1235. Maximum Profit in Job Scheduling

Hard Array Binary Search Dynamic Programming Sorting Leetcode Link

Problem Description

In this scheduling problem, we are given `n` different jobs, each job has a start time, end time, and the profit it yields upon completion. Our primary goal is to find the maximum total profit earned by completing jobs that do not overlap in time. We are allowed to choose jobs in such a way that if one job ends at a certain time `X`, the next job can start exactly at `X`.

The difficulty lies in selecting the optimal set of jobs that maximize our profit while ensuring their time periods do not clash. This is akin to having a busy schedule and trying to fit in as many high-paying gigs as possible, without them overlapping, to maximize earnings.

Intuition

The trick to solving this problem lies in dynamic programming, which allows us to break down the problem into smaller sub-problems and build a solution incrementally.

Firstly, we need to sort the jobs by their end times to process them in order. This ordering will help address jobs sequentially and pick the best possible previous job that doesn't overlap.

With each job considered, we must decide whether to take this job along with the optimal set of jobs before it, or to skip it and stick with the previous optimal set. Here's where dynamic programming comes into play:

We use a Dynamic Programming (DP) table `dp` where each `dp[i]` represents the maximum profit that can be achieved by scheduling jobs up to the i-th job.

We iterate over each job and for each job look backward in time to find a job that finishes before the current one starts, and hence does not conflict with the current job. `bisect_right` is the standard binary search method used to find that point efficiently rather than scanning all previous jobs linearly, saving valuable computation time.

The update rule for our DP table is straightforward: we compare the profit of taking the current job (which includes adding its profit to the best result from jobs that end before it starts) versus not taking it (meaning we stick with the best result up to the previous job). Choosing the better of these two options yields our dp value for the current job.

The last value in our DP table will contain the maximum profit achievable after considering all jobs.

Solution Approach

The implementation of this problem utilizes a sorted list, binary search, and dynamic programming to calculate the maximum profit for scheduling non-overlapping jobs. Each step of the approach is essential for the efficiency and correctness of the solution:

1. **Sorting Jobs**: First, we pair each job's `endTime`, `startTime`, and `profit` together and sort these pairs based on their `endTime`. This is done to consider jobs by the order they finish. The sorting is achieved through the `sorted` function, which sorts the zipped lists of end times, start times, and profits.

   ```
   jobs = sorted(zip(endTime, startTime, profit))
   ```

2. **Dynamic Programming Table (`dp`)**: We define a DP table `dp` of size `n+1`, where `n` is the number of jobs to store the maximum profit up to each job. We initialize this table with zeros.

   ```
   dp = [0] * (n + 1)
   ```

3. **Binary Search**: For each job, we need to find the closest job that finished before the current job's start. This is where `bisect_right` from the `bisect` module is used. It performs a binary search to find the insertion point of the current job's start time in the sorted `jobs` array to ensure no overlap.

   ```
   j = bisect_right(jobs, s, hi=i, key=lambda x: x[0])
   ```

   The callback `lambda x: x[0]` is used to ensure the search operates on the `endTime` of the jobs. `j` is the index where the job can be inserted without conflict, so `dp[j]` corresponds to the maximum profit up to the job that finishes right before the current job can start.

4. **Profit Calculation**: For every job at index `i`, we perform a calculation to determine the maximum profit including the current job versus excluding it. To include the current job, we take the profit of the current job `p` and add it to the profit accumulated until the closest non-conflicting job, which is `dp[j]`. If this sum is greater than the maximum profit without the current job `dp[i]`, we update our `dp` at `i+1` with this larger value; else, we carry over the maximum profit without the current job.

   ```
   dp[i + 1] = max(dp[i], dp[j] + p)
   ```

5. **Result**: After considering all jobs, the maximum profit that can be achieved is stored at the end of the `dp` array. Therefore, the last entry `dp[n]` is returned as the final result.

   ```
   return dp[n]
   ```

By using these steps, the solution efficiently calculates the maximum profit without overlapping jobs by considering each job's contribution to the overall profit only if it adds value compared to the profit obtained without it. It balances between taking the new job or proceeding with the previously accumulated profit, resulting in an optimal scheduling strategy.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have the following set of jobs, with their respective start times, end times, and profits:

```
start time: [1, 2, 3, 4]
end time:   [3, 5, 6, 7]
profit:     [50, 20, 100, 70]
```

There are 4 jobs to consider. Following the solution approach:

1. **Sorting Jobs**: We create job tuples `(endTime, startTime, profit)` and sort them by `endTime` as follows:

   ```
   jobs = sorted(zip([3, 5, 6, 7], [1, 2, 3, 4], [50, 20, 100, 70]))
   // After sorting -> [(3, 1, 50), (5, 2, 20), (6, 3, 100), (7, 4, 70)]
   ```

2. **Dynamic Programming Table (`dp`)**: Here, `n` is 4, so `dp` will have 5 elements initialized to zero:

   ```
   dp = [0, 0, 0, 0, 0]
   ```

3. **Binary Search**: As we compute the `dp` value for each job, we'll need the latest non-conflicting job, found using binary search:
   - For the first job `(3, 1, 50)`, as it's the first there are no previous jobs, so `dp[1]` becomes 50.
   - For the second job `(5, 2, 20)`, binary search (`bisect_right`) will find no non-conflicting job since it overlaps with the first job, hence `dp[2]` is `max(dp[1], dp[0] + 20)` which is `50`.
   - For the third job `(6, 3, 100)`, binary search will find the first job ends right before this one starts. So, `dp[3]` is `max(dp[2], dp[1] + 100)` which is `100`.
   - For the fourth job `(7, 4, 70)`, binary search again finds the second job ends right before this one starts. So, `dp[4]` is `max(dp[3], dp[2] + 70)` which is `120`.
4. **Profit Calculation**: After processing all the jobs, our `dp` table looks like this:

   ```
   dp = [0, 50, 50, 100, 120]
   ```

   This table reads as follows: The maximum profit up to (and including) each job index is recorded in `dp`. For instance, by the fourth job, the maximum profit we can have obtained is 120.

5. **Result**: The result of the contiguous job scheduling for this example is `dp[4]` which equals `120`. This is the maximum profit we can earn without overlapping jobs. Therefore:

   ```
   return dp[4]  # Outputs: 120
   ```

By applying these steps to the example, we efficiently calculate the maximum profit of 120 by scheduling the first and third jobs which do not overlap in time.

Python Solution

```python
from bisect import bisect_right
from typing import List

class Solution:
    def jobScheduling(self, start_times: List[int], end_times: List[int], profits: List[int]) -> int:
        # Combine the job information into a single list and sort by end time.
        jobs = sorted(zip(end_times, start_times, profits))

        # Get the total number of jobs.
        number_of_jobs = len(profits)

        # Initialize dynamic programming table with 0 profits.
        dp = [0] * (number_of_jobs + 1)

        # Iterate over the jobs.
        for i, (current_end_time, current_start_time, current_profit) in enumerate(jobs):
            # Find the rightmost job that doesn't conflict with the current job's start time.
            # Use binary search for efficient querying. 'hi' is set to the current index 'i' for optimization.
            index = bisect_right(jobs, current_start_time, hi=i, key=lambda x: x[0])

            # Update the DP table by choosing the maximum of either taking the current job or not.
            # If taking the current job, add its profit to the total profit of non-conflicting jobs.
            dp[i + 1] = max(dp[i], dp[index] + current_profit)

        # Return the maximum profit which is the last element in the DP table.
        return dp[number_of_jobs]

# Example usage:
# sol = Solution()
# print(sol.jobScheduling([1,2,3,4], [3,4,5,6], [50,10,40,70]))

```

Java Solution

```java
class Solution {
    public int jobScheduling(int[] startTime, int[] endTime, int[] profit) {
        int numJobs = profit.length; // Number of jobs to consider for scheduling
        int[][] jobs = new int[numJobs][3]; // 2D array to hold the job information

        // Populate the jobs array with the start time, end time, and profit for each job
        for (int i = 0; i < numJobs; ++i) {
            jobs[i] = new int[]{startTime[i], endTime[i], profit[i]};
        }

        // Sort the jobs by their end times to facilitate an efficient scheduling strategy
        Arrays.sort(jobs, Comparator.comparingInt(a -> a[1]));

        int[] dp = new int[numJobs + 1]; // Dynamic programming table to hold maximum profits

        // Iterate through each job to compute the maximum profit
        for (int i = 0; i < numJobs; ++i) {
            // Search for the latest job index that does not conflict with the current job's start time
            int lastNonConflictingIndex = findLastNonConflictingJob(jobs, jobs[i][0], i);
            // Update the dp table entry using the maximum of including or excluding the current job
            dp[i + 1] = Math.max(dp[i], dp[lastNonConflictingIndex] + jobs[i][2]);
        }

        // The last entry in the dp table contains the maximum profit obtainable
        return dp[numJobs];
    }

    private int findLastNonConflictingJob(int[][] jobs, int startTime, int upperBound) {
        int left = 0, right = upperBound;
        // Perform binary search to find the rightmost job whose end time is not later than the start time
        while (left < right) {
            int mid = (left + right) >>> 1; // Use unsigned right shift to avoid overflow

            if (jobs[mid][1] > startTime) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        // Return the index of the last job that does not conflict, which is one less than 'left'
        return left == 0 ? 0 : left - 1;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to maximize profit by scheduling jobs such that no two jobs overlap
    int jobScheduling(vector<int>& startTimes, vector<int>& endTimes, vector<int>& profits) {
        int numJobs = profits.size(); // Number of jobs

        // Vector of tuples to store each job's information (end time, start time, profit)
        vector<tuple<int, int, int>> jobs(numJobs);

        // Populate the vector with job information
        for (int i = 0; i < numJobs; ++i) {
            jobs[i] = {endTimes[i], startTimes[i], profits[i]};
        }

        // Sort the jobs based on their end times; this will also sort start times and profits correspondingly
        sort(jobs.begin(), jobs.end());

        // Create a DP vector to store the maximum profit up to the i-th job
        vector<int> dp(numJobs + 1);

        // Iterate over the jobs
        for (int i = 0; i < numJobs; ++i) {
            // Destructure tuple to get current job's start time and profit
            auto [endTime, startTime, profit] = jobs[i];

            // Find the latest job that does not conflict with the current job i.e., job that ends before the start of the current jo
            int latestNonConflictJobIndex = upper_bound(jobs.begin(), jobs.begin() + i, startTime, [&](int time, const auto& job) ->
                return time < get<0>(job);
            }) - jobs.begin();

            // Update the DP table by choosing the maximum profit between
            // - profit from the previous job (excluding the current job)
            // - profit from the current job plus the latest non-conflicting job
            dp[i + 1] = max(dp[i], dp[latestNonConflictJobIndex] + profit);
        }

        // The last element in the DP array will contain the maximum profit that can be achieved.
        return dp[numJobs];
    }
};

```

Typescript Solution

```typescript
function jobScheduling(startTime: number[], endTime: number[], profit: number[]): number {
    const jobCount = startTime.length; // Number of jobs
    const dp = new Array(jobCount).fill(0); // Dynamic programming table to store max profit until job i
    const jobIndices = new Array(jobCount).fill(0).map((_, index) => index); // Create index array for jobs

    // Sort the index array based on start times of the jobs
    jobIndices.sort((a, b) => startTime[a] - startTime[b]);

    // Binary search to find the next job which starts after the current job ends
    const findNextJobIndex = (endTimeToMatch: number): number => {
        let left = 0;
        let right = jobCount;
        while (left < right) {
            const mid = (left + right) >> 1;
            if (startTime[jobIndices[mid]] >= endTimeToMatch) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        return left;
    };

    // Recursive function to calculate the maximum profit using memoization
    const calculateMaxProfit = (currentJob: number): number => {
        if (currentJob >= jobCount) {
            return 0; // Base case: no more jobs to process
        }

        if (dp[currentJob] !== 0) {
            return dp[currentJob]; // If already calculated, return the stored value
        }

        // Find the next index where a job can be started after the current job ends
        const nextJobIndex = findNextJobIndex(endTime[jobIndices[currentJob]]);

        // Recursive step, memoize and return the maximum of choosing or skipping the current job
        return dp[currentJob] = Math.max(
            calculateMaxProfit(currentJob + 1), // Skip the current job
            calculateMaxProfit(nextJobIndex) + profit[jobIndices[currentJob]] // Choose the current job
        );
    };

    // Calculate and return the maximum profit starting from the first job
    return calculateMaxProfit(0);
}

```

Time and Space Complexity

The given Python code defines a method for scheduling jobs to maximize profit, where each job has a start time, end time, and profit associated with it. The code uses dynamic programming to solve the problem and binary search to find the latest job end time that does not conflict with the current job's start time.

Time Complexity:

- Sorting the jobs by their end times at the beginning has a time complexity of `O(N log N)`, where `N` is the number of jobs.
- Iterating through the sorted jobs and using `bisect_right` to perform binary search each have their own complexities:
  - The iteration itself is `O(N)`.
  - The binary search inside the loop via `bisect_right` has a complexity of `O(log i)` for each iteration, where `i` is the current index in the loop (in the worst-case scenario, this is `O(log N)`).

The binary search is called `N` times within the loop, resulting in `O(N log N)` for all binary search calls. Combining the sorting and binary search complexities, the overall time complexity is `O(N log N)`.

Space Complexity:

- The `dp` list has a space complexity of `O(N)`.
- The `jobs` list has a space complexity of `O(N)`, for it stores the zipped version of `endTime`, `startTime`, and `profit`, all of which are lists of length `N`.

The additional space for the variables `i`, `s`, `p`, `j` and the space for function calls is negligible. Thus, the total space complexity of the algorithm is `O(N)`.

Combining time and space complexities, the code has a total time complexity of `O(N log N)` and a total space complexity of `O(N)`.