**Question 1 (a)**

**Algorithm**: GreedyChoice(int C[0…k-1], int U[0…k-1], int k, int N)

**Input**:

* An array C of costs for the k objects
* An array U of utilities of the k objects
* A total budget N

**Output**: An array Q[0…k-1] of quantities to purchase

The code used for sorting in this question is really similar to the sorting code for question 2.

Since there was a lot of code required for writing the sorting algorithm, I have included it here (along with the code to calculate the quantities) to avoid loss of meaning and ambiguity

In this scenario, we have a 2D array whose 1st column refers to the object number, 2nd column to cost, 3rd column for utility and 4th column for cost per utility. We sort the entire 2D array based on sorting the last column which is the cost per utility column. Each row in the array represents a specific object, its cost, utility and cost per utility. We rearrange entire rows so that the rows are sorted by least cost per utility first.

(Most of the code used here is for sorting. The calculation of the quantities themselves didn’t require much code)

*Integer*[] costs = new *Integer*[c.length];

*Integer*[] util = new *Integer*[u.length];

*int*[] q = new *int*[c.length];

*Integer*[][] details = new *Integer*[c.length][4];

*Integer*[] costsPerUtil = new *Integer*[c.length];

for(*int* i = 0; i < costsPerUtil.length; i++) {

costs[i] = *Integer*.valueOf(c[i]);

util[i] = *Integer*.valueOf(u[i]);

/\* we multiply the costs by 100.0 since the

sorting method used below can only sort

integers, not doubles. We multiply by 100.0

so that we have sufficient number of digits

to sort the costPerUtil array \*/

costsPerUtil[i] = (*Integer*) ((*int*)((costs[i] \* 100.0)/util[i]));

}

for(*int* j = 0; j < details[0].length; j++) {

for(*int* i = 0; i < details.length; i++) {

if(j == 0) {

details[i][j] = i;

}

if(j == 1) {

details[i][j] = costs[i];

}

if(j == 2) {

details[i][j] = util[i];

}

if(j == 3) {

details[i][j] = costsPerUtil[i];

}

}

}

*Integer*[][] answer = sort(details);

/\* the method 'sort' sorts the 2D array 'details' based on

its last column which is costPerUtil. all the other

columns also get sorted in such a way so that they

match the CORRESPONDING entries of the last column

from this point lies the code used to calculate

the number of quantities to purchase \*/

for(*int* j = 0; j < q.length; j++) {

*int* quantity = 0;

*int* cost = answer[j][1];

while(cost \* quantity <= n) {

quantity++;

}

if (quantity != 0) {

quantity -= 1;

}

n = n - (*int*) (cost \* quantity);

*int* index = answer[j][0];

q[index] = quantity;

}

return q;

The sort method used above has the following code:

**Algorithm**: sort(Integer[][] details)

**Input**: A 2D array containing object number, cost, utility and cost per utility

**Output**: A 2D array with rows sorted according to least cost per utility first

*int* max = 0;

*Integer*[] costPerUtil = new *Integer*[details.length];

*Integer*[][] arrays = new *Integer*[details.length][details[0].length];

for(*int* i = 0; i < costPerUtil.length; i++) {

*int* num = details[i][details[0].length - 1];

if(max < num) {

max = num;

}

costPerUtil[i] = num;

}

for(*int* j = 0; j < details.length; j++) {

arrays[j] = details[j];

}

*int* maxNumDigits = (*int*) *Math*.log10(max) + 1;

*LinkedList<LinkedList<Integer>>* bucketsForCPerU = new *LinkedList<LinkedList<Integer>>*();

*LinkedList<LinkedList<Integer[]>>* bucketsForArray = new *LinkedList<LinkedList<Integer[]>>*();

for(*int* p = 0; p < 10; p++) {

bucketsForCPerU.add(new *LinkedList<Integer>*());

bucketsForArray.add(new *LinkedList<Integer[]>*());

}

for(*int* j = 1; j <= maxNumDigits; j++) {

for(*int* k = 0; k < costPerUtil.length; k++) {

*int* number = costPerUtil[k];

*int* digit = (*int*) ((number % *Math*.pow(10, j)) / *Math*.pow(10,j-1));

bucketsForCPerU.get(digit).add(number);

bucketsForArray.get(digit).add(arrays[k]);

}

*Integer*[] partiallySorted = new *Integer*[costPerUtil.length];

*Integer*[][] parSorted = new *Integer*[details.length][details[0].length];

*int* numAt = 0;

search:

for(*int* m = 0; m < 10; m++) {

while(!bucketsForCPerU.get(m).isEmpty()) {

partiallySorted[numAt] = bucketsForCPerU.get(m).pollFirst();

parSorted[numAt] = bucketsForArray.get(m).pollFirst();

numAt++;

}

if(numAt == costPerUtil.length) {

break search;

}

}

costPerUtil = partiallySorted;

arrays = parSorted;

}

*Integer*[][] answer = arrays;

return answer;

**Question 2 (a)**

Since the pseudocode might be hard to understand due to a lot of indentation and to avoid any confusions, I am attaching the code that I wrote to sort the array.

Any number can only consist of the digits 0-9 (10 digits). We make 10 ‘buckets’, using a LinkedList of LinkedList of Integers, one for each digit and proceed in the following way:

We first find the maximum number in the given array and find the number of digits in it. That tells us the number of times we have to perform our sorting. We first sort the numbers by their units’ digit and place them in buckets corresponding to their units’ digit. For example, if a number ends with ‘5’ it will be placed in the bucket labeled ‘5’. After this first iteration, we take out the numbers from the buckets from the start of the list to the end of the list, proceeding towards the last bucket.

Then we sort the numbers according to their tens’ digits, hundreds’ digits and so on. After the last iteration, we will obtain our sorted array.

*int* max = 0;

*int*[] copied = new *int*[array.length];

for(*int* i = 0; i < array.length; i++) {

*int* num = array[i];

if(max < num) {

max = num;

}

copied[i] = num;

}

*int* maxNumDigits = (*int*) *Math*.log10(max) + 1;

*LinkedList<LinkedList<Integer>>* buckets = new *LinkedList<LinkedList<Integer>>*();

for(*int* p = 0; p < 10; p++) {

buckets.add(new *LinkedList<Integer>*());

}

for(*int* j = 1; j <= maxNumDigits; j++) {

for(*int* k = 0; k < array.length; k++) {

*int* number = copied[k];

*int* digit = (*int*) ((number % *Math*.pow(10, j)) / *Math*.pow(10,j-1));

buckets.get(digit).add(number);

}

*int*[] partiallySorted = new *int*[array.length];

*int* numAt = 0;

search:

for(*int* m = 0; m < 10; m++) {

while(!buckets.get(m).isEmpty()) {

partiallySorted[numAt] = buckets.get(m).pollFirst();

numAt++;

}

if(numAt == array.length) {

break search;

}

}

copied = partiallySorted;

}

return copied;

**Question 2 (b)**

The running time for the algorithm comes out as:

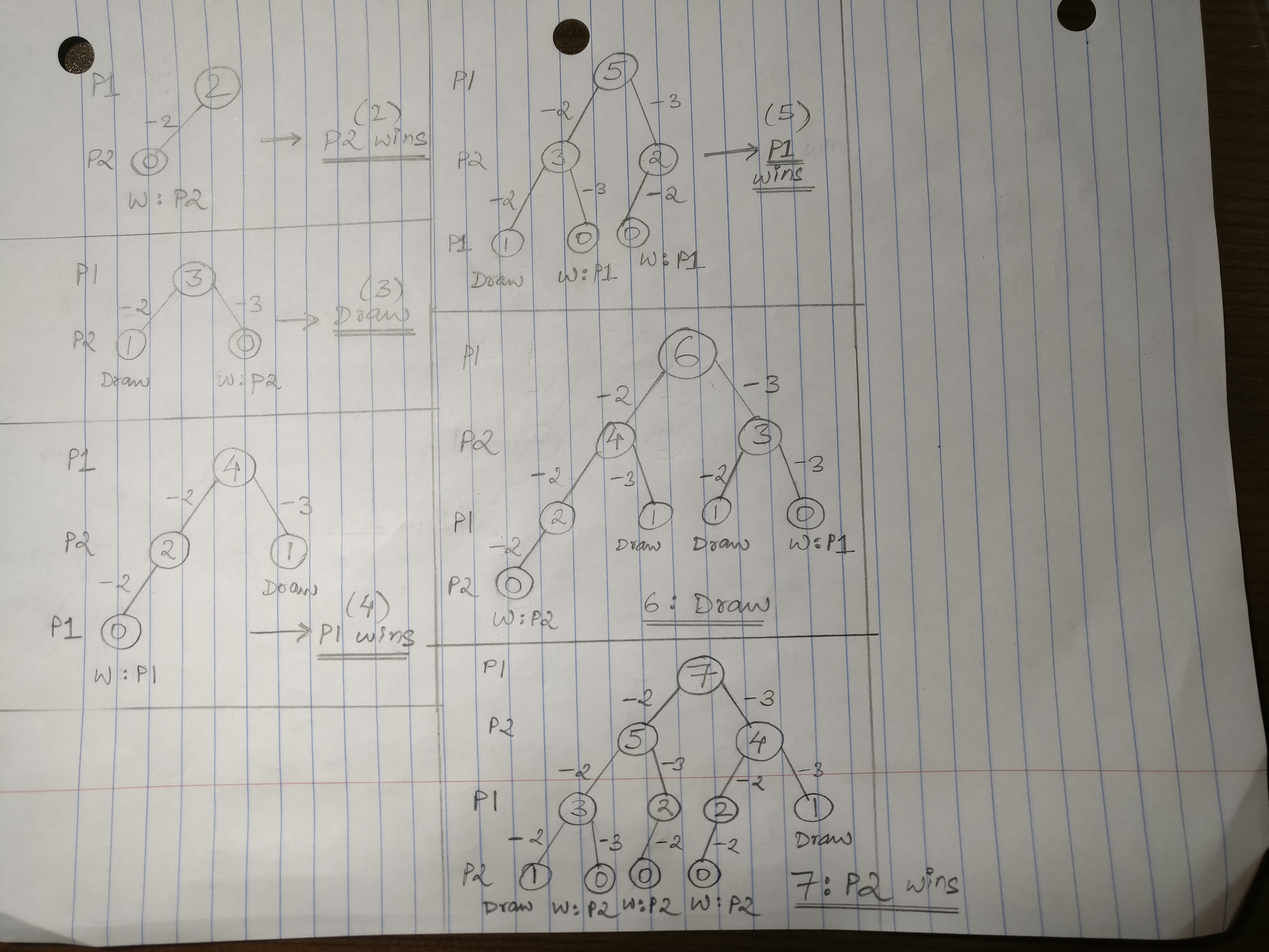
T(n) = 9n + 35 + ( ⎣ log10(max) ⎦ + 1) \* (12n + 176) ------- > O(n)

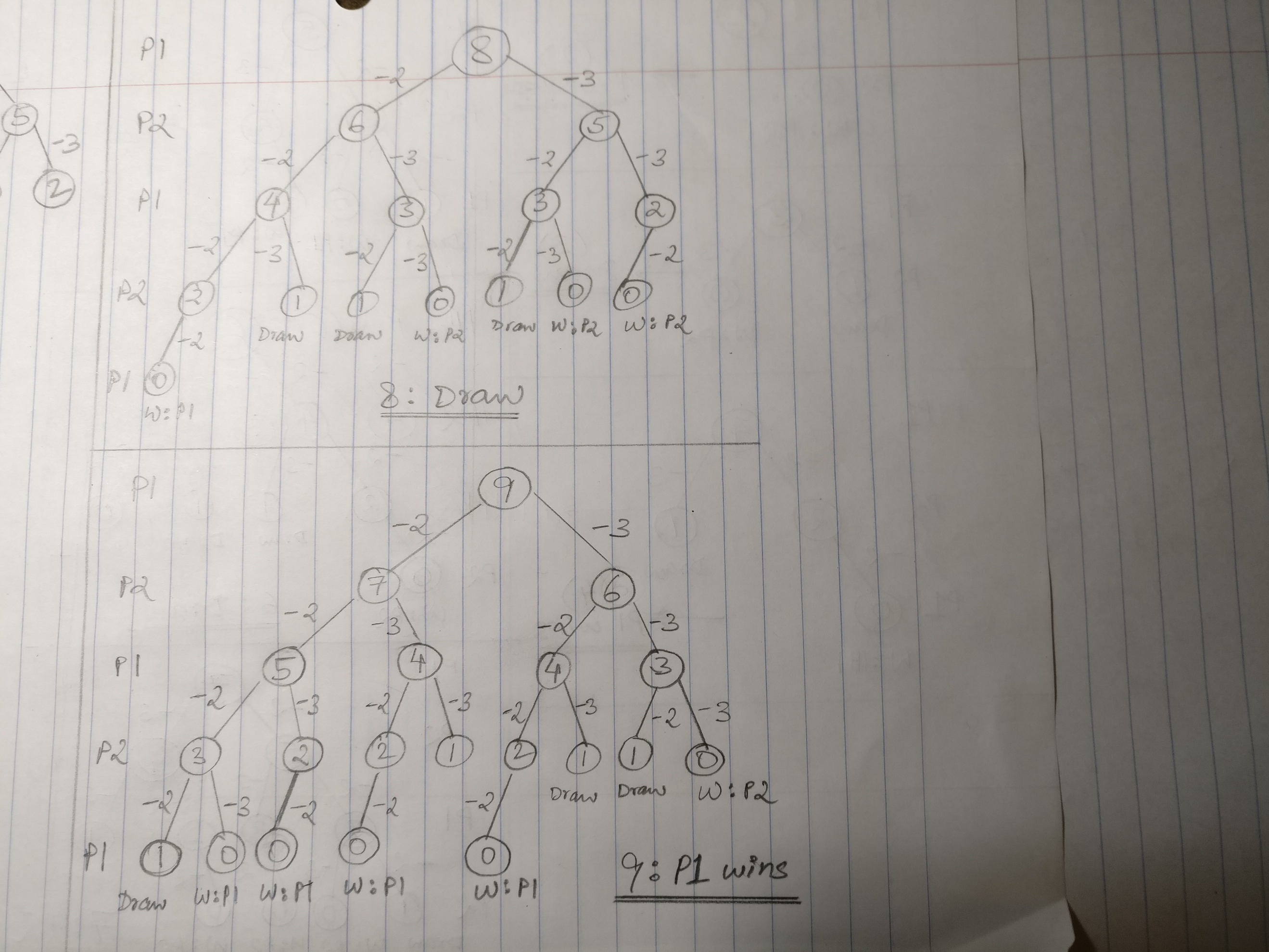
where ⎣ log10(max) ⎦ denotes the floor of log10(max). (Since we are dealing with numbers greater than 1, log will not be negative and so taking the floor is equivalent to casting the value to an integer)

The algorithm cannot be used to sort an **arbitrary** set of numbers because suppose that even though we have a small array (say n = 5 numbers) but the max of those numbers is say 10^64. This would significantly increase the number of steps that we would have to take and the algorithm would waste a lot of time doing extra computations just because we have one number that is super large. This makes the algorithm inefficient for an arbitrary set of integers. This algorithm would be good to use in a situation where the numbers are within some known range

**Question 3 (a)**

The following pictures show the trees if the number of matches were 2,3,4,…9:





As can be seen from the above picture, **P1 would win for 9 matches**.

If we observe carefully, any higher number of matches involves the tree for a lower number of matches. For example, the tree for 6 matches involves the trees for 4 and 3 matches. Since we already know the results of a game starting with 4 and 3 matches with a certain player, we don’t need to calculate the result for these two once again when doing the tree for 6 matches.

The tree for 6 matches could be expressed in a simplified way:

P1 6

/ \

P2 4 3

**P2 wins Draw**

In the case of 4 matches, the player **who starts** wins the game. Therefore, if the game progresses to 4 matches P2 would win. In the case of 3 matches, the game is a draw. Depending on which of the 2 routes P1 goes down either it will end in a draw or P2 will win. Therefore, P1 will move in the direction such that the game ends in a draw. This result is consistent with what we got by drawing out the entire tree for 6 matches.

Applying the same logic for 9 matches:

P1 9

/ \

P2 7 6

**P1 wins Draw**

For 7 matches, using the tree we already have, we know that the player who gets to **play second** wins, therefore, P1 would win if the game progressed to 7 matches and for 6 matches it would be a draw. Therefore, between the two choices of P1’s victory and a draw, P1 goes down its path of victory and therefore **P1 would win the game for 9 matches**

**Question 3 (b)**

Using the above way of simplified trees, we get the following results up to 16 matches (we could go beyond this as well):

Assuming all the games are started by P1 –

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No. of matches | Winner | No. of matches | Winner | No. of matches | Winner |
| 2 | P2 | **7** | P2 | **12** | P2 |
| 3 | Draw | **8** | Draw | **13** | Draw |
| 4 | P1 | **9** | P1 | **14** | P1 |
| 5 | P1 | **10** | P1 | **15** | P1 |
| 6 | Draw | **11** | Draw | **16** | Draw |

It can be clearly seen from the table that the results repeat themselves in cycles of 5. This would continue for further numbers as well and so we can generalize this and find the winner for ‘n’ number of matches.

|  |  |
| --- | --- |
| No. of matches | Winner |
| If n is of the form (5k – 3) | P2 |
| If n is of the form (5k – 2) | Draw |
| If n is of the form (5k – 1) | P1 |
| If n is of the form (5k) | P1 |
| If n is of the form (5k + 1) | Draw |

Where k is a natural number

**Question 4 (a)**

**Algorithm**: eccentricity(vertex u)

**Input**: a vertex u from the graph

**Output**: the eccentricity of u

q ← new Queue()

setVisited(u, true)

setDistance(u, 0)

q.enqueue(u)

eccentricity ← 0

while(!q.empty()) do

w ← q.deque()

eccentricity ← getDistance(w)

for all v ∈ getNeighbors(w) do

if (!getVisited(v)) then

setVisited(v, true)

setDistance(v, getDistance(w) + 1)

q.enqueue(v)

return eccentricity

**Question 4 (b)**

**Algorithm**: is2colorable(vertex u)

**Input**: a graph vertex u

**Output**: true if the graph to which u belongs is 2-colorable, and false otherwise

q ← new Queue()

setVisited(u, true)

setColor(u, 0)

q.enqueue(u)

while(!q.empty()) do

w ← q.deque()

for all v ∈ getNeighbors(w) do

if(!getVisited(v)) then

setVisited(v, true)

setColor(v, 1 – getColor(w))

q.enqueue(v)

else

if(getColor(v) == getColor(w)) then

return false

return true