Job Sequencing Problem

Job Sequencing Problem

Difficulty: **Medium** Accuracy: **34.51%** Submissions: **294K+** Points: **4**

You are given two arrays: **deadline[],** and **profit[],** which represent a set of jobs, where each job is associated with a **deadline**, and a **profit**. Each job takes 1 unit of time to complete, and only one job can be scheduled at a time. You will earn the profit associated with a job only if it is completed by its deadline.

Your task is to find:

1. The **maximum number of jobs** that can be completed within their deadlines.
2. The **total maximum profit** earned by completing those jobs.

Examples :

Input: deadline[] = [4, 1, 1, 1], profit[] = [20, 10, 40, 30]
Output: [2, 60]
Explanation: Job_1 and Job_3 can be done with maximum profit of 60 (20+40).

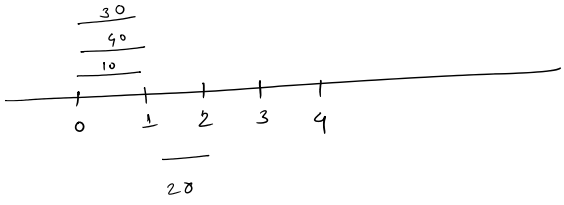
Input: deadline[] = [2, 1, 2, 1, 1], profit[] = [100, 19, 27, 25, 15]
Output: [2, 127]
Explanation: Job_1 and Job_3 can be done with maximum profit of 127 (100+27).

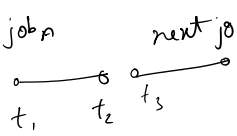
Input: deadline[] = [3, 1, 2, 2], profit[] = [50, 10, 20, 30]
Output: [3, 100]
Explanation: Job_1, Job_3 and Job_4 can be completed with a maximum profit of 100 (50 + 20 + 30).

Constraints:
$1 \leq deadline.size() == profit.size() \leq 10^5$
$1 \leq deadline[i] \leq deadline.size()$
$1 \leq profit[i] \leq 500$

eg: deadline[] = [4, 1, 1, 1]
profit[] = [20, 10, 40, 30]

O/p: [2, 60]





- sort by deadline, then profit

[Naive Approach] Greedy Approach & Sorting - O(n^2) Time & O(n) Space :-

Step by step implementation:-

- Store jobs as pairs of (Profit, Deadline): Since we need to prioritize jobs with higher profits, we pair the profit & deadline together.
- Store jobs based on Profit: We sort the jobs array in descending order of profit so that we prioritize scheduling the most profitable jobs first.
- Create a Slot array: We create a slot[] array of size n (equal to the number of jobs) initialized with zeroes. This array will help track which time slots are occupied.
- Iterate over each job & try to schedule it:
  - For each job, check if it can be placed in an available time slot.
  - The job should be scheduled as late as possible but before it's deadline.

- The job should be [illegible] but before it's deadline.
- If an empty slot is found, schedule the job there, increment the job count, & add its profit to the total.
- After processing all jobs, return the no. of jobs completed & the total profit earned.

```
class Solution {
  public:
    vector<int> jobSequencing(vector<int> &deadline, vector<int> &profit) {
        // code here
        int n=deadline.size();
        int cnt=0;
        int totProfit=0;

        vector<pair<int, int>> jobs;
        for(int i=0;i<n;i++){
            jobs.push_back({profit[i], deadline[i]});
        }
        sort(jobs.begin(), jobs.end(), greater<pair<int, int>>());
        vector<int> slot(n, 0);
        for(int i=0;i<n;i++){
            int start=min(n, jobs[i].second)-1;
            for(int j=start;j>=0;j--){
                if(!slot[j]){
                    slot[j]=1;
                    cnt++;
                    totProfit+=jobs[i].first;
                    break;
                }
            }
        }
        return {cnt, totProfit};
    }
};
```

[Expected Approach] Greedy Approach, Sorting & Priority Queue -
O(n* log(n)) Time & O(n) space:-

The main idea is to sort the jobs based on their deadlines in ascending order. This ensures that jobs with earlier deadlines are processed first, preventing situations where a job with a shorter deadline remains unscheduled because a job with a later deadline was chosen instead. We use a min-heap to keep track of the selected jobs, allowing us to efficiently replace lower-profit jobs when a more profitable job becomes available.

Step by step Implementation:-

- Store jobs as pairs of (Deadline, Profit)
- Sort Jobs based on Deadline: We sort the jobs array in ascending order of deadline so that we prioritize jobs with earlier deadlines are considered first.
- For each job (deadline, profit) in the sorted list:
  - If the job can be scheduled within its deadline (i.e. the no. of jobs scheduled so far is less than the deadline), push

- If the job [illegible]
  no of jobs scheduled so far is less than the deadline), push its profit into the heap.
- If the heap is full (equal to deadline), replace the existing lowest profit job with the current job if it has a higher profit
  - This ensures that we always keep the most profitable jobs within the available slot.

o Traverse through the heap & store the total profit & the count of jobs.

```
class Solution {
  public:
    vector<int> jobSequencing(vector<int> &deadline, vector<int> &profit) {
        // code here
        int n=deadline.size();
        vector<int> ans={0,0};
        vector<pair<int, int>> jobs;
        for(int i=0;i<n;i++){
            jobs.push_back({deadline[i], profit[i]});
        }

        sort(jobs.begin(), jobs.end());

        priority_queue<int, vector<int>, greater<int>> pq;
        for(const auto &job:jobs){
            if(job.first>pq.size()){
                pq.push(job.second);
            }
            else if(!pq.empty() && pq.top()<job.second){
                pq.pop();
                pq.push(job.second);
            }
        }
        while(!pq.empty()){
            ans[1]+=pq.top();
            pq.pop();
            ans[0]++;
        }
        return ans;

    }
};
```

Job Sequencing Problem Using Disjoint Set:-

A greedy solution of time complexity O(n Log n) is already discussed. Below is the simple Greedy Algorithm.

1. Sort all jobs in decreasing order of profit.
2. Initialize the result sequence as first job in sorted jobs.
3. Do following for remaining n-1 jobs
   - If the current job can fit in the current result sequence without missing the deadline, add current job to the result. Else ignore the current job.

The costly operation in the Greedy solution is to assign a free slot for a job. We were traversing each and every slot for a job and assigning the greatest possible time slot(<deadline) which was available.

why to assign greatest time slot(free) to a job?
We assign the greatest possible time slot since if we assign a time slot even lesser than the available one then there might be some other job which will miss its deadline

Using Disjoint Set for Job Sequencing

All time slots are individual sets initially. We first find the maximum deadline of all jobs. Let the max deadline be m. We create m+1 individual sets. If a job is assigned a time slot of t where t >= 0, then the job is scheduled during [t-1, t]. So a set with value X represents the time slot [X-1, X].

We need to keep track of the greatest time slot available which can be allotted to a given job having deadline. We use the parent array of Disjoint Set Data structures for this purpose. The root of the tree is always the latest available slot. If for a deadline d, there is no slot available, then root would

Below are the detailed steps.

- The idea is to Disjoint Sets and create **individual** set for all available **time** slots.
- First find the **maximum** deadline of all the jobs, let's call it b. Now create a disjoint set with **d + 1** nodes, where **each** set is independent of other.
- **Sort** the jobs based on **profit** associated in **descending** order.
- Start with the first job, and for each job **find** the available **slot** which is **closest** to its **deadline**. Occupy the available slot and **merge** the **slot** with **slot-1,** by assigning slot-1 as **parent** of slot. If slot value is **0,** it means **no** slot is available, so move to the next job.
- At last find the sum of all the jobs with allocated slots.

How come find() of disjoint set returns the latest available time slot?

Initially, all time slots are individual slots. So the time slot returned is always maximum. When we assign a time slot 't' to a job, we do union of 't' with 't-1' in a way that 't-1' becomes the parent of 't'. To do this we call union(t-1, t). This means that all future queries for time slot t would now return the latest time slot available for set represented by t-1.

```
#include <bits/stdc++.h>
using namespace std;
class DisjointSet{
public:
    vector<int>parent;
    DisjointSet(int n){
        parent.resize(n+1);
        for(int i=0; i<=n;i++)parent[i]=i;
    }

int find(int s){
    if(s==parent[s])return s;
    return parent[s]=find(parent[s]); }

void merge(int u, int v){
        //update the greatest available
        //free slot to u
        parent[v]=u; }};

bool comp(pair<int,int>a, pair<int,int>b){
        return a.first>b.first;}

vector<int> jobSequencing(vector<int>& id, vector<int>&
    vector<int>& profit){
```

```
vector<int> jobSequencing(vector<int>&
 deadline, vector<int>& profit){
        int n = id.size();
        vector<int> ans = {0, 0};
        vector<pair<int, int>> jobs;
        for(int i=0; i<n; i++){
            jobs.push_back({profit[i], deadline[i]});}

    sort(jobs.begin(), jobs.end(), comp());

    int d = INT_MIN;
    for(int i=0; i<n; i++){
        d = max(d, deadline[i]); }

    // create a disjoint set of d nodes
    DisjointSet ds(d);
    for(int i=0; i<n; i++){
        int slots = ds.find(jobs[i].second);
        if(slots >0){
            ds.merge(ds.find(slots-1), slots);
            ans[1] += jobs[i].first;
            ans[0]++;
        }}
    return ans;
    }
```