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

Algorithms and Datastructures
Summer Term 2024
Sample Solution Exercise Sheet 6

Due: Wednesday, June 5th, 2pm

Exercise 1: Minimum Distance between Values *(10 Points)*

(a) Given an array A that contains n integers. Describe an algorithm that finds indices $i \neq j$ such that $|A[i] - A[j]|$ is minimal among all indices. In other words, the algorithm should compute the entries of A that have the smallest distance. Argue the correctness of your algorithm and show that it runs in time $o(n^2)$. *(5 Points)*

(b) Now, assume that the n numbers from a) are given in a binary search tree B (instead of in an array). Again, give an algorithm that finds the two tree nodes $u \neq v$ such that $|\mathrm{val}(v) - \mathrm{val}(u)|$ is minimal. Show the correctness and explain why the runtime is on $O(n)$. *(5 Points)*

Sample Solution

(a) Algorithm: We first sort the array in time $O(n \log n)$ (e.g. MergeSort). Then we iterate over the sorted array and always store the k for that $|A[k+1] - A[k]|$ is minimal. Note that by the tasks' definition we have to return the original indices i and j. To archieve that, we modify the initial array before we sort it, i.e., we replace every element $A[k]$ by the tuple $(A[k], k)$. Sorting by the first tuple entry let the algorithm work as before, but we have the original indices stored as well.

Correctness: In every sorted array A we have for all k that $... \leq A[k-2] \leq A[k-1] \leq A[k] \leq A[k+1] \leq A[k+2] \leq ....$ Thus, the largest element that is smaller than $A[k]$ is $A[k-1]$ (as otherwise the array wouldn't be sorted correctly) and with the same reasoning the smallest element larger than $A[k]$ is $A[k+1]$. We therefore do not need to compare $A[k]$ with all entries in the array, just with its two neighbors. Since our algorithm compares every neighbor in the sorted array we are guaranteed to find the minimum distance.
Runtime: Sorting takes $O(n \log n)$ time. The iteration and comparisions that follow afterwards can be done in linear time. Thus, the overall runtime is in $O(n \log n) \subset o(n^2)$.

(b) Here we use the In-Order traversal in binary tress. This one always returns the elements of the tree in sorted order. Thus, we can act like in above's task.
Correctness: Follows from the fact that In-Order produces a sorted output and the remaining argument is as in a).
Runtime: The traversal takes $\Theta(n)$ time. Since the comparisons (like in a)) also take linear time the statement of the task is shown.

Exercise 2: *(10 Points)*

Again, given a binary tree B containing n integers. For a path $P = \{r, v_1, v_2, \ldots, b\}$, from the root node r to some leaf b, we define its weight by $w(P) = \sum_{v \in P} \mathrm{val}(v)$. Describe an algorithm that finds the *heaviest* path from the root node to some leaf in B, i.e., the path P that maximizes $w(P)$ for all root-to-leaf path. State that the runtime is in $O(n)$. *(10 Points).*

Sample Solution

We use the Post-Order traversal. Whenever a node v is visited, both his chilred already got visited. Whenever we visit a node v, we compute the heaviest path rooted at v. The weight of v is as follows:

$$\mathrm{val}(v) + \max_{u \text{ child of } v} \{w(P_u)\}$$

Correctness: We will proof that every node v knows the heaviest path rooted at v ending at some leaf. For that, we use induction over the height of the tree rooted at v. When the height is 0, i.e., the tree has just one node, the heaviest path has weight $val(v)$. Now, assume v has at least one child. Since we traverse in Poat-order, all children are already visited. By induction hypothesis, we know the heaviest path rooted at the childrens of v (since the trees rooted at the children are of lower height). Thus, we can compute the heaviest path of v by taking the heavier child and add $val(v)$, what indeed is done by our algorithm.
When the algorithm visits root r, we also know the heaviest path in the whole binary tree.
Runtime: The traverls takes linear in n time. While checking the heaver path of the children simply takes a constant number of checks (since there are at most 2 children). Thus, we overall have a runtime of $O(n)$.