Name: Trevor Buck

ID: 109081318

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: Trevor Buck

ID: 109081318

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.

A = [(1:00, 1:25), (1:30, 1:55),

(2:20, 2:35), (2:00, 2:25),

(2:30, 2:55)]

vs

Greedy
1. 2:20 → 2:35
2. 1:00 → 1:25
3. 1:30 → 1:55

Optimal
1. 1:00 → 1:25
2. 1:30 → 1:55
3. 2:00 → 2:25
4. 2:30 → 2:55

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

A = [(1:00, 3:00), (1:05, 1:15), (1:20, 1:30), (1:35, 1:45) (1:50, 2:00)]

New Greedy vs Optimal

New Greedy	Optimal
1 1:00 - 3:00	1 1:05 - 1:15
	2 1:20 - 1:30
	3 1:35 - 1:45
	4 1:50 - 2:00

Name: Trevor Buck
ID: 109081318

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.

DESCRIPTION
Sort the list by the earliest finishing times
Then take the first appointment. Reject all overlapping appointments. Repeat until you have reached the end of your sorted list

Psuedo

```
counter = 0
sortbyend (A)
optimallist [0] = A[0]
for (i=1 to |A|-1):
    if (A[i].start > optimallist [counter].end):
        counter ++
        optimallist [counter] = A[i]
return optimallist
```

Runtime
sort → Θ(nlogn)
loop → n
$\boxed{\text{total } (\Theta n\log n)}$

Proof by contradiction

Given:

- set of appointments = $s_1, s_2, \dots, s_n$
- optimal solution = $b_1, b_2, \dots, b_o$
- $|S| \neq |O|$ ie, optimal solution has less elements than the total number of possible elements

If S is not an optimal solution, then set O contains an element b_{n+1}. This appointment starts after b_n and hence after s_n. This means that the set of all possible appointments (A) still contains b_{n+1}.

This implies a contradiction and the element b_{n+1} does not exist in the set of optimal solutions and therefor, $n = o$.

Name: Trevor Buck

ID: 109081318

CSCI 3104
Problem Set 6

Profs. Grochow & Layer
Spring 2019, CU-Boulder

2. While your algorithm is clearly efficient and can proveably 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: Trevor Buck
ID: 109081318

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.

LET A = [(1:30, 1:40, 0), (1:45, 1:50, 0)
(2:00, 2:10, 0), (2:15, 2:25, 0)
(1:45, 2:20, 10)]

Greedy		vs	New Greedy	
① 1:30 → 1:40	\$0		① 1:45 → 2:20	\$10
② 1:45 → 1:55	\$0			
③ 2:00 → 2:10	\$0			
④ 2:15 → 2:25	\$0			
total: \$0			total: \$10	

Name: Trevor Buck
ID: 109081318

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

```
int Newgreedy (int i, int dono)

    if (i<0) return 0

    if (appointments[i] overlaps schedule):
        return newgreedy (i-1, dono)

    else {
        int took = newgreedy (i-1,
                      dono + appointments[i].dono)

        int left = newgreedy (i-1, dono)

        return max (took, left)
    }
```

Runtime

took, left → 2^n

everything else → $\Theta(1)$

$\boxed{\text{total} = \Theta(2^n)}$

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

```
sort by end (A)
//FILL TABLE

table[0] = A[0].dono

for (int i=1 to |A|-1):

    int next = next job in array that doesn't overlap
    int total = A[i].profit + A[next].profit
    table[i] = max(total, table[i-1].profit

return table[i-1]
```

Runtime

sort → $\Theta(n\log n)$

Loop → n

$\boxed{\text{total} = \Theta(n\log n)}$

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

Sortbyend (A)

for (i = 1 to |A|)

earlierjob [i] = latestjobthatendsbefore A[i] starts

OPT[0] = 0

for (j = 1 to n)

OPT[j] = max (A[j].dono + OPT[earlierjob[j]],

OPT[j-1])

Runtime

Sort → nlogn

for loop → n

2nd forloop → n

$\boxed{\text{total: sort} \rightarrow \text{nlogn}}$

Name: Trevor Buck

ID: 109081318

CSCI 3104
Problem Set 6

Profs. Grochow & Layer
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: Trevor Buck
ID: 109081318

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.

```
int C(int i, int n, boolean b)
    if (n == Ø) return Ø
    if (i ≤ 2) b = true
    if (b == false) return C(i+1, n, !b)

    int counter = 0
    while (n > d[i])
        counter++
        n = n - d[i]
    return counter + C(i+1, n, !b)
```

Proof by loop invariance

Invariant: conter = number of 'd[k] coins' we used

Initialization:

- counter = 0
- n = # of change still needed
- n > d[1]

Maintenance: if n > d[k] then we can be greedy, and take one of those coins (d[k]). This should decrease 'n' by the amount we took, and increase our counter by one, since we took one coin

Termination: Our loop ends when we cant take any more d[k] coins.

Algorithm logic: Assuming that our loop changes our 'counter' integer by the number of d[k] coins we can take All we need to do is make sure we skip the next denomination (d[k+1]) and then continue until n = 0.

(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.

TABLE: # of cents

denominations

	1	2	3	4	...	n
d_1						
d_2						
d_3						
d_4						
d_5						

$\boxed{\text{Row by row}}$

Steps

- Fill the top row (d_1) in with n
- For d_2, compare by looking backwards d_2 spaces or up 1 space
  - take the lower number -
- THE exception: if ($b==1$) you have to look two spaces upwards in your comparison

Name: Trevor Buck
ID: 109081318

CSCI 3104
Problem Set 6

Profs. Grochow & Layer
Spring 2019, CU-Boulder

(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).

```
// Base case
for (i = 1 to n):
    table[1][i] = i

for (d = 2 to k):
    for (i = 1 to n):
        table[d][i] = min(table[d][i-d.value] + 1,
                          table[d-(b+1)][i])

return table[d][i]
```

where: value = denomination
b = boolean

Runtime

1st loop → n

2nd loop → $|d| * 3^{rd}$ loop $= |d| \cdot n = \boxed{kn}$

worst case

ie upper bound

3rd loop → $n * \Theta(1) = n$

Runtime, lower bound

We still have to fill every box in our table. Our table is size $|A| \cdot n = \boxed{kn}$

Even at our fastest, our runtime is $\boxed{\Theta(kn}$