Lecture 6: Range Queries

Objectives of this lecture

- Answering prefix sum / max queries under entry updates
- Using tree queries to speed up dynamic programs
- Discuss answering interval max queries, as well as implementation.

1 Philosophy

The goal of this lecture is to turn tree based dynamic programming into data structures. and use such data strictures to speed up some dynamic programs.

Recall that the main idea of tree based dynamic programming is that in a rooted tree, we can compute the dynamic program value of a node i by merging together the information of $\textsc{Children}(i)$.

By doing this on a particular tree of our choice, specifically the complete binary tree, we get a data structure that provides query access to ranges of an array under dynamic modifications to it.

This data structure can be used to speed up many algorithms. There the key idea is to figure out an ordering of processing the data so that the key computations become range queries on a slowly changing array.

2 Example Problems

To start, consider the dynamic version of the range sum problem:

Problem 1: Dynamic Prefix Sum under Entry Modification

Maintain an array $A[1 \dots n]$ to support the operations

1. $\textsc{Modify}(i, v)$: set $A[i]$ to v.
2. $\textsc{PrefixSum}(i)$: return the sum of $A[1 \dots i]$.

in $O(\log n)$ time per operation.

Since we deal with both array indices and tree nodes, we will use p, q and etc to denote tree

nodes from here on. We use $\mathrm{L}(p)$, $\mathrm{R}(p)$ and $\textsc{Parent}(p)$ to denote the left and right children of p respectively.

Consider a binary tree (which we will decide later) with the entries of the array as leaves, in order that they appear in the array.

For an internal node p, we let $\textsc{Sum}[p]$ denote the sum over all leaves of its subtree.

That is, we get the DP transition

$$\textsc{Sum}[p] = \textsc{Sum}[\mathrm{L}(p)] + \textsc{Sum}[\mathrm{R}(p)].$$

The base cases are the leaves, which correspond to entries of the array.

First we show that if the leaves are arranged in order of the array, aka the tree is ordered, these DP values give us effective ways of answering queries.

Consider finding the sum $[1 \dots i)$, that is, all the indices strictly before i. We start from the leaf corresponding to index i, and walk upwards to the root. Any time p is a right child, we know that its left sibling's entire subtree is to the left of the path, and should thus be included in the sum. The cost of this is again the height of the tree.

Now consider what happens when we update a leaf p: the only nodes whose sums change are those who subtrees contain p. This means we only need to update $\textsc{Sum}(q)$ for all nodes q that are ancestors of p. So the total cost is the depth of p.

Now the most important idea of this DP-based view of range queries is:

The algorithm designer can choose the tree.

Specifically, we can pick the tree with the smallest height possible: the balanced binary search tree. That's how the costs of all operations are kept to $O(\log n)$. To further simplify things, we put all keys on the leafs of this tree: when n is less than a power of 2, we simply create more 'dummy' nodes so that the tree is complete. This makes the height of the tree $O(\log n)$, so all the operations above take $O(\log n)$ time.

This is known as a range query. It and its extensions have a surprisingly large number of applications in algorithms. In fact, **every** topic in the rest of this course have variants that incorporate range queries/updates.

We start with two of the most classical examples of using range operations:

Problem 2: Inversion Counting

Given a permutation of $1 \dots n$, $A[1 \dots n]$, count the number of inversions in $O(n \log n)$ time. That is, compute in $O(n \log n)$ total time, for each j, compute the number of $i < j$ such that $A[i] > A[j]$.

First, observe that the permutation condition isn't necessary: we only care about the relative ordering of the entries in A. So we can sort and relabel them to values in the range $1 \dots n$ in $O(n \log n)$ time.

We will now count the number of j such that $j > i$ and $A[j] > A[i]$ in increasing order of i.

For each j, all the i before it are possible candidates. This means that when we go from j to $j+1$, the list of candidates only changes by 1: $A[j]$ needs to be considered for $A[j+1]$, but was no longer in consideration for $A[j]$.

This gives an update to the frequency vector, which we store as an array $B[1 \dots n]$. Counting the number of things greater than x is then a suffix sum query.

Formally, we loop through js in increasing order, and maintain an array $B[1 \dots n]$ that starts off as empty. At each j, we:

1. Query $\text{SUM}(B[(A[j]+1) \dots n])$, which is the number of things encountered so far that are greater than $A[j]$. This gives the number of inversions with j as the second value.
2. Increment $B[A[j]]$, this updates the frequency vector (B) from corresponding to $A[1 \dots (j-1)]$ to corresponding to $A[1 \dots j]$.

This is n entry updates and n suffix sum queries on B. Implementing these using the range query structure described above gives a total runtime of $O(n \log n)$.

The same 'loop through each element, update tree and range query' approach can speed up the longest increasing subsequence DP from Lecture 4. The only difference is the range query is now for max, instead of sum.

Problem 3: Longest Increasing Subsequence (LIS)

Given a sequence of comparable elements $a_1, a_2, \dots, a_n$, an increasing subsequence is a subsequence $a_{i_1}, a_{i_2}, \dots, a_{i_{k-1}}, a_{i_k}$ $(i_1 < i_2 < \dots i_k)$ such that

$$a_{i_1} < a_{i_2} < \dots < a_{i_{k-1}} < a_{i_k}.$$

A longest increasing subsequence is an increasing subsequence such that no other increasing subsequence is longer.

Find the length of the longest increasing subsequence of a list of n numbers in $O(n \log n)$ time.

To start, consider the 'natural' dynamic program where $DP[i]$ is the length of the longest increasing subsequence ending at location i. The transition is then

$$DP[i] = 1 + \max_{j<i, a_j<a_i} DP\big[j\big].$$

Again, since all the $A[i]$ values are given ahead of time, and we only care about how they compare, we can change them to integers in the range $1 \dots n$ via sorting in $O(n \log n)$ time, Then the query we are maintaining is a prefix query on the $A[i]$s, of the maximum DP value.

Again, having an array $B[1 \dots n]$ where

$$B[x] = \max_{j:a_j=x} DP\big[j\big]$$

means we just need to query for the maximum of $B[1\ldots(a_j-1)]$.

So we tree states that return the maximum leaf key in the subtree:

$$\textsc{SubtreeMax}[p] = \max_{j:\, j\in \textsc{Subtree}(p)} B[j]$$

and the merge function is

$$\textsc{SubtreeMax}[p] = \max\{\textsc{SubtreeMax}[\mathrm{L}(p)], \textsc{SubtreeMax}[\mathrm{R}(p)]\}.$$

Update then behaves the same as before. So we get a data structure that allows querying prefix maxes as well as updating single entries in $O(\log n)$ time per operation.

We then compute the DP array for LIS in almost the same manner:

1. Initialize $B[1\ldots n] = <0,0,\ldots,0>$, maintain using range query data structure.
2. For $i = 1\ldots n$
   (a) $DP[i] \leftarrow 1 + \textsc{QueryMax}(B[1\ldots(a_i-1)])$.
   (b) $\textsc{Update}(B[a_i], DP[i])$ (need to take max when multiple a_is have same valeu)
3. Return $\max_i DP[i]$.

Note that while there are other ways of optimizing LIS, this method is quite versatile: it naturally genearlizes to the weighted version, where each entry has a weight, and we want to find the max weight, instead of longest, increasing subsequence.

This ability to track maximums actually leads to another issue: querying the maximum of a subinterval instead of just prefix. This issue is not there for sum: we can get the sum of $A[l\ldots r]$ via $A[1\ldots r]-A[1\ldots(l-1)]$.

Problem 4: Dynamic Range Max under Entry Modifications

Maintain an array $A[1\ldots n]$ to support the operations

1. $\textsc{Modify}(i,v)$: set $A[i]$ to v.
2. $\textsc{RangeMax}(l,r)$ return the maximum of $A[l\ldots r]$.

Note that because $A[1]$ can be very large, the max of $A[1\ldots(l-1)]$ and $A[1\ldots r]$ give no information beyond $A[1]$.

Instead we move both l and r from the corresponding leaves up to the root, and take the max of everything 'in between'. That is, we aggregate all subtrees to the right of the path upwards from l and to the left of the path from r, until these paths meet.

Note that since each node has a unique parent, it will not be double counted when summing these two paths.

This doubles the cost of the access, so still $O(\log n)$ time.

—- Optional: segtree implementation details ....

Past treatment of range queries, often called 'segtrees', often go through the tree path access statements in significant detail.

These update/query processes are mostly independent of the definition and maintenance of the info on tree nodes, which (in my opinion) are the crux of the difficulties of showing that a range operation is doable.

Over the past few years, there are a number of libaries that automatically perform the tree traversals and merges of the subtrees. With access to such libraries, it's even possible to code such a data structure by just defining the node type and the merge operation.

We discuss one such example using the AtCoder Library, `https://github.com/atcoder/ac-library/blob/master/document_en/segtree.md`.

Once this library is set up, the sum tracking segtree for inversion counting is

```
segtree<int, [](int a, int b){return a + b;}, [](){return 0;}> B(n);
```

while the max tracking segtree for longest increasing subsequence is

```
segtree<int, [](int a, int b){return max(a, b);}, [](){return (int)0;}> B(n);
```

(there should be a way to just pass in STL's max function, but I don't know how to fix the types for that)

For both of these, modifying entry i to x done by

```
B.set(i, x);
```

while querying the range $B[l \dots r]$ is given by

```
B.query(l, r);
```

Formally, the parameters taken by

```
segtree<S, op, e> seg(int n)
```

are

1. S: the data type.
2. op: the 'combine' operator that takes two copies of S and returns the 'merged' copy.
3. $e()$: function that returns the identity element, which the array gets initialized with.
4. n: length of array.

The only formal requirement (for point update, range query) is that $op()$ is associative, that is, $op(op(a,b),c) = op(a,op(b,c))$. Such a condition is sufficient for things within a subtree to be combined first, which is how we answer queries.

The existence of this kind of encapsulated APIs is why for the purpose of analyzing running times, we focus on defining the tree node 'states', and how they merge (analog of DP transitions).

One more example of a harder one: in an array with possibly negative numbers, we want to find the maximum sum of an interval starting at l, ending at most r. That is,

$$\max_{l \le j \le r} \textsc{Sum}(B[l \dots i]).$$

This is done by storing both the sum, and the max prefix sum in the current node.

The merge function / transition for sum is same as before. The merge function for prefix sum includes a case of taking the sum of left, plus the max prefix sum of the right:

$$\textsc{MaxPrefixSum}[p] = \max\{\textsc{MaxPrefixSum}[\mathrm{L}(p)], \textsc{Sum}[\mathrm{L}(p)] + \textsc{MaxPrefixSum}[\mathrm{R}(p)]\}.$$

In terms of the above code, we now store a pair of numbers, and the merge function needs to take a pair of numbers into account.

```
#define F first
#define S second
#define R return

typedef pair<long long, long long> S; //Sum, MaxPrefixSum
segtree<S, [](S a, S b){R S(a.F+b.F,max(a.S,a.F+b.S));}, [](){R S(0,0);}> B(n);
```

More documentations on this library is at `https://atcoder.github.io/ac-library/production/document_en/index.html`.