Lesson:

Greedy-2

PW SKILLS

Prerequisite:

- Arrays
- Vector
- Greedy Algorithm

Today's Checklist:

- Problems based on Greedy Algorithm

Problem 1: Minimum Cost to cut a board into squares.

A board of length M and width N is given. The task is to break this board into M * N squares such that cost of breaking is minimum. The cutting cost for each edge will be given for the board in two arrays X[] and Y[] representing the cost of cutting horizontal and vertical edges respectively, i.e X[i] represents the cost of cutting a horizontal edge of size i. In short, you need to choose such a sequence of cutting such that cost is minimized. Return the minimized cost.

Input:
M = 6. N = 4
X[] = {2, 1, 3, 1, 4}
Y[] = {4, 1, 2}

The values of M and N are positive integers.
Constraints: 1 <= M, N <= 10^5
The arrays X[] and Y[] represent the costs of cutting horizontal and vertical edges, respectively.
The length of X[] is M - 1, and the length of Y[] is N - 1.
Constraints: 1 <= X[i], Y[i] <= 10^5

Output: 42
Explanation: For above board optimal way to cut into square is:
Total minimum cost in above case is 42. It is
evaluated using following steps.

Initial Value : Total_cost = 0
Total_cost = Total_cost + edge_cost * total_pieces

Cost 4 Horizontal cut Cost = 0 + 4*1 = 4
Cost 4 Vertical cut Cost = 4 + 4*2 = 12
Cost 3 Vertical cut Cost = 12 + 3*2 = 18
Cost 2 Horizontal cut Cost = 18 + 2*3 = 24
Cost 2 Vertical cut Cost = 24 + 2*3 = 30
Cost 1 Horizontal cut Cost = 30 + 1*4 = 34
Cost 1 Vertical cut Cost = 34 + 1*4 = 38
Cost 1 Vertical cut Cost = 38 + 1*4 = 42

Solution: https://pastebin.com/QM8vwpcw

Approach: The approach follows a greedy strategy to minimize the cost of breaking the board into squares. The idea is to always choose the highest cost edge for cutting at each step.

1. The input vectors X and Y represent the costs of cutting each horizontal and vertical edge, respectively.
2. The vectors are sorted in ascending order to ensure that the highest cost edges are considered first during the cutting process.
3. The algorithm iterates through the sorted vectors, selecting the higher cost between the current horizontal and vertical edges. It keeps track of the number of cuts made in both the horizontal (horizontalCuts) and vertical (verticalCuts) directions.
4. The cost of each cut is added to the total cost. For horizontal edges, the cost is multiplied by the number of vertical cuts made so far (verticalCuts), and for vertical edges, the cost is multiplied by the number of horizontal cuts made so far (horizontalCuts).
5. Finally, after iterating through all the edges, the algorithm handles the remaining edges (if any) by adding their costs to the total based on the number of cuts made in the corresponding direction.
6. The total cost is returned as the minimum cost of breaking the board into squares.

By selecting the highest cost edges at each step and considering the number of cuts made in each direction, the approach aims to minimize the overall cost of breaking the board.

The **time complexity** of the solution is O(MlogM + NlogN), where M and N are the dimensions of the board, due to the sorting operation performed on the edge costs.

The **space complexity** is O(1) as the algorithm uses a constant amount of additional space, regardless of the input size.

Output:

```
42

Process finished with exit code 0
```

Problem 2: Leetcode 435. Non-overlapping Intervals

Given an array of intervals where intervals[i] = [starti, endi], return the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

Input: intervals = [[1,2],[2,3],[3,4],[1,3]]
Output: 1
Explanation: [1,3] can be removed and the rest of the intervals are non-overlapping.

Solution: https://pastebin.com/1QECjXp5

Approach:

Sort the intervals based on their end times in non-decreasing order.

1. Initialize a variable count to keep track of the number of intervals to be removed.
2. Initialize a variable prevEnd to store the end time of the first interval.
3. Iterate through the sorted intervals from the second interval onwards:
   - If the start time of the current interval is less than prevEnd, it overlaps with the previous interval.
     - Increment count to indicate that an interval needs to be removed.
   - Otherwise, update prevEnd to the end time of the current interval.
4. Return the value of count.

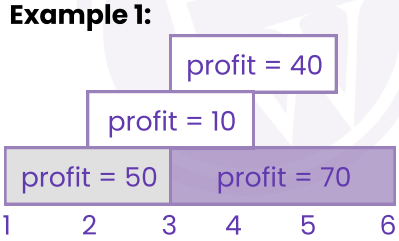
The idea behind this approach is to greedily select the interval with the earliest end time. By doing this, we ensure that the remaining intervals have the maximum possible non-overlapping range.

Time Complexity: O(n log n), where n is the number of intervals. This is because the code sorts the intervals array using Arrays.sort with a comparator, which has a time complexity of O(n log n).
Space Complexity: O(1) since it uses a constant amount of extra space. The sorting operation is performed in-place on the intervals array, and the additional variables used for counting and tracking the previous end time occupy a constant amount of space.

Problem 3:1235. Maximum Profit in Job Scheduling

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.

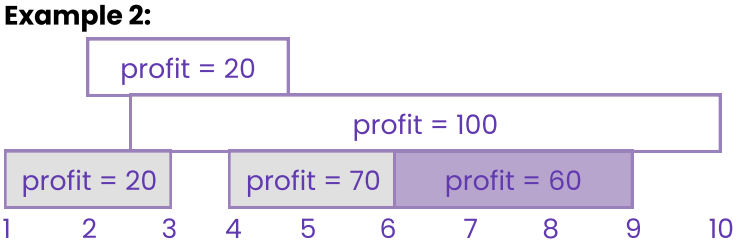
Example 1:



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.

Solution: https://pastebin.com/DGZBSiNc

Explanation:

- This code sorts the jobs based on their start times, processes them in chronological order, and uses a priority queue to efficiently keep track of the maximum profit achievable at each point in time.
- Inside the jobScheduling function, a 2D vector called jobs is created. Each element of jobs represents a job and contains the start time, end time, and profit of that job. The jobs vector is created by iterating over the input vectors and pushing a new vector containing the corresponding values for each job.
- The jobs vector is sorted using sort in ascending order based on the start time of each job. This step is important because it allows us to process the jobs in chronological order.
- A priority queue called pq is created. The priority queue is defined to store vectors of integers, where each vector contains the end time and total profit of a job. The priority queue is sorted based on the end time of the jobs, with the smallest end time at the top. The greater comparator is used to achieve this ordering.
- An integer variable mp is initialized to 0. This variable will store the maximum profit achieved.
- The code then iterates over each job in the sorted jobs vector. For each job, it extracts the start time, end time, and profit.
- The code enters a while loop that continues as long as the priority queue is not empty and the start time of the current job is greater than or equal to the end time of the job at the top of the priority queue. This loop is used to remove any jobs from the priority queue that have end times earlier than the start time of the current job. While removing these jobs, it updates the maximum profit variable mp if a higher profit is found.
- After the while loop, the code pushes the current job into the priority queue. The end time of the current job is used as the priority (key) for sorting, and the total profit of the job is added to the maximum profit obtained from previous jobs (profit + mp).
- Once all jobs have been processed, the code enters another while loop that continues as long as the priority queue is not empty. This loop is used to handle any remaining jobs in the priority queue in case there are no more jobs to process. It updates mp if a higher profit is found.
- Finally, the function returns the maximum profit obtained, which is stored in the variable mp.

Problem 4: Smallest Number

The task is to find the smallest number with given sum of digits as S and number of digits as D.

Example 1:
Input:
S = 9
D = 2
Output:
18

possible with sum = 9

Explanation:
18 is the smallest number
possible with sum = 9
and total digits = 2

Solution: https://pastebin.com/diQcAcW8

Explanation:

1. First, the function checks if the sum of digits (S) is greater than 9 times the number of digits (D). If it is, then it is not possible to form a valid number, so it returns "-1".
2. It initializes an empty string to store the resulting number.
3. While the remaining sum (S) is non-negative and the number of digits (D) is positive, it iterates from 0 to 9 to find the smallest possible digit that can be added to the number.
4. It checks if adding the current digit to the number will not exceed the remaining sum of digits (S) if the remaining digits (D-1) were all 9s. If it satisfies the condition, it adds the digit to the number, decrements the number of digits (D), and subtracts the digit from the remaining sum (S).
5. Finally, it returns the resulting number.

Time complexity: The time complexity of the solution is O(D*9), where D is the number of digits. In the worst case, the inner loop iterates 9 times for each digit.

Space complexity: The space complexity is O(D), as we store the resulting number as a string of length D.

Output:

```
Smallest Number: 18


Process finished with exit code 0
```

Upcoming Class Teaser:

- Dynamic Programming