University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
S. Faour, M. Fuchs, G. Schmid

Algorithm Theory
Sample Solution Exercise Sheet 1

Due: Wednesday, 26th of October, 2022, 11:59 pm

Exercise 1: Landau Notations *(5 Points)*

Prove or disprove the following:

(a) $n \log n \in \Theta(\log(n!))$ *(2 Points)*

(b) $n^3 - 9n^2 \in \Omega(n^3)$ *(1 Point)*

(c) $\left(\log(\sqrt{n})\right)^2 \in \Theta(\log n)$ *(2 Points)*

Sample Solution

(a) True.

- $n \log n \in \Omega(\log(n!))$:

  for all $n \geq 1$, we have that $n \log n = \log n + \log n + ... + \log n \geq \log n + \log(n-1) + ... + \log 1 = \log(n \cdot (n-1) \cdot ... \cdot 1) = \log(n!)$ (choose $c = 1$ and $n_0 = 1$).

- $n \log n \in O(\log(n!))$:

  for all $n \geq 4$, we have that $\log(n!) = \log n + \log(n-1) + ... + \log(\lceil \frac{n}{2} \rceil) + ... + \log 1 \geq \log(\lceil \frac{n}{2} \rceil) + \log(\lceil \frac{n}{2} \rceil) + ... + \log(\lceil \frac{n}{2} \rceil) + \underbrace{\log((\lceil \frac{n}{2} \rceil) - 1) + ... + \log 1}_{\geq 0} \geq \frac{n}{2} \log(\frac{n}{2}) \geq \frac{n}{2}(\log n - 1) \geq \frac{n}{2}(\log n - \frac{\log n}{2}) = \frac{1}{4} n \log n$ for all $n \geq 4$, thus $n \log n \leq 4 \cdot \log(n!)$ (choose $c = 4$ and $n_0 = 4$).

(b) True. For all $n \geq 18$, we have $n^3 \geq 18n^2$ and thus $n^3 - 9n^2 \geq n^3 - 9(\frac{n^3}{18}) = \frac{1}{2} n^3$ (choose $c = 1/2$ and $n_0 = 18$).

(c) False. We show that $\left(\log(\sqrt{n})\right)^2 \notin O(\log n)$ by contradiction. Suppose that $\left(\log(\sqrt{n})\right)^2 \in O(\log n)$, then there exists $c > 0$ and $n_0 \in \mathbb{N}^*$ such that for all $n \geq n_0$

$$\begin{array}{lccc} & \left(\log(\sqrt{n})\right)^2 & \leq & c \log n \\ \Leftrightarrow & \frac{1}{4}(\log n)^2 & \leq & c \log n \\ \Leftrightarrow & (\log n)^2 & \leq & 4c \log n \\ \Leftrightarrow & \log n & \leq & 4c \\ \Leftrightarrow & n & \leq & 2^{4c} \end{array}$$

if we choose $n := \max\{\lceil 2^{4c} \rceil + 1, n_0\} \geq n_0$, then $\lceil 2^{4c} \rceil + 1 \leq n \leq 2^{4c}$, which is a contradiction.

Exercise 2: The Majority Element *(7 Points)*

An array $A[1...n]$ is said to have a *majority element* if more than half of its elements are the same. Given an array, the goal is to design an efficient algorithm to tell whether the array has a majority element or not, and, if so to find that element.
Note that the elements of the array need not to belong to some ordered domain e.g. $\mathbb{N}$, hence there can be no comparisons of the form e.g. $A[i] < A[j]$. However, we can test equality of two elements in $O(1)$-time.

(a) Give an algorithm that solves the problem in $O(n \log n)$-time, argue correctness, and analyze its running time. *(2 Points)*

(b) Try speeding things up and give a linear time algorithm that solves the problem, argue correctness, and analyze its running time. *(5 Points)*
Hint: try reducing the size of the array to at most half via pairing up elements.

Sample Solution

(a) We will use a divide and conquer approach to solve the problem.

Algorithm idea: if $n = 1$, then return $A[1]$. If $n \geq 2$, then divide the input array A into two halves and solve each half recursively to check whether there exists an element that appears more than $n/4$ times in that half. If any of the recursive solutions return an element, then scan it in the other half to see if it appears more than $n/2$ overall. If yes, then return that element as the majority element, else return that no majority element exists in A.

Correctness: we need to show that if indeed there is an element in A that appears more than $n/2$ times then the algorithm returns it, else it returns that such an element doesn't exist. We observe two things: firstly, *"if A has a majority element, then this element will also be a majority element in either one of the two halves of A i.e. it will appear more than $n/4$ times in that half."* Secondly, *there cannot be two distinct majority elements in the same array.* Therefore combining the two observations, if A contains a majority element, then one of the recursive calls will indeed return it and the linear scan on the other half will verify that it appears more than half the size of A overall. Otherwise, the algorithm returns that there is no majority element in A (i.e. this is true even if the recursive calls return a majority element to their corresponding halves, eventually the linear scan will confirm that neither one of them is a majority element of A).

Running time: we get the recurrence relation (for the special case where n is power of 2) $T(n) = 2T(\frac{n}{2}) + O(n)$ and $T(1) = O(1)$, hence we obtain an algorithm of running time $O(n \log n)$.

(b) We will use another divide and conquer approach to solve the same problem.

Algorithm idea: First, we start from the beginning of our input array A and pair up every two consecutive elements until the end of the array (notice that at the end of this pairing up procedure if n happens to be even, then all ellements are paired up, otherwise $A[n]$ will be left unpaired and at this point we test whether it is the majority element via a linear scan of the array; if yes, then return $A[n]$ as the majority element, else we discard it and continue in the following). Also, while doing this pairing up procedure above we simultaneously do a filtering step as follows: for each pair if they are different, then discard both, else they are the same and we can keep only one of the duplicates.
Finally, we repeat doing this filtering step until *either* only one element remains, at which point we can test via a linear scan of the array if it is indeed a majority element and return it, *or* no elements remain and we can directly return that no majority element exists in A.

Correctness: The correctness of the algorithm relies on the following claim *if A has a majority element, then it remains a majority element at the end of the algorithm in the final array* (which will only be one element) and to see why this is, notice the only two cases we have at the end of the algorithm:

- Case 1: no elements remain, thus we can say that in particular no majority element exists, and by the contraposition of the claim we know that no majority element in A should also exist, which is what the algorithm returns.
- Case 2: only one element remains, thus we check if it is a majority element. If yes, we are done, else by the claim we know that A shouldn't have a majority element, which is what the algorithm outputs.

Now, notice that the claim is a direct deduction of the following observation *if an element is a majority element in the original array, it remains a majority element in the filtered array i.e. the remaining array after one filtering step.* Thus we are left to show the observation. Indeed, let X_1, X_2 define the number of appearances of the majority element x in the original array, filtered array respectively. Let Y_1, Y_2 also define the number of appearances of the remaining elements in the original array, filtered array respectively. We initially have $X_1 > Y_1$. Now, let's pay a closer look to what is happening in the filtering step: if we think of all the pairs (a, b) where $a \neq b$, then both will be discarded during the filtering process, hence, the same amount that is reduced from X_1 will also be reduced from Y_1 (if anything even more). Thus we are left with more of pairs of the form (x, x) than (a, a) where $a \neq x$. And hence when keeping one of the duplicates for each same element pair in the filtering step, we will end up with $X_2 > Y_2$ and thus X_2 is more than half the size of the filtered array (i.e. x remains a majority element in the filtered array), which proves what we want.

Running time: Notice that the size of the array is always reduced by at most half after each filtering step and since the filtering can be carried out in linear time and the size of the problem halves after each filtering step, we get the recurrence relation $T(n) \leq T(\frac{n}{2}) + O(n)$ for $n > 2$ and $T(n) = O(1)$ for $n \leq 2$, hence we obrain an algorithm of running time $O(n)$.

Exercise 3: Almost Closest Pairs of Points *(8 Points)*

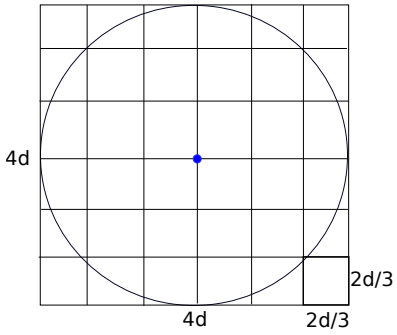
In the lecture, we discussed an $O(n \log n)$-time divide-and-conquer algorithm to determine the closest pair of points. Assume that we are not only interested in the closest pair of points, but in all pairs of points that are at distance at most twice the distance between the closest two points.

(a) How many such pairs of points can there be? It is sufficient to give your answer using big-O notation. *(3 Points)*

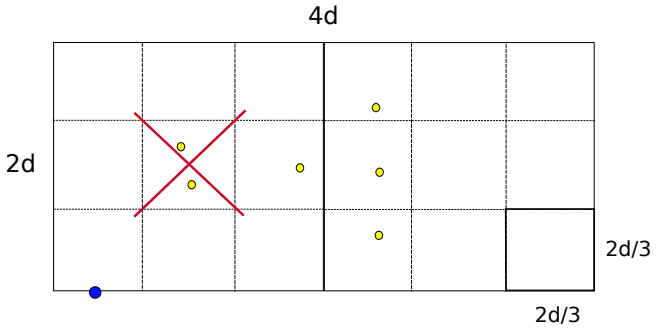
(b) Devise an algorithm that outputs a list with all pairs of points at distance at most twice the distance between the closest two points. Describe what you have to change compared to the closest pair algorithm of the lecture and analyze the running time of your algorithm. *(5 Points).*

Sample 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 $d(p, q) > 2d$ 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 $d := \min(d_\ell, d_r, d_{\ell r})$. $L' := L_\ell \cup L_r \cup L_{\ell r}$; $L := L' \setminus \{< p, q >: d(p, q) > 2d\}$; 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 (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 $T(n) \leq 2 \cdot T(\frac{n}{2}) + 17n$ and we again obtain an algorithm running time of $O(n \log n)$.