CS4102 - ALGORITHMS - FALL 2019

DUE SATURDAY, OCTOBER 12, 2019 AT 11PM VIA COLLAB

H4:ER6QT

Collaboration Policy: You are encouraged to collaborate with up to 4 other students, but all work submitted must be your own independently written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite them. Do not submit a solution that you are unable to explain orally to a member of the course staff.

Collaborators: zh2yn, dc3jr

Sources: Cormen, et al, Introduction to Algorithms, https://www.geeksforgeeks.org/radix-sort/

PROBLEM 1 *Card Shuffling*

Consider the following approach for shuffling a deck of m cards:

> Apply the following procedure for $i = 1, \ldots, m$:
>
> - Select a card uniformly at random from the deck
> - Swap the i^{th} card in the deck with the randomly-selected card

We say that a deck is *perfectly shuffled* if after applying the shuffling algorithm to any initial configuration of the cards, all possible permutations of the deck are *equally* likely.

1. Show that for all $m > 2$, this shuffling algorithm does not yield a perfectly shuffled deck. (Hint: use decision trees).

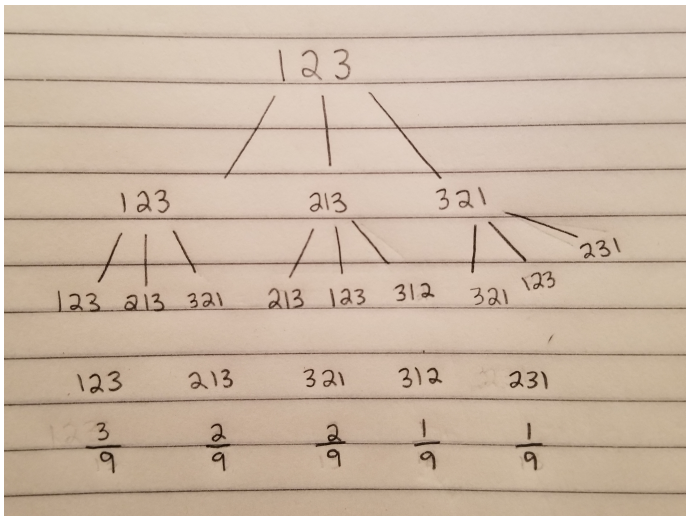

Figure 1: Decision Tree

When $m = 3$, the shuffling algorithm does not yield a perfectly shuffled deck. As can be seen in Figure 1, not all permutations of the deck are equal. With $m = 3$, we have cards 123

at the top. Given that $i = 1$, you first swap the ith(1st) card with the first card, getting 123 at level one. You then swap the first card with the second card to get 213 at level one. Lastly you swap the first card with the third card to get 321 at level one. You redo these steps to get level 2 of the decison tree, where it can be seen that there are unequal permutations, therefore, we don't necessarily have to continue to level three to come to our conclusion.. As shown at the bottom of Figure 1, after applying the shuffling algorithm, the possible permutations of the deck were not all equally likely. The chance of getting the permutation of 123 was $\frac{3}{9}$, the chance of the permutation of 213 and 321 was $\frac{2}{9}$, and the chance of the permutation of 312 and 231 was $\frac{1}{9}$. Therefore, these possible permutations are not equally likely to occur. This proves that for $m > 2$, this shuffling algorithm does not yield a perfectly shuffled deck.

Essentially, there are n! possible permutations. To generalize, using the formula $\frac{m^m}{m!}$ for our case shown in Figure 1, where m^m is the total number of permutations (leaves) and $m!$ is the unique number of permutations, we get $\frac{3^3}{3!}$. This can be simplified to $\frac{27}{6}$, which equals 4.5 occurrences of each permutation. That is impossible. Thus, for all $m > 2$, thus shuffling algorithm does not yield a perfectly shuffled deck.

2. Describe a shuffling algorithm that *perfectly shuffles* any deck of m cards. Your algorithm should run in time $O(m)$. Prove the correctness and running time of your algorithm.

Given a list of cards that are unshuffled, we will select a random position(index) in the list. We will then get the card that is at that randomly selected position and swap it with the last card in the unshuffled list. Therefore, these 2 cards have been shuffled, both the one at that randomly selected position and the last card. To continue shuffling the list of unshuffled cards, we would repeat the process with one less card to shuffle, which can be represented by $m - 1$ cards. We continue doing this algorithm until we have reached the base case of one card remaining as the other cards have been shuffled and sorted, and that one card itself has already been swapped and shuffled as well.

This algorithm can be proven as correct because we make sure that we swap the randomly selected card with the leftmost card at that iteration, and so we efficiently swap the cards, making sure not to mess with them after they have been shuffled. On the other hand, the algorithm in 1.1 did not check for cards that have already been shuffled, ultimately re-shuffling them. This led to the problem of referencing the same permutation multiple times, resulting in the unequal permutation ratios. So in comparison to that algorithm, our algorithm makes sure to randomize and not swap cards that have already been swapped. Hence, in our algorithm, all possible permutations ($m!$) are equally likely. We decrease by one card each time as we go through the algorithm because after the rightmost card has been swapped with the random one, we don't take it into consideration anymore. Therefore, this factorial can be written as such: $m! = m * (m - 1) * (m - 2) * ... * 1$. This results in all possible permutations being equally likely. Repetition of this algorithm will get us to our different permutations.

The runtime of this algorithm is linear $O(m)$ time. This is because each element is accessed only once in order to swap and shuffle any deck of m cards and we pick one card in m total cards m number of times in order to shuffle the deck correctly, and not swap cards that have already been swapped. As mentioned above, we will continue this algorithm until there are no unshuffled cards left in the deck (we move from right to left).

PROBLEM 2 *Goldilocks and the n Bears*

BookWorld needs your help! Literary Detective Thursday Next is investigating the case of the mixed up porridge bowls. Mama and Papa Bear have called her to help "sort out" the mix-up caused by Goldilocks, who mixed up their n bear cubs' bowls of porridge (there are n bear cubs total and n bowls of porridge total). Each bear cub likes his/her porridge at a specific temperature, and thermometers haven't been invented in BookWorld at the time of this case. Since temperature is subjective (without thermometers), we can't ask the bears to compare themselves to one another directly. Similarly, since porridge can't talk, we can't ask the porridge to compare themselves to one another. Therefore, to match up each bear cub with their preferred bowl, Thursday Next must ask the cubs to check a specific bowl of porridge. After tasting a bowl of porridge, the cub will say one of "this porridge is too hot," "this porridge is too cold," or "this porridge is just right."

(a) Give a *deterministic* algorithm for matching up bears with their preferred bowls of porridge which performs $O(n^2)$ total "tastes." Prove that your algorithm is correct and that its running time is $O(n^2)$.

In the deterministic algorithm, you first start by getting a bear cub to taste all the bowls of porridge until it finds one that is "just right." Then, linearly, you repeat this same process for every bear cub until they find their "just right" porridge bowl. Since there are n bear cubs total and n bowls of porridge total, every bear cub will end up finding a bowl of porridge at a specific temperature of his/her liking.

The run time of this algorithm is $O(n^2)$ as in the worst case, it will take the first bear n attempts to find their preferred temperature porridge bowl. Then, the second bear will try finding its preferred porridge bowl by going through the remainder bowls, which takes $n-1$ attempts. With every following bear, there will be one less porridge bowl for them to test, taking one less attempt with each bear, so we shorten by 1 each time. Thus, the total "tastes" for this algorithm can be described as such: $T(n) = n + (n-1) + (n-2) + (n-3) + ... + 1$ (all the way till we reach one remaining bowl). This can be simplified to $T(n) = \frac{n(n+1)}{2}$, which further simplifies to $T(n) = O(n^2)$. Thus, we have proved $O(n^2)$ total "tastes" run time.

(b) Give a *randomized* algorithm which matches bears with their preferred bowls of porridge and performs expected $O(n \log n)$ total "tastes." Prove that your algorithm is correct. Then, intuitively, but precisely, describe why the expected running time of your algorithm is $O(n \log n)$.

In a randomized algorithm, we choose a random bowl of porridge and have each bear cub taste that bowl of porridge. Each bear will then have the option to categorize it as either "too hot", "too cold", or just right. If it's "too hot", you would move the bear cub to the hot category. If it's "too cold", you would move the bear cub to the cold category. If it's "just right", then the bear cub has been satisfied. Then, the bear cub and it's "just right" bowl will act as a pivot to taste every bowl of porridge and judge whether a bowl is too cold or too hot. The cub will then put the porridge bowl in their respective categories of either hot or cold. You recurse this algorithm until you reach your base cases of 1 or 2 remaining bowls, in which every cub will have been matched with their preferred bowls of porridge.

This algorithm can be proven correct because, as with a randomized quicksort algorithm, we start by having a random pivot, which in our case is the bear cub and its random bowl of porridge found to be "just right", that we compare each other bowl

and bear cub by. Each time a cub tastes a bowl, they say whether it's "too hot" or "too cold". This forms the basis of what category a bear cub is placed in. For example, if a bear cub says that the random porridge bowl is "too hot", he or she will be placed in the "cold" category/group. This is because their preferred porridge is colder compared to the one in the random bowl. If a bear says that the random porridge bowl is "too cold", he or she will be placed in the "hot" category/group. This is because their preferred porridge is hotter compared to the one in the random bowl. Then, the one bear cub who found the porridge in the random bowl to be "just right" is used as a pivot to taste every consecutive bowl of porridge and categorize that bowl into the "hot" or "cold" group, respectively. This will correctly place the hot bowls with the bears placed in the hot group, and the cold bowls with the bears placed in the cold group. You keep recursing this algorithm, making a new bear cub a 'pivot' with each recursion (finds the "just right" bowl), until the base cases of 1 or 2 bowls are reached. Therefore, ultimately, each bear cub will find its preferred bowl of porridge.

Note:
Algorithm similar to randomized quicksort, which has an expected running time of $O(nlogn)$

The run time of my algorithm is $O(nlogn)$ total "tastes". Broken down, this is because:

- It takes n steps to have each bear test the random bowl and be categorized in the hot or cold groups or have the bear that says "just"right be matched with the random bowl. Thus is because it requires n number of steps to go through the "list" to get the above mentioned information.

- When we use the satisfied bear with its "just right" bowl as a pivot to separate the porridge bowls into the hot or cold group, this also takes n number of steps to go through the list of bowls.

- These two n steps combined take $2n$ number of steps total: The definition of big-O holds for $c = 2$ and $n_0 = 1$. Thus, $2n = O(n)$.

- When we split the initial list of bears into two sublists/groups, we recursively call our algorithm on each group. This means we do the $O(n)$ number of steps mentioned above $log(n)$ times.

Combining all the aforementioned steps, we get our expected $O(nlogn)$ runtime to perform total "tastes" to match the bears with their preferred bowls of porridge

PROBLEM 3 *Sorting Integers*

(a) Show how to sort n integers in the range 0 to $n^5 - 1$ in $O(n)$ time.

You first start off by converting the integers to base n so that we can later prove/sort in $O(n)$ running time. From there, you need to make sure the integers in question are of the same length. If they are not of the same length, add leading zeros to make them the same length. For example, if you had numbers 512, 181, and 18, you would add a leading zero to 18 to make it "018" to match the length of the other integers. Then, you put the integers into a list from least significant to most significant (stable sort each

digit). Once you do this, you radix sort the list, starting by sorting each element based on their "1's" place. You then use the resulting sorted list to sort each element based on their "10's" place. You then use that resulting sorted list to sort each element based on their "100's" place now. You continue doing this till you've went through all the places.

Radix sort runtime takes $\Theta(d * (n + b))$, where d is the number of digits, b is the base ("radix") and n is the number of values. It takes n number of steps as you convert each integer to base n. As stated in GeeksforGeeks, d would be $O(log_b(k))$ with k being the maximum possible value, which in our case is $n^5 - 1$ since the range is from 0 to $n^5 - 1$. We then set $b = n$ since b is the base for representing numbers and we converted ours to base n. Plugged in with the information we have thus far, we get $log_n(n^5 - 1)$. Using log rule $log_b(b^k) = k$, we can simplify $log_n(n^5 - 1)$ to ~ 5. Using $O(d(n * b))$ where $b = n$, you get $O(5(n + n))$ which simplifies to $O(10n)$. This would be the runtime of the radix sort. Lastly, since we converted the integers to base n, we need to to take n number of steps to convert each integer back to base 10. Thus, our total runtime can be calculated as: $n + 10n + n$ which equals to $12n$, proving that we can sort n integers in $O(n)$ runtime.

(b) Give a linear-time algorithm to sort the ratios of n pairs of integers between 1 and n. Specifically, we need to sort n pairs of the form (a_i, b_i) where $1 \leq a_i \leq n$ and $1 \leq b_i \leq n$ using the sort key $\frac{a_i}{b_i}$ in $O(n)$ time. Prove both the correctness and the running time of your algorithm

One possible linear-time algorithm would be to radix sort the numbers as we did in 3.1. This algorithm, however, will have coordinate pairs as fractions if we apply the sorting key given. Therefore, we need to find another way to turn them into distinct positive integers. The max value is n, therefore, the smallest ratio possible would be $\frac{1}{n}$. In order to get the values to become distinct integers, we would need to the differences between the two smallest ratios to be greater than 1, and then we can truncate the decimal as well. However, if the difference was less than one, there is a possibility that we would end up with the same integer value, not permitting us to properly sort in linear time ($\frac{1}{n-1} - \frac{1}{n}$ must be > 1). If we were to multiply the difference by 2, we would get $\frac{n^2}{n-1} - \frac{n^2}{n}$, which can be further simplified to $\frac{n^2}{n-1} - n$. Now, in order to check that the difference will be greater than one, we can plug in an example number. So, for example, if we plugged in $n = 6$, the result would be $\frac{6^2}{6-1} - 6$, which evaluates to 1.6, which is greater than 1. Seeing this, we need to apply the sorting key to all n pairs and multiply by n^2 to make sure our numbers are distinct integer positives. In order to truncate the decimals, we need to take the floor of the numbers and apply radix sort to them. Thus, in the end of this linear-time algorithm, we will get the desired values by applying (n^2* (the floor of) $\frac{a_i}{b_i}$) to each pair. The runtime for this algorithm is $O(n) + O(n) = O(n)$ as the time is takes to multiply each value takes n time and radix sort takes n time as well.

PROBLEM 4 *Tiling Dominoes, Redux!*

Recall the tiling dominoes problem from lecture, except this time you are given a rectangular grid of size $4 \times n$ with the top right and bottom right corners removed (i.e., there are a total of $4n - 2$ squares). As before, you are also given an endless bag of dominoes, each of size 2×1. Write a dynamic programming algorithm to find the number of *unique* ways to *fully tile* this grid (with the two corners removed). When we say *fully tiled* here, we mean that

every square on the board is covered by dominoes. Prove the correctness and running time of your algorithm.

As a concrete example, when $n = 3$, the solution is 6. We illustrate the board together with the 6 possible tiling configurations in Figure 2 below:

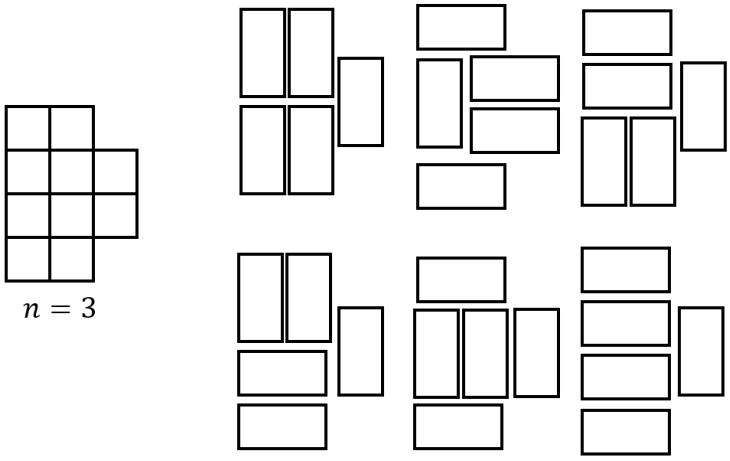

Figure 2: The $n = 3$ board and the 6 ways of tiling the board

We start of by breaking the domino grids into four different shapes. One is $P(n)$, which is considered the big function that has a board with default 4 by n size with four corners removed. Then there is $Q(n)$, which is a perfect rectangle with a size of 4 by n R_1 is a perfect rectangle missing an equivalent of one vertical domino in the top right corner (2 tiles removed) R_2 is a mirror image of R_1, with the bottom right corner missing (2 tiles removed)

Note:
1. n is the number of columns
2. $n - 1$ is the result of 1 less column that we would deal with
3. We start by adding tiles from the rightmost(last) column to the leftmost(first) column
4. The placement of the dominoes yields the cases given below from the shape they form (shapes mentioned above and shown in Figure 3)

Since the example board of size $4n - 2$ resembles that of shape $P(n)$, we can start by using this function to calculate the toal number of ways to tile a board of the said size.

Therefore, we can break $P(n)$ down into two different cases/branches:

1. By adding one domino vertically in the rightmost column, which is missing a top and bottom corner, we get the shape that resembles that of P, so we get $Q(n - 1)$

2. Keeping dynamic programming in mind, we would start by adding two horizontal dominoes to the rightmost column where the corners are missing, and then two horizontal dominoes on the top right and two horizontal dominoes on the bottom right, matching the inner P case (P shape shown in Figure 3).

Using the above cases, we get the equation: $P[n] = P[n-2] + Q[n-1]$

From there, we can break $Q(n)$ into five different cases:

1. For $R_1(n-1)$, we add in two horizontal dominoes in the two top right squares of the grid, and one vertical domino in the two bottom right squares, getting a shape for $R_1(n-1)$.

2. For $R_2(n-1)$, we add in one vertical domino in the two top right squares spaces, and two horizontal dominoes in the four bottom right squares, getting a shape for $R_2(n-1)$.

3. For $Q(n-1)$, we add in two vertical dominoes in the rightmost column, getting a shape for $Q(n-1)$.

4. For $Q(n-2)$, we add in four horizontal dominoes to the two rightmost columns (8 squares), getting the shape for $Q(n-2)$.

5. For $P(n-1)$, we add in one horizontal domino in the two top right squares and one horizontal domino in the two bottom right squares, and one vertical domino in the two middle squares of the the rightmost column, getting the shape for $P(n-1)$.

Using the above cases, we get the equation: $Q[n] = Q[n-2] + Q[n-1] + R_1[n-1] + R_2[n-1] + P[n-1]$

From there, we can break, $R_1(n)$ into two cases:

1. For $R_2(n-1)$, we add two horizontal dominoes to the four bottom rightmost squares, getting the shape of $R_2(n-1)$.

2. For $Q(n-1)$, we add one vertical domino (2 spaces) to the bottom rightmost column where the corner is missing, getting the shape of $Q(n-1)$.

Using the above case, we get the equation: $R_1[n] = Q[n-1] + R_2[n-1]$

From there, we can break, $R_2(n)$ into two cases:

1. For $R_1(n-1)$, we add two horizontal dominoes to the top four rightmost squares, getting the shape of $R_2 = 1(n-1)$.

2. For $Q(n-1)$, we add one vertical domino to the two squares at the top of the rightmost column where the corner is missing, getting the shape of $Q(n-1)$.

Using the above case, we get the equation: $R_2[n] = Q[n-1] + R_2[n-1]$

Discussion:
P, Q, R_1 and R_2 are the four basic shapes we can make when tiling our board of size $4n-2$.

As we tile out boards, we will use the bottom up dynamic programming approach in order to store our values for n at each step of our algorithm, for all shapes P, Q, R_1 and R_2 into arrays of size n, in which we will start using our base cases of $n = 1$ and $n = 2$. For a base case of $n = 0$, the values at that index will equal 0 as they wont necessarily be used when going through our algorithm for our four shapes. This bottom-up dynamic will allow us to solve our problems at the top using our sub problems at the bottom, accessing memory linearly; This will also well help to minimize memory usage as compared to a recursive algorithm. After determining our four possible shapes, as mentioned above, we can determine our base cases to be $n = 1$ and $n = 2$ as these are the smallest possible sizes for n. To describe one of these base cases, when $n = 1$, $P(n), Q(n), R_1(n) and R_2(n)$ all equal 1. This is because there is only one way you can tile a column where n number of columns equals 1. To further elaborate, you can either have one domino fill up two tiles, or have two dominoes fill up 4 tiles vertically in a column.

Note:
The indices $n = [0]$, $n = [1]$ and $n = [2]$ represent the number of columns

In our algorithm, the base cases are as follows:

For $P[0]$ (empty/no column),there's zero solutions, for $P[1]$ (one column), there's only 1 solution, and for $P[2]$ (two columns), there's only 1 solution

For $Q[0]$, there's zero solutions, for $Q[1]$, there's only 1 solution, and for $Q[2]$, there's 5 solutions

For $R_1[0]$, there's zero solutions, for $R_1[1]$, there's only 1 solution, and for $R_1[2]$, there's 2 solutions

For $R_2[0]$, there's zero solutions, for $R_2[1]$, there's only 1 solution, and for $R_2[2]$, there's 2 solutions

Dynamic programming allows for the runtime of this algorithm to be linear $O(n)$. As mentioned before, we used the bottom-up approach to figure out the unique number of ways the board can be fully tiled, so we call $P(n)$. Initially in the algorithm, four arrays are initialized, and each one of these four arrays corresponds to one of the shapes mentioned above: P, Q, R_1, R_2. We have four different for loops that go through the array of solutions for each shape. Therefore, it takes n amount of time for each loop to go through all the values when we call $P(n)$. As mentioned earlier, the bottom-up approach allows us to do this by accessing the memory of the previous solutions, rather than solving it recursively. Hence, the resulting runtime is of $4n$ ($n * 4$, n for each loop). Furthermore, it takes constant time to initialize the values for the base cases. Thus, our algorithm's runtime is $(4n + 1)$, which can be simplified to $O(n)$.

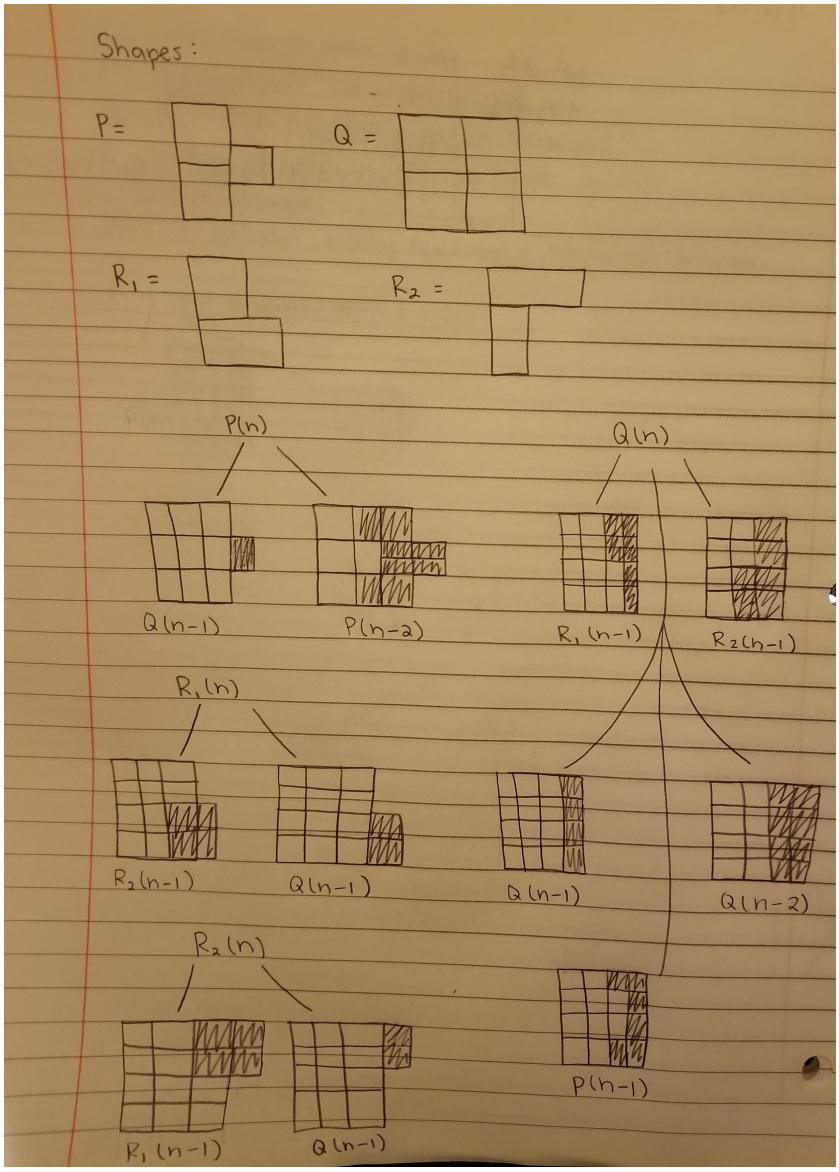

Figure 3: Tiling Board of Four Shapes