Square Root Decomposition

IOI Training Camp 2 - 2022

Benjamin Kleyn

Why?

Sqrt Decomposition allows you to perform operations such as finding the sum or maximum of the elements of a subarray, in $O(\sqrt{N})$ operations, where N is the number of elements in the array.

What?

The idea of sqrt decomposition is to decompose the array into $\sqrt{N}$-sized blocks.

(If N is not a perfect square we can round it up)

Range Sum Query Example

Given an array $a = a_0, a_1, \ldots, a_{15}$

We break the array into 4 blocks with size 4 and precompute their sums in b.

a_0	a_1	a_2	a_3	b_0
a_4	a_5	a_6	a_7	b_1
a_8	a_9	a_{10}	a_{11}	b_2
a_{12}	a_{13}	a_{14}	a_{15}	b_3

Range Sum Query Example

Now, for each query on a range $[l, r]$, we can notice when the range spans over a block, and then only add the value of the block, instead of adding each of the elements in the block.

a_0	a_1	a_2	a_3	b_0
a_4	a_5	a_6	a_7	b_1
a_8	a_9	a_{10}	a_{11}	b_2
a_{12}	a_{13}	a_{14}	a_{15}	b_3

Range Sum Query Example

Sum of range $[2, 9] = a_2 + a_3 + b_1 + a_8 + a_9$

a_0	a_1	a_2	a_3	b_0
a_4	a_5	a_6	a_7	b_1
a_8	a_9	a_{10}	a_{11}	b_2
a_{12}	a_{13}	a_{14}	a_{15}	b_3

Range Sum Query Example

Sum of range $[4, 14] = b_1 + b_2 + a_{12} + a_{13} + a_{14}$

a_0	a_1	a_2	a_3	b_0
a_4	a_5	a_6	a_7	b_1
a_8	a_9	a_{10}	a_{11}	b_2
a_{12}	a_{13}	a_{14}	a_{15}	b_3

Implementation

```
// initialization
int N;
vector<int> a(N);

int S = (int) sqrt(N - 0.1) + 1;
vector<int> b(S, 0);

for (int i = 0; i < N; i++)
    b[i / S] += a[i];
```

Implementation

```
// Function to return sum of range [l, r]
int sum(int l, int r)
{
    int result = 0;
    int l_block = l / S, r_block = r / S;
    if (l_block == r_block)
    {
        for (int i = l; i <= r; i++)
            result += a[i];
    }
    else
    {
        for (int i = l; i < (l_block + 1) * S; i++)
            result += a[i];
        for (int i = l_block + 1; i <= r_block - 1; i++)
            result += b[i];
        for (int i = r_block * S; i <= r; i++)
            result += a[i];
    }

    return result;
}
```

Implementation

```
// Function to change value of element at index i
void update(int i, int value)
{
    b[i / S] += value - a[i];
    a[i] = value;
}
```

Complexity

For each call of the sum function, it will take at most $\sqrt{N}$ iterations to sum the block-values, and at most $2\sqrt{N}$ iterations to sum the individual elements on the sides, resulting in a time complexity of $O(\sqrt{N})$.

The update function in this case is $O(1)$, but in the case of range min- and max queries, one might need to iterate over the block again to find the minimum or maximum, causing it to also have a time complexity of $O(\sqrt{N})$.

For this reason, sqrt decomposition could possibly perform better than a segment or fenwick tree on range sum queries if the number of updates is significantly greater than the number of sum operations. (which is not usually the case)

Another epic use (range updates)

Problem - Process queries of following types:

- Add value d to all array elements in range $[l, r]$.
- Return value of element with index i.

Solution:

Initially, a and b are filled with 0.

For each update, add d to all blocks that are within the range, and to all other individual elements that are at the ends of these blocks. $O(\sqrt{N})$

At any point in time, the value of the element at index i is $a_i + b_k$ where k is the number of the block that contains i.

Implementation

```
const int MaxN = 10000;
const int S = 100;

int a[MaxN] = {0};
int b[S] = {0};
```

```
void update(int l, int r, int dval)
{
    int l_block = l / S, r_block = r / S;
    if (l_block == r_block)
    {
        for (int i = l; i <= r; i++)
            a[i] += dval;
    }
    else
    {
        for (int i = l; i < (l_block+1) * S; i++)
            a[i] += dval;
        for (int i = l_block + 1; i < r_block; i++)
            b[i] += dval;
        for (int i = r_block*S; i <= r; i++)
            a[i] += dval;
    }
}

int value(int index)
{
    return a[index] + b[index];
}
```

Range Updates and sums

```
const int MaxN = 10000;
const int S = 100;

int a[MaxN] = {0};
int b[S] = {0};
int c[S] = {0};

void update(int l, int r, int dval) // O(sqrt(N))
{
    int l_block = l / S, r_block = r / S;
    if (l_block == r_block)
    {
        for (int i = l; i <= r; i++)
            a[i] += dval, c[l_block] += dval;
    }
    else
    {
        for (int i = l; i < (l_block+1) * S; i++)
            a[i] += dval, c[l_block] += dval;
        for (int i = l_block + 1; i < r_block; i++)
            b[i] += dval, c[i] += dval * S;
        for (int i = r_block*S; i <= r; i++)
            a[i] += dval, c[r_block] += dval;
    }
}
```

```
int sum(int l, int r) // O(sqrt(N))
{
    int res = 0;
    int l_block = l / S, r_block = r / S;
    if (l_block == r_block)
    {
        for (int i = l; i <= r; i++)
            res += a[i] + b[l_block];
    }
    else
    {
        for (int i = l; i < (l_block+1) * S; i++)
            res += a[i] + b[l_block];
        for (int i = l_block + 1; i < r_block; i++)
            res += c[i];
        for (int i = r_block*S; i <= r; i++)
            res += a[i] + b[r_block];
    }
    return res;
}
```

Example problem:

codeforces.com/problemset/problem/13/E

E. Holes

time limit per test: 1 second
memory limit per test: 64 megabytes
input: standard input
output: standard output

Little Petya likes to play a lot. Most of all he likes to play a game «Holes». This is a game for one person with following rules:

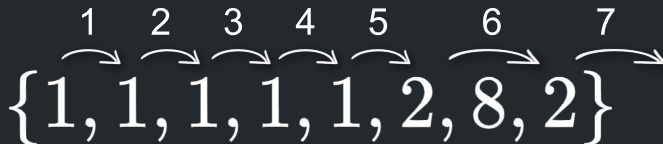
There are N holes located in a single row and numbered from left to right with numbers from 1 to N. Each hole has it's own power (hole number i has the power a_i). If you throw a ball into hole i it will immediately jump to hole $i + a_i$, then it will jump out of it and so on. If there is no hole with such number, the ball will just jump out of the row. On each of the M moves the player can perform one of two actions:

- Set the power of the hole a to value b.
- Throw a ball into the hole a and count the number of jumps of a ball before it jump out of the row and also write down the number of the hole from which it jumped out just before leaving the row.

Petya is not good at math, so, as you have already guessed, you are to perform all computations.

Example Query type 2 :

If you throw ball into first hole, it will take 7 jumps to exit the row, and the last hole it will visit is hole number 8, so the expected output is “8 7”.



Solution :

One could pre-process the array from right to left to store the result for each query in $O(N)$, but then each time there is an update to a hole, we need to repeat this, which will be too slow, so instead we use sqrt decomposition.

For each hole i, use pre-processing to store the last hole that is visited in the same block (last[i]), and how many jumps it takes to get to the next block cnt[i]. For each jump query, we can jump from the current block to the next block instead of from hole to hole, and for each update, we only need to re-process the block of the hole that is changed. A block size of $\sqrt{N}$, gives a time complexity of $O(\sqrt{N})$ for each query.

Mo’s Algorithm

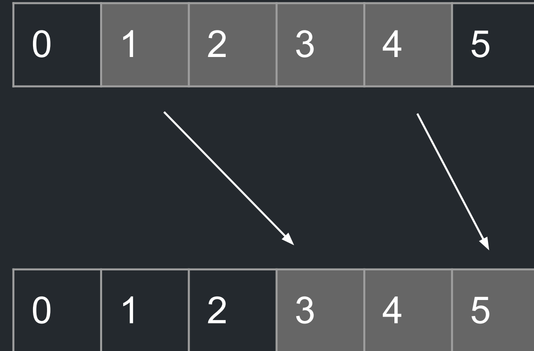
Based on sqrt decomposition, answers queries offline in $O((N+Q)\sqrt{N})$.

Can be used for range mode queries and other range queries that would be too complicated with just normal sqrt decomposition.

Mo's Algorithm

The main idea is to move from one query that we know the answer of to another by adding or removing array elements at either end to get from one range to another.

For example, to move from query [1, 4] to [3, 5], we would need to remove the elements 1 and 2, and add the element 5.



Basic implementation

We start with no elements added and a starting range of [0, -1]. And then we move from query to query updating our current range and data structure (an integer) as we go.

```
int total = 0;
int cur_l = 0, cur_r = -1;
for (int i = 0; i < Q; i++)
{
    int l, r; cin >> l >> r;

    // add elements
    while (cur_l > l)
        total += a[--cur_l];
    while (cur_r < r)
        total += a[++cur_r];
    // remove elements
    while (cur_l < l)
        total -= a[cur_l++];
    while (cur_r > r)
        total -= a[cur_r--];

    cout << total << '\n';
}
```

Sorting queries

To reduce the number of operations, we first process all the queries with their left value in the first block in order of their right value, and then second block in order of their right value, etc.

This way, for each of the Q queries, the left value will change by at most $\sqrt{N}$, and for each of the $\sqrt{N}$ blocks of queries, the right value will change by at most N, resulting in a time complexity of $O((N+Q)\sqrt{N})$.

There are also sorting methods that are faster but more complex.

https://codeforces.com/blog/entry/61203

```
int N;
int S = (int) sqrt(N - 0.1) + 1;

struct Query {int l, r, idx; };

bool operator<(Query p, Query q)
{
    if (p.l / S == q.l / S)
        return p.r < q.r;
    return p.l < q.l;
}

int main()
{
    int Q;
    cin >> Q;
    vector<Query> queries(Q);
    for (int i = 0; i < Q; i++)
    {
        cin >> queries[i].l >> queries[i].r;
        queries[i].idx = i;
    }
    sort(queries.begin(),queries.end());
    vector<int> res(Q);
```

Very general implementation

For queries that are more complex than range sum, we use a different data structure to represent the range [cur_l, cur_r].

In general, we can use this exact code for any application of mo's algorithm, and all we need to change is the add, remove and get_answer functions to fit the data structure.

```
vector<int> mo(vector<Query> queries)
{
    vector<int> res(queries.size());
    sort(queries.begin(), queries.end());
    int cur_l = 0, cur_r = -1;
    for (Query q:queries)
    {
        while (cur_l > q.l)
            add(a[--cur_l]);
        while (cur_r < q.r)
            add(a[++cur_r]);
        while (cur_l < q.l)
            remove(a[cur_l++]);
        while (cur_r > q.r)
            remove(a[cur_r--]);

        res[q.idx] = get_answer();
    }
    return res;
}
```

Range Mode Query

In the case of range mode queries, we can use unordered_map<int, int> to keep count of each number and set<pair<int, int>> to find the mode.

This makes the complexity $O(\log(N)(N+Q)\sqrt{N})$

```
unordered_map<int,int> cnt;
set<pair<int,int>> numz; // {count, value} pairs

void add(int value)
{
    int Count = cnt[value]++;
    numz.insert({Count + 1,value});
    numz.erase({Count, value});
}

void remove(int value)
{
    int Count = cnt[value]--;
    numz.insert({Count - 1,value});
    numz.erase({Count, value});
}

int get_answer()
{
    return (*numz.rbegin()).second;
}
```

Example problem:

codeforces.com/contest/86/problem/D

D. Powerful array

time limit per test: 5 seconds

memory limit per test: 256 megabytes

input: standard input

output: standard output

An array of positive integers $a_1, a_2, ..., a_n$ is given. Let us consider its arbitrary subarray $a_l, a_{l+1}..., a_r$, where $1 \le l \le r \le n$. For every positive integer s denote by K_s the number of occurrences of s into the subarray. We call the *power* of the subarray the sum of products $K_s \cdot K_s \cdot s$ for every positive integer s. The sum contains only finite number of nonzero summands as the number of different values in the array is indeed finite.

You should calculate the power of t given subarrays.

Example: Power of range [1, 6]

$$
\begin{aligned}
&= K_1 \cdot K_1 \cdot 1 \\
&+ K_2 \cdot K_2 \cdot 2 \\
&+ K_3 \cdot K_3 \cdot 3 \\
&= 3^2 \cdot 1 + 2^2 \cdot 2 + 1^2 \cdot 3 = 20
\end{aligned}
$$

Solution:

Copy and paste Mo's Algorithm and change add, remove and get_answer functions.

```
ll total = 0;
vector<int> cnt(1000001, 0);

void add(int val)
{
    total += (2 * (++cnt[val]) - 1) * val;
}

void remove(int val)
{
    total -= (2 * (cnt[val]--) - 1) * val;
}

ll get_ans()
{
    return total;
}
```

Example problem:

codeforces.com/problemset/problem/221/D

D. Little Elephant and Array

time limit per test: 4 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

The Little Elephant loves playing with arrays. He has array a, consisting of n positive integers, indexed from 1 to n. Let's denote the number with index i as a_i.

Additionally the Little Elephant has m queries to the array, each query is characterised by a pair of integers l_j and r_j $(1 \le l_j \le r_j \le n)$. For each query l_j, r_j the Little Elephant has to count, how many numbers x exist, such that number x occurs exactly x times among numbers $a_{l_j}, a_{l_j+1}, \ldots, a_{r_j}$.

Help the Little Elephant to count the answers to all queries.

Example: $\{3, 1, 2, 2, 3, 3, 7\}$

Range [0, 6] has three 3's, two 2's, and one 1.
Thus the answer is 3, since there are three numbers x such that x appears x times in the subarray.

(in this case the values are too big so we can't use an array to keep count) :

```
int res = 0;
unordered_map<int,int> cnt;

void add(int val)
{
    int Count = cnt[val]++;
    if (Count == val)
        res--;
    if (Count + 1 == val)
        res++;
}

void remove(int val)
{
    int Count = cnt[val]--;
    if (Count == val)
        res--;
    if (Count - 1 == val)
        res++;
}

int get_ans()
{
    return res;
}
```

Solution:

Same as the previous one.

Things to check out for more info on this topic

Cp-algorithms page

Errichto’s video

Usaco article

Codeforces blog post about faster query sorting method

My solutions to the example problems I showed:

- Holes
- Powerful Array
- Little Elephant and Array

Any Questions?