Range Tree Applications and Tricks for DS Problems

Max Godfrey 10/04/2022

Pre-Lecture Version

Table of Contents

- Range Trees with Lazy Propagation
- Lazy Create Range Trees
- Optimising DP Recurrences with Range Queries
- Sweeplines
- Maximum Subarray Sum Range Trees

Motivation: Range Update, Range Query

- CSES Range Updates and Sums: https://cses.fi/problemset/task/1735
- Create a data structure supporting the following operations on an array of size N:
  - $\text{Set}(i, j, val)$: Set all elements of the array in $[i, j]$ to val
  - $Add(i, j, val)$: Add val to each element of the array in $[i, j]$
  - $Sum(i, j)$: Return the sum of the array interval $[i, j]$
- You will be asked to handle Q operations

Naïve Range Tree Solution

- A standard Range Tree (Point Update, Range Query) can handle Sum in $O(\log_2 N)$ with ease
- To deal with Set and Add, we *could* run a point update for every element in the range each operation concerns
  - $O(N \log_2 N)$ – Not good enough
  - For Set, it would look something like this:

```
void range_set(int ul, int ur, ll val, int i = 1, int l = 1, int r = N) {
  if (l > ur || r < ul) {
    return;
  }
  if (l == r) {
    st[i] = val;
    return;
  }
  int mid = (l + r) / 2;
  range_set(ul, ur, i * 2, l, mid);
  range_set(ul, ur, i * 2 + 1, mid + 1, r);
  st[i] = st[i * 2] + st[i * 2 + 1];
}
```

Optimising the Range Set routine

- Observe that our current set routine goes to every leaf of the Range Tree representing each index of the array we are supposed to modify
- Does it need to do this? That depends…
  - If we are going to visit those nodes later as part of a future query, then yes
    - The chances of this happening are very low if this operation is called multiple times
  - Most (if not all) of the time, we don't *need* to visit all the leaves – or even get close

- Slight problem:
  - We don't have the foresight to know what operations we will be asked to handle later
    - Or, more to the point, we can't afford it
  - To get around this, what if we marked nodes whose subarrays need setting, but only apply the set operation when we reach it again for another purpose (ie. Add or Sum)?

Why is it called *Lazy* Propagation?

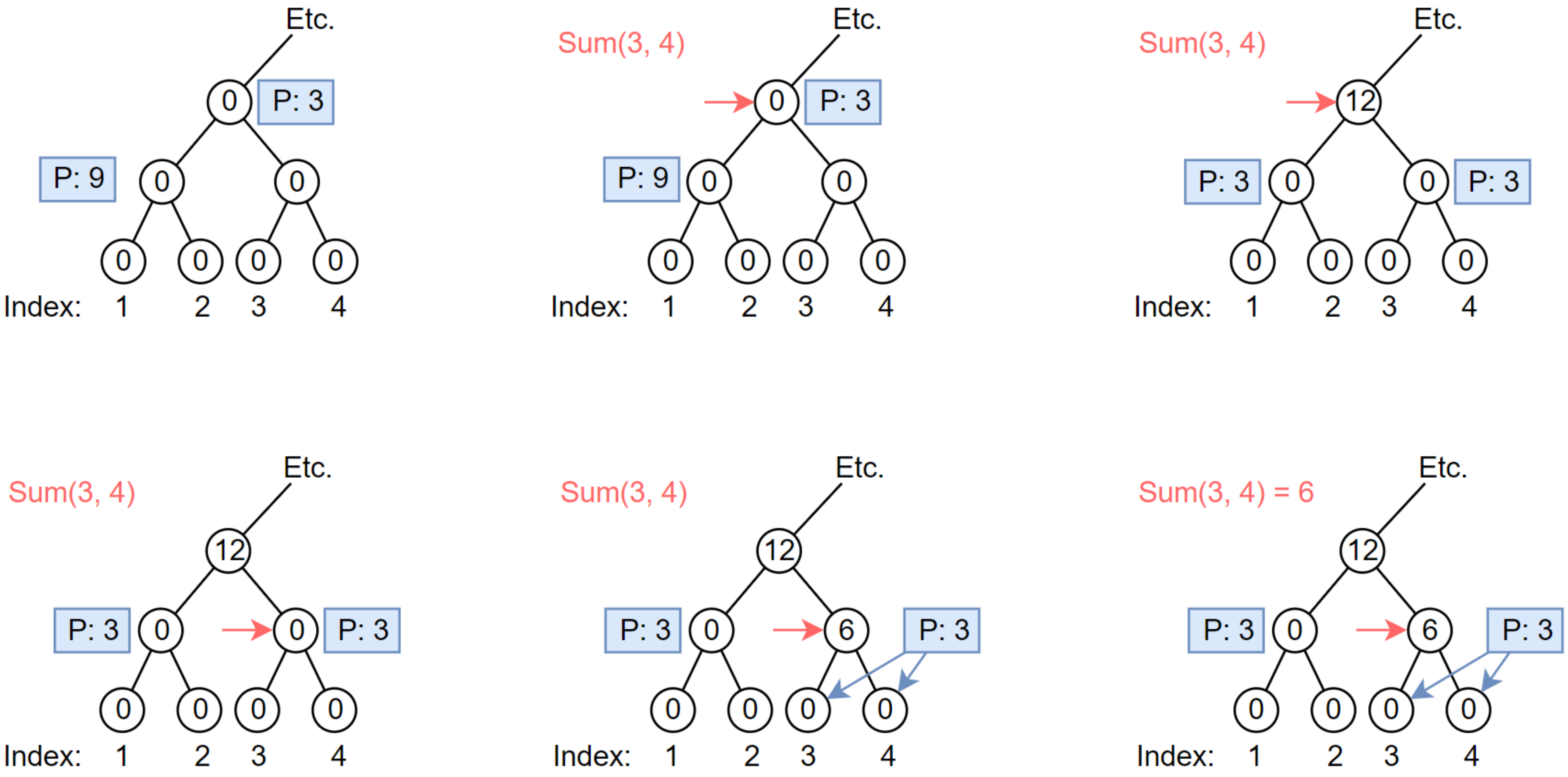
- For each node, we maintain tags indicating that we have some operation(s) pending on the *entire* subarray it covers – we only apply them when we need to traverse the subtree!
  - In this case, for each node we maintain two tags $lazy\_add$ and $lazy\_set$
  - The order that the operations are applied matters!!!
- An example for a DS which only supports Sum and Set:

```cpp
void apply(int i, int l, int r) {
  if (lazy_set[i] != NOSET) {
    st[i] = lazy_set[i] * int64_t(r - l + 1);
    if (l != r) {  // True if we aren't at a leaf
      lazy_set[i * 2] = lazy_set[i];
      lazy_set[i * 2 + 1] = lazy_set[i];
    }
    lazy_set[i] = NOSET;
  }
}
```

```cpp
void range_set(int ul, int ur, ll val, int i = 1, int l = 1, int r = N) {
  apply(i, l, r);
  if (l >= ul && r <= ur) {
    lazy_set[i] = val;
    apply(i, l, r);
    return;
  }
  // Proceed as normal...
}
```

```cpp
ll query(int ql, int qr, int i = 1, int l = 1, int r = N) {
  apply(i, l, r);
  // Proceed as normal...
}
```

Example of Laziness



Combining Operations

- When pushing tags down the tree in $apply$, we need to consider a few things:
  - Pushing a $lazy\_set$ tag onto a node should override all operations it has pending
    - We need to ensure that we get rid of everything pending (ie. Child's $lazy\_add = 0$)
  - Pushing a $lazy\_add$ tag onto a node should add to whatever $lazy\_add$ operations are pending (if any)

- When modifying the value of the current node in $apply$, we need to first apply the pending $lazy\_set$, and then the pending $lazy\_add$ (if applicable)
  - This is because of the way we set things up above: pushing a pending range set will remove all other operations, while a pushing a pending range addition is cumulative

- $Sum(i, j)$ is trivial: just remember to call $apply$ before processing each node

Complexity

- With any lazy operation:
  - Updates stop when the node they are at covers an entire subarray which the update is supposed to be applied to: $O(\log_2 N)$
- The routine $apply$ is only called as part of:
  - Other lazy operations in $O(\log_2 N)$, or query operations in $O(\log_2 N)$
  - Strictly speaking, each update doesn't properly get applied to the array:
    - At most points in time, the Range Tree will store a lot of data which will be overridden by things further up the tree
    - Many updates won't even be applied, because they are either overridden by other updates, or the subarrays they concern are never queried
- Trivially, queries happen in $O(\log_2 N)$

Range Trees with Lazy Propagation Recap

- Great for Range Update, Range Query
  - Most common update operations are Range Set and Range Add

- Range Updates stop recursing when the current update is intended to be applied to the current node’s entire subarray it covers
  - The rest is ‘inadvertently’ handled by other recursive utilities!

- By propagating lazily, we achieve the same complexities as a Range Update, Point Query Range Tree

Lazy Create Range Trees

- Some problems require range queries on a huge area
    - Commonly $N \leq 10^9$, but the upper bound is realistically anything greater than 10^7
    - Building a full Range Tree will use way too much memory

- If the range is massive, it is common for the task to require some kind of range update
    - Lazy Propagation!

Before we continue…

- Can’t we just coordinate compress and then build a full Range Tree over the compressed array?
  - Not most of the time
  - Coordinate compression guarantees us monotonicity and a good space complexity, but in this context, that is about it
    - The gaps between the compressed elements are variable, and this can be problematic where it comes to querying things like range sums
  - Even if the update/query functions work nicely regardless of inconsistent gap sizes, the implementation is terrible – what I am about to demonstrate is a lot nicer…

The Critical Observation

- Imagine that it was possible to build a complete segment tree over the range we are concerned with:
  - After processing all of our Range Updates and Range Queries (using lazy propagation if necessary) there will be a massive number of nodes, particularly near the bottom of the tree which were untouched
  - For instance, the number of leaves we visit across all queries is at most $2Q$
  - The number of nodes which are used in total is $O(Q \log_2 N)$

- Why create so many nodes if we aren't going to use most of them?
- What if we only created nodes as we needed them?

Creating Nodes Lazily

- The overall structure of the routine for a Lazy Create Range Tree operation looks something like this:

```
Query(ql, qr):
  if l > qr || r < ql: return
  if l >= ql && r < ql: return value
  CreateChildren()
  return Merge(leftchild.Query(ql, qr), rightchild.Query(ql, qr))
```

- When children are created, the value stored in the current node must be passed to them somehow:
  - Eg. for a tree designed to handle RMQ, if the current node has value v, we give both of its children a value of v as well

Example: Range Add (Update), Range Sum (Query)

```cpp
struct ST {
  int l, r, val;
  ST *nl, *nr;
  bool has_kids = false;

  int lazy_add = 0;

  ST(int _l, int _r) {
    l = _l;
    r = _r;
    val = 0;
  }

  void apply() {
    if (lazy_add != 0) {
      v += (r - l + 1) * lazy_add;
      if (has_kids) {
        nl->lazy_add += lazy_add;
        nr->lazy_add += lazy_add;
      }
      lazy_add = 0;
    }
  }
```

```cpp
  void upd(int ul, int ur, int uv) {
    apply();
    if (l > ur || r < ul) {
      return;
    }
    if (l >= ul && r <= ur) {
      lazy_add = uv;
      apply();
      return;
    }
    make_children();
    nl->upd(ul, ur, uv);
    nr->upd(ul, ur, uv);
    v = nl->val + nr->val;
  }

  int qry(int ql, int qr) {
    apply();
    if (l > qr || r < ql) {
      return 0;
    }
    if (l >= ql && r <= qr) {
      return val;
    }
    make_children();
    return nl->qry(ql, qr) + nr->qry(ql, qr);
  }
```

```cpp
  inline void make_children() {
    if (!has_kids && l != r) {
      int mid = (l + r) / 2;
      int single = val / (r - l + 1);
      nl = new ST(l, mid);
      nl->val = single * (mid - l + 1);
      nr = new ST(mid + 1, r);
      nr->val = single * (r - mid);
      has_kids = true;
    }
  }
};
```

Lazy Create Range Trees Recap

- Great for maintaining data pertaining to a huge ‘space’
- Only create child nodes when work on their subtrees is required
- Can be combined with all other Range Tree methods (Lazy Prop, Persistence, Multiple Dimensions, etc.)
- Behaves exactly like a Range Tree: all operations are $O(\log_2 N)$

DP + Range Tree

- There are some DP problems in which a state is constructed from one of K different states, and a naïve DP would take $O(NK)$
  - K may be constant, or variable in which case the complexity is $O(N * \max K)$

- Example: *Cannons* (Orac):
  - Sam (a Stuntman) lines up N cannons in a row, the i-th of which has:
    - Distance D_i, meaning that it can fire him up to D_i cannons further to the right
    - Excitement Value E_i, indicating how much the crowd likes the cannon (can be negative)
  - Sam starts in the leftmost cannon, and wishes to be fired to position $N + 1$ (ie. One cannon's length to the right of the rightmost cannon)
  - Given that Sam fires himself from cannon to cannon, what is the maximum sum of Excitement Values he can obtain subject to the constraints above?

Building up to the *Cannons* Full Solution

- Redacted

Tell-tale Signs of a DP + Range Tree Problem

- The recurrence is defined by some number of other states preceding it
  - The states' values do not change (unlike CHT problems, for instance)
- Constructing the state requires some kind of range query on the data

- $O(N \log_2 N)$ is the slowest algorithm that passes
  - Disclaimer: just because it passes doesn't mean it's the intended solution

Sweeplines

- Suboptimal solutions to plane problems can often be optimised by ‘sweeping’ a line parallel to a side of the plane across the data
  - We maintain information which can modify the solution in a Data Structure representing the line (eg. A Set, Range Tree, etc.)
  - “Imagine a line sweeping through the problem and see what happens” – Quang

- Example problem: CSES *Intersection Points* (https://cses.fi/problemset/task/1740)
  - Given N horizontal and vertical lines (specified by start and end coordinates), count the number of line intersections
  - No parallel line segments intersect, and no endpoint of a line segment is an intersection point
  - $1 \leq N \leq 10^5$
  - $-10^6 \leq All\ Coords \leq 10^6$
  - Time Limit: 1 second

Intersection Points

- Naïve Solution:
  - Iterate through all pairs of horizontal and vertical lines, and count the number of intersections
  - $O(N^2)$

Intersection Points

- Better Solution:
  - Withheld before lecture

Maximum Sum Subarray Range Trees

- A non-trivial application of Range Trees is that they can be used to compute the sum of the Maximum Sum Subarray in an array
- To do this, we store four values for each node:
  - $Total$: The total sum of the range covered by the node
  - $Prefix$: The maximum prefix sum in the range covered by the node
  - $Suffix$: The maximum suffix sum in the range covered by the node
  - MSS: The sum of the Maximum Sum Subarray in the range

Defining the Recurrence

- Base case – at a leaf node with value v:
  - $Total = v$
  - $Prefix = v$
  - $Suffix = v$
  - $MSS = v$

- Recurrence – when creating a node from two others ($Left$ and $Right$):
  - Redacted

Defining the Recurrence

- Base case – at a node with value v:
  - $Total = v$
  - $Prefix = v$
  - $Suffix = v$
  - $MSS = v$

- Recurrence – when creating a node from two others ($Left$ and $Right$):
  - Redacted

Defining the Recurrence

- Base case – at a node with value v:
  - $Total = v$
  - $Prefix = v$
  - $Suffix = v$
  - $MSS = v$

- Recurrence – when creating a node from two others ($Left$ and $Right$):
  - Redacted

The Segtree

- With this recurrence:
  - We can update an array value in $O(\log_2 N)$
  - We can query a subarray to find its Max Subarray Sum in $O(\log_2 N)$

- We defined a commutative operation, and built a Range Tree over an array using it
  - Behold the power of the Range Tree!

Your Problemset

- Key Problems
  - Lazy Updates (Lazy Propagation)
  - InstaHarvest (Lazy Create)
  - Cannons (DP + Range Tree)
  - Mapping Neptune (Sweep + Range Tree)
  - Panorama (Max Sum Subarray Range Tree)
- Additional Problems (do in any order)
  - Stargazing
  - Danilee Kelly Visits Greece
  - Maximum Non-Adjacent Subsequence Sum
  - Pam-Can Retires
  - Mountain
  - Lowering Standards
  - Sails

Further Reading – Lazy Create

- A blog I wrote in Y10: https://maxgodfrey2004.github.io/competitive-programming/2019/10/11/lazy-create-segment-trees.html
- USACO Guide: https://usaco.guide/plat/sparse-segtree?lang=cpp