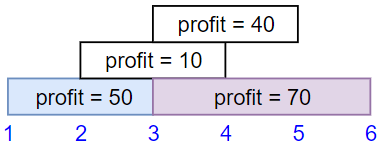
<https://leetcode.com/problems/maximum-profit-in-job-scheduling/description/>

We have n jobs, where every job is scheduled to be done from startTime[i] to endTime[i], obtaining a profit of profit[i].

You're given the startTime, endTime and profit arrays, return the maximum profit you can take such that there are no two jobs in the subset with overlapping time range.

If you choose a job that ends at time X you will be able to start another job that starts at time X.

**Example 1:**

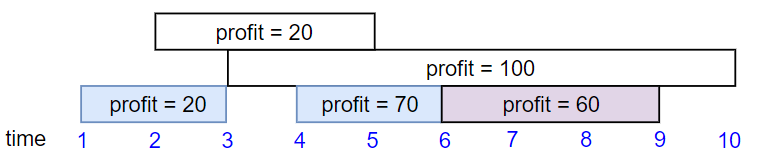


**Input:** startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70]

**Output:** 120

**Explanation:** The subset chosen is the first and fourth job. Time range [1-3]+[3-6] , we get profit of 120 = 50 + 70.

**Example 2:**

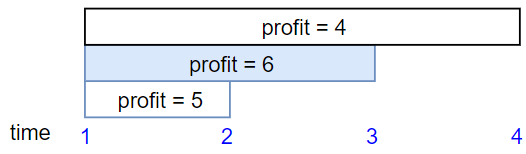


**Input:** startTime = [1,2,3,4,6], endTime = [3,5,10,6,9], profit = [20,20,100,70,60]

**Output:** 150

**Explanation:** The subset chosen is the first, fourth and fifth job. Profit obtained 150 = 20 + 70 + 60.

**Example 3:**



**Input:** startTime = [1,1,1], endTime = [2,3,4], profit = [5,6,4]

**Output:** 6

**Constraints:**

* 1 <= startTime.length == endTime.length == profit.length <= 5 \* 10^4
* 1 <= startTime[i] < endTime[i] <= 10^9
* 1 <= profit[i] <= 10^4

**Attempt 1: 2023-12-18**

**The solution exactly same way as L2008.Maximum Earnings From Taxi, just need to convert 'startTime', 'endTime', 'profit' into 'jobs' array, which is similar to 'rides' array in L2008**

**Solution 1: Native DFS (10min, TLE, 18/31)**

class Solution {

    // The solution exactly same way as L2008.Maximum Earnings From Taxi,

    // just need to convert 'startTime', 'endTime', 'profit' into 'jobs' array,

    // which is similar to 'rides' array in L2008

    public int jobScheduling(int[] startTime, int[] endTime, int[] profit) {

        // Convert given arrays into 'jobs' array

        int n = startTime.length;

        int[][] jobs = new int[n][3];

        for(int i = 0; i < n; i++) {

            jobs[i][0] = startTime[i];

            jobs[i][1] = endTime[i];

            jobs[i][2] = profit[i];

        }

        // Sort 'jobs' array based on 'start' time

        Arrays.sort(jobs, (a, b) -> a[0] - b[0]);

        return helper(jobs, 0);

    }

    private int helper(int[][] jobs, int index) {

        if(index >= jobs.length) {

            return 0;

        }

        int notTakeJob = helper(jobs, index + 1);

        int i;

        for(i = index + 1; i < jobs.length; i++) {

            if(jobs[i][0] >= jobs[index][1]) {

                break;

            }

        }

        int takeJob = jobs[index][2] + helper(jobs, i);

        return Math.max(notTakeJob, takeJob);

    }

}

Time Complexity: O(2^m), m is the number of jobs

Each recursion level we will take two branches, "not take job" and "take job", so worst case is we have to go through m depth recursion, then time complexity is O(m^2)

Space Complexity: O(m)

Recursion Stack: The depth of recursion may go up to m in the worst case if all jobs are taken sequentially with no overlap, which contributes to O(m) space complexity.

**Solution 2: DFS + Memoization (10min)**

class Solution {

    // The solution exactly same way as L2008.Maximum Earnings From Taxi,

    // just need to convert 'startTime', 'endTime', 'profit' into 'jobs' array,

    // which is similar to 'rides' array in L2008

    public int jobScheduling(int[] startTime, int[] endTime, int[] profit) {

        // Convert given arrays into 'jobs' array

        int n = startTime.length;

        int[][] jobs = new int[n][3];

        for(int i = 0; i < n; i++) {

            jobs[i][0] = startTime[i];

            jobs[i][1] = endTime[i];

            jobs[i][2] = profit[i];

        }

        // Sort 'jobs' array based on 'start' time

        Arrays.sort(jobs, (a, b) -> a[0] - b[0]);

        Integer[] memo = new Integer[n];

        return helper(jobs, 0, memo);

    }

    private int helper(int[][] jobs, int index, Integer[] memo) {

        if(index >= jobs.length) {

            return 0;

        }

        if(memo[index] != null) {

            return memo[index];

        }

        int notTakeJob = helper(jobs, index + 1, memo);

        int i;

        for(i = index + 1; i < jobs.length; i++) {

            if(jobs[i][0] >= jobs[index][1]) {

                break;

            }

        }

        int takeJob = jobs[index][2] + helper(jobs, i, memo);

        return memo[index] = Math.max(notTakeJob, takeJob);

    }

}

Time Complexity: O(m^2), m is the number of jobs

First m is for worst case may need m recursion on depth, second m is for each recursion internally we have a for loop to find the next recursion entry

Space Complexity: O(m), m is the number of jobs

Memoization: The dfs function uses a memoization, which will store results for each unique argument (i). This could result in up to m entries in the worst case, giving a space complexity of O(m).

Recursion Stack: The depth of recursion may go up to m in the worst case if all jobs are taken sequentially with no overlap, which contributes to O(m) space complexity.

So both are O(m), final cost is O(m)

**Solution 3: DFS + Memoization + Binary Search (10min)**

class Solution {

    // The solution exactly same way as L2008.Maximum Earnings From Taxi,

    // just need to convert 'startTime', 'endTime', 'profit' into 'jobs' array,

    // which is similar to 'rides' array in L2008

    public int jobScheduling(int[] startTime, int[] endTime, int[] profit) {

        // Convert given arrays into 'jobs' array

        int n = startTime.length;

        int[][] jobs = new int[n][3];

        for(int i = 0; i < n; i++) {

            jobs[i][0] = startTime[i];

            jobs[i][1] = endTime[i];

            jobs[i][2] = profit[i];

        }

        // Sort 'jobs' array based on 'start' time

        Arrays.sort(jobs, (a, b) -> a[0] - b[0]);

        Integer[] memo = new Integer[n];

        return helper(jobs, 0, memo);

    }

    private int helper(int[][] jobs, int index, Integer[] memo) {

        if(index >= jobs.length) {

            return 0;

        }

        if(memo[index] != null) {

            return memo[index];

        }

        int notTakeJob = helper(jobs, index + 1, memo);

        int i = binarySearch(jobs, jobs[index][1], index + 1);

        int takeJob = jobs[index][2] + helper(jobs, i, memo);

        return memo[index] = Math.max(notTakeJob, takeJob);

    }

    // Find lower boundary

    private int binarySearch(int[][] jobs, int val, int lo) {

        int hi = jobs.length - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(jobs[mid][0] >= val) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return lo;

    }

}

Time Complexity: O(mlogm), m is the number of jobs

Sorting the Rides: The initial step in the code is to sort the jobs list, which consists of m jobs. This has a time complexity of O(m log m), where m is the number of jobs.

Dynamic Programming with Memoization (dfs function): The dfs function is used here to implement dynamic programming with memoization. The memoization ensures that each possible starting point of a job is computed at most once. Since memoization caches the results, the maximum number of distinct states to consider is m, the length of the sorted jobs list.

Binary Search (bisect\_left function): Inside the dfs function, binary search is used to find the next non-overlapping job, which has a time complexity of O(log m) per call.

Combining the dynamic programming computation with the binary search, for each of the m calls to dfs, a binary search operation is involved. Thus, each call may contribute up to O(log m) complexity. Given that there are m such calls, the total time complexity from the dynamic programming along with the binary searches is O(m log m).

Considering both sorting and the memoized dfs, the overall time complexity of the algorithm is O(m log m).

Space Complexity: O(m), m is the number of jobs

Sorting: Sorting is done in-place in Python (Timsort), but it may still require O(log m) space for the stack frames used in the sort implementation.

Memoization: The dfs function uses a memoization table (implicitly through the @cache decorator), which will store results for each unique argument (i). This could result in up to m entries in the worst case, giving a space complexity of O(m).

Recursion Stack: The depth of recursion may go up to m in the worst case if all jobs are taken sequentially with no overlap, which contributes to O(m) space complexity.

Thus, combining the space requirements for the sorting, memoization, and recursion stack, the total space complexity is O(m).

In conclusion, the time complexity of the code is O(m log m) and the space complexity is O(m), where m is the number of jobs.

**Solution 4: DP (10min)**

class Solution {

    // The solution exactly same way as L2008.Maximum Earnings From Taxi,

    // just need to convert 'startTime', 'endTime', 'profit' into 'jobs' array,

    // which is similar to 'rides' array in L2008

    public int jobScheduling(int[] startTime, int[] endTime, int[] profit) {

        // Convert given arrays into 'jobs' array

        int n = startTime.length;

        int[][] jobs = new int[n][3];

        for(int i = 0; i < n; i++) {

            jobs[i][0] = startTime[i];

            jobs[i][1] = endTime[i];

            jobs[i][2] = profit[i];

        }

        // Sort 'jobs' array based on 'start' time

        Arrays.sort(jobs, (a, b) -> a[0] - b[0]);

        int[] dp = new int[n + 1];

        dp[n] = 0;

        for(int index = n - 1; index >= 0; index--) {

            int i;

            for(i = index + 1; i < n; i++) {

                if(jobs[i][0] >= jobs[index][1]) {

                    break;

                }

            }

            dp[index] = Math.max(dp[index + 1], jobs[index][2] + dp[i]);

        }

        return dp[0];

    }

}

Time Complexity: O(m^2), m is the number of jobs

First m is for worst case may need m recursion on depth, second m is for each recursion internally we have a for loop to find the next recursion entry

Space Complexity: O(m), m is the number of jobs

Recursion Stack: The depth of recursion may go up to m in the worst case if all jobs are taken sequentially with no overlap, which contributes to O(m) space complexity.

**Solution 5: DP + Binary Search (10min)**

class Solution {

    // The solution exactly same way as L2008.Maximum Earnings From Taxi,

    // just need to convert 'startTime', 'endTime', 'profit' into 'jobs' array,

    // which is similar to 'rides' array in L2008

    public int jobScheduling(int[] startTime, int[] endTime, int[] profit) {

        // Convert given arrays into 'jobs' array

        int n = startTime.length;

        int[][] jobs = new int[n][3];

        for(int i = 0; i < n; i++) {

            jobs[i][0] = startTime[i];

            jobs[i][1] = endTime[i];

            jobs[i][2] = profit[i];

        }

        // Sort 'jobs' array based on 'start' time

        Arrays.sort(jobs, (a, b) -> a[0] - b[0]);

        int[] dp = new int[n + 1];

        dp[n] = 0;

        for(int index = n - 1; index >= 0; index--) {

            int i = binarySearch(jobs, jobs[index][1], index + 1);

            dp[index] = Math.max(dp[index + 1], jobs[index][2] + dp[i]);

        }

        return dp[0];

    }

    // Find lower boundary

    private int binarySearch(int[][] jobs, int val, int lo) {

        int hi = jobs.length - 1;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(jobs[mid][0] >= val) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return lo;

    }

}

Time Complexity: O(mlogm), m is the number of jobs

Sorting the Rides: The initial step in the code is to sort the jobs list, which consists of m jobs. This has a time complexity of O(m log m), where m is the number of jobs.

Dynamic Programming: O(m), single for loop on all jobs

Binary Search: O(logm)

Space Complexity: O(m), m is the number of jobs

Sorting: O(logm)

Dynamic Programming: O(m), single for loop on all rides

So Time Complexity is O(mlogm) and Space Complexity is O(m)