**Question 2 (a)**

Since the pseudocode might be hard to understand due to a lot of indentation and such, I am attaching the code that I wrote to sort the 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:

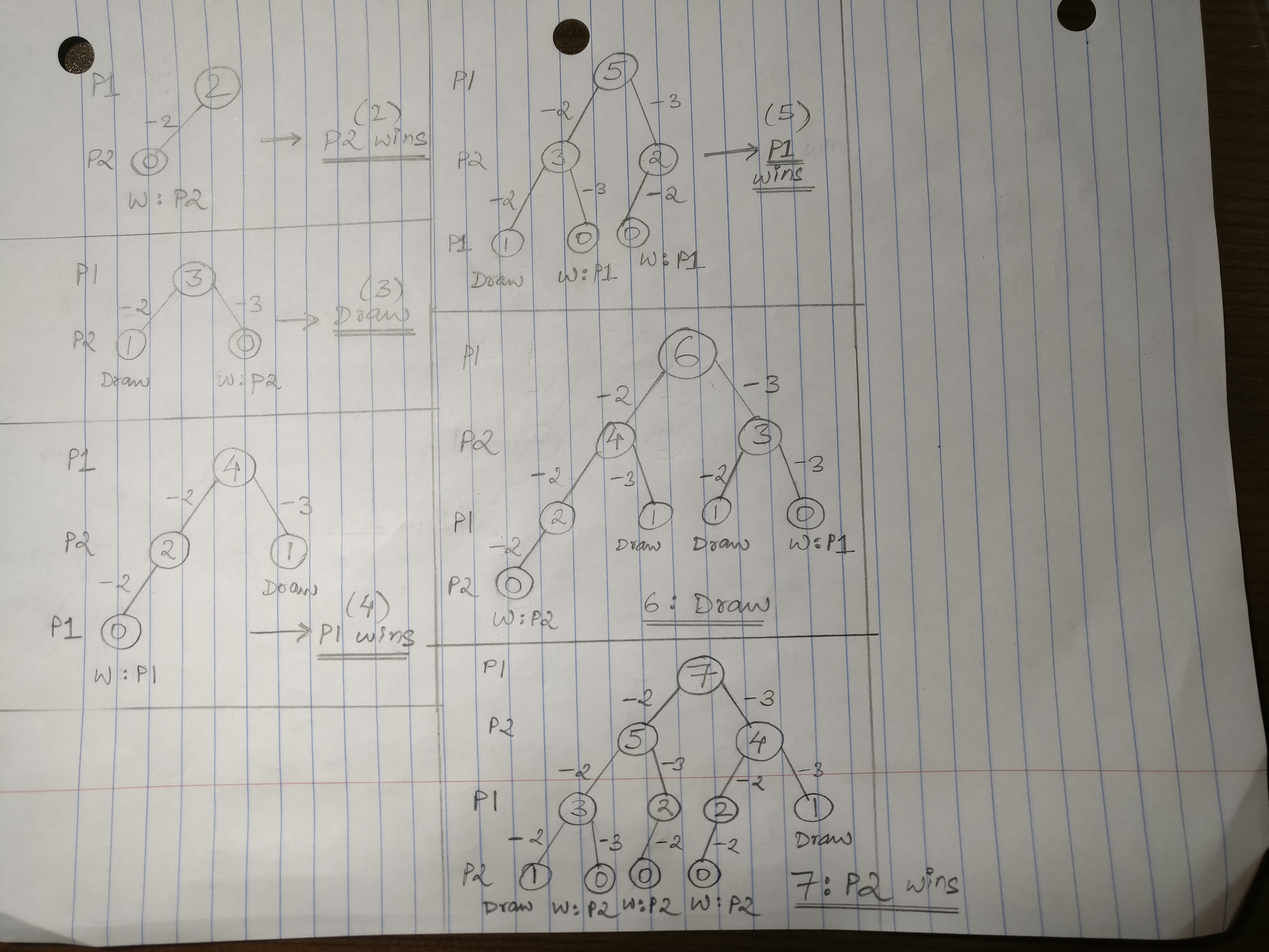
T(n) = 9n + 35 + ( ⎣ log10(max) ⎦ + 1) \* (12n + 176) ------- > O(n)

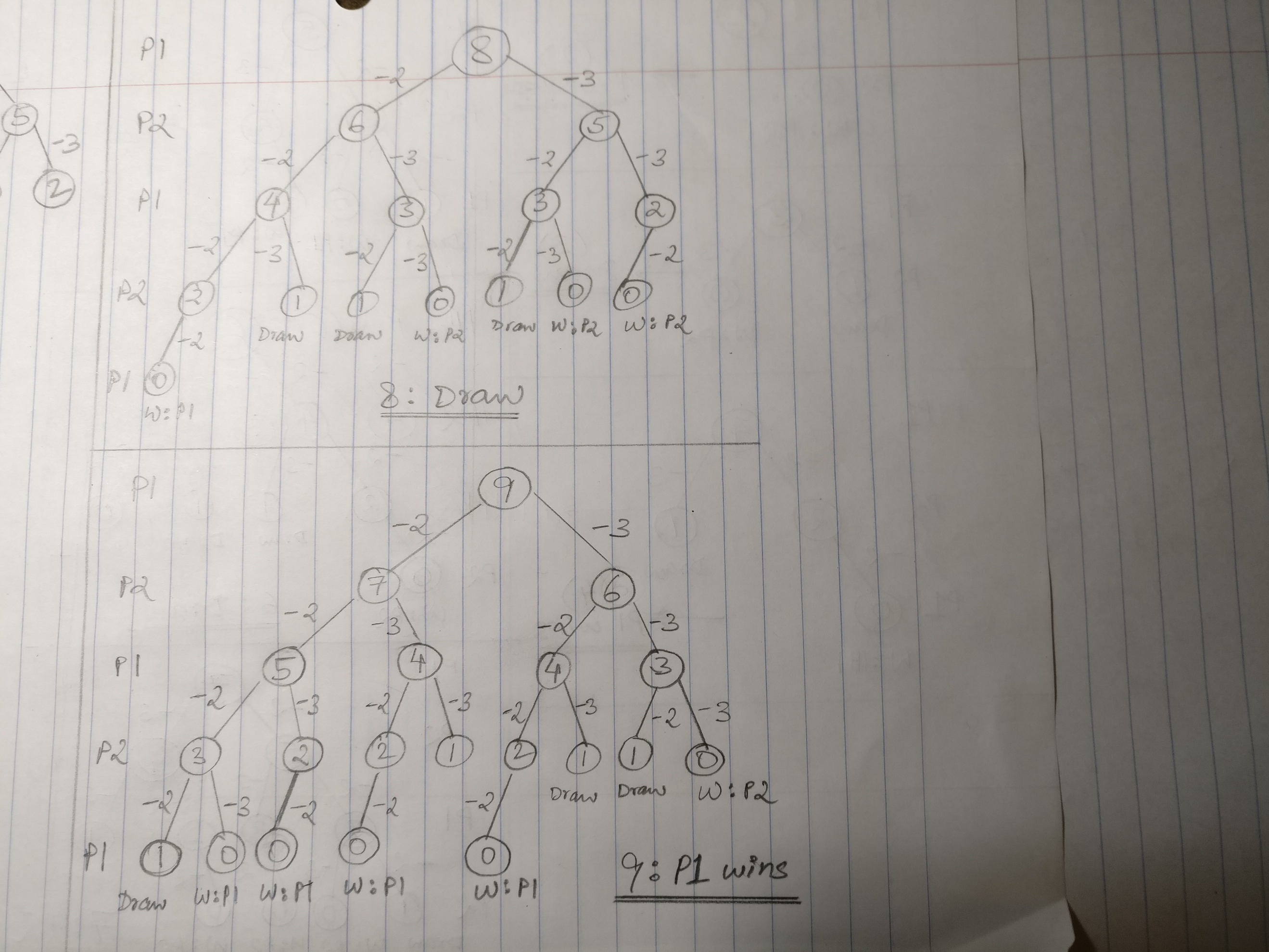
where ⎣ log10(max) ⎦ denotes the floor of log10(max). (Since we are dealing with numbers greater than 1, log will not be negative and so taking the floor is equivalent to casting the value to an integer)

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