Name: Jahan Kuruvilla Cherian

Discussion 1D
4:00pm – 6:00pm
TA: Arman

CS 180 – Homework 5

Exercise 3 – Page 246

The aim of the problem is to find a card such that we have over n/2 occurrences of the card as part of an equivalent class (that is a set of the cards so that they are equal to each other). We can recursively solve this problem using a Divide and Conquer approach.
Essentially divide the set of cards into two piles that are roughly equivalent in sizes and recursively do this to divide each subset in halves. If there are more than n/2 cards that are equivalent in the entire given set then at least one of the sides will have more than half the cards also equivalent to the equivalence class aforementioned. Thus upon each recursive call, at least one of the two calls will return a card belonging to said equivalence class.
However, there can be a majority of equivalent cards in one section of the divide, without the actual equivalence class itself having more than n/2 cards in the overall set. Thus upon getting a majority card per recursive pop, we must check it against all the other cards to ensure that it is indeed holistically a majority. The pseudocode highlighted below demonstrates this algorithm.

```
def majorityCard(list of cards C):
    if |C| == 1: return that card
    if |C| == 2: check if the cards are equivalent, and if so, return either of the card
    C_1 set of the first $\lfloor n/2 \rfloor$ cards, C_2 set of the remaining cards
    if majorityCard(C_1): test returned card against all other cards
    if no card with a majority equivalence yet found:
        if majorityCard(C_2): test returned card against all other cards
    return card from majority equivalence class if found
```

We can prove the correctness of the algorithm by looking at the following observation. If there is an overall majority equivalence class containing said card, then there must be a majority equivalence in at least one of the halves.
The runtime of the algorithm is O(nlogn) for n cards in the given list. This is because of the recursive halfway divide per each step, which ends up reducing the problem set to

halves each time (log$_2$ functionality). The check against other cards is at worst O(n) but doesn't factor into the final runtime. A more formal way of defining this would be that the recurrence for the recursion is: $T(n) \leq 2T\left(\frac{n}{2}\right) + 2n \rightarrow O(nlogn)$.

Exercise 9 – Page 320

a.) To satisfy the first constraint let us just set x to be a constant value that is always above s, say 15. s should have a varying value but always upper bounded by x. To satisfy the second constraint, we can create a sinusoidal like repeating sequence of reboots, that occur over a period of days. To achieve this, we simply start our s at the highest value, say 14 in this case. And then make all the other s values remain at 1. Therefore, the optimal solution would be to reboot every other day, so in a period of 2 days, we always achieve a processing of 14 TB/15 TB and only lose out on 1 TB per day of reboot, resulting in an optimal gain.

b.) Let $opt(i,j)$ represent the optimal amount of work that can be done since day i through to day n, given the most recent reboot was performed j days before. This leads to a two state problem, where we can define the state transitions as follows:

   a. Don't process anything on day i, such that day $i+1$ is the first day after the reboot. This leads to the state equation as $opt(i,j) = opt(i+1,1)$.
   b. Simply continue processing the information such that on day i we process the minimum of the given work x_i and the processing power s_j. This has the equation $opt(i,j) = \min(x_i, s_j) + opt(i+1, j+1)$

Approaching this problem as a top down dynamic programming problem we can say that rebooting on the last day would have no benefit as we wouldn't have another day to take advantage of the reboot, and would just waste the last day on the reboot. Thus we can initialize our state to say $opt(n,j) = \min(x_n, s_j)$. The following pseudocode shows the algorithm in action.

```
def optWork(x, s):
    $opt(n,j) = \min(x_n, s_j)\ for\ all\ j\ in\ range(1,n)$
    for i from n-1 to 1:
        for j from 1 to i:
            $opt(i,j) = \max(\min(x_i, s_j) + opt(i+1, j+1), opt(i+1, 1)$
    return $opt(1,1)$
```

The worst case runtime comes from the nested for loops that can run for effectively n-1 times each leading to O(n^2), wherein each calculation is O(1) and the max is O(1) since it's only between two values.

Exercise 10 – Page 321

a.) Taking the example given in the question with a slight modification, we can show the greedy approach highlighted would fail. Given the following table:

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	50
B	5	1	20	20

The algorithm would say to start at A, and then move at B to stay there until the end, resulting in 5 + 0 + 20 + 20 = 45, while if we just stuck to A we would get 10 + 1 + 1 + 50 = 62 which is clearly higher and thus the optimal solution that the algorithm doesn't give.

b.) Since we have two potential solutions (ending on A or ending on B), we aim to find optimal solutions for both and then just take the max of either. Our states can be defined as $opt(i, A)$ and $opt(i, B)$ representing the maximum value for a plan at minute i that ends of A and B respectively. In either case we have two potentials:

a. At minute $i - 1$ we remain on the same machine, thus leading to $opt(i, A) = a_i + opt(i - 1, A)$ or $opt(i, B) = b_i + opt(i - 1, B)$.

b. In the process of moving from the other machine, which thus means that at minute $i - 2$ we were at the other machine, leading to $opt(i, A) = a_i + opt(i - 2, B)$ or $opt(i, B) = b_i + opt(i - 2, A)$.

These equations lead to the final state equations for A and B as:

$$opt(i, A) = \mathrm{a}_i + \max\,(opt(i - 1, A), opt(i - 2, B))$$
$$opt(i, B) = \mathrm{b}_i + \max\,(opt(i - 1, B), opt(i - 2, A))$$

With the final solution being the max of the two at time *n*, thus solving the problem in a bottom up dynamic programming approach. The initialization would be that $opt(1, A) = a_1$ and $opt(1, B) = b_1$. Because we just do a linear sweep from $i = 2, \dots, n$ computing the optimal solution in O(1) time, we have a runtime of O(n).

Exercise 19 – Page 329

For this problem let us try and break the problem into sub-problems with solutions we can computer. We can greatly simplify the understanding of the problem by assuming s has n total characters, and the repetitions x' and y' of x and y containing exactly n characters then we rephrase by questioning if s is an interleaving of the repetitions x' and y', which prevents us from having to worry about wrap-around, since the lengths of the repetitions are fixed to be the same length.
Essentially if we have $s[j]$ be the j^{th} character of s, and use a slice $s[1:j]$ represent the substring of s from 1 to j, then we can say that s is an interleaving of some x' and y' when the last character of s comes from either x' or y'. When we remove this last character we break the problem down to looking at $s\ [1: n-1]$ with the same prefixes and trying to determine if that is an interleaving. Thus we can say that some $opt(i,j) = true$ if $s[1: j+i]$ is an interleaving of $x'[1:i]$ and $y'[1:j]$. If such an interleaving exists, then as mentioned above we can say that the final character is either $x'[i]$ or $y'[j]$. This gives us the following recurrence relation:

$$opt(i,j) = true\ iff\ (opt(i-1,j) == true\ \&\&\ s[i+j] == x'[i])\,|\,|(opt(i,j-1) == true\ \&\&\ s[i+j] == y'[j])$$

Thus we have the following pseudocode for this algorithm, which will end up running in O(n^2) time to build up our optimal grid and then fill them up in O(1) based on previous sub-problems.

```
def interleave(s,x,y):
    let $x'$, $y'$ be the aforementioned repetitions from $x$ and $y$ respectively
    $opt(0,0) = true$
    for $k$ in range(1 to n):
        for all pairs $(i,j)$ such that $i + j == k$:
            if($opt(i-1,j) == true$ && $s[i+j] == x'[i]$)||($opt(i,j-1) == true$ && $s[i+j] == y'[j]$):
                $opt(i,j) = true$
            else:
                $opt(i,j) = false$
    return true iff there exists some pair $(i,j)$ such that $i + j == n$ such that $opt(i,j) = true$
```

Problem 5 from website

Let us define a state $opt(i)$ that represents the length of the maximum longest non-decreasing subset up to some index i from the original array. By running a bottom up approach for every i^{th} value, running through every previous value j bounded by i we check to see if the number at $i \geq j$ in which case we say that our max length so far is the maximum between the corresponding $opt(j)$ value and the previously known max length. This gives us the following state equation $opt(i) = \max(opt(j)\ \forall\ j \in \{1, \dots, i\}) + 1$ where the +1 indicates the inclusion of the current element. The following pseudocode highlights the algorithm:

```
def longestNonDecreasingSubset(nums):
    $opt(0) = 1$
    fin_max = 0
    for i in range(0 to nums.size()):
        temp_max = 0
        for j in range(0 to i):
            if nums[j] <= nums[i]:
                $temp\_max = \max\ (temp\_max, opt(j))$
        $opt(i) = temp\_\max + 1$
        $fin\_max = \max\ (opt(i), fin\_max)$
    return $fin\_max$
```

This algorithm works because we have an initialization that states an element by itself is by definition a non-decreasing subset. From this point onwards, for every number moving forward, we look through the potential maximum lengths that could be generated by including the numbers up to the current number such that it maintains a non-decreasing sequence. Our optimal solution will store the maximum of such a length, and then be compared to the overall final maximum. Doing a run from 1 to n will then produce the length of the longest non-decreasing subset.

The algorithm runs in O(n^2) time since at worst we could be running through all the previously computed maximum lengths when at position $i == n$ wherein, we will have to run through all $j\ from\ 1\ to\ n$ to find the maximum length before the current position.