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
Lucas Slot, Manuel Wiedmer, Hongjie Chen, Ding Jingqiu

30 October 2023

Algorithms & Data Structures Exercise sheet 6 HS 23

The solutions for this sheet are submitted at the beginning of the exercise class on 06 November 2023.

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 *Finding the i-th smallest key in an AVL tree* **(1 point)**.

Let A be an AVL tree (as described in the lecture) with n nodes. Let $k_1 < k_2 < \ldots < k_n$ be the keys of A, in ascending order. For a given $1 \leq i \leq n$, our goal is to find k_i, the i-th smallest key of A.

(a) Suppose $i = 1$. Describe an algorithm that finds k_1 in $O(\log n)$ time.

Hint: *An AVL tree is a BST (binary search tree).*

Solution:

Since A is a BST, we know that for each node v with key $\text{key}(v)$, the keys of its left subtree are all smaller than $\text{key}(v)$, while the keys of its right subtree are all greater than $\text{key}(v)$. It follows that k_1 can be found by starting at the root node, and then repeatedly moving to the left child, until we arrive at a node without a left child. As A is an AVL tree, it has depth $O(\log n)$, and so this procedure takes time $O(\log n)$.

(b) Describe an algorithm that finds k_i in $O(i \cdot \log n)$ time.

Hint: *You are allowed to make changes to A while executing your algorithm.*

Solution:

Using the algorithm of part (a), we can find the smallest key k_1 of A in time $O(\log n)$. Then we can remove the node with key k_1 from A in time $O(\log n)$, yielding a new AVL tree A' whose smallest key is k_2. Repeating this procedure i times yields k_i, using total time $O(i \cdot \log n)$.

It turns out that we can find k_i in time $O(\log n)$, if we modify the definition of an AVL tree a bit.

(c) Modify the definition of an AVL tree by storing two additional integers $s_l(v), s_r(v) \in \mathbb{N}$ in each node v. Assuming now that A satisfies your modified definition, describe an algorithm that finds k_i in $O(\log n)$ time.

Remark. Your modified definition should still allow for the search, insert and remove operations to be performed in $O(\log n)$ time, but you are not required to prove that this is the case.

Solution:

The additional integers we store are the sizes $0 \leq s_l(v), s_r(v) \leq n$ of the left and right subtree rooted at the left and right child of v, respectively. (This information can be updated in time $O(1)$ during the rebalancing rotations performed during the insert and remove operations). Assuming A is modified so that each node contains these integers, our algorithm to find k_i proceeds as follows.

Let v_0 be the root node of A. Set $i_0 = i$. We want to find the i_0-th smallest key in the subtree rooted at v_0 (which is just A). We consider the following three cases:

(i) If $s_l(v_0) = i_0 - 1$, we know that there are precisely $i_0 - 1$ keys in the subtree rooted in v_0 that are smaller than $\mathrm{key}(v_0)$; namely the keys in the subtree rooted in the *left child* of v_0. But that means that $\mathrm{key}(v_0)$ is the i_0-th smallest key in the subtree rooted in v_0, and so we output $\mathrm{key}(v_0)$.

(ii) If $s_l(v_0) > i_0 - 1$, the i_0-th smallest key lies in the subtree rooted in the *left* child $\mathrm{lc}(v_0)$ of v_0.

(iii) If $s_l(v_0) < i_0 - 1$, the i_0-th smallest key lies in the subtree rooted in the *right* child $\mathrm{rc}(v_0)$ of v_0.

Assuming we are in case (ii) or (iii), the idea is to proceed recursively by applying the same procedure to the subtree rooted in the left (resp. right) child of v_0.

In case (ii), note that the i_0-th smallest key of the subtree rooted in $\mathrm{lc}(v_0)$ is equal to the i_0-th smallest key of A (since all keys in the right subtree are too large). In this case, we can thus set $v_1 = \mathrm{lc}(v_0)$ and $i_1 = i_0$, and apply the procedure above.

In case (iii), things are slightly more complicated. Note that all $s_l(v_0)$ keys in the subtree rooted at $\mathrm{lc}(v_0)$ are smaller than $\mathrm{key}(v_0)$, and that $\mathrm{key}(\mathrm{rc}(v_0)) > \mathrm{key}(v_0)$. That is to say, if we set

$$i_1 = i_0 - (s_l(v_0) + 1),$$

then the i_1-th smallest key in the subtree rooted in $v_1 = \mathrm{rc}(v_0)$ is precisely the i_0-th smallest key in the subtree rooted in v_0 (which is what we are after).

We now repeat this procedure until we reach case (i). Each repetition takes $O(1)$ time. We move one layer down in each repetition. If we ever reach a leaf, we are certainly in case (i). Therefore, the whole algorithm takes at most $O(\log n)$ time (recall that A has depth $O(\log n)$).

Guidelines for correction:

The following 6 elements are important in this exercise. If at least 5 of them are present, award 1 point. If at least 3 are present, award $1/2$ point.

- Explain/mention how BST property allows you to find smallest key in part (a)
- Mention AVL trees have depth $O(\log n)$ in part (a)
- Correct idea for part (b).
- Correct idea (recursive) for part (c).
- Correct case distinction in part (c).
- Correct derivation of the update rules for v_0, i_0 in part (c) (when moving into the right subtree).

Exercise 6.2 *Round and square brackets.*

A string of characters on the alphabet $\{$`A`$, \dots,$ `Z`, `(`, `)`, `[`, `]`$\}$ is called *well-formed* if either

1. It does not contain any round or square brackets, or
2. It can be obtained from an empty string by performing a sequence of the following operations, in any order and with an arbitrary number of repetitions:
   (a) Take two non-empty well-formed strings a and b and concatenate them to obtain ab,
   (b) Take a well-formed string a and add a pair of round brackets around it to obtain `(`a`)`,
   (c) Take a well-formed string a and add a pair of square brackets around it to obtain `[`a`]`.

The above reflects the intuitive definition that all brackets in the string are 'matched' by a bracket of the same type. For example, $s =$ `FOO(BAR[A])`, is well-formed, since it is the concatenation of $s_1 =$ `FOO`, which is well-formed by 1., and $s_2 =$ `(BAR[A])`, which is also well-formed. String s_2 is well-formed because it is obtained by operation 2(b) from $s_3 =$ `BAR[A]`, which is well-formed as the concatenation of well-formed strings $s_4 =$ `BAR` (by 1.) and $s_5 =$ `[A]` (by 2(c) and 1.). String $t =$ `FOO[(BAR])` is not well-formed, since there is no way to obtain it from the above rules. Indeed, to be able to insert the only pair of square brackets according to the rules, its content $t_1 =$ `(BAR` must be well-formed, but this is impossible since t_1 contains only one bracket.

Provide an algorithm that determines whether a string of characters is well-formed. Justify briefly why your algorithm is correct, and provide a precise analysis of its complexity.

Hint: *Use a data structure from the last exercise sheet.*

Solution:

We use a stack providing standard POP, PUSH, and ISEMPTY operations. Given a stack S, S.POP() removes and returns the element on top of the stack, if it exists, and a constant `None` otherwise, while S.PUSH(x) pushes x onto the top of the stack, and S.ISEMPTY() returns a boolean indicating whether the stack is empty or not. Finally, we assume a function EMPTYSTACK that initializes and returns an empty stack. Our algorithm is as follows:

Algorithm 1 Detecting well-formed strings

```
function IsWellFormed($s$)
    $S \leftarrow$ emptyStack()
    for $i \in \{0, ..., |s| - 1\}$ do
        if $s[i] =$ "(" then
            $S$.push("(")
        else if $s[i] =$ "[" then
            $S$.push("[")
        else if $s[i] =$ ")" then
            if $S$.pop() $\neq$ "(" then
                return False
        else if $s[i] =$ "]" then
            if $S$.pop() $\neq$ "[" then
                return False
    return $S$.isEmpty()
```

Correctness. First, we see that we can completely ignore non-bracket characters to determine well-formedness. The correctness of our algorithm then results from the following invariant: for all s, the for loop of ISWELLFORMED(s) terminates (without returning early) in a configuration with an empty stack if and only if s is well-formed.

We can prove this by induction on the length of s.

Base case: If s has length 0, then it is empty. Then s is well-formed and $\textsc{IsWellFormed}(s)$ indeed terminates immediately with an empty stack.

Induction hypothesis: Let $n > 0$. Assume that for all s of length $|s| \leq n-1$, the for loop of $\textsc{IsWellFormed}(s)$ terminates (without returning early) in a configuration with an empty stack if and only if s is well-formed.

Induction step: Let s be a well-formed string of length n. First, assume that s is well-formed. There are three cases:

- If s is of the form ab with $0 < |a|, |b| < |s|$, then by our induction hypothesis the for loop of $\textsc{IsWellFormed}(a)$ and $\textsc{IsWellFormed}(b)$ terminates in a configuration with an empty stack. When running $\textsc{IsWellFormed}(s)$, the first $|a|$ iterations of the for loop are exactly the same as in $\textsc{IsWellFormed}(a)$, and we end up with an empty stack after $|a|$ iterations. Then, we run exactly the same $|b|$ steps as in $\textsc{IsWellFormed}(b)$, ending up again with an empty stack. We successfully return `True`.
- If s is of the form `(`a`)`, then running $\textsc{IsWellFormed}(s)$ first pushes "`(`" onto the stack, and then runs the same steps (from iterations 1 to $|s|-2$) as in $\textsc{IsWellFormed}(a)$, but with the additional "`(`" element remaining at the bottom of the stack. By our induction hypothesis, the stack contains only "`(`" after iteration $|s|-2$, after which iteration $|s|-1$ removes "`(`" from the stack and terminates successfully.
- The argument is similar for $s =$ `[`a`]`.

Conversely, assume that $\textsc{IsWellFormed}(s)$ returns `True`. We distinguish between two cases:

- If S is empty after some intermediate iteration $i \in \{0, \ldots, |s|-2\}$, consider such an i. Then the successful execution of $\textsc{IsWellFormed}(s)$ is exactly the concatenation of two successful executions of $\textsc{IsWellFormed}(s[0..i])$ and $\textsc{IsWellFormed}(s[i+1..|s|-1])$. Hence, by our induction hypothesis, $s[0..i]$ and $s[i+1..|s|-1]$ are well-formed, and their concatenation s is also well-formed.
- If S is never empty in any intermediate iteration, then we observe that the first element is pushed onto the stack and is never popped before the very last iteration. For this last pop to succeed, the first and last character of s must be matching brackets (i.e., `()` or `[]`). Moreover, as the element at the bottom of the stack is never popped except in the last iteration and the final stack is empty, iterations 1 to $|s|-2$ must be exactly identical to a successful execution of $\textsc{IsWellFormed}(s[1..|s|-2])$. Hence, by our induction hypothesis, $s[1..|s|-2]$ is a well-formed string, and so is s which is either `(`$s[1..|s|-2]$`)` or `[`$s[1..|s|-2]$`]`.

Remark. The above constitutes a formal proof of the correctness of the algorithm, provided for the sake of completeness. A more informal argument would also be counted as correct.

Complexity. Each iteration of the for loop has run time complexity $O(1)$: stack operations are $O(1)$, and we execute at most one such operation per iteration, along with at most 5 constant-time tests and at most one constant-time return statement. As there are $|s|$ iterations in total and the rest of the operations are constant-time, we get a total run time complexity in $O(|s|)$.

Dynamic programming.

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

Consider the recurrence

$$\begin{aligned}A_1 &= 1\\ A_2 &= 2\\ A_3 &= 3\\ A_4 &= 4\\ A_n &= A_{n-1} + A_{n-3} + 2A_{n-4} \text{ for } n \geq 5.\end{aligned}$$

(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 2 $A(n)$

```
if $n \leq 4$ then
    return $n$
else
    return $A(n-1) + A(n-3) + 2A(n-4)$
```

(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

$$T(1) = T(2) = T(3) = T(4) = 1$$

and

$$T(n) = T(n-1) + T(n-3) + T(n-4) + d,$$

where d is a positive constant.
The following section is an informal reasoning how we come up with a guess for the lower bound we can prove and is not part of the proof. We assume for one moment that T is monotonously increasing. Note that the proof in the end will not use this assumption, it is just to come up with the statement we want to prove. For $n \geq 5$, we then have $T(n) \geq 3T(n-4)$. Iterating this formula, we get

$$T(n) \geq 3T(n-4) \geq 9T(n-8) \geq \ldots \geq 3^k T(n-4k)$$

for $k \in \mathbb{N}$ with $k < \frac{n}{4}$. Choosing $k \approx \frac{n}{4}$, we come up with the guess $T(n) \geq 3^{n/4}$. However, this is not yet correct for $n = 1, 2, 3, 4$. We fix this by claiming that $T(n) \geq \frac{1}{3} \cdot 3^{n/4}$.

We now continue with the proof and want to show $T(n) \geq \frac{1}{3} \cdot 3^{n/4}$ by induction.

- **Base Case.**
  For $n \in \mathbb{N}$ with $n \leq 4$, we have $T(n) = 1 \geq \frac{1}{3} \cdot 3^{n/4}$ since $n/4 \leq 1$ implies $3^{n/4} \leq 3$.
- **Induction Hypothesis.**
  Assume that for some integer $k \geq 5$ the statement holds for all $k' < k$.

- **Induction Step.**
  We compute

$$\begin{aligned}
T(k) &= T(k-1) + T(k-3) + T(k-4) + d \\
&\geq \frac{1}{3} \cdot 3^{(k-1)/4} + \frac{1}{3} \cdot 3^{(k-3)/4} + \frac{1}{3} \cdot 3^{(k-4)/4} \\
&\geq \frac{1}{3} \cdot \left(3^{(k-4)/4} + 3^{(k-4)/4} + 3^{(k-4)/4}\right) \\
&= \frac{1}{3} \cdot 3 \cdot 3^{k/4-1} \\
&= \frac{1}{3} \cdot 3^{k/4}.
\end{aligned}$$

  Thus, the statement also holds for k.
  By the principle of mathematical induction, $T(n) \geq \frac{1}{3} \cdot 3^{n/4}$ holds for every $n \in \mathbb{N}$.

Hence, the run time of the algorithm in (a) is $T(n) \geq \frac{1}{3} \cdot 3^{n/4} \geq \Omega(C^n)$ for $C = 3^{1/4} > 1$.
Remark: With a bit more care, it can be shown by induction that $T(n) = \Theta(\phi^n)$, where $\phi \approx 1.618$ is the unique positive solution of $x^4 = x^3 + x + 1$.

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

Solution:

Algorithm 3 Compute A_n using memoization

```
memory← $n$-dimensional array filled with $(-1)$s
function A_Mem(n)
    if memory[n] $\neq -1$ then                          ▷ If $A_n$ is already computed.
        return memory[n]
    if $n \leq 4$ then
        memory[n] ← $n$
        return $n$
    else
        $A_n$ ← A_Mem($n-1$) + A_Mem($n-3$) + 2A_Mem($n-4$)
        memory[n] ← $A_n$
        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. Address the following aspects in your solution:

(1) *Definition of the DP table:* What are the dimensions of the table $DP[\ldots]$? What is the meaning of each entry?

(2) *Computation of an entry*: How can an entry be computed from the values of other entries? Specify the base cases, i.e., the entries that do not depend on others.

(3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?

(4) *Extracting the solution*: How can the final solution be extracted once the table has been filled?

(5) *Run time*: What is the run time of your solution?

Solution:

- **Dimensions of the DP table:** The DP table is linear, its size is n.
- **Definition of the DP table:** $DP[k]$ contains A_k for $1 \leq k \leq n$.
- **Calculation of an entry:** Initialize $DP[1]$ to 1, $DP[2]$ to 2, $DP[3]$ to 3 and $DP[4]$ to 4. The entries with $k \geq 5$ are computed by $DP[k] = DP[k-1] + DP[k-3] + 2DP[k-4]$.
- **Calculation order:** We can calculate the entries of DP from smallest to largest.
- **Reading the solution:** All we have to do is read the value at $DP[n]$.
- **Run time:** Each entry can be computed in time $\Theta(1)$, so the run time is $\Theta(n)$.

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 *Jumping game* **(1 point)**.

We consider the jumping game from the lecture for the following array of length $n = 10$:

$$A[1..n] = [2, 4, 2, 2, 1, 1, 1, 1, 5, 2].$$

We start at position 1. From our current position i, we may jump a distance of at most $A[i]$ forwards. Our goal is to reach the end of the array in as few jumps as possible. Recall the dynamic programming solution given for the problem in the lecture, revolving around the numbers:

$$M[k] := \text{'largest position reachable in at most } k \text{ jumps'}.$$

In this exercise, we compare two different methods for computing the $M[k]$.

(a) Consider the recursive relation:

$$\begin{aligned} &M[0] = 1, \\ &M[k] = \text{the maximum element of the array } R_k, \end{aligned} \tag{R}$$

where R_k is the array with indices i in the range $1 \leq i \leq M[k-1]$ and $R_k[i] := A[i] + i$. Compute $M[k]$ for $k = 1, 2, \ldots, K$ using relation (R), where K is the smallest integer for which $M[K] \geq n = 10$. For each $1 \leq k \leq K$, write down the array R_k used in the recursion. Finally, compute $\sum_{k=1}^{K} |R_k|$.

Solution:

k	$M[k]$	R_k
0	1	-
1	3	$[2+1]$
2	6	$[2+1,\ 4+2,\ 2+3]$
3	7	$[2+1,\ 4+2,\ 2+3,\ 2+4,\ 1+5,\ 1+6]$
4	8	$[2+1,\ 4+2,\ 2+3,\ 2+4,\ 1+5,\ 1+6,\ 1+7]$
5	9	$[2+1,\ 4+2,\ 2+3,\ 2+4,\ 1+5,\ 1+6,\ 1+7,\ 1+8]$
$K=6$	14	$[2+1,\ 4+2,\ 2+3,\ 2+4,\ 1+5,\ 1+6,\ 1+7,\ 1+8,\ 5+9]$

So, $\sum_{k=1}^{K} |R_k| = 34$.

(b) Now consider the recursive relation:

$$\begin{aligned} &M'[0] = 1, \\ &M'[1] = 1 + A[1], \\ &M'[k] = \text{the maximum element of the array } R'_k, \end{aligned} \tag{R'}$$

were R'_k is the array with indices i in the range $M'[k-2] < i \leq M'[k-1]$ and $R'_k[i] := A[i] + i$. Compute $M'[k]$ for $k = 1, 2, \ldots, K$ using relation (R'), where K is the smallest integer for which $M'[K] \geq n = 10$. For each $2 \leq k \leq K$, write down the array R'_k used in the recursion. Finally, compute $\sum_{k=1}^{K} |R'_k|$.

Solution:

k	$M'[k]$	R'_k
0	1	-
1	3	-
2	6	$[4+2,\ 2+3]$
3	7	$[2+4,\ 1+5,\ 1+6]$
4	8	$[1+7]$
5	9	$[1+8]$
$K=6$	14	$[5+9]$

So, $\sum_{k=1}^{K} |R'_k| = 8$.

(c*) Now let A be an arbitrary array of size $n \geq 2$ containing positive, non-repeating[1] integers. Let $M[k], M'[k]$ be the numbers computed using relations (R) and (R'), respectively. Prove that $M[k] = M'[k]$ for all $k \geq 0$.

Hint: *Use induction. First show that $M[0] = M'[0]$ and that $M[1] = M'[1]$. Then, use the induction hypothesis '$M[k-2] = M'[k-2]$ and $M[k-1] = M'[k-1]$' to show that* $\max R_k = \max R'_k$ *for all $k \geq 2$.*

Solution:

By definition, $M[0] = M'[0]$. Note that $R_1 := \{1 + A[1]\}$, and so $M[1] = M'[1]$. For the induction step, we want to show that $\max R_k = \max R'_k$ for $k \geq 2$. By definition, $\max R_k \geq \max R'_k$

[1]This assumption is only for convenience in writing the proof.

(as $R_k \supseteq R'_k$). It remains to rule out that $\max R_k > \max R'_k$. Using the induction hypothesis '$M[k-2] = M'[k-2]$ and $M[k-1] = M'[k-1]$', we see that[2]

$$R_k \setminus R'_k = \{i + A[i] : 1 \leq i \leq M[k-2]\} = R_{k-1}.$$

It follows that $\max\left(R_k \setminus R'_k\right) = M[k-1] < \max R_k$. We conclude that $\max R_k = \max R'_k$, and thus $M[k] = M'[k]$, as desired.

Guidelines for correction:

Award $1/2$ points each for part (a) and (b). If there is a mistake in computing the $M[k]$ or R_k (resp. $M'[k]$, R'_k) award 0 points for that part. For $M[6]$ and $M'[6]$, both 10 and 14 should be counted as correct. Do not subtract points for mistakes in computing the sums $\sum_{k=1}^{K} |R_k|$ or $\sum_{k=1}^{K} |R'_k|$.

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.
For part (a), we are going to look at the problem of finding the longest common subsequence in two arrays. So, we are given two arrays, A of length n, and B of length m, and we want to find their longest common subsequence and its length. 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.
For part (b), we are looking at the problem of determining the edit distance between two sequences. We are again given two arrays, A of length n, and B of length m. We want to find 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.

(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:

As described in the lecture, $DP[i, j]$ denotes the size of the longest common subsequence between the strings $A[1 \ldots i]$ and $B[1 \ldots j]$. Note that 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:

[2]For two sets A, B we write $A \setminus B := \{a \in A : a \notin B\}$, that is 'the elements of A that are not in B'.

	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 find some longest common subsequence, we create an array S of length $DP[n,m]$ and then we start moving from cell (n,m) of the DP table in the following way:

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

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

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 $DP[i-1,j-1]$.

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

Using this procedure we find the following longest common subsequence: $S = [7,6,4,5]$.

(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:

As described in the lecture, $DP[i,j]$ denotes the edit distance between the arrays $A[1\dots i]$ and $B[1\dots j]$. Again, we assume that A has indices starting from 1, so $A[1\dots 0]$ is empty, and similarly for B. As in the lecture we have $DP[i,0] = i$ for i between 0 and n, $DP[0,j] = j$ for j between 0 and m and

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

for i between 1 and n and j between 1 and m. Then 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. To get operations that transform A into B, we need to backtrack were the minimum (in the definition of $D[i, j]$) 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:

- 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 $(i - 1, j - 1)$.
- 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 $(i - 1, j - 1)$.
- 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 $(i, j - 1)$.
- 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 $i = 0$. Backtracking the above table 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	Next operation
[“S”, “O”, “R”, “T”]	$i = 4$	Replace T at position 4 by H
[“S”, “O”, “R”, “H”]	$i = 3$	Insert C at position 4
[“S”, “O”, “R”, “C”, “H”]	$i = 3$	-
[“S”, “O”, “R”, “C”, “H”]	$i = 2$	Replace O at position 2 by A
[“S”, “A”, “R”, “C”, “H”]	$i = 1$	Insert E at position 2
[“S”, “E”, “A”, “R”, “C”, “H”]	$i = 1$	-
[“S”, “E”, “A”, “R”, “C”, “H”]	$i = 0$	-