1. **(20 points: each part has 10 points)**

a. Consider an instance of the weighted interval scheduling problem (find maximum weighted non-overlapping intervals). Design an **O(nlogn)** time algorithm to solve the problem. Prove its correctness and analyze its time complexity.

Solution

Algorithm

- Sort the intervals based on finishing time.
- Put all the finishing times in a list: events
- Using dynamic programming
    - Opt[j]: max weights we can collect in the first j time units
- Base case: Opt[first finishing time] = weight of the first interval
- Opt[j] = max of these two cases
- Intuition: we either have to pick an interval ending in j or don't
    - max Opt[i] + Weight($e_{i,j}$)) for all intervals (i, j) ending in j
    - $\mathrm{Opt}[\mathrm{i}-1]$

Time complexity

- Sorting: nlog(n)
- Gathering finishing times in a list: O(n)
- Dynamic Programming: each interval gets checked only once
    - Number of calls: n: for checking intervals + n: Opt[i-1]
    - O(n)
- Note that if we used time instances (check all time points and not gather all finishing times) the time complexity would have been different

Proof: induction

- Base case: k=1: if we have only one interval, its weight is the max collectible weights until its end.
- Assume dynamic programming correctly gives us the answer for all intervals 1..k
    - We prove it works for the next finishing time k+1
- For k^{th} finishin time, there are two cases:
    - No interval that actually ends in that time is selected: Opt[k-1]
    - One of the intervals with this property is selected: checking all the edges

Hence, it correctly builds the solution for any arbitrary k.

b. Design an **O(n)** time algorithm that merges two sorted lists of size **n**.

Solution

Algorithm

- Call the lists l1 and l2
- Compare the first element of both arrays
    - Put the smaller one in another array (of size 2n)
- Repeat the last step until there is no elements left.

Proof

- To prove the resulting array is sorted, we can prove each element is smaller than all the numbers that are inserted after it.
    - The first elements are smaller than other numbers in their own list
    - We pick the smaller of two: the smallest is picked every time.

Time complexity

- We do one comparison per picked element
    - 2n comparisons: O(n)

2. **(15 points)** A Hamiltonian path in a DAG is a path that contains each vertex exactly once. Is the problem of finding a Hamiltonian path in a DAG NP-Complete? Either prove it or design an **O(e+n)** time algorithm for finding a Hamiltonian path in a DAG.

Solution

Algorithm

- Run topological ordering algorithm
    - Take the node with in_degree=0
    - Put it in the ordering
    - Remove its edges
    - Pick the next node with in_degree=0
        - There must be exactly one node.
            - Else return no path can be found

Proof

Note that

- If multiple nodes have out_degree=0 there is no Hamiltonian path
    - At most one can be the end of path
    - We get stuck in one of them
- If multiple nodes have in_degree=0: no Hamiltonian path
    - At most one can be the start of path
    - There is no path to reach them
- **There must be exactly one node with in_degree=0 and one with out_degree=0**
    - We already know in a DAG there must be at least one node with # incoming edges=0
        - Or there would have been been a cycle
            - Go to the parents n+1 times: guaranteed to find a cycle
    - Same proof can be used to prove there is at least a node with out_degree=0

If there is only one node with in_degree=0 at each step, it indicates that:

- The node with in_degree=0 in the next step, is a child of current node

So, the output of the algorithm is an ordering in which there is an edge between every node and the one after it: it is a path going through every node.

We proved the output of our algorithm is a Hamiltonian path. Now we must prove if there is a Hamiltonian path, we will find it:

also proved because the first node in any Hamiltonian path has to be the one with in-degree=0, second one has to be the one with in_degree=0 in the remaining graph, …

- So it is unique

Time Complexity

Each edge is removed once: O(m)

Calculating in_degrees O(n)

- keeping them updated: O(n+m)

Overall: O(n+m)

3. **(15 points)** Given an **nxn** matrix where every row is sorted in increasing order, design an algorithm that outputs the smallest common element in all rows. If there is no common element return -1. Analyze the time complexity of your algorithm.

Input: mat = [[1,2,3,4,5,8], [2,4,5,8,10], [3,5,7,8,9,11], [1,3,5,7,8,9]]
Output: 5

Solution

Algorithm

- Put all elements in hashmaps: One hashmap per row
- Loop over the first row
    - For each number, check all the other rows if the number exists in them

Proof

We need to prove the picked element is
1. the smallest one
2. exists in all rows

1: Smallest common element must also exist in the first row. We check elements in increasing order.
2: obvious (Hashmap works)

Time Complexity

- $O(n^2)$ for putting elements in hashmaps
- n-1 checks per element in first row: n(n-1)

If the space was limited (cannot use hashmaps): Do binary search for each element of first row: $O(n^2 \log(n))$

4. **(15 points)** In a technical interview, you've have been given an array of **n** numbers and you need to find a pair of numbers which are equal to given target value **L**. Numbers can be either positive, negative or both. An $O(n^2)$ time algorithm is trivial so you need to do better than that. Analyze the time complexity of your algorithm.

Example:

sequence = [8, 10, 2, 9, 7, 5]
Target L= 11 Answer (9,2)

Solution

Midterm question.

First approach
Put all the numbers in a hashmap by one iteration of the array. iterate one more time. For each number, check if (L – number) is in the hashmap. Return all the answers.

O(n)

Second approach
Sort the array. For each element a_i, do a binary search for $(L-a_i)$.
O(nlog(n)) sorting + n times binary search: O(nlog(n))

Third approach
Sort the array. Use two pointers: one pointing to the beginning (b) and one to the end(e).

- If sum was smaller than L: b += 1
- If sum was greater than L: e -= 1

O(nlog(n)) sorting

5. **(15 points)** Consider a given set of airline travel segments in a day. For example (Los Angeles, San Francisco)(8:00, 9:15) means there is a flight that leaves Los Angeles at 8 am and arrives in San Francisco at 9:15 am. We also know the number of passengers that can travel on each flight. Design an efficient algorithm that finds the maximum number of passengers that can travel from Los Angeles to New York in a day. What is the time complexity of your algorithm.

Solution

Algorithm

We form the network like this:

- LA is the source vertex.
- For each flight X(time1)->Y(time2)
    - Put a node for X(time1) and Y(time2) and draw an edge with weight=flight capacity
- Draw an edge from X(time1) to X(time2) with infinite capacity if time2 > time1
    - Example: SF(10:30) to SF(13:30) with capacity ∞
- Draw an edge with infinite weight from all New York nodes to destination.

Proof

To prove the original problem and the maximum flow problem over the formed graph G are the same, we should prove:

1. Any legal flow in G can be an arrangement of flights
2. Any arrangement of flight seats can be shown in this network flow form

In this problem, these two statements are obvious.

Proof for 1: X->Y with flow f_{xy}: flow shows the number of passengers that flew from LA in their first flight. Fill the rest of the seats arbitrarily. Flow is < edge capacity so this is valid.

Proof for 2: for each flight, weight of the edge is flight capacity, flow will be the number of passengers that originally departed from LA. To show this is legal, we should check the two conditions:

- Conservation: trivial
- Capacity: trivial

Time complexity

Max Flow algorithm: O(|f|e)

Forming the graph: $O(n^2)$ where n is number of flights

- If we have n/2 flights from SF to NY and n/2 from LA to SF, there will be $O(n^2/4)$ edges

6. **(20 points)** Prove that finding a vertex cover (VC) in a graph is NP-complete. You can assume that finding a maximum independent set (MIS) in a graph is known to be NP-complete.VC is the minimum number of vertices that contains at least one end point of every edge. MIS is the maximum number of vertices that are pairwise dis-connected (there is no edge between them).

Solution

To prove VC is NP-complete, we can prove that the MIS is reducible to VC. Then we can use proof by contradiction:

- If VC was not NP-complete, it could have been used for solving MIS, and hence, MIS is not NP-complete.

Assume we have a vertex cover of size at most k in a graph G=(V, E): call it S

- If there is an edge (u, v) in G:
    - at least one of u and v are in S
- So both of them cannot be in V-S
- Hence, V-S is an independent set

Having an instance of vertex cover, we can solve maximum independent set by calling it once, and taking all the nodes that it does not select.