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.

(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.

Question 2: Zapping Balloons ....

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

Question 3: Stacking Books ....

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

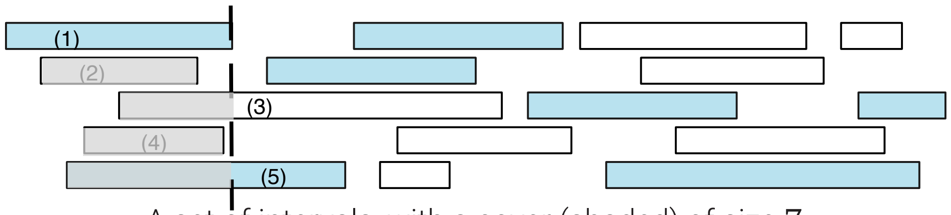
Solved Problem

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.[1] 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. ■

[1] 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.

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)