

Sparse Tables

Sparse tables are all about doing efficient **range queries** on **static arrays**. For example: min, max, sum, gcd.

For **associative functions**, a sparse table can answer range queries in $O(\log_2 n)$.

A function $f(x, y)$ is **associative** if $f(a, f(b, c)) = f(f(a, b), c)$ for all a, b, c .

Addition and multiplication are associative. Subtraction and exponentiation are not:

$$\begin{array}{lcl} \text{Let } f(a, b) = a - b & & \\ f(1, g(2, 3)) & & f(g(1, 2), 3) \\ = f(1, 2 - 3) & & = f(1 - 2, 3) \\ = 1 - (2 - 3) & & = (1 - 2) - 3 \\ = 1 - (-1) & & = (-1) - 3 \\ = 2 & & = -4 \\ \therefore f \text{ is not associative} \end{array}$$

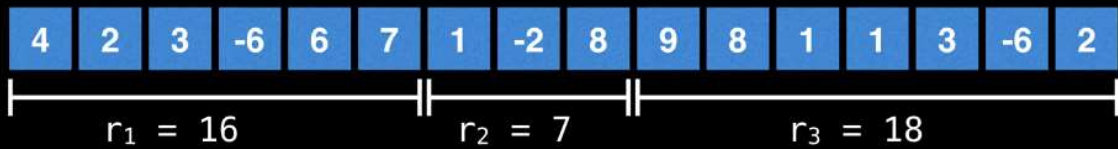
When the **range query combination function** is **overlap friendly**, then range queries on a sparse table can be answered in **$O(1)$** .

Being **overlap friendly** means that a function yields the same answer regardless of whether it is combining ranges which overlap or those that do not.

We say a binary function $f(x, y)$ is overlap friendly if:
 $f(f(a, b), f(b, c)) = f(a, f(b, c))$ for all valid a, b, c

Example:

Let $f(x, y) = x + y$



$$f(f(r_1, r_2), f(r_2, r_3)) = f(f(16, 7), f(7, 18)) = f(23, 25) = 48$$

Calculating $f(r_1, f(r_2, r_3))$ gives us a different result:

$$f(r_1, f(r_2, r_3)) = f(16, f(7, 18)) = f(16, 25) = 41$$

$\therefore f(x, y)$ is not overlap friendly.

Q: Which of these are "overlap friendly"?

$$f(a, b) = 1 * b$$

$$f(a, b) = a * b$$

$$f(a, b) = \min(a, b)$$

$$f(a, b) = \max(a, b)$$

$$f(a, b) = a + b$$

$$f(a, b) = a - b$$

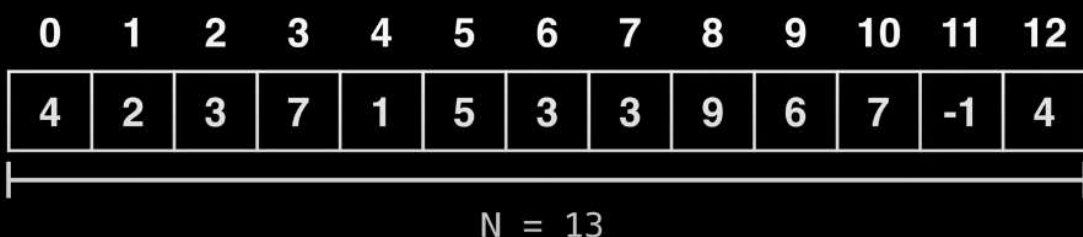
$$f(a, b) = (a * b) / a, a \neq 0$$

$$f(a, b) = \gcd(a, b)$$

The idea behind a sparse table is to precompute answers for all intervals of size 2^x to efficiently answer range queries $[l, r]$

Let N be the size of input value array, and let 2^P be the largest power of 2 that fits in the length of the entire values array.

$$P = \text{floor}(\log_2 N) = \text{floor}(\log_2 13) = 3$$



Begin by initializing a table with $P + 1$ rows and N columns. Then, fill the first row with the input values.

Each cell (i, j) represents the answer for the range $[j, j + 2^i)$ in the original array.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0, 2^0	4	2	3	7	1	5	3	3	9	6	7	-1	4
1, 2^1													
2, 2^2													
3, 2^3													

For example, cell $(2, 5)$ represents the answer for the range $[5, 9)$. If we're building a min sparse table, then the blue cell would have a value of 3.

We do not need to consider cells with invalid intervals.

For example, we want to build a min sparse table to be able to do min range queries.

The range combination function will be $f(x, y) = \min(x, y)$ and we will use it to combine range queries.

The way we are going to build the table is by reusing the already computed range values of previous cells.

The current cell (i, j) represents the range $[j, j + 2^i)$ which always has even length. This range can be broken down into two sub intervals (which we will have already computed).

More specifically, the range for the cell (i, j) can be split into a left interval $[j, j + 2^{i-1})$ and a right interval $[j + 2^{i-1}, j + 2^i)$ whose values would correspond to the cells $(i - 1, j)$ and $(i - 1, j + 2^{i-1})$ respectively.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0, 2^0	4	2	3	7	1	5	3	3	9	6	7	-1	4
1, 2^1			3		1		3		6				
2, 2^2			1				3						
3, 2^3			1										

The blue cell consists of red cell values, red cell values consist of green cell values, etc.

The dynamic programming idea:

$$dp[i][j] = f(dp[i-1][j], dp[i-1][j+2^{(i-1)}]) = \min(dp[i-1][j], dp[i-1][j+2^{(i-1)}])$$

So how do we find a minimum value for a range?

In our table we have already precomputed the answer for all intervals of length 2^x .

Let k be the largest power of two that fits in the length of the range between $[l, r]$.

Knowing k we can easily do a lookup in the table to find the minimum in between ranges $[l, l + k - 1]$ (**left interval**) and $[r - k + 1, r]$ (**right interval**) to find the answer for $[l, r]$. The left and the right intervals may overlap, but this does not matter (given the **overlap friendly property**) as long as the entire range is covered.

Example: what is the minimum value between $[1, 11]$?

1. Find the value p , which gives the largest possible 2^p that fits the range $[1, 11]$:

Interval length:

$$len = l - r + 1 = 11 - 1 + 1 = 11$$

$$p = \text{floor}(\log_2 len) = \text{floor}(\log_2 11) = \text{floor}(3.321) = 3$$

2. Find k , which is 2^p

$$k = 2^p = 2^3 = 8$$

3. Knowing p and k we can do a lookup for the left and right interval ranges:

$$\min(t[p][1], t[p][r - k + 1])$$











We are looking at row p because we know that it is the maximum number which will fit 2^p in the interval --> two lookups are enough here (blue cells), and we do not need to combine 4 interval lookups if we looked at the second row, for example.

$$\min(t[3][1], t[3][11 - 8 + 1])$$

$$\min(t[3][1], t[3][4])$$

$$\min(1, -1)$$

$$= -1$$

	0	1	2	3	4	5	6	7	8	9	10	11	12
0, 2^0	4	2	3	7	1	5	3	3	9	6	7	-1	4
1, 2^1	2	2	3	1	1	3	3	3	6	6	-1	-1	
2, 2^2	2	1	1	1	1	3	3	3	-1	-1			
3, 2^3	1	1	1	1	-1	-1							





Some functions such as multiplication and summation are **associative**, but not overlap friendly. A sparse table can still handle these types of queries, but in $O(\log_2 n)$ rather than $O(1)$. The issue is that overlapping intervals would yield the wrong answer.

The alternative approach to performing a range query is to do a *cascading query* on the sparse table by breaking the range $[l, r]$ into smaller ranges of size 2^x which **do not overlap**.

Example of an **associative function query**:

Consider an input array $[1, 2, -3, 2, 4, -1, 5]$

Suppose we want to find the product of all elements between $[0, 6]$ using a sparse table. First, we would construct a sparse table like we did before:

	0	1	2	3	4	5	6
0, 2^0	1	2	-3	2	4	-1	5
1, 2^1	2	-6	-6	8	-4	-5	
2, 2^2	-12	-48	24	-40			

1. Break interval $[0, 6]$ into powers of 2: $[0, 2^2) \cup [4, 4 + 2^1) \cup [6, 6 + 2^0)$. Important: always start with the **largest power** of 2 fitting in the interval.
2. Lookup the value of each interval in the table and take the product of all the intervals:
 $table[2][0] * table[1][4] * table[0][6] = -12 * -4 * 5 = 240$

Pseudocode:

Pseudocode

```
# The number of elements in the input array
N = ...

# P, short for power. The largest  $2^P$  that fits in N
P = ... # calculated as:  $\text{floor}(\log_2(N))$ 

# A quick lookup table for  $\text{floor}(\log_2(i))$ ,  $1 \leq i \leq N$ 
log2 = ... # size N+1, index 0 unused.

# The sparse table containing integer values
dp = ... # P+1 rows and N columns

# Index Table (IT) associated with the values in the
# sparse table. This table is only useful when we want
# to query the index of the min (or max) element in
# the range [l, r] rather than the value itself.
# The index table doesn't make sense for most other
# range query types like gcd or sum.
it = ... # P+1 rows and N columns
```

```
function BuildMinSparseTable(values):
    N = length(values)
    P = floor(log(N) / log(2))

    # Quick lookup table for  $\text{floor}(\log_2(i))$ ,  $1 \leq i \leq N$ 
    log2 = [0,0,...,0,0] # size N+1
    for (i = 2; i <= N; i++):
        log2[i] = log2[i/2] + 1

    # Fill first row
    for (i = 0; i < N; i++):
        dp[0][i] = values[i]
        it[0][i] = i

    for (p = 1; p <= P; p++):
        for (i = 0; i + (1 << p) <= N; i++):
            left = dp[p-1][i]
            right = dp[p-1][i+(1<<(p-1))]
            dp[p][i] = min(left, right)

    # Save/propagate the index of smallest element
    if left <= right:
        it[p][i] = it[p-1][i]
    else:
        it[p][i] = it[p-1][i+(1<<(p-1))]
```



```
# Query the smallest element in the range [l, r], O(1)
function MinQuery(l, r):
    len = r - l + 1
    p = log2[len]
    left = dp[p][l]
    right = dp[p][r - (1 << p) + 1]
    return min(left, right)
```

```
# Query the smallest element in the range [l, r] by doing a
# cascading min query, O(log2(n)).
function CascadingMinQuery(l, r):
    min_val = +∞
    for (p = log2[r - l + 1]; l <= r; p = log2[r - l + 1]):
        min_val = min(min_val, dp[p][l])
        l += (1 << p)
    return min_val
```

```
# Returns in index of the minimum element in the range [l, r]
# in the input values array. If there are multiple smallest
# elements, the index of leftmost is returned.
function MinIndexQuery(l, r):
    len = r - l + 1
    p = log2[len]
    left = dp[p][l]
    right = dp[p][r - (1 << p) + 1]
    if left <= right:
        return it[p][l]
    return it[p][r - (1 << p) + 1]
```

Source Code:


```

/**
 * Min sparse table example
 *
 * <p>Download the code: <br>
 * $ git clone https://github.com/williamfiset/algorithms
 *
 * <p>Run: <br>
 * $ ./gradlew run -Palgorithm=datastructures.sparsetable.examples.MinSparse
 *
 * <p>Construction complexity:  $O(n \log n)$ , query complexity:  $O(1)$ 
 *
 * @author William Fiset, william.alexandre.fiset@gmail.com
 */
package com.williamfiset.algorithms.datastructures.sparsetable.examples;

// Sparse table for efficient minimum range queries in  $O(1)$  with  $O(n \log n)$  sp
public class MinSparseTable {

    // Example usage:
    public static void main(String[] args) {
        // index values: 0, 1, 2, 3, 4, 5, 6
        long[] values = {1, 2, -3, 2, 4, -1, 5};
        MinSparseTable sparseTable = new MinSparseTable(values);

        System.out.println(sparseTable.queryMin(1, 5)); // prints -3
        System.out.println(sparseTable.queryMinIndex(1, 5)); // prints 2

        System.out.println(sparseTable.queryMin(3, 3)); // prints 2
        System.out.println(sparseTable.queryMinIndex(3, 3)); // prints 3

        System.out.println(sparseTable.queryMin(3, 6)); // prints -1
        System.out.println(sparseTable.queryMinIndex(3, 6)); // prints 5
    }

    // The number of elements in the original input array.
    private int n;

    // The maximum power of 2 needed. This value is  $\text{floor}(\log_2(n))$ 
    private int P;

    // Fast log base 2 logarithm lookup table,  $1 \leq i \leq n$ 
    private int[] log2;

    // The sparse table values.
    private long[][] dp;

    // Index Table (IT) associated with the values in the sparse table. This t
    // is only useful when we want to query the index of the min (or max) elem
    // in the range  $[L, r]$  rather than the value itself. The index table doesn
    // make sense for most other range query types like gcd or sum.
    private int[][] it;

    public MinSparseTable(long[] values) {
        n = values.length;
        P = (int) (Math.log(n) / Math.log(2));
        .
        -
        -
        -

```

```

dp = new long[P + 1][n];
it = new int[P + 1][n];

for (int i = 0; i < n; i++) {
    dp[0][i] = values[i];
    it[0][i] = i;
}

log2 = new int[n + 1];
for (int i = 2; i <= n; i++) {
    log2[i] = log2[i / 2] + 1;
}

// Build sparse table combining the values of the previous intervals.
for (int p = 1; p <= P; p++) {
    for (int i = 0; i + (1 << p) <= n; i++) {
        long leftInterval = dp[p - 1][i];
        long rightInterval = dp[p - 1][i + (1 << (p - 1))];
        dp[p][i] = Math.min(leftInterval, rightInterval);

        // Propagate the index of the best value
        if (leftInterval <= rightInterval) {
            it[p][i] = it[p - 1][i];
        } else {
            it[p][i] = it[p - 1][i + (1 << (p - 1))];
        }
    }
}

// Do a min query on the interval [L, r] in O(1).
//
// We can get O(1) query by finding the smallest power of 2 that fits with
// the interval length which we'll call k. Then we can query the intervals
// [L, L+k] and [r-k+1, r] (which likely overlap) and apply the function
// again. Some functions (like min and max) don't care about overlapping
// intervals so this trick works, but for a function like sum this would
// return the wrong result since it is not an idempotent binary function
// (aka an overlap friendly function).
private long queryMin(int l, int r) {
    int length = r - l + 1;
    int p = log2[length];
    int k = 1 << p; // 2 to the power of p
    return Math.min(dp[p][l], dp[p][r - k + 1]);
}

// Returns the index of the minimum element in the range [L, r].
public int queryMinIndex(int l, int r) {
    int length = r - l + 1;
    int p = log2[length];
    int k = 1 << p; // 2 to the power of p
    long leftInterval = dp[p][l];
    long rightInterval = dp[p][r - k + 1];
    if (leftInterval <= rightInterval) {
        return it[p][l];
    } else {
        return it[p][r - k + 1];
    }
}

```

```
    } else {  
        return it[p][r - k + 1];  
    }  
}  
}
```