



Range Minimum Query (Square Root Decomposition and Sparse Table)

We have an array $arr[0 \dots n-1]$. We should be able to efficiently find the minimum value from index L (query start) to R (query end) where $0 \leq L \leq R \leq n-1$. Consider a situation when there are many range queries.

4

Example:

```
Input: arr[] = {7, 2, 3, 0, 5, 10, 3, 12, 18};  
       query[] = [0, 4], [4, 7], [7, 8]
```

```
Output: Minimum of [0, 4] is 0  
        Minimum of [4, 7] is 3  
        Minimum of [7, 8] is 12
```

A **simple solution** is to run a loop from L to R and find minimum element in given range. This solution takes $O(n)$ time to query in worst case.

Another approach is to use **Segment tree**. With segment tree, preprocessing time is $O(n)$ and time to for range minimum query is $O(\log n)$. The extra space required is $O(n)$ to store the segment tree. Segment tree allows updates also in $O(\log n)$ time.

Can we do better if we know that array is static?

How to optimize query time when there are no update operations and there are many range minimum queries?

Below are different methods.

Method 1 (Simple Solution)

A Simple Solution is to create a 2D array `lookup[][]` where an entry `lookup[i][j]` stores the minimum

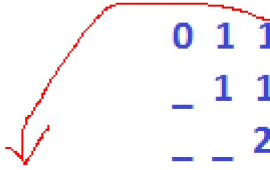


value in range $\text{arr}[i..j]$. Minimum of a given range can now be calculated in $O(1)$ time.

7	2	3	0	5	10	3	12	18
0	1	2	3	4	5	6	7	8

arr[]

Index of
minimum
in arr[0..2]



0	1	1	3	3	3	3	3	3
-	1	1	3	3	3	3	3	3
-	-	2	3	3	3	3	3	3
-	-	-	3	3	3	3	3	3
-	-	-	-	4	4	6	6	6
-	-	-	-	-	5	6	6	6
-	-	-	-	-	-	6	6	6
-	-	-	-	-	-	-	7	7
-	-	-	-	-	-	-	-	8

**lookup[i][j] stores
index of minimum
value in range
arr[i] ... arr[j]**

lookup[][]

```
// C++ program to do range minimum query in O(1) time with O(n*n)
// extra space and O(n*n) preprocessing time.
#include<bits/stdc++.h>
using namespace std;
#define MAX 500
```

```
// lookup[i][j] is going to store index of minimum value in
// arr[i..j]
int lookup[MAX][MAX];
```

```
// Structure to represent a query range
struct Query
{
    int L, R;
};
```

```
// Fills lookup array lookup[n][n] for all possible values of
// query ranges
```

```
void preprocess(int arr[], int n)
{
```

```
    // Initialize lookup[][] for the intervals with length 1
    for (int i = 0; i < n; i++)
        lookup[i][i] = i;
```

```
    // Fill rest of the entries in bottom up manner
```

```
    for (int i=0; i<n; i++)
    {
```

```
        for (int j = i+1; j<n; j++)
```

```
            // To find minimum of [0,4], we compare minimum of
            // arr[lookup[0][3]] with arr[4].
```

```
            if (arr[lookup[i][j - 1]] < arr[j])
                lookup[i][j] = lookup[i][j - 1];
```

```
            else
                lookup[i][j] = j;
```



```

    }
}

// Prints minimum of given m query ranges in arr[0..n-1]
void RMQ(int arr[], int n, Query q[], int m)
{
    // Fill lookup table for all possible input queries
    preprocess(arr, n);

    // One by one compute sum of all queries
    for (int i=0; i<m; i++)
    {
        // Left and right boundaries of current range
        int L = q[i].L, R = q[i].R;

        // Print sum of current query range
        cout << "Minimum of [" << L << ", "
              << R << "] is " << arr[lookup[L][R]] << endl;
    }
}

// Driver program
int main()
{
    int a[] = {7, 2, 3, 0, 5, 10, 3, 12, 18};
    int n = sizeof(a)/sizeof(a[0]);
    Query q[] = {{0, 4}, {4, 7}, {7, 8}};
    int m = sizeof(q)/sizeof(q[0]);
    RMQ(a, n, q, m);
    return 0;
}

```

[Run on IDE](#)

Output:

```

Minimum of [0, 4] is 0
Minimum of [4, 7] is 3
Minimum of [7, 8] is 12

```

This approach supports query in $O(1)$, but preprocessing takes $O(n^2)$ time. Also, this approach needs $O(n^2)$ extra space which may become huge for large input arrays.

Method 2 (Square Root Decomposition)

We can use Square Root Decompositions to reduce space required in above method.

Preprocessing:

- 1) Divide the range $[0, n-1]$ into different blocks of \sqrt{n} each.
- 2) Compute minimum of every block of size \sqrt{n} and store the results.

Preprocessing takes $O(\sqrt{n} * \sqrt{n}) = O(n)$ time and $O(\sqrt{n})$ space.



7	2	3	0	5	10	3	12	18
0	1	2	3	4	5	6	7	8

arr[]

1	3	6
---	---	---

lookup[]

lookup[0] is index of minimum in arr[0..2],

lookup[1] is index minimum of arr[3..5]

lookup[2] is index minimum of arr[6..8]

Query:

1) To query a range [L, R], we take minimum of all blocks that lie in this range. For left and right corner blocks which may partially overlap with given range, we linearly scan them to find minimum.

Time complexity of query is $O(\sqrt{n})$. Note that we have minimum of middle block directly accessible and there can be at most $O(\sqrt{n})$ middle blocks. There can be at most two corner blocks that we may have to scan, so we may have to scan $2 * O(\sqrt{n})$ elements of corner blocks. Therefore, overall time complexity is $O(\sqrt{n})$.

Refer [Sqrt \(or Square Root\) Decomposition Technique | Set 1 \(Introduction\)](#) for details.

Method 3 (Sparse Table Algorithm)

The above solution requires only $O(\sqrt{n})$ space, but takes $O(\sqrt{n})$ time to query. Sparse table method supports query time $O(1)$ with extra space $O(n \log n)$.

The idea is to precompute minimum of all subarrays of size 2^j where j varies from 0 to $\log n$. Like method 1, we make a lookup table. Here $\text{lookup}[i][j]$ contains minimum of range starting from i and of size 2^j . For example $\text{lookup}[0][3]$ contains minimum of range [0, 7] (starting with 0 and of size 2^3)

Preprocessing:

How to fill this lookup table? The idea is simple, fill in bottom up manner using previously computed values.

For example, to find minimum of range [0, 7], we can use minimum of following two.

- Minimum of range [0, 3]
- Minimum of range [4, 7]



Based on above example, below is formula,

```
// If arr[lookup[0][2]] <= arr[lookup[4][2]],
// then lookup[0][3] = lookup[0][2]
If arr[lookup[i][j-1]] <= arr[lookup[i+2j-1-1][j-1]]
    lookup[i][j] = lookup[i][j-1]

// If arr[lookup[0][2]] > arr[lookup[4][2]],
// then lookup[0][3] = lookup[4][2]
Else
    lookup[i][j] = lookup[i+2j-1-1][j-1]
```

7	2	3	0	5	10	3	12	18
0	1	2	3	4	5	6	7	8

arr[]

0	1	3	3
1	1	3	3
2	3	3	—
3	3	3	—
4	4	6	—
5	6	6	—
6	6	—	—
7	7	—	—
8	—	—	—

**lookup[i][j] contains index of
minimum in range from arr[i] to
arr[i + 2^j - 1]**

lookup[][]

Query:

For any arbitrary range [L, R], we need to use ranges which are in powers of 2. The idea is to use closest power of 2. We always need to do at most one comparison (compare minimum of two ranges which are powers of 2). One range starts with L and ends with “L + closest-power-of-2”. The other range ends at R and starts with “R – same-closest-power-of-2 + 1”. For example, if given range is (2, 10), we compare minimum of two ranges (2, 9) and (3, 10).

Based on above example, below is formula,

```
// For (2,10), j = floor(Log2(10-2+1)) = 3
j = floor(Log(R-L+1))

// If arr[lookup[0][3]] <= arr[lookup[3][3]],
// then RMQ(2,10) = lookup[0][3]
```



```

If arr[lookup[L][j]] <= arr[lookup[R-(int)pow(2,j)+1][j]]
    RMQ(L, R) = lookup[L][j]

// If arr[lookup[0][3]] > arr[lookup[3][3]],
// then RMQ(2,10) = lookup[3][3]
Else
    RMQ(L, R) = lookup[i+2j-1-1][j-1]

```

Since we do only one comparison, time complexity of query is $O(1)$.

Below is C++ implementation of above idea.

```

// C++ program to do range minimum query in O(1) time with
// O(n Log n) extra space and O(n Log n) preprocessing time
#include<bits/stdc++.h>
using namespace std;
#define MAX 500

// lookup[i][j] is going to store index of minimum value in
// arr[i..j]. Ideally lookup table size should not be fixed and
// should be determined using n Log n. It is kept constant to
// keep code simple.
int lookup[MAX][MAX];

// Structure to represent a query range
struct Query
{
    int L, R;
};

// Fills lookup array lookup[][] in bottom up manner.
void preprocess(int arr[], int n)
{
    // Initialize M for the intervals with length 1
    for (int i = 0; i < n; i++)
        lookup[i][0] = i;

    // Compute values from smaller to bigger intervals
    for (int j=1; (1<<j)<=n; j++)
    {
        // Compute minimum value for all intervals with size 2^j
        for (int i=0; (i+(1<<j)-1) < n; i++)
        {
            // For arr[2][10], we compare arr[lookup[0][3]] and
            // arr[lookup[3][3]]
            if (arr[lookup[i][j-1]] < arr[lookup[i + (1<<(j-1))][j-1]])
                lookup[i][j] = lookup[i][j-1];
            else
                lookup[i][j] = lookup[i + (1 << (j-1))][j-1];
        }
    }
}

// Returns minimum of arr[L..R]
int query(int arr[], int L, int R)
{
    // For [2,10], j = 3
    int j = (int)log2(R-L+1);

    // For [2,10], we compare arr[lookup[0][3]] and
    // arr[lookup[3][3]],
    if (arr[lookup[L][j]] <= arr[lookup[R - (1<<j) + 1][j]])
        return arr[lookup[L][j]];

    else return arr[lookup[R - (1<<j) + 1][j]];
}

```



```
// Prints minimum of given m query ranges in arr[0..n-1]
void RMQ(int arr[], int n, Query q[], int m)
{
    // Fills table lookup[n][Log n]
    preprocess(arr, n);

    // One by one compute sum of all queries
    for (int i=0; i<m; i++)
    {
        // Left and right boundaries of current range
        int L = q[i].L, R = q[i].R;

        // Print sum of current query range
        cout << "Minimum of [" << L << ", "
              << R << "] is " << query(arr, L, R) << endl;
    }
}

// Driver program
int main()
{
    int a[] = {7, 2, 3, 0, 5, 10, 3, 12, 18};
    int n = sizeof(a)/sizeof(a[0]);
    Query q[] = {{0, 4}, {4, 7}, {7, 8}};
    int m = sizeof(q)/sizeof(q[0]);
    RMQ(a, n, q, m);
    return 0;
}
```

[Run on IDE](#)

Output:

```
Minimum of [0, 4] is 0
Minimum of [4, 7] is 3
Minimum of [7, 8] is 12
```

So sparse table method supports query operation in $O(1)$ time with $O(n \log n)$ preprocessing time and $O(n \log n)$ space.

This article is contributed by **Ruchir Garg**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure Competitive Programming array-range-queries Segment-Tree

Please write to us at contribute@geeksforgeeks.org to report any issue with the [Login to Improve this Article](#) above content.

Recommended Posts:

[Segment Tree | Set 2 \(Range Minimum Query\)](#)

[Sqrt \(or Square Root\) Decomposition Technique | Set 1 \(Introduction\)](#)





Sparse Table

We have briefly discussed sparse table in [Range Minimum Query \(Square Root Decomposition and Sparse Table\)](#) **4.3**

Sparse table concept is used for fast queries on a set of static data (elements do not change). It does preprocessing so that the queries can answered efficiently.

Example Problem 1 : Range Minimum Query

We have an array $arr[0 \dots n-1]$. We need to efficiently find the minimum value from index L (query start) to R (query end) where $0 \leq L \leq R \leq n-1$. Consider a situation when there are many range queries.

Example:

```
Input: arr[] = {7, 2, 3, 0, 5, 10, 3, 12, 18};  
       query[] = [0, 4], [4, 7], [7, 8]
```

```
Output: Minimum of [0, 4] is 0  
        Minimum of [4, 7] is 3  
        Minimum of [7, 8] is 12
```

The idea is to precompute minimum of all subarrays of size 2^j where j varies from 0 to $\log n$. We make a table $lookup[i][j]$ such that $lookup[i][j]$ contains minimum of range starting from i and of size 2^j . For example $lookup[0][3]$ contains minimum of range $[0, 7]$ (starting with 0 and of size 2^3)

How to fill this lookup or sparse table?

The idea is simple, fill in bottom up manner using previously computed values. We compute ranges with current power of 2 using values of lower power of two. For example, to find minimum of range $[0, 7]$ (Range size is power of 3), we can use minimum of following two.

- a) Minimum of range [0, 3] (Range size is power of 2)
- b) Minimum of range [4, 7] (Range size is power of 2)

Based on above example, below is formula,

```
// Minimum of single element subarrays is same
// as the only element.
lookup[i][i] = arr[i]

// If lookup[0][2] <= lookup[4][2],
// then lookup[0][3] = lookup[0][2]
If lookup[i][j-1] <= lookup[i+2j-1-1][j-1]
    lookup[i][j] = lookup[i][j-1]

// If lookup[0][2] > lookup[4][2],
// then lookup[0][3] = lookup[4][2]
Else
    lookup[i][j] = lookup[i+2j-1-1][j-1]
```

	0	1	2	3	4	5	6	7	8
	7	2	3	0	5	10	3	12	18
	arr[]								
		j							
		0	1	2	3				
i	0	7	2	0	0	lookup[i][j] contains minimum in range from arr[i] to arr[i + 2^j - 1]			
	1	2	2	0	0				
	2	3	0	0	-				
	3	0	0	0	-				
	4	5	5	3	-				
	5	10	3	3	-				
	6	3	3	-	-				
	7	12	12	-	-				
	8	18	-	-	-				

For any arbitrary range [l, R], we need to use ranges which are in powers of 2. The idea is to use closest power of 2. We always need to do at most one comparison (compare minimum of two ranges which are powers of 2). One range starts with L and ends with “L + closest-power-of-2”. The other range ends at R and starts with “R – same-closest-power-of-2 + 1”. For example, if given range is (2, 10), we compare minimum of two ranges (2, 9) and (3, 10).

Based on above example, below is formula,

```
// For (2, 10), j = floor(Log2(10-2+1)) = 3
j = floor(Log(R-L+1))

// If lookup[0][3] <= lookup[3][3],
// then min(2, 10) = lookup[0][3]
If lookup[L][j] <= lookup[R-(int)pow(2, j)+1][j]
```

```
min(L, R) = lookup[L][j]
```

```
// If lookup[0][3] > arr[lookup[3][3],  
// then min(2, 10) = lookup[3][3]
```

```
Else
```

```
min(L, R) = lookup[i+2j-1][j-1]
```

Since we do only one comparison, time complexity of query is $O(1)$.

Below is C++ implementation of above idea.

```
// C++ program to do range minimum query  
// using sparse table  
#include <bits/stdc++.h>  
using namespace std;  
#define MAX 500  
  
// lookup[i][j] is going to store minimum  
// value in arr[i..j]. Ideally lookup table  
// size should not be fixed and should be  
// determined using n Log n. It is kept  
// constant to keep code simple.  
int lookup[MAX][MAX];  
  
// Fills lookup array lookup[][] in bottom up manner.  
void buildSparseTable(int arr[], int n)  
{  
    // Initialize M for the intervals with length 1  
    for (int i = 0; i < n; i++)  
        lookup[i][0] = arr[i];  
  
    // Compute values from smaller to bigger intervals  
    for (int j = 1; (1 << j) <= n; j++) {  
        // Compute minimum value for all intervals with  
        // size 2^j  
        for (int i = 0; (i + (1 << j) - 1) < n; i++) {  
            // For arr[2][10], we compare arr[lookup[0][7]]  
            // and arr[lookup[3][10]]  
            if (lookup[i][j - 1] <  
                lookup[i + (1 << (j - 1))][j - 1])  
                lookup[i][j] = lookup[i][j - 1];  
            else  
                lookup[i][j] =  
                    lookup[i + (1 << (j - 1))][j - 1];  
        }  
    }  
}  
  
// Returns minimum of arr[L..R]  
int query(int L, int R)  
{  
    // Find highest power of 2 that is smaller  
    // than or equal to count of elements in given  
    // range. For [2, 10], j = 3  
    int j = (int)log2(R - L + 1);  
  
    // Compute minimum of last 2^j elements with first  
    // 2^j elements in range.  
    // For [2, 10], we compare arr[lookup[0][3]] and  
    // arr[lookup[3][3]],  
    if (lookup[L][j] <= lookup[R - (1 << j) + 1][j])  
        return lookup[L][j];  
  
    else
```

```

        return lookup[R - (1 << j) + 1][j];
    }

// Driver program
int main()
{
    int a[] = { 7, 2, 3, 0, 5, 10, 3, 12, 18 };
    int n = sizeof(a) / sizeof(a[0]);
    buildSparseTable(a, n);
    cout << query(0, 4) << endl;
    cout << query(4, 7) << endl;
    cout << query(7, 8) << endl;
    return 0;
}

```

Run on IDE

Output:

```

Minimum of [0, 4] is 0
Minimum of [4, 7] is 3
Minimum of [7, 8] is 12

```

So sparse table method supports query operation in $O(1)$ time with $O(n \log n)$ preprocessing time and $O(n \log n)$ space.

Example Problem 2 : Range GCD Query

We have an array $arr[0 \dots n-1]$. We need to find the **greatest common divisor** in the range L and R where $0 \leq L \leq R \leq n-1$. Consider a situation when there are many range queries

Examples:

```

Input : arr[] = {2, 3, 5, 4, 6, 8}
        queries[] = {(0, 2), (3, 5), (2, 3)}
Output : 1
         2
         1

```

We use below properties of GCD:

- GCD function is associative [$\text{GCD}(a, b, c) = \text{GCD}(\text{GCD}(a, b), c) = \text{GCD}(a, \text{GCD}(b, c))$], we can compute GCD of a range using GCDs of subranges.
- If we take GCD of an overlapping range more than once, then it does not change answer. For example $\text{GCD}(a, b, c) = \text{GCD}(\text{GCD}(a, b), \text{GCD}(b, c))$. Therefore like minimum range query problem, we need to do only one comparison to find GCD of given range.

We build sparse table using same logic as above. After building sparse table, we can find all GCDs by breaking given range in powers of 2 and add GCD of every piece to current answer.

```

// C++ program to do range minimum query
// using sparse table
#include <bits/stdc++.h>
using namespace std;
#define MAX 500

// lookup[i][j] is going to store GCD of
// arr[i..j]. Ideally lookup table
// size should not be fixed and should be
// determined using n Log n. It is kept
// constant to keep code simple.
int table[MAX][MAX];

// it builds sparse table.
void buildSparseTable(int arr[], int n)
{
    // GCD of single element is element itself
    for (int i = 0; i < n; i++)
        table[i][0] = arr[i];

    // Build sparse table
    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]);
}

// Returns GCD of arr[L..R]
int query(int L, int R)
{
    // Find highest power of 2 that is smaller
    // than or equal to count of elements in given
    // range. For [2, 10], j = 3
    int j = (int)log2(R - L + 1);

    // Compute GCD of last 2^j elements with first
    // 2^j elements in range.
    // For [2, 10], we find GCD of arr[lookup[0][3]] and
    // arr[lookup[3][3]],
    return __gcd(table[L][j], table[R - (1 << j) + 1][j]);
}

// Driver program
int main()
{
    int a[] = { 7, 2, 3, 0, 5, 10, 3, 12, 18 };
    int n = sizeof(a) / sizeof(a[0]);
    buildSparseTable(a, n);
    cout << query(0, 2) << endl;
    cout << query(1, 3) << endl;
    cout << query(4, 5) << endl;
    return 0;
}

```

[Run on IDE](#)

Output:

```

1
1
5

```

This article is contributed by **sunny6041**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to