Albert-Ludwigs-Universität, Inst. für Informatik
Prof. Dr. Fabian Kuhn
H. Ghodselahi, Y. Maus November 10, 2016

Algorithm Theory, Winter Term 2016/17
Problem Set 2 - Sample Solution

Exercise 1: Almost Closest Pairs of Points (16 points)

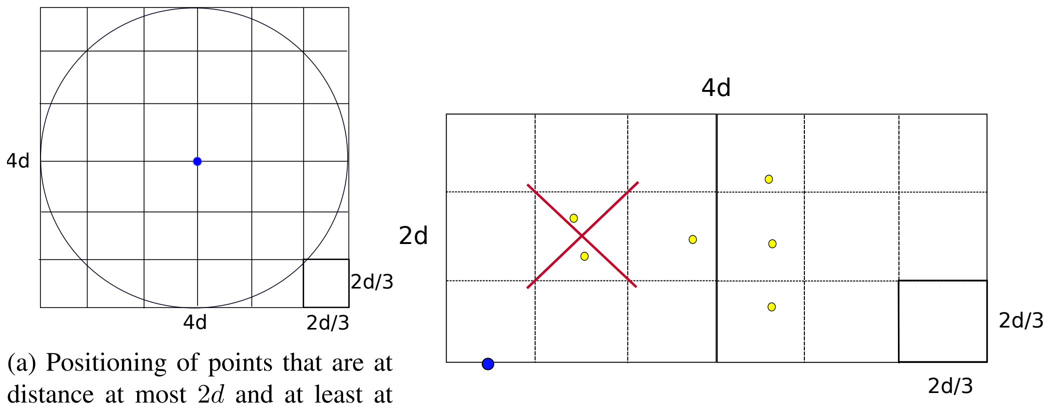
In the lecture, we discussed an $\mathcal{O}(n \log n)$-time divide-and-conquer algorithm to determine the closest pair of points. Assume that we are not only interested in the closest pair of points, but in all pairs of points that are at distance at most twice the distance between the closest two points.

a) **(4 points)** How many such pairs of points can there be? It is sufficient to give your answer using big-$\mathcal{O}$ notation.

b) **(12 points)** Devise an algorithm that outputs a list with all pairs of points at distance at most twice the distance between the closest two points. Describe what you have to change compared to the closest pair algorithm of the lecture and analyze the running time of your algorithm.

Solution

The recursive algorithm for finding the closest pair of points, that was presented in the lecture, recursively divided the set of points on the plane into two sets and found the minimal distance as $\min\{d_\ell, d_r, d_{\ell r}\}$, where d_ℓ and d_r are the minimal distances among the pairs of points in both sets and $d_{\ell r}$ is the minimal distance between pairs of points that lie in different sets. It was shown that finding $d_{\ell r}$ has linear complexity and the overall running time of the algorithm is $O(n \log n)$.



(a) Positioning of points that are at distance at most $2d$ and at least at distance d around the actual point we are checking.

(b) There is at most one point in each small square. We need to look only at most at 17 points.

Figure 1: Each small square can contain at most one point inside.

a) Let us assume that the closest pair of points is already known and the distance between them is d. For each point p we evaluate how many points there can be in distance $2d$. Figure 1a shows how the points at distance at most $2d$ from the center point can be covered using 36 squares with side length $2d/3$. As $2\sqrt{2}d/3 < d$ each such square can contain at most one point (including points on the boundary) and since the center point is part of 4 squares, the figure shows that a point can have at most 32 other points at distance at most $2d$. This way we count each pair twice and thus the number of pairs of points at distance at most $2d$ is at most $\frac{32}{2}n = O(n)$.

b) Now, let us modify the divide-and-conquer algorithm from the lecture, in order to solve our task.

As in the closest pair algorithm of the lecture, after sorting points by their x-coordinate, we divide the set of points into a left subset S_l and a right subset S_r of equal size and we recursively find the smallest distance d_ℓ (d_r) in the left (right) half, as well as the list L_ℓ (L_r) that contains all pairs of points in S_ℓ (S_r) that are in distance at most $2d_\ell$ ($2d_r$) of each other. For the merging step we find the smallest distance between points that are on different sides of the division line and at the same time compile a list of pairs of points at distance at most $2\min\{d_\ell, d_r\}$ such that the points lie on different sides. We compute $d = \min\{d_\ell, d_r, d_{\ell r}\}$, concatenate the three lists and remove all pairs $< p, q >$ for which $\text{dist}(p, q) > 2d$ from the combined list. Finally, we return d and the list with all pairs of points at distance at most $2d$.

dist(S): returns $< d, L, Y >$

Divide: Divide (sorted) set of points S in two equally sized sets S_ℓ and S_r.

Conquer: Apply algorithm to both sets S_ℓ and S_r. Compute minimal distances d_ℓ, d_r, lists of point pairs L_ℓ and L_r and lists Y_ℓ and Y_r of points sorted by their y-coordinate.

Merge: Compute $d_{\ell r}$ and $L_{\ell r}$ and return $d := \min(d_\ell, d_r, d_{\ell r})$. $L' := L_\ell \cup L_r \cup L_{\ell r}$; $L := L \setminus \{< p, q >: \text{dist}(p, q) > 2d\}$; return L. Merge Y_ℓ and Y_r into a new sorted list Y.

Let us take a closer look at merge step of the algorithm. For the merge step of the algorithm that was looking for the closest pair of points, it was shown that for each point, at most 7 points that lie in a rectangle $d \times 2d$ have to be checked to find $d_{\ell r}$ correctly. This resulted in a merge step of cost $O(n)$. For finding $d_{\ell r}$ we do the same, but for constructing $L_{\ell r}$ we consider a rectangle $2d \times 4d$. To illustrate our idea, consider the picture below (Figure 1b). If we divide this rectangle into squares $\frac{2d}{3} \times \frac{2d}{3}$ each, we can see that each of such square can contain at most one point (because d is our current minimal distance). There are 18 such squares and thus within the list Y we need to compare any point p only with its 17 successors. This leads us to the same recurrence relation $T(n) = 2 \cdot T(\frac{n}{2}) + cn$ and we again obtain an algorithm running time of $O(n \log n)$.

Exercise 2: Polynomial Multiplication using FFT (10 points)

Let $p(x)$ be a polynomial of degree n and $q(x)$ a polynomial of degree m. If $m = n$ the multiplication algorithm in the lecture (using FFT) yields to $\mathcal{O}(n \log n)$ runtime. Now suppose $n > m$. How can one do the multiplication in $\mathcal{O}(n \log m)$ time?

Solution:

In the lecture we have seen that two polynomials which both have degree m can be multiplied in time $\mathcal{O}(m \log m)$. We will use this as a blackbox to devise an algorithm which multiplies two polynomials p_1 and p_2 with $deg(p_1) = n$ and $deg(p_2) = m$, $n > m$ in time $\mathcal{O}(n \log m)$. Without loss of generality we assume that m divides n. Our goal is to compute $q(x) := p_1(x) \cdot p_2(x)$.

Let $d = n/m \in \mathbb{N}$ and let $p_1(x) = \sum_{i=0}^{n} a_i x^i$ be given. Define for $j = 0, \dots, d-1$

$$p_{1,j}(x) := \begin{cases} \sum_{i=jm+1}^{(j+1)m} a_i x^{i-jm}, & j \geq 1 \\ \sum_{i=0}^{m} a_i x^i, & j = 0 \end{cases}$$

Note that $p_1(x) = \sum_{j=0}^{d-1} x^{jm} \cdot p_{1,j}$. Then the degree of each of the polynomials is at most m, i.e., $deg(p_{1,j}) \leq m$. Essentially, we represent the polynomial p_1 by d polynomials $p_{1,0}, \dots, p_{1,d-1}$ each with degree at most m. Then we multiply p_2 with each of the polynomials which can be done in time $\mathcal{O}(m \log m)$ with the algorithm which was presented in the lecture (we call it `FFTMultiplier`). Due to the distributive law we can compute q by adding up the results, which takes only $\mathcal{O}(m)$ time for each of the d polynomials.
We give pseudocode of the described procedure in Algorithm 1.

Algorithm 1: Multiplying two polynomials in $\mathcal{O}(n \log m)$

```
Input: Polynomials $p_1(x), p_2(x)$ with $n = deg(p_1(x)) > deg(p_2(x) = m$, $d = n/m \in \mathbb{N}$
Output: Returns the polynomial $p_1(x) \cdot p_2(x)$
$q \leftarrow 0$
for $j \leftarrow 1$ to $d-1$            /* Runtime:  $n/m$ iterations of the loop.        */
do
    if $j=0$ then
        $p_{1,j}(x) \leftarrow \sum_{i=0}^{m} a_i x^i$        /* Runtime:  $\mathcal{O}(m)$                    */
    else
        $p_{1,j}(x) \leftarrow \sum_{i=jm+1}^{(j+1)m} a_i x^{i-jm}$        /* Runtime:  $\mathcal{O}(m)$            */
    end
    $p_{3,j} \leftarrow$ FFTMultiplier$(p_2, p_{1,j}))$        /* Runtime:  $\mathcal{O}(m \log m)$        */
    $q(x) \leftarrow q(x) + x^{jm} \cdot p_{3,j(x)}$        /* Runtime:  $\mathcal{O}(m)$                */
end
return $q(x)$
```

The runtime of the algorithm is $\mathcal{O}(n \log m)$ as the execution of each loop can be performed in time $\mathcal{O}(m \log m)$ and there are $d = n/m$ iterations.

Exercise 3: Interval Scheduling (14 points)

In the *interval scheduling* problem, we are given a set of intervals each with starting and ending times. The goal is to select a largest possible non-overlapping set of intervals. Let us assume overlaps at the boundaries are fine. In the lecture, we have studied a greedy algorithm called *shortest available interval* for solving the problem. We have seen that the algorithm fails to optimally solve the problem.
Show that the above greedy algorithm returns a set of intervals in which the size of the set is **at least half** of the number of intervals provided by the optimal algorithm.

Solution:

Suppose I is the set of all intervals. Let S and S^* denote the sets of all intervals picked by the greedy algorithm called Greed and by an optimal algorithm, respectively.

Recap on Greed**:** At the beginning S is empty. Basically, every time Greed picks the shortest interval among all intervals available in I and adds it to S, removes the interval that is already picked and all other intervals that overlap with that from I. Greed repeats this procedure as long as I is not empty.

Observation 1. Greed guarantees that every interval in I overlaps with at least **one** interval in S.
Proof. Let us assume that there exists an interval in I that does not overlap with any interval in S. With respect to Greed, the interval must have been picked by Greed and been in S that contradicts our assumption. □

Observation 2. Greed guarantees that every interval in S overlaps with at most **two** non-overlapping intervals in I.
Proof. Again we provide a simple proof by contradiction. Suppose that there exists an interval $i \in S$ that overlaps with more than two non-overlapping intervals in I. This implies that at least one of these non-overlapping intervals that overlaps with i is shorter than i. Therefore, the shorter interval must have been picked by Greed rather than i since the Greed every time picks the shortest available interval. Hence, this contradicts our assumption. □

Claim. $|S^*| \leq 2 \cdot |S|$.
Proof. To show that the claim holds, it is sufficient to prove that

(1) every interval in S^* overlaps with at least **one** interval in S and

(2) each interval in S overlaps with at most **two** intervals in S^*.

The statements of (1) and (2) immediately follow from the Observation 1 and Observation 2, respectively, since S^* is the set of non-overlapping intervals such that $S^* \subseteq I$. □