Sqrt Decomposition

Table of Contents

Sqrt Decomposition is a method (or a data structure) that allows you to perform some common operations (finding sum of the elements of the sub-array, finding the minimal/maximal element, etc.) in $O(\sqrt{n})$ operations, which is much faster than $O(n)$ for the trivial algorithm.

First we describe the data structure for one of the simplest applications of this idea, then show how to generalize it to solve some other problems, and finally look at a slightly different use of this idea: splitting the input requests into sqrt blocks.

Sqrt-decomposition based data structure

Given an array $a[0 \dots n-1]$, implement a data structure that allows to find the sum of the elements $a[l \dots r]$ for arbitrary l and r in $O(\sqrt{n})$ operations.

Description

The basic idea of sqrt decomposition is preprocessing. We'll divide the array a into blocks of length approximately $\sqrt{n}$, and for each block i we'll precalculate the sum of elements in it $b[i]$.

We can assume that both the size of the block and the number of blocks are equal to $\sqrt{n}$ rounded up:

$$s = \lceil \sqrt{n} \rceil$$

Then the array a is divided into blocks in the following way:

$$\underbrace{a[0], a[1], \dots, a[s-1]}\_\text{b}[0], \underbrace{a[s], \dots, a[2s-1]}\_\text{b}[1], \dots, \underbrace{a[(s-1) \cdot s], \dots, a[n-1]}\_\text{b[s-1]}$$

The last block may have fewer elements than the others (if n not a multiple of s), it is not important to the discussion (as it can be handled easily). Thus, for each block k, we know the sum of elements on it $b[k]$:

$$b[k] = \sum_{i=k \cdot s}^{\min(n-1, (k+1) \cdot s - 1)} a[i]$$

So, we have calculated the values of $b[k]$ (this required $O(n)$ operations). How can they help us to answer each query $[l, r]$? Notice that if the interval $[l, r]$ is long enough, it will contain several whole blocks, and for those blocks we can find the sum of elements in them in a single operation. As a result, the interval $[l, r]$ will contain parts of only two blocks, and we'll have to calculate the sum of elements in these parts trivially.

Thus, in order to calculate the sum of elements on the interval $[l, r]$ we only need to sum the elements of the two "tails": $[l \dots (k+1) \cdot s - 1]$ and $[p \cdot s \dots r]$, and sum the values $b[i]$ in all the blocks from $k+1$ to $p-1$:

$$\sum\_i = l^r a[i] = \sum\_i = l^{(k+1) \cdot s - 1} a[i] + \sum\_i = k+1^{p-1} b[i] + \sum\_i = p \cdot s^r a[i]$$

Note: When $k = p$, i.e. l and r belong to the same block, the formula can't be applied, and the sum should be calculated trivially.

This approach allows us to significantly reduce the number of operations. Indeed, the size of each "tail" does not exceed the block length s, and the number of blocks in the sum does not exceed s. Since we have chosen $s \approx \sqrt{n}$, the total number of operations required to find the sum of elements on the interval $[l, r]$ is $O(\sqrt{n})$.

Implementation

Let's start with the simplest implementation:

```
// input data
int n;
vector<int> a (n);

// preprocessing
int len = (int) sqrt (n + .0) + 1; // size of the block and the number of blocks
vector<int> b (len);
for (int i=0; i<n; ++i)
    b[i / len] += a[i];

// answering the queries
for (;;) {
    int l, r;
  // read input data for the next query
    int sum = 0;
    for (int i=l; i<=r; )
        if (i % len == 0 && i + len - 1 <= r) {
            // if the whole block starting at i belongs to [l, r]
            sum += b[i / len];
            i += len;
        }
        else {
            sum += a[i];
            ++i;
        }
}
```

This implementation has unreasonably many division operations (which are much slower than other arithmetical operations). Instead, we can calculate the indices of the blocks c_l and c_r which contain indices l and r, and loop through blocks $c_l + 1 \dots c_r - 1$ with separate processing of the "tails" in blocks c_l and c_r. This approach corresponds to the last formula in the description, and makes the case $c_l = c_r$ a special case.

```
int sum = 0;
int c_l = l / len,   c_r = r / len;
if (c_l == c_r)
    for (int i=l; i<=r; ++i)
        sum += a[i];
else {
    for (int i=l, end=(c_l+1)*len-1; i<=end; ++i)
        sum += a[i];
    for (int i=c_l+1; i<=c_r-1; ++i)
        sum += b[i];
    for (int i=c_r*len; i<=r; ++i)
        sum += a[i];
}
```

Other problems

So far we were discussing the problem of finding the sum of elements of a continuous subarray. This problem can be extended to allow to **update individual array elements**. If an element $a[i]$ changes, it's sufficient to update the value of $b[k]$ for the block to which this element belongs ($k = i/s$) in one operation:

$$b[k] += a_{new}[i] - a_{old}[i]$$

On the other hand, the task of finding the sum of elements can be replaced with the task of finding minimal/maximal element of a subarray. If this problem has to address individual elements' updates as well, updating the value of $b[k]$ is also possible, but it will require iterating through all values of block k in $O(s) = O(\sqrt{n})$ operations.

Sqrt decomposition can be applied in a similar way to a whole class of other problems: finding the number of zero elements, finding the first non-zero element, counting elements which satisfy a certain property etc.

Another class of problems appears when we need to **update array elements on intervals**: increment existing elements or replace them with a given value.

For example, let's say we can do two types of operations on an array: add a given value δ to all array elements on interval $[l, r]$ or query the value of element $a[i]$. Let's store the value which has to be added to all elements of block k in $b[k]$ (initially all $b[k] = 0$). During each "add" operation we need to add δ to $b[k]$ for all blocks which belong to interval $[l, r]$ and to add δ to $a[i]$ for all elements which belong to the "tails" of the interval. The answer a query i is simply $a[i] + b[i/s]$. This way "add" operation has $O(\sqrt{n})$ complexity, and answering a query has $O(1)$ complexity.

Finally, those two classes of problems can be combined if the task requires doing **both** element updates on an interval and queries on an interval. Both operations can be done with $O(\sqrt{n})$ complexity. This will require two block arrays b and c: one to keep track of element updates and another to keep track of answers to the query.

There exist other problems which can be solved using sqrt decomposition, for example, a problem about maintaining a set of numbers which would allow adding/deleting numbers, checking whether a number belongs to the set and finding k-th largest number. To solve it one has to store numbers in increasing order, split into several blocks with $\sqrt{n}$ numbers in each. Every time a number is added/deleted, the blocks have to be rebalanced by moving numbers between beginnings and ends of adjacent blocks.

Mo's algorithm

A similar idea, based on sqrt decomposition, can be used to answer range queries (Q) offline in $O((N + Q)\sqrt{N})$. This might sound like a lot worse than the methods in the previous section, since this is a slightly worse complexity than we had earlier and cannot update values between two queries. But in a lot of situations this method has advantages. During a normal sqrt decomposition, we have to precompute the answers for each block, and merge them during answering queries. In some problems this merging step can be quite problematic. E.g. when each queries asks to find the **mode** of its range (the number that appears the most often). For this each block would have to store the count of each number in it in some sort of data structure, and we cannot longer perform the merge step fast enough any more. **Mo's algorithm** uses a completely different approach, that can answer these kind of queries fast, because it only keeps track of one data structure, and the only operations with it are easy and fast.

The idea is to answer the queries in a special order based on the indices. We will first answer all queries which have the left index in block 0, then answer all queries which have left index in block 1 and so on. And also we will have to answer the queries of a block is a special order, namely sorted by the right index of the queries.

As already said we will use a single data structure. This data structure will store information about the range. At the beginning this range will be empty. When we want to answer the next query (in the special order), we simply extend or reduce the range, by adding/removing elements on both sides of the current range, until we transformed it into the query range. This way, we only need to add or remove a single element once at a time, which should be pretty easy operations in our data structure.

Since we change the order of answering the queries, this is only possible when we are allowed to answer the queries in offline mode.

Implementation

In Mo's algorithm we use two functions for adding an index and for removing an index from the range which we are currently maintaining.

```
void remove(idx);  // TODO: remove value at idx from data structure
void add(idx);     // TODO: add value at idx from data structure
int get_answer();  // TODO: extract the current answer of the data structure
```

```
int block_size;

struct Query {
    int l, r, idx;
    bool operator<(Query other) const
    {
        return make_pair(l / block_size, r) <
               make_pair(other.l / block_size, other.r);
    }
};

vector<int> mo_s_algorithm(vector<Query> queries) {
    vector<int> answers(queries.size());
    sort(queries.begin(), queries.end());

    // TODO: initialize data structure

    int cur_l = 0;
    int cur_r = -1;
    // invariant: data structure will always reflect the range [cur_l, cur_r]
    for (Query q : queries) {
        while (cur_l > q.l) {
            cur_l--;
            add(cur_l);
        }
        while (cur_r < q.r) {
            cur_r++;
            add(cur_r);
        }
        while (cur_l < q.l) {
            remove(cur_l);
            cur_l++;
        }
        while (cur_r > q.r) {
            remove(cur_r);
            cur_r--;
        }
        answers[q.idx] = get_answer();
    }
    return answers;
}
```

Based on the problem we can use a different data structure and modify the `add`/`remove`/`get_answer` functions accordingly. For example if we are asked to find range sum queries then we use a simple integer as data structure, which is 0 at the beginning. The `add` function will simply add the value of the position and subsequently update the answer variable. On the other hand `remove` function will subtract the value at position and subsequently update the answer variable. And `get_answer` just returns the integer.

For answering mode-queries, we can use a binary search tree (e.g. `map<int, int>`) for storing how often each number appears in the current range, and a second binary search tree (e.g. `set<pair<int, int>>`) for keeping counts of the numbers (e.g. as count-number pairs) in order. The `add` method removes the current number from the second BST, increases the count in the first one, and inserts the number back into the second one. `remove` does the same thing, it only decreases the count. And `get_answer` just looks at second tree and returns the best value in $O(1)$.

Complexity

Sorting all queries will take $O(Q \log Q)$.

How about the other operations? How many times will the `add` and `remove` be called?

Let's say the block size is S.

If we only look at all queries having the left index in the same block, the queries are sorted by the right index. Therefore we will call `add(cur_r)` and `remove(cur_r)` only $O(N)$ times for all these queries combined. This gives $O(\frac{N}{S}N)$ calls for all blocks.

The value of `cur_l` can change by at most $O(S)$ during between two queries. Therefore we have an additional $O(SQ)$ calls of `add(cur_l)` and `remove(cur_l)`.

For $S \approx \sqrt{N}$ this gives $O((N+Q)\sqrt{N})$ operations in total. Thus the complexity is $O((N+Q)F\sqrt{N})$ where $O(F)$ is the complexity of `add` and `remove` function.

Tips for improving runtime

- Block size of precisely $\sqrt{N}$ doesn't always offer the best runtime. For example, if $\sqrt{N} = 750$ then it may happen that block size of 700 or 800 may run better. More importantly, don't compute the block size at runtime - make it `const`. Division by constants is well optimized by compilers.
- In odd blocks sort the right index in ascending order and in even blocks sort it in descending order. This will minimize the movement of right pointer, as the normal sorting will move the right pointer from the end back to the beginning at the start of every block. With the improved version this resetting is no more necessary.

```
bool cmp(pair<int, int> p, pair<int, int> q) {
    if (p.first / BLOCK_SIZE != q.first / BLOCK_SIZE)
        return p < q;
    return (p.first / BLOCK_SIZE & 1) ? (p.second < q.second) : (p.second > q.second);
}
```

You can read about even faster sorting approach here.

Practice Problems

- UVA - 12003 - Array Transformer
- UVA - 11990 Dynamic Inversion
- SPOJ - Give Away
- Codeforces - Till I Collapse
- Codeforces - Destiny
- Codeforces - Holes
- Codeforces - XOR and Favorite Number
- Codeforces - Powerful array
- SPOJ - DQUERY