign(l, r, v), and range query max(l, r). Could we still use persistent tags to handle this?

Answer

- If there are ways to **transform it into commutative,** then yes.
- For this range assign operations, we can maintain the **timestamp** of the assign operations in the tags too. We can then perform the query by ONLY applying the operations if the timestamp is later, and update the time.

Segment Tree D&C (線段樹分治)

- You can also think this as Segment Tree on Time (**時域線段樹**), because each node is not representing position, not representing value, but representing a Time interval.
- It aggregates temporal information rather than position or value.

Motivation

- A classic problem this tackle is called **Dynamic Connectivity**
  - Add(u, v): add edge (u, v) to the graph
  - Delete(u, v): delete an existing edge (u, v) on the graph
  - Query(u, v): check whether if two nodes u and v are connected

Segment Tree D&C (線段樹分治)

Motivation

- A classic problem this tackle is called **Dynamic Connectivity**
  - Add(u, v): add edge (u, v) to the graph
  - Delete(u, v): delete an existing edge (u, v) on the graph
  - Query(u, v): check whether if two nodes u and v are connected

We can think of each edge on the graph being **alive** for a certain lifespan, say from **time T_start to T_end**.

- You need to answer queries about the graph specific points in time, considering only the edge currently alive.
- We maintain the active edge with a segment tree over time.

Segment Tree D&C - Core Concept

Concept

- We need to be able to offline process the edge changes. If online is needed, you will need advanced DS like Link-Cut Tree.

- The Timeline: We build a segment tree that covers the total time range of our problem, say from [1, T].
    - A leaf node in this segment tree represents a specific point in time.
    - An internal node represents an interval of time.

Segment Tree D&C - Core Concept

Concept

- Adding Operations
  - An operation (e.g., an edge existing from T_start to T_end) is "added" to O(log Q_time) nodes in this time-segment tree.
  - Each segment tree node u will store a list **(e.g., vector<Operation>)** of all operations that are active **throughout its entire time interval [L, R]**.
    - You can think of them as “persistent tags”.
    - As you can see, doing **push down is infeasible**, because the Operations often cannot be compressed and it is too costly to propagate.

Segment Tree D&C - The Algorithm

Algorithm

- The Time-Segment Tree only store the lifetime of each edge. How do we get the state of the graph at a specific timestamp?
    - Query for each leaf, go down from the root like Persistent Tag?
    - If all edge live from 1 to T (all tags are located at the root), this would **apply all tags per query**, costing O(MQ log T) time.

Segment Tree D&C - The Algorithm

Algorithm

- We can maintain the graph by moving forward in time step by step, applying the difference only.
  - We will answer the query when we reach the timestamp.
- How? Perform a DFS on this Time-Segment Tree
  - **Reach a new node?** Add all the newly "alive" edges
  - **Reach a leaf node?** Arrive at a particular timestamp
  - **Leave a node?** Remove all the edge added from this node
- We often need to use alongside a data structure like DSU that supports rollback.

Segment Tree D&C - The Algorithm

Algorithm simple pseudo code of solve_dfs(id, time_L, time_R):

1. For each **Operation op** that was placed in seg_tree[id].operations:
   - Apply this op to our DSU (e.g. dsu.add(op.u, op.v)).
   - Crucially, the **DSU must record the changes it made so we can undo them** later.
2. If this is a leaf node (time_L == time_R):
   - We answer any queries scheduled for this time_L using the DSU's current state.
3. Else (if it's an internal node):
   - Recursively call solve_dfs(left_child, time_L, mid), call solve_dfs(right_child, mid + 1, time_R).
4. Rollback
   - Now, we undo the DSU changes that were made in step 2, usually in the reverse order they were applied. This restores the DSU to the state it was in before we processed this node's operations.

Segment Tree D&C - DSU with Rollback

DSU with Rollback

- Essentially, the DSU must be able to revert operations.
- How is this usually done?
  - One way is to use union by size/rank but **without path compression** during the unite operations whose effects we want to roll back. Path compression makes rollback tricky.
  - We explicitly store what changed, usually in a stack.
    - e.g., for add(u, v), we usually end with the line **dsu[head_u] = head_v**.
    - We can store **(head_u, dsu[head_u])** before this line, so we can revert this change.
    - (this is a simplified version, you will need to log the size/rank updates too)
- The add operation logs these changes, and the rollback step uses these logs to revert.

Segment Tree D&C - Complexity

Complexity

- Roughly O(M * log T * T_DSU_rollback + Q)
  - **M**: Number of operations.
  - **Q**: Number of queries.
  - **T_DSU_rollback**: Cost of a DSU operation that supports rollback (e.g., log N for a DSU without path compression during add).

Segment Tree D&C

Conceptual Question

- Why do we need the DSU to support a "rollback" operation? What would go wrong if we used a standard DSU with path compression?

Segment Tree D&C

Conceptual Question

- Why do we need the DSU to support a "rollback" operation? What would go wrong if we used a standard DSU with path compression?

Answer

- DSU don’t easily support delete edges, only support adding. That’s why we need to use a segment tree to maintain the lifetimes of the edges. Rollback is a easier alternative to support.
- Path compression optimizes by flattening trees, it erases the history of how the nodes are connected, just keeps the crucial info, the tree head.

Segment Tree D&C

A remark on data structure with rollback:

- Actually, a lot of data structure can support rollback, if you log the state changes in a stack-like structure.
- A key to determine what operation can support rollback is: whether the operation is cheap, or just amortized cheap.
    - Amortized time complexity does not work well with rollback, because in the worst case the expensive operations would be performed multiple times.

Building Skyline Trick (樓房重建) [Counting Prefix Maximum]

Next, we will talk about a trick, which got popularized by the following problem.

Problem (Luogu P4198 樓房重建):

- We have N positions. At position i, a building can have height H_i. Its slope is H_i/i.
- Operations:
  - Update the height H_x of the building at position x.
  - Query: How many buildings are **visible from the origin (0, 0) if we look rightwards**?
  - (A building is visible if its slope is greater than the slopes of all buildings to its left).
- The trick is interesting because it shows that the pull up function can be a lot more complicated than simple arithmetic functions.

Building Skyline Trick - Core Concept

Core Concept

- Our segment tree will cover the indices 1 to N.
- What does a node u (representing range [L, R]) need to store?

1. **max:** The maximum slope H_j/j for any building j in its range [L,R].
2. **ans:** Visible building count in [L, R]
    - **UNDER THE ASSUMPTION** that the problem only consists of the segment [L, R].
    - (meaning that, this value **omits the influence** of all the buildings in [1, L - 1])

Building Skyline Trick - Core Concept

Core Concept

- Our segment tree will cover the indices 1 to N.
- What does a node u (representing range [L, R]) need to store?

1. **max:** The maximum slope H_j/j for any building j in its range [L,R].
2. **ans:** Visible building count in [L, R]

- Only **point update** is needed in the task, so let's figure out how to maintain these info during the path upward (and assume all other nodes have maintained their info correctly)
    - max is quite easy to maintain, so we'll focus on ans.

Building Skyline Trick - Pull up

The magic happens in **pull_up(id, lc_id, rc_id)** for ans:

- First, let's directly use answer to its left child.
    - **node[id] = nodes[lc_id].ans**
- Then, we would need to get the **contribution to the answer** from [M+1, R].
    - We could **not use nodes[rc_id].ans directly** because that assumes that the leftmost building in the range is unobstructed.

Building Skyline Trick - Pull up

The magic happens in **pull_up(id, lc_id, rc_id)** for ans:

- First, let’s directly use answer to its left child.
  - **node[id] = nodes[lc_id].ans**
- Then, we would need to get the **contribution to the answer** from [M+1, R].
- Instead, we find that contribution by a special query function:
  - **calc(rc_idx, M + 1, R, nodes[lc_id].max)**
  - This query asks: *"How many buildings are visible in the right child's range, **given** that the maximum slope we encountered from the left child's range was nodes[lc_idx].max?"*

- Our next target is to find a efficient way to implement this function.

Building Skyline Trick - Calc function

Let's look closer at **calc(id, node_L, node_R, prev_max)**:

- This function calculates how many buildings in current range are visible, given that the maximum slope seen **just before** this node_L was prev_max.
- Base Case (a leaf representing a single building)
    - If slope[id] > prev_max, it's visible, so return 1. Else, return 0.

Building Skyline Trick - Calc function

- Recursive Step for an internal node
  - Suppose lc = left_child, rc = right child.
- **Case 1: If nodes[lc].max <= prev_max**:
  - This means all buildings in the left child's range are "hidden" by the prev_max we came in with.
  - So, any visible buildings must come entirely from the right child's range.
  - We recurse: **return calc(rc, ..., prev_max)**.

Building Skyline Trick - Calc function

- **Case 2: else (nodes[lc].max > prev_max):**
    - This means some buildings in the left child **might** be visible (they are steeper than prev_max_slope).
    - The number of visible buildings from the left part is what we get by calling:
        - **visible_from_left = calc(lc, ..., prev_max).**
    - Now for the right part
        - The term **(nodes[id].ans - nodes[lc].ans)** actually gives us the number of buildings visible in the right child
        - Why?
            - The prev_max has no effect on the right child range, so we can use previously calculated results

Building Skyline Trick - Calc function

```
calc(id, node_L, node_R, prev_max)
    if (id is a leaf node):
        return (nodes[id].max > prev_max ? 1 : 0)
    else:
        if (nodes[lc].max <= prev_max):
            return calc(id * 2 + 1, …, prev_max)
        else:
            return calc(id * 2, …, prev_max) +
                   (nodes[id].ans - nodes[lc].ans)
```

- Observe that this function is O(log N) because it always recurses into at most 1 child for the recursive part.

Building Skyline Trick - Complexity

What's the complexity?

- A point update (changing one building's height) becomes O(log^2 N).
  - The tree traversal for the update is O(log N).
  - Each pull_up along that path involves calling **calc**, which itself is an O(log N) traversal.
- Querying the total number of visible buildings from the start is just reading nodes[root].ans, which is O(1).

- You can also do range query in this task, utilizing **calc(..., -inf)**. The query must be done left to right, and passing previous slope max to next calc() params.

Building Skyline Trick - Takeaway

The Key Takeaway here:

- Segment tree nodes can store quite complex, problem-specific information.
- The pull_up (or merge) operation doesn't have to be simple arithmetic.
  - It can be a **recursive, query-like function itself**, operating on the children.
  - This structural idea unlocks a whole new class of problems.

Conclusion - What We've Seen Today

Let's summarize the Segment Tree we've explored:

- Dynamic Node Allocation (動態開點): for huge, sparse ranges.
- Value Range Trees (值域線段樹): for queries based on values.
- Persistent Tags (標記永久化): an alternative to standard lazy propagation.
- Segment Tree D&C (線段樹分治): for offline processing of operations with lifespans.
- Building Skyline Trick (樓房重建): for complex pull_up behaviour

Conclusion - The True Nature of Segment Trees

So, what's the big picture? What truly is the nature of a segment tree?

- It's not just one fixed data structure; it's a flexible framework fundamentally built on Divide and Conquer.
- Its real power comes from how **you** define these three things for your specific problem:
    - 1. **What information** each node in the tree stores. (This is like the "state" of a subproblem).
    - 2. **How to combine** (pull_up) information from child nodes to the parent. (This is how you solve a larger problem from its subproblem solutions).
    - 3. **How updates** affect the nodes and how they propagate. (This is how you modify the state).

The real art and skill lie in figuring out how to map your problem's state and transitions onto this versatile framework.

Deeper Dive: The Algebraic Foundations of Segment Trees

Warning!

The next parts are for building a deeper understanding of segment trees.

- It requires some understanding of Abstract Algebra
- In simpler words, “if some particular rules are satisfied, then it can be used in some applications.

You can skip all of the whole “Deeper Dive” section if you are just getting started with Segment Tree, and come back later.

Deeper Dive: The Algebraic Foundations of Segment Trees

So far, we've seen many powerful segment tree variants.

- But **why** do they work so reliably? What are the fundamental properties that allow us to **decompose problems and combine solutions** this way?
- The generality of segment trees comes from underlying algebraic structures.
    - Understanding these helps ensure correctness and allows us to design segment trees for new, complex problems.

- Let's explore two key areas:

  1. The information in nodes and how it's combined (pull_up).
  2. The properties of lazy tags and their application.

Deeper Dive: Node Information & The pull_up Operation

The pull_up Operation

- The core of a segment tree is combining information from children to represent a larger range.
- The data stored in nodes (e.g., sum, min, a matrix) and the pull_up operation often form an algebraic structure called a **Monoid 幺半群**.

Deeper Dive: Monoid

What's a Monoid?

- A Monoid (S, •, id) has three parts:

  1. A Set S: This is simply the **type** of data your segment tree nodes will store.

  2. A Binary Operation •: This is your pull_up or combine function. It takes two elements from S (from two children) and produces another element for S (for the parent).

  3. An Identity Element id: A special element in S.
- Let's look at the properties of • and id.

Deeper Dive: Property 1 - Associativity of pull_up

Key Property for pull_up: **Associativity**

- The binary operation • (our pull_up function) **must** be **associative**.
- This means for any three pieces of data a, b, c that our nodes can store: (a • b) • c must be equal to a • (b • c).

Deeper Dive: Property 1 - Associativity of pull_up

Why is associativity absolutely crucial?

- A segment tree node combines data from its children. These children might represent ranges of very different sizes.
- The way information is grouped as we go up the tree can vary. For example, ([1, 1]•[2, 2]) • [3, 4] vs [1, 1] • ([2, 2]•([3, 3]•[4, 4])).

- Associativity guarantees that **no matter how the ranges are subdivided and recombined** by the tree structure, the final result for any given range is consistent.

Deeper Dive: Property 2 - The Identity Element

Key Property for pull_up: Identity Element

- A Monoid also needs an **identity element**, let's call it id, within our set S.
- This id has the property that for any data a in S:
  - a • id = a
  - id • a = a
  - (Combining a with id leaves a unchanged.)

What is this identity element in practice?

- It's the value a query should return for an **empty range**.
- Or, it's the sensible default/initial value when you start combining things.

*Strictly speaking, a segment tree **do not necessarily** need an identity element (there are implementation to not require it) -> semi-group is important

Deeper Dive: Property 2 - The Identity Element

Key Property for pull_up: Identity Element

Examples:

- For Range Sum: **id = 0** (sum + 0 = sum).
- For Range Minimum Query: **id = +infinity** (min(val, +inf) = val).
- For Range Maximum Query: **id = -infinity** (max(val, -inf) = val).
- For Building Skyline problem: **id = (max = -infinity, ans = 0)**.

- A query that doesn't find any full segment tree nodes within its range should effectively start with this identity.

Deeper Dive: Summarizing the Monoid for pull_up

- So, for the pull_up part of our segment tree to be well-behaved, the **(data_type, combine_function, identity_value) should form a Monoid**.

- This **ensures** that no matter how the tree structure partitions a query range into maximal segments, **combining their stored information will yield the same, correct result**.
- This is a fundamental reason why segment trees are so general! If you can define these for your problem, a segment tree might be applicable.

Deeper Dive: Now Let's Talk About Tags (Lazy Propagation)

Focus: Lazy Propagation Tags & Their Properties

- When we use lazy propagation, we introduce tags. Let's think about their properties.

- Let **Data** be the set of values our nodes can hold (our Monoid set S).
- Let **TagType** be the set of operations/tags we can apply (e.g., "add X", "set to Y", "apply chmin with V").

Deeper Dive: Now Let's Talk About Tags (Lazy Propagation)

Focus: Lazy Propagation Tags & Their Properties

- We need to define two key functions for tags:

1. apply_op(data_value, operation_tag): Takes a current data value and a tag, and returns the new data value after the tag is applied.

- Example: apply_op(current_sum, "add 5") would update current_sum.

2. compose_tags(existing_tag, new_tag): If a node already has existing_tag and a new_tag arrives, this function combines them into a single tag that represents their sequential effect.

- Example: compose_tags("add 5", "add 3") results in an "add 8" tag.

Deeper Dive: Tag Property 1 - Distributivity

Tag Property: Distributivity

- How should a tag interact with our pull_up operation?

When we push a tag T down from a parent to its children L and R:

- We expect

  apply_op(combine(data_L, data_R) , T)

- to be equivalent (or lead to the same result) as

  combine(apply_op(data_L, T) , apply_op(data_R, T))

- Essentially, applying a tag to a combined result should be the same as applying the tag to the individual parts and then combining them.

Deeper Dive: Tag Property 1 - Distributivity

Tag Property: Distributivity

- Example (Range Add Tag delta, Range Sum Data):
  (sum_L + sum_R) + delta * (len_L + len_R) (applying tag to combined sum)
- should be consistent with
  (sum_L + delta * len_L) + (sum_R + delta * len_R) (applying to parts, then combining).
- This property ensures that applying a tag at a higher level and then pushing it down maintains correctness.

Deeper Dive: Tag Property 2 - Associativity of Tag Composition

Tag Property: Associativity of compose_tags

- Our compose_tags(tag_A, tag_B) function, which defines how two tags combine if tag_B is applied after tag_A, should itself be associative.
- If we have three tags t1, t2, t3 arriving sequentially:

  compose_tags(compose_tags(t1, t2) , t3)
- should be equivalent to

  compose_tags(t1, compose_tags(t2, t3)).
- This is important because multiple updates might affect a range, leading to multiple tags being aggregated on a node before they are eventually pushed down. Associativity ensures the final composed tag is consistent.

Deeper Dive: Tag Property 2 - Associativity of Tag Composition

Tag Property: Associativity of compose_tags

- Example:
  ((t_add5 then t_add3) then t_add2)
- is same as
  (t_add5 then (t_add3 then t_add2))
- Both become t_add10.

Deeper Dive: Tag Property 3 - Commutativity of Tags (Optional)

Tag Property: Commutativity of compose_tags (Sometimes true, very helpful)

- Commutativity means compose_tags(t1, t2) is the same as compose_tags(t2, t1).
- If tags are commutative, the order in which multiple update operations (that haven't been pushed down yet) hit a node doesn't change the final composed tag.

Deeper Dive: Tag Property 3 - Commutativity of Tags (Optional)

Tag Property: Commutativity of compose_tags (Sometimes true, very helpful)

- **Commutative**:
    - "Range Add X" and "Range Add Y" tags are commutative.
      (Add X then Add Y = Add Y then Add X).
- **Not Commutative**:
    - "Range Add X" and "Range Set to Y" are generally **not** commutative.
    - ("Add 5, then Set to 10" results in 10).
    - ("Set to 10, then Add 5" results in 15).
- If your tags are not commutative, your compose_tags function must correctly implement the desired order of operations (e.g., a "set" operation usually overwrites any prior "add" operations).

Deeper Dive: Why Do These Algebraic Properties Matter?

So, why did we go through all this theory about Monoids, associativity, etc.?

1. **Correctness**
   - pull_up forms a monoid means the segment tree will work correctly for any query range.
   - How tags distribute over the data combination is crucial for lazy propagation to be correct
2. **Framework for Generalization**
   - This algebraic view shows the segment tree isn't just for sums and mins.
   - If you can define your problem's data, an associative combine operation with an identity, and how update operations (tags) interact with your data and each other, you can likely use segment tree.

Deeper Dive: Why Do These Algebraic Properties Matter?

So, why did we go through all this theory about Monoids, associativity, etc.?

3. **Designing Advanced Segment Trees**

- For complex problems like Segment Tree Beats or Building Skyline example, the whole problem as a whole is very complicated.
- Thinking about these underlying abstract properties helps you ease the mental burden.
- You just need to use the correct framework and design the corresponding
  - **Data**
  - **Tags**

Practice Tasks (Warning: hard)

- T192 - Colorful Strip
  https://judge.hkoi.org/task/T192
- T232 - Challenge of Hanoi
  https://judge.hkoi.org/task/T232
- PC2325 - Walk of Love
  https://judge.hkoi.org/task/PC2325

- These are all raw Segment Tree / data structure tasks.
- Hardest tasks usually involves Segment Tree as only one part, requiring you to be skillful in multiple topics (e.g. T251 Moving Marvellous Marbles).

Extra Materials

More lectures on interesting Segment Tree usages:

Data Structure (IV) 2025 - Persistent Segment Tree
https://assets.hkoi.org/training2025/ds-iv.pdf

Training Camp 2023 - Segment Tree in Graph
https://assets.hkoi.org/training2023/graph-construction.pdf

Training Camp 2024 - Potential Segment Tree / Segment Tree Beats
https://assets.hkoi.org/training2024/ds-potential.pdf

References

Alex Wei - Advanced Segment Tree Part 1 (Simplified Chinese)
https://www.cnblogs.com/alex-wei/p/18356369/SegmentTreePart1

Pink Rabbit - Segment Tree and Prefix Maximum (Simplified Chinese)
https://www.cnblogs.com/PinkRabbit/p/Segment-Tree-and-Prefix-Maximums.html