Square Root Decomposition - Not Just Blocking

Translation compiled by Kirito Feng and Dessertion

WANG YUETONG

CTSC 2014

Contents

Abstract

Square root decomposition has become more popular, and the time complexity of OI tasks has also diversified; it is no longer "simple" time complexities like $\mathcal{O}(N)$, $\mathcal{O}(N^2)$, $\mathcal{O}(N \log N)$. Square root decomposition appears in many data structures problems, and its most common forms are decomposing an array into blocks and Mo's algorithm. However, square root decomposition is by no means limited to the aforementioned algorithms, and can appear in many other algorithms, accompanied by novel ideas. Through in-depth study of these algorithms, it becomes possible to obtain a better time complexity than square root decomposition. This paper introduces several applications of square root decomposition in non-graph problems, and attempts to transplant this idea to obtain better algorithms.

§1 Sample Problems

§1.1 Easier Examples

§1.1.1 Example 1: Codeforces Round 80 Div 1 D

Given an array A of length N ($1 \leq N \leq 300\,000$), you have M ($1 \leq M \leq 300\,000$), queries of the form (x, y) which ask you to compute $A[x] + A[x+y] + A[x+2y] + A[x+3y] + \cdots + A[x+ky]$, where k is the largest integer such that $x + ky \leq N$. The time limit is 4 seconds.

This question is different from many questions we have seen the the past, as most data structures are good at queries on continuous intervals, however, this problem is asking for disjoint intervals. As such, traditional data structures such as segment trees are insufficient to solve this problem.

However, we can see that if we suppose all y values to be the same, then we can partition the array's indices by their residue modulo y. This then results in each query becoming a contiguous range, and you can maintain this with a standard data structure.

The problem is that we need to maintain such data structures for $y = 1$, $y = 2, 3, 4, \ldots$, all the way to N. But consider this: if y is relatively large, do we even need to preprocess it? Wouldn't brute force be sufficient? Each brute force pass takes $\mathcal{O}\left(\frac{N}{y}\right)$. Therefore, setting $K = \sqrt{N}$, we precompute the case $1 \leq y \leq K$, and brute force all $y > K$. Then the time complexity will be $\mathcal{O}(N\sqrt{N})$.

Let's review this question. It cannot be said that it's *not* a data structures problem, but it does require a usage of square root decomposition not typically seen in most data structure problems. Specifically, two constraints in this problem are mutually bound together in a "product" relationship: the number of elements visited in a brute force solution, and the number of queries we preprocess. We can bound both constraints, design separate algorithms for each case, and combine the two. Let's look at another simple application of this idea:

§1.1.2 Example 2: TopCoder SRM 589 Div 1 Level 3

Given a binary string of length N ($1 \leq N \leq 300$) and a positive integer M ($1 \leq M \leq N$), you can either flip any arbitrary bit, or the first kM bits, where k is an integer. What is the minimum number of operations required to obtain a binary string such that its prefix of length $N - M$ equals its suffix of length $N - M$?

Firstly, we observe that the length of each segment and the number of said segments are mutually bounded: their product is bounded by the length of the string. Thus we consider two cases, based on the value of $\sqrt{N}$.

Remark. I have no idea what they're talking about at this point, so this is my solution to the problem...

Firstly, we observe that since the ith bit must equal the $i+M$th bit, it must also equal the $i+2M$th bit, and so on. Thus each block of M bits must be equal to each other.

If $M \le \sqrt{N}$, then have many small blocks. Thus for each block, we can compute the minimum cost needed to obtain each representation, as well as the minimal cost. This takes $\mathcal{O}\left(\frac{N}{M} \cdot 2^{\sqrt{N}}\right)$ time. We can then use dynamic programming to determine the number of flips. Our dynamic programming state is which block we are currently on, and the bitmask that we have chosen. Our transition is to either pick the same bitmask, or reverse all blocks before the current one. This gives us an $\mathcal{O}\left(N \cdot 2^{\sqrt{N}}\right)$ solution.

If $M > \sqrt{N}$, then we have a small number of large blocks. We instead consider all subsets of the $\frac{N}{M}$ flips we can perform. As performing a flip twice cancels it out, each flip should be performed at most once, so we can represent this with a bitmask. Finally, we can solve for the string's representation with a greedy algorithm. For each bit, pick the one that results in less flips, as the ith bit is independent of the $i+1$th bit. This gives us an $\mathcal{O}\left(N \cdot 2^{\sqrt{N}}\right)$ solution.

§1.2 Further Applications

§1.2.1 Example 3: IOI 2009 Regions

Given a tree with N nodes, each node is assigned to one of R regions. You have Q queries of the form r_1 r_2, where you have to count the number of pairs (e_1, e_2) such that e_1 belongs to r_1, e_2 belongs to r_2, and e_1 is an ancestor of e_2.

$N, Q \le 200000, R \le 25000$.

We can solve this problem by binary searching over the Euler tour. There can only be $\frac{N}{C}$ regions that have a size greater than C. For each of these regions, we can preprocess all possible answers when said region is r_1 in $\mathcal{O}(N)$ per region, which gives us a time complexity of $\mathcal{O}\left(\frac{N^2}{C}\right)$.

When we answer a query, if r_1 is one of these $\frac{N}{C}$ regions, then we can answer the query in $\mathcal{O}(1)$. Otherwise, we can iterate all C of the nodes, and binary search for the number of nodes in its subtree that belong to r_2. This gives us a time complexity of $\mathcal{O}(C \log N)$.

Most people would be tempted to set $C = \sqrt{N}$, but this gives $\mathcal{O}(N\sqrt{N} + Q\sqrt{N}\log N)$, which is too slow. Instead, we set $C = \sqrt{\frac{N}{\log N}}$, so we have a final time complexity of $\mathcal{O}((N+Q)\sqrt{N \log N})$.

Remark. An alternate $\mathcal{O}((N+Q)\sqrt{N})$ solution also exists.

For each region with more than $\sqrt{N}$ nodes, we can preprocess the answer to all queries that include this region in $\mathcal{O}(N)$.

When we answer a query, if either region has more than $\sqrt{N}$ nodes, we can answer the query in $\mathcal{O}(1)$. Otherwise, we can iterate the $\mathcal{O}(\sqrt{N})$ nodes in r_1, and find the starting and end points of each node in the Euler tour, and then count the number of nodes in each interval in $\mathcal{O}(\sqrt{N})$ time.

Thus we obtain a time complexity of $\mathcal{O}((N+Q)\sqrt{N})$.

This example was harder than the previous two, and even by IOI standards was not very easy. We can take advantage of various different algorithms to ensure the time complexity is sufficient to pass, including using non-standard block sizes.

§2 More Square Root Decomposition Techniques

Remark. This translates to something like "other ways of finding two related things that mutually bound each other with their product", but that's one hell of a mouthful.

§2.1 Size Discrimination

This is different from the idea of splitting into blocks, as this one uses two different algorithms based on the size of a certain factor.

§2.1.1 Example 4

You are given an undirected graph with N nodes and M edges, $N, M \leq 100000$. All of the nodes are initially coloured black. You have two types of operations: flip the colour of a node (from black to white and vice-versa), or query the number of adjacent black nodes of a node.

Observe that in a completely random case, brute force will suffice, as most nodes have a very limited outdegree. While in theory, this is an $\mathcal{O}(N)$ algorithm, the small outdegree of most nodes means that it is likely to pass.

For nodes with a larger outdegree, we observe that there are fewer of them. We can choose $\sqrt{N}$ as our cutoff point, as this typically works, and can also be adjusted in the end when we are trying to optimize our time complexity. If the node has an outdegree less than $\sqrt{N}$, then we can just solve them with brute force. Otherwise, as there are only $\sqrt{N}$ nodes with outdegree larger than $\sqrt{N}$, we can iterate all of them every time we do an update, keeping track of the number of adjacent black nodes. Any query with one of these nodes can be answer in $\mathcal{O}(1)$.

This gives us an $\mathcal{O}(Q\sqrt{N})$ solution.

§2.2 An Easier Example

We will now talk about an easier, but still interesting, example problem.

§2.2.1 Example 5: JSOI 2013 Burning Bridge Plan

You are given an array $A_1, A_2, A_3, \dots A_N$ of $N \leq 100000$ values, where $1000 \leq A_i \leq 2000$. You are to choose a (possibly empty) subset of these. Let us say you choose K numbers, $A_{p_1}, A_{p_2}, A_{p_3}, \dots, A_{p_K}$, such that $p_1 < p_2 < p_3 < \cdots < p_k$. We define the cost of this subset as $1 \times A_{p_1} + 2 \times A_{p_2} + 3 \times A_{p_3} + \cdots + K \times A_{p_K}$.

This set of K numbers splits the remaining array into $K+1$ (possibly empty) segments. Let T_i denote the sum of the ith segment. Let us consider all $T_i > M$. We define the total cost to be $1 \times A_{p_1} + 2 \times A_{p_2} + 3 \times A_{p_3} + \cdots + K \times A_{p_K}$ plus the sum of all $T_i > M$. How can we choose the numbers such that the total cost is minimized?

Note the total sum is no more than 2×10^8.

We have an obvious $\mathcal{O}(N^3)$ algorithm. Let $F[i][j]$ denote the best answer if we consider $A[1, 2, \dots i]$ and set $K = j$.

$$F[i][j] = \begin{cases} \min(F[k][j-1] + A_i \times j) & \text{if } A_{k+1} + \cdots + A_{i-1} \le M \\ \min(F[k][j-1] + A_i \times j + A_{k+1} + \cdots + A_{i-1}) & \text{if } A_{k+1} + \cdots + A_{i-1} > M \end{cases}$$

for $0 \le k < i$.

Using a monoqueue, we can decrease the time complexity to $\mathcal{O}(N^2)$. However, as the original question has $N = 100000$, we need further optimizations.

We note that unusual limit $1000 \le A_i \le 2000$. The upper bound is normal, but the lower bound is unusual. We note that as we pick more A_i, the cost to do so increases, thus we cannot pick too many.

If we don't pick any any A_i, then the cost is upper bounded by $2000N$, as we have only 1 T_i.

If we pick k A_i, the total cost is at least $\frac{k(k+1)}{2}1000 \ge 500k^2$.

Thus for the answer to be optimal, we must satisfy $500k^2 \le 2000N \implies k \le 2\sqrt{N}$.

Thus there are only $\mathcal{O}(N\sqrt{N})$ total states, and thus we are done.

This problem is presented because much like the previous problems, it is solved in $\mathcal{O}(N\sqrt{N})$, but it is interesting where the $\mathcal{O}(\sqrt{N})$ factor comes from - it comes from the analysis of the lower bound.

This problem was motivated by another problem on TopCoder: You are given K points, divided into N layers. There are edges between some points on adjacent layers, and between some points on layers 1 and N. Find the maximum independent set of this graph. The constraints are $K \le 100$ and $10 \le N \le 100$.

To solve this problem, we need to use properties of bipartite graphs and matchings. However, if N is odd, then we do not have a bipartite graph. As $10 \le N$ and $K \le 100$, there must exists a layer with at most 10 points. Thus we can enumerate all subsets of this layer, converting it into a chain.

The takeaway is that if you can get the upper and lower bounds within a constant multiple of each other, then analysis will often lead to a square root-based solution.

§2.3 Memory Optimization

Memory is very rarely the bottleneck in OI problems, but it can still be helpful to look into memory optimizations.

§2.3.1 Example 6: Codeforces Round 79 Div 1E

You are given N candies and M stones. At every step, you can eat either a candy or a stone, as long as you leave at least one candy and at least one stone left. After each step, your score increases by $(A[N-i] + B[M-j])\text{mod}$, where i and j are the number of candies and stones you have eaten, respectively. Maximize your total score.

If we ignore the unusual memory limit, we can solve this with simple dynamic programming. Let $F[i][j]$ be the optimal answer after eating i candies and j stones. Then we obtain

$$F[i][j] = (A[N-i] + B[M-j]) \bmod P + \max(F[i-1][j], F[i][j-1]).$$

Then by tracking which transitions we picked, we can easily reconstruct the answer.

Note that if we store the Kth row of F, we can easily recompute the $k+1$th row of F. Thus we store every $\sqrt{N}$th row, thus we can reconstruct the answer in $\mathcal{O}(N^2)$ time and $\mathcal{O}(N\sqrt{N})$ memory.

§3 Square Root Factor Appearing Naturally In Problems

There are some times where we aren't intentionally trying to create a square root factor in our calculations, but it will appear nonetheless.

For example, finding the result of $\lfloor \frac{N}{i} \rfloor$ for all i such that $1 \leq i \leq N$. For this problem, there are at most $\mathcal{O}(\sqrt{N})$ different answers. It's not difficult to prove this result. For $i \leq \sqrt{N}$, there are at most $\sqrt{N}$ different answers; for $i > \sqrt{N}$, all of the answers must lie in the interval $[1, \sqrt{N}]$, so there are also at most $\sqrt{N}$ answers.

There are some uses for this behaviour.

§3.1 Example 7: POI 2007 Queries

There are $N \leq 50\,000$ queries. In each query, you are given 3 positive integers, a, b, d, and you are asked to find the number of pairs of integers (x, y) which satisfy $x \leq a, y \leq b, \gcd(x, y) = d$.

This is a math problem. Let's firstly simplify down the problem. Note that $\gcd(x, y) = d$ can be easily turned into $\gcd(x, y) = 1$, by changing a and b into $\lfloor \frac{a}{b} \rfloor$ and $\lfloor \frac{b}{d} \rfloor$. So this reduces to finding pairs of relatively prime numbers where one is $\leq a$, the other is $\leq b$.

This is very similar to a NOI 2010 problem, Energy Harvesting. In both cases, we can use the Principle of Inclusion-Exclusion to solve the problem. A rough sketch of the solution is as follows. Let $\mathrm{F}(x)$ represent how many pairs of numbers there exist such that one doesn't exceed a, one doesn't exceed b, and the GCD of the pair is a multiple of x. Then, it follows that $\mathrm{F}(x) = (\lfloor \frac{a}{x} \rfloor)(\lfloor \frac{b}{x} \rfloor)$. Now, using Inclusion-Exclusion, the answer equals $\mathrm{F}(1) - \mathrm{F}(2) - \mathrm{F}(3) - \mathrm{F}(5) + \mathrm{F}(6) - \mathrm{F}(7) \ldots$, where each term $\mathrm{F}(x)$ is multiplied by another function, $\mathrm{p}(x)$. $\mathrm{p}(x)$ here is the Mobius function.

Remark. Let $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$. Then we define $\mathrm{p}(n) = \mathrm{p}\left(p_1^{e_1}\right) \mathrm{p}\left(p_2^{e_2}\right) \cdots \mathrm{p}\left(p_k^{e_k}\right)$, where $p_1, p_2, \ldots, p_k$ are primes, $e_i \geq 1$ for $i = 1, 2, \ldots, k$ and

$$p\left(p_i^{e_i}\right) = \begin{cases} -1 & e_i = 1 \\ 0 & e_i > 1 \end{cases}$$

Now the problem boils down to quickly finding $\sum \mathrm{F}(i) \cdot \mathrm{p}(i)$. We saw earlier that $\lfloor \frac{N}{i} \rfloor$ had at most $\mathcal{O}(\sqrt{N})$ distinct values. As such, $\mathrm{F}(x)$ also only has at most $\mathcal{O}(\sqrt{N})$ distinct values.

For a segment of $\mathrm{F}(x)$ with equal values, we just need to find the sum of $\mathrm{p}(x)$ for that segment to get the answer for that segment. By maintaining the prefix sum of $\mathrm{p}(x)$ and calculating the values of $\mathrm{F}(x)$, we can find the answer in only $\mathcal{O}(\sqrt{N})$ time per query, and pass this problem.

Last year (2013), there was a CTSC paper about the Mobius function by 王迪. We did not go into depth about the mathematical properties of the Mobius function in this paper, it simply came up during our discussion of the ways the square root factor can naturally crop up in certain problems.