IOI Training Week 7
Advanced Data Structures

Tim Dumol

Contents

1 Range Minimum Query

This section rotates around a common problem:

Definition 1 (Range Minimum Query (RMQ))**.** Given an integer array of fixed length, answer a set of queries of the form: "What is the minimum element in the range of the array from `i` to `j`?". The contents of the array may change between queries.

The naïve solution for RMQ has no setup time, and $O(n)$ query time. We can improve on this by adding some setup time, and using some additional memory. We will discuss two approaches: square-root decomposition, and segment trees.

1.1 Square-root (sqrt) Decomposition

The idea behind sqrt decomposition is simple: preprocess the array into $\sqrt{n}$ chunks of size $\sqrt{n}$ each (thus consuming $O(\sqrt{n})$ extra memory), so that we can perform the query in $O(\sqrt{n})$ time, by using the preprocessed chunks to compute the minimum for the parts of the range that have a full intersection with the chunks, and then traversing the remaining at most $2\left(\sqrt{n}-1\right)$ elements uncovered by the chunks[1]. To elaborate, in code[2]:

```
struct SqrtDecomp {
  vector<int>* arr;
  vector<int> chunks;
  int chunk_size;
  int n_chunks;
  SqrtDecomp(vector<int> const *arr) : arr(arr) {
    chunk_size = (int)sqrt(arr->size());
    n_chunks = (int)ceil(arr->size()/(double)n_chunks);
    chunks.resize(n_chunks);
```

[1] One can see that this can be extended to any associative operation.
[2] code is untested, if it's wrong, feel free to correct

```
    for (int i = 0; i < n_chunks; ++i) {
      // cap computed in advance to avoid recomputing
      const int cap = min(n_chunks * (i+1), arr->size());
      // assumption: all input values less than 1 << 30
      chunks[i] = 1 << 30;
      for (int j = i*chunk_size; j < cap; ++j) {
        chunks[i] = min(arr[j], chunks[i]);
      }
    }
  }

  // end is exclusive
  int query(int begin, int end) {
    int left = (int)ceil(begin/(double)chunk_size);
    int right = (int)floor(end/(double)chunk_size);
    int ans = 1 << 30;
    if (left <= right) {
      ans = min(ans, min_element(chunks.begin() + left, chunks.begin() + right));
    }
    if (start % chunk_size != 0) {
      ans = min(ans, min_element(arr->begin() + begin, arr->begin() + ↙
          ↳ chunk_size*left));
    }
    if (end % chunk_size != 0) {
      ans = min(ans, min_element(arr->begin() + chunk_size*right, arr->begin() + ↙
          ↳ end));
    }
    return ans;
  }
};
```

The code to update the sqrt decomposition is an exercise left to the reader (you don't need to submit it).

1.2 Segment Trees

But a query time $O(\sqrt{n})$ is still pretty slow. Can we do faster? The answer is yes. We can get $O(\log(n))$ query time with $O(n)$ extra memory, by using a segment tree.

Definition 2 (Segment Tree)**.** A segment tree over an array of length n (for simplicity, let's say it's a power of two – extending to a non-power of two is an exercise for the reader) is a balanced binary[3] tree with n leaves, each corresponding to an element in the array. Each internal vertex of the segment tree has a value corresponding to the minimum of all vertices under its subtree.

Since a segment tree is a complete binary tree, it can be represented similarly as a heap, using only a single-dimensional integer array. Furthermore, a segment tree has $2n-1$ vertices, and can be constructed in $O(n)$ time. For querying, we recursively traverse the segment tree, stopping when the segment covered by a vertex is wholly included in the query range. It can be shown that this results in $O(\log(n))$ time[4].

```
struct ST {
  typedef int el_type;
  vector<el_type> tree;
  int n;
  ST(vector<el_type> const& arr) {
    n = arr.size();
    tree.resize(2*arr.size());
```

[3] technically you can use any arity, but for simplicity let's say binary

[4] https://cs.stackexchange.com/questions/37669/time-complexity-proof-for-segment-tree-implementation-of-the-ranged-sum-proble

```
    build(0, arr, 0, arr.size());
  }

  inline int left(int idx) const { return 2*idx + 1; }
  inline int right(int idx) const { return 2*idx + 2; }

  // end is exclusivel, as usual
  int build(int idx, vector<el_type> const& arr, int start, int end) {
    if (start + 1 == end) {
      return (tree[idx] = arr[start]);
    }
    const int mid = (start + end)/2;
    return (tree[idx] = min(build(left(idx), arr, start, mid), build(right(idx), ↙
        ↳ arr, mid, end)));
  }

  int query(int start, int end) {
    return query(start, end, 0, 0, n);
  }

  int query(int start, int end, int idx, int tree_start, int tree_end) {
    if (start <= tree_start && tree_end <= end) {
      return tree[idx];
    } else {
      const int mid = (tree_start + tree_end)/2;
      return min(query(start, end, left(idx), tree_start, mid), query(start, ↙
          ↳ end, right(idx), mid, tree_end));
    }
  }
};
```

The code to update a single element is an exercise left to the reader (you don't need to submit it). Furthermore, through the usage of lazy propagation, you can make it so that you can update a range of elements in the segment tree (also an exercise left to the reader – hint: you need to only add a single piece of additional information, if your update is adding an integer to a given range).

Similar to sqrt decomposition, you can adapt the segment tree to any associative operation (e.g., range sum query).

1.3 Notes

Coincidentally, RMQ turns out to be deeply related to another (somewhat rarer, but also standard) problem, LCA:

Definition 3 (Lowest Common Ancestor (LCA))**.** Given a rooted tree, `T`, answer a set of queries of the form: "What is the vertex of `T` that is farthest from the root, that is an ancestor of both vertices `x` and `y`?".

The naïve solution for LCA has no setup time, and $O(n)$ query time, and it turns out you can preprocess LCA into an RMQ problem, and vice versa. You can also use square-root decomposition for LCA, although of a different form. To learn more about this (and asymptotically better solutions to RMQ), check out the RMQ and LCA tutorial on Topcoder (c.f. references).

2 Self-Balancing Binary Search Trees

Recall that most library implementations of the `map` and `set` datastructures use a self-balancing binary search tree (usually a Red-Black tree), allowing query, delete, and insert operations in amortized $O(\log(n))$ time. Usually, this is good enough for our purposes, but sometimes we need to augment each vertex with some additional information in order to enable a special kind of query not supported by library implementations.

For example, you may have a dynamically changing ordered list of integers, and have to find the index of an arbitrary integer in the list.

Now, the most commonly used trees in library code are AVL trees and Red-Black trees, because of their good asymptotic characteristics. However, they are a pain to code, and have a lot of edge cases. Thus, for competitive programming, we usually implement simpler trees: either scapegoat trees or treaps. In this discussion, we'll focus on treaps (because the author considers them easier to implement and understand)[5].

A treap is a binary search tree augmented with a randomized priority value on each of its vertices, and kept balanced by maintaining a heap (priority queue) structure on its vertices (i.e., each vertex should have a smaller priority than its children). Because a random heap is balanced, a treap is probabilistically balanced. The hard part in implementing a treap is in maintaining the heap structure, as one will have to rotate vertices that violate the heap structure, while maintaining the BST structure.

```
struct TreapNode {
  int start, end;
  int priority;
  TreapNode *kids[2];
  TreapNode *parent;
  TreapNode(){}
  void init(int start, int end, TreapNode *parent);
  void update_aug() {
    // whatever augmentation you need
  }
};

// we cache the objects so that we don't need to dynamically
// allocate objects in heap (this is also known as arena
// allocation)
TreapNode cache[3000024];
int cctr;
void TreapNode::init(int start, int end, TreapNode *parent) {
  kids[0] = kids[1] = NULL;
  this->parent = parent;
  this->start = start;
  this->end = end;
  priority = rand();
}

struct Treap {
  TreapNode *root;
  Treap() : root(NULL) {}
  // dir: 0 is left, 1 is right
  void rotate(TreapNode *node, int dir) {
    TreapNode *rkid = node->kids[dir^1];
    if (node->parent) {
      if (node == node->parent->kids[0]) {
        node->parent->kids[0] = rkid;
      } else {
        node->parent->kids[1] = rkid;
      }
    } else {
      root = rkid;
    }
    rkid->parent = node->parent;
    node->kids[dir^1] = rkid->kids[dir];
    if (node->kids[dir^1]) {
```

[5]But briefly: scapegoat trees remain balanced by maintaining some statistics on their subtrees, and if a subtree is sufficiently "imbalanced", it completely reconstructs that subtree. This rebalancing is done rarely enough that operations on a scapegoat tree remain $O(\log(n))$ time.

```
      node->kids[dir^1]->parent = node;
    }
    // finally, transplant orig to new parent
    rkid->kids[dir] = node;
    node->parent = rkid;

    // update augmentation
    node->update_aug();
    rkid->update_aug();
  }
  void add(int start, int end) {
    if (root == NULL) {
      root = &cache[cctr++];
      root->init(start, end, NULL);
      return;
    }
    // look for node to insert at
    TreapNode *p = root;
    TreapNode *node = NULL;
    while (true) {
      if (start < p->start) {
        if (!p->kids[0]) {
          node = p->kids[0] = &cache[cctr++];
          break;
        } else p = p->kids[0];
      } else {
        if (!p->kids[1]) {
          node = p->kids[1] = &cache[cctr++];
          break;
        } else p = p->kids[1];
      }
    }
    node->init(start, end, p);
    // maintain augmentation
    TreapNode *p2 = p;
    while (p2 != NULL) {
      p2->update_aug();
      p2 = p2->parent;
    }

    // rotate
    while (node->parent && node->priority > node->parent->priority) {
      if (node == node->parent->kids[0]) {
        rotate(node->parent, 1);
      } else {
        rotate(node->parent, 0);
      }
    }
  }
  int search(TreapNode *p, int start, int end) {
    // problem-dependent
  }
  int search(int start, int end) {
    return search(root, start, end);
  }
};
```

3 Bonus: More interesting data structures

This is just a listing of interesting data structures that you may want to look into:

- Binary Index Tree (BIT) (aka Fenwick Tree) https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/ – used to get prefix sums; functionality is a subset of Segment Tree, but has faster to type implementation
- k-d tree https://en.wikipedia.org/wiki/K-d_tree – allows you to query the nearest neighbor of a point among a dynamic set of points in k-dimensional space, in $O(\log(n))$ time. (out of IOI scope)
- Skip list https://en.wikipedia.org/wiki/Skip_list – randomized data structure with performance characteristics similar to a balanced binary search tree (very rare usage in competitive programming)

4 Problems

1. Reverse Prime (https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2657)
2. Frequent Values (https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=2176)
3. Census (https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2272)
4. Can you answer these queries VIII (http://www.spoj.com/problems/GSS8/)
5. Permutations (https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=2520)
6. Ahoy, Pirates! (https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2397)
7. XOR on Segment (http://codeforces.com/problemset/problem/242/E)
8. Robotic Sort http://www.spoj.com/problems/CERC07S/

4.1 Bonus Problems

These problems are ungraded.

1. Alien Abduction (https://icpcarchive.ecs.baylor.edu/index.php?option=com_onlinejudge&Itemid=8&category=559&page=show_problem&problem=4056)
2. Genetics (https://www.codechef.com/problems/GENETICS)
3. Little Elephant and Tree (http://codeforces.com/problemset/problem/258/E)

5 References

1. RMQ and LCA tutorial https://www.topcoder.com/community/data-science/data-science-tutorials/range-minimum-query-and-lowest-common-ancestor/