Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Johannes Lengler, David Steurer
Kasper Lindberg, Lucas Slot, Hongjie Chen, Manuel Wiedmer

28 October 2024

Algorithms & Data Structures — Exercise sheet 6 — HS 24

The solutions for this sheet are submitted on Moodle until 03 November 2024, 23:59.

Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

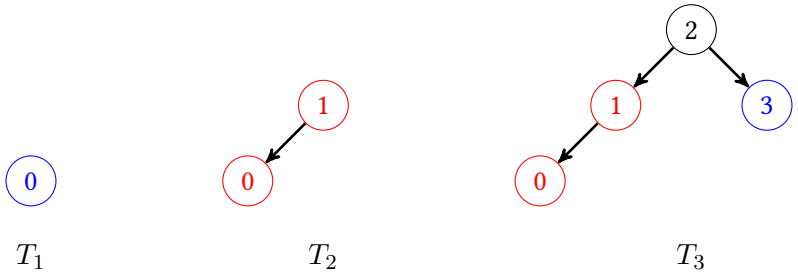
You can use results from previous parts without solving those parts.

Data structures.

Exercise 6.1 *Fibonacci trees* (1 point).

For $k \in \mathbb{N}$, we define the *Fibonacci tree* T_k recursively:

- The trees T_1 and T_2 are binary search trees with 1 and 2 nodes respectively, as depicted below.
- For $k \geq 3$, the tree T_k is constructed as follows. We start with a root node with key $\mathrm{Fib}(k+1)-1$. Then, we add as the left subtree of this root node the tree T_{k-1}. Finally, we add as a right subtree the tree T_{k-2}, but with all keys increased by $\mathrm{Fib}(k+1)$.



Note: To achieve full points for this exercise, it is enough to submit parts **(e)** and **(f)**. The other parts are **not worth any points**. When solving a part, you are allowed to use any earlier parts, even if you did not solve them.

(a)* Show that the tree T_k is a binary search tree for all $k \in \mathbb{N}$.

Hint: *First show that the keys in T_k are between 0 and $\mathrm{Fib}(k+2)-2$.*

(b) Show that the T_k has exactly $\mathrm{Fib}(k+2)-1$ nodes for all $k \in \mathbb{N}$.

(c) Show that the tree T_k has height exactly k for all $k \in \mathbb{N}$.

(d) Show that the tree T_k is an AVL tree for all $k \in \mathbb{N}$.

(e) Show that, in fact, for any $k \in \mathbb{N}$, and any node u in T_k that is not a leaf, we have $|h_l(u)-h_r(u)| = 1$ where $h_l(u)$ is the height of the left subtree of u and $h_r(u)$ is the height of the right subtree of u.

Hint: *Use induction. You will need to show multiple base cases. Be careful how you formulate the induction hypothesis.*

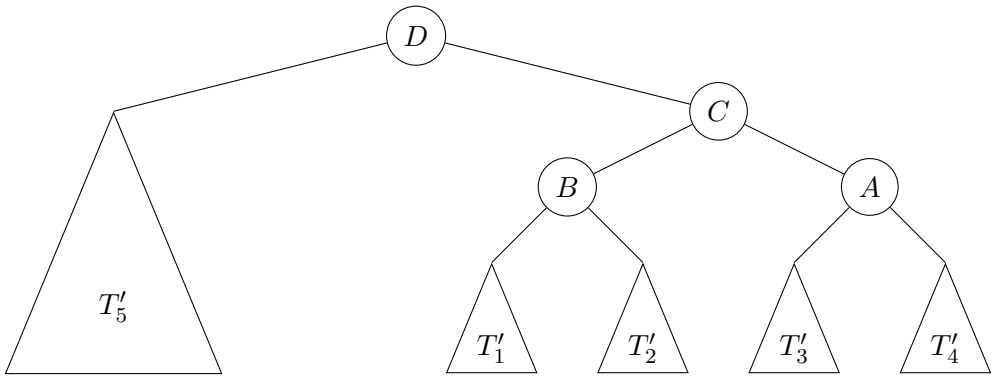
(f) Let $k \geq 3$ and odd. Show that the tree T_k contains a leaf at depth exactly $(k-1)/2$. (Here, the depth of a node is defined as the number of predecessors it has in the tree).

Hint: *Draw the tree T_5. Identify which leaf is at depth $(5-1)/2 = 2$.*

(g)* We call a leaf u in an AVL tree B *critical* if B is no longer an AVL tree after removing u. Otherwise, we call it *non-critical*. Show that, for any $k \geq 2$, the AVL tree T_k has exactly 1 non-critical leaf (and so all other leaves are critical).

Exercise 6.2 *AVL Deduction.*

Let T be an AVL tree and suppose we performed one node insertion to T to get T' (we haven't rebalanced yet). Only partial information about T' is given. The graph of T' looks like:



where $T_1', T_2', \ldots, T_5'$ are all subtrees whose nodes satisfy the AVL condition. Note that their leaves are not necessarily at the same level.

The left and right subtree heights of B are both 1 and the left and right subtree heights of A are 2 and 3 respectively. Furthermore assume that D does not satisfy the AVL condition.

(a) What are the heights of the left and right subtrees of C?

(b) Which subtree was the node inserted in?

(c) What are the heights of the left and right subtrees of D?

(d) Draw the tree after the necessary single or double rotation to rebalance T' and restore the AVL condition for all nodes. Also note the new left and right subtree heights for nodes A, B, C and D.

Dynamic programming.

Exercise 6.3 *Introduction to dynamic programming* **(1 point)**.

Consider the recurrence

$$\begin{aligned} A_1 &= 1 \\ A_2 &= 2 \\ A_{2n+1} &= \frac{1}{2}(A_{2n} + A_{2n-1}) \\ A_{2n+2} &= 2\left(\frac{1}{A_{2n}} + \frac{1}{A_{2n-1}}\right)^{-1} \text{ for } n \geq 1. \end{aligned}$$

(a) Provide a recursive function (using pseudo code) that computes A_n for $n \in \mathbb{N}$. You do not have to argue correctness.

(b) Lower bound the run time of your recursion from (a) by $\Omega(C^n)$ for some constant $C > 1$.

(c) Improve the run time of your algorithm using memoization. Provide pseudo code of the improved algorithm and analyze its run time.

(d) Compute A_n using bottom-up dynamic programming and state the run time of your algorithm. In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps? Describe the calculation order in pseudocode.
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

(e)* Assume that the sequence of A_n converges to some positive number in the limit. Prove that $\lim_{n\to\infty} A_n = \sqrt{2}$.

Hint: *Expand out the product* $A_{2n+2} \cdot A_{2n+1}$.

Exercise 6.4 *Maximum almost subarray sum* **(1 point)**.

The maximum subarray sum problem from the lecture asks for the sum of the largest *contiguous* subarray in a given array. The maximum almost subarray sum problem asks instead for the sum of the largest contiguous subarray with one element possibly missing from inside this subarray.

We consider the following array of length $n = 10$.

$$A[1..n] = [3, 2, -2, 1, -2, -3, 4, 1, -3, 4]$$

In this exercise we'll take the tools from the solution of maximum subarray sum and extend it to two different methods of computing this maximum almost subarray sum.

(a) The dynamic programming solution for maximum subarray sum given in the lecture revolves around the numbers

$$R[k] := \text{maximum subarray sum of } A[1..k] \text{ which includes index } k$$

for $0 \le k \le n$, where we define $R[0] = 0$. Then to get the maximum subarray sum we output $\max_k R[k]$.

In lecture we saw that this array R satisfies the recursive relation

$$R[0] = 0$$
$$R[k] = \max\{A[k], A[k] + R[k-1]\} \text{ for } 1 \le k \le n$$

Compute the array for $1 \le k \le n = 10$.

(b) Define the following modification of R

$$R'[k] := \text{maximum almost subarray sum of } A[1..k] \text{ which includes index } k \text{ and skips an entry}^1,$$

where we define $R'[0] = R'[1] = R'[2] = 0$ (because for $k \in \{0, 1, 2\}$, there is no almost subarray of $A[1..k]$ that skips an entry). Assume that the following recursive relation is correct

$$\begin{aligned} R'[3] &= A[3] + R[1] \\ R'[k] &= \max\{A[k] + R'[k-1], A[k] + R[k-2]\} \text{ for } 4 \leq k \leq n. \end{aligned}$$

Compute $R'[k]$ for $1 \leq k \leq n = 10$. How can the final solution for the maximum almost subarray sum of A be extracted from R and R'? Write down the formula as a function of the entries of these two arrays.

(c) Similar to the array R we can define

$$S[k] := \text{maximum subarray sum of } A[k..n] \text{ which includes index } k$$

Assume the following recursive relation is correct

$$\begin{aligned} S[n+1] &= 0 \\ S[k] &= \max\{A[k], A[k] + S[k+1]\} \text{ for } 1 \leq k \leq n \end{aligned}$$

Compute $S[k]$ for $1 \leq k \leq n$. How can the final solution for the maximum almost subarray sum of A be extracted from R and S? Write down the formula as a function of the entries of these two arrays.

(d)* Give an argument for why the recursive relations for $R'[k]$ and $S[k]$ in parts (b) and (c) are correct.

Exercise 6.5 *Longest common subsequence and edit distance.*

In this exercise, we are going to consider two examples of problems that have been discussed in the lecture. In the following, we are given two arrays, A of length n, and B of length m, and we want to find the "change" between A and B using two different metrics.

(a) We are going to look at the problem of finding the longest common subsequence of A and B. The subsequence does not have to be contiguous. For example, if $A = [1, 8, 5, 2, 3, 4]$ and $B = [8, 2, 5, 1, 9, 3]$, a longest common subsequence is $8, 5, 3$ and its length is 3. Notice that $8, 2, 3$ is another longest common subsequence.

(b) We are looking at the problem of determining the edit distance A and B, i.e., the smallest number of operations in "change", "insert" and "remove" that are needed to transform one array into the other. If for example $A =$ ["A", "N", "D"] and $B =$ ["A", "R", "E"], then the edit distance is 2 since we can perform 2 "change" operations to transform A to B but no less than 2 operations work for transforming A into B.

The algorithms for computing the longest common subsequence and for computing the edit distance that have been discussed in the lecture are the subject of the following subtasks.

[1] Skipping an entry here means that we take a contiguous subarray and remove one element from the inside (i.e. not the first or last entry) of this subarray.

(a) Given are the two arrays

$$A = [7, 6, 3, 2, 8, 4, 5, 1]$$

and

$$B = [3, 9, 10, 8, 7, 1, 2, 6, 4, 5].$$

Use the dynamic programming algorithm from the lecture to find the length of a longest common subsequence and the subsequence itself. Show all necessary tables and information you used to obtain the solution.

(b) Define the arrays

$$A = [\text{“S”}, \text{“O”}, \text{“R”}, \text{“T”}]$$

and

$$B = [\text{“S”}, \text{“E”}, \text{“A”}, \text{“R”}, \text{“C”}, \text{“H”}].$$

Use the dynamic programming algorithm from the lecture to find the edit distance between these arrays. Also determine which operations one needs to achieve this number of operations. Show all necessary tables and information you used to obtain the solution.