Sparse Table - GeeksforGeeks

Source: https://www.geeksforgeeks.org/sparse-table/

Sparse Table Last Updated : 29 Apr, 2025 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 be answered efficiently. Range Minimum Query Using Sparse Table You are given an integer array arr of length n and an integer q denoting the number of queries. Each query consists of two indices L and R (0 ≤ L ≤ R < n), and asks for the minimum value in the subarray arr[L…R]. Example: Input: arr[] = [7, 2, 3, 0, 5, 10, 3, 12, 18] queries[][] = [[0, 4], [4, 7], [7, 8]] Output: 0 3 12 Explanation: For query 1, the subarray spanning indices 0 through 4 contains the values 7, 2, 3, and 0, and the minimum among them is 0. Similarly, the minimum value in range [4, 7] and [7, 8] are 3 and 12 respectively. Approach: The idea is to precompute the minimum values for all subarrays whose lengths are powers of two and store them in a table so that any range-minimum query can be answered in constant time. We build a 2D array lookup where lookup[i][j] holds the minimum of the subarray starting at i with length 2^j (j varies from to Log n where n is the length of the input array). 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 a bottom-up manner using previously computed values. We compute ranges with current power of 2 using values of lower power of two. Each entry for length 2^j is derived by combining two overlapping subarrays of length 2^(j–1) that were computed in the previous step. For example, to find a minimum of range [0, 7] (Range size is a power of 3), we can use the minimum of following two. a) Minimum of range [0, 3] (Range size is a power of 2) b) Minimum of range [4, 7] (Range size is a power of 2) Based on above example, below is formula, // Minimum of single element subarrays is same // as the only element. lookup[i][0] = arr[i]

// If lookup[0][2] <= lookup[4][2], // then lookup[0][3] = lookup[0][2] If lookup[i][j-1] <= lookup[i+2 j-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+2 j-1][j-1] Follow the below given steps: Initialize lookup[i][0] = arr[i] for every 0 ≤ i < n . For each j from 1 up to ■log■n■ , and for each i from 0 to n – 2^j : Compute lookup[i][j] as the minimum of the two halves: the interval beginning at i of length 2^(j–1) (lookup[i][j–1]) the interval beginning at i + 2^(j–1) of length 2^(j–1) (lookup[i + 2^(j–1)][j–1]) To answer a query on the range [L, R] : Let k = ■log■(R – L + 1)■ . The minimum over [L, R] is min(lookup[L][k], lookup[R – 2^k + 1][k]) . How do we handle individual queries? Find highest power of 2 that is smaller than or equal to count of elements in given range [L, R]. we get j = floor(log2(R - L + 1)) For [2, 10], j = 3 Compute minimum of last 2^j elements with first 2^j elements in range. For [2, 10], we compare lookup[0][3] (minimum from 0 to 7) and lookup[3][3] (minimum from 3 to 10) and return the minimum of two values. C++ #include <bits/stdc++.h> using namespace std ; // Fills lookup array lookup[][] in bottom up manner. vector < vector < int >> buildSparseTable (vector < int > & arr) { int n = arr . size (); // create the 2d table vector < vector < int >> lookup (n + 1 , vector < int > (log2 (n) + 1)); // Initialize 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 ++) { 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]; } } return lookup ; } // Returns minimum of arr[L..R] int query (int L , int R , vector < vector < int >> & lookup) { // Find highest power of 2 that is smaller // than or equal to count of elements in range int j = (int) log2 (R - L + 1); // Compute minimum of last 2^j elements with first // 2^j elements in range. if (lookup [L][j] <= lookup [R - (1 << j) + 1][j]) return lookup [L][j]; else return lookup [R - (1 << j) + 1][j]; } vector < int > solveQueries (vector < int >& arr , vector < vector < int >>& queries) { int n = arr . size (); int m = queries . size (); vector < int > result (m); // Build the sparse table vector < vector < int >> lookup = buildSparseTable (arr); // Process each query for (int i = 0 ; i < m ; i ++) { int L = queries [i][0]; int R = queries [i][1]; result [i] = query (L , R , lookup); } return result ; } int main () { vector < int > arr = { 7 , 2 , 3 , 0 , 5 , 10 , 3 , 12 , 18 }; vector < vector < int >> queries = { { 0 , 4 }, { 4 , 7 }, { 7 , 8 } }; vector < int > res = solveQueries (arr , queries); for (int i = 0 ; i < res . size (); i ++) { cout << res [i] << " " ; } return 0 ; } Java public class GfG { // Fills lookup array lookup[][] in bottom up manner. public static int [][] buildSparseTable (int [] arr) { int n = arr . length ; // create the 2d table int [][] lookup = new int [n + 1][(int)(Math . log (n) / Math . log (2)) + 1] ; // Initialize 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 ++) { 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] ; } } return lookup ; } // Returns minimum of arr[L..R] public static int query (int L , int R , int [][] lookup) { // Find highest power of 2 that is smaller // than or equal to count of elements in range int j = (int)(Math . log (R - L + 1) / Math . log (2)); // Compute minimum of last 2^j elements with first // 2^j elements in range. if (lookup [L][j] <= lookup [R - (1 << j) + 1][j]) return lookup [L][j] ; else return lookup [R - (1 << j) + 1][j] ; } public static int [] solveQueries (int [] arr , int [][] queries) { int n = arr . length ; int m = queries . length ; int [] result = new int [m] ; // Build the sparse table int [][] lookup = buildSparseTable (arr); // Process each query for (int i = 0 ; i < m ; i ++) { int L = queries [i][0] ; int R = queries [i][1] ; result [i] = query (L , R , lookup); } return result ; } public static void main (String [] args) { int [] arr = { 7 , 2 , 3 , 0 , 5 , 10 , 3 , 12 , 18 }; int [][] queries = { { 0 , 4 }, { 4 , 7 }, { 7 , 8 } }; int [] res = solveQueries (arr , queries); for (int i = 0 ; i < res . length ; i ++) { System . out . print (res [i] + " "); } } } Python import math # Fills lookup array lookup[][] in bottom up manner. def buildSparseTable (arr): n = len (arr) # create the 2d table lookup = [[0] * (int (math . log (n , 2)) + 1) for _ in range (n + 1)] # Initialize for the intervals with length 1 for i in range (n): lookup [i][0] = arr [i] # Compute values from smaller to bigger intervals for j in range (1 , int (math . log (n , 2)) + 1): # Compute minimum value for all intervals with # size 2^j for i in range (0 , n - (1 << j) + 1): 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] return lookup # Returns minimum of arr[L..R] def query (L , R , lookup): # Find highest power of 2 that is smaller # than or equal to count of elements in range j = int (math . log (R - L + 1 , 2)) # Compute minimum of last 2^j elements with first # 2^j elements in range. if lookup [L][j] <= lookup [R - (1 << j) + 1][j]: return lookup [L][j] else : return lookup [R - (1 << j) + 1][j] def solveQueries (arr , queries): n = len (arr) m = len (queries) result = [0] * m # Build the sparse table lookup = buildSparseTable (arr) # Process each query for i in range (m): L = queries [i][0] R = queries [i][1] result [i] = query (L , R , lookup) return result if __name__ == "__main__" : arr = [7 , 2 , 3 , 0 , 5 , 10 , 3 , 12 , 18] queries = [[0 , 4], [4 , 7], [7 , 8]] res = solveQueries (arr , queries) for x in res : print (x , end = " ") C# using System ; public class GfG { // Fills lookup array lookup[][] in bottom up manner. public static int [][] buildSparseTable (int [] arr) { int n = arr . Length ; // create the 2d table int cols = (int)(Math . Log (n) / Math . Log (2)) + 1 ; int [][] lookup = new int [n + 1][]; for (int i = 0 ; i < n + 1 ; i ++) lookup [i] = new int [cols]; // Initialize 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 ++) { 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]; } } return lookup ; } // Returns minimum of arr[L..R] public static int query (int L , int R , int [][] lookup) { // Find highest power of 2 that is smaller // than or equal to count of elements in range int j = (int)(Math . Log (R - L + 1) / Math . Log (2)); // Compute minimum of last 2^j elements with first // 2^j elements in range. if (lookup [L][j] <= lookup [R - (1 << j) + 1][j]) return lookup [L][j]; else return lookup [R - (1 << j) + 1][j]; } public static int [] solveQueries (int [] arr , int [][] queries) { int n = arr . Length ; int m = queries . Length ; int [] result = new int [m]; // Build the sparse table int [][] lookup = buildSparseTable (arr); // Process each query for (int i = 0 ; i < m ; i ++) { int L = queries [i][0]; int R = queries [i][1]; result [i] = query (L , R , lookup); } return result ; } public static void Main (string [] args) { int [] arr = { 7 , 2 , 3 , 0 , 5 , 10 , 3 , 12 , 18 }; int [][] queries = { new int []{ 0 , 4 }, new int []{ 4 , 7 }, new int []{ 7 , 8 } }; int [] res = solveQueries (arr , queries); for (int i = 0 ; i < res . Length ; i ++) { Console . Write (res [i] + " "); } } } JavaScript // Fills lookup array lookup[][] in bottom up manner. function buildSparseTable (arr) { const n = arr . length ; // create the 2d table const cols = Math . floor (Math . log2 (n)) + 1 ; const lookup = Array . from ({ length : n + 1 }, () => Array (cols). fill (0)); // Initialize for the intervals with length 1 for (let i = 0 ; i < n ; i ++) lookup [i][0] = arr [i]; // Compute values from smaller to bigger intervals for (let j = 1 ; (1 << j) <= n ; j ++) { // Compute minimum value for all intervals with // size 2^j for (let i = 0 ; (i + (1 << j) - 1) < n ; i ++) { 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]; } } return lookup ; } // Returns minimum of arr[L..R] function query (L , R , lookup) { // Find highest power of 2 that is smaller // than or equal to count of elements in range const j = Math . floor (Math . log2 (R - L + 1)); // Compute minimum of last 2^j elements with first // 2^j elements in range. if (lookup [L][j] <= lookup [R - (1 << j) + 1][j]) return lookup [L][j]; else return lookup [R - (1 << j) + 1][j]; } function solveQueries (arr , queries) { const n = arr . length ; const m = queries . length ; const result = new Array (m); // Build the sparse table const lookup = buildSparseTable (arr); // Process each query for (let i = 0 ; i < m ; i ++) { const L = queries [i][0]; const R = queries [i][1]; result [i] = query (L , R , lookup); } return result ; } const arr = [7 , 2 , 3 , 0 , 5 , 10 , 3 , 12 , 18]; const queries = [[0 , 4], [4 , 7], [7 , 8]]; const res = solveQueries (arr , queries); console . log (res . join (' ')); Output 0 3 12 Time Complexity: O(n * log n), sparse table method supports query operation in O(1) time with O(n Log n) preprocessing time. Auxiliary Space: O(n * log n) Range GCD Query Using Sparse Table You are given an integer array arr of length n and an integer q denoting the number of queries. Each query consists of two indices L and R (0 ≤ L ≤ R < n), and asks for the greatest common divisor in the subarray arr[L…R]. Example: Input: arr[] = [2, 3, 5, 4, 6, 8] queries[][] = [[0, 2], [3, 5], [2, 3]] Output: 1 2 1 Explanation: For query 1, gcd of elements 2, 3, and 5 is 1 (as all three are primes). For query 2, gcd of elements 4, 6, and 8 is 2. For query 3, gcd of elements 5 and 4 is 1 (as 4 and 5 are co-primes). Approach: The idea is to exploit the associativity and idempotence of GCD to answer any range■GCD query in constant time after preprocessing. Since GCD(a, b, c) = GCD(GCD(a, b), c) = GCD(a, GCD(b, c)) and taking GCD of an overlapping element more than once does not change the result (e.g. GCD(a, b, c) = GCD(GCD(a, b), GCD(b, c))), we can build a sparse table over powers of two just like in the range■minimum query. Any query range [L, R] can then be covered by two overlapping power■of■two intervals whose GCDs we combine to get the answer. How to handle individual queries? Find highest power of 2 that is smaller than or equal to count of elements in given range [L, R]. we get j = floor(log2(R - L + 1)) For [2, 10], j = 3 Compute GCD of last 2^j elements with first 2^j elements in range. For [2, 10], we compute GCD of lookup[0][3] (GCD of 0 to 7) and GCD of lookup[3][3] (GCD of 3 to 10). Follow the below given steps: For every index i, set lookup[i][0] = arr[i]. For j = 1 to ■log■n■, and for each i from 0 to n − 2■: lookup[i][j] = GCD(lookup[i][j–1], lookup[i + 2^(j–1)][j–1]) To answer a query [L, R]: Let k = ■log■(R − L + 1)■ The GCD over [L, R] is GCD(lookup[L][k], lookup[R − 2■ + 1][k]) Below is given the implementation: C++ #include <bits/stdc++.h> using namespace std ; // Fills lookup array lookup[][] in bottom up manner. vector < vector < int >> buildSparseTable (vector < int > & arr){ int n = arr . size (); // create the 2d table vector < vector < int >> lookup (n + 1 , vector < int > (log2 (n) + 1)); // GCD of single element is element itself for (int i = 0 ; i < n ; i ++) lookup [i][0] = arr [i]; // Build sparse table for (int j = 1 ; j <= log2 (n); j ++) for (int i = 0 ; i <= n - (1 << j); i ++) lookup [i][j] = __gcd (lookup [i][j - 1], lookup [i + (1 << (j - 1))][j - 1]); return lookup ; } // Returns GCD of arr[L..R] int query (int L , int R , vector < vector < int >> & lookup) { // Find highest power of 2 that is smaller // than or equal to count of elements int j = (int) log2 (R - L + 1); // Compute GCD of last 2^j elements with first // 2^j elements in range. return __gcd (lookup [L][j], lookup [R - (1 << j) + 1][j]); } vector < int > solveQueries (vector < int >& arr , vector < vector < int >>& queries) { int n = arr . size (); int m = queries . size (); vector < int > result (m); // Build the sparse table vector < vector < int >> lookup = buildSparseTable (arr); // Process each query for (int i = 0 ; i < m ; i ++) { int L = queries [i][0]; int R = queries [i][1]; result [i] = query (L , R , lookup); } return result ; } int main () { vector < int > arr = { 2 , 3 , 5 , 4 , 6 , 8 }; vector < vector < int >> queries = { { 0 , 2 }, { 3 , 5 }, { 2 , 3 } }; vector < int > res = solveQueries (arr , queries); for (int i = 0 ; i < res . size (); i ++) { cout << res [i] << " " ; } return 0 ; } Java public class GfG { public static int [][] buildSparseTable (int [] arr) { int n = arr . length ; // create the 2d table int [][] lookup = new int [n + 1][(int)(Math . log (n) / Math . log (2)) + 1] ; // GCD of single element is element itself for (int i = 0 ; i < n ; i ++) lookup [i][0] = arr [i] ; // Build sparse table for (int j = 1 ; j <= (int)(Math . log (n) / Math . log (2)); j ++) for (int i = 0 ; i <= n - (1 << j); i ++) lookup [i][j] = gcd (lookup [i][j - 1] , lookup [i + (1 << (j - 1))][j - 1]); return lookup ; } public static int query (int L , int R , int [][] lookup) { // Find highest power of 2 that is smaller // than or equal to count of elements int j = (

int)(Math . log (R - L + 1) / Math . log (2)); // Compute GCD of last 2^j elements with first // 2^j elements in range. return gcd (lookup [L][j] , lookup [R - (1 << j) + 1][j]); } public static int [] solveQueries (int [] arr , int [][] queries) { int n = arr . length ; int m = queries . length ; int [] result = new int [m] ; // Build the sparse table int [][] lookup = buildSparseTable (arr); // Process each query for (int i = 0 ; i < m ; i ++) { int L = queries [i][0] ; int R = queries [i][1] ; result [i] = query (L , R , lookup); } return result ; } private static int gcd (int a , int b) { return b == 0 ? a : gcd (b , a % b); } public static void main (String [] args) { int [] arr = { 2 , 3 , 5 , 4 , 6 , 8 }; int [][] queries = { { 0 , 2 }, { 3 , 5 }, { 2 , 3 } }; int [] res = solveQueries (arr , queries); for (int i = 0 ; i < res . length ; i ++) { System . out . print (res [i] + " "); } } } Python import math def buildSparseTable (arr): n = len (arr) # create the 2d table lookup = [[0] * (int (math . log2 (n)) + 1) for _ in range (n + 1)] # GCD of single element is element itself for i in range (n): lookup [i][0] = arr [i] # Build sparse table for j in range (1 , int (math . log2 (n)) + 1): for i in range (0 , n - (1 << j) + 1): lookup [i][j] = math . gcd (lookup [i][j - 1], lookup [i + (1 << (j - 1))][j - 1]) return lookup def query (L , R , lookup): # Find highest power of 2 that is smaller # than or equal to count of elements j = int (math . log2 (R - L + 1)) # Compute GCD of last 2^j elements with first # 2^j elements in range. return math . gcd (lookup [L][j], lookup [R - (1 << j) + 1][j]) def solveQueries (arr , queries): n = len (arr) m = len (queries) result = [0] * m # Build the sparse table lookup = buildSparseTable (arr) # Process each query for i in range (m): L = queries [i][0] R = queries [i][1] result [i] = query (L , R , lookup) return result if __name__ == "__main__" : arr = [2 , 3 , 5 , 4 , 6 , 8] queries = [[0 , 2], [3 , 5], [2 , 3]] res = solveQueries (arr , queries) for i in range (len (res)): print (res [i], end = " ") C# using System ; public class GfG { public static int [][] buildSparseTable (int [] arr) { int n = arr . Length ; // create the 2d table int [][] lookup = new int [n + 1][]; for (int i = 0 ; i <= n ; i ++) lookup [i] = new int [(int)(Math . Log (n) / Math . Log (2)) + 1]; // GCD of single element is element itself for (int i = 0 ; i < n ; i ++) lookup [i][0] = arr [i]; // Build sparse table for (int j = 1 ; j <= (int)(Math . Log (n) / Math . Log (2)); j ++) for (int i = 0 ; i <= n - (1 << j); i ++) lookup [i][j] = GCD (lookup [i][j - 1], lookup [i + (1 << (j - 1))][j - 1]); return lookup ; } public static int query (int L , int R , int [][] lookup) { // Find highest power of 2 that is smaller // than or equal to count of elements int j = (int)(Math . Log (R - L + 1) / Math . Log (2)); // Compute GCD of last 2^j elements with first // 2^j elements in range. return GCD (lookup [L][j], lookup [R - (1 << j) + 1][j]); } public static int [] solveQueries (int [] arr , int [][] queries) { int n = arr . Length ; int m = queries . Length ; int [] result = new int [m]; // Build the sparse table int [][] lookup = buildSparseTable (arr); // Process each query for (int i = 0 ; i < m ; i ++) { int L = queries [i][0]; int R = queries [i][1]; result [i] = query (L , R , lookup); } return result ; } private static int GCD (int a , int b) { return b == 0 ? a : GCD (b , a % b); } public static void Main (string [] args) { int [] arr = { 2 , 3 , 5 , 4 , 6 , 8 }; int [][] queries = new int [][] { new int []{ 0 , 2 }, new int []{ 3 , 5 }, new int []{ 2 , 3 } }; int [] res = solveQueries (arr , queries); for (int i = 0 ; i < res . Length ; i ++) { Console . Write (res [i] + " "); } } } JavaScript // Fills lookup array lookup[][] in bottom up manner. function buildSparseTable (arr) { const n = arr . length ; // create the 2d table const lookup = Array . from ({ length : n + 1 }, () => Array (Math . floor (Math . log2 (n)) + 1). fill (0)); // GCD of single element is element itself for (let i = 0 ; i < n ; i ++) lookup [i][0] = arr [i]; // Build sparse table for (let j = 1 ; j <= Math . floor (Math . log2 (n)); j ++) for (let i = 0 ; i <= n - (1 << j); i ++) lookup [i][j] = gcd (lookup [i][j - 1], lookup [i + (1 << (j - 1))][j - 1]); return lookup ; } // Returns GCD of arr[L..R] function query (L , R , lookup) { // Find highest power of 2 that is smaller // than or equal to count of elements const j = Math . floor (Math . log2 (R - L + 1)); // Compute GCD of last 2^j elements with first // 2^j elements in range. return gcd (lookup [L][j], lookup [R - (1 << j) + 1][j]); } function solveQueries (arr , queries) { const n = arr . length ; const m = queries . length ; const result = new Array (m); // Build the sparse table const lookup = buildSparseTable (arr); // Process each query for (let i = 0 ; i < m ; i ++) { const L = queries [i][0]; const R = queries [i][1]; result [i] = query (L , R , lookup); } return result ; } function gcd (a , b) { return b === 0 ? a : gcd (b , a % b); } const arr = [2 , 3 , 5 , 4 , 6 , 8]; const queries = [[0 , 2], [3 , 5], [2 , 3]]; const res = solveQueries (arr , queries); console . log (res . join (" ")); Output 1 2 1 Time Complexity: O(n * log n) Auxiliary Space: O(n * log n) Comment Article Tags: Article Tags: Misc Advanced Data Structure Technical Scripter DSA Arrays array-range-queries + 2 More