COL351 Assignment 3

Garvit Dhawan, Pratik Nimbalkar

TOTAL POINTS

52 / 60

QUESTION 1

1 Particle interaction **15 / 15**

✓ **+ 15 pts** *Correct Solution*

QUESTION 2

2 Non-dominated points **15 / 23**

✓ **+ 8 pts** *Only Correct Algorithm*

✓ **+ 3 pts** *Correct reasoning for time complexity.*

+ 4 *Point adjustment*

+3 partial proof, -3 for incorrect steps in algo.

1 Sequence of steps are not correct.

2 Not a proper proof. Read the rubrics item to see what you missed (partial marks added)

QUESTION 3

Majority 22 pts

3.1 part a **10 / 10**

✓ **+ 10 pts** *Correct*

3.2 part b **12 / 12**

✓ **+ 12 pts** *Correct*

COL351 Assignment 3

Garvit Dhawan (2020CS50425)

Pratik Nimbalkar (2020CS10607)

1. Particle Interaction

According to the problem, we have to calculate the Coulomb force Fj on each particle due to other particles in nlogn time. The Coulomb Force expression on particle j, Fj is given by the following expression:

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}$$

The above expression can be obtained from the polynomial multiplication of two polynomials P(x) and Q(x) where the coefficient vectors of P(x) and Q(x) are as follows:

$P = (q_1, q_2, \ldots . q_n)$

Q = (-1/(n-1)^2, -1/(n-2)^2,..... -1/9, - 1/4, -1, 0, 1, 1/4 , 1/9, ….., 1/(n-2)^2, 1/(n-1)^2)

These vectors when written in form of polynomials is given by:

P(x) = $\sum_{0}^{n-1} q_{i+1}x^i$

Q(x) = $\sum_{i=0, i\neq n-1}^{2n-2} \frac{|i-(n-1)|}{(i-(n-1))^3}x^i$

and the coefficient of x^{n-1} is 0.

We know that the convolution of 2 vectors is given as follows:
$(x * y)(\tau) = \Sigma\ (i = -\infty \text{ to } +\infty)\ [x(i)\ y(\tau - i)]$
We can also find Fj in terms of P and Q algebraically.

To calculate the Fj terms for all particles, we can use FFT i.e. Fast Fourier Transform method. This is given by:

Algorithm:

1. We calculate the Fourier Transform for array P and Q as discussed in class in O(nlogn) time.
2. We know that convolution in the frequency domain is equivalent to multiplication in the time domain. Hence we calculate the product of FT(P) and FT(Q) in O(n) time. Basically we are writing the convolution as a product in the frequency transform domain.
3. Now for calculating the convolution of P and Q, we calculate the Inverse Fourier Transform of the product. This step can be completed in O(nlogn) time.
4. We can obtain every term using the expression of F(j) from the convolution. This step takes O(n) time.

Overall Time complexity of the algorithm becomes O(nlogn) + O(n) + O(nlogn) + O(n) which is O(nlogn).

Explanation:

Computing coefficients of polynomials P(x) and Q(x) takes O(n) time.
Computing product of P(x) and Q(x) takes O(nlogn) time using FFT and Inverse FFT.
Lastly, computing forces Fj will take O(n) time.
Hence total time complexity is O(nlogn).

Correctness:

Given: Two polynomials $A(x) = a_0 + a_1x + \cdots + a_nx^n$ and $B(x) = b_0 + b_1x + \cdots + b_nx^n$, with degree less than equal to 'n' and integer coefficients.

Find: Product $A(x) \cdot B(x) = c_0 + c_1x + c_2x^2 + \cdots + c_{2n}x^{2n}$ (Say, $C(x)$)

Definition: Convolution

Vector $[c_0, c_1, c_2, \ldots, c_{2n}]$ is referred as "Convolution" of vectors $[a_0, a_1, \ldots, a_n]$ and $[b_0, b_1, \ldots, b_n]$.

$$c_i = (a_0\ b_i) + (a_1\ b_{i-1}) + (a_2\ b_{i-2}) + \cdots + (a_i\ b_0) = \sum_{j=0}^{i} a_j b_{i-j}$$

From this class slide, we know that the coefficient vector of the resultant polynomial is the convolution of the coefficient vectors of the multiplicants.
We define polynomial R(x) = C * P(x) * Q(x)
Where C is the constant that appears in Coulomb law expression

The coefficient vector of R(x) is the convolution of the coefficient vectors of P(x) and Q(x) multiplied by C.
Hence we can prove the correctness of the algorithm by using polynomial multiplication of P(x) and Q(x).
Basically we write this polynomial as

$$R[i]x^i = C\sum_j P[j]x^j Q[i-j]x^{i-j}$$

We can find the required coulomb force as follows:
$F[i]x^{(i+n-2)} = Cq_i R[i+n-2]$

⇒

$$\sum_{j\neq i-1} q_{j+1}x^j \frac{|((i+n-2)-j)-(n-1)|}{(((i+n-2)-j)-(n-1))^3)}x^{i+n-2-j}$$

Computing the term inside the summation ⇒
$q_{j+1}\frac{|i-j-1|}{(i-j-1)^3}x^{i+n-2}$

We can replace j by j-1 while changing the summation bounds:
⇒

$$R[i+n-2]x^{i+n-2} = C\sum_{j\neq i} q_j \frac{|i-j|}{(i-j)^3}x^{(i+n-2)}$$

This can also be written as:
⇒

$$R[j+n-2]x^{j+n-2} = C\sum_{i\neq j} q_i \frac{|j-i|}{(j-i)^3}x^{j+n-2}$$

…. Interchanging i and j

The summation can be broken into 2 parts

1. With i<j
2. With i>j

⇒

$$R[j+n-2]x^{(j+n-2)} = C\sum_{i<j} q_i \frac{|j-i|}{(j-i)^3}x^{j+n-2} + C\sum_{i>j} q_i \frac{|j-i|}{(j-i)^3}x^{j+n-2}$$

$$= C\sum_{i<j} \frac{q_i}{(j-i)^2}x^{j+n-2} - C\sum_{i>j} \frac{q_i}{(j-i)^2}x^{j+n-2}$$

Now we can write R[j+n-2] x^(j+n-2) as (Fj / qj)*x^(j+n-2)

Hence

$$\frac{F_j}{q_j}x^{j+n-2} = C\sum_{i<j} \frac{q_i}{(j-i)^2}x^{j+n-2} - C\sum_{i>j} \frac{q_i}{(j-i)^2}x^{j+n-2}$$

Therefore

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}$$

Hence we proved that the convolution of P and Q will contain an entry of the form:

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}$$

and it will be the coefficient of x^(i+n-2) in the product of P and Q

2. Non dominated points

We say that a point pi dominates another point pj if xi > xj and yi > yj . A point is said to be non-dominated if there is no point in P which

1 Particle interaction **15 / 15**

✓ **+ 15 pts** *Correct Solution*

The summation can be broken into 2 parts

1. With i<j
2. With i>j

⇒

$$R[j+n-2]x^{(j+n-2)} = C\sum_{i<j} q_i \frac{|j-i|}{(j-i)^3}x^{j+n-2} + C\sum_{i>j} q_i \frac{|j-i|}{(j-i)^3}x^{j+n-2}$$

$$= C\sum_{i<j} \frac{q_i}{(j-i)^2}x^{j+n-2} - C\sum_{i>j} \frac{q_i}{(j-i)^2}x^{j+n-2}$$

Now we can write R[j+n-2] x^(j+n-2) as (Fj / qj)*x^(j+n-2)

Hence

$$\frac{F_j}{q_j}x^{j+n-2} = C\sum_{i<j} \frac{q_i}{(j-i)^2}x^{j+n-2} - C\sum_{i>j} \frac{q_i}{(j-i)^2}x^{j+n-2}$$

Therefore

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}$$

Hence we proved that the convolution of P and Q will contain an entry of the form:

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}$$

and it will be the coefficient of x^(i+n-2) in the product of P and Q

2. Non dominated points

We say that a point pi dominates another point pj if xi > xj and yi > yj . A point is said to be non-dominated if there is no point in P which

dominates it. Here P = {pi = (xi , yi) | 1 ⩽ i ⩽ n}. Also, we assume that all the x-coordinate and y-coordinate values are distinct.

We have to design a divide and conquer strategy to compute the set of ALL non-dominated points in set P in O(n log n) time.
The pseudo code for the algorithm which we use to find the set of all non-dominated points is as follows:

Function to find NonDominated(P)
1 if P = ∅, return ∅
2 if |P| = 1, return P
3 Find the median x-coordinate of points in P. Let this be Mx.
4 Let L be the set of all the points having x-coordinate ≤ Mx.
5 Let R be the set of all the points having x-coordinate > Mx.
6 Find the point in R which has the maximum y coordinate. Let it be 'b'.
7 Delete 'b' from R.
8 Delete every point dominated by 'b' in L and in R.
9 Let S(L) =NonDominated(L) and S(R) = NonDominated(R)
10 Return S(L) ∪ S(R) ∪ {'b'}

Correctness and Explanation:
Here we are defining a recursive divide and conquer strategy to find the Set of all non dominated points of given set P. When P is empty there will obviously be no non dominated or dominated points.
When it has a single element, then the non dominated points will be 1. This point is not dominated since it cannot follow the condition that both its coordinates are > than coordinates of any other point as there is no other point.

We divide the region of points given by P in 2 halves (L and R) with respect to their median Mx.
We find and delete the maximum y coordinate point in R. Then we delete every point dominated by 'b' in L and R.
Then we recurse on L and R and combine the results.

We prove our algorithm correctness by induction.

Base case: The number of points in P= 1. In this case, the number of non- dominated points will be 1 as explained above.

Induction Hypothesis:
If size of array < N, then our algorithm gives the correct answer.

Induction Step:
Given that our algorithm is correct for array size <N, we need to prove that it is correct for array size = N.
For array size = N, we will divide the points according to the median which will have two parts L and R each of size N/2 ((N+1)/2, (N-1)/2 [if N is odd]). Then we apply our algorithm recursively on L and R.

We can say that no dominated points remain in the solution.

This is because by induction hypothesis, S(L) and S(R) will give the correct answer (since each is of size $\lfloor N/2 \rfloor$ which is < N).

Therefore, S(L) and S(R) would have no dominated points in themselves (and the points which are dominated in L and R would also be dominated in P), and there will not be any dominated points in the final solution also, because any points in L which were dominated by any point in R were already removed (since 'b' was the point with highest y-coordinate, if a point in L is dominated by any point in R, it would also be dominated by 'b' also) and no point in R can be dominated by any point in L since all points in L have x-coordinate less than all points in R.

We can also say that no non-dominated point is lost in our algorithm.

This is because 'b' is included in our solution and we only removed points which were dominated by 'b'.
And, S(L) and S(R) would also contain all remaining non-dominated points of P, since if a point is non-dominated in P, it would also be non-dominated in L and R, and by induction hypothesis S(L) and S(R) would not lose any non-dominated point.
Hence, the solution is correct.

Time Complexity:

Our algorithm divides the problem into two subproblems of size $\lfloor N/2 \rfloor$ and takes linear time to calculate median (as discussed in class), as well as linear time to find 'b' and to remove points dominated by 'b'.

Therefore,

$T(N) = 2*T(N/2) + O(N)$
Which we know that comes to be $T(N) = O(N*\log(N))$

3.Majority

a)
An array A is said to have a majority element if more than half of its entries are the same. We have to check whether there is any majority element present or not in the array and output it if there exists a majority element in O(nlogn) time.
We solve this problem using Divide and Conquer strategy. To find the majority element of an array A of size n we split it into two arrays A1 and A2 each of size $\lfloor N/2 \rfloor$. Knowing whether any majority element is present in A1 and A2 and if yes, what is the majority element will help us to compute the majority element of A.

Algorithm:
If the number of elements in array A = 0, then no majority element.
If the number of elements in array A = 1, then that particular element of A is the majority element.
Else
We divide the array A into 2 parts A1 and A2. We check for the majority element in A1 and A2. We will recursively apply this algorithm on A1 and A2 to further break it into multiple parts and then compile the smaller parts to give the solution for larger parts. While compiling the smaller parts solutions, there are 5 possible different cases as mentioned below to find the solution for the combined larger part. Here smaller parts are basically the arrays obtained when a larger part is divided into 2 in our algorithm.

After dividing the array A into 2 parts:

i. A1 has a majority element M1, A2 has a majority element M2 and both are same (M1 = M2 = M): In this case we can see that the majority element of A will also be M.

Proof: Since M is the majority element of A1, number of times it would occur in A1>(size of A1)/2 i.e. > (N/2)/2 i.e. > N/4
Similarly, the number of times M would occur in A2 > N/4

2 Non-dominated points 15 / 23

✓ **+ 8 pts** *Only Correct Algorithm*

✓ **+ 3 pts** *Correct reasoning for time complexity.*

+ 4 *Point adjustment*

+3 partial proof, -3 for incorrect steps in algo.

1. Sequence of steps are not correct.

2. Not a proper proof. Read the rubrics item to see what you missed (partial marks added)

Therefore,

$T(N) = 2*T(N/2) + O(N)$
Which we know that comes to be $T(N) = O(N*\log(N))$

3.Majority

a)
An array A is said to have a majority element if more than half of its entries are the same. We have to check whether there is any majority element present or not in the array and output it if there exists a majority element in O(nlogn) time.
We solve this problem using Divide and Conquer strategy. To find the majority element of an array A of size n we split it into two arrays A1 and A2 each of size $\lfloor N/2 \rfloor$. Knowing whether any majority element is present in A1 and A2 and if yes, what is the majority element will help us to compute the majority element of A.

Algorithm:
If the number of elements in array A = 0, then no majority element.
If the number of elements in array A = 1, then that particular element of A is the majority element.
Else
We divide the array A into 2 parts A1 and A2. We check for the majority element in A1 and A2. We will recursively apply this algorithm on A1 and A2 to further break it into multiple parts and then compile the smaller parts to give the solution for larger parts. While compiling the smaller parts solutions, there are 5 possible different cases as mentioned below to find the solution for the combined larger part. Here smaller parts are basically the arrays obtained when a larger part is divided into 2 in our algorithm.

After dividing the array A into 2 parts:

i. A1 has a majority element M1, A2 has a majority element M2 and both are same (M1 = M2 = M): In this case we can see that the majority element of A will also be M.

Proof: Since M is the majority element of A1, number of times it would occur in A1>(size of A1)/2 i.e. > (N/2)/2 i.e. > N/4
Similarly, the number of times M would occur in A2 > N/4

Hence the number of times it would occur in A > N/4 + N/4 i.e. >N/2. Thus by definition, we can say that there exists a majority element of array A which is M.

ii. A1 does not have a majority element, A2 also does not have any majority element: In this case, there would not be any majority element of A.
Proof: Since there does not exist any majority element of A1, the element of A1 which will occur the maximum number of times would be at most equal to (size of A1)/2 i.e. N/4. Because if it occurs more than N/4 times in A1, it would be the majority element of A1 which is not the case. Similarly the element which occurs the maximum number of times in A2 will at most occur N/4 times. In the worst case scenario, these 2 maximum occurring elements of A1 and A2 will be the same and would occur at most N/4 + N/4 i.e. N/2 times in A.
Thus, there would not be any element occurring > N/2 times in A. Hence A does not have any majority element.

iii. A1 has a majority element M1, A2 has a majority element M2, both are different: There can be 3 answers possible in this case

1. Majority element in A is M1 (if (number of times M1 occuring in A1) + (number of M1s present in A2) > N/2)
2. Majority element in A is M2 (if (number of times M2 occuring in A2) + (number of M2s present in A2) > N/2)
3. No majority element exists in A (otherwise)

Proof:
The number of times M1 occurs in A1 > N/4 and the number of times M2 occurs in A2 is > N/4.

a. Let the number of M1s present in A2 be x

Now, if (number of times M1 occuring in A1) + x > N/2, we can say that the majority element of A is M1
If (number of times M1 occurs in A1) + x <= N/2 we move to the next step.

b. Let the number of M2s present in A2 be y

Now, if (number of times M2 occurs in A2) + y > N/2, we can say that the majority element of A is M2.

If (number of times M2 occurs in A2) + y <= N/2 we move to the next step.

c. In this case, A would not have any majority element since the sum of number of times M1 occurring in A1 and x is <= N/2 and also the sum of number of times M2 occurring in A2 and y is <= N/2
   Also we can claim that there would not be any other element who is occurring > N/2 times in A. For this to take place, it should appear more than N/4 times in at least A1 or A2 so that its sum with occurrences in A2 or A1 respectively can be > N/2. And the only element occurring > N/4 times in A1 is M1 and >N/4 times in A2 is M2, both these cases have been covered by us above. Hence no majority element in A.

iv. A1 has a majority element M1, A2 does not have any majority element. In this case, either the majority element of A would be M1 (((number of times M1 occurs in A1)+ (number of times M1 occurs in A2)) > N/2) or there would be no majority element (otherwise)
Proof:
The number of times M1 occurs in A1 > N/4. Let the number of times it appears in A2 be x. If (number of times M1 occurs in A1) + x > N/2, then the majority element of A would be M1.
Else if the (number of times M1 appears in A1) + x < N/2, then there won't be any majority element of A (by definition).
Also, we can say that there would not be any other possible Solution as the majority element of A. This is because, if there is any majority element M (let's assume) then it must be present > N/4 times in at least one of A1 and A2 (by pigeon-hole principle) If it is present > N/4 times in A1, it would become the majority element of A1 i.e. M1 (this case is already covered) and if M is present > N/4 times in A2, it would be majority element of A2 which is contradicting our case assumption that A2 has no majority element.

v. A2 has a majority element M2, A1 does not have any majority element. In this case, either the majority element of A would be M2 (((number of times M2 occurs in A2)+ (number of times M2 occurs in A1)) > N/2) or there would be no majority element

(otherwise)

Proof:
The number of times M2 occurs in A2 > N/4. Let the number of times it appears in A1 be x. If (number of times M2 occurs in A2) + x > N/2, then the majority element of A would be M2.
Else if the (number of times M2 appears in A2) + x < N/2, then there won't be any majority element of A (by definition).
Also, we can say that there would not be any other possible Solution as the majority element of A. This is because, if there is any majority element M (let's assume) then it must be present > N/4 times in at least one of A1 and A2 (by pigeon-hole principle) If it is present > N/4 times in A2, it would become the majority element of A2 i.e. M2 (this case is already covered) and if M is present > N/4 times in A1, it would be majority element of A1 which is contradicting our case assumption that A1 has no majority element.

These 5 cases are the only possibilities because A1, A2 either do have majority element or they do not have, if they have any majority element, it can either be same or different. Our cases includes all these possibilities.

During the compilation, we will have to perform maximum linear order time operations in the 5 cases:
i. Both same case: Just need to check whether M1= M2 or not (constant time)
ii. No majority in both: No extra operations
iii. Both have different majority elements case: Firstly we will compare M1 and M2 (constant time). Then for part a, we need to perform O(n) operations to check number of M1s in A1 as well as A2, for part b, O(n) operations to check no. of M2s in A1as well as in A2, part c, no extra operations. Hence total O(n) order extra operations.
iv. Only one has a majority element: We check Number of M1s present in A2. This can be performed in O(n) time.
v. Only one has a majority element: We check Number of M2s present in A1. This can be performed in O(n) time.

Time Complexity Analysis:
Let T(n) be the required time to find the majority element in an array of size n.

Therefore the time required to find the majority element in an array of size n/2 will be T(n/2).
After dividing the array into 2 parts, we are performing at maximum O(n) time operations as written above.
Hence T(n) = 2*T(n/2) + O(n)
Which we know after solving results in T(n) = O(nlogn).

b)
Now, we need to design a linear-time algorithm for this problem.

Algorithm:
(i) Pair up the elements of A to get n/2 pairs (where n is size of array):

a. If n is even: We will get perfect n/2 pairs. We divide the n elements into n/2 pairs as follows: Take 1st element, 2nd element and make a pair (pair1), then take 3rd element and 4th element and make a pair (pair2) and so on. The last pair (pair n/2) would consist of (n-1)th and nth elements.
b. If n is odd: We will get (n-1)/2 pairs and an extra element. We divide as follows: Take 1st element, 2nd element and make a pair (pair1), then take 3rd element and 4th element and make a pair (pair2) and so on. The last pair (pair n/2) would consist of (n-2)th and (n-1)th element. Thus total (n-1)/2 pairs and 1 unpaired element (nth).

If n is odd, then we check whether the unpaired element occurs >n/2 times in O(n) time. If yes, then we return it. Else we discard it and apply the below mentioned algorithm on remaining (n-1)/2 pairs.

(ii) Look at each pair: if the two elements are different, discard both; if they are the same, keep one of them:
We define a new array A'. For both even and odd n, we look at all the different pairs formed and check whether both elements of each pair are the same or not. If they are the same, we add that element (only once for a pair) to A'. If different, we will not add any element to A'.
Then we recursively apply this algorithm on A', i.e. again divide it into pairs.

(iii) If at any step while discarding the pairs which do not have same elements, we discard all (no elements remain), then we conclude that there is no majority element present. If we finally get a single element at last, it will be the majority element of our array A.

3.1 part a **10 / 10**

✓ **+ 10 pts** *Correct*

Therefore the time required to find the majority element in an array of size n/2 will be T(n/2).
After dividing the array into 2 parts, we are performing at maximum O(n) time operations as written above.
Hence T(n) = 2*T(n/2) + O(n)
Which we know after solving results in T(n) = O(nlogn).

b)
Now, we need to design a linear-time algorithm for this problem.

Algorithm:
(i) Pair up the elements of A to get n/2 pairs (where n is size of array):

a. If n is even: We will get perfect n/2 pairs. We divide the n elements into n/2 pairs as follows: Take 1st element, 2nd element and make a pair (pair1), then take 3rd element and 4th element and make a pair (pair2) and so on. The last pair (pair n/2) would consist of (n-1)th and nth elements.
b. If n is odd: We will get (n-1)/2 pairs and an extra element. We divide as follows: Take 1st element, 2nd element and make a pair (pair1), then take 3rd element and 4th element and make a pair (pair2) and so on. The last pair (pair n/2) would consist of (n-2)th and (n-1)th element. Thus total (n-1)/2 pairs and 1 unpaired element (nth).

If n is odd, then we check whether the unpaired element occurs >n/2 times in O(n) time. If yes, then we return it. Else we discard it and apply the below mentioned algorithm on remaining (n-1)/2 pairs.

(ii) Look at each pair: if the two elements are different, discard both; if they are the same, keep one of them:
We define a new array A'. For both even and odd n, we look at all the different pairs formed and check whether both elements of each pair are the same or not. If they are the same, we add that element (only once for a pair) to A'. If different, we will not add any element to A'.
Then we recursively apply this algorithm on A', i.e. again divide it into pairs.

(iii) If at any step while discarding the pairs which do not have same elements, we discard all (no elements remain), then we conclude that there is no majority element present. If we finally get a single element at last, it will be the majority element of our array A.

(iv) If we get an element at the end of our algorithm in (iii), we will check the occurrences of this element in A in linear time. If (no. of occurrences)> (size of A)/2, we output that element. Else we output “No majority element”.

Correctness:

Let the majority element (if present) of A be M. We need to prove that this element M would be carried forward in each step and at last we would be left with this element M which is our answer.
Let the number of Ms present in A be x. If this is the majority element, then x> n/2.
Now we are dividing the elements into pairs. Let there be k pairs in which both the elements are ‘M’, i.e. the pair is (M,M) and there are k such pairs. Hence there would be k Ms in A’. Thus, we can say that (x-2k) Ms are left (which are not paired as (M,M)). These x-2k Ms would be paired with other (x-2k) elements which are not M. These (x-2k) pairs would not appear in A’ since both elements of these pairs are different.
Till now, x Ms and (x-2k) elements other than M have been paired.
Thus total (2x-2k) elements have been paired. Hence (n- 2x +2k) elements are yet unpaired. Therefore maximum (n-2x+2k)/2 pairs are possible which can appear in A’.
Hence length of A’ will be less than or equal to k + (n-2x+2k)/2 (since initially we formed k pairs of (M,M) type)
=> length of A’ <= (2k + n/2 - x) ….1
Now we know that x>n/2
Thus n/2-x <0
=> (2k+n/2-x) <2k …..2
From 1 and 2,
Length of A’ < 2k
I.e. k> (length of A’)/2
Hence we proved that Number of Ms in A’ > (Length of A’)/2
i.e. M is the majority element of A’ as well.

Thus at each recursive step the majority element of the current array would be the same as the majority element of the previous array (here we form the current array from the previous array by selecting only those elements which appear twice in the formed pairs according to our algorithm).

If n is odd and the algorithm returns M, then M occurs more than (n-1)/2 times

i.e., x>(n-1)/2
Or x>=(n-1)/2+1
=> x>= (n+1)/2
Therefore, M occurs more than half times the size of A. Hence, it is the correct majority element of A.

Thus, we have proved that if A has a majority element, our algorithm correctly returns it.

Note that if at any recursion step, the output is “No majority element”, then it implies that the input will also not have any majority element.
Since, the statement (if no majority element in A’ then no majority element in A) is the contrapositive of the statement (if majority element in A then majority element in A’) and we know that a conditional statement is logically equivalent to its contrapositive.

We are checking the output of recursion before returning it (in step iv), in this check, the count of the output of recursion will be less than half of size of A if there is no majority element in A.Therefore, our algorithm will correctly return “No majority element” if A does not have a majority element.

Hence Proved.

<u>Time Complexity Analysis:</u>

At each step, we are performing O(n) operations in step iv (while checking the number of occurrences of the output of recursion) and in step ii (while comparing the elements of each pair). We recurse on array A’ which is of size at most n/2.

Hence T(n) = T(n/2) + O(n)
Let k = log(n) with base 2
=> T(n) = T(n/2) + cn
=> T(n/2) = T(n/4) +cn/2
.
.
.

$T(n/2^k) = T(1) + c$

$\Rightarrow T(n) = T(1) + cn + cn/2 + cn/4 + \ldots\ldots + c$
$\Rightarrow T(n) = T(1) + cn\,(1 + 2^{(-1)} + 2^{(-2)} + \ldots\ldots + 2^{(-k)})$
$\Rightarrow T(n) = T(1) + cn\,(1*(1 - (2^{(-k)}))/ (1-(½)))$
$\Rightarrow T(n) = T(1) + cn\,(2* (1- 2^{(-\log n)}))$
$\Rightarrow T(n) = T(1) + cn\,(2* (1- n^{(-\log 2)}))$
$\Rightarrow T(n) = T(1) + cn\,(1- (1/n))$ …. ($\log 2 = 1$)
$\Rightarrow T(n) = T(1) + cn - c$
$\Rightarrow T(n) = O(n)$ …. (since $T(1)$ is constant time and c is arbitrary constant)

Hence we find a linear time algorithm to find the majority element.

3.2 part b **12 / 12**

✓ **+ 12 pts** *Correct*