CSCI 3104 Spring 2018 Problem Set 6
Erika Bailon (09/28)

Problem 1

(15 pts) An exhibition is going on in Hogsmeade today from $t = 0$ (9 am) to $t = 720$ (6 pm), and world-renowned wizards will attend! There are n wizards $W_1, \cdots, W_n$ 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 $t_1 = 51$ and $t_2 = 150$, then all 6 of the wizards will see the ad.

(a) *Greedy algorithm $\mathcal{A}$ 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.*

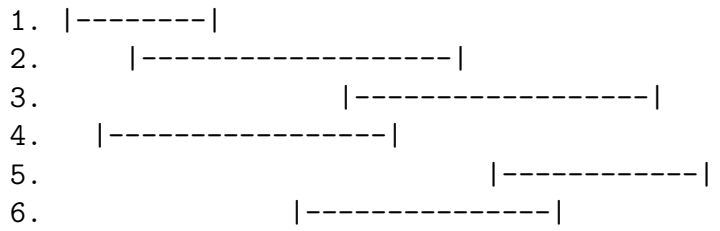
Give an example where this algorithm shows more than the minimum number of ads needed.

To give an example lets assume that we have 6 wizards that have the following

times of arrival (a) and departure (d):

Wizard	[a, d]
1.Eric Oropeza	[5, 15]
2.Jason Lubrano	[8, 55]
3.Luis Vazquez	[40, 120]
4.Selena Quintanilla	[7, 50]
5.Jarrod TheCA	[20, 140]
6.Erika Bailon	[30, 90]

Given this table we can see a diagram like this one (with the order we have them in the table):



The greedy algorithm would want to pick first the maximum number of wizards that can see the ad at the same time. As we can see in the diagram, and evaluating the information in the table, the algorithm will pick the 4 wizards that can see the ad at the same time, which are wizard 2, 3, 4, and 6. After the algorithm picked the MAXIMUM number of wizards that can see the add at the same time, it will go to the next wizards to evaluate using the same process, but the 2 wizards left have no overlap. Therefore, it will have to show the ad once for wizard number 1, and then show the ad for wizard number 5. This will give us a **total of 3 showings**.

With an optimal algorithm, we could obtain a better result. Sorting the wizards by time of leaving the exhibition will help reduce the number of ads to show. We can see that wizard number 1 is the $first$ wizard to leave so we can show an ad right at the time when he's leaving, so he can still see the add, and all the wizards that are present in that time will get to see the ad too, which will be wizard number 2 and 4. We eliminate this wizards and we are left with 3 more wizards. We see the $first$ one to leave again which is the wizard number 6. We repeat showing the ad at the time he's going to leave so he's still able to see it and we see that the wizards that are able to see it at the same time are the 2 wizards that were left, wizard 5 and 3. Therefore, the ad **was shown 2 times** and we covered all the wizards.

That is an example of when the greedy algorithm showed more ads than what was necessary

(b) *(10 pts extra credit) Let W_j 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 $t_1, t_2, \cdots, t_k$ 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 t_1 can be deleted, and the remaining ads still form a valid solution. (Five sentences. Hint: suppose deleting t_1 leaves results in a wizard not seeing the ad; think about when that wizard must have arrived and left relative to t_1, s_j, e_j. Prove a contradiction.)

S3. If $t_1 < e_j$, then t_1 can be modified to be equal to e_j , while still remaining a valid solution. (Three sentences. Hint: suppose setting $t_1 := e_j$ leaves a wizard uncovered–that is, without having seen an ad–then when should that wizard have arrived and left? Prove a contradiction.)

S1. Prove $\mathbf{t_1 \leq e_j}$

Proof by Contradiction:

Lets assume that $t_1 > e_j$. This means that the time when the first ad t_1 is shown, is greater than the time of when the $first$ wizard leaves, e_j. If the ad is shown after the $first$ wizard leaves, then this wizard W_j will not be able to see the add. Therefore, *not all the wizards* get to see the add. This contradicts the assumption that *the ad time covers all wizards* and therefore $t_1 \leq e_j$.

S2. If $\mathbf{t_1 < s_j}$ can be deleted, and the remaining ads still form a valid solution

Proof by contradiction:

Lets assume that if we have a time t_1 smaller than the time when the first wizard arrives s_j, then it **cannot** be deleted. This will mean that there is another wizard W_h who got to the exhibition $first$ but left $after$ our wizard W_j. Which means that $s_h \leq s_j \leq e_j \leq e_h$. Since the ad was shown before s_j, then our wizard W_j, since *is contained* in the time of wizard W_h, will not get to see the ad because the algorithm will go to e_h to look at the best time to show the add for the wizard that are present $after$ e_h. W_j will not see the ad, and that contradicts our assumption, therefore t_1 **can** be deleted and we consider a better time.

S3. If $\mathbf{t_1 < e_j}$, then $\mathbf{t_1}$ can be modified to be equal to $\mathbf{e_j}$

Proof by contradiction:

We assumed that t_1 CANNOT be modified to be equal to the time e_j. Now lets assume that we have a wizard W_k that arrives at the exact moment when W_j leaves, this means $e_j = s_k$, and they are our only two wizards. Because of the condition fo t_1 being strictly smaller than e_j then we will have to show the ad to W_j before he leaves and then we would have to show the ad to wizard W_k also before he leaves.

This will not give an optimal solution since the ads would have to be displayed twice even though we have the overlap $e_j = s_k$. Therefore, we can modify $t_1 < e_j$ to be $t_1 <= e_j$ to give an optimal solution for the overlapping cases such as $e_j = s_k$.

(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.*

First **(i)**

```
def optimalSolution(allWizards):
    arrayOfAds = [ ]
    sort(allWizards)   //sorting  by end time = [e1 < e2 <... <en]
    while (len(allWizards) > 0):
        showingTimes = allWizards [0,1]  //flag to catch the end time
        wizardsToRemove = [ ]
        for (i=1 to len(allWizards-1)):
            if (allWizards [i,0] <= showingTimes)
            and (showingTimes <= allWizards[i,1])
                wizardsToRemove.append(i)
        for j in wizardsToRemove:
            delete allWizards[j] from allWizards
        arrayOfAds.append(showingTimes)
    return arrayOfAds
```

Second **(ii)**

To prove that the algorithm is correct based on the conditions we gave on $1b$, we look at the base case where we have one wizard. We have an already sorted array of wizards because we only have one. The size of the array is 1, which is greater than 0 so we set a showing time at the time when the wizard leaves, that is way the array is set to $[0, 1]$ because is like turning one the ending time. We do not have more wizards so we go to append the array of ads and we return it to know when we are showing the ad.

Now, if we have more than one wizard, we enter the *while loop* set the array of ending times and then we enter our for *loop*. We then fall into our if statement and we don't check all the elements, we are limiting our loop to those that fall into the conditions that their time or arrival and time of leaving are in between the times of the $first$ wizard to leave that we picked. All the wizards that fall into that condition of being between the times of our $first$ wizard to leave, we append them to the array that contains the wizards to be removed. Then we remove them from the original array. This guarantees that we WOULD NOT check at those wizards again because they already saw the ad. Then we

append our array of ads with the showing time we have found from the first wizard that is leaving. Then we enter the loop ALL OVER AGAIN so we keep working with the same process on the wizards that haven't seen the ad. This is guaranteed to happen with all of the wizards because we only get rid of the ones that saw the ad together with our first wizard to leave.

For the running time, we start with the sorting running time which is $O(n \lg(n))$. For the *while loop* we get:

$$\sum_{i=i}^{n} n - i + 1 = n(n - i + 1) = n^2 - in + n = n^2.$$

Therefore from the *while loop* we obtain $O(n^2)$.
From this two running times we obtain that:

$T(n) = n \lg(n) + n^2$ for which we obtain $\mathbf{\Theta(n^2)}$

Third **(iii)**
We can see that our wizards are already sorted by leave time.

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]

We take the time of our $first$ wizard to leave which is $Minerva$. We set the ad to be shown at the time when she leaves. We then check which wizards are in the range of that time and we end up eliminating *Minerva, Harry and Ron*. We append the array with that time showing. Then we go and look again at out array which now only contains *Hermione, Gilderoy and Viktor*. The first one to leave is $Hermione$ so we set the ad at the time she leaves, we check which wizards are in that time and we see that *Gilderoy and Viktor* are in there at that time, so we delete *Hermione, Gilderoy and Viktor*. We then append the array that contains the time of showings with this time. We have no more wizards to show the add to so we return the array **[51, 155]**. Because those were the times we append.

Problem 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 people–that is, he never assigns someone just a part of one of their ranges.

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 $\mathcal{O}(n \log n)$ time. Explain your solution in plain English as well, and prove an $\mathcal{O}(n \log n)$ upper bound on its running time. (Hint: Start with the earliest range.)*

```
def letsTile(times):
    theShifts = [ ]
    sortFunction(times)                //we sort by start time
    temp = empty
    i=0
    while i < n:
        if temp == empty:
            temp = times[i]
            i++
            while (i < n and times[i,0] <= temp[0] and temp[1] < times[i,1]:
                temp = times[i]
                i++
            add temp to theShifts.       //found the best person's start time
        j = -1.    //to reset every subset
        while (i <  n and times[i,0] <= temp[1]:
            if times[i,1] > temp[1]:
                if j<0 or times[i,1] > times[j,1]:
                    j = 1     //making sure we get rid of useless candidates
            i++
        if j > 0:
            temp = times[j]
            add temp to theShifts
            j = -1
        if times[i,0] > temp[1]:
            temp = empty
    return theShifts
```

**Please note - I am writing the running through the explanation so I can do the running time at the end of the explanation. Also I don't know what is plain English, for me this is plain English, hope is enough.

This algorithms works because we are again as in problem 1, finding subproblems that at the end we put together to find the optimal solution. We first sort our array **(O(n lg(n)))** by the start time because this way we can have better subproblems that will not considerer more elements than needed due to the following:
We enter the first *while* loop **(O(n))** that will help go through each subset we are going to created after finding the best person based on the start time, and the best person based on the end time. After entering the first *while* loop we check that our temp is empty **(C)**, this way we don't override any solution and also for it to finish if we only have a base case, with one person. After we check our temp variable we grab the first time and increment our index so we can start making the comparisons to pick the best person for the shift. We enter the second *while* loop **($\Theta(1)$ this is because we limit the number of elements check that will never go to n so it's a constant, same explanation for the next constant times)** to check if that first person we picked is the best one, we look at who's overlapping with the times of the person and we keep the best one based on the first start time and how long is the person staying **(C)**, we store it in our temp variable. Then we add **(C)** that best start time to our array where we are going to put all the persons that qualify for the best shifts. Then we start another counter that will be reset each time to allow us to analyze each subset. We start now another *while* loop **($\Theta(1)$)** to look for the best person to cover the shift based on its end time to divide that subset and return the best persons and start the new subset. We find the best end time with the if statement **(C)** setting the flags of ending time of each person and setting it equal to our counter j so then once we find it we can add it to our array **(C)** where we have the best candidates for each shift. We have a last if statement to divide our subset **(C)**, and we reset our temp to empty so we are able to find again the best start and the best end time for our next subset. Then we go back to our first if statement and start all over again for the next subset. This algorithms will end when we have found all the best shifts because once we looked at all the possible times, our counter will run out the range compared to n so it will take us out of the first while loop and will return the array with the best times for each shift.

Running time:
$T(n) = O(n \lg(n)) + O(n)$ (The constants and the $\Theta(n)$ are smaller and are constants that when are multiplied by $O(n)$ we still have the worst case scenario as $O(n)$.
Therefore we get $\mathbf{O(n \lg(n))}$

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

As I stated in my "plain english" explanation the algorithm works because it divides

the array given in subsets and always finds the best person who has the best starting time that last enough to have the minimum number of person covering the shift by the time the best person with end time leaves in each subset.

For our base case lets assume we have three people S_1 , S_2 and S_3 such that $s_1 < l_1 < s_3 < l_3 < s_2 < l_2$ .We are going to enter the the while loop and set our temp to S_1. In the while loop with the conditions we make sure that the s_i we are going to take is the smallest between the other start times. We then insert s_1 on our array $theShifts$. We have found the best person to start the shift. Then we go to the other while loop and check which one has the best ending time, then with the second if statement inside that loop we are going to check which person's time are overlapping with the ending time we have already selected and if it does then is useless for us because there's no point to have two persons from which one is going to just be overlapping the entire time, so we set j equals to one, only with the person who's starting time overlaps with the person we selected as the best start time but who doesn't overlap with another person that overlaps with him/her but doesn't overlaps with the best person with start time. Then we add this person to our array with the best shifts and we insert all the rest, in this case S_2 to the temp array so we can delete it. At this point we have reached out iterations of i and will fall out of the loops and will return our array with the solutions S_1 and S_3. Since we have proven that it works for one subset, we are guaranteed that it will work for all the subsets in the array.

Problem 3

(20 pts) We saw on the previous problem set that the cashier?s (greedy) algorithm for making change doesn?t handle arbitrary denominations optimally. In this problem you?ll 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* $d_1, d_2, \cdots, d_k$*, with* $d_1 < d_2 < \cdots < d_k$*, 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_{i-1} *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* $i \in \{1, \cdots, k\}$*,* $n \in \mathbb{N}$*, and* $b \in \{0, 1\}$*, let* $C(i, n, b)$ *denote the number of cursed coins needed to make* n *cents in change using only the first* i *denominations* $d_1, d_2, \cdots, d_i$*, where* d_{i-1} *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_{i-1} *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 Cases:
C(1,n,b) = n where n is the pennies for n cents
C(i,n,b) = n if n < d_2.

Now:
If $d_i > n$ and $i > 2$ the cursed coins are C(i-1,n,0)
If $d_i \leq n$ and $b = 0$ the cursed coins are C(i,n-d_1,1) or C(i-1,n,0)
If $d_i \leq n$ and $b = 1$ the cursed coins are min(C(i,n-d_i,1),C(i-2,n,0))
We can construct our piecewise function as follows:

$$C(i, n, b) = \begin{cases} C(i - 1 - b, n, 0) \text{ if } d_i > n \\ min(C(i, n - d, 1), C(i - 1, n, 0)) \text{ if } d_i \leq n \text{ and } b = 0 \\ min(C(i, n - d, 1), C(i - 2, n, 0)) \text{ if } d_i \leq n \text{ and } b = 1 \end{cases}$$

Our recurrence relation is correct proving with induction. Our base case, when n is pennies, then pennies is the first denomination we have, then we are able to give change with n pennies since we only have cursed coins when $i \geq 2$. To prove that it works for $n + 1$ we assumed is true for the best case, therefore once we have i greater than 2, we use the table we have with the denominations to eliminate the denominations that are before the ith denomination we are using. Because we are using the minimum based on the results from previous denominations, we

are guaranteed to obtain the minimum over all the possibles ways of returning the change.

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

Our dynamic programming table will be a 3D table since we have the *ith* axis, the n axis, and the boolean axis which is either 0 or 1. We will start by filling up the base cases. Secondly we will filled out each iteration with the minimum between $C(i, n-d, 1), C(i-1, n, 0)$ or the minimum between $C(i, n-d, 1), C(i-2, n, 0)$ depending on the boolean value of each denomination. We are filling up from top to bottom, from left to right.

(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).*

```
def koins(mination,n):
    k=len(mination)
    theDynamic = new int[k][n][2] //creating 3D array of size k*n*2
    for b=0 to 1:
        for m=1 to n:
            theDynamic[1][m][b] = n
        for m=1 to mination[2]-1:
            for i=1 to k:
                theDynamic[i][m][b] = n
    for i=2 to k:
        for m=mination[2] to n:
            if mination[i] > m:
                for b=0 to 1:
                    theDynamic[i][m][b] = theDynamic[i-1-b][n][0]
            else:
                if b==0:
                    theDynamic[i,m,b] = min(theDynamic[i][n-theDynamic[i]][1],
                    theDynamic[i-1][n][0])
                else:
                    theDynamic[i][m][k] = min(theDynamic[i][n-theDynamic[i]][1],
                    theDynamic[i-2][n][0])
        return theDynamic
```

Running time:
$T(n) = 2(n + k(d_2 - 1) + Kn$
$T(n) = 2n + 2d_2K - 2K + Kn$
$T(n) = K \cdot n$
$\Rightarrow \mathbf{\Theta(Kn)}$

We can give a Theta bound because in our algorithm we are considering both, the worst case, and the best case, therefore the running time is already a Theta bound. The algorithms runs entirely the same amount of times. There are no conditions that will stop it before its done evaluating.

WORKED WITH = Jarrod the CA and Eric OropezaElwood