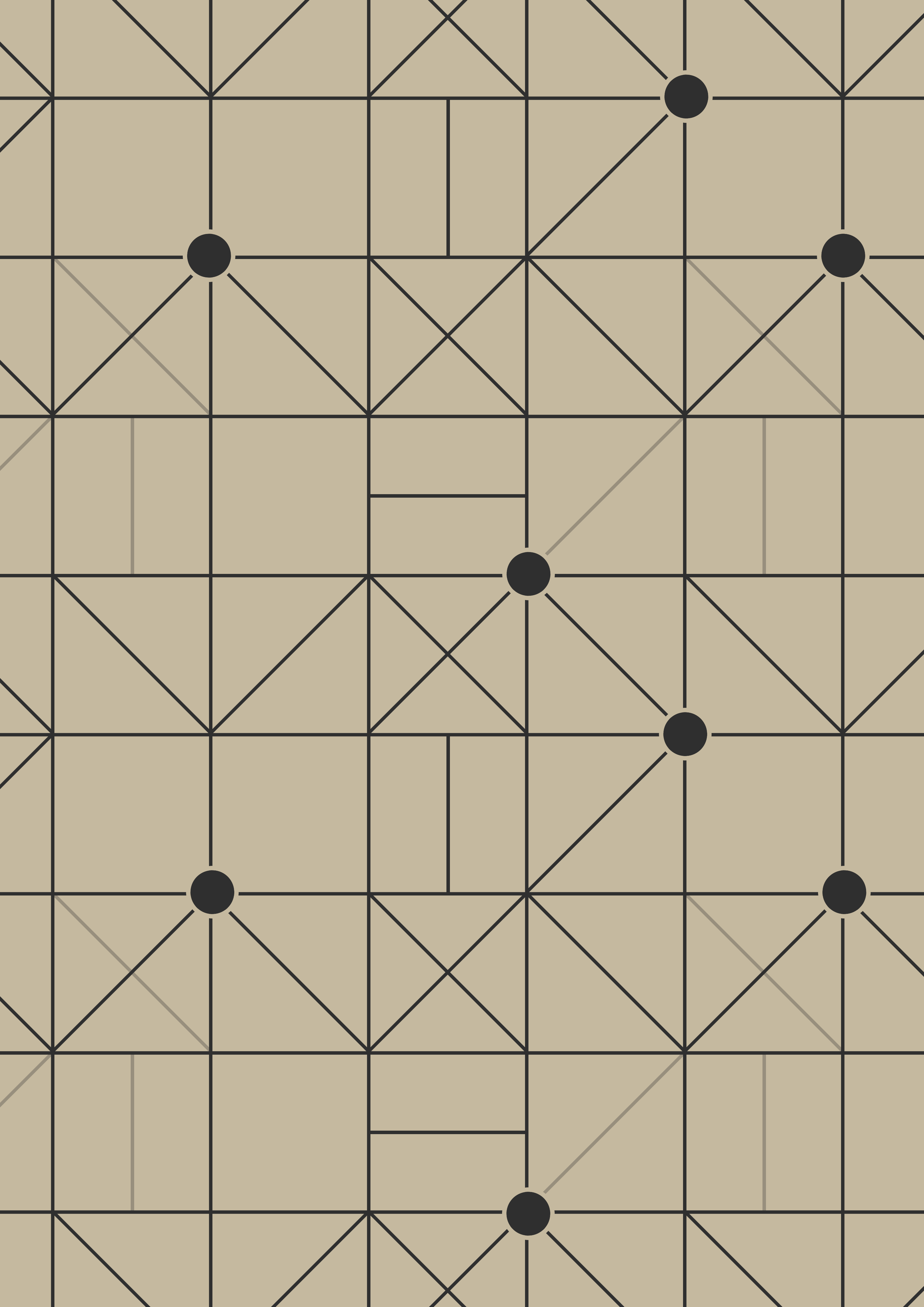

ALGORITHMEN + DATENSTRUKTUREN - ÜBUNGSSTUNDE 7

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.

a) We always go to the left child starting at the root as soon as we reached a leaf we have found the smallest key the height of the tree is $O(\log n) \rightarrow$ our runtime

b) When we are allowed to make changes, we can always find smallest as in a and then delete that node which is as seen in the lecture also in $O(\log n)$. We do that i times which justifies the $O(i \log(n))$ runtime.

c) The new s-values describe how many leaves are in the left, middle, and right subtree respectively we show that this value can be updated in $O(1)$ for insert and delete

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?

a)
```
A(n)
    if $n < 5$
        return n
    else
        return $A(n-1) + A(n-3) + 2A(n-4)$
```

b) Recursion: $\begin{cases} 1 & , \text{if } n \leq 4 \\ T(n-1) + T(n-3) + T(n-4) + d & , \text{otherwise} \end{cases}$

We bound $T(n)$ by:

$T(n) \geq 3T(n-4) \geq 9T(n-8) \geq \ldots \geq 3^k T(n-4k)$ and $k < \frac{n}{4}$

$T(n) \geq 3^{n/4}$ but because of the base cases we do $T(n) \geq \frac{1}{3} 3^{n/4}$

BC: $n \leq 4$ $T(n) = 1 \geq \frac{1}{3} 3^{n/4}$ as $n/4 \leq 1$

IH: Assume for some integer ≥ 5 the property holds for all $k' < k$

IS: $T(k) = T(k-1) + T(k-3) + T(k-4) + d$

c)
```
mem ← n-dimensional array with -1   |to mark non-visited spaces
func B(n)
    if n<5
        return n
    else if mem[n] ≠ -1
        return mem[n]
    else
        mem[n] ← B(n-1)+B(n-3)+2B(n-4)
        return mem[n]
```

d)
```
for i ← 1,...,n
    if i<5
        dp [i]= i
    else
        dp [i]= dp [i-1] + dp [i-3] + 2· dp [i-4]
return dp [n]
```

1. size n (1-dimensional)
2. dp [i] = A_i
3. dp [i] = i , if i<5
   dp [i] = dp [i-1] + dp [i-3] + 2 dp [i-4], otherwise
4. from i=1 to i=n
5. each entry in $\Theta(1)$ → total of $\Theta(n)$