Square Root Decomposition

Spencer Compton

CPSI 2018: Squires Lecture 1

1 Introduction

Much of the interesting things we can do with Square Root Decomposition boil down to a few simple mathematical statements:

- $N = \sqrt{N} \times \sqrt{N}$
- $\sqrt{N} = \frac{N}{\sqrt{N}}$

2 $\sqrt{N}$ Bucketing

Consider a problem where we are given an array A of size N and want to support the following queries:

- $Add(i, X)$: Add X to $A[i]$
- $Sum(L, R)$: Return the sum of all $A[i]$ where $L <= i <= R$

We could approach this problem with a BIT (also known as Fenwick Tree) or Segment Tree to support both queries in $O(\log N)$, but we will ignore these approaches this lecture. Using prefix sums or brute force would allow us to support one of these queries in $O(N)$ and the other in $O(1)$, but we can use Square Root Decomposition to support both queries in $O(\sqrt{N})$.

We can divide A into B buckets of size S. We can use the equation $B = \frac{N}{S}$ to show that if $S = \sqrt{N}$ then $B = \sqrt{N}$. If we use this idea of bucketing, we can break our Sum queries into three parts.

1. At most S elements that are part of buckets not completely included in the query.
2. At most B buckets that are completely included in the query.
3. At most S more elements that are part of buckets not completely included in the query.

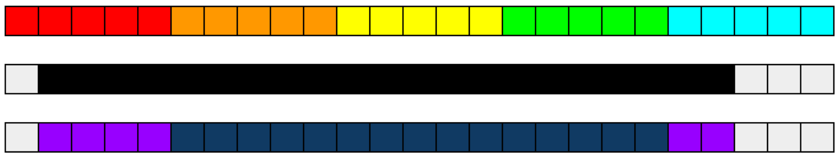

Figure 1: Visualization of a query $[1, 21]$ on an A where $N = 25$ (0-indexed).

If we implement our *Add* queries such that we always store the sum of elements in each bucket, then we can answer *Sum* queries in $O(B + S)$ or $O(\sqrt{N})$. With our current queries, we can implement *Add* in $O(1)$. However, what if we change the *Add* query to be the following:

- $Add(L, R, X)$: Add X to all $A[i]$ where $L <= i <= R$

We break down our new *Add* queries the same way we broke down our *Sum* queries. We manually add X to the elements in the first and third part of our query because this is $O(S)$ work, but need to be more clever for the buckets that are completely covered. For each bucket, we will store a "lazy" value that represents a value that should be added to every element inside that bucket, but our program is too lazy to do it right now (because it would be too much work). We will deal with *Add* queries simply by adding X to the lazy value of all buckets that are completely covered. For the $<= 2$ buckets that are partially included in our *Sum* query we will add their lazy value to all of their elements and then set the lazy values to zero. After this, we can execute the *Sum* query as previously described.

3 $\sqrt{N}$ Complexity Analysis

Sometimes we can use observations relating to the nature of $\sqrt{N}$ to design faster algorithms. One classic example is prime factorization. Consider the following question:

- Why can we stop checking for prime divisors of N after we check all integers in range $[2, \sqrt{N}]$?

In a similar vein, consider the following (less classic) question:

- Arup has a total of N dollars. He also has $<= N$ bills, each worth a positive integer amount of dollars. What is the maximal number of distinct values that can be written on Arup's bills?

4 Alternate Operations

Sometimes when we are solving a problem, queries fall into two different types based on some property. Queries that don't satisfy this property might be answerable in a smaller number of operations, while queries that do satisfy this property might be more expensive to answer but occur less frequently. In some cases, we can use ideas similar to the ones discussed in $\sqrt{N}$ Complexity Analysis to prove the rarity of these more expensive queries. Consider the following problem:

- There are N towns of beavers numbered from 1 to N in the descending order of their heights. No two towns have the same height. There are M canals connecting two different towns unidirectionally. The ith canal $(1 <= i <= M)$ flows from town S_i to town E_i. These canals flow from high towns to low towns. You cannot move against flow of the canals. Bitaro, a beaver, has N friends, one of whom lives in each of N towns. Bitaro is going to have parties Q times, inviting his friends. It is known for the jth $1 <= j <= Q$ party that Y_j friends are too busy to attend it. The jth party is held in town T_j, so his friends who cannot go from their towns to town T_j only via canals cannot attend it either. Other friends come to the party. Each friend come to the town where the party is held via canals. There may be several paths they can take. But Bitaro's friends love canals, so they must choice one of the paths which have the largest number of canals. Bitaro wonders how many canals the attendant who uses the largest number of canals uses. Bounds: $1 <= N <= 10^5$, $0 <= M <= 2 \times 10^5$, $1 <= Q <= 10^5$, $Y_1 + Y_2 + ... + Y_Q <= 10^5$.

How does the problem become easier if no friend is ever busy? How can we move on from there?

5 $\sqrt{N}$ Buffer

Another idea common in Square Root Decomposition is breaking the queries themselves into $\sqrt{N}$ buckets. Answer Q of the following queries for an undirected graph in $O(Q\sqrt{Q})$:

1. Add edge (i, j)
2. Delete edge (i, j)
3. Return whether or not i and j are connected.

6 Mo's Algorithm

We can also be more clever with how we divide the queries into buckets, as is highlighted by Mo's Algorithm, where queries are broke into $\sqrt{N}$ buckets based on their starting value. Consider the following problem:

- Given an array A of size N support the following queries in $O((N+Q) \times \sqrt{N})$. $query(L, R)$: Return the number of distinct values in range $[L, R]$.

7 Practice Problems

JOI Spring Contest 2018: Day 3, Problem 2 Bitaro

https://oj.uz/problem/view/JOI18_bitaro

Codeforces Yandex.Algorithm 2011 Round 2, Problem D

http://codeforces.com/contest/86/problem/D

8 Additional Resources

http://acm.math.spbu.ru/ sk1/mm/lections/mipt2016-sqrt/mipt-2016-burunduk1-sqrt.en.pdf

https://blog.anudeep2011.com/mos-algorithm/