Lecture Notes: An Output-Sensitive Algorithm for 2D Maxima

Yufei Tao
Department of Computer Science and Engineering
Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

January 22, 2024

In this lecture, we will revisit the *maxima problem* defined. Let us recall the relevant definitions. Given two different points (x_1, y_1) and (x_2, y_2) in $\mathbb{R}^2$, we say that the former *dominates* the latter if $x_1 \geq x_2$ and $y_1 \geq y_2$ (note that the two equalities cannot hold simultaneously because these are two different points). Let P be a set of n points in $\mathbb{R}^2$. A point $p \in P$ is a *maximal point* of P if p is not dominated by any point in P. The goal is to report all the maximal points of P efficiently. In the example of Figure 1, points 1, 2, 6, and 8 should be reported.

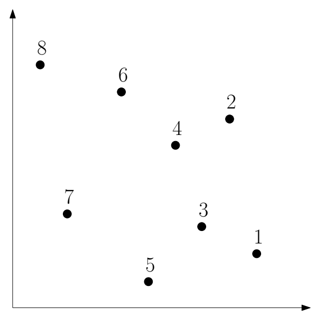

Figure 1: An example

The problem can be easily solved in $O(n \log n)$ time. Today, we will give an *output-sensitive* algorithm that finishes in $O(n \log k)$ time, where k is the number of maximal points. The technique behind this algorithm can be deployed to obtain output-sensitive algorithms for other problems as well, e.g., convex hull.

Before continuing, let us first observe a simple $O(nk)$ time algorithm. First, find the rightmost point p of P in $O(n)$ time, which must be a maximal point. Then, in $O(n)$ time remove from P (i) all the points dominated by p and also (ii) p itself. Now, the rightmost point of (the remaining) P is also guaranteed to be a maximal point. Repeat the above steps until P becomes empty.

1 Utilizing An Upper Bound of k

Let us first assume that, by magic, we know an upper bound $\hat{k}$ of k (e.g., $\hat{k} = n$ is a trivial upper bound). We will design an algorithm whose efficiency depends on $\hat{k}$.

First, divide P by x-coordinate into $\hat{k}$ subsets $P_1, ..., P_{\hat{k}}$ such that

- every point in P_i has a larger x-coordinate than all the points in P_j for any $1 \leq i < j \leq \hat{k}$;
- $|P_1| = |P_2| = ... = |P_{\hat{k}}| = O(n/\hat{k})$.

This can be done in $O(n \log \hat{k})$ time using a standard rank selection algorithm (see appendix).

Next, we process the subsets P_i in ascending order of i. As an invariant, after P_i has been processed, we must have computed the maximal points of $P_1 \cup ... \cup P_i$ (observe that they must also be maximal points of P). We achieve the purpose as follows. First, all the maximal points of P_1 are found in $O(t_1 \cdot |P_1|) = O(t_1 \cdot n/\hat{k})$ time, where t_1 is the number of those points. In general, assuming that the invariant holds after P_i, we process P_{i+1} as follows. Let p^* be the highest of all the maximal points in $P_1 \cup ... \cup P_i$. Scan P_{i+1} to remove all the points dominated by p^*. Then, find all the maximal points of the *remaining* P_{i+1} in $O(t_{i+1} \cdot |P_{i+1}|) = O(t_{i+1} \cdot n/\hat{k})$ time, where t_{i+1} is the number of those points. Note that all these points must also be maximal points of $P_1 \cup ... \cup P_{i+1}$. Overall, we spend $O((n/\hat{k}) \sum_{i=1}^{k} t_i) = O((n/\hat{k}) \cdot k) = O(n)$ time.

We thus have proved:

Lemma 1. *If an upper bound $\hat{k}$ of k is known, we can find all the maximal points in at most $cn \log \hat{k}$ time for some constant c.*

2 The Final Algorithm

Lemma 1 is not immediately helpful: if we set $\hat{k}$ to the trivial bound n, then the running time $O(n \log \hat{k})$ is no better than $O(n \log n)$. Next, we will employ the lemma in a clever way to achieve the desired $O(n \log k)$ bound.

The main idea is to ask the algorithm take a guess k' of k. Initially, the algorithm sets k' to 1 and, if $k' < k$ (i.e., k' fails to be an upper bound of k, we increase our guess k' and repeat. Crucially, we can *detect* whether $k' < k$ in $O(n \log k')$ time, thanks to Lemma 1. Specifically, we simply run the algorithm of Section 1 by setting $\hat{k} = k'$, and keep monitoring the algorithm's cost (this means counting the number of unit-time atomic operations in the RAM model). If $k' \geq k$, then by Lemma 1, the algorithm should terminate within $cn \log k'$ time. Hence, as soon as the algorithm's cost reaches $1 + cn \log k'$, we can manually terminate the algorithm and declare that $k' < k$.

Motivated by this, we start with $k' = 2^1$. If $k' < k$, we increase k' to 2^2 and try again. In general, if $k' = 2^{2^i}$ is still smaller than k, the next k' we try is $\min\{2^{2^{i+1}}, n\}$. Clearly, this algorithm will eventually find all the maximal points: it does so when k' is at least k for the first time.

Suppose that eventually the algorithm stops at $k' = 2^{2^i}$ for some integer $i \geq 0$. The total running time is:

$$
\begin{aligned}
& O\left(n \log 2^{2^0} + n \log 2^{2^1} + n \log 2^{2^2} + n \log 2^{2^3} + ... + n \log 2^{2^i}\right) \\
= \; & O\left(n \left(2^0 + 2^1 + 2^2 + ... + 2^i\right)\right) \\
= \; & O(n \cdot 2^i)
\end{aligned}
$$

How large is 2^i? The definition of i implies $2^{2^{i-1}} < k$, namely, $2^i < 2 \log_2 k$. We thus have obtained an algorithm solving the maxima problem in $O(n \cdot 2^i) = O(n \log k)$ time.

Appendix: Multi-Rank Selection

Let S be a set of n real values. We say that a value $v \in S$ has rank i if $|\{u \in S \mid u \geq v\}| = i$ (i.e., the largest value in S has rank 1, the second largest rank 2, ...). Given any rank $r \in [1, n]$, the element with rank r can be selected in linear time $O(n)$ using a textbook rank selection algorithm.

In the *multi-rank selection problem*, suppose we are given k ranks $r_1, ..., r_k$ in ascending order, and need to find the k corresponding elements. This is do-able in $O(n \log k)$ time as follows. Without loss of generality, let us assume that k is a power of 2. We first pick the median of $r_{k/2}$ of $\{r_1, ..., r_k\}$, and find the element e with rank $r_{k/2}$. Then, divide S into S_1 and S_2 such that (i) the former includes all the elements of S at least e, and (ii) the latter includes the other elements of S. We now recurse on two instances of the multi-rank selection problem: the first one on S_1 with ranks $r_1, ..., r_{k/2}$, and the second one on S_2 with ranks $r_{1+k/2} - k/2, r_{2+k/2} - k/2, ..., r_k - k/2$.

Let us analyze the running time. Define $f(n, k)$ as the time of the above algorithm when k ranks are to be computed from an input set of size n. If $k = 1$, we know $f(n, k) = O(n)$. For $k > 1$, we have:

$$f(n, k) = f(|S_1|, k/2) + f(n - |S - 1|, k/2).$$

Solving the recurrence gives $f(n, k) = O(n \log k)$.