Name: Keaton Whitehead
ID: 104668391

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

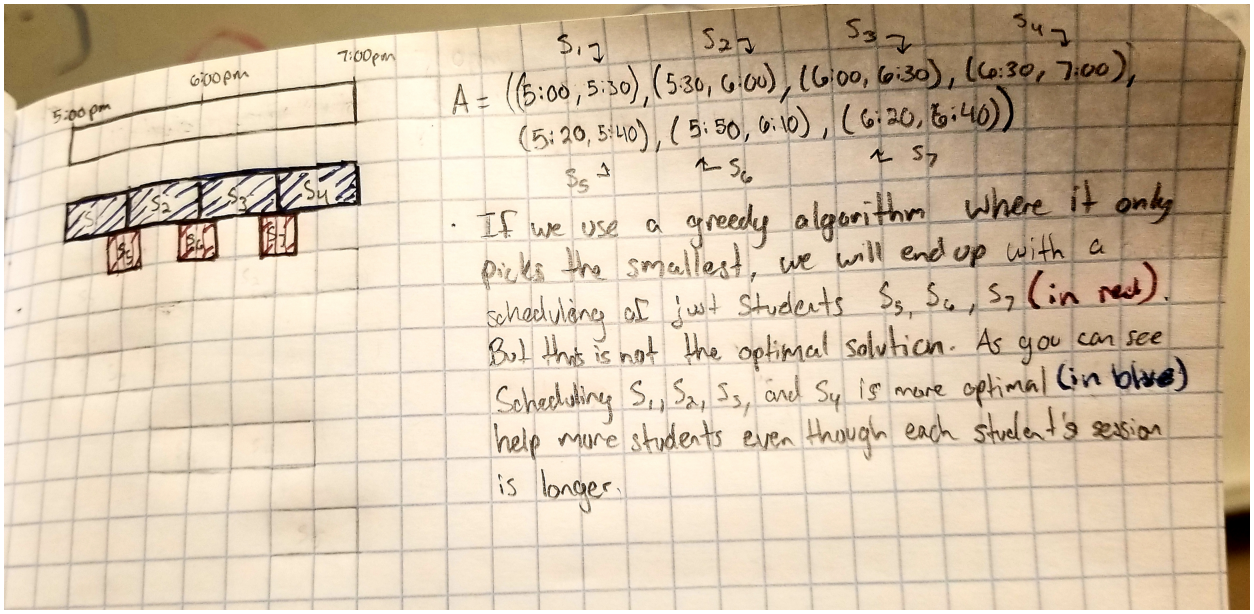
Quick links: 1a 1b 1c 2a 2b 2c 2d 3a 3b 3c

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).

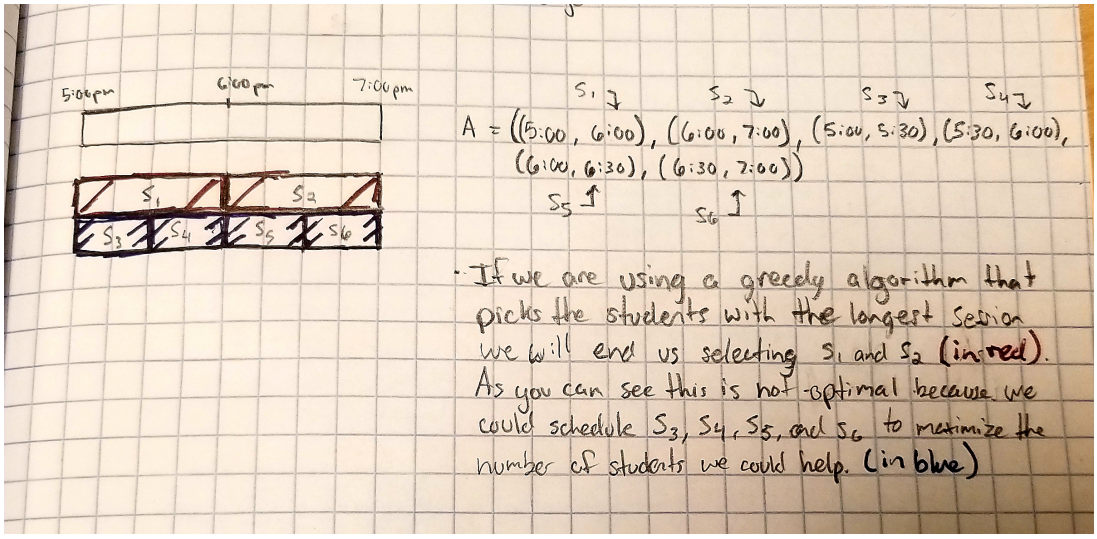
Name: Keaton Whitehead
ID: 104668391

CSCI 3104 **Profs. Grochow & Layer**
Problem Set 6 **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.



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



(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.

i. Let GREED = our greedy algorithm, and let U = set of all possible appointments.
ii. Let S = the subset of appointments that maximize the number of students produced by our greedy algorithm
iii. Let S* be any optional set of appointments, and does not include the greedy choice

Theorem: GREED will produce an optimal subset of appointments
Proof by Contradiction:

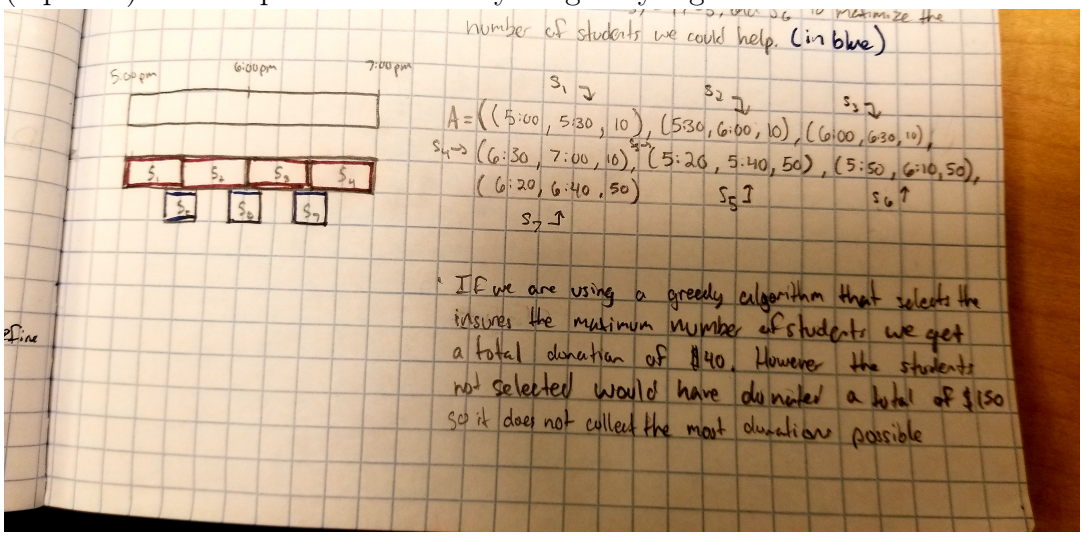
i. Since S* is optimal and does not include the greedy choice, we can modify S* with a greedy choice and show it will not take more time.
ii. In our case, the greedy choice is defined by taking the smallest end - start times, Ex: A ={(5:00,5:30),(5:30,6:00),(6:00,6:30),(6:30, 7:00),(5:20,5:40),(5:50,6:10),(6:20,6
GREED will produce: {(5:00,5:30),(5:30,6:00),(6:00,6:30),(6:30, 7:00) }
iii. **Lemma:** If S is a subset of the appointments that maximize the number of students and S* is an optional subset of appointments then for any appointment x, f(k,S) $\leq$ f(k,S*) where f(k,S) and f(k,S*) is the kth time in the greedy algorithm S and the optimal solution S* respectfully.
iv. If we say k = —S— then f(k,S) $\leq$ f(k,S*). In f(k,S*), there exists an k+1 appointment whose start and end time are after f(k,S*)
v. We know that S produces a subset of appointments that maximize the number of students which means S = U but if we assume $|S|<|S*|$ then S* includes an extra appointment which contradicts U.
vi. By contradiction, the greedy algorithm is optimal

2. While your algorithm is clearly efficient and can probably help the most students, you begin to receive complaints from students that you didn't 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: Keaton Whitehead
ID: 104668391

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

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



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

```
Def maxDough(A,B,n)
    if(length(A) == 0):
        return (b,n)
    else if(length(A) == 1 and length(B) == 0):
        n = n + A[0].dontation
        return (A,n)
    tempTime = [0][0][0]
    maxDonation = 0
    for(i->A)
        if(i.dontaion>maxDonation):
        maxDonation = i.donation
        tempTime = i
    timeColisions = []
    for(i->A)
        if(i[0][1] collides with tempTime[o][1] and i != tempTime)
            timeColisions.append(i)
    x = maxDonaiton(timeColisions,[],0)
    if(x[1]>tempTime.donation):
        A.delete(tempTime)
        maxDonation(A,B,n)
    else:
        B.append(tempTime)
        A.delete(tempTime)
        n = n + tempTime.donation
        maxDonation(A,B,n)
```

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

```
Def maxDough(A,B,n,memo)
    if(length(A) == 0):
        return (b,n)
    else if(length(A) == 1 and length(B) == 0):
        n = n + A[0].dontation
        return (A,n)
    tempTime = [0][0][0]
    maxDonation = 0
    for(i ->A)
        if(i.dontaion>maxDonation):
            maxDonation = i.donation
            tempTime = i
        if(i conflicts with memo[tempTime])                memo[tempTime].append(i)
    x = maxDonaiton(memo[tempTime,[],0)
    if(x[1]>tempTime.donation):
        A.delete(tempTime)
        maxDonation(A,B,n)
    else:
        B.append(tempTime)
        A.delete(tempTime)
        n = n + tempTime.donation
        maxDonation(A,B,n)
```

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

Look at 2c, The memoization makes the algorithm bottom up.

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: Keaton Whitehead
ID: 104668391

(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.

$$T(n) = 2T(n/2) + C$$

// i = amount of denominations in wallet
// n = change you want to make
//b = if the coin has been used before

f(i,n,b){ //base case
if(n == d[k-i+1]){ return 1 // return coin}
if(n == 0) { return 0}
if (i <= 1) { return n }
if(D[k-1+1] <n) {
if(b == 0){ return min(**f(i,n-d[k-i+1],1)**, **f(i-1,n,0))** }
else { return min(**f(i,n-d[k-i+1],1)**, **f(i-2,n,0))**}
}
else {
if(b == 0){ return **f(i-1,n,0)**}
else { return **f(i-2,n,0)**}
}

If you look at the algorithm I use can see that for a worse case we will always fall the function at least twice each iteration, which is how I go a = 2, and the worst case also includes that every other coin can not be used to maximize the amount of coins that we have to use so the number in the array we pass each time is n/2. Then we add on a constant because we always have the same number of simple atomic operations each iteration. Using Master theorem we find that we get a time complexity of $\theta(nlongn)$

(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.

Based on our code our table should be filled in top down. The recurrence relation builds off of its previous values, but it first fills out all the base cases. We also know that a 3 dimensional table is made with dimensions n by k by 2. The 2 is constructed because of the flags placed on the coins if they have been used or not to determine if we implement the curse or not. The first layer will be b = 0 and the second being b = 1. The case where f(1,n,b) = n' will fill the entire row with n itself, which makes sense because if you are only allowed to use the smallest denomination, then it will be 1 therefore = n. From there three conditionals are used to determine which recurrence relation is used to fill the table. These recurrence relations will loop through all of the way until they reach their base case, then will roll back up adding a +1 where necessary. Which recurrence relation that is used is determined if the previous coin has been used, and if the n - denomination ≥ 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).

```
d = []
memo = [][][] // table of size k by n by b
f(i,n,b){       //base case
    if(memo[d[k-i+1]][n][d]) { return memo[d[k-i+1]][n][b]}
    else {
        if(n == d[k-i+1]){ return 1 // return + 1 coin used}
        if(n == 0) { return 0}
        if (i <= 1) { return n }
        if(d[k-1+1] <n) {
            if(b == 0){ value = min(f(i,n-d[k-i+1],1)+1,f(i-1,n,0)
                memo[d[k-i+1]][n][b] = value
                return value}
            else { value = min(f(i,n-d[k-i+1],1)+1,f(i-2,n,0)
                memo[d[k-i+1]][n][b] = value
                return value}
        }
        else {
            if(b == 0){ return f(i-1,n,0)}
            else { return f(i-2,n,0)}
        }
```

The new memoized function will have a time of θ (2kn), no matter if it is the worst or the best case. This is proven in the code and it will always fill the entire array for the worst and best case therefor it is the upper and lower bound.