17 Range Queries

Introduction

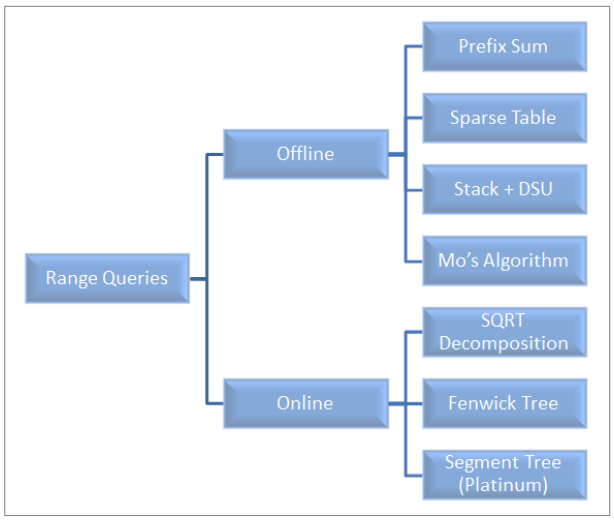
Range query problems generally requires to compute a value based on a subarray of an array. For example the answer of the range sum query (a, b) is the sum of values in range [a, b].

The typical range queries are

- sum of elements
- min/max element
- number of distinct elements
- XOR sum queries
- GCD (greatest common divisor) of elements
- LCM (least common multiple) of elements
- the most frequent element
- number of elements smaller than K
- counting some special elements

Queries can be on a static (immutable) array (without updates, **offline queries**) or on a dynamic array (with updates, **online queries**). At the same time, queries and updates can be on single element (**point query**, **point update**) of on a subarray (range query, **range update**).

Different algorithms and data structures are used according to whether the queries and updates are offline or online, and whether they are point queries/updates and range queries/updates. Online query solutions can be used for the same problem in an offline query solution, but not the other way around.



Range Sum Queries without Updates

Given an array *A[]* of integers of size *N*. We need to compute the sum of elements from index *i* to index *j*. The queries consisting of *i* and *j* index values will be executed multiple times.

Sample Input

```
5 2
2 4 -3 6 4
```

```
1 4
2 5
```

Sample Output

```
9
11
```

Solution with Prefix Sum Array

Precompute the prefix sum array *pre[]* with *N* elements. Let *pre[i]* stores sum of elements from *A[0]* to *A[i]*. To answer a *query (i, j)*, print *pre[j] – pre[i-1]*. This algorithm runs in *O(N)* space and *O(N+Q)* time.

Exercise: 2D Prefix Sum

How can you apply prefix sum solution in a 2D grid where each query is a sub-matrix?

Range Minimum/Maximum Queries (RMQ) without Updates

In computer science, a range minimum query (RMQ) solves the problem of finding the minimal value in a sub-array of an array of comparable objects. Range minimum queries have several use cases in computer science, such as the lowest common ancestor problem and the longest common prefix problem (LCP).

Solution 1) with Sparse Table Data Structure

Sparse Table is a data structure that allows answering range queries. It can answer most range queries in *O(logN)*, but its true power is answering range minimum queries (or equivalent range maximum queries). For those queries it can compute the answer in *O(1)* time.

The only drawback of this data structure is, that it can only be used on immutable arrays. This means, that the array cannot be changed between two queries. The complete data structure has to be recomputed if any element in the array changes.

Intuition

Any non-negative number can be uniquely represented as a sum of decreasing powers of two. This is just a variant of the binary representation of a number. E.g. $13=(1101)_2 = 8+4+1$. For a number x there can be at most $\lfloor Log2x \rfloor$ summands.

By the same reasoning any interval can be uniquely represented as a union of intervals with lengths that are decreasing powers of two. E.g. $[2, 14]=[2,9] \cup [10,13] \cup [14,14]$, where the complete interval has length 13, and the individual intervals have the lengths 8, 4 and 1 respectably. And also here the union consists of at most $\lceil \log2(\text{length of interval}) \rceil$ many intervals.

The main idea behind Sparse Tables is to precompute all answers for range queries with power of two length. Afterwards a different range query can be answered by splitting the range into ranges with power of two lengths, looking up the precomputed answers, and combining them to receive a complete answer.

Precomputation

We will use a 2-dimensional array for storing the answers to the precomputed queries. $st[i][j]$ will store the answer for the range $[i,i+2j-1]$ of length $2j$. The size of the 2-dimensional array will be $MAXN \times (K+1)$, where *MAXN* is the biggest possible array length. K has to satisfy $K \geq \lfloor Log2MAXN \rfloor$, because

2[Log2MAXN] is the biggest power of 2 range, that we have to support. For arrays with reasonable length ($\leq 10^7$ elements), K=25 is a good value.

```
int st[MAXN][K + 1]
```

Because the range `[i,i+2j-1] of length 2j` splits nicely into the ranges `[i,i+2j-1-1]` and `[i+2j-1,i+2j-1]`, both of `length 2j-1`, we can generate the table efficiently using dynamic programming:

```
for (int i = 0; i < N; i++)
     st[i][0] = f(array[i]);

for (int j = 1; j <= K; j++)
for (int i = 0; i + (1 << j) <= N; i++)
st[i][j] = f(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);
```

The function f will depend on the type of query. For range sum queries it will compute the sum, for range minimum queries it will compute the minimum. The time complexity of the precomputation is O(NlogN).

Range-Min Queries

These are the queries where the Sparse Table shines. When computing the minimum of a range, it doesn't matter if we process a value in the range once or twice. Therefore instead of splitting a range into multiple ranges, we can also split the range into only two overlapping ranges with power of two length. E.g. we can split the range [1,6] into the ranges [1,4] and [3,6]. The range minimum of [1,6] is clearly the same as the minimum of the range minimum of [1,4] and the range minimum of [3,6]. So we can compute the minimum of the range [L,R] with:

`min(st[L][j],st[R-2j+1][j]) where j=log2(R-L+1)`

This requires that we are able to compute `log2(R-L+1)` fast. You can accomplish that by precomputing all logarithms:

```
int log[MAXN + 1];
log[1] = 0;
for (int i = 2; i <= MAXN; i++)
log[i] = log[i / 2] + 1;
```

Afterwards we need to precompute the Sparse Table structure. This time we define f with `f(x,y)=min(x,y).`

```
int st[MAXN][K + 1];

for (int i = 0; i < N; i++)
     st[i][0] = array[i];

for (int j = 1; j <= K; j++)
for (int i = 0; i + (1 << j) <= N; i++)
st[i][j] = min(st[i][j - 1], st[i + (1 << (j - 1))][j - 1])
```

And the minimum of a range `[L, R]` can be computed with:

```
int j = log[R - L + 1];
int minimum = min(st[L][j], st[R - (1 << j) + 1][j]);
```

Time complexity for a Range Minimum Query is *O(1).*

```
//Range min query with no updates. Sparse Table implementation
//O(nlgn) precalculation, O(1) for a query. O(nlgn) space

#include "pch.h"
#include <vector>
#include <algorithm>
#include <fstream>

using namespace std;

ifstream cin("1.in");
ofstream cout("1.out");

int N, Q;
vector<int> numbers;
vector<vector<int>> st;      //sparse table
int M;                       //number of columns in the sparce table
//--------------------------------------
void readInput()
{
      cin >> N >> Q;
      numbers.resize(N);     //0 based
      for (int &num:numbers)
            cin >> num;
}
//--------------------------------------
//Sparse table stores index of minimum element
void constructSparseTable()
{
      int M = log2(N) + 1;              //# of columns

      st.resize(N, vector<int>(M));     //N rows and M columns

      //First column
      for (int i = 0; i < N; i++)
            st[i][0] = i;

      //Rest of the sparse table. Column manner filling.
      for (int j = 1; j < M; j++)                         //j is the column index
            for (int i = 0; i + pow(2,j) <= N; i++)       //i is the row index
            {
                  st[i][j] = st[i][j - 1];
                  if (numbers[st[i + j][j - 1]] < numbers[st[i][j]])
                        st[i][j] = st[i + j][j - 1];
            }
}
//--------------------------------------
void answerQueries()
{
      for (int i = 0; i < Q; i++)
      {
            int start, end;
            cin >> start >> end;
            start--, end--;             //input vector is 1 based. Subtract 1.

            int len = end - start + 1;
            int k = log2(len);
            int res = numbers[st[start][k]];
            res = min(res, numbers[st[start + len - pow(2, k)][k]]);
            cout << res << endl;
      }
}
//--------------------------------------
int main()
{
      readInput();
      constructSparseTable();
      answerQueries();
}
```

```
/*
Sample Input
7 3
2 4 -3 6 1 7 4
1 4
6 7
4 6

Sample Output
-3
4
1
*/
```

Solution 2) with Stack and DSU Data Structures

We are given an array a[] and we have to compute some minima in given segments of the array.

The idea to solve this problem with DSU is the following: We will iterate over the array and when we are at the ith element we will answer all queries (L, R) with R == i. To do this efficiently we will keep a DSU using the first i elements with the following structure: the parent of an element is the next smaller element to the right of it. Then using this structure the answer to a query will be the a[find_set(L)], the smallest number to the right of L.

This approach obviously only works offline, i.e. if we know all queries beforehand.

It is easy to see that we can apply path compression. And we can also use Union by rank, if we store the actual leader in an separate array.

```
//Consider that DSU data structure has been implemented here.

struct Query {
	int L, R, idx;
};

vector<int> answer;
vector<vector<Query>> container
```

container[i] contains all queries with R == i.

```
stack<int> s;
for (int i = 0; i < n; i++) {
	while (!s.empty() && a[s.top()] > a[i]) {
		parent[s.top()] = i;   //DSU
		s.pop();
	}
	s.push(i);
	for (Query q : container[i]) {
		answer[q.idx] = a[find_set(q.L)]; //DSU
	}
}
```

MO's Algorithm for Offline Queries

Introduction

Mo's algorithm is a generic idea. It applies to the following class of problems:

> You are given array Arr of length N and Q queries. Each query is represented by two numbers L and R, and it asks you to compute some function Func with subarray Arr[L..R] as its argument.

For the sake of brevity we will denote Func([L, R]) as the value of Func on subarray Arr[L..R].

If this sounds too abstract, let's look at specific example:

> There is an integer array Arr of length N and Q queries. For each i, query #i asks you to output the sum of numbers on subarray [Li, Ri], i.e. Arr[Li] + Arr[Li + 1] + … + Arr[Ri].

Here we have Func([L, R]) = Arr[L] + Arr[L + 1] + ... + Arr[R].

This does not sound so scary, does it? You've probably heard of solutions to this problem using Segment Trees or Binary Indexed Trees, or even prefix sums.

Mo's algorithm provides a way to **answer all queries in O((N + Q) * sqrt(N) * F)** time with at least O(Q) additional memory. Meaning of F is explained below.

The algorithm is applicable if all following conditions are met:

1. Arr is not changed by queries;
2. All queries are known beforehand (techniques requiring this property are often called "offline algorithms");
3. If we know Func([L, R]), then we can compute Func([L + 1, R]), Func([L - 1, R]), Func([L, R + 1]) and Func([L, R - 1]), each in **O(F)** time.

Due to constraints on array immutability and queries being known, Mo's algorithm is inapplicable most of the time. Thus, knowing the method can help you on rare occasions. But. Due to property #3, the algorithm can solve problems that are unsolvable otherwise. If the problem was meant to be solved using Mo's algorithm, then you can be 90% sure that it cannot be accepted without knowing it. Since the approach is not well-known, in situations where technique is appropriate, you will easily overcome majority of competitors.

Basic overview

We have Q queries to answer. Suppose we answer them in order they are asked in the following manner:

```
for i = 0..Q - 1:
      L, R = query #i
      for j = L..R:
            do some work to compute Func([L, R])
```

This can take Ω(N * Q) time. If N and Q are of order 105, then this would lead to time limit exceeded.

But what if we answer queries in different order? Can we do better then?

Definition:

Segment [L, R] is a continuous subarray Arr[L..R], i.e. array formed by elements Arr[L], Arr[L + 1], …, Arr[R]. We call L **left endpoint** and R **right endpoint** of segment [L, R]. We say that index i belongs to segment [L, R] if L ≤ i ≤ R.

Notation

Throughout this tutorial "x⁄y" will mean "integer part of x divided by y". For instance, 10⁄4 = 2, 15⁄3 = 5, 27⁄8 = 3;

By "sqrt(x)" we will mean "largest integer less or equal to square root of x". For example, sqrt(16) = 4, sqrt(39) = 6;

Suppose a query asks to calculate Func([L, R]). We will denote this query as [L, R] - the same way as the respective argument to Func;

Everything is 0-indexed.

We will describe Mo's algorithm, and then prove its running time.

The approach works as follows:

1. Denote BLOCK_SIZE = sqrt(N);
2. Rearrange all queries in a way we will call "Mo's order". It is defined like this: [L1, R1] comes earlier than [L2, R2] in Mo's order if and only if:
    a) L1⁄BLOCK_SIZE < L2⁄BLOCK_SIZE
    b) L1⁄BLOCK_SIZE == L2⁄BLOCK_SIZE && R1 < R2
3. Maintain segment [mo_left, mo_right] for which we know Func([mo_left, mo_right]). Initially, this segment is empty. We set mo_left = 0 and mo_right = -1;
4. Answer all queries following Mo's order. Suppose the next query you want to answer is [L, R]. Then you perform these steps:
    a) while mo_right is less than R, extend current segment to [mo_left, mo_right + 1];
    b) while mo_right is greater than R, cut current segment to [mo_left, mo_right - 1];
    c) while mo_left is greater than L, extend current segment to [mo_left - 1, mo_right];
    d) while mo_left is less than L, cut current segment to [mo_left + 1, mo_right].
5. This will take O((|left - L| + |right - R|) * F) time, because we required that each extension\deletion is performed in O(F) steps. After all transitions, you will have mo_left = L and mo_right = R, which means that you have successfully computed Func([L, R]).

Example Problem

Given an array of size N. All elements of array <= N. You need to answer M queries. Each query is of the form L, R. You need to answer the count of values in range [L, R] which are repeated at least 3 times.

Example: Let the array be {1, 2, 3, 1, 1, 2, 1, 2, 3, 1} (zero indexed)

Query: L = 0, R = 4. Answer = 1. Values in the range [L, R] = {1, 2, 3, 1, 1} only 1 is repeated at least 3 times.

Query: L = 1, R = 8. Answer = 2. Values in the range [L, R] = {2, 3, 1, 1, 2, 1, 2, 3} 1 is repeated 3 times and 2 is repeated 3 times. Number of elements repeated at least 3 times = Answer = 2.

SQRT Decompositions for Online Queries

Sqrt (or Square Root) Decomposition Technique is one of the most common query optimization technique used by competitive programmers. This technique helps us to reduce Time Complexity by a factor of **sqrt(n).**

The key concept of this technique is to decompose given array into small chunks specifically of size sqrt(n).

Let's say we have an array of n elements and we decompose this array into small chunks of size sqrt(n). We will be having exactly sqrt(n) such chunks provided that n is a perfect square. Therefore, now our array on n elements is decomposed into sqrt(n) blocks, where each block contains sqrt(n) elements (assuming size of array is perfect square).

Let's consider these chunks or blocks as an individual array each of which contains sqrt(n) elements and you have computed your desired answer(according to your problem) individually for all the chunks. Now, you need to answer certain queries asking you the answer for the elements in range l to r(l and r are starting and ending indices of the array) in the original n sized array.

As we have already precomputed the answer for all individual chunks and now we need to answer the queries in range l to r. Now we can simply combine the answers of the chunks that lie in between the range l to r in the original array. So, if we see carefully here we are jumping sqrt(n) steps at a time instead of jumping 1 step at a time as done in naive approach. Let's just analyze its Time Complexity and implementation considering the below problem :

Problem :

```
Given an array of n elements. We need to answer q
queries telling the sum of elements in range l to
r in the array. Also the array is not static i.e
the values are changed via some point update query.
```

Range Sum Queries are of form : **Q l r**, where l is the starting index r is the ending index

Point update Query is of form : **U idx val**, where idx is the index to update val is the updated value

```
// C++ program to demonstrate working of Square Root
// Decomposition.
#include "iostream"
#include "math.h"
using namespace std;

#define MAXN 10000
#define SQRSIZE  100

int arr[MAXN];              // original array
int block[SQRSIZE];         // decomposed array
int blk_sz;                 // block size

// Time Complexity : O(1)
void update(int idx, int val)
{
      int blockNumber = idx / blk_sz;
      block[blockNumber] += val - arr[idx];
      arr[idx] = val;
}

// Time Complexity : O(sqrt(n))
int query(int l, int r)
{
      int sum = 0;
```

```
	while (l < r and l%blk_sz != 0 and l != 0)
	{
		// traversing first block in range
		sum += arr[l];
		l++;
	}
	while (l + blk_sz <= r)
	{
		// traversing completely overlapped blocks in range
		sum += block[l / blk_sz];
		l += blk_sz;
	}
	while (l <= r)
	{
		// traversing last block in range
		sum += arr[l];
		l++;
	}
	return sum;
}

// Fills values in input[]
void preprocess(int input[], int n)
{
	// initiating block pointer
	int blk_idx = -1;

	// calculating size of block
	blk_sz = sqrt(n);

	// building the decomposed array
	for (int i = 0; i < n; i++)
	{
		arr[i] = input[i];
		if (i%blk_sz == 0)
		{
			// entering next block
			// incementing block pointer
			blk_idx++;
		}
		block[blk_idx] += arr[i];
	}
}

// Driver code
int main()
{
	// We have used separate array for input because
	// the purpose of this code is to explain SQRT
	// decomposition in competitive programming where
	// we have multiple inputs.
	int input[] = { 1, 5, 2, 4, 6, 1, 3, 5, 7, 10 };
	int n = sizeof(input) / sizeof(input[0]);

	preprocess(input, n);

	cout << "query(3,8) : " << query(3, 8) << endl;
	cout << "query(1,6) : " << query(1, 6) << endl;
	update(8, 0);
	cout << "query(8,8) : " << query(8, 8) << endl;
	return 0;
}
```

Output:

```
query(3,8) : 26
query(1,6) : 21
query(8,8) : 0
```

Exercise: SQRT Decomposition for Range Updates with Lazy Propagation

Consider that updates are given in the range like add or subtract a to all the elements in the range l to r. One of its solution would be to update all the elements of the range one by one (point update) but its complexity would be O(N*sqrt(N)). So, to handle such situations here comes Lazy Propagation.

Lazy propagation is range update optimization (usually in a segment tree solution) implemented that performs range updates in O(sqrt(N)) time.

Consider the range sum problem. Given an array of integers and q queries of two types. In update query, we need to add x to all the elements in the range l to r. In second type of query, we need to find the sum of the elements in a given range say l to r.

The idea behind the algorithm is don't not update a node until needed which will avoid the repeated sharing.

How can you apply the lazy propagation technique in sqrt decomposition solution not only for range sum queries but also for other type of queries?

Hint: Use a lazy array where each element stores the lazy value of a sqrt block.

Useful Links and References

https://www.geeksforgeeks.org/range-sum-queries-without-updates/
https://en.wikipedia.org/wiki/Range_minimum_query
https://cp-algorithms.com/data_structures/sparse-table.html
https://www.hackerearth.com/practice/notes/mos-algorithm/
https://blog.anudeep2011.com/mos-algorithm/
https://www.geeksforgeeks.org/sqrt-square-root-decomposition-technique-set-1-introduction/
https://medium.com/nybles/understanding-range-queries-and-updates-segment-tree-lazy-propagation-and-mos-algorithm-d2cd2f6586d8#:~:text=Lazy%20propagation%20is%20range%20update,the%20range%20l%20to%20r.