CS345 Theoretical Assignment 1

Abhibhav Garg 150010
Abhisek Panda 150026

September 30, 2017

1 Question 1

1.1 Output Sensitive Algorithm

The divide and conquer algorithm is used, with a small modification: A linear time operation, namely, finding out the point with the maximum y coordinate in the left divided half is found, and all points dominated by this point are removed, which again takes linear time. This guarantees that one **non dominated point of the complete set** is found before every pair of recursive calls, as opposed to first finding non dominated points in subproblems. Since there are 2^k recursive calls at the k^{th} level of the recursion tree, this gives us bounds of $\log h$ on the recursion depth. The detailed proofs and analysis is given below.

Pseudocode:

Algorithm 1 Non Dominated Points in $n \log h$

```
ndPoints ← emptyList                                   ▷ Final list of non dominated points
function GETND(p)
    xmax = max_x(p), ymax = max_y(p).
    if xmax == ymax then
        ndPoints.append(ymax) ▷ Base case, if the same point has max x and y coordinate, it is the only nd point
        return
    end if
    rp ← Right half points of p along median_x
    ymax ← max_x(max_y(rp))
    ndPoints.append(ymax)                              ▷ ymax is a guaranteed nd point
    p ← p \ dom(ymax).                                 ▷ dom returns set of points dominated by argument
    lp ← Left half points of p along median_x
    rp ← Points to the right of ymax
    getND(lp)
    getND(rp)
    return
end function
```

Proof of Correctness: We attempt to show that the point that we add to the global list of non dominated points is not dominated by any point in the complete set of n points, and not just in the set of points that make up the current subproblem. Thus, in what follows, when the points to the right of or left of a point are mentioned, this includes every point to the right (or left), and not just the subproblem.

The algorithm first splits the points according to the median along the x axis. From the right half, we pick the maximum y coordinate point, $ymax$. This point is definitely a non dominated point, since every point to the left of it has lower x coordinate and every point to the right of it has smaller y coordinate. Thus, we can add this to our set of non dominated points, and can remove every point dominated by it.

If we show that the $ymax$ point gotten on recursive calls on the splits of the points both belong to the set of non dominated points, we are done.

Consider the left split. When we further split this into two halfs and take the maximum y coordinate of the second half, by the same argument as above, all points on the left of it have lower x coordinate and all points to the right have lower y coordinate, and thus this new $ymax$ is a non dominated point.

On the left half, the argument is slightly trickier. Taking the $ymax$ on the right half of this split gives us a point whose y coordinate is maximum only amongst the subproblem, and not all the points as required. But, since we removed all points lower than the parent $ymax$, we can guarantee that this point has higher y coordinate that all points on its right, and is thus not dominated.

This way, we guarantee that a non dominated point is found before every set of recursive calls. The base case is fairly straightforward as the only way for there to be a single non dominated point is if it dominates every point.

Time Complexity Analysis At every level of the recursion, we do $O(n)$ work to check the base case, split the points, find $ymax$, remove the points dominated by it, and split once again, before calling recursive calls. The depth of the recursion tree, as mentioned above, is $\log h$, since at level k there are 2^k subproblems, each finding one non dominated point before calling more child processes.

$O(n)$ work at each level, and $\log h$ levels gives us a time complexity of $O(n \log h)$. Note that general methods such as Master Theorem cannot be used to find the time complexity, since the recurrence relationship is not well defined.

1.2 Extension to Three Dimensions

1.2.1 Online Algorithm in Two Dimensions

A balanced binary search tree is maintained for this problem. This tree maintains the current list of non dominated points, sorted by one of the coordinates, say x.

Algorithm: When a new point is seen, its successor in the tree is found.

If the new point is dominated by its successor, then it is ignored. Note that if the point is not dominated by the successor in the tree, it cannot be dominated by any point seen so far. This is because, if it were dominated by some other point that was not the successor, such a point would have a greater x coordinate that the successor(by definition of successor) and a greater y coordinate, since it dominates the new point, and thus the successor would not have been in our structure.

If not, then the predecessor is found. If the new point dominates this predecessor, the predecessor is removed. Then, the next predecessor is found, and this process is repeated till the predecessor is not dominated by the new point. At this point, the new point is added to the structure.

At any step, if the point we see has the highest x coordinate, or when looking for a predecessor we get a NULL node, the new point is added to the structure because it is a non dominated point by definition.

Proof of Correctness

The proof is by method of invariant. The invariant is that at any time k, the structure has all the non dominated points of the set of points seen till k. This is trivially true at $k = 0$ and $k = 1$, since the first point has maximum x coordinate by definition.

Let this be true at some point t. At time $t + 1$, we get a new point, and check it according to the rules above. The fact that the invariant holds is by construction - if the point is dominated by the successor, it is removed, else we have proved that it is a non dominated point, and is thus added to the structure, after removing every point it dominates. Since the addition of a point cannot turn a dominated point into a non dominated one, no non dominated point escapes our data structure, and this completes the proof.

Time Complexity: In the case where the new point is dominated, or the case where it does not dominate the predecessor, the time to insert is simply a constant number of red black tree operations, and is thus logarithmic in the number of points seen till that point.

No such guarantees exist when points need to be removed. However, note that every point enters and exits the data structure atmost once each. This guarantees, that upto time k, the maximum work done for a given point is atmost proportional to $2 * \log k$, and thus the time complexity for inserting k points is $O(k \log k)$.

1.2.2 Algorithm for Three Dimensions

The fact that if a point is not dominated by the successor along one axis, then it is a non dominated point fails in 3 dimensions. Thus, the trivial algorithm requires us to check every incoming point against every existing point, which gives bad time complexity. In order to fix this, the points are first sorted along one of the axis, and taken in order. This lets us reuse the structure from above to solve the problem.

Algorithm: The points are first sorted along one of the coordinates, say z. A balanced search tree similar to above is maintained for the projection of the points on the $x - y$ plane, and points are considered one at a time, from the highest to lowest z coordinate. This data structure however, does not maintain all the points that are currently non-dominated. A separate list is maintained for that purpose. It only maintains all points that are required to check if the incoming points are non dominated.(We only add to the list if we are certain of non dominancy, thus no delete operations are needed).

The operations are as follows: If a new point is seen, which is dominated in the projection on the $x - y$ plane by a point i, then it is dominated in 3D space by i, by virtue of decreasing z coordinate and is thus not included in the structure.

If the new point dominates a point j in the projection, the z coordinate of j is checked. If it is the same as that of the new point, then j is deleted. If not, then though j is dominated in the $x - y$ projection, its z coordinate is higher. Further, every point that we are yet to see has lower z coordinate than j. Further, the new point dominates every point that we are yet to see, that is dominated by j, since the new point has higher x and y coordinates than j, and every point yet to be seen has lower z coordinate. Thus, j can be removed from the structure, and put in the list, since having it in the structure is redundant.

Finally, once every point is exhausted, every point in the structure is also added to the list and then returned.

Proof of Correctness The invariant here is the fact that the list only contains elements that we know to be non dominated, and that the structure plus the list contain all the non dominated points seen till the time of insertion of the last point.

As before, the statement is trivially true at time 0 and 1. Let it be true at time k. Since the structure has all non dominated points at time k, so does the array, by construction. It has also been shown that elements in the array can be forgotten about.

When a new point is seen, if its projection is dominated, so is it and it can be ignored. This follows from the fact that we add elements in the decreasing order of z coordinates. If a point dominates the existing points on the projection, the existing points(unless the z coordinate) are the same, unlike in 2D, are still non dominating points, since the new point has lesser z coordinate. Further, removing them does no harm, since if a newer point is going to be dominated by the removed point, it will also be dominated by the point we just added. Thus it can be added to the array(This is actually important to do, since otherwise, the most basic claim that if a point is not dominated by the successor, then it is not dominated at all, fails).

Thus, if any point is dominated, it will necessarily be removed from the structure, and if a point is not dominated, it can never be removed from the structure plus the list. This completes the proof.

Time Complexity: Sorting the array takes $O(n \log n)$ time initially. After that, we perform the same operations as those in the previous problem of the online $2 - D$ algorithm, with some constant extra amount of work at every step for checking z coordinates. This gives us $O(n \log n)$ time complexity.

2 Question 2

2.1 Force on charged particles linearly placed at regular intervals

The force on charged particle j due to all other charged particles is given by:

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

Since the charges are linearly spaced, we can substitute x_k simply with k for each $1 <= k <= n$.

Taking inspiration from the polynomial multiplication algorithm, an $O(nlogn)$ algorithm, which is significantly faster than the $O(n^2)$ trivial algorithm, we can calculate the forces much faster if we are able to construct 2 polynomials such that the coefficients in their product are the required quantities.

A natural guess for the first of those 2 polynomials will be:

$$\begin{aligned} P(x) &= q_1x + q_2x^2 + q_3x^3 + ... + q_nx^n \\ &= \sum_{k=1}^{k=n} q_kx^k \end{aligned}$$

Lets try to construct the second polynomial, if possible, from $P(x)$. Fixing some notations, let $C_k(T(x))$ denote the coefficient of x^k in $T(x)$.

Consider:

$$\begin{aligned} Q(x) &= \frac{x^1}{1^2} + \frac{x^2}{2^2} + \frac{x^3}{3^2} + \frac{x^4}{4^2} + ... + \frac{x^n}{n^2} \\ &= \sum_{1}^{n} \frac{x^n}{n^2} \end{aligned}$$

$$\begin{aligned} C_k(P(x) * Q(x)) &= \frac{q_1}{(k-1)^2} + \frac{q_2}{(k-2)^2} + \frac{q_3}{(k-3)^2} + ... + \frac{q_{k-1}}{1^2} \\ &= \sum_{l=1}^{l=k-1} \frac{q_l}{(k-l)^2} \\ &= \sum_{l<k} \frac{q_l}{(k-l)^2} \end{aligned}$$

This looks promising since this is proportional to the force exerted on k^{th} particle by the charges on its left(force in +x direction).

Next, consider the polynomial

$$\begin{aligned} R(x) &= \frac{1}{x.1^2} + \frac{1}{x^2.2^2} + \frac{1}{x^3.3^2} + \frac{1}{x^4.4^2} + ... + \frac{1}{x^n.n^2} \\ &= \sum_{1}^{n} \frac{1}{x^kk^2} \\ C_k(P(x) * R(x)) &= \frac{q_{k+1}}{1^2} + \frac{q_{k+2}}{2^2} + \frac{q_{k+3}}{3^2} + ... + \frac{q_{k+(n-k)}}{(n-k)^2} \\ &= \sum_{l=1}^{l=n-k} \frac{q_{k+l}}{l^2} \\ &= \sum_{l>k} \frac{q_l}{(n-(l+1))^2} \end{aligned}$$

Observe that $C_k(P(x) * R(x))$ is proportional to the force exerted on charge k by all the charged particles to the right of it.

Thus, the required force on the k^{th} charged particle can be formulated as:

$$\begin{aligned} F_k &= Kq_k * [C_k(P(x) * Q(x)) - C_k(P(x) * R(x))] \\ &= Kq_k C_k[P(x) * (Q(x) - R(x))] \end{aligned}$$

However note that $R(x)$ is not a polynomial since it contains negative powers. This can be easily overcome by multiplying x^n to $(Q(x) - R(x))$ and checking the coefficient of x^{n+k} instead of x^k for finding F_k. Thus the final algorithm becomes:

- Step 1: Construct polynomials: $P(x) = \sum_{k=0}^{k=n} q_k x^k$, $Q(x) = x^n * \sum_{k=1}^{k=n} \frac{x^k}{k^2}$ and $R(x) = \sum_{k=1}^{k=n} \frac{x^n}{x^k k^2}$
- Step 2: Find the product polynomial $F(x) = P(x) * (Q(x) - R(x))$
- Step 3: Report force on k^{th} particle as $Kq_k * C_{n+k}(F(x))$, i.e, Kq_k* (coefficient of x^{n+k} in $F(x)$).

2.2 Time Complexity Analysis:

The first step would take linear time for constructing each of the 3 polynomials since all the coefficients can be calculated in O(1) time.

The second step is the standard polynomial multiplication algorithm(as discussed in class) which can be solved in $O(n \log n)$ time by using divide and conquer strategy.

The third step would again take linear time since force on each particle can be reported in O(1) time.

Thus, the overall time complexity of the algorithm would be $O(n \log n)$ which is quite faster than the $O(n^2)$ naive algorithm.