Ternary Search - GeeksforGeeks

Source: https://www.geeksforgeeks.org/ternary-search/

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 Ternary Search Last Updated : 19 Sep, 2025 Ternary search is a divide-and-conquer search algorithm used to find the position of a target value within a monotonically increasing or decreasing function or in a unimodal array (e.g., U-shaped or ∩-shaped). Unlike binary search, which splits the array into two parts, ternary search divides the range into three equal parts by choosing two mid-points: mid1 = l + (r - l) / 3 mid2 = r - (r - l) / 3 When to use Ternary Search When working with a sorted or unimodal array (e.g., strictly decreasing then increasing, or vice versa). When you need to optimize a unimodal function (e.g., finding the minimum or maximum of a quadratic function). When solving problems like finding the bitonic point in a bitonic sequence. When evaluating a real-valued function where the function has a single local minimum or maximum. When solving certain geometric or numeric optimization problems (e.g., finding minimum distance, cost, etc.). Working of Ternary Search Given an array that first strictly decreases and then strictly increases, we want to find the index of the minimum element. This kind of array is known as a U-shaped or unimodal array. Input: arr [] = [9, 7, 5, 2, 3, 6, 10] Output: 3 Explanation: The minimum of the given array is 2, which is at index 3. Try it on GfG Practice Binary search works well for monotonic arrays (strictly increasing or strictly decreasing), but here the array is split into two parts: A decreasing segment An increasing segment This is exactly where ternary search fits best. It repeatedly narrows the search space by checking two middle points (mid1, mid2) and discarding one-third of the range based on the values at those points. Ternary Search Logic (for this problem): Let mid1 and mid2 be two points dividing the current interval into three equal parts: If arr[mid1] < arr[mid2], the minimum lies to the left of mid2 If arr[mid1] > arr[mid2], the minimum lies to the right of mid1 We continue this until we reduce the search space to a small enough size to directly find the minimum. Step by Step Implementation: Initialize low = 0 and high = n - 1. Create a variable minIndex = -1 to store the answer. Start a loop that runs while low <= high. Calculate mid1 = low + (high - low) / 3. Calculate mid2 = high - (high - low) / 3. If arr[mid1] == arr[mid2], then: => Set low = mid1 + 1. => Set high = mid2 - 1. => Set minIndex = mid1. Else if arr[mid1] < arr[mid2], then: => Set high = mid2 - 1. => Set minIndex = mid1. Else (i.e., arr[mid1] > arr[mid2]), then: => Set low = mid1 + 1. => Set minIndex = mid2. Repeat the loop until low > high. Return minIndex as the result. C++ #include <iostream> #include <vector> using namespace std ; int findMinIndex (vector < int >& arr) { int low = 0 , high = arr . size () - 1 ; int minIndex = -1 ; while (low <= high) { // divide the range into three parts int mid1 = low + (high - low) / 3 ; int mid2 = high - (high - low) / 3 ; // if both mid1 and mid2 point to equal // values narrow the search if (arr [mid1] == arr [mid2]) { // Move towards the center low = mid1 + 1 ; high = mid2 - 1 ; // tentatively store mid1 as // potential minimum minIndex = mid1 ; } // if arr[mid1] < arr[mid2], the minimum lies in the // left part (including mid1) else if (arr [mid1] < arr [mid2]) { high = mid2 - 1 ; // update with better candidate minIndex = mid1 ; } // is arr[mid1] > arr[mid2], the minimum lies in the // right part (including mid2) else { low = mid1 + 1 ; // update with better candidate minIndex = mid2 ; } } return minIndex ; } int main () { vector < int > arr = { 9 , 7 , 1 , 2 , 3 , 6 , 10 }; int idx = findMinIndex (arr); cout << idx << endl ; return 0 ; } C #include <stdio.h> // Function to find the index of the minimum element int findMinIndex (int arr [], int n) { int low = 0 , high = n - 1 ; int minIndex = -1 ; while (low <= high) { // divide the range into three parts int mid1 = low + (high - low) / 3 ; int mid2 = high - (high - low) / 3 ; // if both mid1 and mid2 point to equal // values narrow the search if (arr [mid1] == arr [mid2]) { // Move towards the center low = mid1 + 1 ; high = mid2 - 1 ; // tentatively store mid1 as // potential minimum minIndex = mid1 ; } // if arr[mid1] < arr[mid2], the minimum lies in the // left part

(including mid1) else if (arr [mid1] < arr [mid2]) { high = mid2 - 1 ; // update with better candidate minIndex = mid1 ; } // is arr[mid1] > arr[mid2], the minimum lies in the // right part (including mid2) else { low = mid1 + 1 ; // update with better candidate minIndex = mid2 ; } } return minIndex ; } int main () { int arr [] = { 9 , 7 , 1 , 2 , 3 , 6 , 10 }; int n = sizeof (arr) / sizeof (arr [0]); int idx = findMinIndex (arr , n); printf ("%d \n " , idx); return 0 ; } Java class GfG { public static int findMinIndex (int [] arr) { int low = 0 , high = arr . length - 1 ; int minIndex = - 1 ; while (low <= high) { // divide the range into three parts int mid1 = low + (high - low) / 3 ; int mid2 = high - (high - low) / 3 ; // if both mid1 and mid2 point to equal // values narrow the search if (arr [mid1] == arr [mid2]) { // Move towards the center low = mid1 + 1 ; high = mid2 - 1 ; // tentatively store mid1 as // potential minimum minIndex = mid1 ; } // if arr[mid1] < arr[mid2], the minimum lies in the // left part (including mid1) else if (arr [mid1] < arr [mid2]) { high = mid2 - 1 ; // update with better candidate minIndex = mid1 ; } // is arr[mid1] > arr[mid2], the minimum lies in the // right part (including mid2) else { low = mid1 + 1 ; // update with better candidate minIndex = mid2 ; } } return minIndex ; } public static void main (String [] args) { int [] arr = { 9 , 7 , 1 , 2 , 3 , 6 , 10 }; int idx = findMinIndex (arr); System . out . println (idx); } } Python def findMinIndex (arr): low = 0 high = len (arr) - 1 minIndex = - 1 while low <= high : # divide the range into three parts mid1 = low + (high - low) // 3 mid2 = high - (high - low) // 3 # if both mid1 and mid2 point to equal # values narrow the search if arr [mid1] == arr [mid2]: # Move towards the center low = mid1 + 1 high = mid2 - 1 # tentatively store mid1 as # potential minimum minIndex = mid1 # if arr[mid1] < arr[mid2], the minimum lies in the # left part (including mid1) elif arr [mid1] < arr [mid2]: high = mid2 - 1 # update with better candidate minIndex = mid1 # is arr[mid1] > arr[mid2], the minimum lies in the # right part (including mid2) else : low = mid1 + 1 # update with better candidate minIndex = mid2 return minIndex def main (): arr = [9 , 7 , 1 , 2 , 3 , 6 , 10] idx = findMinIndex (arr) print (idx) C# using System ; class GfG { public static int findMinIndex (int [] arr) { int low = 0 , high = arr . Length - 1 ; int minIndex = - 1 ; while (low <= high) { // divide the range into three parts int mid1 = low + (high - low) / 3 ; int mid2 = high - (high - low) / 3 ; // if both arr[mid1] and arr[mid2] point to equal // values narrow the search if (arr [mid1] == arr [mid2]) { // Move towards the center low = mid1 + 1 ; high = mid2 - 1 ; // tentatively store mid1 as // potential minimum minIndex = mid1 ; } // if arr[mid1] < arr[mid2], the minimum lies in the // left part (including mid1) else if (arr [mid1] < arr [mid2]) { high = mid2 - 1 ; // update with better candidate minIndex = mid1 ; } // is mid1 > mid2, the minimum lies in the // right part (including mid2) else { low = mid1 + 1 ; // update with better candidate minIndex = mid2 ; } } return minIndex ; } public static void Main (string [] args) { int [] arr = { 9 , 7 , 1 , 2 , 3 , 6 , 10 }; int idx = findMinIndex (arr); Console . WriteLine (idx); } } JavaScript function findMinIndex (arr) { let low = 0 , high = arr . length - 1 ; let minIndex = - 1 ; while (low <= high) { // divide the range into three parts let mid1 = low + Math . floor ((high - low) / 3); let mid2 = high - Math . floor ((high - low) / 3); // if both arr[mid1] and arr[mid2] point to equal // values narrow the search if (arr [mid1] === arr [mid2]) { // Move towards the center low = mid1 + 1 ; high = mid2 - 1 ; // tentatively store mid1 as // potential minimum minIndex = mid1 ; } // if arr[mid1] < arr[mid2], the minimum lies in the // left part (including mid1) else if (arr [mid1] < arr [mid2]) { high = mid2 - 1 ; // update with better candidate minIndex = mid1 ; } // is mid1 > mid2, the minimum lies in the // right part (including mid2) else { low = mid1 + 1 ; // update with better candidate minIndex = mid2 ; } } return minIndex ; } // Driver Code const arr = [9 , 7 , 1 , 2 , 3 , 6 , 10]; const idx = findMinIndex (arr); console . log (idx); Output 2 Time Complexity: O(2 × log 3 n) Auxiliary Space: 1 Complexity Analysis of Ternary Search Time Complexity: Worst case: O(log 3 n) Average case: Θ(log 3 n) Best case: Ω(1) Auxiliary Space: O(1) Refer to here for more details Binary search Vs Ternary Search The time complexity of ternary search is slightly worse than binary search, primarily because ternary search performs more comparisons per iteration. Binary search is ideal for finding values or extrema in monotonic functions (strictly increasing or decreasing), as it splits the search space into two parts. On the other hand, ternary search is better suited for finding the maximum or minimum in unimodal functions , where the function first increases and then decreases (or vice versa). Note: Ternary search can also be applied to monotonic functions, but it is generally less efficient than binary search due to its higher number of comparisons. Refer to here for more details Note: The array or function domain must follow a specific structure (e.g., sorted, monotonic, or unimodal) for ternary search to work correctly. Comment Article Tags: Article Tags: DSA