COL351 Assignment 3

Avval Amil - 2020CS10330
Ujjwal Mehta - 2020CS10401

October 2022

1 Particle Interaction

For this question, we will use the fact that two polynomials of $\mathcal{O}(n)$ degree can be multiplied in $\mathcal{O}(n \cdot log(n))$ time using the Fast Fourier Transform.
First, we define two functions as follows -

$$f_1(x) = \sum_{i=1}^{n} a_i \cdot x^i = \sum_{i=1}^{n} q_i \cdot x^i \tag{1}$$

$$f_2(x) = \sum_{i=-(n-1), i\neq 0}^{n-1} b_i \cdot x^i = \sum_{i=-(n-1), i\neq 0}^{n-1} \frac{sgn(i)}{i^2} \cdot x^i \tag{2}$$

Here, $sgn(i)$ is the signum function defined to be 1 if the argument i is positive, else it takes the value -1.
Note that $f_2(x)$ is not a polynomial, we will first give our analysis using these functions then show how the answers can be computed using polynomial multiplication.

Claim 1

In the function $f(x) = f_1(x) \cdot f_2(x)$, let the coefficient of x^i be denoted by c_i. Then $c_i = \frac{F_i}{Cq_i}$ where F_i is the total force on the particle at i^{th} position with charge q_i and C is electromagnetic force constant.

Proof

We can see that on multiplying the functions $f_1(x)$ and $f_2(x)$, the function $f(x)$ will still be a rational function which can be represented as $f(x) = \sum_i c_i \cdot x^i$ for some range of i.
Also, we can see that the coefficient of x^i can be written as -

$$c_i = \sum_{j=1}^{n} a_j \cdot b_{i-j}$$

This is simply seen by the fact that if we take each term of the polynomial $f_1(x)$ (let's say of degree j) then to make a coefficient of degree i in the product $f_1(x) \cdot f_2(x)$ we have to take the term of with a power of x equal to $(i-j)$ from $f_2(x)$ (note that $(i-j)$ can be negative). So -

$$c_i = \sum_{j=1, j\neq i}^{n} q_j \cdot \frac{sgn(i-j)}{(i-j)^2}$$

The inequality $j \neq i$ is present as the coefficient of x^0 (b_0) is zero in the function $f_2(x)$.

If we look at the term $C \cdot q_i \cdot c_i$:

$$Cq_ic_i = \sum_{j=1, j\neq i}^{n} Cq_iq_j \cdot \frac{sgn(i-j)}{(i-j)^2} = \sum_{j<i} \frac{Cq_iq_j}{(i-j)^2} - \sum_{j>i} \frac{Cq_iq_j}{(i-j)^2}$$

This term is precisely the force on the i^{th} particle as described in the question. Hence, we can say that $F_i = Cq_ic_i$ or that $c_i = \frac{F_i}{Cq_i}$. Hence, proved

Claim 2

Define $f_3(x) = x^{n-1} \cdot f_2(x)$. Then $f_3(x)$ is a polynomial and let $g(x) = f_1(x) \cdot f_3(x) = \sum_i d_i \cdot x^i$ for some range of i. Note that $g(x)$ is now a polynomial as both $f_1(x)$ and $f_2(x)$ are now polynomials. We claim that $d_{i+n-1} = c_i$.

Proof

We know that $f(x) = f_1(x) \cdot f_2(x) = \sum_i c_i \cdot x^i$. So, we can write-

$$g(x) = f_1(x) \cdot f_3(x) = x^{n-1} \cdot f_1(x) \cdot f_2(x) = x^{n-1} \sum_i c_i \cdot x^i = \sum_i c_i \cdot x^{i+n-1}$$

But as we also know that $g(x) = \sum_i d_i \cdot x^i$. Thus -

$$\sum_i d_i \cdot x^i = \sum_j c_j \cdot x^{j+n-1}$$

But, as two polynomials are equal iff all their coefficients are the equal, we can say that on choosing a value of j such that $j + n - 1 = i$, the coefficients c_j and d_i have to be equal. Thus, $d_i = c_j$ if $j + n - 1 = i$ or $j = i - n + 1$. This implies that $d_i = c_{i-n+1}$ or $d_{i+n-1} = c_i$ (replace i with $i+n-1$). Hence, proved.

From Claim 1 and Claim 2, we can say that given the polynomial $g(x) = f_1(x) \cdot f_3(x)$, the coefficient of x^{i+n-1} in $g(x)$ is given by $d_{i+n-1} = \frac{F_i}{Cq_i}$.

Algorithm

The algorithm for computing the force on all the particles now given.

Step 1

Construct polynomials

$$f_1(x) = \sum_{i=1}^{n} q_i \cdot x^i$$

and

$$f_3(x) = \sum_{i=-(n-1), i\neq 0}^{n-1} \frac{sgn(i)}{i^2} \cdot x^{i+n-1}$$

This step takes $\mathcal{O}(n)$ time.

Step 2

Find the product of the polynomials $g(x) = f_1(x) \cdot f_3(x)$ using the Fast Fourier Transform algorithm done in class such that $g(x) = \sum_{j=1}^{3n-2} d_j \cdot x^j$. This step takes $\mathcal{O}(n \cdot log(n))$ time (also done in class).

Step 3

The final force on the i^{th} particle can be found out as $F_i = Cq_i d_{i+n-1}$. The forces on all the particles can be found using this relation. Since we are computing the forces for all the particles, this step takes $\mathcal{O}(n)$ time.

Proof of Correctness

The proof of correctness that F_i does indeed equal to the overall force on the particle follows by the proofs of the claims we did above as we showed that on multiplying the two polynomials the coefficients that we get can be used to find out the force in the same way as described in the algorithm.

Time Complexity Analysis

The total times for all the steps of the algorithm is $\mathcal{O}(n) + \mathcal{O}(n \cdot log(n)) + \mathcal{O}(n)$ which is simply $\mathcal{O}(n \cdot log(n))$.

2 Non-Dominated Points

Solution: Here in this question, we have been given a set of points $P = \{p_i = (x_i, y_i) | 1 \leq i \leq n\}$ with $x_i, y_i > 0$, for $i \in [1, n]$ and there are no two points with same x-coordinate or y-coordinate and we need to design an $O(n \log(n))$ time algorithm in order to find the set of all the non-dominated points in P. Now in order to solve this problem, we have the following algorithm :-

Algorithm:
// This function returns the set of all the non-dominated points for given point list P.

function non_dominated_points(P) {
// Here we will already have the collection of points P sorted with respect to the x-coordinate of the points(so that we can find median point with respect to x-coordinate each time easily)

Step 1:- Find the median of points with respect to the x-coordinate in set P. (Let's say that this point is $p_{median} = (x_{median}, y_{median})$) (Note that this can be calculated easily since we already have the x-coordinates in sorted order in P)

Step 2a:- If the number of elements in set P is equal to 1 or 0 then return whole set P as answer.

Step 2b:- Else Generate 2 new sets of points P_{left} and P_{right} by filtering the original set P such that for all points $p_{left} = (x_{left}, y_{left}) \in P_{left}$, we have $x_{left} \leq x_{median}$ and for all points $p_{right} = (x_{right}, y_{right}) \in P_{right}$, we have $x_{right} > x_{median}$ (This can be done by single traversal of points in P and the order of elements in P_{left} and P_{right} will remain sorted in x-coordinate since P already has points sorted in x-coordinate).

Step 3:- Get the non-dominated sets of points for P_{left} and P_{right} by recursively calling the non_dominated_points function on both of these i.e. get 2 new sets of non-dominated points $answer_{left}$ = non_dominated_points(P_{left}) and $answer_{right}$ = non_dominated_points(P_{right}).

Step 4:- Traverse the P_{right} and get the maximum value of y-coordinate in this set (Let's say this value is y_{max}).

Step 5:- Traverse and remove all the points in $answer_{left}$ which have the y-coordinate value less than y_{max}.

Step 6:- Finally merge the filtered $answer_{left}$ set and the $answer_{right}$ set and return this as our final answer (i.e. return $answer_{left} \cup answer_{right}$).

Final Algorithm :
Step 1 :- Sort the given set of points P in increasing order of their x-coordinate.

Step 2 :- Return non_dominated_points(P).

Proof of Correctness :
In order to prove correctness of our algorithm, we first prove correctness of the function non_dominated_points using **strong induction**.

Proof:
Induction Base Case: Since when the number of points in P is equal to 0 or 1 then the whole set P is non-dominated as their will be no other point in P which dominates a given point in P, Hence the base case is true.

Induction Hypothesis: When the number of points in set P is $\leq (n-1)$, then the given function computes correctly the set of all the non-dominated points.

Induction: When the number of points in P is $= n$, then upon using this algorithm we first split set P around it's x-coordinate median point, and compute all the non-dominated points on the list P_{left} and P_{right} and since the number of points in these sets would be less then n, by induction hypothesis, they will give us the correct set of all non-dominated points for them respectively. Now if we examine these non-dominated points sets then we can say that all the non-dominated points of list P_{right} will have their x-coordinate larger than all the points in P_{left} (since we split around median) so no point in P_{left} can dominate the non-dominated points of P_{right}. Next, if we find the maximum y-coordinate among the points of P_{right} then all the non-dominated points in P_{left} whose y-coordinate is less then this maximum will not be non-dominated points for P since this maximum y-coordinate point in P_{right} will dominate these points (as $x_{max} > x_{point}$ and $y_{max} > y_{point}$. All the remaining non-dominated points of P_{left} will be non-dominated for P too, (since there is no point in P_{right} whose y-coordinate is more then these points and they are already non-dominated in P_{left} already). Hence on combining these 2 sets (filtered non-dominated points of P_{left} and non-dominated points of P_{right}) we will get all the non-dominated points of P. Hence our induction is proved with the above reasoning.

Final Algorithm Proof of Correctness:
Here since we first sort the points of P in increasing order of x-coordinate and then we call the function non_dominated_points whose correctness we have already proved, hence we can say with certainty that our algorithm computes the correct result.

Time Complexity Analysis:

Here in order to do time complexity analysis, we will add the individual time complexities for each step and then develop a recursive relation. Let's say that time to compute answer for P of size n is $T(n)$ (for function non_dominated_points).
$\Rightarrow T(n) = O(1) + O(n) + 2 \cdot T(n/2) + O(n) + O(n)$
(Note that here Step 1 takes $O(1)$ time to compute(as points already sorted

in x-coordinate), Step 2 takes $O(n)$ time as we need to traverse the whole list of points, Step 3 takes $2 \cdot T(n/2)$ time as we are calling recursively the same function on size $n/2$ and Step 5 and Step 6 takes $O(n)$ each since we again need to traverse the points.)

Hence we get the $T(n)$ as :-
$\Rightarrow T(n) = 2 \cdot T(n/2) + O(n)$
Hence by direct result of **master's theorem** we can write $T(n) = O(n \log(n))$.
Total time of Algorithm = Time of sorting $P + O(n \log(n))$.
$\Rightarrow$ **Hence Total Time of Algorithm =** $O(n \log(n))$ (since sorting can be done in $O(n \log(n))$ time. Hence we have designed the algorithm with required constraints.

3 Majority

Part (a)

We will first prove a claim, and then give the algorithm and its proof of correctness.

Claim 1

Let A be an array and A_1 and A_2 be its left and right halves respectively. If the array A has a majority element (let's say m), then m is a majority element of at least one of A_1 and A_2.

Proof

We will prove the above claim using contradiction. Assume that the number of elements in A is n and the majority element m of A (with let's say M occurrences in A) is neither a majority element of A_1 nor of A_2. Also, let sizes of A_1 and A_2 be n_1 and n_2 respectively. Thus, $n_1 + n_2 = n$
Since m is a majority element of A, the number of occurrences of it in A must be strictly larger than $n/2$. This means that $M > n/2$. Also, let the number of occurrences of m in A_1 and A_2 respectively be M_1 and M_2. This implies that

$$M_1 + M_2 = M \tag{3}$$

This is because the sum of occurrences in the left and right half are the total occurrences. Also, it implies that $M_1 < (n_1)/2$ and $M_2 < (n_2)/2$. This is because the element m is not majority in either of the left or right subarrays. But, on adding these inequalities we get -

$$M_1 + M_2 < (n_1 + n_2)/2 = n/2$$

But, using equation (3) this implies that $M < n/2$ which means that m could not have been a majority element of A which is a contradiction. Hence, proved.

Algorithm

The algorithm to solve the problem using divide and conquer is now given. This algorithm is a recursive one and we are assuming that it will return the majority element of the array A it is given as input if it exists otherwise it will tell us that no majority element exists. -

Step 1
If the number of elements in the array A is one, then return that element itself as the majority element. Otherwise, split the array about its middle element into two subarrays A_1 and A_2.

Step 2
Recursively compute the majority elements of the two subarrays.

Step 3 If there is no majority element in either subarray, then return 'No majority element found'. If only one of the subarrays has a majority element m, then count the occurrences of m in the total array A. If the total occurrences are more than $n/2$ then return m as the majority element otherwise return 'No majority element found'. If both the subarrays have majority elements m_1 and m_2 (which may or may not be distinct), then find the number of occurrences of both these elements in the array A and if any one has number of occurrences more than $n/2$ then return that as the majority. If neither has number of occurrences more than $n/2$ then return 'No majority element found'.

Proof of Correctness

The proof of correctness of this algorithm will be using strong induction.

Base Case
When the number of elements in the array is 1, then it is the majority element as $1 > 1/2$ and our algorithm also returns the same. Thus, the base case is correctly computed by the algorithm.

Induction Hypothesis
For a given integer k, if the size of the input array to the algorithm is less than k, then the algorithm correctly returns the output.

Induction Step
Let $k > 1$ be the size of the input array A. Then two subarrays A_1 and A_2 will be created from A and the answer will be recursively computed on them. Using Claim 1 we can say that if the majority element of A exists then it will surely be a majority element of either A_1 or of A_2. Also, by the induction hypothesis we know that if a majority element exists in either of the subarrays, it will be returned correctly. So, after checking the majority elements of the two subarrays returned whether they have more than $n/2$ elements or not, if we find a majority then we are done, and if we don't then we can be sure that there exists no majority (as claimed in Claim 1). Hence, proved.

Time Complexity Analysis

In the algorithm described, let $T(n)$ be the time taken to find the majority on an input array of size n. Step 1 of the algorithm takes $\mathcal{O}(n)$ time. For step 2, it will take $2 \cdot T(n/2)$ time as we are calling the algorithm recursively on two subarrays of half the size. The last step takes $\mathcal{O}(n)$ time as we have to iterate through the array twice in the worst case if both the subarrays return a majority element.
So, overall the total time can be described using the following recurrence equation -

$$T(n) = 2 \cdot T(n/2) + \mathcal{O}(n) \tag{4}$$

By the Master Theorem for recurrence equations, we can say that $T(n) \in \mathcal{O}(n \cdot log(n))$. Hence, we have designed the required algorithm.

Part (b)

First we describe a procedure for constructing a smaller array using a given array A (of even size) which we will be using afterwards and prove a claim for the same procedure.

Procedure 1

Step 1
Pair the elements of the array A of even size n such that two consecutive elements form one pair. Maintain an empty array (list) B which will contain some elements in the future.

Step 2
For each pair in the array A, if the elements of the pair are the same, add exactly one of those identical elements to the list B, otherwise don't add any element to the list B. After iterating through all the pairs, return B.

Let the array B returned by this procedure be denoted as $B = Proc(A)$.

Claim 1

If for two arrays A and B, if $B = Proc(A)$, where $Proc()$ is the procedure described above, if A has a majority element m then m is also a majority element of B.

Proof

Let the majority element of A be m. Then in array A after pairing consecutive elements, there can be three types of pairs -

1. m paired with m

2. m paired with a such that $a \neq m$
3. a paired with b such that $a \neq m$ and $b \neq m$ (although they may be separately equal to each other)

Let there be x pairs of the first kind, y pairs of the second kind and z pairs of the third kind. Then the total number of elements in array A are $2(x+y+z)$. Also, the number of occurrences of m in array A are $2x+y$ (as it appears twice in each pair of first type and once in each pair of second type and zero times in pairs of the third type). As m is the majority element in A, we can say that -

$$2x + y > 2(x + y + z)/2$$

Which can be simplified to -

$$x > z \tag{5}$$

Now, in the array $B = Proc(A)$, we know that all pairs of the first type will contribute exactly 1 element equal to m to B. Also, we know that the pairs of the second type will not contribute any element to B as their elements are unequal. Finally, the pairs of the third type may or may not contribute an element to B (this is because if the pair is (a, b) then we don't know whether a and b are equal or not, we only know that they are not equal to m). What we can say is that there can be at most z elements which are contributed to B by pairs of the third type. This is because each pair can contribute at most one element. Also, these elements are guaranteed to not be equal to m.
Thus, in the array B we can say that there will be x elements equal to m, and at most z elements not equal to m. Let the elements not equal to m in B be z'. The size of B is thus $x + z'$. Also, we have the following inequality -

$$z' < z$$

Using inequality (5), we can deduce that -

$$z' < z < x$$

Now, add x to both sides of the above inequality to get -

$$x + z' < 2x$$

As $x + z' = Size(B)$ this can be rewritten as -

$$(Size(B))/2 < x$$

The above inequality implies that m is also a majority element in the array B, as the number of occurrences x of it in the array exceed half the size of the array. Hence, we have proved the claim.

Claim 2

Let A be an array of odd length $(2n + 1)$ for some positive integer n. Let the first element of A be t and let the rest of the array (which does not contain t) be denoted by A'. Then if A' has a majority element m then m is also a majority element of A.

Proof

As m is a majority element of A', it implies that the number of occurrences of m (let's say x) are greater than size of A'. This means that -

$$x > (2n + 1 - 1)/2 = n$$

or that -

$$x \geq n + 1$$

This is because x is an integer. However, we see that $x \geq (n + 1) > (2n + 1)/2$ and hence m must also be a majority element of A as its number of occurrences are more than half the size of A. Hence, Proved.

Claim 3

Let A be an array of odd length $(2n + 1)$ for some positive integer n. Let the first element of A be t and let the rest of the array (which does not contain t) be denoted by A'. Then if A' has no majority element but A has a majority element, it is equal to t.

Proof

The proof of this will be by contradiction. Assume that there is a majority element of A (let's say m) which is not equal to t. Then the number of occurrences of it in A must be greater than $(2n + 1)/2 > n$. However, since all its occurrences occur in A' (it is not equal to t), it must have greater than n occurrences in A', but since the size of A' is $2n$, this means that m is a majority element in A' as well. This is a contradiction. Hence, if A has a majority and A' does not, then t is the majority element of A. Hence, proved.

Algorithm to Compute Majority Element if it Exists

Step 1
If the input array A has 1 element, then return that element as the majority element.

Step 2a
This step is done if size of array A is even. Create the array $B = Proc(A)$ as described in the procedure above. Recursively compute the majority element of B. Let this element be m. Check to see if the number of occurrences of m in A are more than $n/2$ where n is the size of A. If this is true, then return m as the majority element, otherwise return 'No Majority Element Found'. Also, if no majority element was returned for B, again return 'No Majority Element Found'.

Step 2b
This step is done if the size of the array A is odd. Remove the first element of

the array (let's say it is t) and store it. On the remaining array A' (which is of even length), apply step 2a. If a majority element is found then return it. If no majority element is found then check the number of occurrences of t in the array A. If t is a majority element return it, and if it is not return 'No Majority Element Found'.

Proof of Correctness

Again, we can prove the correctness of this algorithm using strong induction.

Base Case
The number of elements in the array A is one. In this case our algorithm returns that element itself as the majority which is correct. Thus, the base case holds.

Induction Hypothesis
For any size of the input array less than a given integer k, the algorithm returns the correct majority element if it exists, otherwise it returns 'No Majority Element Found'.

Induction Step
Let size of the input array A be k (k is even). First, the algorithm computes $B = Proc(A)$. We know that if the array A had a majority element, then it will also be a majority element of B. This is done in Claim 1 proved before. Also, by the induction hypothesis we can say that the majority element of B is correctly returned, since the size of B is at most half the size of A. After getting the majority element of B, we can check to see whether it is a majority element of A or not. This is because while we know that a majority element of A will be a majority element of B, the converse does not hold and hence checking is required. But if no majority element of B is returned, we can be sure that there is no majority element in A as well. Hence, the correctness of the algorithm is proved when k is even.
When k is odd, using Claim 2 we can say that on doing step 2b of the algorithm, if we find a majority element of A', it is also a majority element of A. However, if we don't find a majority element of A', then only possible majority element of A is t (by Claim 3). Hence, we check whether t is a majority element or not and then return the same. Hence, the correctness is also proved when k is odd.

Time Complexity Analysis

Let the time taken to compute the majority element of an array A of size n using this algorithm be given by $T(n)$. The time taken for step 1 of this algorithm is constant or $\mathcal{O}(1)$. For step 2a, we can see that firstly computing $B = Proc(A)$ takes $\mathcal{O}(n)$ time. This is because we have to iterate through all the elements to make pairs and then take at most one element from each pair. After this, the time taken to compute the majority element of B will be at most $T(n/2)$. This is because the maximum size of B is $n/2$. Finally, if we are returned with

a majority element of B, we will have to find the number of occurrences of it in A which will require another traversal over all elements of A and thus need $\mathcal{O}(n)$ time. For step 2b, we first have to repeat step 2a, and after that traverse the array at most once. So, step 2b takes at most $\mathcal{O}(n)$ time more than step 2a. Combining all these, we can formulate the following recurrence relation -

$$T(n) \leq T(n/2) + \mathcal{O}(n)$$

Using the Master Theorem to solve this recurrence relation, we get that the time taken on an input of size n is $T(n) \in \mathcal{O}(n)$. Thus, we have designed an algorithm with the desired properties.