Bryan Arnold

1/25/16

CSE 3500

Homework 1

1. In this problem, it is assumed that both the columns and rows of each matrix being multiplied together are of size n. So, there is no need to check whether the two matrices can be multiplied together, as their dimensions are the same. Now, I'll use 2D arrays to represent the matrices with both dimensions being size n. Let A and B represent the 2d arrays representing the matrices for this algorithm design, as follows:

   Input: Matrices A and B, both with n rows and n columns.

   Create new 2D matrix C, with dimensions being n x n, for output of multiplied matrices A and B.

   Create new integer called sum to track each sum in each matrix position of the new matrix C.

```
For loop i, with i starting at 0 and going to less than the length of A (which is n), i++ //row length

        For loop j, starting at 0 and going to less than the length of first row of matrix A (so A[0].length), j++ //col length

                For loop k, starting at 0 and going to less than the length of a dimensions of matrix A, k++ //col/row length (Note: doesn't matter which matrix we use for these loop lengths, since the matrices are of the same dimensions).

        sum = sum + A[i][k] * B[k][j]

end for loop k

C[i][j] = sum; //puts sum of multiplied matrices into correct position

sum = 0; //resets sum variable for next position computation

end for loop j

end for loop i
```

Output the matrix C, the multiplication of A and B, as the output.

For this program, each multiplication and addition of two numbers is considered an operation. So, each time a loop counter iterates, the sum integer is added to itself and the multiplication of two matrices, are considered operations. Since there are three loops, all going the same length of less than n, this would make the runtime n * n * n, or n^3. The addition of sum and the matrices multiplications onto sum itself is only a runtime of 2, so given n dimensions this equates to 2n. So, the overall runtime is n^3 + 2n or O(n^3).

2. This problem is very similar in nature to the problem regarding matching couples we did in class, a stability algorithm problem. So, a Gale Shapley algorithm as follows would show that there exists stable assignment, due to the fact the following algorithm not only matches hospitals to students, but also terminates, as all algorithms should:

   Input: a list of all students and hospitals, all of which are free and not assigned or have any students yet.

```
while there exists a student s who is free and not applied to each hospital's program do

    Let s be this student, and h be the most favorable and not yet applied to hospital.

    if h is free then

        Assign s to hospital h

    else
        Let s1 be h1's currently assigned student.

        if h prefers s1 over s then

            s remains free

        else
            s1 becomes free and s is assigned to h

        else if h prefers s1 over s and h1 prefers s over s1

            s1 is assigned to hospital h and s is assigned to hospital h1

        end if

    end if

end while
```

Output: Declare all hospital’s positions filled by students and display this somehow

This algorithm will eventually terminate when the list of students who have not already applied to all hospital’s programs end, leaving most students assigned to a hospital, but some without as predicted. All the hospitals should have all assignments filled. This is just like the matching problem of couples done in class.

3. This pancake sorting by diameter problem is very like how a selection sort works. It is similar because we want to bring the biggest pancake not yet sorted to the top with one flip, and then take it down to its final position with one more, then repeat this for the remaining pancakes. Since you keep track of the biggest, not currently sorted pancake, you then must move on to the next biggest once that pancake is sorted. This means that for the given number of pancakes n, the runtime gets smaller and smaller for each iteration as the number of sorted pancakes increases. My algorithm is as follows:

Input: input the stack of pancakes in an unsorted order of diameters with their diameters.

Create a new variable called maxDiameter = this tracks the pancake with the biggest diameter. Also, **create** another variable which tracks where in the stack this pancake will be called place.

For loop i, starts at 0 and goes to less than the size of the pancake stack, i++

Set the maxDiameter to the pancake on the top of the stack, then set place to 0 as well.

For loop j, starts at 0 then goes to less than the size of the pancake stack minus I, j++

if pancake in stack[j] has a bigger diameter than the maxDiameter

Set the maxDiameter to stack[j] and the new place value to j as well to track this new pancake.

end if

end for j

Let r = the stack size minus 1 and minus i, and let g = the place value denoted earlier

While g is less than or equal to r **do**

Swap the pancakes in positions denoted by the values of g and r

Then add one to g and subtract one from r

```
end while
end for i
```

Output: Display the now sorted pancake stack in order of diameter somehow

This algorithm will sort the pancakes by diameter size, the biggest being on the bottom and the smallest being on top. The runtime of a selection sort's worst case is known to be n^2, as well as my algorithm will take n^2 time, but in the case of this problem, the runtime can be 2n. The memory allocated would be n for the number of pancakes, and you'd only have to go through the pancakes once making it n as well. This would have allocated (n + k) of memory, k is the size of the first stack flip, and bounded by n – 1, making it 2n – 1, or 2n. I couldn't figure out the algorithm for this runtime, but I could figure it out for n^2 using a selection sort like procedure.

4. If I were Eric in this problem, my goal would be to get two other people at least to agree with my decision so I don't get eaten by the wolves. So, to do this I thought about the problem starting from one person. Obviously, with one person stealing all the bars, they'd just get all of them. Next, with two people, it is stated that at least half of the people there must agree to the decision, so only one vote would be needed out of the two people to make this plan a reality. So, if I were the one person making the plan to split the gold bars, I'd simply take them all, since my idea counts as one vote. This would maximize my profit as well as fulfill the requirements for a valid splitting of the gold bars. Next, with three people, I'd have to convince one of the other people to agree to my plan. Looking back at my previous instance of two people, the person right after me in experience level would want me dead, so he can make the plan that I would've made and taken all 100 bars. So, to avoid this, I'd have to make a deal with the person with the lowest experience. I'd take 99 bars and give them 1, explaining to them if my plan fails, the next person in experience above them would just take all 100 bars and they'd get nothing to begin with, so they'd be forced to take the one bar as their max profit if they want anything at all. So, in the three-person scenario, I'd take 99, the next person gets non, and the last person gets 1. Now, moving on to 4 people, I'd only need half of the total people there to agree to my plan, so I'd only need one other person to agree with me. In this situation, I'd have to convince the person that is second to last in experience with me to agree to the plan. This is because, knowing the instance of three people, the person next in line after me would want me dead so he can make a deal with the last person in experience level. So, to force the second lowest experienced person into agreeing with me, I'd tell them of this situation and offer them 1 gold bar or they'd get none when I'm now dead and they're in the 3-person scenario. So, for 4 people, I'd take 99 bars, the next person would get 0, then the next 1 bar, then the last none. This maximizes my profits and forces another to comply with me if they want any bars at all, and keeps me alive. There is a pattern going on here to force others to comply if they want a cut at all. I must

give a gold bar to the person after the next person in experience level to force them to comply. More specifically, for odd numbered amounts of people I must give it to every other person after me in experience level (so it'd go like me yes, next person none, next person 1, and so on) and the last one being the least experienced person. I would do the same for even numbered amounts of people except I'd end by giving the person before the least experienced person a gold bar. This pattern would allow me to constantly be maximizing profits over everyone else I'm forcing to comply, if they want a cut at all, as well as keeping me alive.

Now that I can start to see a pattern, I can determine what I should do in the scenario I am in if I am Eric. His scenario has 5 people, in which he is the most experienced. This means I'd have to get 2 other people to agree with my plan to stay alive and maximize profits. Given the pattern I just found out, I would need to give gold bars to Chuck and Adam to force them to comply. Since they all want to maximize profits, therefore want the people in front of them dead, this is the only way to force them to comply if they want money at all, seeing as the second to last person can take all 100 if they get their way. So, I'd distribute the wealth as follows, Eric-98, Dave-0, Chuck-1, Bob-0, Adam-1. This scenario would yield the maximum profit for Eric, forcing others to comply if they want any money at all, keeping him alive, and preventing the second to last person from taking all the money since everyone is logical and greedy. This would also satisfy all of them wanting to be alive, as they would die if they didn't comply. This is the best solution for maximum profits for Eric.

Extra Credit: As far as increasing the number of robbers, the same strategy still holds for even and odd numbered amounts of robbers. Give one to the person after the next person in experience level, stop at the least experienced for odd numbered number of robbers, and stop at the second to last experienced for even numbered number of robbers. This strategy holds true for 6 robbers, 7 robbers… all the way up until a certain number of robbers, where you'd have to split the bars into fractions, which isn't practical to do in a cave. The number of bars you must give up as the highest in experience level can be described as n div 2, so the number of people divided by 2, the quotient. In the case of 5 people, 5 div 2 is 2, and Eric had to give up two bars, one to Adam and one to Chuck. This holds true for all numbers of robbers until n div 2 becomes 100, meaning you have no bars in the end. So, simply solve for n div 2 = 100, and this is when this method no longer works. Solving this would simply be 200 robbers. To live, profit with at least one bar, and be the highest experience level, there must be only 199 robbers max. In regards to the question, when is it better not to be first in experience to make the splitting deal and is it always a better position, it's when the number of robbers exceeds 199, since you won't get any gold bars, and going higher than 200 would leave you dead, since no one will agree to your plan. So, to purpose the optimal splitting method pattern, there must be 199 robbers or less, and it bad to go first anywhere 200 robbers and up (or be anywhere in line above the 199th to last position).