Name: Jon Abramson

ID: 107084898

CSCI 3104
Problem Set 6

Profs. Grochow & Layer
Spring 2019, CU-Boulder

Quick links: 1a 1b 1c 2a 2b 2c 2d 3a 3b 3c

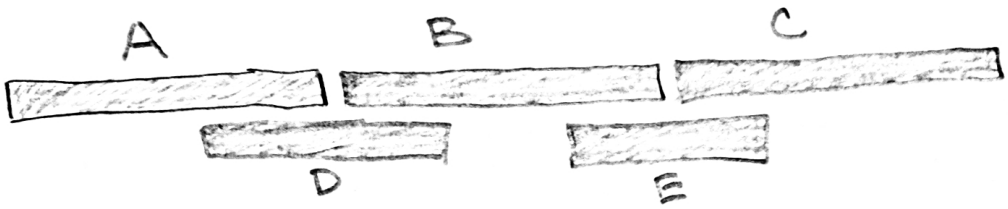
1. As a budding expert in algorithms, you decide that your semester service project will be to offer free technical interview prep sessions to your fellow students. Not surprisingly, you are immediately swamped with appointment requests at all different times from students applying many different companies, some with more rigorous interviews than others (i.e., some will need more help than others). Let A be the set of n appointment requests. Each appointment a_i in A is a pair $(start_i, end_i)$ of times and $end_i > start_i$. To manage all of these requests and to help the most student students that you can, you develop a greedy algorithm to help you manage which appointments you can keep and which ones you have to drop (you can only tutor one student at a time).

Name: Jon Abrahamson
ID: 107028898

CSCI 3104
Problem Set 6

Profs. Grochow & Layer
Spring 2019, CU-Boulder

(a) (2 points) Draw an example with at least 5 appointments where a greedy algorithm that selects the shortest appointment will fail.

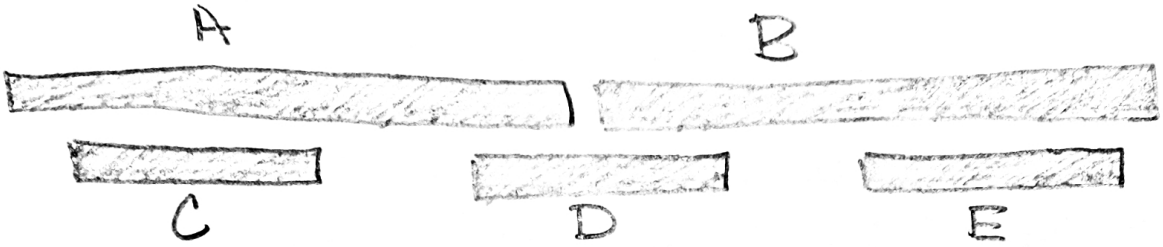


$D_{end} - D_{start}$ & $E_{end} - E_{start}$ are the two smallest values and only allow for those two students to be tutored using this greedy algorithm. The optimal solution is to take students A, B, C despite having larger time commitment

Name: Jon Abrahamson
ID: 107024892

CSCI 3104 | **Profs. Grochow & Layer**
Problem Set 6 | **Spring 2019, CU-Boulder**

(b) (2 points) Draw an example with at least 5 appointments where a greedy algorithm that selects the longest appointment will fail.



Students A & B have the longest appointments, but of the 5, only these two can be tutored using this greedy algorithm. An optimal algorithm would select Students C, D, and E.

Name: Jon Abrahamson
ID: 107084898

CSCI 3104
Problem Set 6

Profs. Grochow & Layer
Spring 2019, CU-Boulder

(c) (6 points) Describe and prove correctness for a greedy algorithm that is guaranteed to choose the subset of appointments that will help the maximum number of students that you help.

The optimal greedy algorithm would find and select the student who would finish his tutoring time the earliest. The student with the smallest end$_i$. From there, throw out any students who overlap with the chosen student. once finished, repeat the steps until all students have been sorted

Proof by contradiction

if set x is all of the students $\{x_1, x_2, \ldots x_n\}$ and set z is the optimal set of students $\{z_1, z_2, \ldots z_m\}$, then x is only optimal if $n=m$ and $m>n$.

$\therefore$ z has an extra element z_{m+1}.

z_{m+1} starts after the end of z_m and so also after x_n ends

$\therefore$ After throwing out all students who conflict with $\{x_1, x_2, \ldots x_n\}$

The algorithm stops after x_n, so it does not look to see if z_{m+1} works (assuming it does) then the set produced by the greedy algorithm would not be optimal because it left out student z_{m+1}. Thus contradicting

2. While your algorithm is clearly efficient and can provably help the most students, you begin to receive complaints from students that you didnt help (i.e., their appointment was not part of the optimal solution). One of the students even offers to pay extra, which gives you a great idea. You will now allow students to make a donation to your favorite charity to make it more likely that their job will be selected. Let each appointment in this new set of appointments A be a triple $(start_i, end_i, donation_i)$ of start and end times and donation amounts where $end_i > start_i$ and $donationu_i > 0$. You now need to update your algorithm to handle these donations along with the requested appointment times. In this new environment, you are trying to maximize the amount of money you raise for your charity.

Name: Jon Abrahamson
ID: 107084898

CSCI 3104
Problem Set 6

Profs. Grochow & Layer
Spring 2019, CU-Boulder

(a) (2 points) Give a specific case were your greedy algorithm would fail.

A = [(1, 3, 5), (5, 7, 20), (5:30, 7:45, 50), (7, 9, 5), (8, 9, 100)]

The algorithm from 1c would give subset {(1,3,5), (5,7,20), (7,9,5)}

This would yield a donation of $5 + $20 + $5 = 30$

However the optimal subset would yield $155

{(1,3,5), (5:30, 7:45, 50), (8,9,100)}

So, the greedy algorithm failed to give the optimal subset

(b) (5 points) Give a recursive algorithm that would solve this new case.

```
mergeSort(A)
MaxD(A, current, chosenAppts):
    if len(A) == 0:
        return 0
    else if len(A) == 1:
        return A[0].donation
    for nextAvail = current-1 to 0:
        if A[nextAvail].endTime < A[current].startTime:
            Previous[current] = nextAvail
            break; # next non-overlapping appt
    dTaken = A[current].donation + MaxD(A, nextAvail, chosenAppts)
    dNotTaken = MaxD(A, current+1, chosenAppts)
    maxDonation = dNotTaken
    if dTaken > dNotTaken:
        chosenApps.append(current)
        maxDonation = dTaken
    return maxDonation
```

This algorithm maximizes the amount of donations received by recursively calling itself on each possible subset of non-overlapping appointments. Starting with the first appointment, we find both the total donations if we take the current appointment & total donations if we do not take the current appointment. We do this by recursively calling our function on both options, either adding the current appointments donation to a recursive call starting at the next non-overlapping appointment or executing a recursive call starting at the next appointment

Runtime-Analysis:

$$T(n) = T(n-i) + \Theta(n)$$

when we are taking an appointment, where i is the number of appointments that were past in order to get to the next non-overlapping appointment

$$\therefore T(n) = T(n-1) + \Theta(n)$$

For the case when we don't take an appointment, we can add those together since we do a recursive call on both cases

$$\therefore T(n) = T(n-i) + T(n-1) + \Theta(an)$$

This gives a complexity of $\Theta(n^2)$. Sorting function is $\Theta(n \log n)$ $n \log n < n^2$ so the complexity remains at $\Theta(n^2)$

(c) (3 points) Add memoization to this algorithm.

```
MergeSort(A)
MaxDnem(A, current, chosenAPPTS):
    memo = []
    if memo[current] != NULL:
        return memo[current]
    if len(A) == 0:
        return 0
    else if len(A) == 1:
        return A[0].donation
    for nextAvail = current + 1 to ∞:
        if A[nextAvail].endTime < A[current].startTime:
            Previous[current] = NextAvail
            break;
    dTaken = A[current].donation + MaxDnem(A, nextAvail, chosenAppts)
    dNotTaken = MaxDnem(A, current+1, chosenAppts)
    Max donation = dNotTaken
    if dTaken > dNotTaken:
        chosenAPPS.append(current)
        MaxDonation = dTaken
    memo[current] = maxDonation
    return maxDonation
```

(d) (10 points) Give a bottom-up dynamic programming algorithm.

```
MergeSort(A)
MaxDBU(A, current, chosenApps):
    overlapping = Prev Appt that overlaps
    PrevFit = Prev appt that doesn't overlap
    dTaken = A[I].duration + MaxDBU(A, PrevFit, ChosenApps)
    dNotTaken = MaxDBU(A, overlapping, chosenApps)
    max = dNotTaken
    if dTaken > dNotTaken:
        max = dTaken
        ChosenApps.append(A[i])
    return max
```

Name: Jon Abrahamson
ID: 107084898

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 6 **Spring 2019, CU-Boulder**

3. (30 pts) 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 $d_1, d_2, \ldots, d_k$, with $d_1 > d_2 > \ldots > d_k$, and we need to provide n cents in change. We will always have $d_k = 1$, 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 $i = k-1$, 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).

Name: Jon Abrahamson
ID: 107084898

CSCI 3104
Problem Set 6

Profs. Grochow & Layer
Spring 2019, CU-Boulder

(a) (10 points) For $i \in \{1, \ldots, k\}$, $n \in \mathbb{N}$, 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 last i denominations $d_{k-i+1}, d_{k-i+2}, \ldots, d_k$, where d_{k-i+2} 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 the next denomination d_{k-i+2} 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.

$$C(i,n,b) = \begin{cases} C(i-1-b, n, b) \rightarrow n < d_k \\ \min(C(i-1, n, 0), C(i, n-d_i, 1)+1) \rightarrow n \geq d_k \wedge b = 0 \\ \min(C(i-2, n, 0), C(i, n-d_i, 1)+1) \rightarrow n \geq d_k \wedge b = 1 \\ n \rightarrow i = k \vee n < d_{k+1} \\ 1 \rightarrow n = d_k \end{cases}$$

Proof of Correctness:

Base Cases: We know $C(k, n, b) = n$ (like with 1 denomination) will take n of those coins to make n change. Similarly, if $n > d_{k+1}$, then $C(i, n, b) = n$ for any i (n couldn't be represented with any denominations larger than d_k) so it would take n of d_k to make n change

Case 1: $n = d_k$, in this case it will take 1 coin of d_k to make 'n'

Case 2: $n < d_k$ because d_i is greater than n, it is not possible to make n change with any coins greater or equal to d_i

Case 3: $n \geq d_k$. When $b=1$, the minimum number of coins used will be related to one of two possibilities. One of which could be $C(i, n-d_i, 1)+1$. Here it will take one more coin of d_i to make change than for $n-d_i$. Or, the minimum number of coins could be equal to $C(i-2, n, 1)$, because d_{i-1} denomination is cursed and can't be used, but the next lowest denomination can be. So, For $b=1$, $C(i, n, b)$ will be the minimum of these 2 values

When $n \geq d_i$ and $b=0$, the value of $C(i, n, b)$ will also be the minumum of the two values. It can be equal to $C(i, n-d_i, 1)+1$ as if the user decides to add one coin of d_k to $n-d_i$ to make n, they will no longer be able to use the previous denominations. The other value it could be equal to is $C(i-1, n, 0)$ if it's optimal to use the d_k coin, and use only values of lower denominations. So, For $b=0$, $C(i, n, b)$ will be the minimum of these 2 values

(b) (10 points) Based on your recurrence relation, describe the order in which a dynamic programming table for $C(i, n, b)$ should be filled in.

The algorithm will need to build a 3-dimensional array with size k·n·2. The base cases will be filled where any value C(i,n,b) the value in this cell will just be 'n'. Also any value $n < d_{k+1}$, the value will also be 'n'. Now, the algorithm will begin filling the table starting at the lowest row value that is not already fully filled. It will then fill in the row beggining at d_{k+1} till the end. It starts filling the row for b=0, then for b=1. After the row is filled it will repeat with the next lowest valve row that hasn't been filled, using previous rows and the valves to the left of the current cell in order to fully fill the table.

∴ Once filled the solution will be at row k, column n (the bottom right) with b=0

(c) (10 points) 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).

```
CursedChange(d, n)
    k = length(d)
    table = 3D array size [k, n, 2]
    for j = 1 to n:          # initialize base case      C1 n time
        table[1, j, 0] = j
        table[1, j, 1] = j
    for j = 1 to d[2] - 1:   # initialize base case      n((c2(d_{k+1} - 1)) time
        for i = 1 to k:
            table[i, j, 0] = j
            table[i, j, 1] = j
    for i = 2 to k           # rest of table
        for j = d[2] to n:
            for b = 0 to 1:  # runs twice   so time for triple nested for loop
                                           is (k-2)((n-d_{k+1})(2C3))
                if d[i] == j:
                    table[i, j, b] = 1
                else if d[i] > j:
                    table[i, j, b] = table[i - 1 - b, n, b]
                else:
                    if (b == 0):
                        table[i, j, b] = min(table[n - d[i], 1] + 1, table[i - 1, n, 0])
                    else:
                        table[i, j, b] = min(table[i - 2, n, 1], table[i, n - d[i], 1])
    return min(table[k, n, 0])
```

$$2kn + C_1 n + n(c_2(d_{k+1} - 1)) + (k-2)((n - d_{k+1})(2C_3)) = \Theta(kn)$$