Homework 9

CS/ECE 374 B

Due 8 p.m. Tuesday, December 3

- Remember that if you use a greedy algorithm, you must prove that it will always arrive at an optimal solution
- Not all the questions in this set can be solved with a greedy algorithm.
- Make sure that you analyze your algorithms complexity and that your algorithms are efficient; solutions slower than the reference solution will lose points.

Question 1: Fueling Up. . . . .

(a) You are driving along a highway with refueling stations. Each station is at distance $D[i]$ from your starting point. Your car holds enough gas to travel up to 100 miles before refueling. Assume that you start out with an empty tank, but there is a refueling station at your starting point (i.e., $D[0] = 0$). Assume, likewise, that there is a fueling station at your destination, $D[n]$. Design and analyze an efficient algorithm that computes the minimum the number of refueling stops you have to make to reach your destination, or return ∞ if this is impossible.

Solution: You can just simply compute the distance between the current station and the next station, and if it is larger than the distance you can travel with the current amount of fuel, you refuel at the current state. Otherwise, you do not stop at the current station. Finally, if the distance between two consecutive stations is larger than 100, return infinity.

```
def FindMinStations(i, fuel, n):
    if i == n:
        return 0
    elif (D[i+1] − D[i] > 100):
        return \infinity
    elif (D[i+1] − D[i] > fuel):
        return FindMinStations(i+1, 100 − (D[i+1] − D[i]))
    else:
        return FindMinStations(i, fuel − (D[i+1] − D[i]))
```

The runtime of this algorithm is $\Theta(n)$

Now we want to prove the correctness of the algorithm.

Suppose there is a possible solution. Let S be solution from your greedy algorithm and x be the station that you visit first in S. Let S' be any list of stations that does not contain x and z be the first station in S'. The greedy algorithm chooses the first station where $D[i] - D[0]$ is the maximum value and less than 100. Therefore, we know that $x > z$. $len(S') = 1 + MinStation(D[z+1...n])$ and $len(S) = 1 + MinStation(D[x+1...n])$, which shows that S is not worse than S'. Therefore, the solutio from the greedy algorithm is optimal.

■

(b) Now suppose that you have a choice of routes to get to your destination. The road network is represented as an undirected graph $G = (V, E)$ with weighted edges representing road segments. Fueling stations are located at some of the crossroads (i.e., vertices), so each vertex has a flag to specify whether it contains a fueling station or not. Design and analyze an efficient algorithm to compute the minimum number of

refueling stops you have to make to travel from a given source s to a destination d. Again, your car can travel up to 100 miles before refueling, and you can assume that both s and d have a refueling stop at them.

Solution: We construct a new unweighted graph $G' = (V, E')$ that uses the same vertices as G but has an edge between two vertices u and v if the distance between u and v in G is at most 100. Every valid path in G from s to d that refuels at stops $v_1, \ldots, v_k$ corresponds to the path in G' $s \to v_1 \to \cdots \to v_k \to d$. To compute the shortest such path we can run BFS, since G' is unweighted.

To construct G', we need to compute shortest paths between all pairs of nodes. One way to do this is to call Dijkstra's algorithm using every vertex as the starting node.[1]

```
def FindMinStation(G, s, d):
    # all pairs distance dictionary
    all_pairs = {}
    for v in G.V:
        # assume that Dijkstra returns a dictionary d[] that maps
        # nodes to their shortest path distance from v
        all_pairs[v] = dijkstra(G, v)

    # construct the graph G'
    Gp = Graph()
    Gp.V = G.V
    Gp.E = { (u,v) for u in G.v for v in G.v if all_pairs[u][v] <= 100 }

    # run BFS on G' starting at s and find the distance to d
    n_stations = BFS_distance(Gp, s, d)
    return n_stations
```

This algorithm performs V calls to Dijkstra's algorithm, which has complexity $\Theta(VE \log V)$. It then constructs the new graph G' which could have up to V^2 edges, so $O(V^2)$. Finally we call BFS on G' which has complexity $\Theta(E' + V') = O(V^2 + V)$. This gives us an overall complexity of $\Theta(VE \log V + V^2)$ or simply $\Theta(VE \log V)$ if V is $O(E)$ (which is true in graphs where the minimum node degree is 1). ■

Question 2: Zapping Balloons ....

Solve question 25, parts (a), (b), and (c), from chapter 4 in the textbook. Do not solve part (d).

Solution: (a) We first need to find the angle of the ray that does not intersect with any balloons. This is done in two steps: first we need to check if any balloon intersects with 0. If not, then 0 is the angle. Otherwise, we sort the angles by starting point and find the first balloon with an angle that is larger than both the largest end angle of the 0-intersecting balloons and the largest ending angle of the balloons that start before it:

```
def find_nonint_ray(angles):
    """ angles is a list of starting / ending angles in each balloon.
    Note that a[0] < a[1] unless the balloon covers the angle 0 """
    if not any(s > e for s,e in angles): # no angles intersect 0
        return 0
    # find latest end of balloon that intersects with s
    latest_end = max(e for s,e in angles if s > e)
    angles.sort(lambda x: x[0]) # sort by starting angle
    for s,e in angles:
        if s > latest_end: # found a gap
```

[1] You can optimize slightly by only starting the search at vertices that have a fueling station at them, stopping the execution of Dijkstra once the minimum element in the priority queue has distance > 100, and reusing past computations by adding edges from u to v of length $d(u, v)$, but none of these change the asymptotic complexity in the general case.

```
            return (latest_end - s) / 2
        else:
            # update end of latest balloon
            latest_end = max(latest_end, e)
    raise ValueError("No non-intersecting angle")
```

We can then reorient adjust the balloons to make the non-intersecting ray be at position 0. This leaves us with a problem identical to problem 4. Our greedy strategy is to sort the array by ending times and zap the balloon with the earliest ending time. To show that this strategy is correct, consider the balloon with the earliest ending time and let s, e be its starting and ending times. An optimal solution must have a ray that is between s and e; shifting this ray to be equal to e will result in no fewer balloons being zapped and thus is still an optimal solution. Removing the zapped balloons and recursing shows that the entire solution is optimal.

```
def adjust(angle):
    """ adjust an angle to be positive """
    if angle < 0:
        angle += 360

def count_zaps_adjusted(angles):
    """ counts zaps in angles that have been adjusted so that they
    are all positive, and which have been sorted by ending point """
    last_zap = -1
    count_zaps = 0
    for s, e in angles:
        if s > last_zap:
            # balloon not covered by last zap, use its
            # ending angle as next zap
            last_zap = e
            count_zaps += 1
    return count_zaps


def min_zaps_with_gap(balloons):
    angles = [find_angles(b) for b in balloons]
    start_ray = find_nonint_ray(angles)
    adjusted_angles = [(adjust(s-start_ray),adjust(e-start_ray)) for s,e in angles]
    adjusted_angles.sort(lambda b: b[1]) # sort by ending angle
    return count_zaps_adjusted(adjusted_angles)
```

The runtime complexity of this algorithm is $\Theta(n \log n)$ since the loop bodies, iterating along all balloons, execute in constant time, and sorting the angles takes $\Theta(n \log n)$

(b) In this part, we force the existence of a gap by first sending a zap at angle 0 and then using the previous solution to zap the remaining balloons. Observe that any optimal set of zaps for the full set must also zap all of the balloons that are remaining after the shot at angle 0. Therefore, the optimal set of zaps for the remaining balloons is at most as large as the optimal set of zaps for all of the balloons. Adding the zap at angle 0, the number of zaps is at most one larger than the optimal, as required by the problem.

```
def almost_min_zaps(balloons):
    angles = [find_angles(b) for b in balloons]
    # find balloons that are not zapped by ray at 0
    remaining_angles = [(s,e) for s,e in angles if s < e]
    remaining_angles.sort(lambda b: b[1]) # sort by ending angle
```

```
        return count_zaps_adjusted(remaining_angles) + 1
```

Again the algorithm is $\Theta(n \log n)$ because of the sorting step.

(c) For this part, we use the following observations:

1. There exists an optimal schedule where one of the rays is at the starting angle of some balloon. We can see this to be true by taking any ray and moving it counterclockwise until it is equal to the starting angle of some balloon. This move does not change the set of zapped balloons.
2. Taking out the balloons that are zapped by one ray in an optimal schedule, the remaining zaps are optimal for zapping the remaining balloons.

We thus iterate over all choices of balloons as the starting point and zap each one, then repeat part (a)/(b). The only extra work is that we have to make sure that we only sort the array once, rather than once per iteration, which would result in $O(n^2 \log n)$ complexity.

```
def min_zaps(balloons):
    angles = [find_angles(b) for b in balloons]
    angles.sort(lambda b: b[1]) # sort by ending point
    min_so_far = len(angles)
    for i in range(len(angles)):
        starting_ray = angles[i][0]
        # rotate the array to be angles that follow i, followed by angles that precede i
        rotated_angles = angles[i+1:] + angles[:i]
        # adjust them to be relabeled by positive angles, rotated to make starting_ray = 0
        adjusted_angles = [(adjust(s—starting_ray),adjust(e—starting_ray]) for s,e in rotated_angles]
        # remove any balloons already zapped [which, after adjusting will contain the angle 360]
        remaining_angles = [(s,e) for s,e in adjusted_angles if e < 360 or s > 360]
        min_so_far = min(min_so_far, count_angles_adjusted(remaining_angles))
    return min_so_far
```

We can see that the body of the loop has steps that are at most linear in the number of balloons, resulting in $\Theta(n^2)$ complexity.

■

Question 3: Stacking Books ....

Solve question 21, parts (a), (b), and (c), from chapter 4 in the textbook.

Solution:

(a) We use greedy algorithm to solve this problem. Since each shelf must store a contiguous interval of the given sequence of books, we can put the books on the shelf in the order the cataloging system suggests. New shelves are needed if and only if the current shelf cannot fit the next book in order.

```
1: function MINSHELVES(H, T)
2:     shelves = 0
3:     remaining = L
4:     for i in range(len(H)) do
5:         if T[i] > remaining then
6:             shelves += 1
7:             remaining = L
8:         remaining = remaining - T[i]
9:     return shelves
```

The runtime for this algorithm is O(n), where n is the number of books, since we simply loop through all number of books, and checking if to place the book on the current shelf or the next shelf only takes O(1) time.

Proof: Let $G = g_1, \ldots, g_n$ be the placement of the books on the shelves by the greedy algorithm, where g_i is the shelf number that is used to hold the ith book, and let $O = o_1, \ldots, o_n$ be an optimal placement of the books. To simplify the explanation, add a 0-th book to both placements, with $g_0 = o_o = 1$.

Suppose that O and G agree on the placement of rhte first j books; i.e., $j+1$ is the first index where they differ. Contiguous placement requires that $g_{j+1} = g_j$ or $g_{j+1} = g_j + 1$; likewise for o_{j+1}. Furthermore, we must have that $o_{j+1} = o_j + 1 (= g_j + 1)$ because otherwise we would have $g_{j+1} = g_j + 1$, but we know that book j would fit on shelf g_j (since it is placed there by the optimal order), and so it must have been placed there by the greedy algorithm. Since book $j+1$ fits on shelf g_j in the greedy placement, we can change the optimal placement by moving it to shelf o_j to produce $O' = o_1, \ldots, o_j, g_{j+1} (= o_j = g_j), o_{j+2}, \ldots, o_n$. This placement agrees with g_j on the first $j+1$ books and uses the same number of shelves as O, and is therefore also optimal.

So given an optimal placement that agrees with G on the first j books, we can produce another optimal placement that agrees with G on $j+1$ books. By induction, we can see that there is an optimal placement that agrees with G on all n books; i.e., the greedy placement is optimal. □

(b) Suppose we have three books with H = [1, 10, 10] and T = [5, 5, 5]. Also, assume we have shelves of length 10. If using the algorithm in part (a), we will have to put the first two books on the first shelf and the third book on the second shelf, resulting in a height of 20. However, if we put the first book on the first shelf and the second, third books on the second shelf, we will get a smaller shelf height of 11. Using this counter example, we proved that the greedy algorithm from part (a) does not always give the best solution to this problem.

(c) We use dynamic programming to solve this problem. First, this problem can be solved recursively using the following recurrence. minHeight(i) is the height needed to place the first i books.

$$minHeight(i) = \begin{cases} 0 & \text{if } k = 0 \\ min(minHeight(k) + max(H[k+1], H[k+2], ..., H[i]) | \\ T[k+1] + T[k+2] + ... + T[i] \leq L, 0 \leq k < i) & \text{if } k > 0 \end{cases}$$

To calculate minHeight(i), we calculate heights of all possible arrangements, where books k+1 to i can fit in one shelf, as minHeight(k) + max(H[k+1], H[k+2], ..., H[i]) and take the minimum height. We can save the minHeight(i) in a dp array in a bottom up manner.

```
1: function MINHEIGHT(H, T, L, n)
2:     create an array minHeight[ ] of size n
3:     minHeight[0] = 0
4:     for i from 0 to n do
5:         remaining = L
6:         maxHeight = 0
7:         for k from i to 1 do
8:             maxHeight = max(maxHeight, H[k])
9:             remaining = L - T[k]
10:            if  remaining <= 0 then
11:                break
12:            minHeight[i] = min(minHeight[i], minHeight[k-1] + maxHeight)
13:    return minHeight[n]
```

From the algorithm, we have two nested loops, both of approximately size n. So the runtime for this dynamic programming algorithm is $O(n^2)$

■

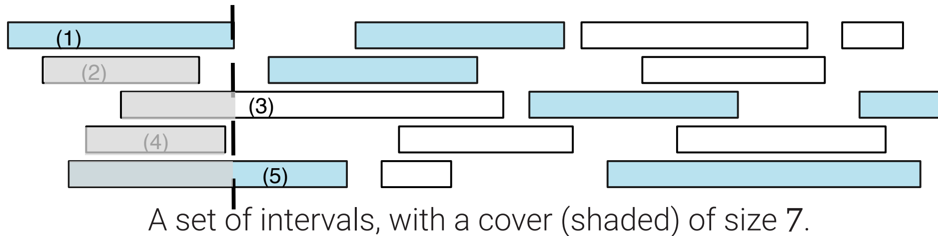
Solved Problem

Question 4: Interval Cover ....

Solve question 3 in chapter 4 of the textbook

Solution: There are two basic observations motivating the solution:

1. The interval that starts first must be included in the cover
2. When deciding between two intervals that start at the same time



A set of intervals, with a cover (shaded) of size 7.

The first observation means that in the figure the first interval we must add to the cover is interval (1). We can then remove any part of any interval that is covered by (1) (shown in grey). Intervals (2) and (4) now completely disappear, while for intervals (3) and (5) we adjust the start time to be the ending time of interval (1) (dashed line). We now apply observation 2 to select interval (3) out of these, and continue.

We can turn this into an algorithm below. Note that it just implements the strategy above, most of the complexity is in bookkeeping. In particular, we need to distinguish when the current interval overlaps with some other intervals that extend past it and the case when there's a gap between intervals.

```
def mimimum_cover(intervals):
    """ intervals are specified as a tuple (S,F) denoting the starting and finishing point of each interval """
    intervals.sort()                          # sort by starting point

    # current selected interval. (assumes that len(intervals) > 0)
    cur_start, cur_finish = intervals[0]
    # next interval to be selected, which is the interval tha partially overlaps with the current one
    # and has the latest finish time
    next_finish = None

    count = 1 # count the currently selected (first) interval

    for start, finish in intervals[1:]:
        if start == cur_start:                # this interval starts at the same time as the currently selected interval
            if finish > cur_finish:           # upgrade to this interval if it ends later
                cur_finish = finish
        elif start < cur_finish:              # this interval partially overlaps with the current
            if (finish > cur_finish and       # this interval is not entirely covered
                (next_finish is None or next_finish < finish)):
                next_finish = finish          # this is the (new) next interval
        else:                                 # this interval starts after the current interval
            if next_finish is not None:       # deal with partial overlap
                count += 1                    # select the next interval
                cur_start = cur_finish
                cur_finish = next_finish
                next_finish = None
                if cur_finish > start:        # this interval partially overlaps with the next (new current) interval
                    if cur_finish < finish:   # not entirely covered
                        next_finish = finish
                    continue
            count += 1                        # select this interval
            cur_start, cur_finish = start, finish
    if next_finish is not None:
        count += 1                            # count the last partially overlapping interval
    return count
```

Analysis: For n intervals the loop runs $n-1$ times and it is easy to see that the loop body performs constant work, so the complexity is dominated by the sort call, which is $\Theta(n \log n)$.

Proof of optimality: Our implementation is equivalent to following the following process repeatedly:

1. If there are no intervals partially covered by the currently selected intervals, select the uncovered interval that starts first as the next interval. If there is a tie, select the one that ends latest
2. If the currently selected intervals partially cover some intervals, select the next interval among those, again choosing the one that ends latest

We must prove that, for any optimal cover that includes the currently selected intervals, there is an optimal cover that includes the next one chosen by our algorithm. The optimality of the result will follow by induction.

More formally, suppose that our set of intervals I has an optimal cover $O \subset I$ and a subset $S \subset O$. Let U be the elements of S completely uncovered by S and P be the set of intervals partially covered. Assume further that all elements in P have their starting point in S (it is easy to see that our algorithm always maintains this property).

First consider the case that $P = \emptyset$. Let F be the elements of U that share the earliest starting time. $O - S$ must include one element in F, call it f. Our algorithm chooses the element in F with the latest ending time, f'.

Since f' starts at the same time as f and lasts at least as long, f' covers f, and therefore $S+\{f'\}+(O-S-\{f\})$ is an optimal cover for I. Note also that since f' has the earliest start among elements of U, $S+\{f'\}$ maintains the property that any partially covered elements must have their starting points in $S+\{f'\}$.

Now consider $P \neq \emptyset$. Then $O-S$ must include some element of P, p.[2] Our algorithm chooses the element p' with the latest ending time. Then $S+\{p'\}$ covers $S+\{p\}$ and therefore $S+\{p'\}+(O-S-\{p\})$ is an optimal cover for I.

Alternate solution: We can use the same strategy, but adjust the interval set by "removing" any portion of an interval that is covered by the currently selected cover. This leads to a simpler implementation but $\Theta(n^2)$ complexity

```
1  def min_cover2(intervals):
2      count = 0
3      while len(intervals) > 0:
4          # find best interval to select first
5          best = 0
6          for i in range(1, len(intervals)):
7              if (intervals[i][0] < intervals[best][0] or # starts earlier or
8                  # starts at the same time and is longer
9                  (intervals[i][0] == intervals[best][0] and intervals[i][1] > intervals[best][1])):
10                 best = i
11         # select intervals
12         count += 1
13         cur_start, cur_finish = intervals.pop(best)
14         # select intervals not entirely covered, adjusting the starting time to be >= finish
15         intervals = [(max(cur_finish,start),finish) for start,finish in intervals if finish > cur_finish]
16     return count
```

Note that we may be able to speed up the finding of the best interval by sorting and/or using a priority queue, but line 15 is still $O(n)$ resulting in quadratic overall complexity. ■

Rubric:

- 2 pt for identifying a correct greedy strategy
- 2 pt for proof of optimality
  - -1 for each minor mistake in strategy or proof, but
  - -10 if strategy is wrong or proof is wrong or missing
- 2 pt for correct pseudocode or Python implementation
  - -0.5 for not handling corner cases
  - -1 for other mistakes
- 2 pt for correct algorithm runtime analysis
- 2 pt for efficiency
  - -1 for $\Theta(n^2)$
  - -2 for slower polynomial algorithm
  - -2 if runtime analysis is incorrect
  - -4 for exponential solution

A correct greedy algorithm without a (mostly) correct optimality proof will receive 0 points.

A correct $\Theta(n \log n)$ DP algorithm receives 6 points (missing the first two items).

A correct backtracking solution will receive 2 points (2/2 correct implementation, 2/2 correct analysis, -4/2 for efficiency)

[2] To be precise, this requires that our definition of "partially covered" include the case where the end of an interval in S is equal to the start of an interval in P. The algorithm above follows that convention.