Acta et Commentationes, Exact and Natural Sciences ISSN: 2537-6284
Volume 2(14), 2022, Pages 101–105 E-ISSN: 2587-3644
CZU 004.627:519.61 DOI: 10.36120/2587-3644.v14i2.101-105

Finding N bits using $O(\frac{N}{\log N})$ sums

Sergiu Corlat, Veaceslav Guzun, and Victor Vorona

Abstract. The problem we are trying to solve sounds as follows: You are given N bits. Find the value of each bit. We will show a technique which enables finding the values of N bits using $O(\frac{N}{\log N})$ subsequence-sum queries. The algorithm consists of two phases: Constructing the queries for each layer and using the queries for a particular layer to get the value of every bit. We described the following technique in this blog [1], which was inspired by this article [2]. It should be noted that this number of queries is indeed the optimal one for finding all N bits of a binary array, since each subsequence-sum queries offers us at most $\log_2 N$ bits of information.
Keywords: binary array, query problem, divide et impera, optimization.

Aflarea a N biţi folosind $O(\frac{N}{\log N})$ sume

Rezumat. Problema pe care încercăm să o rezolvăm sună astfel: vi se oferă N biți. Găsiți valoarea fiecărui bit. Vom prezenta o tehnică care permite aflarea valorilor a N biți folosind $O(\frac{N}{\log N})$ interogări de sume pe subsecvențe. Algoritmul se divide în două faze: construirea interogărilor pentru fiecare strat și utlizarea acestora pentru un strat particular pentru a obține valoarea fiecărui bit. Am descris această tehnică în articolul de blog [1], care dezvoltă rezultatele din articolul lui Zhenting Zhu din universitatea Tsinghua [2]. Trebuie să mentionăm faptul că acest număr de interogări este într-adevăr optimal pentru găsirea tuturor N biților unui șir binar, deoarece fiecare interogare de sumă de subsecvențe ne oferă cel mult $\log_2 N$ biți de informație.
Cuvinte-cheie: șir binar, problemă de interogare, divide et impera, optimizare.

1. Introduction

1.1. Core of the Problem

Finding the whole array of elements by knowing some information about some of its subsequences is a popular problem in computer science and can be found in many forms. In this case, we will explain how to find each element in a binary array (an array consisting only of zeroes and ones) by only being able to query the sums of some subsequences of it and try to minimize the number of queries we perform.

We will treat the problem as an interactive one. Initially, the only information about the binary array b we have is its size, and we can ask the interactor several questions in the following format:

What is the sum of the subsequence of the values on positions: $p_1, p_2, ..., p_k$. In other words, you will give the sequence $p_1, p_2, ..., p_k$ to the interactor and it will return you the value of $\sum_{i=1}^{k} b_{p_i}$.

After querying some number of sums, we should be able to tell the value of the element on each position.

1.2. Main Idea

The main idea of the algorithm involves a divide-and-conquer-like approach [3] which will work in two phases. In the first phase, the set of queries will be constructed, and the second phase will reconstruct the array values. We will show that it is possible to reconstruct the whole array using $O(\frac{N}{\log N})$ [4] well-built queries, and will also explain how the queries should be constructed.

2. Overview

2.1. Notations

In the coming explanation we will use the following notations:

- x_i – refers to the position i
- A – any capital letter (except S) refers to a set of points x_i
- $v_A = \sum b_{x_i}$ for $x_i \in A$
- k – the layer we are currently considering
- S_i – A set of queries

We will also use 0-indexing when talking about the array's elements' positions.

2.2. Explanation

The idea is to use a divide-and-conquer-like approach but in two phases. The first phase will be the construction of the queries we will ask at the end and the second phase will reconstruct the array of elements by having the answers to the relevant subsequences already obtained after the first phase.

The first phase

As it is a divide-and-conquer-like idea, we are going to work with layers. Let's say that for the k^{th} layer we use 2^k queries and that by using them we can find out the value of f_k elements.

Using the idea that is described below we will be able to make the following recurrence possible: $f_{k+1} = 2 \cdot f_k + 2^k - 1$.

Firstly, we will need to set our base case, which is $k = 0$. So, for $k = 0$, we will have $f_0 = 1$ and the query set will be 1. This means we will find out the value of a single element using a query.

Now, what we are trying to achieve in order to make the recurrence possible is to form the new block (f_{k+1}), using two blocks of size f_k, and find $2^k - 1$ additional elements in the process. Let's say k_1 will denote the first block of the f_k elements we will use, and k_2 the second such block.

The first query is used to get the sum on $[f_k, 2 \cdot f_k)$ – the sum of the second block. Then we add two new queries for each **non-last** query in S_{k_1} and S_{k_2}. First one is $S_{k_1}[i] \cup S_{k_2}[i]$. Second one is $S_{k_1}[i] \cup ([f_k, 2 \cdot f_k)/S_{k_2}[i]) \cup x_{(2 \cdot f_k + i)}$.

The last query is for the entire range $[0, f_{k+1})$. It's easy to see that now, we have used exactly 2^{k+1} queries. Now, why don't we lose any value in the process? And how will we be able to recursively [5] obtain the elements back? This will be clear in the second phase of the algorithm.

The second phase

Having answered all the S_{k+1} queries, we can calculate all the $v_{S_{k_1}[i]}$ and $v_{S_{k_2}[i]}$.

Now, when we reach a k with a value of $f_k >= n$ we can stop there. Let's assume $n = f_k$ since it will be easier to work with it (when n is smaller than f_k we can just think of it as appending $f_k - n$ zeroes at the end since they won't influence the sum at all). Using the set of queries responsible for the k^{th} layer we can in fact now reconstruct the whole sequence, recursively going from the k^{th} layer to the $(k-1)^{th}$ one (but consider each layer can have multiple blocks).

First of all, the only information relevant for the k^{th} layer are:

- The query set for the corresponding block of the corresponding layer.
- The block we are currently at (can be dealt with using an offset value in the recursion).

So we will store them when going recursively.

Firstly, let's set our base case: $k = 0$. We are now sure that only one element is responsible for this block from this layer, so we can just set the value of the b_x-th bit (where x is some offset value we use to keep track of the block) to v_{S_0} (since that's the sum for a single element which we've seen at the build-up).

Now, since the base case is already dealt with, here's how we will go to the $(k-1)^{th}$ layer:

We will need to reconstruct the previous query sets for the first block and the second block of size f_{k-1}, and set the values for the other $2^{(k-1)} - 1$ values respectively (since they aren't part of any of the blocks they shouldn't be part of the recursion either).

Let's denote the numbers of 1-s in $[f_k, 2 \cdot f_k)$ with c. It's obvious that $c = v_{S[0]}$. We will now go through every pair of queries, starting from 1. That means we will be analyzing queries $S_1[i] \cup S_2[i]$ and $S_1[i] \cup ([f_{k-1}, 2 \cdot f_{k-1})/S_2[i]) \cup x_{(2 \cdot f_{k-1}+i)}$.

- $v_{S[2 \cdot i+1]} = v_{S_1[i]} + v_{S_2[i]}$
- $v_{S[2 \cdot i+2]} = v_{S_1[i]} + c - v_{S_2[i]} + b_{2 \cdot f_{k-1}+i}$

In this case we will calculate 3 values: $v_{S_1[i]}, v_{S_2[i]}, b_{2 \cdot f_{k-1}+i}$.

- $v_{S_1[i]} = \lfloor \frac{v_{S[2 \cdot i+1]} + v_{S[2 \cdot i+2]} - c}{2} \rfloor$
- $v_{S_2[i]} = \lceil \frac{v_{S[2 \cdot i+1]} - v_{S[2 \cdot i+2]} + c}{2} \rceil$
- $b_{2 \cdot f_{k-1}+i} = (v_{S[2 \cdot i+1]} + v_{S[2 \cdot i+2]} - c) \wedge 1$

The only remaining queries to answer are $v_{S_1[2^{k-1}]}$ and $v_{S_2[2^{k-1}]}$ as they were not added in S.

- $v_{S_2[2^{k-1}]} = c = v_{S[0]}$
- $v_{S_1[2^{k-1}]} = v_{S[2^k]} - c - \sum_{i=0}^{2^{k-1}-1} b_{2 \cdot f_{k-1}+i}$

After calculating this, we could use the divide-and-conquer property specified earlier and go down a layer. We are going to do this recursively from layer k til layer 0. The last layer will consist of only 1 bit and only 1 sum, the value of said bit.

3. Visual Representation

Representation of how the algorithm works for $k = 3$ (Figure 1). Here is the color coding we used:

(1) The green squares represent the queries responsible for the first block of the previous layer.
(2) The orange squares represent the queries responsible for the second block of the previous layer.
(3) The blue squares represent the queries that query the whole second block excluding the elements from the query of the second block.
(4) The red squares represent the last $2^k - 1$ bits.

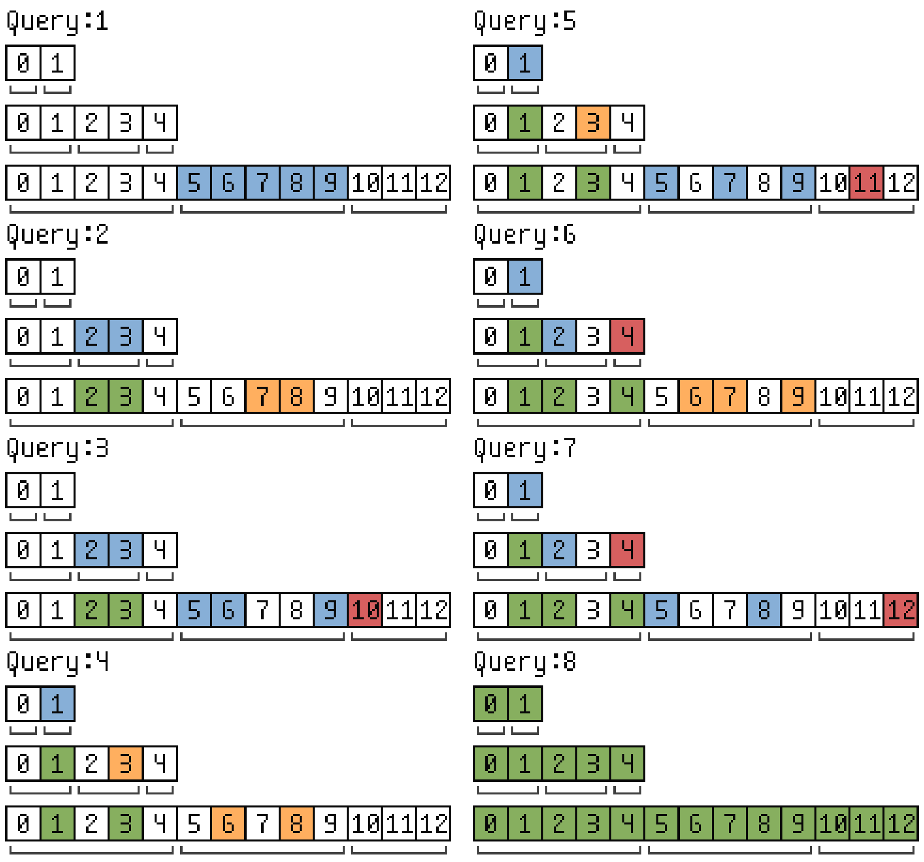

Figure 1. The second phase for $k = 3$

References

[1] http://codeforces.com/blog/entry/105188

[2] http://codeforces.com/blog/entry/82924

[3] Knuth, Donald *The Art of Computer Programming: Volume 3 Sorting and Searching.* (1998). p. 159. ISBN 0-201-89685-0.

[4] Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford. *Introduction to Algorithms (Third ed.).* MIT. (2009). p. 53

[5] Dijkstra, Edsger W. ”Recursive Programming”. *Numerische Mathematik.* (1960). 2 (1): 312–318.

Received: November 03, 2022 *Accepted: December 21, 2022*

(Corlat Sergiu) Moldova State University, 60 Alexei Mateevici St., Chişinău, Moldova
E-mail address: `sergiu.corlat@gmail.com`

(Guzun Veaceslav, Vorona Victor) Liceul Teoretic Orizont