CS345 : Algorithms II
Semester I, 2021-22, CSE, IIT Kanpur

Assignment I

Deadline : 11:55 PM, 22 August 2020.

Most Important guidelines

- It is only through the assignments that one learns the most about the algorithms and data structures. You are advised to refrain from searching for a solution on the net or from a notebook or from other fellow students. Remember - **Before cheating the instructor, you are cheating yourself**. The onus of learning from a course lies first on you. So act wisely while working on this assignment.
- Refrain from collaborating with the students of other groups. If any evidence is found that confirms copying, the penalty will be very harsh. Refer to the website at the link: https://cse.iitk.ac.in/pages/AntiCheatingPolicy.html regarding the departmental policy on cheating.

General guidelines

1. There are four problems in this assignment: 2 Difficult problems, 1 Moderate, and 1 Easy. Each difficult problem carries 100 marks, the moderate one carries 75 marks, and the easy one carries 50 marks. Attempt **only** one of the 4 problems.

2. You are strongly discouraged to submit the scanned copy of a handwritten solution. Instead, you should prepare your answer using any text processing software (LaTex, Microsoft word, ...). The final submission should be a single pdf file.

3. You need to justify any claim that you make during the analysis of the algorithm. But you must be formal, concise, and precise.

4. If you are asked to design an algorithm, you may state the algorithm either in plain English or a pseudocode. But it must be formal, complete, unambiguous, and easy to read. You must not submit any code (in C++ or C, python, ...).

5. **Naming the file**:
   The submission file has to be given a name that reflects the information about the assignment number, version attempted (difficult/moderate/esay), and the roll numbers of the 2 students of the group. For example, you should name the file as **D_1_Rollnumber1_Rollnumber2.pdf** if you are submitting the solution for the difficult version of the 1st assignment. In a similar manner, the name should be **M_1_Rollnumber1_Rollnumber2.pdf** and **E_1_Rollnumber1_Rollnumber2.pdf** if you are submitting the solution for the moderate problem and the easy problem respectively of the 1st assignment.

6. Both students of a group have to upload the identical copy of their final submission. Be careful during the submission of an assignment. Once submitted, it can not be re-submitted.

7. Deadline is strict. Make sure you upload the assignment well in time to avoid last minute rush.

8. Contact TA at the email address: devansh@cse.iitk.ac.in for all queries related to the submission of the assignment. Avoid sending any such queries to the instructor.

Difficult

Faster algorithm for Non-dominated points in a plane

Recall the problem of non-dominated problem discussed in the second lecture of this course. We discussed two algorithms for this problem. The first algorithm takes $O(nh)$ time, where h is the number of non-dominated points in the given set P. The second algorithm, which was based on the divide and conquer paradigm, takes $O(n \log n)$ time. As a part of this assignment, you have to design an $O(n \log h)$ time algorithm for non-dominated points. Interestingly, you have to use the insight from the first algorithm to just *slightly* modify the second algorithm so that its running time is improved to $O(n \log h)$. You must describe the algorithm and also provide the complete details of the analysis of its running time.
Remark: Note that $O(n \log h)$ is superior to $O(n \log n)$ in those cases where the number of non-dominated points are very few. In fact, it can be shown that if n points are selected randomly uniformly from a unit square, then the expected(average) number of non-dominated points is just $O(\log n)$.

OR

Non-dominated points in 3 dimensions

You can extend the notion of non-dominated points to 3 dimensions as well. Spend some time to realize that it is not so simple to apply divide and conquer strategy to compute the non-dominated points in 3 dimensions. Some of you may be tempted to solve this problem by reducing an instance of this problem to three instances of 2-dimensional problem (by projecting the points on x-y, y-z, x-z plane). But that will be wrong (think over it to convince yourself). Interestingly, there is a very simple elegant algorithm using a simple data structure that computes non-dominated points in 3 dimensions. The purpose of this exercise is to make you realize this fact.

1. Design an algorithm that receives n points in x-y plane one by one and maintains the non-dominated points in an online fashion. Upon insertion of ith point, the algorithm should take $O(\log i)$ time to update the set of non-dominated points.
   Note: It is perfectly fine if your algorithm only guarantees a bound of $O(i \log i)$ on the total time for insertion of i points. It is not necessary that your algorithm achieves $O(\log i)$ bound on the processing carried out during insertion of ith point.

2. Design an $O(n \log n)$ time algorithm to compute non-dominated points of a set of n points in 3 dimensions. You must use part (a) above carefully.

Moderate

Convex Hull

In Lecture 2, we discussed an $O(n \log^2 n)$ time algorithm to compute the convex hull of a given set of n points in a plane. If we can improve the time complexity of the Conquer step of this algorithm to linear, this will result in an $O(n \log n)$ time algorithm for convex hull. You have to modify the current Conquer step so that it takes at most linear time. You must provide a complete analysis of the modified Conquer step as well.

Easy

Counting Double-Inversions

You are given an array A storing n numbers. A pair (i, j) with $0 \leq i < j \leq n-1$ is said to be a double-inversion if $A[i] > 2A[j]$. Design and analyze an $O(n \log n)$ time algorithm based on divide and conquer paradigm to compute the total number of double-inversions in A.

The sketch of the solutions and the grading policy

D1

Algorithms

There are 2 algorithms for this problem. Both of them are similar to the divide and conquer algorithm discussed in the class with the a subtle variation:
Algorithm 1:

1. We split the input set into 2 sets of nearly the same size after computing the x-median. Let these sets be the Left Half set and the Right Half set.
2. First invoke the recursive call for the Right Half set. This also provides the point of maximum y-coordinate from the Right Half set. Let the corresponding maximum y-coordinate be $y_{\max}$.
3. Eliminate all those points from the Left Half set whose y-coordinate is less than $y_{\max}$. If the resulting set has at least one point, invoke the recursive call on this set.

Algorithm 2:

1. We split the input set into 2 sets of nearly the same size after computing the x-median. Let these sets be Left Half set and the Right Half set.
2. Compute the point of the maximum y-coordinate from the Right Half set. Let it be p'. Surely, p' is a non-dominated point. Output it.
3. Eliminate all those points from the Left Half set as well as the Right Half set that are dominated by p'. For each of the resulting sets, invoke the recursive call for computing its non-dominated points unless it is an empty set.

Grading Policy: 40 marks are for the design of the algorithm and 10 marks are for showing that the algorithm outputs all the non-dominated points and no dominated point from the given set. The remaining 50 marks are for the time complexity analysis.

Time complexity analysis

The time complexity of both the algorithms is $O(n \log h)$ where h is the number of non-dominated points in the given set P. There are 2 totally different approaches to show this. Majority of the students have pursued one of these approaches though many of them made serious mistakes. Of course, there are other ways to establish the $O(n \log h)$ bound.

1st Approach

This approach works for both 1st Algorithm and the 2nd Algorithm. This approach is to establish the time complexity bound using a proof by induction. Choose any arbitrary but fixed n; and then we induct on h from $h = 1$ to $h = n$. Define term $T(n, h)$ as follows: $T(n, h)$ is the worst case time complexity of the algorithm taken over all sets P of n points with exactly h non-dominated points. It is easy to observe that $T(n, 0) = 0$
The inductive assertion is the following: There exists a constant c such that for each $1 \leq h \leq n$, $T(n, h) \leq cn(1 + \log_2 h)$.
Let the time complexity spent during in the divide step and the combine step for input of size n be at most $c'n$ for some constant c'.

The key point to focus on during the proof is that we need to show the existence of constant c in terms of c' for which the assertion holds for all values of h.

Base case ($h = 1$) is easy to establish if we choose $c \geq 2c'$ (try it on your own). For the induction step, we need to express the time complexity in terms of the time complexity of the recursive call of the right Half set and the pruned Left Half set. After giving suitable arguments, one can arrive at the following inequality:

$$T(n, h) \leq \max_{0 \leq h' \leq h} (T(n/2, h') + T(n/2, h - h') + c'n)$$

Using Induction hypothesis, we plug in the values for the terms on the right side. After carrying out simple manipulation, you can show that $T(n,h) \leq cn(1+\log_2 h)$ if $c > c'$.

Using this inequality and the inequality of the base case, we can conclude that $T(n,h) < cn(1+\log_2 h)$ for all values of h provided $c > 2c'$.

A common mistake: A common mistake that most of the students have made in their proof by induction is that they have not remained stuck to the original constant c. Instead, they have chosen the constants at their ease. This is seriously wrong. To demonstrate the seriousness, look at the following (wrong) proof by induction to show that time complexity of merge sort is $O(n)$.

Claim: $T(n) \leq cn$.

(Wrong) Proof: We make use of the following recurrence for the time complexity of the Merge sort:

$$T(n) = 2T(n/2) + c'n$$

So using I.H., $T(n/2) \leq cn$. Hence $T(n) \leq 2cn/2 + c'n$.

Hence $T(n) \leq c_0 n$ if we choose $c_0 > c + c'$.

The penalty for mishandling the constants will be from 20 to 35 marks depending upon the extent of mishandling.

2nd Approach

This approach is for the 2nd algorithm. The key observation is that each recursive call is associated with a unique non-dominated point. This is the point that is computed during the divide step (the point p' in the description of the 2nd algorithm). So the number of recursive calls is upper bounded by h. We calculate the time spent in the divide as well as combine step of each recursive call of 2nd Algorithm and sum it up and establish a bound of $O(n \log_2 h)$ on this time.

First observe that for a recursive call at i level, the amount of computation we perform in the divide and combine step is $O(n/2^i)$. There are at most 2^i recursive calls at level i of the recursion tree. So for the first $\log_2 h$ levels, the total time spent in divide and combine step of all the recursive calls is obviously $O(n \log_2 h)$. For the remaining levels, note that the time spent in the divide and combine step of each recursive call is $O(n/h)$. Since the total number of recursive calls is $\leq h$, so the total time spent in all recursive calls beyond level $\log_2 h$ is $O(n/h \times h) = O(n)$. Hence the total time complexity of the algorithm is $O(n \log_2 h)$.

A serious mistake that most of the students have made is to state and use the following (wrong) assertion:

The height of the recursion tree is $O(\log_2 h)$.

But this assertion is wrong for Algorothm 1 as well as Algorithm 2. The worst case height of the recursion tree for the new algorithm is still $\Theta(\log_2 n)$. Consider the following input as a counterexample of the above assertion for both the algorithms:

Input: Let $k = \log_2 n$. Consider a stair case structure defined by k points. Let these points be labeled $p_1, \ldots p_k$ as we move from the left to the right. We shall now insert n more points as follows. Insert $n/2$ points below the staircase but to the left of p_1. Insert $n/4$ points below the staircase but to the left of p_2 and to the right of p_1; and so on ...

Analysis of the algorithm for the above input and its output: It is easy to observe that for the above input, the number recursive calls is $\log_2 n$. However, the number of non-dominated points is $h = k = \log n$. In fact, the recursion tree is a skewed tree. The total time complexity of the algorithm for this case is $O(n)$.

The maximum marks for analysis that make use of the (wrong) assertion "The height of the recursion tree is $O(\log_2 h)$" will be at most 10 out of 50.

D2

Part 1 is 60 marks, while Part 2 is 40 marks.

Part 1

The **Algorithm 1** guarantees a bound of $O(i \log i)$ on the total time for insertion of i points.
Algorithm 1:(20)

1. Maintain a Balanced Binary Search Tree T(Red-Black trees or AVL trees) with points sorted according to x-coordinate. This tree will contain the Non-dominated points.
2. Insert point $P := (x_p, y_p)$ in T. If $succ(T, P).y \geq y_p$, $Delete(T, P)$ and return as the point P is dominated.
3. If $pred(T, P).y \leq y_p$, $Delete(T, pred(P))$. Repeat till the condition is false, or $pred(P)$ is NULL.

Proof of Correctness (20)

1. It can easily proved that in the Dominating Set no 2 points will have same x or y coordinates.
2. From definition of Non-dominating points, it can be proved that the BST of Non-dominating points according to x-coordinate, will be BST wrt to y-coordinate in Descending order.
3. Using 2, we can say that if $pred(P)$ has larger y than P then all elements with smaller x-coordinate will have y-coordinate larger than P. Also, if $succ(P)$ has smaller y than P, then all points with larger x than P will not be dominating P.
4. The algorithm ensures the conditions mentioned in 3, and hence T will have non-dominating points.

Time complexity (20)

It is important to see that for a certain insertion, $O(i)$ many points may need to be deleted(when deleting predecessor's). Thus, each step will not be necessarily $O(\log i)$.

For better analysis, consider the processing for each point. Every point is once inserted, and will be deleted only once. For each point, one *succ* call and one *pred* call(where $pred(T, P).y > y_p$) is done while insertion, and if deleted another *pred* call. Thus, the processing for each point is $O(\log i)$ and overall running time after i insertions is $O(i \log i)$.

Part 2

Algorithm(10)

Algorithm:

1. Maintain a list L and an online 2D system of ND points(made in Part 1).Sort all points wrt an axis in descending order, let's say z.
2. Insert all points one-by-one in descending order wrt z-axis into the ND set.
3. If a point is accepted into the ND set, put it in the list L. At end Output L.

Proof of Correctness(25)

If a point P is accepted into the system, then for all the other points with z larger than z_p one of x or y-coordinate is smaller than P. All other points have smaller z and hence will not be able to dominate P. Thus, every point inserted in L will be a ND point.

If a point P is not accepted in ND set, then there must exist Q, such that $x_Q > x_P$ and $y_Q > y_P$. Since, the points were inserted in descending order, $z_Q > z_P$, and hence Q dominates P. Thus, every point that is not inserted in L is dominated.

Showing both of the above is necessary for correct proof of correctness.

Time Complexity(5)

Sorting wrt z coordinate using mergesort will take $O(n \log n)$ time. The online system will take $O(n \log n)$ time total using Algorithm 1 or 2 of part 1. Thus, complete Algorithm runs in $O(n \log n)$ time.

Moderate Problem

In the class, we discussed an $O(n \log n)$ time algorithm to compute upper tangent of left convex hull and the right convex hull. If we are able to do this step in $O(n)$ time, this will result in $O(n \log n)$ time algorithm for convex hull.

In order to achieve it, the key idea is the following. Let p be any point in the upper hull of the left convex hull and q is any point in the upper hull of the right convex hull. Let L be the line joining p and q. If L is upper tangent (both left convex hull and right convex hull are below it), we are done. It takes $O(1)$ time to determine it using the basic tools we discussed during this problem. Otherwise, suppose L cuts left convex hull. In this case, can you see that there is a "set of points" of the upper hull of the left convex hull that can be discarded, i.e., none of them will ever form the upper tangent of the two convex hulls. In a similar way, if L cuts the right convex hull, a set of points of the upper hull of the right convex hull that can be discarded. Give rigorous arguments in support of these observations.

Solution 1: Based on the key idea stated above, there is a solution for the conquer step that is very much similar to the merge step in the merge sort. We start processing the points of the upper tangent of left convex hull in anticlockwise direction and the points of the upper tangent of the right convex hull in the clockwise direction.

Solution 2: Based on the key idea stated above, there is a solution for the conquer step that is very much similar to binary search in 2 sorted arrays. If the upper hulls of the left convex hull and the right convex hull are given in "sorted" order, it may take $O(\log n)$ time to compute the upper tangent. However, the overall time complexity of the resulting algorithm will still be $O(n \log n)$ because the conquer step has to spend $O(n)$ time in doing the "basic" tasks - (think over it).