CSCI 3104 Spring 2018 Problem Set #6
Alison Ostlund (05/22)
Worked with: Kyle Ward, Hannah Haines, Vamshi Arugonda, Matt Donovan, Kyle Gronberg

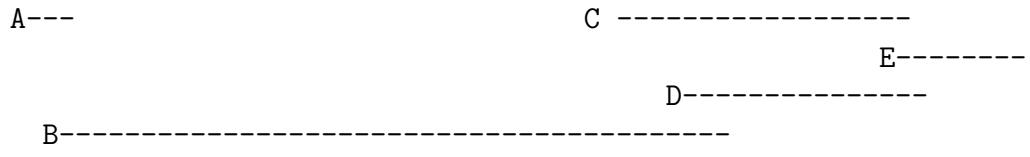
---

1. (15 pts) An exhibition is going on in Hogsmeade today from t = 0 (9am) to t = 720 (6pm), and world-renowned wizards will attend!! There are n wizards W1, . . . , W_n and each wizard W_j will attend during a time interval I_j : $[s_j, e_j]$where in $0 \leq s_j < e_j \leq 720$. Note: the ends of the interval are inclusive. The stores in Hogsmeade want to broadcast magical ads in the sky during the exhibition, multiple times during the day.In particular, each wizard must see the ad but the store also wants to minimize the number of times the ad must be shown. For example:

```
            Wizard | [s,e]
Minerva McGonagall | [3, 51]
      Harry Potter | [6,60]
       Ron Weasley | [6,99]
  Hermione Granger | [105,155]
 Gilderoy Lockhart | [121, 178]
       Viktor Krum | 86, 186]
```

Then, if the ad is shown at times t1 = 51 and t2 = 150, then all 6 of the wizards will see the ad.

(a) Greedy algorithm A selects a time instance when the maximum number of wizards are present simultaneously. An ad is scheduled at this time and the wizards who see this ad are then removed from further consideration. The algorithm A is then applied recursively to the remaining wizards. Give an example where this algorithm shows more than the minimum number of ads needed.



```
WizardA[1,10] WizardB[5,65] WizardC[26,70] WizardD[40,75] WizardE[68,82]
Greedy: 3 ads
Optimal: 2 ads
```

This is an example of when the algorithm shows more than the minimum number of ads needed. The max amount of times is when wizards B,C,D are present, so an ad will show during these times. After this ad these three wizards will be removed from further consideration, leaving behind the two other wizards A and E who have not seen an ad. These two wizards are not present during the same time range so in order for all wizards to see the ads, two separate ads will have to be shown during the times of wizards A and E. This causes a total of 3 ads to be shown when the optimal amount would've been 2 ads for [A and B] and [C,D, and E].

(b) (10 pts extra credit) Let Wj represent the renowned wizard who leaves first and let $[s_j, e_j]$ be the time interval for W_j . Suppose we have some solution $t_1, t_2, ..., t_k$ for the ad times that cover all wizards. Let t_1 be the earliest ad time. Prove the following facts for the earliest scheduled ad (at time t_1). For each part, your proof must clearly spell out the argument. Overly long explanations or proofs by examples will receive no credit.
S1. Prove $t_1 \leq e_j$. (Three sentences. Hint: proof by contradiction.)
S2. If $t_1 < s_j$, then t1 can be deleted, and the remaining ads still form a valid solution. (Five sentences. Hint: suppose deleting t1 leaves results in a wizard not seeing the ad; think about when that wizard must have arrived and left relative to t1, sj , ej. Prove a contradiction.)
S3. If $t_1 < e_j$, then t1 can be modified to be equal to ej, while still remaining a valid solution. (Three sentences. Hint: suppose setting t1 := ej leaves a wizard uncoveredthat is, without having seen an adthen when should that wizard have arrived and left? Prove a contradiction.)

S1 : $t_1 \leq e_j$: Let $t_1 \geq e_j$, this is saying that the first ad is aired after the first wizard leaves. If this is true then not every wizard will see the ad which cannot not happen, therefore $t_1 \leq e_j$ is a valid statement.
S2 : $t_1 < s_j$ then delete t_1. Let $t_1 > s_j$. If this is true the first ad will show after the the first wizard has arrived. So if $t_1 < s_j$ this means that the first ad will be shown before the first wizard has even arrived. Since there will b no wizards present before s_j occurs $t_1 < s_j$ is impossible because that means there would

have to be a wizard present before s_j had even arrived. Because this case cannot happen, t_1 can be deleted.
S3 : $t_1 < e_j$ then $t_1 = e_j$. If $t_1 = e_j$ then the first wizard is leaving as the first as is shown and therefore will see the ad. If $t_1 > e_j$ then the wizard leaves before the ad is shown which then implies that the wizard has left before e_j,which is impossible because the Wizard has arrived at s_j and leaves at e_j. So $t_1 = e_j$ because the wizard will not have left yet.

(c) (c) Use the results stated in (1b) to design a greedy algorithm that is optimal.
(i) Write pseudocode for your algorithm.
(ii) Prove that your algorithm is correct (assuming the results stated in (1b)) and give its running time complexity.
(iii) Demonstrate the solution your algorithm yields when applied to the n = 6 example above.

i.)

```
RunAd()
   adTime = []       \\ O(1)
   while wizards != NULL    \\O(n)
      exit = getMin(wizards) \\first wizard to leave - exit time
      for j = 0 in wizards  \\O(n) \\getting start times of other wizards
          if wizard[j]sj <= exit \\compares exit time to any start time less
                 delete wizard[j] + shift left \\ deletes any wizard from lis
      adTime.append(exit) \\appends exit time to list
   return adTime
```

ii.) The running time is at most $\mathcal{O}(n^2)$ since there is a for-loop nested within a while loop. Assuming all parts of (1b) are correct,the algorithm first tries to find the instances where there are wizards present. From there the algorithm can set exit to the time of the first wizard to leave. Then it goes into a for loop where the algorithm gets the start times of the other wizards. From there it compares the start times of the other wizards to the current wizards exit time. If there is any start times less than this time, those wizards will get deleted(S2). Then the list will shift left to get the other wizards left in the list. Once out of the for

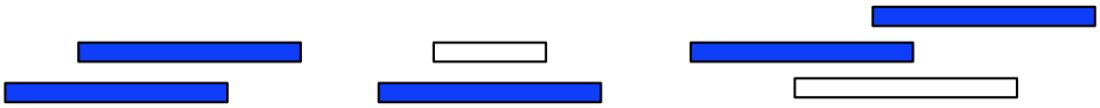
loop, it appends the exit time and re enters the loop to get the next exit time, ultimately getting the range of times for the ads. This ensures the correctness of the algorithm.

iii.) When applied to the example above:

```
First Loop:
 -exit = McGonagall [3, 51] - exit time of 51
 -Compares 51 to start time of Harry Potter ( 6 < 51) and Ron Weasley (6 < 51),
 - Deletes Harry Potter, Ron Weasley and McGonagall from list
 -Appends 51
 Second Loop:
 - exit = Hermione Granger [105,155] - exit time of 155
 - Compares 155 to start times of Gilderoy Lockhart(121< 155) and Viktor Krum (
 - Deletes Gilderoy Lockhart,Viktor Krum, and Hermione Granger from list
 - Appends 155
 -Nothing left in list, return appended list.

 -Gives range of (51 - 155) for optimal ad time so that all wizards will see th
```

2. (20 pts) Professor Dumbledore needs helpers to watch the gates as much as is possible. In order to minimize disruption to their class schedules, he asks students and professors when they are available, and they each provide a set of time ranges. To simplify scheduling matters, Prof. Dumbledore will simply select a set of these ranges and assign the relevant peoplethat is, he never assigns someone just a part of one of their ranges. Given a set S of n time ranges on a given day, he asks you to find a subset T of these ranges which is covering, in the sense that every time that could be covered by someone according to all the ranges S, is covered by one of the ranges in T. Your goal is to minimize the size of T (=the number of ranges it contains, regardless of how long they are). For the following, assume that Dumbledore gives you an input consisting of a single array S where the i-th element S[i] describes the i-th range as a pair $(s_i\ ,\ l_i)$ where $s_i < l_i$



(a) In pseudo-code, give a greedy algorithm that computes the minimum-size covering subset T in O(n log(n)) time. Explain your solution in plain English as well, and prove an O(n log(n)) upper bound on its running time. (Hint: Start with the earliest range.)

```
MinRange(times[])
   T = []     \\ O(1)
   QuickSort using the start times  \\ uses nlogn
   while i <= n     \\ n
     start = times[si]  ]
     end = times[li]
     i++
     while times[si] == start \\ <n
          if times [li] > end
             end = times[li]
             max = times[wi]
           i++
     while times[si] > start && times [si] < end  \\ <n
           if times [li] > end
              end = times[li]
              max = times[wi]
           i++
     T.append(max)
 return T
```

Total Time complexity: O(nlogn + n + n + n+ n) = O(n log n)

The algorithm above takes in a list of the class schedules and sorts it by the start time of the schedules using quick sort. This then sets the si to the first start time in the list and the li to that same persons end time.The algorithm then checks if anyone else has a similar start time . If there is someone else with the same start time but has a longer range of time where the end time is larger, then the algorithm decides to use this person for the shift and extends the li to this persons end time. Once everyones schedules have been checked against the same si, the algorithm will find people who have overlapping schedules. It will find anyone whose start is after the set start time for the current shift and whose start time is before the set end time for the overlapping schedule. The algorithm will do this

and keep finding more people to cover the shifts and keep on checking all of their end times. If anyones end time are later than the sets end, the set end time will extend to this time, making it the new n time range and that persons shift. Once there are no more overlapping times, then the algorithm is done because it has solved that sub problem and will move onto the next group of people in the list of schedule times.

(b) Prove that your algorithm is correct, in particular, that it correctly computes the minimum-size covering subset.

```
S1: times [li] > end
if two people have similar start times then we set the end = li. If li was >
end that means that this person has a schedule that ends after the sets end
time. This would cause the shift to end before the next persons end shift
is, which is impossible because someone has to work the whole shift so end =
li.
S2:  while times[si] > start && times [si] < end
          if times [li] > end
   If overlapping schedules, check if set start si < persons start time and
if that start time is less than the end of the set. If si > start time that
means the persons shift starts before the actual shift starts.No one can
start working their shift before the actual shift has started. And if si> end
this would mean that the persons shift would start after the set has event
ended, which is impossible because no one can work after the set shift has
ended.
   Then if li > end if someones shift ends after the set shift, then this
person is working longer than needed. So need to set li = end so they are
working the entire shift.
```

3. (20 pts) We saw on the previous problem set that the cashiers (greedy) algorithm for making change doesnt handle arbitrary denominations optimally. In this problem youll develop a dynamic programming solution which does, but with a slight twist. Suppose we have at our disposal an arbitrary number of cursed coins of each denomination $d_1, d_2, ..., d_k$, with $d_1 < d_2 < ... < d_k$, and we need to provide n cents in change. We will always have $d_1 = 1$, so that we are assured we can make change for any value of n. The curse on the coins is that in any one exchange between people, with the exception

of i = 2, if coins of denomination d_i are used, then coins of denomination d_{i1} cannot be used. Our goal is to make change using the minimal number of these cursed coins (in a single exchange, i.e., the curse applies).

(a) For i ϵ 1, . . . , k, n ϵ N, and b ϵ 0, 1, let C(i, n, b) denote the number of cursed coins needed to make n cents in change using only the first i denominations $d_1, d_2, ..., d_i$, where d_{i1} is allowed to be used if and only if i $\leq$ 2 or b = 0. That is, b is a Boolean flag variable indicating whether we are excluding denomination d_{i1} or not (b = 1 means exclude it). Write down a recurrence relation for C and prove it is correct. Be sure to include the base case.

```
 Base Case:
C(1,n,b) = 1  - use pennies
C(i,n,b) = 1 for n <= d_2 -1  - if smaller than smallest coin size use pennies

when b=1 cannot use i -1 (next denomination) so recurrence relation is:

    1.{ C( i-1-b,n,0) if di > n
C = 2.{ min[C(i,n-di,1)+1 , C(i-1,n,0)]  if di <= n , b = 0
    3.{ min[C(i, n-di,1)+1 , C(i-2, n, 0)] if di <= n, b =1
1. if coin denomination is larger than amount n
2. if haven't used current coin either use current coin or next denomination
3. case when cant use current coin -1 ( next denomination) so use current coin

Ex:
if n = 19,
cannot use 25cents bc di > n (case 1)
can use case 2, next denomination is 10 cents, 10 < 19, so
can either use current coin of 10 cents to get 19 - 10 = 9 or use next coin
denomination of 5 cents, 19- 5 = 11. Use min, so use dime and 9 cents is
left. Next case because we used the current coin of 10 cents, so b =1, we
cannot use the next denomination of 5 cents, so have to use either the
current coin again of 10 cents or base case where we use 9 pennies.  9 < 10
so cannot use current coin again, return to base case.
```

(b) Based on your recurrence relation, describe the order in which a dynamic programming table for C(i, n, b) should be filled in.

i don't know.

(c) Based on your description in part (b), write down pseudocode for a dynamic programming solution to this problem, and give a Θ bound on its running time (remember, this requires proving both an upper and a lower bound).

i don't know.