Algorithmen und Datenstrukturen
Exam Prep Sheet

Lejs Behrić

HS23

This document has been designed to give students of the Algorithms and Data Structures course an overview of all exam-relevant topics and to provide specific tips on how to best solve typical exam tasks. If any mistakes are found, please contact me at lbehric@inf.ethz.ch. This document does not claim to be complete and correct and I do not guarantee that other topics are not also relevant to the exam.

Contents

1 Asymptotic Notation

1.1 Important Definitions

Let N be an infinite subset of $\mathbb{N}$ and $f, g : N \to \mathbb{R}^+$.

- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f \leq O(g)$, but $f \neq \Theta(g)$.
- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C \in \mathbb{R}^+$, then $f = \Theta(g)$.
- If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, then $f \geq \Omega(g)$, but $f \neq \Theta(g)$.

1.2 Order of asymptotic growth of different functions

Here $f < g$ means that f grows asymptotically slower than $g \implies f \leq O(g) \wedge f \neq \Theta(g)$

$$\cdots < \log(\log(n)) < \log(n) < \sqrt{n} < n^{0.9} < n < n\log n < n\sqrt{n} < n^2 < 2^n < e^n < n! < n^n < \ldots$$

Remember that for arbitrary constants $a, b, c, d \in \mathbb{R}^+$: $\log_a(n^b) = \Theta(\log_c(n^d))$.

1.3 Useful upper/lower bounds for sums

Bounds for sums, where f is monotonically increasing: $i \leq j \implies f(i) \leq f(j)$

$$\sum_{i=1}^{n} f(i) \leq \sum_{i=1}^{n} f(n) = n \cdot f(n) \leq O(n \cdot f(n))$$

$$\sum_{i=1}^{n} f(i) \geq \sum_{i=\lceil n/2 \rceil}^{n} f(i) \geq \sum_{i=\lceil n/2 \rceil}^{n} f(\lceil n/2 \rceil) = (n - \lceil n/2 \rceil + 1) \cdot f(\lceil n/2 \rceil) \geq \Omega(n \cdot f(n/2))$$

1.4 Useful formulas for resolving sums

Sum of integers and Geometric sum formula

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$$

$$\sum_{i=0}^{n} q^i = \frac{q^{n+1} - 1}{q - 1} = \Theta(q^n)$$

1.5 Methods for solving common asymptotic notation tasks

1. **By intuition:** For this you should memorize 1.2

   E.g.: $n^3 + \sqrt{n} \overset{?}{\leq} O(n^2) \rightarrow$ False

2. **Rearrange fraction:** $\frac{g}{h} \leq O(f) \iff g \leq O(f \cdot h)$ $(h \neq 0)$

   E.g.: $\frac{n}{\log(n)} \overset{?}{\leq} O(\sqrt{n} \cdot \log(n)) \iff n \overset{?}{\leq} O(\sqrt{n} \cdot \log(n)^2) \rightarrow$ False

3. **Find upper/lower bound:** Use bounds in 1.3

   E.g.: $\sum_{i=1}^{\lceil\sqrt{n}\rceil} i^3 \overset{?}{\geq} \Omega(n^2) \iff \lceil\frac{\sqrt{n}}{2}\rceil(\lceil\frac{\sqrt{n}}{2}\rceil)^3 \geq \frac{n^2}{16} \overset{?}{\geq} \Omega(n^2) \rightarrow$ True

4. **Resolve sum with one of the formulas:** Use formulas in 1.4

   E.g.: $\frac{n^2}{2} - \sum_{i=1}^{n-1} i \overset{?}{\leq} O(n) \iff \frac{n^2}{2} - (\frac{n^2}{2} - \frac{n}{2}) \overset{?}{\leq} O(n) \rightarrow$ True

5. **Deal with recurrences:** Try to find upper/lower bound of recurrence

   E.g.: Let $a_1 = 1$ and $a_{n+1} = 3a_i + 1$ $\forall i \geq 2$ Then $a_n \overset{?}{\leq} O(4^n)$.

   Notice that a_n is monotonically increasing, and $a_{n+1} \leq 4a_n$.

   Therefore $a_n \leq 4a_{n-1} \leq 4 \cdot 4a_{n-2} \leq ... \leq O(4^n) \rightarrow$ True

2 Induction proofs and recurrence relations

2.1 Induction proofs

The classical proof by induction is concerned with showing that a statement is valid for all natural numbers $n \in \mathbb{N}$. This statement does not have to be a mathematical formula directly, but can also be much more general, such as an invariant for example. An induction proof consists of the following components:

1. *Induction Start:* Show that the statement holds for $n = 1$.
2. *Induction Hypothesis:* We assume that the statement holds for some arbitrary $n \in \mathbb{N}$.
3. *Induction Step:* Show that the validity of the statement for n (induction hypothesis) implies the validity of the statement for $n + 1$.

A generalised variant of induction allows a stronger induction hypothesis, namely that the statement is valid for all $k \leq n$ (from which, of course, the validity of the statement for $n + 1$ must then also be concluded in the induction step). Induction can also begin for a number $n_0 > 1$ (but then the statement is no longer valid for all $n \in \mathbb{N}$, but only for all $n \in \mathbb{N}$ with $n \geq n_0$).

2.2 Recurrence relations

To recognise a pattern in recurrences, it often helps to write out the first few terms. For recurrences of the form $a_n = ca_{n-1} + d$ and $a_0 = b$, a general closed formula exists:

$$a_n = \begin{cases} b + nd & \text{if } c = 1 \\ c^n b + \frac{c^n - 1}{c - 1} \cdot d & \text{if } c \neq 1 \end{cases}$$

If you have to prove an equation with a recurrence by induction, then use the definition of a_0 in the base case and the definition of a_{n+1} in the induction step to conclude the proof.

3 Search Trees

3.1 Heaps

3.2 Binary Search Trees

A binary tree is

- either a leaf, i.e. the tree is empty,
- or it consists of an inner node v with two trees $T_l(v)$ and $T_r(v)$ as left and right successors respectively. The tree $T_l(v)$ is called the left subtree of v, $T_r(v)$ is called the right subtree of v.

Each tree has exactly one node without an operator, the so-called root, which is called `root` in the following algorithm descriptions. In each inner node v, we store

- a key `v.Key`,
- a pointer `v.Left` to the left successor node (i.e. to the root of the left subtree $T_l(v)$ and *not* to the subtree as a whole),
- a pointer `v.Right` to the right successor node (i.e. to the root of the right subtree $T_r(v)$)
- and a pointer `v.Right` to the right successor node (i.e. to the root of the right subtree $T_r(v)$).

If the left (or right) successor of an inner node is a leaf, then we set `v.Left` (or `v.Right`) to **null**. A pointer to the root is then sufficient to represent the entire tree. The tree is thus fully represented by pointers to the corresponding successors, and not, for example, as an array (as with heaps), although it is not yet clear how this helps in the search for a given key. A heap, for example, could also be represented by corresponding pointers, but it is unclear how to search there efficiently.

Therefore, we now introduce another criterion: A binary search tree is a binary tree that fulfils the **search tree property**: Each inner node v stores a key k, all keys stored in the left subtree $T_l(v)$ of v are smaller than k and all keys stored in the right subtree $T_r(v)$ of v are larger than k.

3.3 AVL Trees

AVL Property

Rotations

Heap Condition

Heap Operations

4 Searching and Sorting Algorithms

4.1 Searching Algorithms

4.2 Sorting Algorithms

4.3 Comparison of Sorting Algorithms

5 Dynamic Programming

6 Graphs

6.1 Definitions

6.2 BFS / DFS and Topological Sorting

6.3 Shortest Paths

6.4 Minimum Spanning Trees