HW3

▼ **Question 1**

1 You have N ropes each with length $L_1, L_2, \ldots, L_N$, and we want to connect the ropes into one rope. Each time, we can connect 2 ropes, and the cost is the sum of the lengths of the 2 ropes. Develop an algorithm such that we minimize the cost of connecting all the ropes.

Solution

```
// create a min-heap (using a priority queue), lets say pq
// add all N ropes to pq. The elements will get arranged in sorted order, as per the property of the Min-Heap
// keep a global_sum counter to store the final answer
while (pq.size() > 1) {
  int a = pq.remove();
  int b = pq.remove();
  int sum = a + b;
  global_sum += sum;
  pq.add(sum);
}

return global_sum;
```

💡 Choosing the Min-Heap (implemented using a PriorityQueue) selects the 2 smallest length ropes for concatenation, which is a greedy approach. The above algorithm returns the minimum cost of connecting N ropes into a single rope.

▼ **Question 2**

2 You have a bottle that can hold L liters of liquid. There are N different types of liquid with amount $L_1, L_2, \ldots, L_N$ and with value $V_1, V_2, \ldots, V_N$. Assume that mixing liquids doesn't change their values. Find an algorithm to store the most value of liquid in your bottle.

Solution

It has been assumed that liquids can be taken as fractional, which means that this problem is the same as fractional knapsack where we have to maximize the value, keeping in mind the capacity of the bottle, i.e. L liters.

Consider the following algorithm -

```
// create an class Item which has 2 data members - capacity, value
class Item {
    public int val;
    public int wt;
```

```
    Item(int val, int wt) {
        this.val = val;
        this.wt = wt;
    }
}

// each liquid-value map can be encapsulated in the above Item class

// in order to get the most liquid in the bottle, the greedy approach that we choose is to maximize the selection of those 'Items'
// initialize 2 variables - total_weight and total_value to keep track of the final weights and values once the capacity (maximum
// let C be the total capacity of the bottle
for each item in the list of items {
  let val be the value of this 'item'
  let weight be the weight of this 'item'

  if (weight + total_weight <= C) {
    total_value += val;
    total_weight += weight;
  } else {
    let remaining_weight = C - (weight + total_weight)
    total_value += remaining_weight * (val / weight)
    break;
}

return: total_value;
```

💡 Greedy Approach: Fetching max of (value/weight) from the available map of bottle amount → value
The above algorithm returns the maximum_value of the liquid that is possible

---

▼ **Question 3**

3 Suppose you were to drive from USC to Santa Monica along I-10. Your gas tank, when full, holds enough gas to go p miles, and you have a map that contains the information on the distances between gas stations along the route. Let $d_1 < d_2 \ldots < d_n$ be the locations of all the gas stations along the route where d_i is the distance from USC to the gas station. We assume that the distance between neighboring gas stations is at most p miles. Your goal is to make as few gas stops as possible along the way. Give the most efficient algorithm to determine at which gas stations you should stop and prove that your strategy yields an optimal solution. Give the time complexity of your algorithm as a function of n.

Solution

Greedy Approach - In an attempt to make as few stops as possible, we aim for covering the maximum distance that is possible without running out of gas (i.e. within the possible p miles) from the start point. This means we cover the maximum distance first before stopping to refill. Lets say you are at i^{th} stop, ideally, you should have enough gas to reach the $(i+1)^{th}$ gas station. If there is not enough gas, stop at the i^{th} gas station for refueling. Note that this i^{th} gas station is within the possible p miles that a full gas of tank can cover.

Proof of correctness - *Greedy algorithm stays ahead*

Let's say our algorithm generates a list of gas stations that we need to stop on - $(g_1, g_2, g_3, ..., g_n)$

Assume that an optimal solution exists which outputs the following list of gas stations where we should stop - $(h_1, h_2, h_3, .., h_n)$

Since our choice of gas station g_1 was within p miles limit, and was the maximum distance that could be covered with full tank limitations and availability of the gas stations, no other algorithm could have gone to $(g_1+1)^{th}$ gas station as the vehicle would have run out of gas. This implies that the output of the optimal solution should be $h_1 < g_1$. The same

argument can be repeated recursively for other gas stations as well. Since our solution outputs a solution that follows the assumed optimal solution, our solution is also optimal.

Time Complexity - $O(n)$

▼ **Question 4**

4 (a) Consider the problem of making change for n cents using the fewest number of coins. Describe a greedy algorithm to make change consisting of quarters(25 cents), dimes(10 cents), nickels(5 cents) and pennies(1 cents). Prove that your algorithm yields an optimal solution. (Hints: consider how many pennies, nickels, dimes and dime plus nickels are taken by an optimal solution at most.)

(b) For the previous problem, give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Assume that each coin's value is an integer. Your set should include a penny so that there is a solution for every value of n.

Solution

a. Assuming that we have infinite supply of quarters, dimes, nickels and pennies, the greedy approach would be to use up as many quarters as we can in order to make the change for n cents. This approach would provide the fewest number of coins possible.

Consider the following algorithm -

```
// let there be an list, list = {25, 10, 5, 1} i.e. the quarters, dimes, nickels and pennies - arranged in this same order.
// let n be the number of cents we want to provide coins for.
// let 'ans' be a global variable to store the fewest number of coins for a given amount of cents
while (n > 0 && list.size() > 0) {
      let max = peek and remove first element of the list
      ans += n / max;
      n %= max;
  }

  if (n > 0) {
      return -1;
  }
  return ans;
}
```

b. Set of coins denominations for which the greedy algorithm does not apply is the following -

```
Coin denominations available = [16, 10, 8, 1]
Cents to make = 18
```

The optimal solution would have been 2 coins (10 + 8) , but the above algorithm provides the answer as 3 coins (16 + 1 + 1)

▼ **Question 5 - Solve Kleinberg and Tardos, Chapter 3, Exercise 3.**

3. The algorithm described in Section 3.6 for computing a topological ordering of a DAG repeatedly finds a node with no incoming edges and deletes it. This will eventually produce a topological ordering, provided that the input graph really is a DAG.

   But suppose that we're given an arbitrary graph that may or may not be a DAG. Extend the topological ordering algorithm so that, given an input directed graph G, it outputs one of two things: (a) a topological ordering, thus establishing that G is a DAG; or (b) a cycle in G, thus establishing that G is not a DAG. The running time of your algorithm should be $O(m + n)$ for a directed graph with n nodes and m edges.

Solution

Logic to identify whether a cycle exists in a given graph, G = We will traverse each node of the graph, and if each node has at least 1 incoming edge, we know that a graph G contains a cycle C. If there is no cycle present and there is a node that has no incoming edge, we know that we have a DAG and we can find and print its topological order.

```
To compute whether a graph has a cycle and if not, output the topological ordering of G:
if there exists a node v with no incoming edges {
  Delete v from G
  Recursively compute a topological ordering of G−{v}
  and append this order after v
}
else {
  // if there exists an incoming edge for each node, it means that a cycle is present
  we follow the edge going into the node.
  store the node v, traversed in a set, S
  we keep on doing this for every node until we arrive at a node that as already been visited
  if a node has already been visited, we terminate the algorithm, and all the nodes in the set S is the cycle (in reverse order)
}
```

💡 Since we traverse each node and each edge once, the time complexity of this algorithm is $O(m + n)$ where $m = number of vertices$ and $n = number of edges$

▼ **Question 6 - Solve Kleinberg and Tardos, Chapter 4, Exercise 4.**

4. Some of your friends have gotten into the burgeoning field of *time-series data mining*, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges—what's being bought—are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain "patterns" in them—for example, they may want to know if the four events

   `buy Yahoo, buy eBay, buy Yahoo, buy Oracle`

   occur in this sequence S, in order but not necessarily consecutively.

   They begin with a collection of possible *events* (e.g., the possible transactions) and a sequence S of n of these events. A given event may occur multiple times in S (e.g., `Yahoo` stock may be bought many times in a single sequence S). We will say that a sequence S' is a *subsequence* of S if there is a way to delete certain of the events from S so that the remaining events, in order, are equal to the sequence S'. So, for example, the sequence of four events above is a subsequence of the sequence

   `buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle`

   Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of S. So this is the problem they pose to you: Give an algorithm that takes two sequences of events—S' of length m and S of length n, each possibly containing an event more than once—and decides in time $O(m+n)$ whether S' is a subsequence of S.

Solution

Here, given 2 sets, S and S', we need find wither S' is a subsequence of S

Algorithm -

```
// size of S = n and S' = m
// consider that S = {s1, s2, s3, .... , si}
// consider that S' = {s'1, s'2, s'3, .... , s'j}
// let {k1, k2, .... , k} be the set of subsequence of S' that is found in the given S
Initially, let i = j = 1
while (i <= n && j <= m)
  if (sk of S and s'k of S' are equal)
    let kj = i;
    let i = i + 1 and j = j + 1 // since a match is found, both pointers are incremented
  otherwise i = i + 1 // the pointer to the longer list moves forward if no match is found
  endif
end while
when j = m + 1, return the subsequence found.
else return that no subsequence of S' for the given S is possible
```

💡 Since each sequence is being traversed only once, and the pointers are being incremented by i, the worst case complexity of this algorithm is $O(n)$, where n is the length of the superset from which a subsequence needs to be found.

Each operation takes $O(1)$ time and in the worst case, there can be n operations, hence the Big-Oh is $O(n)$

▼ **Question 7**

7 (Not Graded) There are N tasks that need to be completed by 2 computers A and B. Each task i has 2 parts that takes time a_i(first part) and b_i (second part) to be completed. The first part must be completed before starting the second part. Computer A does the first part of all the tasks while computer B does the second part of all the tasks. Computer A can only do one task at a time, while computer B can do any amount of tasks at the same time. Find an O(nlogn) algorithm that minimizes the time to complete all the tasks, and give a proof of why the solution is optimal.

Solution

This solution is the same as minimizing the lateness problem for Interval Scheduling.

Arrange part (b) of each task in decreasing order since these will be completed by Computer B parallel.

To be done later for personal practise. Submitted as it is since it is an ungraded problem.