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.

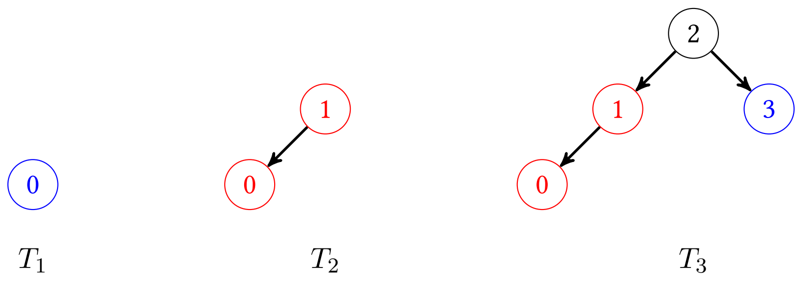
The solutions are intended to help you understand how to solve the exercises and are thus more detailed than what would be expected at the exam. All parts that contain explanation that you would not need to include in an exam are in grey.

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 $\text{Fib}(k+1)-1$. 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 $\text{Fib}(k+1)$.



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* $\text{Fib}(k+2)-2$.

Solution:

We first show the statement in the hint using induction.

Base case: From the figure above, we see that the statement holds for T_1 and T_2.

Induction hypothesis: Let $p \in \mathbb{N}$, with $p \geq 2$ and suppose the statement holds for p and for $p-1$.

Induction step: Consider the tree T_{p+1}. Let R be its root, which has key $\mathrm{Fib}(p+2)-1$. By the induction hypothesis, the keys in the left subtree of R (which is T_p) are between 0 and $\mathrm{Fib}(p+2)-2$. Furthermore, the keys in the right subtree of R (which is T_{p-1}, but incremented by $\mathrm{Fib}(p+2)$) are between $\mathrm{Fib}(p+2)$ and $\mathrm{Fib}(p+2)+(\mathrm{Fib}(p+1)-2)=\mathrm{Fib}(p+3)-2$. Thus, the smallest key in T_{p+1} is (at least) 0 and the largest key is (at most) $\mathrm{Fib}(p+3)-2=\mathrm{Fib}((p+1)+2)-2$.

By the principle of mathematical induction, we can conclude the statement of the hint holds for all $k \in \mathbb{N}$.

We now prove that T_k is a binary search tree for all $k \in \mathbb{N}$, again by induction.

Base case: From the figure above, we see that the statement holds for T_1 and T_2.

Induction hypothesis: Let $p \in \mathbb{N}$, with $p \geq 2$ and suppose the statement holds for p and for $p-1$.

Induction step: Consider the tree T_{p+1}. Let R be its root, which has key $\mathrm{Fib}(p+2)-1$. By construction, the left subtree of R is T_p. Therefore, by the hint, all keys in the left subtree are smaller than $\mathrm{Fib}(p+2)-2$, which is strictly smaller than the key of R. On the other hand, the right subtree of R is T_{p-1}, but with all keys increased by $\mathrm{Fib}(p+2) > \mathrm{Fib}(p+2)-1$. By the hint, all keys were originally at least 0, so now they are strictly bigger than the key of R. So the binary search tree condition is satisfied at R. For all nodes in the left subtree of R, the keys are exactly as they were in T_p, and by the induction hypothesis, all these nodes thus satisfy the binary search tree condition. For all nodes in the right subtree of R, the keys are exactly as they were in T_{p-1}, only increased by a constant. The constant increase does not affect the binary search tree condition, and so we can again apply the induction hypothesis.

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

Solution:

We show this using induction.

Base case: From the figure above, we see that the statement holds for T_1 and T_2.

Induction hypothesis: Let $p \in \mathbb{N}$, with $p \geq 2$ and suppose the statement holds for p and for $p-1$.

Induction step: The number of nodes in T_{p+1} is equal to the number of nodes in T_p, plus the number of nodes in T_{p-1}, plus one, by construction. Using the induction hypothesis, this equals

$$(\mathrm{Fib}(p+2)-1)+(\mathrm{Fib}(p+1)-1)+1=\mathrm{Fib}(p+3)-1=\mathrm{Fib}((p+1)+2)-1.$$

By the principle of mathematical induction, T_k has $\mathrm{Fib}(k+2)-1$ nodes for all $k \in \mathbb{N}$.

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

Solution:

We show this using induction.

Base case: From the figure above, we see that the statement holds for T_1 and T_2.

Induction hypothesis: Let $p \in \mathbb{N}$, with $p \geq 2$ and suppose the statement holds for p and for $p-1$.

Induction step: By construction, the height of T_{p+1} is one plus the maximum of the heights of T_p and T_{p-1}. Using the induction hypothesis, this is

$$1+\max\{p, p-1\}=1+p.$$

Thus, by the principle of mathematical induction, the tree T_k has exactly height k for any $k \in \mathbb{N}$.

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

Solution:

Part (a) shows that T_k is a binary search tree. To show that T_k satisfies the AVL condition, we can use the same proof as below in (e) (which shows something even stronger).

(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.*

Solution:

We show this using induction.

Base case: From the figure above, we see that the statement holds for T_1 and T_2.

Induction hypothesis: Let $p \in \mathbb{N}$, with $p \geq 2$ and suppose the statement holds for p and for $p-1$.

Induction step: Consider the tree T_{p+1}. Let R be its root. By construction, the left subtree of R is T_p, and the right subtree is T_{p-1} (but with different keys). By part (c) we therefore have $h_l(R) = p$ and $h_r(R) = p-1$, meaning $|h_l(R) - h_r(R)| = 1$. Now let u be any other node in T_{p+1} that is not a leaf. Then u corresponds to a node of T_p, or of T_{p-1}, in the sense that its left and right subtree in T_{p+1} are exactly as they were in T_p or T_{p-1}. By the induction hypothesis, we thus have $|h_l(u) - h_r(u)| = 1$. So the statement holds for $p+1$.

By the principle of mathematical induction, we get that for any $k \in \mathbb{N}$ and any node u in T_k that is not a leaf we have $|h_l(u) - h_r(u)| = 1$.

(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* $(5-1)/2 = 2$.

Solution:

For $k = 3$, we see the statement holds from the figure above. Let $k \geq 5$ and odd. Let R_0 be the root of T_k. Then the right subtree of R_0 is equal to T_{k-2} (but with different keys). In turn, the right subtree of the root R_1 of the right subtree of R_0 is equal to T_{k-4}, and so forth. Therefore, the node $R_{(k-1)/2}$ that is reached by 'taking the right child' $(k-1)/2$ times is a leaf (its right and left subtree are equal to those of T_1, i.e., empty). Furthermore, it has depth $(k-1)/2$ (its predecessors are $R_0, R_1, \ldots R_{(k-3)/2}$).

(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).

Solution:

In the lecture, we have seen that any AVL tree of height h must have at least $\mathrm{Fib}(h+2)-1$ nodes. By parts (b), (c), (d), we know that T_p is an AVL tree of height p with exactly $\mathrm{Fib}(p+2)-1$ nodes. Therefore, if we remove any leaf from T_p, it can no longer be an AVL tree, *unless* by removing this leaf we also decrease the height of the tree. The latter only happens when the leaf we remove is in the bottom layer of the tree, i.e. at maximum depth $p-1$. It turns out that there is precisely one such leaf. To show this, note that, by construction, and using part (c), any leaf at maximum depth in T_p must be in the left subtree of the root (which is equal to T_{p-1}), and must be at maximum depth

$p-2$ in that subtree. As T_1 and T_2 both have a single leaf at maximum depth, the statement follows by induction.

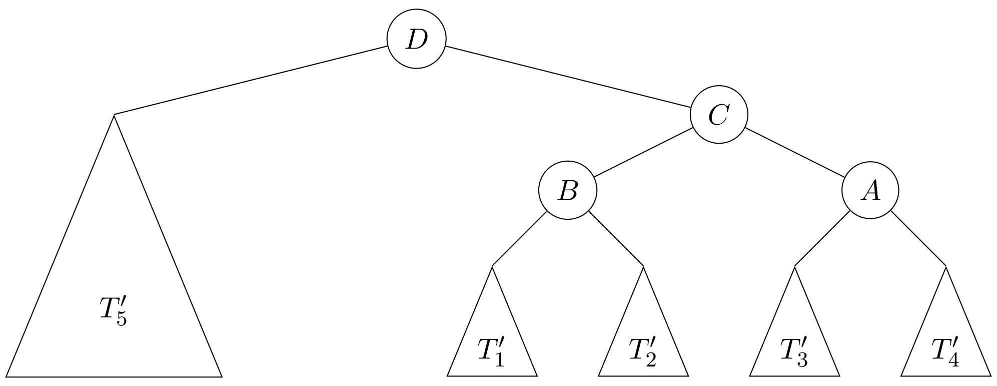
As a further exercise, you could try to formulate the final few sentences of the proof above as a formal induction proof.

Guidelines for correction:

Award 1/2 point for part (e) and 1/2 point for part (f). In part (e), it is important that two base cases are mentioned, and that the induction hypothesis applies to both p and $p-1$. It is not important to mention that the keys of the right subtree are changed. In part (f), the important part is to have some sort of 'telescoping' argument. Do not subtract points for 'out of bounds' errors in the reasoning (e.g., talking about T_{k-4} without specifying that $k > 4$ etc.). If part (f) is solved using induction instead, do not subtract points for not including p and $p-1$ in the induction hypothesis if points were already subtracted for the same mistake in part (e).

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?

Solution:

Since B has height 1 in both subtrees, C's left subtree must have height 2. A's largest subtree has height 3, so C's right subtree must have height 4.

(b) Which subtree was the node inserted in?

Solution:

The subtrees all satisfy the AVL condition and C is the highest node to not satisfy the condition. Since before insertion all nodes satisfied the AVL condition, we know it must have been the right subtree of A, T'_4, which got the node insertion because the right subtree height of A is what leads to C not satisfying the AVL condition.

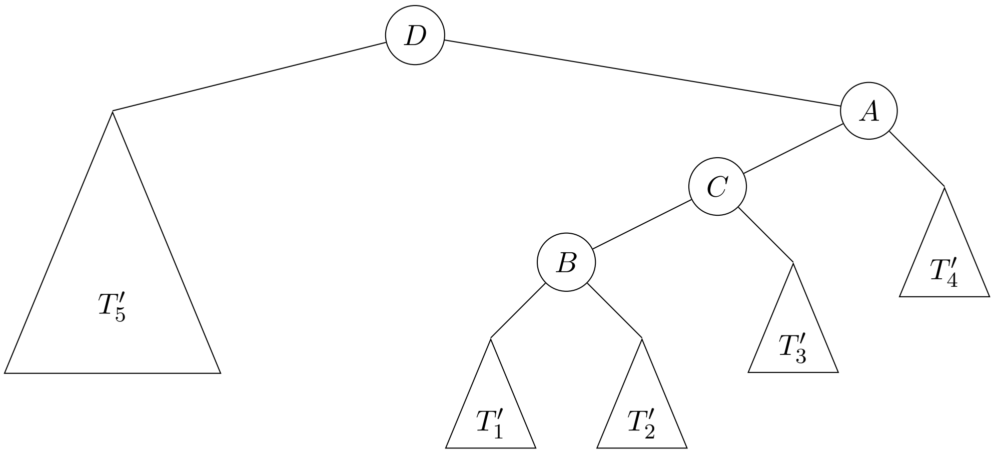
(c) What are the heights of the left and right subtrees of D?

Solution:

The right subtree height of D is 5, based on our deduction of C's subtree heights. Since T was an AVL tree and D had right subtree height 4 then, but in T', D does not satisfy the AVL condition, it must be that the left subtree has height 3.

(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.

Solution:



For A, left: 3, right: 3.
For B, left: 1, right: 1.
For C, left: 2, right: 2.
For D, left: 3, right: 4.

Dynamic programming.

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

Consider the recurrence

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

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

Solution:

Algorithm 1 $A(n)$

```
if $n \leq 2$ then
    return $n$
else if $n$ is odd then
    return $\frac{1}{2}(A(n-1) + A(n-2))$
else
    return $2\left(\frac{1}{A(n-2)} + \frac{1}{A(n-3)}\right)^{-1}$
```

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

Solution:

The number $T(n)$ of operations for a call $A(n)$ is given by the recurrence

$$T(1) = T(2) = 1,$$

if n is odd then

$$T(n) = T(n-1) + T(n-2) + d,$$

and if n is even then

$$T(n) = T(n-2) + T(n-3) + d',$$

where d and d' are positive constants.
We present some intuition on how to come up with the proof. Assume that T is monotonously increasing, then for $n \geq 3$ and say n is odd, we have

$$\begin{aligned}
T(n) &\geq T(n-1) + T(n-2) \\
&\geq (T(n-3) + T(n-4)) + (T(n-3) + T(n-4)) \\
&= 2(T(n-3) + T(n-4)) \\
&\geq 2((T(n-5) + T(n-6)) + (T(n-5) + T(n-6))) \\
&= 4(T(n-5) + T(n-6)) \\
\ldots &\geq 2^k(T(n-(2k+1)) + T(n-(2k+2)))
\end{aligned}$$

So when we solve for $n - (2k+1) = 2$ we get $k = (n-3)/2$. Note that $T(2) = T(1) = 1$ as a our base cases so we can simplify the last line to $2^{(n-3)/2} \cdot 2 = \frac{1}{2} \cdot 2^{(n+1)/2}$.

A similar expansion can be done for n even, getting

$$T(n) \geq 2^k(T(n-(2k+2))) + T(n-(2k+3)))$$

and again $n-(2k+2) = 2$ means we get $k = (n-4)/2$ so the last line will be $2^{(n-4)/2} \cdot 2 = \frac{1}{2} \cdot 2^{n/2}$.

Then to make these two cases agree, we have that $(n+1)/2 \geq \lceil n/2 \rceil$ and $n/2 \geq \lceil n/2 \rceil$ for both even and odd n. Thus incorporating this into both cases we have $T(n) \geq \frac{1}{2} \cdot 2^{\lceil n/2 \rceil}$. We show that $T(n) \geq \frac{1}{2} \cdot 2^{\lceil n/2 \rceil}$ by induction.

- **Base Case.**
  For $n \in \mathbb{N}$ with $n \leq 2$, we have $T(n) = 1 = \frac{1}{2} \cdot 2^{\lceil n/2 \rceil}$ since $\lceil n/2 \rceil = 1$ in this range. We technically have two base cases here to deal with $n = 1$ and $n = 2$ though both take 1 step to execute so the math can be simplified. For the rest of the proof, we will continue via two cases: even and odd. This is due to even and odd A_ns being defined differently. However, as long as the inductive hypothesis can be satisfied for either case, the proof still works.

- **Induction Hypothesis.**
  Assume that for some integer $k \geq 3$ the statement holds for all $k' < k$.
- **Induction Step.**
  We proceed by cases. If k is odd then

$$\begin{aligned} T(k) &= T(k-1) + T(k-2) + d \\ &\geq \frac{1}{2} \cdot 2^{\lceil (k-1)/2 \rceil} + \frac{1}{2} \cdot 2^{\lceil (k-2)/2 \rceil} \\ &= \frac{1}{2} \cdot 2^{(k-1)/2} + \frac{1}{2} \cdot 2^{(k-1)/2} \\ &= \frac{1}{2} \cdot 2^{(k-1)/2+1} \\ &= \frac{1}{2} \cdot 2^{(k+1)/2} \\ &= \frac{1}{2} \cdot 2^{\lceil k/2 \rceil}. \end{aligned}$$

  If k is even then a very similar manipulation gives

$$\begin{aligned} T(k) &= T(k-2) + T(k-3) + d \\ &\geq \frac{1}{2} \cdot 2^{\lceil (k-2)/2 \rceil} + \frac{1}{2} \cdot 2^{\lceil (k-3)/2 \rceil} \\ &= \frac{1}{2} \cdot 2^{(k-2)/2} + \frac{1}{2} \cdot 2^{(k-2)/2} \\ &= \frac{1}{2} \cdot 2^{(k-2)/2+1} \\ &= \frac{1}{2} \cdot 2^{k/2} \\ &= \frac{1}{2} \cdot 2^{\lceil k/2 \rceil}. \end{aligned}$$

  Thus, the statement also holds for k.
  By the principle of mathematical induction, $T(n) \geq \frac{1}{2} \cdot 2^{\lceil n/2 \rceil}$ holds for every $n \in \mathbb{N}$.

Hence, the run time of the algorithm in (a) is $T(n) \geq \frac{1}{2} \cdot 2^{\lceil n/2 \rceil} \geq \Omega(C^n)$ for $C = 2^{1/2} > 1$.

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

Solution:

Algorithm 2 Compute A_n using memoization

```
memory$\leftarrow$ $n$-dimensional array filled with $(-1)$s
function A_MEM(n)
    if memory[n] $\neq -1$ then                          $\triangleright$ If $A_n$ is already computed.
        return memory[n]
    if $n \leq 2$ then
        memory[n] $\leftarrow n$
        return $n$
    else if $n$ is odd then
        $A_n \leftarrow 1/2 \cdot$ (A_Mem($n-1$) + A_Mem($n-2$))
        memory[n] $\leftarrow A_n$
        return $A_n$
    else
        $A_n \leftarrow 2/$ (1/A_Mem($n-2$) + 1/A_Mem($n-3$))
        memory[n] $\leftarrow A_n$
        return $A_n$
```

When calling A_Mem(n), each A_k for $1 \leq k \leq n$ is computed exactly once and then stored in memory. Thus the run time of A_Mem(n) is $\Theta(n)$.

(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?

Solution:

- **Dimensions of the DP table:** The DP table is linear, its size is n.
- **Subproblems:** $DP[k]$ contains A_k for $1 \leq k \leq n$.
- **Recursion:** Initialize $DP[1]$ to 1, $DP[2]$ to 2. The entries with $k \geq 3$ are computed by $DP[k] = \frac{1}{2}(DP[k-1]+DP[k-2])$ for k odd and $DP[k] = 2/(1/DP[k-1]+1/DP[k-2])$ for k even (this is correct by the recursive definition of $A[k]$).
- **Calculation order:** We can calculate the entries of DP as follows:

```
for $i = 1 \dots n$ do
    Compute $DP[i]$.
```

- **Extracting the solution:** All we have to do is read the value at $DP[n]$.
- **Running time:** Each entry can be computed in time $\Theta(1)$, so the run time is $\Theta(n)$.

(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* $A_{2n+2} \cdot A_{2n+1}$.

Solution:

We show that $A_{2n+2} \cdot A_{2n+1} = 2$ for all $n \geq 0$ by induction.

- **Base Case.**
  For $n = 0$ we have $A_2 \cdot A_1 = 2 \cdot 1 = 2$.
- **Induction Hypothesis.**
  Assume the statement holds for some $k \geq 0$.
- **Induction Step.**
  Expanding out the definition of $A_{2(k+1)+2} = A_{2k+4}$ and $A_{2(k+1)+1} = A_{2k+3}$ we have

$$\begin{aligned} A_{2k+4} \cdot A_{2k+3} &= \frac{2}{\frac{1}{A_{2k+2}} + \frac{1}{A_{2k+1}}} \cdot \frac{A_{2k+2} + A_{2k+1}}{2} \\ &= \frac{A_{2k+2} + A_{2k+1}}{\frac{A_{2k+2}+A_{2k+1}}{A_{2k+2} \cdot A_{2k+1}}} \\ &= A_{2k+2} \cdot A_{2k+1} \\ &= 2, \end{aligned}$$

  using our induction hypothesis in the last step. Thus it holds for $k + 1$.

By the principle of mathematical induction, $A_{2n+2} \cdot A_{2n+1} = 2$ for ever $n \geq 0$.

We can assume that the limit of A_n exists, so set $\lim_{n\to\infty} A_n = a > 0$. Then

$$a = \lim_{n\to\infty} A_n = \lim_{n\to\infty} A_{2n+2} = \lim_{n\to\infty} \frac{2}{A_{2n+1}} = \frac{2}{a}$$

Therefore we conclude that $a^2 = 2$ and since $a > 0$ it must be that $a = \sqrt{2}$.

Guidelines for correction:

The following 4 points are important elements of this exercise. If 3 or 4 of them are solved correctly, 1 point should be awarded, for 1 or 2, 1/2 point should be awarded and if none are solved correctly, no points should be awarded.

- Correctly giving a recursive algorithm for part (a).
- Correct idea for the lower bound and proof idea using induction in part (b) (small errors are ok but the idea needs to be clearly visible).
- Correct idea that with memoization we only need to compute each A_k only once for part (c) (small errors in the code are ok but the argument why it becomes linear time should be there).
- Correct definition of the DP table and answers to the questions in part (d) (small errors are ok, but in general 5 out of the 6 questions should be answered correctly).

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 $0 \leq k \leq n$, 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

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

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

Solution:

Index	1	2	3	4	5	6	7	8	9	10
R	3	5	3	4	2	-1	4	5	2	6

(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.

Solution:

Index	1	2	3	4	5	6	7	8	9	10
R'	0	0	1	6	4	1	6	7	4	9

The solution is then $\max_{1 \leq i \leq n}\{R[i], R'[i], 0\}$ since we want to allow for both a subarray with a skip and one without a skip. We include 0 to account for the empty subarray (this is not strictly necessary since $R'[1] = 0$ by definition).

[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.

(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

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

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.

Solution:

Index	1	2	3	4	5	6	7	8	9	10
S	5	2	0	2	1	3	6	2	1	4

The solution is then $\max_{1 \leq i \leq n}\{R[i] + S[i+2], R[i], 0\}$ (we set $S[n+1] = S[n+2] = 0$) where the first value in the max includes a subarray ending at i ($R[i]$) and a subarray starting at $i+2$ ($S[i+2]$) thus encoding the skip at position $i+1$. The second value in the max is just if there is no skip. We include 0 to account for the empty subarray.

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

Solution:

For $R'[k]$, since we must skip an entry, and have elements on both sides of the skip, it only makes sense once $k \geq 3$. Therefore $R'[0] = R'[1] = R'[2] = 0$. For $k = 3$ it must be exactly the subarray sum of 1 and 3, skipping over 2, meaning $R'[3] = A[3] + R[1]$. Otherwise for higher values of k the maximum subarray which includes index k but skips an entry either includes $k-1$ or skips $k-1$. In the first case this then contains the maximum subarray which includes $k-1$ and skips an entry ($R'[k-1]$) and then appends $A[k]$. In the second case we skip index $k-1$ and then must include index $k-2$ if $k \geq 3$, and this will be the maximum subarray containing index $k-2$ ($R[k-2]$).

For $S[k]$, the argument is essentially the same as for $R[k]$ but going in the other direction. So a maximum subarray of $A[k..n]$ which includes index k either includes $k+1$ or doesn't. If it does, then our maximum subarray must contain the maximum subarray from $[(k+1)..n]$ which includes $k+1$ ($S[k+1]$). Otherwise it is just $A[k]$.

Guidelines for correction:

Award 1 point if all three parts are correct, 1/2 point if two of the three parts are correct and 0 otherwise. Allow for other valid formulas for extracting the right answer if they 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.

(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.

Solution:

We first recall the DP algorithm for finding the longest common subsequence between two arrays $A[1...n]$ and $B[1...m]$ from the lecture:

1. *Dimensions of the DP table*: The table is of size $(n + 1) \times (m + 1)$.

2. *Subproblems*: $DP[i, j]$ is the length of the longest common subsequence of $A[1...i]$ and $B[1...j]$.

3. *Recursion*: Initialize $DP[0, j] = 0$ and $DP[i, 0] = 0$ for all i, j and then compute

$$DP[i,j] = \begin{cases} \max\{DP[i, j-1], DP[i-1, j]\} & \text{if } A[i] \neq B[j] \\ \max\{DP[i, j-1], DP[i-1, j], DP[i-1, j-1] + 1\} & \text{if } A[i] = B[j]. \end{cases}$$

   The idea of this recursion is to split the recursion in two cases. If $i \neq j$, the i-th and j-th character of A, B are not equal, then the longest common subsequence of of $A[1...i]$ and $B[1...j]$ is one of the longest common subsequences of $A[1...i-1], B[1...j]$ and $A[1...i], B[1...j-1]$. Otherwise, it is also possible to extend the longest common subsequence of $A[1...i-1], B[1...j-1]$ by 1.

4. *Calculation Order*: We calculate the table DP as follows:

```
for $i = 0 \dots n$ do
    for $j = 0 \dots m$ do
        Compute $DP[i][j]$.
```

5. *Extracting the Solution*: The length of the longest common subsequence is $DP[n, m]$. To find a sequence of this length, we can use backtracking. To this end, we start moving from cell (n, m) of the DP table in the following way (we save the longest common subsequence in an array S of length $DP[n, m]$):

   i. If we are in cell (i, j) and $DP[i-1, j] = DP[i, j]$, we move to cell $(i-1, j)$.

   ii. Otherwise, if $DP[i, j-1] = DP[i, j]$, we move to $(i, j-1)$.

iii. Otherwise, by definition of the DP table, $DP[i-1, j-1] = DP[i, j] - 1$ and $A[i] = B[j]$, so we assign $S[DP[i, j]] \leftarrow A[i]$ and then we move to $(i-1, j-1)$.

iv. We stop when $i = 0$ or $j = 0$.

6. *Running Time*: Each entry is computed in time $O(1)$. Since there are $n \cdot m$ entries the total running time is $O(nm)$.

As in the description above, for the arrays given in this exercise, we assume that A has indices between 1 and 8, so $A[1 \ldots 0]$ is empty, and similarly for B. Then we get the following DP-table:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	0	1	1	1	2	2	2
3	0	1	1	1	1	1	1	1	2	2	2
4	0	1	1	1	1	1	1	2	2	2	2
5	0	1	1	1	2	2	2	2	2	2	2
6	0	1	1	1	2	2	2	2	2	3	3
7	0	1	1	1	2	2	2	2	2	3	4
8	0	1	1	1	2	2	3	3	3	3	4

Backtracking as explained in the description of the DP table above, we get the following:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1
2	0	0	0	0	0	1	1	1	2	2	2
3	0	1	1	1	1	1	1	1	2	2	2
4	0	1	1	1	1	1	1	2	2	2	2
5	0	1	1	1	2	2	2	2	2	2	2
6	0	1	1	1	2	2	2	2	2	3	3
7	0	1	1	1	2	2	2	2	2	3	4
8	0	1	1	1	2	2	3	3	3	3	4

To get the longest common subsequence, for every diagonal step from (i, j) to $(i+1, j+1)$, we add $A[i+1] = B[j+1]$ to the sequence. Thus, the longest common subsequence between the arrays A and B is $(A[3], A[5], A[6], A[7]) = (3, 8, 4, 5)$.

(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.

Solution:

We first recall the DP algorithm for determining the edit distance between two arrays $A[1...n]$ and $B[1...m]$ from the lecture:

1. *Dimensions of the DP table*: The table is of size $(n+1) \times (m+1)$.
2. *Subproblems*: $DP[i,j]$ is the edit distance of $A[1...i]$ and $B[1...j]$.
3. *Recursion*: Again, we assume that A has indices starting from 1, so $A[1 \dots 0]$ is empty, and similarly for B. Initialize $DP[0,0] = 0$, $DP[i,0] = i$ for $1 \leq i \leq n$, $DP[0,j] = j$ for $1 \leq j \leq m$ and then compute

   $$DP[i,j] = \max\left(DP[i-1,j]+1, DP[i,j-1]+1, DP[i-1,j-1] + \begin{cases} 0 & \text{if} A[i] = B[j] \\ 1 & \text{if} A[i] \neq B[j] \end{cases}\right)$$

   for $1 \leq i \leq n$ and $1 \leq j \leq m$.

   The idea behind this approach is to consider all possible modifications to the current positions i, j: We could remove the i-th character of A, then the edit distance is $DP[i-1,j]+1$. We could remove the j-th character of B, then the edit distance is $DP[i,j-1]+1$. Finally, if we change the current character of A or B to make them equal, the edit distance is $DP[i-1,j-1]+1$ or $DP[i-1,j-1]$ in case the characters were already the same. Note that for deletions, we only have to consider the case where we remove either $A[i]$ or $B[j]$ as removing both is always more costly than changing one of the strings.
4. *Calculation Order*: We calculate the table DP as follows:

   ```
   for $i = 0 \dots n$ do
       for $j = 0 \dots m$ do
           Compute $DP[i][j]$.
   ```

5. *Extracting the Solution*: The number of operations in an optimal sequence is $DP[n,m]$. To extract such a sequence, we can again use backtracking. We need to backtrack were the minimum (in the definition of $DP[i,j]$) comes from, starting in cell $(i,j) = (n,m)$. At every point the first part of the array is $A[1..i]$ and the second part of the array is $B[j+1..m]$ and it remains to transform $A[1..i]$ to $B[1..j]$. If we are in cell (i,j), we do the following:
   1. If $DP[i,j] = DP[i-1,j-1]$ and $A[i] = B[j]$, we do not need to do an operation and move on to cell $(i-1,j-1)$.
   2. Otherwise, if $DP[i,j] = DP[i-1,j-1]+1$ and $A[i] \neq B[j]$, we do a "change" operation at position i (we change $A[i]$ to $B[j]$) and move on to cell $(i-1,j-1)$.
   3. Otherwise, if $DP[i,j] = DP[i,j-1]+1$, we do an "insert" operation at position $i+1$ (we insert $B[j]$) and move on to cell $(i,j-1)$.
   4. Otherwise, $DP[i,j] = DP[i-1,j]+1$ and we do a "remove" operation at position i (we remove $A[i]$) and move on to cell $(i-1,j)$.

   We stop when $i = 0$ and $j = 0$.
6. *Running Time*: Each entry is computed in time $O(1)$. Since there are $n \cdot m$ entries the total running time is $O(nm)$.

In our example, we get the following DP-table:

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	2	3	4	5	6
1	1	0	1	2	3	4	5
2	2	1	1	2	3	4	5
3	3	2	2	2	2	3	4
4	4	3	3	3	3	3	4

So, the edit distance between A and B is 4. Backtracking the above table as described in point 5 above gives the following path.

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	2	3	4	5	6
1	1	0	1	2	3	4	5
2	2	1	1	2	3	4	5
3	3	2	2	2	2	3	4
4	4	3	3	3	3	3	4

And we get the following operations and intermediate arrays:

Current array	Value of i	Value of j	Next operation
[“S”, “O”, “R”, “T”]	$i = 4$	$j = 6$	Replace T at position 4 by H
[“S”, “O”, “R”, “H”]	$i = 3$	$j = 5$	Insert C at position 4
[“S”, “O”, “R”, “C”, “H”]	$i = 3$	$j = 4$	-
[“S”, “O”, “R”, “C”, “H”]	$i = 2$	$j = 3$	Replace O at position 2 by A
[“S”, “A”, “R”, “C”, “H”]	$i = 1$	$j = 2$	Insert E at position 2
[“S”, “E”, “A”, “R”, “C”, “H”]	$i = 1$	$j = 1$	-
[“S”, “E”, “A”, “R”, “C”, “H”]	$i = 0$	$j = 0$	-