CS170–Fall 2015 — Homework 2 Solutions

Nguyet Minh Duong, SID 24444722

Collaborators: Henry Kwan & James Carlson

1.Recurrence Relations

(a) $\Theta(n)$ by using Stronger Master Theorem. By looking at the difference constants and comparing its computated values, we realize that $d = 1 > log_b a = log_4 3$, which by the theorem states that Θ of this case is $\Theta(n)$.

(b) $\Theta(n^{log_3 45})$ by using Stronger Master Theory. This is the case because $d = 3 < log_b a = log_3 45$. And as Stronger Master Theorem has proven, the value of Θ is $\Theta(n^{log_b a})$.

(c)

$$\begin{cases} c < 1 & \Theta(1) \\ c = 1 & \Theta(n) \\ c > 1 & \Theta(c^n) \end{cases}$$

These cases have been proven from Homework 1, problem 4. The reason for this is because $T(n) = T(n-1) + c^n$. This means that we're recursively adding $c^n + c^{n-1} + c^{n-2} + ... + c^1 = \sum_{i=1}^{n} c^i$. So this creates the same problem from the previous homework assignment, where we had to explain why.

(d) So we know that $T(n) = \Theta(2^k)$, where k is what we need to figure out... because we know from $T(n)$ that we will be multiplying 2 in certain amount of time. This equation will have to do a lot with math manipulation. Note that we are looking for what k is.

$2 \geq 2^{(\frac{1}{2})^k log_2 n}$ – the reason for this is because of the condition given at the very beginning where it minimizes out to $n = 2$. And essentially, our entire goal (on the RHS) is simply to see how many square roots we need to do/go through before we reach 2 or greater. It is essentially the math behind it.

$$2 \geq 2^{(\frac{1}{2})^k log_2 n} \tag{1}$$

$$log_2 2 = 1 \geq (\frac{1}{2})^k log_2 n \tag{2}$$

$$2^k \geq log_2 n \tag{3}$$

$$log_2 2^k = k \geq log_2(log_2 n) \tag{4}$$

Now we replace what k is in our original formula above:

$$T(n) = \Theta(2^{log_2(log_2 n)}) = T(n) = \Theta(log_2 n) \tag{5}$$

Hence our run time is $\Theta(log_2 n)$

2. Goldilocks' Problem

Main Idea

We choose a pivot point in the Goldilocks by picking the first Goldilock to compare it to all of the soups. We remember which soup(s) are hot, which soup(s) are cold, and the soup that's just right. We split it into two groups: a hot soup list, and a cold soup list. Then we compare the rest of the Goldilock to the just-right soup, and send all the Goldilocks who found it too hot to the cold soup list, and the ones who found it too cold to the hot soup list. We then recurse the algorithm on the smaller split lists.

Pseudocode

```
Function FindPair(G, S):
  if G.size equals 1:
    return pair(G[0], S[0])
  fG = G[0] and remove G[0] from array G
  let soupC := array of soups that fG finds too cold
  let soupH := array of soups that fG finds too hot
  let rS := soup that fG finds just right and remove it from array S
  let GCold := array of Goldilocks who finds rS too cold
  let GHot := array of Goldilocks who finds rS too hot
  return pair(fG, rS) + FindPair(GCold, soupH) + FindPair(GHot, soupC)
```

Proof of Correctness

Case 1:

If there's only one Goldilock and one soup, by the phrasing of the question, they must be each other's pairs because every Goldilock must have a matching soup.

Case 2:

If there are more than one Goldilock (call her G_1), we can use G_1's preferences on all of the soups as a way to split the list into two. As you know from the description, there is three preference G_1 can take: too hot, too cold, and just right. Only one soup can be just right for G_1, call this S_1. The rest of the soups must either be too hot or too cold. From this, we created two separate list of preference in relation to G_1'. From this, we can compare the rest of the Goldilocks to S_1, and we can see where the rest of the Goldilocks should check for their soups.

Since S_1 is already paired with G_1, all other Goldilocks must find S_1 either too hot or too cold. And so, after comparing the rest of the Goldilocks to S_1, we will have split the rest of the Goldilocks into two new groups. And because the soups are relative to each other, if G_1 finds a soup (call this S_x) too hot, a Goldilock who finds S_1 too hot will definitely find S_x too hot, and vice versa.

Case 2.1:

If a Goldilocks who is not G_1 finds S_1 too hot, she will surely find all soups that G_1 finds too hot, too hot. In this case, this Goldilock should be put in the same comparison list as the soups that are too cold for G_1. Now we cut our search amount some.

Case 2.2:

If a Goldilocks who is not G_1 finds S_1 too cold, she will surely find all soups that G_1 finds too cold, too cold. In this case, this Goldilock should be put in the same

comparison list as the soups that are too hot for G_1. Now we cut our search amount some.

Now we repeat the process on the too cold and too hot list. Eventually we should hit our base case, as we constantly pick one and find its match, then split the list into twos.

Running Time Analysis
$\Theta(nlogn)$

Justification
Note that this is not the actual Θ in the supreme worst case scenerio. However at least 50% of the values being picked can be seen as a good value, as any values between 25^{th} and 75^{th} percentile. Now using the Lemma: *On average a fair coin needs to be tossed two times before a "head" is seen.* The reason we can use this is because a coin's probability on landing as a head is 50%, the same percentage as our percentage of getting a "good" pick. Hence our chances of picking a very low value or a very high value is slim and can be fixed within two picks.

Hence whenever we pick, we should be splitting our array into halves. In this case, it will give us approximately $T(n) = 2T(n/2) + \Theta(n)$. The run is $\Theta(n)$ because we compare twice on every level, but constants are removed when it comes to run times. And by using Master Theorem, we get out run time of the entire algorith to be $Theta(nlogn)$.

3. Stock Market Hindsight

(a) We let $B[1] = 0$ and $B[n] = B[n-1] + A[n-1]$. Then we get array B, such that $B[j] - B[i] = \sum_{k=i}^{j-1} A[k]$. Since we want the profit we make between day_i and day_j, if we have $B[i]$ to be the total amount that could have been collected the first day to day_i, and similarly with $B[j]$ and day_j, if we take the difference between the two values, it would subtract out all the days we did not invest and only give us the sum we've collected between day_i and day_j.

(b) Our algorithm above should take $\Theta(n)$, where n is the size of the array A. The reason for this is because we go through every single value of $A[k]$ one at a time.

The algorithm that we can design to calculate the biggest profit margin can be the same algorithm as the one described in Homework 1, problem 6 – Mario's Workout. As stated in part A, the difference between spot in array B is the profit (or lack of) that can be obtained when we start at a specific spot and wait a positive (to the right) number of days. Now simplifying it, we see that we are trying to find the biggest positive difference between two dates – similar to Mario's Workout but with dates instead of platforms. As a result, seeing the barebone of this problem allows us to apply the same algorithm from Mario's Workout onto this problem.

The run time of our new algorithm will still be $\Theta(n)$, using the run time from last week's correct solution plus the run time for part a. Since the run time will become $\Theta(2n)$, we can ignore the constant.

4. Majority Elements

(a) **Main Idea**

We are going to split the array into twos, until it reaches the base case of either one or two. If it's one, it automatically return that value as the majority and return 1 as how many times it appears. If it is two, it returns the majority correct – meaning null if both values are not the same or the appropriate majority value with 2 as how many times it appears. The program will recursively question which side's appearance value is bigger, if it's equal to each other then the new return values are null or return the appropriate values.

Pseudocode

```
Function Majority(A):
  if A.size = 1:
    return maj := A[0]
    return amt := 1
  L := left half of A
  R := right half of A
  majL := Majority(L).maj
  amtL := Majority(L).amt
  majR := Majority(R).maj
  amtR := Majority(R).amt
  if majL = majR:
    return maj := majR
    return amt := amtR + amtL
  else if amtL > amtR:
    return maj := majL
    return amt := amtL
  else if amtL < amtR:
    return maj := majR
    return amt := amtR
  else if amtL = amtR:
    return null for both values
  else if both values are null:
    return null for both
  else if one of the values is null:
    return maj := the maj that is not null
    return amt := the amt that is not null
```

Proof of Correctness

Case 1.

There is only one value, that value is automatically the majority, and you would return the comparison amount as 1.

Case 2.

There are more than one value, we would choose whichever value has the higher comparison amount because of what a majority is defined to be (to be higher than than 50%). However if both have the same comparison values, it will return null due to the fact that by being equal in amount, it defies the definition of a majority. Otherwise if both are null, there's simply nothing to compare and return null.

This will recurse until it finally return a majority or a null as an answer, where null means there's no majority.

Running Time Analysis
$\Theta(nlogn)$

Justification
Because we split the problem continuously into two's until we reach n = 1, therefore $T(n) = 2T(n/2) + \Theta(n)$. There is a $\Theta(n)$ on every level because we are doing a comparison on every level to see which one has the higher value. Using Master Theorem, the equation above shows that run time is $\Theta(nlogn)$.

(b) **Main Idea**
Everything gets paired off, and then we run through the algorithm canceling out all pairs that do not have the same value. If the two values are the same, we only keep one of the values. Afterwards, we repeat the process until we're down to one value.

Pseudocode

```
Function Majority(A):
  if A.size = 1: return A[0]
  if A.size is odd:
    AStore := A[0] and remove A[0] from A
  ASplit := array of pairs formed from values in A (no duplicates)
  AVals := array of values from ASplit that appeared twice as pairs
  Repeat until AVals.size <= 2:
    ASplit := array of pairs formed from values in A (no duplicates)
    AVals := array of values from ASplit that appeared twice as pairs
  if AStore has value and AVals.size = 0:
    return AStore
  else if AStore has value and AVals.size = 1:
    return AStore if they are equal | null otherwise
  else if Astore has Value and AVals.size = 2:
    return value that appeared twice | null otherwise
  else if AStore has no value and AVals.size = 0:
    return null
  else if AStore has no value and AVals.size = 1:
    return AVals[0]
  else if AStore has no value and AVals.size = 2:
    return null
```

Proof of Correctness
If there is only one value, that has to be the majority. Otherwise if we split it off into pairs and hold the one value as a tie breaker if the array size is odd (since we can't split into all pairs). Then we go through all the pairs and pick out the values that appear twice in a pair. We repeat this process until the array is down to only two or less values. Because majority means it needs to be more than half, having another value makes it half. That's why we ignore the pairs that do not have the same values.

When we are down to two or less values, if they are the same, we return either one. If they are the same and the original array was odd, we used the odd value we stored originally as the tie breaker – which will either return the one that appeared twice otherwise we just return null as there are no majority.

Running Time Analysis
$\Theta(n)$

Justification
The reason for this is we are just continously running over the same array again and again. That makes our time as $T(n) = T(n/2) + \Theta(n)$. Using Master Theorem, our run time is now $\Theta(n)$.

5. Local Maxima in Matrices

The key to this problem is the fact that all values are distinct. What this means is that there must be an absolute highest value in the entire matrix. Along with this, because it is the highest value, all values surrounding it must be smaller than it. Hence it is a local maximum. And by the description of problem, the matrix only has one maximum – meaning that all other values in the matrix will form in such a way that will not create another local maximum. This concept is the same when broken down to an array - in a column.

First, it is key that we use an algorithm in the same way as a binary search (which has the run time of $\Theta(logn)$). We will start with just one column to find its only local maximum. We will start in the middle of the column, and check both sides (can be thought of as branches). If both sides are smaller than the value we checked – it is our local maximum by definition, and it is the only one. Otherwise we continue the algorithm on the side (branch) with the higher value. We know it is not on the other side because it will result in more than one local maximum due to the key fact we came up with the paragraph above – because there's no duplicates, the maximum value will always be the peak. And if the values next to the bigger value gets smaller, that's another peak. Hence why it cannot be correct by the description.

Similarly, we apply the same logic to looking on the left & right of the matrix – using the comparison node as the node in the middle. We will then continue our search on the side with the bigger value, due to the same logic as paragraph 2. As a result, the run time of going through each column is $\Theta(logn)$, and within each column we need to run another $\Theta(logn)$ to search for the maximum there. Hence our total running algorithm time is $\Theta(log^2 n)$.

The proof to binary search has already been solved before.

6. Upper Triangular Matrix Multiplication

In order for his proof to work, all of the sub-matrices must be a triangular matrix, however the sub-matrix B cannot be a triangular matrix as it must be fully filled out in order to make the bigger matrix a triangular matrix. Hence his recursive algorithm does not work.