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, Markus Püschel, David Steurer
Kasper Lindberg, Kostas Lakis, Lucas Pesenti, Manuel Wiedmer

27 October 2025

Algorithms & Data Structures — Exercise sheet 6 — HS 25

The solutions for this sheet are submitted on Moodle until 02 November 2025, 23:59.

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

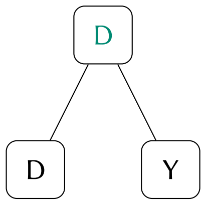
You can use results from previous parts without solving those parts.

Data structures.

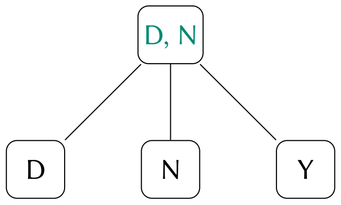
Exercise 6.1 *2-3 Trees* **(1 point)**.

In this exercise, we will practice inserting to and deleting from 2-3 trees.

(a) We will consider the string "DYNAMIC!" and we want to observe the state of the tree after each insertion. We start with the letters D, Y already in the tree:

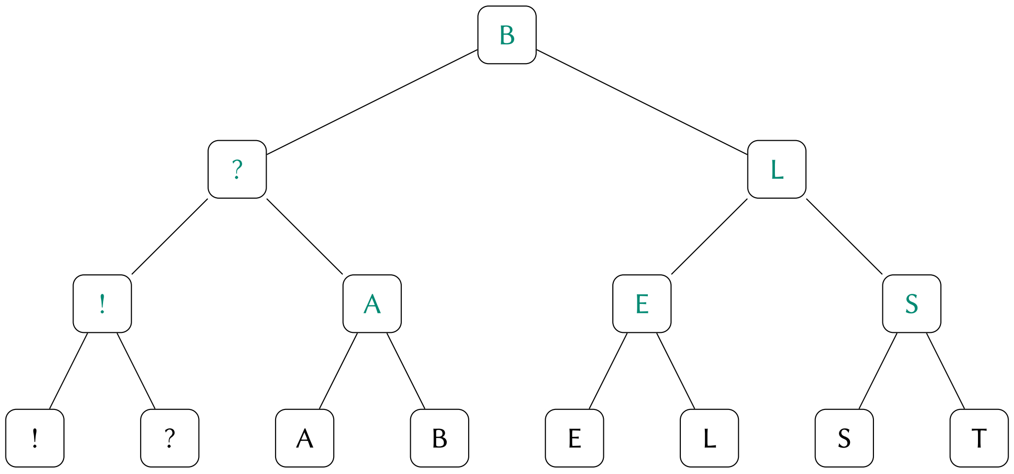


When we insert "N", the tree changes to:



Draw the state of the tree after each subsequent insertion (A, M, I, C, !). Note that (by the ASCII codes for the characters), "!" comes before any letter. For convenience, the full ordering is : "!, A, C, D, I, M, N, Y".

(b) We will now consider the string "TABLES?!" instead and look at deletions. After inserting the characters one by one, the tree looks like this:



Show the state of the tree after deleting the characters T, A, L, E, S, B (on purpose does not spell TABLES) in that order. For convenience, the full ordering is : “!, ?, A, B, E, L, S, T”.

Exercise 6.2 *Finding the i-th smallest key in a 2-3 tree.*

Let T be a 2-3 tree (as described in the lecture) with n leaves. Let $k_1 < k_2 < \ldots < k_n$ be the keys of T, in ascending order. For a given $1 \leq i \leq n$, our goal is to find k_i, the i-th smallest key of T.

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

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

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

It turns out that we can find k_i in time $O(\log n)$, if we modify the definition of a 2-3 tree a bit.

(c) Modify the definition of a 2-3 tree by storing three additional integers $s_l(v), s_m(v), s_r(v) \in \mathbb{N}$ in each node v (to be used in some way of your choice). Assuming now that T satisfies your modified definition, describe an algorithm that finds k_i in $O(\log n)$ time.

Remark. One would technically need to redefine the *search*, *insert* and *delete* operations. You do not need to be very careful about the details here, just describe in a high level what the changes would be with regard to the three new integers stored (but the resulting operations should still need only logarithmic time in the worst case). The important thing to consider here is how one utilizes the values of these integers to support finding k_i in logarithmic time. For this, you need to be precise.

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.

(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. 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?

Exercise 6.4 *Coin Conversion* **(1 point)**.

Suppose you live in a country where the transactions between people are carried out by exchanging coins denominated in dollars. The country uses coins with k different values, where the smallest coin has value of $b_1 = 1$ dollar, while other coins have values of $b_2, b_3, \ldots, b_k$ dollars. You received a bill for n dollars and want to pay it *exactly* using the smallest number of coins. Assuming you have an unlimited supply of each type of coin, define OPT to be the minimum number of coins you need to pay exactly n dollars. Your task is to calculate OPT. All values $n, k, b_1, \ldots, b_k$ are positive integers.

Example: $n = 17, k = 3$ and $b = [1, 9, 6]$, then $\mathrm{OPT} = 4$ because 17 can be obtained via 4 coins as $1 + 1 + 9 + 6$. No way to obtain 17 with three or less coins exists.

(a) Consider the pseudocode of the following algorithm that "tries" to compute OPT.

Algorithm 1

```
1: Input: integers $n, k$ and an array $b = [1 = b_1, b_2, b_3, \ldots, b_k]$.
2:
3: $counter \leftarrow 0$
4: while $n > 0$ do
5:     Let $b[i]$ be the value of the largest coin $b[i]$ such that $b[i] \leq n$.
6:     $n \leftarrow n - b[i]$.
7:     $counter \leftarrow counter + 1$
8: Print("min. number of required coins = ", $counter$)
```

Algorithm 1 does not always produce the correct output. Show an example where the above algorithms fails, i.e., when the output does not match OPT. Specify what are the values of n, k, b, what is OPT and what does Algorithm 1 report.

(b) Consider the pseudocode below. Provide an upper bound in O notation that bounds the time it takes a compute $f[n]$ (it should be given in terms of n and k). Give a short high-level explanation of your answer. For full points your upper bound should be tight (but you do not have to prove its tightness).

Algorithm 2

```
Input: integers $n, k$. Array $b = [1 = b_1, b_2, b_3, \dots, b_k]$.

Let $f[1 \dots n]$ be an array of integers.
$f[0] \leftarrow 0$                                    ▷ Terminating condition.
for $N \leftarrow 1 \dots n$ do
    $f[N] \leftarrow \infty$          ▷ At first, we need $\infty$ coins. We try to improve upon that.
    for $i \leftarrow 1 \dots k$ do
        if $b[i] \leq N$ then
            $val \leftarrow 1 + f[N - b[i]]$     ▷ Use coin $b[i]$, it remains to optimally pay $N - b[i]$.
            $f[N] \leftarrow \min(f[N], val)$
Print($f[n]$)
```

(c) Let $\mathrm{OPT}(N)$ be the answer (min. number of coins needed) when $n = N$. Algorithm 2 (correctly) computes a function $f[N]$ that is equal to $\mathrm{OPT}(N)$. Formally prove why this is the case, i.e., why $f[N] = OPT(N)$.

Hint: *Use induction to prove the invariant $f[n] = OPT(n)$. Assume the claim holds for all values of $n \in \{1, 2, \dots, N-1\}$. Then show the same holds for $n = N$.*

(d) Rewrite Algorithm 2 to be recursive and use memoization. The running time and correctness should not be affected.

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 $A = [\text{"A"}, \text{"N"}, \text{"D"}]$ and $B = [\text{"A"}, \text{"R"}, \text{"E"}]$, 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.

(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.