COL351 Assignment 3

Viraj Agashe, Vishwas Kalani

TOTAL POINTS

60 / 60

QUESTION 1

1 Particle interaction 15 / 15

✓ **+ 15 pts Correct Solution**

+ 8 pts Algorithm Correctness

+ 4 pts Algorithm Partial Correct

+ 0 pts Incorrect Algorithm

+ 5 pts Correctness Proof

+ 2.5 pts Correctness Proof Partial Correct

+ 2 pts Time Complexity Analysis

+ 3 pts Partial Correct

QUESTION 2

2 Non-dominated points 23 / 23

✓ **+ 23 pts Correct**

+ 0 pts Incorrect/Unattempted

+ 8 pts Only Correct Algorithm

+ 6 pts Proof of Correctness (Each vertex not in solution must be dominated)

+ 6 pts Optimal algorithm (Any point **p** in our output **O** is not dominated by other point **q** in **O**)

+ 3 pts Correct reasoning for time complexity.

QUESTION 3

Majority 22 pts

3.1 part a 10 / 10

✓ **+ 10 pts Correct**

+ 3 pts Algorithm Correctness

+ 5 pts Correctness Proof

+ 2 pts Running Time Analysis Correctness

+ 1 pts Algorithm partial

+ 2 pts Correctness Proof Partial

+ 0 pts Incorrect Algorithm

3.2 part b 12 / 12

✓ **+ 12 pts Correct**

+ 2 pts Algorithm Correctness

+ 8 pts Algorithm Correctness Proof

+ 2 pts Time Analysis

+ 1 pts Partially Correct Algorithm

+ 4 pts Partially Correct Proof

+ 0 pts Incorrect Algorithm/Algorithm not given

1 Particle Interaction

1.1 Algorithm

We propose the following algorithm to find the force F_j on each particle j:

1. Consider the two polynomials $Q(x)$ and $R(x)$ given by:

$$Q(x) = q_1 + q_2 x + ... + q_n x^{n-1}$$

$$R(x) = \sum_{i=0}^{2n-2} r_i x^i$$

   Here, the coefficient r_i is defined as,

$$r_i = \begin{cases} \frac{-C}{(n-i-1)^2} & i \in [0, n-2] \\ 0 & i = n-1 \\ \frac{C}{(i-n+1)^2} & i \in [n, 2n-2] \end{cases}$$

2. Compute the product polynomial $F(x) = Q(x) \times R(x)$ of the above two polynomials using the Fast-Fourier Transform (FFT) algorithm.

3. The force on the j-th particle F_j is given by q_j times the coefficient of x^{j+n-2} in the product $F(x)$.

1.2 Proof of Correctness

Theorem 1. *The force F_j on the j-th particle is the coefficient of x^{j+n-2} in the product polynomial $F(x)$ times q_j.*

Proof. Let the coefficient of x^k in $F(x)$ be f_k. We know that,

$$f_k = \sum_{i=1}^{k+1} q_i r_{k-i+1}$$

Therefore, we know that:

$$f_{j+n-2} = \sum_{i=1}^{j+n-1} q_i r_{j+n-i-1}$$

Notice that by definition we have, (substituting $j+n-i-1$ for i)

$$r_{j+n-i-1} = \begin{cases} -\frac{C}{(i-j)^2} & i \in [j+1, n+j-1] \\ 0 & i = j \\ \frac{C}{(j-i)^2} & i \in [0, j-1] \end{cases}$$

So we have,

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

This is the expression for F_j as required. ■

1.3 Time Complexity Analysis

We claim that the above algorithm is $O(n \log n)$. We argue this as follows:

- Computing each coefficient for the polynomial $Q(x)$ and $R(x)$ is an $O(1)$ operation, and so the overall computation of the coefficients is $O(n)$.
- Using the FFT algorithm, we can multiple polynomials of degree $\leq 2n$ in $O(2n \log 2n)$.

Therefore, we can find the compute all of the forces F_j in time $O(n \log n)$.

1 Particle interaction 15 / 15

- ✓ **+ 15 pts Correct Solution**
- **+ 8 pts** Algorithm Correctness
- **+ 4 pts** Algorithm Partial Correct
- **+ 0 pts** Incorrect Algorithm
- **+ 5 pts** Correctness Proof
- **+ 2.5 pts** Correctness Proof Partial Correct
- **+ 2 pts** Time Complexity Analysis
- **+ 3 pts** Partial Correct

2 Non Dominated Points

2.1 Algorithm

The algorithm for finding the set of non-dominated points is as follows :

1. We sort all the points in P by x producing a list P_x
2. The first level of recursion will work as follows, with all further levels working in a completely analogous way. We define Q to be the set of points in the first $\lfloor \frac{n}{2} \rfloor$ positions of the list P_x (the "left half") and R to be the set of points in the final $\lceil \frac{n}{2} \rceil$ positions of the list P_x (the "right half").
3. Using the list P_x, we can further create lists Q_x and R_x in $O(n)$ time.
4. We now recursively determine the set of non-dominated points in Q (with access to the lists Q_x). Suppose that Q_{opt} is the set of non dominated points in Q and similarly R_{opt} is the set of non dominated points in R. (P_{opt} is the set of non dominated points in the entire set P)
5. **Merge step :** Add all the points of the set R_{opt} to P_{opt}. Let Y_R be the maximum y coordinate among the y coordinate of all the points in the set R_{opt}. (Y_R **will be the** y **coordinate of the first point in the set** R_{opt} **sorted as per the** x **coordinates**).
6. Traverse through the set Q_{opt} and add those points (x_i, y_i) to P_{opt} which satisfy $y_i > Y_R$. (**All the points in the sorted set** Q_{opt} **after the first dominated point will be dominated by some point in** R_{opt})
7. The resulting set P_{opt} from the recursive procedure above will be the set of non-dominated points.

2.2 Time Complexity Analysis

1. The time for sorting the set of points P as per the x coordinates in the beginning of the algorithm is $O(n \log(n))$.
2. The recursive equation for the operations of splitting and merging is:

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

$$T(n) = O(n \log(n))$$

   We are generating the lists Q_x and R_x in $O(n)$ time. The time to compute the set of non-dominated points in the sets Q and R is $T(\frac{n}{2})$ each. The operations of further merging the solution by traversing through the list Q_{opt} and comparing with the maximum of R_{opt} are also $O(n)$.
3. Therefore the overall time complexity of the algorithm is $T(n) + O(n \log(n)) = O(n \log(n))$

2.3 Proof of Correctness

Theorem 2. *For a set of points P, the set of non dominated points in the right half R of P (that is R_{opt}) will not be dominated by any point in the set P*

Proof. The set of points in R_{opt} are not dominated by any point in the set R by definition. Consider any point in L, the x coordinate of the point in L will be less than x coordinate of all the points in R_{opt} (as we are splitting according to the x coordinates in the split step). Therefore any point in L can't dominate any point on R_{opt}.

Theorem 3. *For a set of points P, the set of non dominated points in the left half Q of P(that is Q_{opt}) which have y coordinate greater than the maximum y coordinate in the set R_{opt} will not be dominated by any point in the set P*

Proof. The set of points in Q_{opt} are not dominated by any point in the set Q by definition. Let $p_l = (x_l, y_l)$ be a point in Q_{opt} such that $y_l > Y_R$ where Y_R be the maximum y coordinate among all the points in R_{opt}. Let $p_r = (x_r, y_r)$ be a point in R with largest x coordinate which dominates the point p_l. Thus we have $x_r > x_l$ and $y_r > y_l > Y_R$. Let us say X_R is the x-coordinate of the non dominated point with maximum y coordinate. Consider the case $x_r > X_R$, then the point p_r dominates (X_R, Y_R) which can't be the case as all the x and y coordinates are unique and (X_R, Y_R) is a non dominated point. If $x_r < X_R$, then the point p_r is not dominated by any point in the set P and Y_R can't be the maximum y coordinate of a non dominated point in R. Hence proved by contradiction.

Theorem 4. *The algorithm correctly outputs the set of non dominated points in the set P*

Proof. All the components of the proof have already been worked out, so here we just summarize how they fit together. Following is proof by induction on the length of set P
Base case : The algorithm correctly computes the set of non dominated points for $|P| \leq 3$
Induction hypothesis : The algorithm correctly computes the set of non-dominated points in the sets Q and R used in the recursive calls.
Induction step : Using the hypothesis and **Theorem** 2 and **Theorem** 3, we have that all the points in P_{opt} added as per the merge step are not dominated by any point in P. No other point in R outside R_{opt} can be added to P_{opt} because it is dominated by some point in R and thus by some point in P. No other point except those added by the algorithm from Q_{opt} can be part of P_{opt} as the point (X_R, Y_R) dominates that point. Hence P_{opt} is the correct set of non-dominated points in P.■

2 Non-dominated points 23 / 23

✓ **+ 23 pts Correct**

+ 0 pts Incorrect/Unattempted

+ 8 pts Only Correct Algorithm

+ 6 pts Proof of Correctness (Each vertex not in solution must be dominated)

+ 6 pts Optimal algorithm (Any point **p** in our output **O** is not dominated by other point **q** in **O**)

+ 3 pts Correct reasoning for time complexity.

3 Majority Element Part I

3.1 Algorithm

We propose the following algorithm for finding the majority element of an array A:

1. If the size of the array is n, split the array A into two arrays A_1 and A_2 of sizes $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$.
2. Recursively find the majority element of both the subarrays. Let these be a_1 and a_2 respectively. Note that:
   - For an array of size 1, the majority element is the only element of the array itself.
   - If A_2 does not have a majority element, do the below steps only for a_1, and vice-versa for A_1.
   - If both A_1 and A_2 do not have a majority element, then A does not have a majority element, return False.
3. Count the frequency of a_1 and a_2 in the original array A. Let the frequencies by $f(a_1)$ and $f(a_2)$.
4. If $f(a_1) > \lfloor \frac{n}{2} \rfloor$, then a_1 is the majority element. If $f(a_2) > \lfloor \frac{n}{2} \rfloor$ then a_2 is the majority element.
5. Otherwise, A does not have a majority element. Return False.

3.2 Time Complexity Analysis

We claim that the time complexity of the algorithm is $O(n \log n)$. Let the time to find the majority element of the array A of size n be $T(n)$. We argue this as follows:

- Divide Step: When we divide the array into two subarrays of half the size, the time taken is $2 \times T\left(\frac{n}{2}\right)$
- Conquer Step: When we combine the results from both subarrays, we will need to iterate atmost twice over the entire array to count the frequencies of a_1 and a_2. Thus, the conquer step is $O(n)$.

Thus, the recursive relation for the time complexity is

$$T(n) \leq 2 \times T\left(\frac{n}{2}\right) + O(n)$$

This is the same recursive relation as MergeSort, and the solution is $T(n) = O(n \log n)$.

3.3 Proof of Correctness

To prove the correctness of our algorithm, we will first prove the correctness of our conquer step, and then argue the correctness of the algorithm using induction. Note that in all following proofs, A_1 and A_2 refer to the subarrays obtained by dividing A in half, and a_1, a_2 are their majority elements.

Theorem 5. *a can be the majority element of array A only if a is the majority element of atleast one of A_1 and A_2.*

Proof. Suppose a is the majority element of A. By definition, the frequency $f(a) > \frac{n}{2}$. Suppose to the contrary that a is not the majority element of either A_1 or A_2. Then the frequencies of a in A_1 and A_2 satisfy $f_1(a) \leq \frac{\lfloor \frac{n}{2} \rfloor}{2}$ and $f_2(a) \leq \frac{\lceil \frac{n}{2} \rceil}{2}$. Therefore we must have,

$$f(a) = f_1(a) + f_2(a) \leq \frac{\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil}{2} \leq \frac{n}{2}$$

This contradicts $f(a) > \frac{n}{2}$. Therefore, a must be the majority element of atleast one of A_1 and A_2. ■

Now, we will argue the correctness of our algorithm using the Principle of Mathematical Induction. We induct on the size of the array, n.

Base Case: For an array of size 1, by definition the majority element is the sole element of the array, which is what the algorithm returns (see step 2 of algorithm).

Inductive Hypothesis: Suppose the algorithm computes the correct majority element or returns if no such element exists for all arrays of size $k < n$.

Inductive Step: Consider an array of size $k = n$. Then, we divide the array into two subarrays of sizes $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$. By the inductive hypothesis, the algorithm computes the correct result for the subarrays. Now, by theorem 5 we know the majority element of A must be one of a_1 or a_2 (if they exist). Therefore, we explicitly check if a_1 and a_2 are the majority elements by counting their frequencies in the original array. If $f(a_1) > \frac{n}{2}$ or $f(a_2) > \frac{n}{2}$, the algorithm returns those elements, otherwise it returns False, as required by theorem 5. Therefore, the algorithm is correct. ■

3.1 part a **10 / 10**

✓ **+ 10 pts Correct**

+ 3 pts Algorithm Correctness

+ 5 pts Correctness Proof

+ 2 pts Running Time Analysis Correctness

+ 1 pts Algorithm partial

+ 2 pts Correctness Proof Partial

+ 0 pts Incorrect Algorithm

4 Majority Element Part II

We devise a linear-time algorithm, with an additional input, which is a tiebreaker element t (which may be null) that will help deal with odd length arrays. The algorithm is as follows:

1. If $n = 0$, the majority element is simply the tiebreaker element t.
2. Otherwise, obtain an array A' from A by the following procedure:
   - Pair up the elements of A. If $|A|$ is odd, there will be a single unpaired element.
   - For each pair, if the two elements are different, discard both.
   - If the elements of the pair are the same, add one of them to A'.
3. Recursively apply the procedure to the array A'. The tiebreaker element t_1 for the recursive call is decided as follows:
   - t_1 is the odd unpaired element of A if $|A| = n$ is odd.
   - $t_1 = t$ if n is even.
4. Say the element returned by the recursive call is a. Verify if the element returned is actually the majority element, by a single pass over the array.
5. If the above check passes, then a is the majority element of the array A. Else return False.

Note that at the top-level, the algorithm will be called with $t =$ null.

4.1 Psuedocode

Algorithm 1 Majority(A,t)

Input: Array $A = [a_0, a_1, ...a_{n-1}]$ with $|A| = n$, tiebreaker element t
Output: Majority element of A, if it exists

```
if n = 0 then
    return t
end if
A' ← [ ]                                              ▷ Array of pairs
while i < n − 1 do
    if A[i] = A[i + 1] then
        A' ← A' + A[i]
    end if
    i ← i + 2
end while
if n is odd then
    t_1 ← A[n − 1]
else
    t_1 ← t
end if
m ← Majority(A', t_1)
if checkMajority(m, A) then                           ▷ Checks if m is majority element
    return m
else
    return False
end if
```

4.2 Time Complexity Analysis

We claim that the algorithm is linear time, i.e. $O(n)$. We argue this as follows:

- In the divide step, we create the array A' from A, whose size is atmost $\frac{n}{2}$, since we keep atmost one element from each pair. So, the time for solving the subproblem is $T\left(\frac{n}{2}\right)$.
- Besides this, for pairing up $\frac{n}{2}$ elements, we need to iterate through the list. Also, finally we need to check if the given element is actually in the majority, which is another linear time check. Therefore, the recursive relation for the algorithm is:

$$T(n) \leq T\left(\frac{n}{2}\right) + O(n)$$

- Using the Master's Theorem for recursive relations, here we have: $a = 1$, $b = 2$ and $d = 1$. Since $\frac{a}{b^d} < 1$, we get the time complexity as $O(n)$.

4.3 Proof of Correctness

We will argue the proof of correctness of the algorithm by first stating and proving the following theorem, and then applying induction to prove the divide and conquer step. Note that in the following discourse, $f(x, L)$ refers to the frequency of x in an array L. We also slightly restate the condition for an element to be a majority element of an array (to include the tiebreaker element) as satisfying one of the following conditions:

- $f(a, A) > \frac{|A|}{2}$
- $f(a, A) = \frac{|A|}{2}$ and $t = a$ (i.e. tiebreaker element is a)

Note that this definition is identical to the original definition of a majority element, since the tiebreaker element is also an element of the original array, but we essentially assign it a smaller weight for the purpose of the recursive algorithm to handle odd length arrays. At the top level of the recursion (i.e. for the original array), we will have t to be null.

Let A' be the elements of A after pairing and discarding as per the algorithm. Under the above definitions, let us now consider the following theorem:

Theorem 6. *Suppose there exists a majority element a given an array A and a tiebreaker element t. Then a is also the majority element given the combination of array A' and tiebreaker t_1. (here, t_1 is as defined in the algorithm)*

Proof. Let $n = |A|$, the frequency of a in A be $m = f(a, A)$, and the "excess" frequency of a in A be $f_e = m - (n - m) = 2m - n$. Then, by the above definition of majority element, the condition for a to be a majority element of A is

$$(f_e > 0) \vee (f_e = 0 \ \wedge \ a = t)$$

Now we consider the combination A' and t'. Let q represent the number of elements different from a in A. We will consider two cases:

1. **Case I: n is even.** Note that since n is even, $f_e = 2m - n$ must be even. Thus we have the condition as $(f_e >= 2) \vee (f_e = 0 \ \wedge \ a = t)$. Now note that there are $f_e + q$ elements of a and q elements different from a in the array A. Therefore, atleast f_e elements of m will be paired up amongst themselves, and the condition becomes $(f_e >= 1) \vee (f_e = 0 \ \wedge \ a = t)$ which is the same as $(f_e > 0) \vee (f_e = 0 \ \wedge \ a = t)$. Therefore, a is a majority element for A' and t_1.
2. **Case II: n is odd.** In this case, $|A'|$ pairs will be formed and one element t_1 will be unmatched. Therefore:

- If $t_1 \neq a$, then note that the array $A \setminus t_1$ is of even length, and further for $A \setminus t_1$, $f_e > 0$. This is the same as case I (n is even).
- If $t_1 = a$, the only troublesome case is when $f_e = 1$ initially, otherwise the proof proceeds similarly to above. Then, for the array $A \setminus t_1$ we have $f_e = 0$, and so there may be a tie in A' (i.e. $f_e = 0$ for A' due to all q elements of a getting paired up with elements different from a). However, since $t_1 = a$, it can break the tie. So we have $f_e = 0 \ \wedge \ a = t$, and therefore a is a majority element for A' and t_1.

Hence, the element a is also a majority element given the combination A' and tiebreaker t_1. ■

Now, we will argue the correctness of our algorithm using the Principle of Mathematical Induction. We induct on the size of the array n.

Base Case: For an array of size 0, the majority element must be the tiebreaker element t. Thus the base case is verified. (Note that at the top level, the algorithm is called with t initialized to null. Thus, for an empty input array the algorithm returns null which is the behaviour as required.)

Inductive Hypothesis: Suppose the algorithm computes the correct majority element or returns if no such element exists for all arrays of size $k < n$ and tiebreaker elements t.

Inductive Step: Consider an array of size $k = n$. Then, after pairing up the elements and selecting/discarding as suggested by the algorithm, we are left with an array of size atmost $\lfloor \frac{n}{2} \rfloor$ and the tiebreaker element t_1. By the inductive hypothesis, the algorithm computes the correct majority element for this array A' and tiebreaker t_1. Let this element be a. Now, by theorem 6 we know that if A has a majority element, then it must be a. Therefore, all that remains to be checked is that if A has a majority element in the first place. Therefore, we verify if $2 \times f(a, A) > |A|$. If yes, we return a, otherwise we return False, as required by theorem 6. Therefore, the algorithm is correct. ■

3.2 part b **12 / 12**

✓ **+ 12 pts Correct**

+ **2 pts** Algorithm Correctness
+ **8 pts** Algorithm Correctness Proof
+ **2 pts** Time Analysis
+ **1 pts** Partially Correct Algorithm
+ **4 pts** Partially Correct Proof
+ **0 pts** Incorrect Algorithm/Algorithm not given

COL351: Assignment 3

Vishwas Kalani (2020CS10411)
Viraj Agashe (2020CS10567)

September 2022

Contents