
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.*

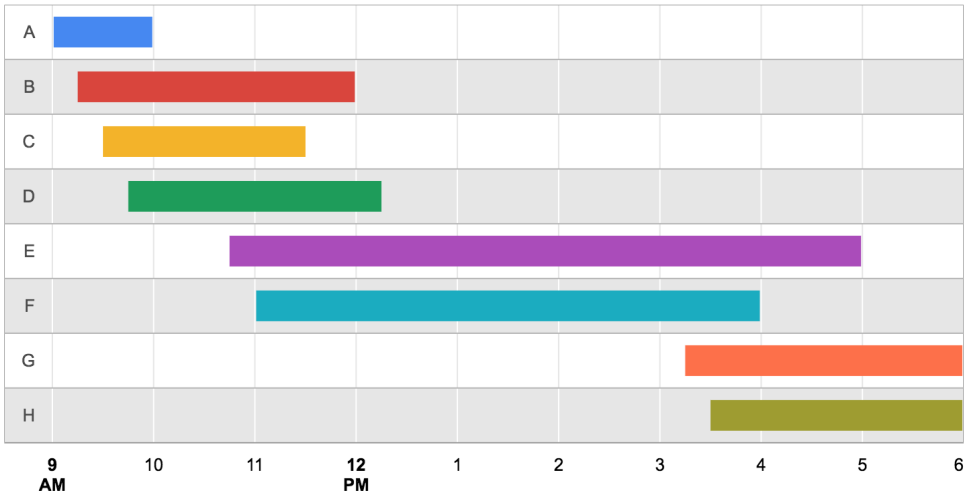
For example:

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.



`Used https://jsfiddle.net for the graph. The code is appended at the end of the file.`

What we can see from the graph above is:
First intersection: Wizards A, B, C, D
Second intersection: Wizards B, C, D, E, F
Third intersection: Wizards E, F, G, H

This is what is going to happen when we use Greedy algorithm A:
- In the first recursion, the algorithm will choose the second intersection(B, C, D, E, F)
- In the second recursion, G and H.
- In the third, and the last, A will be chosen.
So, the number of the shown ads will be 3 when we use this algorithm.

However, if we choose (E, F, G, H) in the first recursion, then in the second recursion all the remaining wizards(A, B, C, D) will intersect.
So, the minimum number of ads is, actually, 2. This means the algorithm can show more ads than the minimum number of ads needed.

(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 $t_1 \leq e_j$
Let's assume $t_1 > e_j$ is true. This means that the first ad will be shown after the first wizard leaves. That cannot be true because W_j will miss the ad. Hence, $t_1 \leq e_j$ is true.

S2. If $t_1 < s_j$, then t_1 can be deleted, and the remaining ads still form a valid solution
Let's assume deleting t_1 results a wizard not seeing the ad. This means the wizard that didn't see the ad, let's call him A, arrived before t_1, and left after t_1 and before s_j. That means A has left before the first wizard to leave, W_j, and that contradicts the assumption that W_j is the first wizard to leave.
Hence, if $t_1 < s_j$, it can be deleted and no wizard will miss the ad.

S3. If $t_1 < e_j$, then t_1 can be modified to be equal to e_j, while still remaining a valid solution
Let's assume changing t_1 to be equal to e_j leaves a wizard uncovered. This means that wizard left before e_j. We know this cannot be true since e_j is when the first wizard leaves.
Hence, if $t_1 < e_j$, it can be changed to be equal to e_j while still remaining a valid solution.

(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) Pseudocode

```
magical_algorithm(A):
    """
    A is a list of wizards' time intervals
    return a list of the ads
    """
    T = [] #empty
    index = 0
    A = quicksort(A) #sort A in an increasing order depending on the departure time
    #running time of quick sort is nlog(n)
    while(A not empty):
        T.append(A[0][1]) #append the first wizard leaving time to T list
        A.remove(A[0]) #remove it and shift the element(intervals) to the left
        for element in A:
            #element is an interval starts from s_j to e_j
            if T[index] in element:
                A.remove(element)

        index += 1

    return T
```

`(ii) Prove correctness and give running time complexity`
Using loop invariant:
`Initialization:` Before starting the loop, T is empty. That is always going to be true because there are no ads shown and the list of wizards' intervals is not modified.
`Maintenance:` A is sorted before the loop, so the earliest elements is chosen. It will be appended to T list and removed from A. We will compare the element's leaving time if it intersects with any other element. If so, we will remove those elements. This is true because in the next iteration, we will append the second ad's time, and will compare it to the other elements.
`Termination:` After the loop, we will have a list that have the times of the ads to be shown. This will always hold true because if we have one wizard or more, there will be ads to be shown. If there are no wizards, then no ads will be shown.

The running time complexity is $O(n^2)$ because we have two inner loops. It is big-o because the size of the inner loop is less than the size of the outer loop.

`(iii) Demonstration`
`Before the while loop:`
A = [[3,51], [6, 60], [6, 99], [105, 155], [121, 178], [86, 186]]
T = []

`After first iteration:`
A = [[105, 155], [121, 178], [86, 186]]
T = [51]

`After second iteration:`
A = []
T = [51, 155]

We can see that we got an optimal solution. Two ads will be shown at times 51 and 155.

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.

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 $\mathcal{O}(n \log n)$ 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.)*

```
#Written in python
0 def magic(A):
1     A = sorted(A) #assume it is quicksort, running time is nlog(n)
2     T = []
3     T.append(A[0])
4     index = 0
5     for i in range(1, len(A)):
6          if(T[index][1] > A[i][0]):
7               T[index] = [T[index][0], max(T[index][1], A[i][1])]
8     else:
9          index += 1;
10          T.append(A[i])
11
12   return T
```

`line 1`: Sort the list A, which has the all time ranges, in an increasing order
`line 2`: Create a list T, which eventually will store the minimum ranges from A
`lines 3, 4`: Store the first element(range) from A at T. Create index variable to keep track of T list.
`line 5`: Iterate through the array A beginning from the second element

`lines 6-7:` Compare the ending time of the range(i) with the starting time of range(i+1). If range(i) ends after the starting time of range(i+1), then modify the range in T, so the ending time will the maximum of [range(i) ending time, range(i+1) ending time].
`lines 8-10:` Else, store range(i+1) in the list T.
Iterate again through the loop.

`Prove running time is` $O(n \log n)$`:`
We will iterate through the loop once [lines 5-10], and that is $n - 1$ iterations.
In line 1, we sort the array using quick sort, which we proved before is $\Theta(n \log n)$
So, $T(n) = n \log n + (n - 1) + C$, which is in asymptotic analysis is $O(n \log n)$

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

Using loop invariant:

`Initialization [lines 1-4]:`
Before entering the loop, T has one element which is the first range in the set of ranges. This is the initial condition and it is always true for all sets of ranges.

`Maintenance: [lines 5-10]:`
At the beginning of the loop, if range(i) ending time is greater than range(i+1) starting time, modify range(i) [which is in list T] so that range(i) and range(i+1) are merged by taking the max between range(i) ending time and range(i+1) ending time.Else, append range(i+1) in the T list.
Then we will do the same thing in each iteration. In each iteration, we merge the ranges if they intersect, append the second range otherwise. We can see that we compute the minimum-size covering subsets of the ranges by storing them in T. So, this is always going to be true

`Termination:`
After the termination of the loop. We will have a T list that has at least one element if A is not empty, and no element if A is empty. Therefore, this is always going to be true for all sets of ranges.

Problem 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* $d_1, d_2, \cdots, d_k$*, 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.*

The recurrence relation is:
$C(i, n, b) = C(i - 2, n\%d_i, b) + \lfloor n/d_i \rfloor$

And there are two base cases:
$C(1, n, 0) = 1$ [when $n > 0$]
$C(1, n, 0) = 0$ [when $n = 0$]

I proven the recurrence relation is correct from the code I wrote.
It is below in part c and explain itself.

(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 do not 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).*

```
    def C(i, n, b, d):
'''
i is the index (starting from 1)
n is the amount of money to make change for
b is a flag
d is a list of denominations
return number of coins used
'''
numOfCoins = 0

#base case
if(b == 0):
tot = 0
# print "i ", i
# print "n ", n
while(tot < n):
# print tot
if(d[1] <= (n - tot)): #second element
tot += d[1]
else:
tot += d[0]
numOfCoins += 1
else:
tot = 0
while(tot < n):
if(d[i-1] <= (n-tot)):
tot += d[i-1]
numOfCoins += 1
else:
if(i-2 <= 2):
numOfCoins += C(i-2, n-tot, 0, d)
else:
numOfCoins += C(i-2, n-tot, 1, d)
break

return numOfCoins
```

The running time is $\Theta(n \log n)$
Because we use recurrence calls for $i - 2$, and it is kind of a balanced tree.
The depth of balanced trees is $\Theta(n \log n)$

Problem 1(a) code:
I found a sample code, modified it to draw the graph

```
google.charts.load('current', {'packages':['timeline']});
google.charts.setOnLoadCallback(drawChart);

function drawChart() {
  var data = google.visualization.arrayToDataTable([
    ['Activity', 'Start Time', 'End Time'],
    ['A',
     new Date(2014, 10, 15, 9),
     new Date(2014, 10, 15, 10)],
    ['B',
     new Date(2014, 10, 15, 9, 15),
     new Date(2014, 10, 15, 12)],
    ['C',
     new Date(2014, 10, 15, 9, 30),
     new Date(2014, 10, 15, 11, 30)],
    ['D',
     new Date(2014, 10, 15, 9, 45),
     new Date(2014, 10, 15, 12, 15)],
    ['E',
     new Date(2014, 10, 15, 10, 45),
     new Date(2014, 10, 15, 17)],
    ['F',
     new Date(2014, 10,  15, 11),
     new Date(2014, 10,  15, 16)],
    ['G',
     new Date(2014, 10, 15, 15, 15),
     new Date(2014, 10,  15, 18)],
    ['H',
     new Date(2014, 10,  15, 15, 30),
     new Date(2014, 10,  15, 18)]
  ]);

  var options = {
    height: 450,
  };

  var chart = new google.visualization.Timeline(document.getElementById('chart_div'));

  chart.draw(data, options);
}
```

I worked with:
Yazeed Almuqwishi, Abdullah Bajkhaif, Mahmoud Almansori, Firas Almakhrouki, Yousif Barnawi, Omar Mohammed