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

Algorithm Theory
Sample Solution Exercise Sheet 1

Due: Friday, 27th of October, 2023, 10:00 am

Exercise 1: Landau and Recursions *(8 Points)*

(a) Is $\left(\log(\sqrt{n})\right)^2 \in \Theta(\log n)$ true? Justify. *(2 Points)*

Consider two real square matrices X and Y each of size $n \times n$. The goal is to find the matrix multiplication of X and Y i.e. $Z = X \times Y$, which is also of size $n \times n$.

(b) Use the principle of divide and conquer to design an algorithm that finds Z in time $O(n^3)$ and analyze its running time.
Hint: try dividing the matrices into submatrices of size $\frac{n}{2} \times \frac{n}{2}$. *(3 Points)*

(c) Let $X = \begin{pmatrix} A & L \\ G & O \end{pmatrix}$ and $Y = \begin{pmatrix} T & H \\ E & O' \end{pmatrix}$, where A, L, G, O, T, H, E, O' are matrices of size $\frac{n}{2} \times \frac{n}{2}$.
Let $Z = \begin{pmatrix} ? + P_4 - P_2 + ? & ? + ? \\ ? + ? & ? + P_5 - P_3 - ? \end{pmatrix}$
Fill in Z using the following $\frac{n}{2} \times \frac{n}{2}$ matrices:
$P_1 = A(H - O')$,
$P_2 = (A + L)O'$,
$P_3 = (G + O)T$,
$P_4 = O(E - T)$,
$P_5 = (A + O)(T + O')$
$P_6 = (L - O)(E + O')$,
$P_7 = (A - G)(T + H)$

then show that time complexity for finding Z can get significantly better than in part (b).
(3 Points)

Sample Solution

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

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

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.

(b) We will assume n to be a power of two in both parts (b) and (c). The idea is to partition the $n \times n$ matrix into 4 blocks (submatrices) of size $\frac{n}{2} \times \frac{n}{2}$. Thus

$$X = \begin{pmatrix} A & L \\ G & O \end{pmatrix}, Y = \begin{pmatrix} T & H \\ E & O' \end{pmatrix}, Z = \begin{pmatrix} Z_1 & Z_2 \\ Z_3 & Z_4 \end{pmatrix},$$

where each of the submatrices are of size $\frac{n}{2} \times \frac{n}{2}$. The matrix multiplication can now be seen as

$$\begin{pmatrix} Z_1 & Z_2 \\ Z_3 & Z_4 \end{pmatrix} = \begin{pmatrix} A & L \\ G & O \end{pmatrix} \begin{pmatrix} T & H \\ E & O' \end{pmatrix} = \begin{pmatrix} AT + LE & AH + LO' \\ GT + OE & GH + OO' \end{pmatrix} \quad (1)$$

Thus, a matrix multiplication algorithm that uses divide and conquer can now: compute the smaller multiplications of pairs of half-sized submatrices recursively (i.e. 8 of those products to compute), and then for the combine step we have to do a constant number of matrix addition steps (i.e. 4 in this case). Moreover, if $n = 2$, then the half-sized submatrices can be multiplied in constant time.

Therefore if $T(n)$ is the time that it takes to multiple two matrices of size $n \times n$, then the running time of the algorithm is given by the following recursion:

$$T(n) = \begin{cases} O(1) \text{ for } n \leq n_0 \\ 8T(n/2) + 4O(n^2) = 8T(n/2) + O(n^2) \end{cases}$$

Finally, applying Master's theorem gives us that $T(n) \in O(n^3)$.

(c) We will still partition each X, Y, and Z into 4 blocks (submatrices) of size $\frac{n}{2} \times \frac{n}{2}$, but we will view the partitioned matrix Z differently, we do so by filling the gaps of Z via comparing it with (1), thus we get in the
top left: $P_5 + P_4 - P_2 + P_6$
top right: $P_1 + P_2$
bottom left: $P_3 + P_4$
bottom right: $P_1 + P5 - P3 - P7$

Using the same divide and conquer strategy as in (b) will use up less number of recursive calls and still a constant number of matrix addition steps for the combine step, hence the running time is now given by the following recursion:

$$T(n) = \begin{cases} O(1) \text{ for } n \leq n_0 \\ 7T(n/2) + O(n^2) \end{cases}$$

Finally, applying Master's theorem gives us that $T(n) \in O(n^{\log 7})$ which is approx $O(n^{2.8})$.

Exercise 2: K-th Smallest Element *(5 Points)*

Consider two arrays A and B of unique integers such that they are sorted in an increasing order and of size m and n, respectively. Let $k \leq m, n$, give an algorithm that finds the k-th smallest integer in the sorted union of the two arrays in time $O(\log k)$. Argue correctness and analyze its running time. *Remark: for simplicity, one may consider k to be a power of two.*

Sample Solution

The high level idea of the algorithm is as follows: we first notice that one can ignore all $A[i]$ and $B[j]$ for $k \leq i, j \leq m, n$ respectively, as we are looking for the k-th smallest element. For simplicity, we assume that k is a power of two.

Now, we initiate $i = k/2$ and $j = k - i$ and define a counter:=$k/4$. While the counter $\geq 1/2$, check if $A[i-1] \geq B[j-1]$, then we can ignore the second half of A, only consider the second half of B and set $i := i-$counter and $j := j+$counter and update the counter:=counter/2, and recurse (now we have the same subproblem as the original but with half the size of the problem of instance input); else, we ignore the second half of B and only consider the second half of A and set $i := i+$counter and $j := j-$counter and update the counter:=counter/2, and recurse.
At the end of the while loop, we are left with one element in A and one element in B, we compare them and output the smaller one as the k-th element in the sorted union of the two arrays A and B.

The rough intuition of the correctness is that in every recursive step $r = 1, ..., \log k$ (inside the while loop), we are identifying $k/2^r$ new distinct elements (all of them coming from either A or B) that we can show that they are amongst the first $k-1$ elements in the sorted array of $A \cup B$. Thus at the end of the while loop, we would have: identified all the $k/2 + k/4 + ... + 2 + 1 = k - 1$ (cf geometric series) elements of the first $k-1$ elements in the sorted array of $A \cup B$, and thus we would have also figured out the *indices* i and j where $i + j = k - 2$ such that $max\{A[i], B[j]\}$ is the $k-1$-th smallest element in the sorted array of $A \cup B$. Moreover, since at the end of the while loop, we are left with one element in A i.e. $A[i+1]$ and one element in B i.e. $B[j+1]$, the smaller of the two will be our desired output i.e. the k-th smallest element in the sorted array of $A \cup B$.
Indeed, one can show that for every recursive step $r \in \{1, .., \log k\}$ of the while loop, we can identify $k/2^r$ new distinct elements (all of them coming from either A or B) that we can show that they are amongst the first $k-1$ elements in the sorted array of $A \cup B$ and another $k/2^r$ new distinct elements (all of them coming from either A or B) that we can safely ignore and show that none can be the k-the smallest element in the sorted array of $A \cup B$. Indeed, for $r = 1$, we look at the two medians $A[\frac{k}{2} - 1]$ and $B[\frac{k}{2} - 1]$ and suppose w.l.o.g. $A[\frac{k}{2} - 1] < B[\frac{k}{2} - 1]$, the algorithm can safely ignore the second half of B, the reason is that in the sorted array of $A \cup B$, element $B[\frac{k}{2} - 1]$ can only appear after all elements $A[i]$ for all $i \leq \frac{k}{2} - 1$ appear (thus after at least $k-1$ elements appearing). So for all $j > \frac{k}{2} - 1$, none of the $B[j]$ elements can be the k-th smallest element in the sorted array of $A \cup B$, hence we can safely ignore them. Now, we can deduce that the first $k/2$ elements of A i.e. elements $A[i]$ for all $i \leq \frac{k}{2} - 1$, must be amongst the first $k-1$ elements in the sorted array of $A \cup B$, hence we can ignore the first half of A and focus on the second half of A. Then one can continue the argument for the rest of r-cases similarly or by induction on r .

The running time is $O(2\log(k)) = O(\log k)$, which can be seen as as if we are doing binary search on both arrays.

Exercise 3: Triangle with Shortest Perimeter *(7 Points)*

Let $P = \{(x_i, y_i) \in \mathbb{R}^2 \mid i = 1, \ldots, n\}$ be a set of n points in $\mathbb{R}^2$. Given three distinct points $a, b, c \in P$ they span a triangle with *perimeter*

$$peri(a, b, c) = d(a, b) + d(b, c) + d(a, c),$$

where $d(\cdot, \cdot)$ determines the euclidean distance of two points.

(a) Assume e to be the shortest perimeter of a triangle formed by 3 points from P, how many triangles (formed by 3 points from P) can there be in a rectangle of size $e/5 \times e/4$? *(2 Points)*

(b) Describe a $\mathcal{O}(n \cdot \log(n))$ algorithm which finds the smallest triangle perimeter in P. Argue shortly the correctness of your algorithm and its running time. *(5 Points)*

Sample Solution

(a) None, since in any rectangular of size $e/5 \times e/4$ there cannot be more than two points either from P_l or P_r since the minimum perimeter of both left and right sets is e. Moreover, if we assume there

are 3 points from P inside a rectangular that exists fully on the left or right of the median(P) side such that it is of size $e/5 \times e/4$, then the perimeter of these three points will be $< e$ (note that the longest distance between any two points in a rectangular with size $e/5 \times e/4$ is less than $e/3$).

(b) This problem is similar to the problem of finding the closest pair in a set of points as discussed in the lecture. Hence, we **adapt** that algorithm to solve the problem of finding the shortest triangle perimeter. To do this, only the **combine** part of given algorithm in the lecture needs to be modified to solve this problem.

Algorithm 1 ShortestPeri(P)

Input: A set P of n points in $\mathbb{R}^2$ sorted by x-coordinate
Output: Shortest triangle perimeter
if $|P| < 3$ **then**
  Sort P according to y-coordinate **return** ∞ and sorted P
$x_0 \leftarrow median(P)$
Divide the points into the two sets P_l and P_r (almost equal in size) by their x-coordinate
▷ let x_p be the x-coordinate of p
▷ division such that: $x_p \leq x_0$ for all $p \in P_l$ and $x_p \geq x_0$ for all $p \in P_r$
$e_l \leftarrow ShortestPeri(P_l)$
$e_r \leftarrow ShortestPeri(P_r)$
Merge sorted P_l and P_r to obtain sorted P by y-coordinate
▷ similar to Mergesort
$e \leftarrow \min\{e_l, e_r\}$
$S := \{p \in P : |x_0 - x_p| \leq e/2\}$
$e' \leftarrow minLR(P_l \cap S, P_r \cap S, e)$
return $\min\{e, e'\}$ and P sorted by y-coordinates

Algorithm 2 $minLR(L, R, e)$

$e' \leftarrow \infty$
for z in$L$ in increasing order by y-coordinate **do**
  $M_z = \{p \in L \cup R || y_z - y_p| \leq \frac{e}{2}\}$ ▷ y_p is the y-coordinate of point p
  for triangle $(z, p_2, p_3) \in M_z$ **do**
    $e' \leftarrow \min\{e', peri(z, p_2, p_3)\}$
return e';

Implementation and Running time

Pre-Sorting

The algorithm assumes that the input set P is sorted according to its x-coordinates. This can be done in a precomputation in $\mathcal{O}(n \cdot \log(n))$.

Divide

x_0 can be found in $\mathcal{O}(n)$ and the set P can be split into P_l and P_r in $\mathcal{O}(n)$ as well. The division step can be implemented such that P_l and P_r are sorted according to their x-coordinate as well.

Conquer/Correctness

After dividing the points into two sets with equal sizes, there can be **three cases** for those three points that span the triangle with shortest perimeter:

(a) they are **all** in P_l,

(b) they are **all** in P_r,

(c) **at least one** of them is in P_l and **at least one** of them is in P_r.

The minimum perimeter of the first two cases is e and obtained by the recursion. The third case is taken care of in $minLR(P_l \cap S, P_r \cap S, e)$: It is sufficient to check triangles (p_1, p_2, p_3) such that $p_1 \in P_l \cap S$, $p_2 \in P_r \cap S$ and $p_3 \in (P_l \cup P_r) \cap S$. (otherwise, there should be at least one point outside of S, w.l.o.g. let $p_1 \notin S$. Then $d(p_1, p_2) > \frac{e}{2}$ and $d(p_2, p_3) + d(p_3, p_1) > d(p_1, p_2) > \frac{e}{2}$ (triangle inequality for $d(\cdot, \cdot)$)). Hence $peri(p_1, p_2, p_3)$ would be greater than e and does not have to be considered). Furthermore a fixed $z = p_1 \in P_l$ the perimeter $peri(z, p_2, p_3)$ can only be less than e if the y-coordinates of z and $p_i, i = 1, 2$ do not differ by more than $\frac{e}{2}$. Hence it is correct to only consider triangles which z forms with points from M_z.

Sorting according to y-coordinate:

As in Mergesort the merge of the sorted sets P_l and P_r can be done in $\mathcal{O}(n)$.

Computing e' with $minLR()$:

A fixed point z cannot form a triangle with perimeter less than e with points outside of M_z (a similar argument as when we restricted investigation to the set S). (**The algorithm uses that L and R are sorted according to y-coordinates to determine M_z in time $\mathcal{O}(n)$**).

The outer loop in $minLR()$ is run through at most $|L|$ times. The crucial point now is that $|M_z| \leq C$ for some constant C. Thus the inner loop is run through at most C times, i.e., the running time of $minLR()$ is in $\mathcal{O}(|L|)$.

Proof of $|M_z| \leq 24$ for all $z \in P_l$:

We divide M_z into 10 rectangles of size $e/5 \times e/4$ each. In any rectangular of size $e/5 \times e/4$ that is fully contained in the left or right side of the median(P), there cannot be more than two points either from P_l or P_r a shown in part (a) of the exercise. Now there's a small subtilty that for the two $e/5 \times e/4$-sized rectangles in the middle (the ones where the median cuts them), each one of them will be divided in two equal smaller rectangles of size $e/10 \times e/2$ where one is fully contained in the left and the other in the right, and by the same reasoning as in part (a) of this exercise the smaller rectangles can't have more than 2 nodes from P inside them, thus $|M_z| \leq 24$ and because $|M_z| \leq 24$ we need to check only $\binom{24}{2}$ triangles in the inner most for-loop.

Combining the running times of dividing ($\mathcal{O}(n)$), merge for y-coordinate sorting ($\mathcal{O}(n)$) and conquering/computing e' ($\mathcal{O}(n)$) we obtain that the recurrence relation is,

$$T(n) = 2T(n/2) + \mathcal{O}(n), \quad T(1) = 1.$$

Using Master theorem we know that $T(n) \in \mathcal{O}(n \cdot \log(n))$. Together with the sorting part, the total running time for the Algorithm 1 is $\mathcal{O}(n \cdot \log(n))$.