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).*

(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.*

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.

Exercise 6.2 *Round and square brackets.*

A string of characters on the alphabet $\{\texttt{A}, \ldots, \texttt{Z}, \texttt{(}, \texttt{)}, \texttt{[}, \texttt{]}\}$ 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 $\texttt{(}a\texttt{)}$,
   (c) Take a well-formed string a and add a pair of square brackets around it to obtain $\texttt{[}a\texttt{]}$.

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.*

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 *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 $\sum_{k=1}^{K} |R_k|$.

(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 $\sum_{k=1}^{K} |R'_k|$.

(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 $k \geq 2$.*

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

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

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 $A = [\text{"A"}, \text{"N"}, \text{"D"}]$ 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.

(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.