Sparse table

39 CodeMonk Code Monk Sparse Table Data Structures Range query

Introduction

Imagine you are watching a movie and your friend assures you there is nothing interesting between 00:10 and 01:01 (hh:mm format). Also, he/she retells you the story from that time span. Now you don't need to watch it - and you've just saved 51 minutes of your precious time! Moreover, you can continue to watch right from the interesting part.

Now imagine you have many friends. Each of them has watched some portion of the film, and if combined, they've watched the entire movie. Then you don't have to go to the movie theater, provided that your friends are eloquent enough. You just ask them about scenario, and they retell it to you, piece by piece.

After this lyrical digression, let's move to our competitive programming world. Sparse table is a data structure. It often serves as a substitute for segment tree in case of immutable data.

Say you have an array **Arr** and you want to perform some queries. Each query should compute function **F** over subarray [L, R]: **F(Arr_L, Arr_{L+1}, ..., Arr_R)**. With sparse table, you can do **each query in O(log(N))** (N is the size of Arr), with **initial O(N * log(N)) preprocessing**.

Sparse table can be applied if and only if:

1. Arr is *immutable* (i.e. queries do not change it);
2. Function F is *associative*: F(a, b, c) = F(F(a, b), c) = F(a, F(b, c)).

Sparse table is easy to code and it is quite fast. Still the problems with completely immutable data are somewhat rare. I find sparse table one of my personal favourite approaches.

The method

Let's go back to the preface about friends and movies. It would be very convenient to have such friends. They would retell you needed information, and you just have to combine it to get the "big picture".

Imagine an array Table, such that **Table[i][j] = F(Arr_i, Arr_{i+1}, ..., Arr_{i+2^j-1})**.

You can view every cell of that array as a "friend", who has computed function F over small portion of the array. Each portion has a size, which is a **power of two**. We will show later how it matters, and now let's take a look at a more or less specific example.

Suppose you are asked to compute $F(Arr_i, Arr_{i+1}, \ldots, Arr_{i+12})$. Then you can do the following:

1. Ask friend at Table[i][3] about $F(Arr_i, Arr_{i+1}, \ldots, Arr_{i+7})$. Call this value x_1;
2. Ask friend at Table[i + 8][2] about $F(Arr_{i+8}, Arr_{i+9}, \ldots, Arr_{i+11})$. Call this value x_2;
3. Ask friend at Table[i + 12][0] about $F(Arr_{i+12})$. Call this value x_3;
4. Compute $F(x_1, x_2, x_3)$ to get the answer.

Notice that it took only 4 actions to compute F over subarray of size 13. However, in this case you knew which friends to ask in order to minimize overall number of questions.

You may already know that every non-negative integer number has **binary representation** (in numeral system base 2). For example:

$13_{10} = 1101_2$

$15_{10} = 1111_2$

$68_{10} = 1000100_2$

$198_{10} = 11000110_2$

This representation is unique for any given number. In other words, if you have numbers 2^0, 2^1, ..., 2^k (for some **k** ≥ 1, each number is presented once), then you can represent *any* integer N up to $2^0 + 2^1 + \ldots + 2^k = 2^{k+1} - 1$ as the sum of some of those numbers.
This can be done with the following pseudocode:

```
for i = k..0:
  if N >= 2^i: // 2^i means "two to the power of i"
    N -= 2^i
    print('add 2^i')
```

The strategy here is to greedily pick highest i such that $2^i \le N$ and subtract it from N. It works, because if we represent N as a k+1-bit binary number, then we add precisely those i, where i-th bit is set.
For example, take N = 113 and k = 7. Then N represented as k+1=8-bit binary number is 01110001. We iterate from i = 7 to 0:

i = 7, N = 113 = 01110001. Is $113 \ge 2^7 = 128$? No, then do nothing;

i = 6, N = 113 = 01110001. Is $113 \ge 2^6 = 64$? Yes, then we set N = N - 64 = 113 - 64 = 49;

i = 5, N = 49 = 00110001. Is $49 \ge 2^5 = 32$? Yes, then we set N = N - 32 = 49 - 32 = 17;

i = 4, N = 17 = 00010001. Is $17 \ge 2^4 = 16$? Yes, then we set N = N - 16 = 17 - 16 = 1;

i = 3, N = 1 = 00000001. Is $1 \ge 2^3 = 8$? No, then do nothing;

i = 2, N = 1 = 00000001. Is $1 \ge 2^2 = 4$? No, then do nothing;

i = 1, N = 1 = 00000001. Is $1 \ge 2^1 = 2$? No, then do nothing;

i = 0, N = 1 = 00000001. Is $1 \ge 2^0 = 1$? Yes, then we set N = N - 1 = 1 - 1 = 0.

The end. We conclude that $113 = 2^6 + 2^5 + 2^4 + 2^0$.

You can notice that we do not need to write down binary representation of N to make the algorithm work. It just serves as an illustrative proof.

We can use the above reasoning to **compute F(Arr_L, Arr_{L+1}, ..., Arr_R)** for given L and R. We query some of our friends in the following manner:

```
answer = ZERO
L' = L
for i=k..0:
  if L' + 2^i - 1 <= R:
    answer = F(answer, Table[L'][i]) // F is associative, so this operation is meaningful
    L' += 2^i
```

- **ZERO** is **neutral element** for calculating function F, such that F(ZERO, x) = F(x, ZERO) = x for any x.
- In the above code we assume that **$R - L + 1 < 2^{k+1}$** (we can choose such k, given bounds on N).
- The reasoning behind condition **"$L' + 2^i - 1 \leq R$"** is such: we want to know, whether we should ask friend at Table[L][i] for help. He\she accounts for subarray of 2^i elements: Arr_L, Arr_{L+1}, ..., Arr_{L+2^i-1}. If that subarray lies in [L, R], then we reach out to mentioned friend.

To ensure the algorithm is accurate, notice that on every iteration of *for* loop we have **answer = F(Arr_L, Arr_{L+1}, ..., $Arr_{L'-1}$)** (with the exception that if L' = L then answer is ZERO).

Also note that at every step we add 2^i only in those cases, where R - L + 1 (size of subarray from L to R) has **i-th bit set** in binary representation. That means that we're adding a total of R - L + 1 to L', which is initially equal to L.
This implies that after loop ends, we have L' = L + (R - L + 1) = R + 1. Thus, in the end answer = F(Arr_L, Arr_{L+1}, ..., Arr_R).

This algorithm for answering queries with Sparse Table works in O(k), which is **O(log(N))**, because we choose minimal k such that $2^{k+1} > N$.

Now we know how to answer queries if we have Table at hand. But **how do we get it**? This is the easiest way:

```
for i=0..N-1: // assuming Arr is indexed from 0
  Table[i][0] = F(Arr[i])
for j=1..k: // assuming N < 2^(k+1)
  for i=0..N-2^j:
    Table[i][j] = F(Table[i][j - 1], Table[i + 2^(j - 1)][j - 1])
```

Here we are building Table[i][j] only for i, j such that $i + 2^j < N$. We use the fact that $2^j = 2^{j-1} + 2^{j-1}$: if we know Table[i][j] for fixed j and all meaningful i, then we can derive Table[r][j +

1] for every meaningful r. How do we do that?

- Take some r, such that $r + 2^{j+1} < N$. Suppose we want to obtain $Table[r][j + 1] = F(Arr_r, Arr_{r + 1}, ..., Arr_{r + 2^{j + 1} - 1})$.
- Split the subarray into two parts:
  $P_1 = Arr_r, Arr_{r + 1}, ..., Arr_{r + 2^j - 1}$ of size $(r + 2^j - 1) - r + 1 = 2^j$
  and
  $P_2 = Arr_{r + 2^j}, Arr_{r + 2^j + 1}, ..., Arr_{r + 2^{j + 1} - 1}$ of size $(r + 2^{j + 1} - 1) - (r + 2^j) + 1 = 2^{j + 1} - 2^j = 2^j$.
- Note that $\mathbf{F(Arr_r, Arr_{r + 1}, ..., Arr_{r + 2^{j + 1} - 1}) = F(F(P_1), F(P_2))}$ by associativity of F.
- Since sizes of P_1 and P_2 are both powers of two, then we can find $F(P_1)$ in Table[r][j] and $F(P_2)$ in Table[$r + 2^j$][j]. Then **Table[r][j + 1] = F(Table[r][j], Table[$r + 2^j$][j])**, which is expressed in the inner loop above.

Outer loop runs in O(k), inner loop runs in O(N). Thus, in total we get O(N * k) = **O(N * log(N))** time complexity for Sparse Table creation.

Example problems

In this section we'll describe some well-known problems and solutions to them using Sparse Table.
Remember: to solve a problem with Sparse Table, you must ensure that

1. Queries do not change the data;
2. Function F is associative.

If this is the case, the implementation will usually look as follows:

1. Read the data;
2. Build sparse table;
3. Read queries one-by-one and answer them immediately.

Range sum query

> Provided with an integer array Arr of length N, answer Q queries:
>
> given L and R, $0 \le L, R < N$, find $\mathbf{Arr_L + Arr_{L + 1} + ... + Arr_R}$.
>
> N and Q are up to 10^5, $|Arr_i| \le 10^9$.

Here we have **F(x, y) = x + y**, with the neutral element ZERO being actual zero (i.e. **0**).
We set k to 16, because 2^{17} is larger than 10^5 (highest possible N), while 2^{16} is not.

The solution would look something like this (written in C++):

```
const int k = 16;
const int N = 1e5;
```

```
const int ZERO = 0; // ZERO + x = x + ZERO = x (for any x)

long long table[N][k + 1]; // k + 1 because we need to access table[r][k]
int Arr[N];

int main()
{
    int n, L, R, q;
    cin >> n; // array size
    for(int i = 0; i < n; i++)
        cin >> Arr[i];

    // build Sparse Table
    for(int i = 0; i < n; i++)
        table[i][0] = Arr[i];
    for(int j = 1; j <= k; j++) {
        for(int i = 0; i <= n - (1 << j); i++)
            table[i][j] = table[i][j - 1] + table[i + (1 << (j - 1))][j - 1];
    }

    cin >> q; // number of queries
    for(int i = 0; i < q; i++) {
        cin >> L >> R; // boundaries of next query, 0-indexed
        long long answer = ZERO;
        for(int j = k; j >= 0; j--) {
            if(L + (1 << j) - 1 <= R) {
                answer = answer + table[L][j];
                L += 1 << j; // instead of having L', we increment L directly
            }
        }
        cout << answer << endl;
    }
    return 0;
}
```

Notice that in C++ expression (1 << j) means 2^j.

Range minimum query

> Same as previous problem, but the query asks for **min(Arr_L, Arr_{L+1}, ..., Arr_R)**.

Here we have **F(x, y) = min(x, y)**. The neutral element **ZERO is 10^9 + 1**, because the problem tells us that $|Arr_i| \le 10^9$, and therefore any possible element of the array is less than the neutral element. This implies that min(x, ZERO) = min(ZERO, x) = x for any x that we can see in Arr.

Implementation (again in C++):

```
const int k = 16;
const int N = 1e5;
const int ZERO = 1e9 + 1; // min(ZERO, x) = min(x, ZERO) = x (for any x)

int table[N][k + 1]; // k + 1 because we need to access table[r][k]
int Arr[N];

int main()
{
    int n, L, R, q;
    cin >> n; // array size
    for(int i = 0; i < n; i++)
        cin >> Arr[i]; // between -10^9 and 10^9

    // build Sparse Table
    for(int i = 0; i < n; i++)
        table[i][0] = Arr[i];
    for(int j = 1; j <= k; j++) {
        for(int i = 0; i <= n - (1 << j); i++)
            table[i][j] = min(table[i][j - 1], table[i + (1 << (j - 1))][j - 1]);
    }

    cin >> q; // number of queries
    for(int i = 0; i < q; i++) {
        cin >> L >> R; // boundaries of next query, 0-indexed
        int answer = ZERO;
        for(int j = k; j >= 0; j--) {
            if(L + (1 << j) - 1 <= R) {
                answer = min(answer, table[L][j]);
                L += 1 << j; // instead of having L', we increment L directly
            }
        }
        cout << answer << endl;
    }
    return 0;
}
```

Range greatest common divisor query

> Provided with an integer array Arr of length N, answer Q queries:
>
> given L and R, $0 \le L, R < N$, find **gcd(Arr_L, Arr_{L+1}, …, Arr_R)**.
>
> N and Q are up to 10^5, $1 \le Arr_i \le 10^9$.

Here *gcd* stands for "greatest common divisor", i.e. gcd(x_1, x_2, …, x_p) = y if and only if:

1. x_i is divisible by y for every $1 \le i \le p$;
2. y is the greatest among numbers, which satisfy the above property;
3. if $x_i = 0$ for every $1 \le i \le p$, then y is undefined.

Here we have **F(x, y) = gcd(x, y)**. There is an algorithm for finding gcd of two numbers (called Euclid's algorithm), so you are not obligated to know how it is computed. C++ has this algorithm implemented in function *__gcd(x, y)*.
What about gcd(x, y, z)? How do we obtain that?
It turns out that gcd(x, y, z) = gcd(gcd(x, y), z) = gcd(x, gcd(y, z)). Intuitively, this happens because no matter how you order operands and take gcd-s, the greatest common divisor is still the same. Since now we know our function is associative, we can plug-in Sparse Table to solve the problem.
One more thing to notice: **ZERO = 0**, because 0 is divisible by any other number, therefore gcd(0, x) = gcd(x, 0) = x for x > 0.

Implementation (C++):

```
const int k = 16;
const int N = 1e5;
const int ZERO = 0; // gcd(ZERO, x) = gcd(x, ZERO) = x (for any x > 0)

int table[N][k + 1]; // k + 1 because we need to access table[r][k]
int Arr[N];

int main()
{
    int n, L, R, q;
    cin >> n; // array size
    for(int i = 0; i < n; i++)
        cin >> Arr[i]; // between 1 and 10^9

    // build Sparse Table
    for(int i = 0; i < n; i++)
        table[i][0] = Arr[i];
    for(int j = 1; j <= k; j++) {
        for(int i = 0; i <= n - (1 << j); i++)
            table[i][j] = __gcd(table[i][j - 1], table[i + (1 << (j - 1))][j - 1]);
    }
```

```
    cin >> q; // number of queries
    for(int i = 0; i < q; i++) {
        cin >> L >> R; // boundaries of next query, 0-indexed
        int answer = ZERO;
        for(int j = k; j >= 0; j--) {
            if(L + (1 << j) - 1 <= R) {
                answer = __gcd(answer, table[L][j]);
                L += 1 << j; // instead of having L', we increment L directly
            }
        }
        cout << answer << endl;
    }
    return 0;
}
```


Number of contiguous subarrays with gcd equal to 1

This problem itself is not about queries, but Sparse Table is still useful here. This is a more tricky application of the method, and might not be suitable for beginners.

> You have an array of integers Arr of length N. You must count number of pairs of integers (L, R) such that:
>
> 1. $0 \leq L \leq R < N$;
> 2. $gcd(Arr_L, Arr_{L+1}, ..., Arr_R) = 1$.
>
> N is up to 10^6.

Let's start with some observations:

1. If we brute-force every pair (L, R) satisfying restriction #1, it will take roughly N * (N - 1) / 2 operations, which is about $5 * 10^{11}$. It's too much to process in reasonable online-judge time;
2. Let $g = gcd(x, y)$, $f = gcd(x, y, z)$. Then $f \leq g$, because $f = gcd(g, z)$, and greatest common divisor of two integers can not be greater than any of them;
3. If we fix L and compute values $x_L = gcd(Arr_L)$, $x_{L+1} = gcd(Arr_L, Arr_{L+1})$, ..., $x_{N-1} = gcd(Arr_L, Arr_{L+1}, ..., Arr_{N-1})$, then we have $x_L \geq x_{L+1} \geq ... \geq x_{N-1}$ (it follows from the above point).

Hence, for each fixed L we can apply binary search to find **smallest R ≥ L** such that $gcd(Arr_L, Arr_{L+1}, ..., Arr_R) = 1$, because gcd is monotonous (subject to increasing the subarray). After we found such R, we know for sure that further extending subarray to the right provides gcd equal to 1. We count all such pairs (L, R') with N > R' ≥ R. There are N - R of them. This will take **O(N * log(N) * log(N))** time: one log(N) for binary search, other log(N) for inner sparse table computations.

Instead of binary searching we will maintain smallest such R similar to how we count gcd on a segment. **Be sure to understand how querying Sparse Table work in general before you continue reading.**

- Firstly, we set R = L;
- Then for i=k..0 we decide whether we need to move R further by 2^i units;
- Why would we need to move it at all? Since we are ultimately looking for R to satisfy $gcd(Arr_L, Arr_{L+1}, ..., Arr_R) = 1$, we are not interested if that gcd is greater than 1 (notice that it can not be less that 1 in any situation);
- And if the latter is the case, i.e. we found out that $gcd(Arr_L, Arr_{L+1}, ..., Arr_R, Arr_{R+1}, ..., Arr_{R+2^i-1}) > 1$, then we need to increase R by 2^i, because $gcd(Arr_L, Arr_{L+1}, ..., Arr_R) > 1$ for sure (remember, we have monotonous function). After that addition we continue to loop further. In the end we will obtain the desired R;
- This will work in O(log(N)) for fixed L, getting us to **O(N * log(N))** solution in total.

See the implementation for details:

```
const int k = 16;
const int N = 1e5;
const int ZERO = 0; // gcd(ZERO, x) = gcd(x, ZERO) = x (for any x > 0)

int table[N][k + 1]; // k + 1 because we need to access table[r][k]
int Arr[N];

int main()
{
    int n;
    cin >> n; // array size
    for(int i = 0; i < n; i++)
        cin >> Arr[i]; // between 1 and 10^9

    // build Sparse Table
    for(int i = 0; i < n; i++)
        table[i][0] = Arr[i];
    for(int j = 1; j <= k; j++) {
        for(int i = 0; i <= n - (1 << j); i++)
            table[i][j] = __gcd(table[i][j - 1], table[i + (1 << (j - 1))][j - 1]);
    }

    // main part of the solution
    long long answer = 0;
    for(int i = 0; i < n; i++) {
```

```
        int R = i; // we will move R forward as long as gcd(Arr_i, Arr_i+1, ..., Arr_R) != 1
        // or until R reaches n.

        int g = ZERO;
        for(int j = k; j >= 0; j--) {
            if(R + (1 << j) - 1 >= n)
                continue; // we do not want to exceed array size

            if(__gcd(g, table[R][j]) > 1) {
                // Even if we add 2^j more values, gcd is still > 1. Therefore,
                // we move R forward and update gcd appropriately.
                g = __gcd(g, table[R][j]);
                R += 1 << j;
            }
        }

        // In the end, either R = n or gcd(Arr_i, Arr_i+1, ..., Arr_R) = 1.
        answer += n - R;
    }

    cout << answer << endl;
    return 0;
}
```

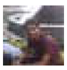

Afterword

The application of Sparse Table is not limited to arrays only. For example, you can use it on trees to find *Lowest Common Ancestor* in O(log(N)) per query. However, this is slightly off-topic (because knowledge of graphs, trees and depth-first search is needed) and I decided not to include it in this article.

COMMENTS (15) SORT BY: Relevance

Join Discussion...

Cancel Post



hardik agrawal Edited 9 months ago

very nice article... thanks :) would be helpful if lowest common ancestor using this technique is explained..

3 votes • Reply • Message • Permalink

Lawliet_Yagami Edited 4 months ago

The time complexity to query in your implementation is log*(n) (not to be confused with log(n), log*(n) is iterated logarithm) which can be reduced to O(1) by choosing 2 [L,R] values such that they cover all the nodes. Read this for more https://www.topcoder.com/community/data-science/data-science-tutorials/range-minimum-query-and-lowest-common-ancestor/#Lowest%20Common%20Ancestor%20(LCA)

0 votes • Reply • Message • Permalink

Mike Koltsov Author 3 months ago

"can be reduced to O(1) by choosing 2 [L,R] values such that they cover all the nodes" This can only be done for idempotent functions (`min` or `max` are good examples). But some functions do not allow such reduction: `sum`, for example.

I can come up with a test which gives Omega(log(n)) time complexity for a query. It's as simple as taking such L, R that R - L + 1 is `111111...1` in binary form. On each iteration, we will erase exactly one `1` from it.

1 vote • Reply • Message • Permalink

Lawliet_Yagami 3 months ago

Thaks for sharing! I didn't have that in mind! And, thanks for the amazing tutorial!

2 votes • Reply • Message • Permalink

maddela sai karthik 10 months ago

thanks for this awesome tutorial mike

2 votes • Reply • Message • Permalink

Tahanima Chowdhury a year ago

Thank You for this awesome tutorial. I have been struggling with Data Structure for quite a long time. Today I've started with Sparse table & came to know about this tutorial. I would really appreciate if you keep making tutorials like this specially of Data Structures.

1 vote • Reply • Message • Permalink

SANDEEP KUMAR 2 years ago

Nicely written,and Please give a tutorial on treaps..:)

0 votes • Reply • Message • Permalink

Mike Koltsov Author 2 years ago

Thank you! Unfortunately, treap is a data structure I'm not mastered yet.

0 votes • Reply • Message • Permalink

competitivecoder 2 years ago

Hey mike.It was very very well written article.I would love to see some more from you.

0 votes • Reply • Message • Permalink

Mike Koltsov Author 2 years ago

Sure, you can check the list of my articles here: https://www.hackerearth.com/notes/u/misha/

0 votes • Reply • Message • Permalink

competitivecoder 2 years ago

Thank you mike for all articles.Can you tell me one thing ?How do you choose 'k' here.If I have understood it well k is floor(logbase2(N)).Am I right or we have to calculate it by calculator or something before starting ?

0 votes • Reply • Message • Permalink

Mike Koltsov Author 2 years ago

You are right, k is somewhere about log_2(N). But it's easy to make a mistake:
1. Maybe you forget +1 or -1 somewhere and your k gets wrong
2. Maybe the log function behaves unexpectedly and returns 4.9999 instead of 5
So, my approach for choosing k is to look at the constraints for N. For example, if N can be as high as 10^6, then you should take the least k such that 2^(k + 1) >= 10^6. In this case, k = 19.
If you ask me, I do this manually each time.
▲ 0 votes ● Reply ● Message ● Permalink

competitivecoder 2 years ago
Thank you :-)
▲ 0 votes ● Reply ● Message ● Permalink

Ayush Chaturvedi 2 years ago
great work any body wants to practice can try this
https://www.hackerrank.com/contests/codeagon/challenges/sherlock-and-subarray-queries
▲ 0 votes ● Reply ● Message ● Permalink

Pulkit Mehta 10 months ago
I hope we can use this data structure if I consider base 3 instead of two and making required changes.
▲ 0 votes ● Reply ● Message ● Permalink

AUTHOR

Mike Koltsov
Saint Petersburg, Russia
3 notes

TRENDING NOTES

Python Diaries Chapter 3 Map | Filter | For-else | List Comprehension
written by Divyanshu Bansal

Bokeh | Interactive Visualization Library | Use Graph with Django Template
written by Prateek Kumar

Bokeh | Interactive Visualization Library | Graph Plotting
written by Prateek Kumar

Python Diaries chapter 2
written by Divyanshu Bansal

Python Diaries chapter 1
written by Divyanshu Bansal

more ...

ABOUT US

Blog

Engineering Blog

Updates & Releases

Team

Careers

In the Press

HACKEREARTH

API

Chrome Extension

CodeTable

Developer Profile

Resume

Get Badges

Campus Ambassadors

Get Me Hired

Privacy

Terms of Service

DEVELOPERS

AMA

Code Monk

Judge Environment

Solution Guide

Problem Setter Guide

Practice Problems

HackerEarth Challenges

College Challenges

College Ranking

Organise Hackathon

Hackathon Handbook

Competitive Programming

Open Source

EMPLOYERS

Developer Sourcing

Lateral Hiring

Campus Hiring

Hackathons

FAQs

Customers

REACH US

Ground Floor, Salarpuria Business Center,
4th B Cross Road, 5th A Block,
Koramangala Industrial Layout,
Bangalore, Karnataka 560095, India.

contact@hackerearth.com