Lazy Propagation in Segment Tree - GeeksforGeeks

Source: https://www.geeksforgeeks.org/lazy-propagation-in-segment-tree/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Lazy Propagation in Segment Tree Last Updated : 23 Jul, 2025 Segment tree is introduced in previous post with an example of range sum problem. We have used the same "Sum of given Range" problem to explain Lazy propagation How does update work in Simple Segment Tree? In the previous post , update function was called to update only a single value in array. Please note that a single value update in array may cause multiple updates in Segment Tree as there may be many segment tree nodes that have a single array element in their ranges. Below is simple logic used in previous post. 1) Start with root of segment tree. 2) If array index to be updated is not in current node's range, then return 3) Else update current node and recur for children. Below is code taken from previous post. CPP /* A recursive function to update the nodes which have the given index in their range. The following are parameters tree[] --> segment tree si --> index of current node in segment tree. Initial value is passed as 0. ss and se --> Starting and ending indexes of array elements covered under this node of segment tree. Initial values passed as 0 and n-1. i --> index of the element to be updated. This index is in input array. diff --> Value to be added to all nodes which have array index i in range */ void updateValueUtil (int tree [], int ss , int se , int i , int diff , int si) { // Base Case: If the input index lies outside the range // of this segment if (i < ss || i > se) return ; // If the input index is in range of this node, then // update the value of the node and its children st [si] = st [si] + diff ; if (se != ss) { int mid = getMid (ss , se); updateValueUtil (st , ss , mid , i , diff , 2 * si + 1); updateValueUtil (st , mid + 1 , se , i , diff , 2 * si + 2); } } JavaScript /* A recursive function to update the nodes which have the given index in their range. The following are parameters tree[] --> segment tree si --> index of current node in segment tree. Initial value is passed as 0. ss and se --> Starting and ending indexes of array elements covered under this node of segment tree. Initial values passed as 0 and n-1. i --> index of the element to be updated. This index is in input array. diff --> Value to be added to all nodes which have array index i in range */ function updateValueUtil (tree , ss , se , i , diff , si) { // Base Case: If the input index lies outside the range // of this segment if (i < ss || i > se) return ; // If the input index is in range of this node, then // update the value of the node and its children st [si] = st [si] + diff ; if (se != ss) { let mid = getMid (ss , se); updateValueUtil (st , ss , mid , i , diff , 2 * si + 1); updateValueUtil (st , mid + 1 , se , i , diff , 2 * si + 2); } } // The code is contributed by Arushi jindal. Java /* A recursive function to update the nodes which have the given index in their range. The following are parameters tree[] --> segment tree si --> index of current node in segment tree. Initial value is passed as 0. ss and se --> Starting and ending indexes of array elements covered under this node of segment tree. Initial values passed as 0 and n-1. i --> index of the element to be updated. This index is in input array. diff --> Value to be added to all nodes which have array index i in range */ public static void updateValueUtil (int [] tree , int ss , int se , int i , int diff , int si) { // Base Case: If the input index lies outside the range // of this segment if (i < ss || i > se) return ; // If the input index is in range of this node, then // update the value of the node and its children st [si] = st [si] + diff ; if (se != ss) { int mid = getMid (ss , se); updateValueUtil (st , ss , mid , i , diff , 2 * si + 1); updateValueUtil (st , mid + 1 , se , i , diff , 2 * si + 2); } } // The code is contrbuted by Arushi Jindal. Python3 # /* A recursive function to update the nodes which have the given # index in their range. The following are parameters # tree[] --> segment tree # si --> index of current node in segment tree. # Initial value is passed as 0. # ss and se --> Starting and ending indexes of array elements # covered under this node of segment tree. # Initial values passed as 0 and n-1. # i --> index of the element to be updated. This index # is in input array. # diff --> Value to be added to all nodes

which have array # index i in range */ def updateValueUtil (tree , ss , se , i , diff , si): # Base Case: If the input index lies outside the range # of this segment if i < ss or i > se : return # If the input index is in range of this node, then # update the value of the node and its children st [si] = st [si] + diff if se != ss : mid = getMid (ss , se) updateValueUtil (st , ss , mid , i , diff , 2 * si + 1) updateValueUtil (st , mid + 1 , se , i , diff , 2 * si + 2) # The code is contributed by Arushi jindal. C# /* A recursive function to update the nodes which have the given index in their range. The following are parameters tree[] --> segment tree si --> index of current node in segment tree. Initial value is passed as 0. ss and se --> Starting and ending indexes of array elements covered under this node of segment tree. Initial values passed as 0 and n-1. i --> index of the element to be updated. This index is in input array. diff --> Value to be added to all nodes which have array index i in range */ public static void updateValueUtil (int [] tree , int ss , int se , int i , int diff , int si) { // Base Case: If the input index lies outside the range // of this segment if (i < ss || i > se) return ; // If the input index is in range of this node, then // update the value of the node and its children st [si] = st [si] + diff ; if (se != ss) { int mid = getMid (ss , se); updateValueUtil (st , ss , mid , i , diff , 2 * si + 1); updateValueUtil (st , mid + 1 , se , i , diff , 2 * si + 2); } } // The code is contrbuted by Arushi Jindal. What if there are updates on a range of array indexes? For example add 10 to all values at indexes from 2 to 7 in array. The above update has to be called for every index from 2 to 7. We can avoid multiple calls by writing a function updateRange() that updates nodes accordingly. CPP /* Function to update segment tree for range update in input array. si -> index of current node in segment tree ss and se -> Starting and ending indexes of elements for which current nodes stores sum. us and ue -> starting and ending indexes of update query diff -> which we need to add in the range us to ue */ void updateRangeUtil (int si , int ss , int se , int us , int ue , int diff) { // out of range if (ss > se || ss > ue || se < us) return ; // Current node is a leaf node if (ss == se) { // Add the difference to current node tree [si] += diff ; return ; } // If not a leaf node, recur for children. int mid = (ss + se) / 2 ; updateRangeUtil (si * 2 + 1 , ss , mid , us , ue , diff); updateRangeUtil (si * 2 + 2 , mid + 1 , se , us , ue , diff); // Use the result of children calls to update this // node tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2]; } Java void updateRangeUtil (int si , int ss , int se , int us , int ue , int diff) { scss Copy code // out of range if (ss > se || ss > ue || se < us) return ; // Current node is a leaf node if (ss == se) { // Add the difference to current node tree [si] += diff ; return ; } // If not a leaf node, recur for children. int mid = (ss + se) / 2 ; updateRangeUtil (si * 2 + 1 , ss , mid , us , ue , diff); updateRangeUtil (si * 2 + 2 , mid + 1 , se , us , ue , diff); // Use the result of children calls to update this // node tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2] ; } Python3 # Python def updateRangeUtil (si , ss , se , us , ue , diff): # out of range if (ss > se or ss > ue or se < us): return # Current node is a leaf node if (ss == se): # Add the difference to current node tree [si] += diff return # If not a leaf node, recur for children. mid = (ss + se) // 2 updateRangeUtil (si * 2 + 1 , ss , mid , us , ue , diff) updateRangeUtil (si * 2 + 2 , mid + 1 , se , us , ue , diff) # Use the result of children calls to update this # node tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2] C# /* Function to update segment tree for range update in input array. si -> index of current node in segment tree ss and se -> Starting and ending indexes of elements for which current nodes stores sum. us and ue -> starting and ending indexes of update query diff -> which we need to add in the range us to ue */ void UpdateRangeUtil (int si , int ss , int se , int us , int ue , int diff) { // out of range if (ss > se || ss > ue || se < us) return ; // Current node is a leaf node if (ss == se) { // Add the difference to current node tree [si] += diff ; return ; } // If not a leaf node, recur for children. int mid = (ss + se) / 2 ; UpdateRangeUtil (si * 2 + 1 , ss , mid , us , ue , diff); UpdateRangeUtil (si * 2 + 2 , mid + 1 , se , us , ue , diff); // Use the result of children calls to update this node tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2]; } JavaScript /* Function to update segment tree for range update in input array. si -> index of current node in segment tree ss and se -> Starting and ending indexes of elements for which current nodes stores sum. us and ue -> starting and ending indexes of update query diff -> which we need to add in the range us to ue */ function updateRangeUtil (si , ss , se , us , ue , diff) { // out of range if (ss > se || ss > ue || se < us) return ; // Current node is a leaf node if (ss == se) { // Add the difference to current node tree [si] += diff ; return ; } // If not a leaf node, recur for children. let mid = Math . floor ((ss + se) / 2); updateRangeUtil (si * 2 + 1 , ss , mid , us , ue , diff); updateRangeUtil (si * 2 + 2 , mid + 1 , se , us , ue , diff); // Use the result of children calls to update this // node tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2]; } // The code is contributed by Arushii Goel. Lazy Propagation - An optimization to make range updates faster When there are many updates and updates are done on a range, we can postpone some updates (avoid recursive calls in update) and do those updates only when required. Please remember that a node in segment tree stores or represents result of a query for a range of indexes. And if this node's range lies within the update operation range, then all descendants of the node must also be updated. For example consider the node with value 27 in above diagram, this node

stores sum of values at indexes from 3 to 5. If our update query is for range 2 to 5, then we need to update this node and all descendants of this node. With Lazy propagation, we update only node with value 27 and postpone updates to its children by storing this update information in separate nodes called lazy nodes or values. We create an array lazy[] which represents lazy node. Size of lazy[] is same as array that represents segment tree, which is tree[] in below code. The idea is to initialize all elements of lazy[] as 0. A value 0 in lazy[i] indicates that there are no pending updates on node i in segment tree. A non-zero value of lazy[i] means that this amount needs to be added to node i in segment tree before making any query to the node. Below is modified update method. // To update segment tree for change in array // values at array indexes from us to ue. updateRange(us, ue) 1) If current segment tree node has any pending update, then first add that pending update to current node. 2) If current node's range lies completely in update query range. ....a) Update current node ....b) Postpone updates to children by setting lazy value for children nodes. 3) If current node's range overlaps with update range, follow the same approach as above simple update. ...a) Recur for left and right children. ...b) Update current node using results of left and right calls. Is there any change in Query Function also? Since we have changed update to postpone its operations, there may be problems if a query is made to a node that is yet to be updated. So we need to update our query method also which is getSumUtil in previous post . The getSumUtil() now first checks if there is a pending update and if there is, then updates the node. Once it makes sure that pending update is done, it works same as the previous getSumUtil(). Below are programs to demonstrate working of Lazy Propagation. C++ // Program to show segment tree to demonstrate lazy // propagation #include <iostream> using namespace std ; #define MAX 1000 // Ideally, we should not use global variables and large // constant-sized arrays, we have done it here for simplicity. int tree [MAX] = { 0 }; // To store segment tree int lazy [MAX] = { 0 }; // To store pending updates /* si -> index of current node in segment tree ss and se -> Starting and ending indexes of elements for which current nodes stores sum. us and ue -> starting and ending indexes of update query diff -> which we need to add in the range us to ue */ void updateRangeUtil (int si , int ss , int se , int us , int ue , int diff) { // If lazy value is non-zero for current node of segment // tree, then there are some pending updates. So we need // to make sure that the pending updates are done before // making new updates. Because this value may be used by // parent after recursive calls (See last line of this // function) if (lazy [si] != 0) { // Make pending updates using value stored in lazy // nodes tree [si] += (se - ss + 1) * lazy [si]; // checking if it is not leaf node because if // it is leaf node then we cannot go further if (ss != se) { // We can postpone updating children we don't // need their new values now. // Since we are not yet updating children of si, // we need to set lazy flags for the children lazy [si * 2 + 1] += lazy [si]; lazy [si * 2 + 2] += lazy [si]; } // Set the lazy value for current node as 0 as it // has been updated lazy [si] = 0 ; } // out of range if (ss > se || ss > ue || se < us) return ; // Current segment is fully in range if (ss >= us && se <= ue) { // Add the difference to current node tree [si] += (se - ss + 1) * diff ; // same logic for checking leaf node or not if (ss != se) { // This is where we store values in lazy nodes, // rather than updating the segment tree itself // Since we don't need these updated values now // we postpone updates by storing values in lazy[] lazy [si * 2 + 1] += diff ; lazy [si * 2 + 2] += diff ; } return ; } // If not completely in rang, but overlaps, recur for // children, int mid = (ss + se) / 2 ; updateRangeUtil (si * 2 + 1 , ss , mid , us , ue , diff); updateRangeUtil (si * 2 + 2 , mid + 1 , se , us , ue , diff); // And use the result of children calls to update this // node tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2]; } // Function to update a range of values in segment // tree /* us and eu -> starting and ending indexes of update query ue -> ending index of update query diff -> which we need to add in the range us to ue */ void updateRange (int n , int us , int ue , int diff) { updateRangeUtil (0 , 0 , n -1 , us , ue , diff); } /* A recursive function to get the sum of values in given range of the array. The following are parameters for this function. si --> Index of current node in the segment tree. Initially 0 is passed as root is always at' index 0 ss & se --> Starting and ending indexes of the segment represented by current node, i.e., tree[si] qs & qe --> Starting and ending indexes of query range */ int getSumUtil (int ss , int se , int qs , int qe , int si) { // If lazy flag is set for current node of segment tree, // then there are some pending updates. So we need to // make sure that the pending updates are done before // processing the sub sum query if (lazy [si] != 0) { // Make pending updates to this node. Note that this // node represents sum of elements in arr[ss..se] and // all these elements must be increased by lazy[si] tree [si] += (se - ss + 1) * lazy [si]; // checking if it is not leaf node because if // it is leaf node then we cannot go further if (ss != se) { // Since we are not yet updating children os si, // we need to set lazy values for the children lazy [si * 2 + 1] += lazy [si]; lazy [si * 2 + 2] += lazy [si]; } // unset the lazy value for current node as it has // been updated lazy [si] = 0 ; } // Out of range if (ss > se || ss > qe || se < qs) return 0 ; // At this point we are sure that pending

lazy updates // are done for current node. So we can return value // (same as it was for query in our previous post) // If this segment lies in range if (ss >= qs && se <= qe) return tree [si]; // If a part of this segment overlaps with the given // range int mid = (ss + se) / 2 ; return getSumUtil (ss , mid , qs , qe , 2 * si + 1) + getSumUtil (mid + 1 , se , qs , qe , 2 * si + 2); } // Return sum of elements in range from index qs (query // start) to qe (query end). It mainly uses getSumUtil() int getSum (int n , int qs , int qe) { // Check for erroneous input values if (qs < 0 || qe > n -1 || qs > qe) { cout << "Invalid Input" ; return -1 ; } return getSumUtil (0 , n -1 , qs , qe , 0); } // A recursive function that constructs Segment Tree for // array[ss..se]. si is index of current node in segment // tree st. void constructSTUtil (int arr [], int ss , int se , int si) { // out of range as ss can never be greater than se if (ss > se) return ; // If there is one element in array, store it in // current node of segment tree and return if (ss == se) { tree [si] = arr [ss]; return ; } // If there are more than one elements, then recur // for left and right subtrees and store the sum // of values in this node int mid = (ss + se) / 2 ; constructSTUtil (arr , ss , mid , si * 2 + 1); constructSTUtil (arr , mid + 1 , se , si * 2 + 2); tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2]; } /* Function to construct segment tree from given array. This function allocates memory for segment tree and calls constructSTUtil() to fill the allocated memory */ void constructST (int arr [], int n) { // Fill the allocated memory st constructSTUtil (arr , 0 , n -1 , 0); } // Driver program to test above functions int main () { int arr [] = { 1 , 3 , 5 , 7 , 9 , 11 }; int n = sizeof (arr) / sizeof (arr [0]); // Build segment tree from given array constructST (arr , n); // Print sum of values in array from index 1 to 3 cout << "Sum of values in given range = " << getSum (n , 1 , 3); // Add 10 to all nodes at indexes from 1 to 5. updateRange (n , 1 , 5 , 10); // Find sum after the value is updated cout << " \n Updated sum of values in given range = " << getSum (n , 1 , 3); return 0 ; } // This code is contributed by shivanisinghss2110 C // Program to show segment tree to demonstrate lazy // propagation #include <stdio.h> #include <math.h> #define MAX 1000 // Ideally, we should not use global variables and large // constant-sized arrays, we have done it here for simplicity. int tree [MAX] = { 0 }; // To store segment tree int lazy [MAX] = { 0 }; // To store pending updates /* si -> index of current node in segment tree ss and se -> Starting and ending indexes of elements for which current nodes stores sum. us and ue -> starting and ending indexes of update query diff -> which we need to add in the range us to ue */ void updateRangeUtil (int si , int ss , int se , int us , int ue , int diff) { // If lazy value is non-zero for current node of segment // tree, then there are some pending updates. So we need // to make sure that the pending updates are done before // making new updates. Because this value may be used by // parent after recursive calls (See last line of this // function) if (lazy [si] != 0) { // Make pending updates using value stored in lazy // nodes tree [si] += (se - ss + 1) * lazy [si]; // checking if it is not leaf node because if // it is leaf node then we cannot go further if (ss != se) { // We can postpone updating children we don't // need their new values now. // Since we are not yet updating children of si, // we need to set lazy flags for the children lazy [si * 2 + 1] += lazy [si]; lazy [si * 2 + 2] += lazy [si]; } // Set the lazy value for current node as 0 as it // has been updated lazy [si] = 0 ; } // out of range if (ss > se || ss > ue || se < us) return ; // Current segment is fully in range if (ss >= us && se <= ue) { // Add the difference to current node tree [si] += (se - ss + 1) * diff ; // same logic for checking leaf node or not if (ss != se) { // This is where we store values in lazy nodes, // rather than updating the segment tree itself // Since we don't need these updated values now // we postpone updates by storing values in lazy[] lazy [si * 2 + 1] += diff ; lazy [si * 2 + 2] += diff ; } return ; } // If not completely in rang, but overlaps, recur for // children, int mid = (ss + se) / 2 ; updateRangeUtil (si * 2 + 1 , ss , mid , us , ue , diff); updateRangeUtil (si * 2 + 2 , mid + 1 , se , us , ue , diff); // And use the result of children calls to update this // node tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2]; } // Function to update a range of values in segment // tree /* us and eu -> starting and ending indexes of update query ue -> ending index of update query diff -> which we need to add in the range us to ue */ void updateRange (int n , int us , int ue , int diff) { updateRangeUtil (0 , 0 , n -1 , us , ue , diff); } /* A recursive function to get the sum of values in given range of the array. The following are parameters for this function. si --> Index of current node in the segment tree. Initially 0 is passed as root is always at' index 0 ss & se --> Starting and ending indexes of the segment represented by current node, i.e., tree[si] qs & qe --> Starting and ending indexes of query range */ int getSumUtil (int ss , int se , int qs , int qe , int si) { // If lazy flag is set for current node of segment tree, // then there are some pending updates. So we need to // make sure that the pending updates are done before // processing the sub sum query if (lazy [si] != 0) { // Make pending updates to this node. Note that this // node represents sum of elements in arr[ss..se] and // all these elements must be increased by lazy[si] tree [si] += (se - ss + 1) * lazy [si]; // checking if it is not leaf node because if // it is leaf node then we cannot go further if (ss != se) { // Since we are not yet updating children os si, // we need to set lazy values for the children lazy [si * 2 + 1] += lazy [si];

lazy [si * 2 + 2] += lazy [si]; } // unset the lazy value for current node as it has // been updated lazy [si] = 0 ; } // Out of range if (ss > se || ss > qe || se < qs) return 0 ; // At this point we are sure that pending lazy updates // are done for current node. So we can return value // (same as it was for query in our previous post) // If this segment lies in range if (ss >= qs && se <= qe) return tree [si]; // If a part of this segment overlaps with the given // range int mid = (ss + se) / 2 ; return getSumUtil (ss , mid , qs , qe , 2 * si + 1) + getSumUtil (mid + 1 , se , qs , qe , 2 * si + 2); } // Return sum of elements in range from index qs (query // start) to qe (query end). It mainly uses getSumUtil() int getSum (int n , int qs , int qe) { // Check for erroneous input values if (qs < 0 || qe > n -1 || qs > qe) { printf ("Invalid Input"); return -1 ; } return getSumUtil (0 , n -1 , qs , qe , 0); } // A recursive function that constructs Segment Tree for // array[ss..se]. si is index of current node in segment // tree st. void constructSTUtil (int arr [], int ss , int se , int si) { // out of range as ss can never be greater than se if (ss > se) return ; // If there is one element in array, store it in // current node of segment tree and return if (ss == se) { tree [si] = arr [ss]; return ; } // If there are more than one elements, then recur // for left and right subtrees and store the sum // of values in this node int mid = (ss + se) / 2 ; constructSTUtil (arr , ss , mid , si * 2 + 1); constructSTUtil (arr , mid + 1 , se , si * 2 + 2); tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2]; } /* Function to construct segment tree from given array. This function allocates memory for segment tree and calls constructSTUtil() to fill the allocated memory */ void constructST (int arr [], int n) { // Fill the allocated memory st constructSTUtil (arr , 0 , n -1 , 0); } // Driver program to test above functions int main () { int arr [] = { 1 , 3 , 5 , 7 , 9 , 11 }; int n = sizeof (arr) / sizeof (arr [0]); // Build segment tree from given array constructST (arr , n); // Print sum of values in array from index 1 to 3 printf ("Sum of values in given range = %d \n " , getSum (n , 1 , 3)); // Add 10 to all nodes at indexes from 1 to 5. updateRange (n , 1 , 5 , 10); // Find sum after the value is updated printf ("Updated sum of values in given range = %d \n " , getSum (n , 1 , 3)); return 0 ; } Java // Java program to demonstrate lazy propagation in segment tree class LazySegmentTree { final int MAX = 1000 ; // Max tree size int tree [] = new int [MAX] ; // To store segment tree int lazy [] = new int [MAX] ; // To store pending updates /* si -> index of current node in segment tree ss and se -> Starting and ending indexes of elements for which current nodes stores sum. us and eu -> starting and ending indexes of update query ue -> ending index of update query diff -> which we need to add in the range us to ue */ void updateRangeUtil (int si , int ss , int se , int us , int ue , int diff) { // If lazy value is non-zero for current node of segment // tree, then there are some pending updates. So we need // to make sure that the pending updates are done before // making new updates. Because this value may be used by // parent after recursive calls (See last line of this // function) if (lazy [si] != 0) { // Make pending updates using value stored in lazy // nodes tree [si] += (se - ss + 1) * lazy [si] ; // checking if it is not leaf node because if // it is leaf node then we cannot go further if (ss != se) { // We can postpone updating children we don't // need their new values now. // Since we are not yet updating children of si, // we need to set lazy flags for the children lazy [si * 2 + 1] += lazy [si] ; lazy [si * 2 + 2] += lazy [si] ; } // Set the lazy value for current node as 0 as it // has been updated lazy [si] = 0 ; } // out of range if (ss > se || ss > ue || se < us) return ; // Current segment is fully in range if (ss >= us && se <= ue) { // Add the difference to current node tree [si] += (se - ss + 1) * diff ; // same logic for checking leaf node or not if (ss != se) { // This is where we store values in lazy nodes, // rather than updating the segment tree itself // Since we don't need these updated values now // we postpone updates by storing values in lazy[] lazy [si * 2 + 1] += diff ; lazy [si * 2 + 2] += diff ; } return ; } // If not completely in rang, but overlaps, recur for // children, int mid = (ss + se) / 2 ; updateRangeUtil (si * 2 + 1 , ss , mid , us , ue , diff); updateRangeUtil (si * 2 + 2 , mid + 1 , se , us , ue , diff); // And use the result of children calls to update this // node tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2] ; } // Function to update a range of values in segment // tree /* us and eu -> starting and ending indexes of update query ue -> ending index of update query diff -> which we need to add in the range us to ue */ void updateRange (int n , int us , int ue , int diff) { updateRangeUtil (0 , 0 , n - 1 , us , ue , diff); } /* A recursive function to get the sum of values in given range of the array. The following are parameters for this function. si --> Index of current node in the segment tree. Initially 0 is passed as root is always at' index 0 ss & se --> Starting and ending indexes of the segment represented by current node, i.e., tree[si] qs & qe --> Starting and ending indexes of query range */ int getSumUtil (int ss , int se , int qs , int qe , int si) { // If lazy flag is set for current node of segment tree, // then there are some pending updates. So we need to // make sure that the pending updates are done before // processing the sub sum query if (lazy [si] != 0) { // Make pending updates to this node. Note that this // node represents sum of elements in arr[ss..se] and // all these elements must be increased by lazy[si] tree [si] += (se - ss + 1) * lazy [si] ; // checking if it is not leaf node because if // it is leaf node then we cannot go further if (ss != se) { // Since we are not yet updating children os si, // we need

to set lazy values for the children lazy [si * 2 + 1] += lazy [si] ; lazy [si * 2 + 2] += lazy [si] ; } // unset the lazy value for current node as it has // been updated lazy [si] = 0 ; } // Out of range if (ss > se || ss > qe || se < qs) return 0 ; // At this point sure, pending lazy updates are done // for current node. So we can return value (same as // was for query in our previous post) // If this segment lies in range if (ss >= qs && se <= qe) return tree [si] ; // If a part of this segment overlaps with the given // range int mid = (ss + se) / 2 ; return getSumUtil (ss , mid , qs , qe , 2 * si + 1) + getSumUtil (mid + 1 , se , qs , qe , 2 * si + 2); } // Return sum of elements in range from index qs (query // start) to qe (query end). It mainly uses getSumUtil() int getSum (int n , int qs , int qe) { // Check for erroneous input values if (qs < 0 || qe > n - 1 || qs > qe) { System . out . println ("Invalid Input"); return - 1 ; } return getSumUtil (0 , n - 1 , qs , qe , 0); } /* A recursive function that constructs Segment Tree for array[ss..se]. si is index of current node in segment tree st. */ void constructSTUtil (int arr [] , int ss , int se , int si) { // out of range as ss can never be greater than se if (ss > se) return ; /* If there is one element in array, store it in current node of segment tree and return */ if (ss == se) { tree [si] = arr [ss] ; return ; } /* If there are more than one elements, then recur for left and right subtrees and store the sum of values in this node */ int mid = (ss + se) / 2 ; constructSTUtil (arr , ss , mid , si * 2 + 1); constructSTUtil (arr , mid + 1 , se , si * 2 + 2); tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2] ; } /* Function to construct segment tree from given array. This function allocates memory for segment tree and calls constructSTUtil() to fill the allocated memory */ void constructST (int arr [] , int n) { // Fill the allocated memory st constructSTUtil (arr , 0 , n - 1 , 0); } // Driver program to test above functions public static void main (String args []) { int arr [] = { 1 , 3 , 5 , 7 , 9 , 11 }; int n = arr . length ; LazySegmentTree tree = new LazySegmentTree (); // Build segment tree from given array tree . constructST (arr , n); // Print sum of values in array from index 1 to 3 System . out . println ("Sum of values in given range = " + tree . getSum (n , 1 , 3)); // Add 10 to all nodes at indexes from 1 to 5. tree . updateRange (n , 1 , 5 , 10); // Find sum after the value is updated System . out . println ("Updated sum of values in given range = " + tree . getSum (n , 1 , 3)); } } // This Code is contributed by Ankur Narain Verma Python3 # Python3 implementation of the approach MAX = 1000 # Ideally, we should not use global variables # and large constant-sized arrays, we have # done it here for simplicity. tree = [0] * MAX ; # To store segment tree lazy = [0] * MAX ; # To store pending updates """ si -> index of current node in segment tree ss and se -> Starting and ending indexes of elements for which current nodes stores sum. us and ue -> starting and ending indexes of update query diff -> which we need to add in the range us to ue """ def updateRangeUtil (si , ss , se , us , ue , diff) : # If lazy value is non-zero for current node # of segment tree, then there are some # pending updates. So we need to make sure # that the pending updates are done before # making new updates. Because this value may be # used by parent after recursive calls # (See last line of this function) if (lazy [si] != 0) : # Make pending updates using value # stored in lazy nodes tree [si] += (se - ss + 1) * lazy [si]; # checking if it is not leaf node because if # it is leaf node then we cannot go further if (ss != se) : # We can postpone updating children we don't # need their new values now. # Since we are not yet updating children of si, # we need to set lazy flags for the children lazy [si * 2 + 1] += lazy [si]; lazy [si * 2 + 2] += lazy [si]; # Set the lazy value for current node # as 0 as it has been updated lazy [si] = 0 ; # out of range if (ss > se or ss > ue or se < us) : return ; # Current segment is fully in range if (ss >= us and se <= ue) : # Add the difference to current node tree [si] += (se - ss + 1) * diff ; # same logic for checking leaf node or not if (ss != se) : # This is where we store values in lazy nodes, # rather than updating the segment tree itself # Since we don't need these updated values now # we postpone updates by storing values in lazy[] lazy [si * 2 + 1] += diff ; lazy [si * 2 + 2] += diff ; return ; # If not completely in rang, but overlaps, # recur for children, mid = (ss + se) // 2 ; updateRangeUtil (si * 2 + 1 , ss , mid , us , ue , diff); updateRangeUtil (si * 2 + 2 , mid + 1 , se , us , ue , diff); # And use the result of children calls # to update this node tree [si] = tree [si * 2 + 1] + \ tree [si * 2 + 2]; # Function to update a range of values # in segment tree ''' us and eu -> starting and ending indexes of update query ue -> ending index of update query diff -> which we need to add in the range us to ue ''' def updateRange (n , us , ue , diff) : updateRangeUtil (0 , 0 , n - 1 , us , ue , diff); ''' A recursive function to get the sum of values in given range of the array. The following are parameters for this function. si --> Index of current node in the segment tree. Initially 0 is passed as root is always at' index 0 ss & se --> Starting and ending indexes of the segment represented by current node, i.e., tree[si] qs & qe --> Starting and ending indexes of query range ''' def getSumUtil (ss , se , qs , qe , si) : # If lazy flag is set for current node # of segment tree, then there are # some pending updates. So we need to # make sure that the pending updates are # done before processing the sub sum query if (lazy [si] != 0) : # Make pending updates to this node. # Note that this node represents sum of # elements in arr[ss..se] and all these # elements must be increased by lazy[si] tree [si] += (se - ss + 1) * lazy [si

]; # checking if it is not leaf node because if # it is leaf node then we cannot go further if (ss != se) : # Since we are not yet updating children os si, # we need to set lazy values for the children lazy [si * 2 + 1] += lazy [si]; lazy [si * 2 + 2] += lazy [si]; # unset the lazy value for current node # as it has been updated lazy [si] = 0 ; # Out of range if (ss > se or ss > qe or se < qs) : return 0 ; # At this point we are sure that # pending lazy updates are done for # current node. So we can return value # (same as it was for query in our previous post) # If this segment lies in range if (ss >= qs and se <= qe) : return tree [si]; # If a part of this segment overlaps # with the given range mid = (ss + se) // 2 ; return (getSumUtil (ss , mid , qs , qe , 2 * si + 1) + getSumUtil (mid + 1 , se , qs , qe , 2 * si + 2)); # Return sum of elements in range from # index qs (query start) to qe (query end). # It mainly uses getSumUtil() def getSum (n , qs , qe) : # Check for erroneous input values if (qs < 0 or qe > n - 1 or qs > qe) : print ("Invalid Input"); return - 1 ; return getSumUtil (0 , n - 1 , qs , qe , 0); # A recursive function that constructs # Segment Tree for array[ss..se]. # si is index of current node in segment # tree st. def constructSTUtil (arr , ss , se , si) : # out of range as ss can never be # greater than se if (ss > se) : return ; # If there is one element in array, # store it in current node of # segment tree and return if (ss == se) : tree [si] = arr [ss]; return ; # If there are more than one elements, # then recur for left and right subtrees # and store the sum of values in this node mid = (ss + se) // 2 ; constructSTUtil (arr , ss , mid , si * 2 + 1); constructSTUtil (arr , mid + 1 , se , si * 2 + 2); tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2]; ''' Function to construct segment tree from given array. This function allocates memory for segment tree and calls constructSTUtil() to fill the allocated memory ''' def constructST (arr , n) : # Fill the allocated memory st constructSTUtil (arr , 0 , n - 1 , 0); # Driver Code if __name__ == "__main__" : arr = [1 , 3 , 5 , 7 , 9 , 11]; n = len (arr); # Build segment tree from given array constructST (arr , n); # Print sum of values in array from index 1 to 3 print ("Sum of values in given range =" , getSum (n , 1 , 3)); # Add 10 to all nodes at indexes from 1 to 5. updateRange (n , 1 , 5 , 10); # Find sum after the value is updated print ("Updated sum of values in given range =" , getSum (n , 1 , 3)); # This code is contributed by AnkitRai01 C# // C# program to demonstrate lazy // propagation in segment tree using System ; public class LazySegmentTree { static readonly int MAX = 1000 ; // Max tree size int [] tree = new int [MAX]; // To store segment tree int [] lazy = new int [MAX]; // To store pending updates /* si -> index of current node in segment tree ss and se -> Starting and ending indexes of elements for which current nodes stores sum. us and eu -> starting and ending indexes of update query ue -> ending index of update query diff -> which we need to add in the range us to ue */ void updateRangeUtil (int si , int ss , int se , int us , int ue , int diff) { // If lazy value is non-zero // for current node of segment // tree, then there are some // pending updates. So we need // to make sure that the pending // updates are done before making // new updates. Because this // value may be used by parent // after recursive calls (See last // line of this function) if (lazy [si] != 0) { // Make pending updates using value // stored in lazy nodes tree [si] += (se - ss + 1) * lazy [si]; // checking if it is not leaf node because if // it is leaf node then we cannot go further if (ss != se) { // We can postpone updating children // we don't need their new values now. // Since we are not yet updating children of si, // we need to set lazy flags for the children lazy [si * 2 + 1] += lazy [si]; lazy [si * 2 + 2] += lazy [si]; } // Set the lazy value for current node // as 0 as it has been updated lazy [si] = 0 ; } // out of range if (ss > se || ss > ue || se < us) return ; // Current segment is fully in range if (ss >= us && se <= ue) { // Add the difference to current node tree [si] += (se - ss + 1) * diff ; // same logic for checking leaf node or not if (ss != se) { // This is where we store values in lazy nodes, // rather than updating the segment tree itself // Since we don't need these updated values now // we postpone updates by storing values in lazy[] lazy [si * 2 + 1] += diff ; lazy [si * 2 + 2] += diff ; } return ; } // If not completely in rang, but // overlaps, recur for children, int mid = (ss + se) / 2 ; updateRangeUtil (si * 2 + 1 , ss , mid , us , ue , diff); updateRangeUtil (si * 2 + 2 , mid + 1 , se , us , ue , diff); // And use the result of children calls to update this // node tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2]; } // Function to update a range of values in segment // tree /* us and eu -> starting and ending indexes of update query ue -> ending index of update query diff -> which we need to add in the range us to ue */ void updateRange (int n , int us , int ue , int diff) { updateRangeUtil (0 , 0 , n - 1 , us , ue , diff); } /* A recursive function to get the sum of values in given range of the array. The following are parameters for this function. si --> Index of current node in the segment tree. Initially 0 is passed as root is always at' index 0 ss & se --> Starting and ending indexes of the segment represented by current node, i.e., tree[si] qs & qe --> Starting and ending indexes of query range */ int getSumUtil (int ss , int se , int qs , int qe , int si) { // If lazy flag is set for current node // of segment tree, then there are // some pending updates. So we need to // make sure that the pending updates // are done before processing // the sub sum query if (lazy [si] != 0) { // Make pending updates to this // node. Note that this node // represents sum of elements // in arr[ss..se] and all these // elements must

be increased by lazy[si] tree [si] += (se - ss + 1) * lazy [si]; // checking if it is not leaf node because if // it is leaf node then we cannot go further if (ss != se) { // Since we are not yet // updating children os si, // we need to set lazy values // for the children lazy [si * 2 + 1] += lazy [si]; lazy [si * 2 + 2] += lazy [si]; } // unset the lazy value for current // node as it has been updated lazy [si] = 0 ; } // Out of range if (ss > se || ss > qe || se < qs) return 0 ; // At this point sure, pending lazy updates are done // for current node. So we can return value (same as // was for query in our previous post) // If this segment lies in range if (ss >= qs && se <= qe) return tree [si]; // If a part of this segment overlaps // with the given range int mid = (ss + se) / 2 ; return getSumUtil (ss , mid , qs , qe , 2 * si + 1) + getSumUtil (mid + 1 , se , qs , qe , 2 * si + 2); } // Return sum of elements in range from index qs (query // start) to qe (query end). It mainly uses getSumUtil() int getSum (int n , int qs , int qe) { // Check for erroneous input values if (qs < 0 || qe > n - 1 || qs > qe) { Console . WriteLine ("Invalid Input"); return - 1 ; } return getSumUtil (0 , n - 1 , qs , qe , 0); } /* A recursive function that constructs Segment Tree for array[ss..se]. si is index of current node in segment tree st. */ void constructSTUtil (int [] arr , int ss , int se , int si) { // out of range as ss can // never be greater than se if (ss > se) return ; /* If there is one element in array, store it in current node of segment tree and return */ if (ss == se) { tree [si] = arr [ss]; return ; } /* If there are more than one elements, then recur for left and right subtrees and store the sum of values in this node */ int mid = (ss + se) / 2 ; constructSTUtil (arr , ss , mid , si * 2 + 1); constructSTUtil (arr , mid + 1 , se , si * 2 + 2); tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2]; } /* Function to construct segment tree from given array. This function allocates memory for segment tree and calls constructSTUtil() to fill the allocated memory */ void constructST (int [] arr , int n) { // Fill the allocated memory st constructSTUtil (arr , 0 , n - 1 , 0); } // Driver program to test above functions public static void Main (String [] args) { int [] arr = { 1 , 3 , 5 , 7 , 9 , 11 }; int n = arr . Length ; LazySegmentTree tree = new LazySegmentTree (); // Build segment tree from given array tree . constructST (arr , n); // Print sum of values in array from index 1 to 3 Console . WriteLine ("Sum of values in given range = " + tree . getSum (n , 1 , 3)); // Add 10 to all nodes at indexes from 1 to 5. tree . updateRange (n , 1 , 5 , 10); // Find sum after the value is updated Console . WriteLine ("Updated sum of values in given range = " + tree . getSum (n , 1 , 3)); } } // This code contributed by Rajput-Ji JavaScript // JS program to demonstrate lazy // propagation in segment tree const MAX = 1000 ; // Ideally, we should not use global variables and large // constant-sized arrays, we have done it here for simplicity. const tree = new Array (MAX). fill (0); // To store segment tree const lazy = new Array (MAX). fill (0); // To store pending updates /* si -> index of current node in segment tree ss and se -> Starting and ending indexes of elements for which current nodes stores sum. us and ue -> starting and ending indexes of update query diff -> which we need to add in the range us to ue */ function updateRangeUtil (si , ss , se , us , ue , diff) { // If lazy value is non-zero for current node of segment // tree, then there are some pending updates. So we need // to make sure that the pending updates are done before // making new updates. Because this value may be used by // parent after recursive calls (See last line of this // function) if (lazy [si] != 0) { // Make pending updates using value stored in lazy // nodes tree [si] += (se - ss + 1) * lazy [si]; // checking if it is not leaf node because if // it is leaf node then we cannot go further if (ss != se) { // We can postpone updating children we don't // need their new values now. // Since we are not yet updating children of si, // we need to set lazy flags for the children lazy [si * 2 + 1] += lazy [si]; lazy [si * 2 + 2] += lazy [si]; } // Set the lazy value for current node as 0 as it // has been updated lazy [si] = 0 ; } // out of range if (ss > se || ss > ue || se < us) { return ; } // Current segment is fully in range if (ss >= us && se <= ue) { // Add the difference to current node tree [si] += (se - ss + 1) * diff ; // same logic for checking leaf node or not if (ss != se) { // This is where we store values in lazy nodes, // rather than updating the segment tree itself // Since we don't need these updated values now // we postpone updates by storing values in lazy[] lazy [si * 2 + 1] += diff ; lazy [si * 2 + 2] += diff ; } return ; } // If not completely in rang, but overlaps, recur for // children, const mid = Math . floor ((ss + se) / 2); updateRangeUtil (si * 2 + 1 , ss , mid , us , ue , diff); updateRangeUtil (si * 2 + 2 , mid + 1 , se , us , ue , diff); // And use the result of children calls to update this // node tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2]; } // Function to update a range of values in segment // tree /* us and eu -> starting and ending indexes of update query ue -> ending index of update query diff -> which we need to add in the range us to ue */ function updateRange (n , us , ue , diff) { updateRangeUtil (0 , 0 , n - 1 , us , ue , diff); } /* A recursive function to get the sum of values in given range of the array. The following are parameters for this function. si --> Index of current node in the segment tree. Initially 0 is passed as root is always at' index 0 ss & se --> Starting and ending indexes of the segment represented by current node, i.e., tree[si] qs & qe --> Starting and ending indexes of query range */ function getSumUtil (ss , se , qs , qe , si) { // If lazy flag is set for current node of segment tree, // then there are some pending updates. So

we need to // make sure that the pending updates are done before // processing the sub sum query if (lazy [si] != 0) { // Make pending updates to this node. Note that this // node represents sum of elements in arr[ss..se] and // all these elements must be increased by lazy[si] tree [si] += (se - ss + 1) * lazy [si]; // checking if it is not leaf node because if // it is leaf node then we cannot go further if (ss != se) { // Since we are not yet updating children os si, // we need to set lazy values for the children lazy [si * 2 + 1] += lazy [si]; lazy [si * 2 + 2] += lazy [si]; } // unset the lazy value for current node as it has // been updated lazy [si] = 0 ; } // Out of range if (ss > se || ss > qe || se < qs) { return 0 ; } // At this point we are sure that pending lazy updates // are done for current node. So we can return value // (same as it was for query in our previous post) // If this segment lies in range if (ss >= qs && se <= qe) { return tree [si]; } // If a part of this segment overlaps with the given // range const mid = Math . floor ((ss + se) / 2); return getSumUtil (ss , mid , qs , qe , 2 * si + 1) + getSumUtil (mid + 1 , se , qs , qe , 2 * si + 2); } // Return sum of elements in range from index qs (query // start) to qe (query end). It mainly uses getSumUtil() function getSum (n , qs , qe) { // Check for erroneous input values if (qs < 0 || qe > n - 1 || qs > qe) { console . log ("Invalid Input"); return - 1 ; } return getSumUtil (0 , n - 1 , qs , qe , 0); } // A recursive function that constructs Segment Tree for // array[ss..se]. si is index of current node in segment // tree st. function constructSTUtil (arr , ss , se , si) { // out of range as ss can never be greater than se if (ss > se) { return ; } // If there is one element in array, store it in // current node of segment tree and return if (ss == se) { tree [si] = arr [ss]; return ; } // If there are more than one elements, then recur // for left and right subtrees and store the sum // of values in this node let mid = Math . floor ((ss + se) / 2); constructSTUtil (arr , ss , mid , si * 2 + 1); constructSTUtil (arr , mid + 1 , se , si * 2 + 2); tree [si] = tree [si * 2 + 1] + tree [si * 2 + 2]; } /* Function to construct segment tree from given array. This function allocates memory for segment tree and calls constructSTUtil() to fill the allocated memory */ function constructST (arr , n) { // Fill the allocated memory st constructSTUtil (arr , 0 , n - 1 , 0); } // Driver program to test above functions let arr = [1 , 3 , 5 , 7 , 9 , 11]; let n = arr . length ; // Build segment tree from given array constructST (arr , n); // Print sum of values in array from index 1 to 3 console . log ("Sum of values in given range =" , getSum (n , 1 , 3)); // Add 10 to all nodes at indexes from 1 to 5. updateRange (n , 1 , 5 , 10); // Find sum after the value is updated console . log ("Updated sum of values in given range =" , getSum (n , 1 , 3)); Output: Sum of values in given range = 15 Updated sum of values in given range = 45 Time Complexity: O(n) Auxiliary Space: O(MAX) Related Topic: Segment Tree Comment Article Tags: Article Tags: Advanced Data Structure DSA array-range-queries Segment-Tree