HW #3 - Problem 2

Ryan St.Pierre (ras70)

September 27, 2017

Problem 2A

The algorithm can be described with the following steps:

1. Let $i = 1$.
2. Choose the student that can do the most consecutive steps starting at step i.
3. Let the student chosen in Step 2 do their consecutive steps $i....j$.
4. Set $i = j + 1$
5. If j equals the number of steps, n, the algorithm is complete, else repeat Steps 2-5.

To make this algorithm a little more concrete pseudo-code is provided below.

```
/*
n: number of steps
m: number of students
b: array of list of steps each student can do.
    ith value in b gives the steps student i can do
*/
int [] physicsExperiment(int n, int m, b) {

    a[] = new int[m];

    for (i = 1; i<=n; i++) {
        (student, numberSteps) = maximumNumberOfSteps(i,b);

        for j = 0 to numberSteps - 1
            a[i+j-1] = student;

        i = i + numberSteps - 1;
        // we want i = i + numberSteps
        // i will equal this value after i++ is applied
    }

    return a;
}

/*
returns a tuple (a,b)
a: student that can do the most steps starting at step i
b: number of steps student a can do
*/
maximumNumberOfSteps(i,b);
```

Running time

The running time is given by,

$$\mathbf{O(nm)}$$

The reasoning behind this running time is kind of difficult to see, even from the pseudo-code above. From the pseudo-code it is clear that there is a loop over n, hence the n dependence. The work done in the *for* loop is given by the work done by *maximumNumberOfSteps*. This method's signature is given but the code is not provided.

To find the student with the longest consecutive steps starting at a given step the *maximumNumberOfSteps* method must loop through all of the students and their respective lists. If we assume the average list size of steps for each student is w then we could naively say that the running time is $O(nmw)$. However, this would be incorrect because there is actually a dependence, an inverse dependence to be more specific, between m and w. As w increases each student can do more steps. Thus, the number of iterations in the *for* loop containing n increases. This is equivalent in the pseudo-code to incrementing i by the number of steps taken at the end of the n dependent *for* loop.

To see the $O(nm)$ running time it is probably helpful to think about the worst cases - when w is large and when n is large. The upper bound for w is n. In this case each student can do all of the steps. Therefore, we must do $O(nm)$ work to find the student that can do the most steps, but the outer for loop in the pseudo-code only executes once. Thus, in the case when $w = n$ the running time is $O(nm)$. In the other boundary case, when each student can only do one step $O(m)$ work needs to be done to calculate the student that can do the most steps at each step. This is done at each of the n steps. Therefore, in this case the running time is still $O(nm)$.

To visualize this running time we can picture a n by m grid, where n is the steps that must be covered and m is the number of student. Each time we chose a student to do work the operation is m times w. We can then picture this m by w rectangle taking up space in the grid. We then repeat this process until the entire n by m space is covered. Therefore, the running time is dependent on $n * m$ and not dependent on w, the average student step size.

Problem 2B

Proof of Correctness

Assume that there is a better solution OPT that is more optimal than the greedy algorithm solution ALG.

Let $a[1...n]$ be the solution for ALG, containing the students chosen for steps 1 through n and let $b[1...n]$ be the solution for OPT. Let k be the first index at which OPT choses a different student than ALG, that is k is the first instance where $a[k] \neq b[k]$. Let j be the index of the last consecutive step that the

student at $a[k]$ completes before switching. That is, let j be the largest value such that $a[k] = a[k+1] = a[k+2] = ... = a[j]$ holds, where $j \geq [k]$.

A solution OPT' can be constructed by swapping values from a for k to j to b. That if c is the solution to OPT' then,

$$\begin{cases} c[i] = b[i] = a[i] & 1 \leq i < k \\ c[i] = a[i] & k \leq i \leq j \\ c[i] = b[i] & i > j \end{cases}$$

This switch can be done because ALG specifies that student in a can do the steps done by the students in b for steps $k \leq i \leq j$.

The claim can be made that **OPT' is at least as optimal as OPT or better**. To support this fact several cases must be explored. This is shown below.

Case i

$$a[k] \neq a[k-1]$$

This case corresponds to the case when ALG chooses a new student at step k - meaning there is a switch from step $k-1$ to k. Since $a[i] = b[i]$ for $i < k$, this means OPT must also have a switch from $k-1$ to k. Additionally, by design ALG chooses the student that can do the most consecutive steps at starting at step k. If we let m be the largest value such that $b[k] = b[k+1] = b[k+2] = ... = b[m]$ holds, where $m \geq [k]$, then $m \leq j$ by algorithm design of ALG.

Thus, between steps $k-1$ to $j+1$ OPT must have at least two switches. In the same space, between steps $k-1$ to $j+1$ OPT' has two switches. This is probably best seen in the cases (piece-wise) statement above. Therefore, since OPT' has the same or fewer switches than OPT from $k-1$ to $j+1$ and OPT and OPT' are identical elsewhere then OPT' must be just as good or better than OPT.

Case ii

$$a[k] = a[k-1]$$

This corresponds to the case when OPT differs from ALG at a point where a student chosen by ALG is committing a series of steps. In other words, the assumption that the student chosen by OPT at k finishes committing steps before the student chosen by ALG at some point before k through at least step k does not hold. However, it can still be shown that OPT' is at least as optimal as OPT.

Given the condition that $b[k] \neq b[k-1]$ OPT must have a switch at the $k-1$ to k step. Additionally, since $a[k] = a[k-1]$ and $c[k] = a[k]$, $c[k-1] = a[k-1]$ OPT' must **not** have a switch from the $k-1$ to k step. Thus, through step k OPT' has one less switch than OPT.

Again let m be the largest value such that $b[k] = b[k+1] = b[k+2] = ... = b[m]$ holds, where $m \geq [k]$. Unlike *Case i* we can make no statement about m

with respect to j. At minimum OPT has no switches from step k to $j+1$. This corresponds to the case when $m > j$. If $m \leq j$ then OPT has at least 1 switch from k to $j+1$. OPT' has at most one switch from step k to $j+1$. Therefore, OPT' eliminates the switch from $k-1$ to k and has at most one more switch in the space k to $j+1$ in comparison to OPT. OPT and OPT' are identical elsewhere OPT' must be at least as good as or better than OPT.

Through these two cases it has been shown that OPT' can be constructed in a way that is better or just as optimal as OPT. Additionally, OPT' better resembles the ALG solution. If we iterate this process eventually OPT' will approach ALG, while still being better or just as good as the assumed optimal solution OPT. Thus, ALG must be just as good or better than the optimal solution OPT. If it is just as good then ALG must also be optimal. If ALG is better than OPT then a contradiction holds and OPT is not optimal. However, in this case the contradiction implies that there is no solution more optimal than ALG, making ALG optimal. Therefore, in both cases we have proven that ALG is the optimal algorithm.