Activity Selection - GeeksforGeeks

Source: https://www.geeksforgeeks.org/activity-selection-problem-greedy-algo-1/

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 Activity Selection Last Updated : 8 Sep, 2025 Given two arrays start[] and finish[] , representing the start and finish times of activities. A person can perform only one activity at a time, and an activity can be performed only if its start time is greater than the finish time of the last chosen activity. Find the maximum number of activities that can be performed without overlapping. Examples: Input: start[] = [1, 3, 0, 5, 8, 5], finish[] = [2, 4, 6, 7, 9, 9] Output: 4 Explanation: A person can perform at most four activities. The maximum set of activities that can be performed is {0, 1, 3, 4} (these are the indexes in the start[] and finish[] arrays). Input: start[] = [10, 12, 20], finish[] = [20, 25, 30] Output: 1 Explanation: A person can perform at most one activity. Try it on GfG Practice Table of Content [Naive Approach] Generate All Subsets - O(2^n) Time and O(n) Space [Expected Approach 1] - Using Sorting - O(n * log(n)) Time and O(n) Space [Expected Approach 2] - Using Priority Queue - O(n * log(n)) Time and O(n) Space [Naive Approach] Generate All Subsets - O(2 n) Time and O(n) Space Generates all possible subsets of activities, where each subset represents a potential selection. For each subset, we check if the activities are mutually non-overlapping by comparing their time intervals pairwise. If a subset is valid and contains more activities than our current maximum, we update the maximum count. In the end, we get the size of the largest valid subset. Note: Generating all possible subsets and checks them one by one will lead to an exponential time complexity, which becomes impossible to compute for larger values of n. Hence, this approach is not practical. [Expected Approach 1] - Using Sorting - O(n * log(n)) Time and O(n) Space The problem can be solved using a greedy approach . The idea is that whenever multiple activities overlap, we should pick the one that finishes earliest, because finishing sooner leaves the most room to schedule upcoming activities. This way, the schedule stays flexible, and more activities can fit without conflict. To achieve this, we sort all activities by their finishing times. Then, starting with the first one, we repeatedly pick the next activity that both starts after the last selected activity ends and has the earliest finish time among the remaining ones. C++ #include <iostream> #include <vector> #include <algorithm> using namespace std ; bool compare (const vector < int >& a , const vector < int >& b) { return a [1] < b [1]; } int activitySelection (vector < int > & start , vector < int > & finish) { vector < vector < int >> arr ; for (int i = 0 ; i < start . size (); i ++) { arr . push_back ({ start [i], finish [i]}); } // Sort activities by finish time sort (arr . begin (), arr . end (), compare); // At least one activity can be performed int count = 1 ; // Index of last selected activity int j = 0 ; for (int i = 1 ; i < arr . size (); i ++) { // Check if current activity starts // after last selected activity finishes if (arr [i][0] > arr [j][1]) { count ++ ; // Update last selected activity j = i ; } } return count ; } int main () { vector < int > start = { 1 , 3 , 0 , 5 , 8 , 5 }; vector < int > finish = { 2 , 4 , 6 , 7 , 9 , 9 }; cout << activitySelection (start , finish); return 0 ; } Java import java.util.Arrays ; import java.util.Comparator ; public class GfG { public static int activitySelection (int [] start , int [] finish) { int n = start . length ; int [][] arr = new int [n][2] ; for (int i = 0 ; i < n ; i ++) { arr [i][0] = start [i] ; arr [i][1] = finish [i] ; } // Sort activities by finish time Arrays . sort (arr , Comparator . comparingInt (a -> a [1])); // At least one activity can be performed int count = 1 ; // Index of last selected activity int j = 0 ; for (int i = 1 ; i < n ; i ++) { // Check if current activity starts // after last selected activity finishes if (arr [i][0] > arr [j][1]) { count ++ ; // Update last selected activity j = i ; } } return count ; } public static void main (String [] args) { int [] start = { 1 , 3 , 0 , 5 , 8 , 5 }; int [] finish = { 2 , 4 , 6 , 7 , 9 , 9 }; System . out . println (activitySelection (start , finish)); } } Python def activitySelection (start , finish): arr = list (zip (start , finish)) # Sort activities by finish time arr . sort (key = lambda x : x [1]) # At least one activity can be

performed count = 1 # Index of last selected activity j = 0 for i in range (1 , len (arr)): # Check if current activity starts # after last selected activity finishes if arr [i][0] > arr [j][1]: count += 1 # Update last selected activity j = i return count if __name__ == '__main__' : start = [1 , 3 , 0 , 5 , 8 , 5] finish = [2 , 4 , 6 , 7 , 9 , 9] print (activitySelection (start , finish)) C# using System ; using System.Collections.Generic ; public class GfG { public static int activitySelection (int [] start , int [] finish) { List < List < int >> arr = new List < List < int >> (); for (int i = 0 ; i < start . Length ; i ++) { arr . Add (new List < int > { start [i], finish [i] }); } // Sort activities by finish time arr . Sort ((a , b) => a [1]. CompareTo (b [1])); // At least one activity can be performed int count = 1 ; // Index of last selected activity int j = 0 ; for (int i = 1 ; i < arr . Count ; i ++) { // Check if current activity starts // after last selected activity finishes if (arr [i][0] > arr [j][1]) { count ++ ; // Update last selected activity j = i ; } } return count ; } public static void Main () { int [] start = new int [] { 1 , 3 , 0 , 5 , 8 , 5 }; int [] finish = new int [] { 2 , 4 , 6 , 7 , 9 , 9 }; Console . WriteLine (activitySelection (start , finish)); } } JavaScript function activitySelection (start , finish) { let arr = []; for (let i = 0 ; i < start . length ; i ++) { arr . push ([start [i], finish [i]]); } // Sort activities by finish time arr . sort ((a , b) => a [1] - b [1]); // At least one activity can be performed let count = 1 ; // Index of last selected activity let j = 0 ; for (let i = 1 ; i < arr . length ; i ++) { // Check if current activity starts // after last selected activity finishes if (arr [i][0] > arr [j][1]) { count ++ ; // Update last selected activity j = i ; } } return count ; } // Driver Code const start = [1 , 3 , 0 , 5 , 8 , 5]; const finish = [2 , 4 , 6 , 7 , 9 , 9]; console . log (activitySelection (start , finish)); Output 4 [Expected Approach 2] - Using Priority Queue - O(n * log(n)) Time and O(n) Space Instead of sorting, we can also use a min-heap (priority queue) to directly process activities in order of their finishing time. By storing each activity as (finish, start) in the heap, the activity with the earliest finish time is always available at the top. We then apply the same greedy rule as above: pick it if its start time is after the last chosen activity's finish, update the finish time, and continue. C++ #include <iostream> #include <vector> #include <queue> using namespace std ; int activitySelection (vector < int > & start , vector < int > & finish) { int ans = 0 ; // Minimum Priority Queue to sort activities in // ascending order of finishing time (end[i]). priority_queue < pair < int , int > , vector < pair < int , int >> , greater < pair < int , int >>> p ; for (int i = 0 ; i < start . size (); i ++) { p . push (make_pair (finish [i], start [i])); } // to store the end time of last activity int finishtime = -1 ; while (! p . empty ()) { pair < int , int > activity = p . top (); p . pop (); if (activity . second > finishtime) { finishtime = activity . first ; ans ++ ; } } return ans ; } int main () { vector < int > start = { 1 , 3 , 0 , 5 , 8 , 5 }; vector < int > finish = { 2 , 4 , 6 , 7 , 9 , 9 }; cout << activitySelection (start , finish); return 0 ; } Java import java.util.PriorityQueue ; class GfG { static int activitySelection (int [] start , int [] finish) { int n = start . length ; int ans = 0 ; // Min Heap to store activities in ascending order // of finish time PriorityQueue < int []> p = new PriorityQueue <> ((a , b) -> Integer . compare (a [0] , b [0])); for (int i = 0 ; i < n ; i ++) { p . add (new int [] { finish [i] , start [i] }); } // to store the end time of the last selected activity int finishtime = - 1 ; while (! p . isEmpty ()) { int [] activity = p . poll (); if (activity [1] > finishtime) { finishtime = activity [0] ; ans ++ ; } } return ans ; } public static void main (String [] args) { int [] start = { 1 , 3 , 0 , 5 , 8 , 5 }; int [] finish = { 2 , 4 , 6 , 7 , 9 , 9 }; System . out . println (activitySelection (start , finish)); } } Python import heapq def activitySelection (start , finish): ans = 0 # Minimum Priority Queue to sort activities in # ascending order of finishing time (end[i]). p = [] for i in range (len (start)): heapq . heappush (p , (finish [i], start [i])) # to store the end time of last activity finishtime = - 1 while p : activity = heapq . heappop (p) if activity [1] > finishtime : finishtime = activity [0] ans += 1 return ans if __name__ == "__main__" : start = [1 , 3 , 0 , 5 , 8 , 5] finish = [2 , 4 , 6 , 7 , 9 , 9] print (activitySelection (start , finish)) C# using System ; using System.Collections.Generic ; class MinHeap { private List < (int end , int start) > heap = new List < (int end , int start) > (); // Get parent/child index helpers private int Parent (int i) => (i - 1) / 2 ; private int Left (int i) => 2 * i + 1 ; private int Right (int i) => 2 * i + 2 ; private void Swap (int i , int j) { var temp = heap [i]; heap [i] = heap [j]; heap [j] = temp ; } // Push element into heap public void Push ((int end , int start) val) { heap . Add (val); int i = heap . Count - 1 ; // Fix heap property (bubble up) while (i > 0 && heap [Parent (i)]. end > heap [i]. end) { Swap (i , Parent (i)); i = Parent (i); } } // Pop smallest element (by end time) public (int end , int start) Pop () { if (heap . Count == 0) throw new InvalidOperationException ("Heap is empty"); var root = heap [0]; heap [0] = heap [heap . Count - 1]; heap . RemoveAt (heap . Count - 1); Heapify (0); return root ; } // Heapify from index i private void Heapify (int i) { int l = Left (i), r = Right (i), smallest = i ; if (l < heap . Count && heap [l]. end < heap [smallest]. end) smallest = l ; if (r < heap . Count && heap [r]. end < heap [smallest]. end) smallest = r ; if (smallest != i) { Swap (i , smallest); Heapify (smallest); } } public int Count => heap . Count ; } class GfG { static int activitySelection (int [] start , int [] finish) { int ans = 0 ; // Custom MinHeap (by end time) MinHeap pq = new MinHeap (); for (int i = 0 ; i < start . Length ; i ++) { pq . Push ((finish [i], start

[i])); } // to store the end time of last activity int finishtime = - 1 ; while (pq . Count > 0) { // get activity with smallest end time var activity = pq . Pop (); if (activity . start > finishtime) { finishtime = activity . end ; ans ++ ; } } return ans ; } static void Main () { int [] start = { 1 , 3 , 0 , 5 , 8 , 5 }; int [] finish = { 2 , 4 , 6 , 7 , 9 , 9 }; Console . WriteLine (activitySelection (start , finish)); } } JavaScript function activitySelection (start , finish) { let ans = 0 ; // Priority queue to store activities // based on finishing time. let activities = new PriorityQueue (); for (let i = 0 ; i < start . length ; i ++) { activities . enqueue ({ finish : finish [i], start : start [i] }, finish [i]); } // to store the end time of last activity let finishtime = - 1 ; while (! activities . isEmpty ()) { let activity = activities . dequeue (); if (activity . start > finishtime) { finishtime = activity . finish ; ans ++ ; } } return ans ; } class PriorityQueue { constructor () { this . elements = []; } enqueue (element , priority) { let added = false ; for (let i = 0 ; i < this . elements . length ; i ++) { if (priority < this . elements [i]. priority) { this . elements . splice (i , 0 , { element , priority }); added = true ; break ; } } if (! added) { this . elements . push ({ element , priority }); } } dequeue () { return this . elements . shift (). element ; } isEmpty () { return this . elements . length === 0 ; } } // Driver Code let start = [1 , 3 , 0 , 5 , 8 , 5]; let finish = [2 , 4 , 6 , 7 , 9 , 9]; console . log (activitySelection (start , finish)); Output 4 Comment Article Tags: Article Tags: Greedy DSA Amazon Activity Selection Problem Morgan Stanley Flipkart Facebook MakeMyTrip Visa + 5 More