



## Algorithms and Data Structures Winter Term 2021/2022 Sample Solution Exercise Sheet 2

### Exercise 1: $\mathcal{O}$ -notation

Prove or disprove the following statements. Use the *set definition* of the  $\mathcal{O}$ -notation (lecture slides week 2, slides 11 and 12).

- (a)  $4n^3 + 8n^2 + n \in \mathcal{O}(n^3)$
- (b)  $2^n \in o(n!)$
- (c)  $2 \log n \in \Omega((\log n)^2)$
- (d)  $\max\{f(n), g(n)\} \in \Theta(f(n) + g(n))$  for non-negative functions  $f$  and  $g$ .

### Sample Solution

(a) True. Choose  $n_0 = 1$  and  $c = 13$ . For  $n \geq n_0$  we have  $n^3 \geq n^2 \geq n$  and hence  $4n^3 + 8n^2 + n \leq 13n^3 = cn^3$ .

(b) True. Let  $c > 0$ . Choose  $n_0 = \max\{\frac{1}{c}, 8\}$ . For  $n \geq n_0$  we have

$$c \cdot n! \stackrel{n \geq 1/c}{\geq} \frac{1}{n} \cdot n! = (n-1)! \geq (n-1) \cdot (n-2) \cdot \dots \cdot \lfloor \frac{n}{2} \rfloor \stackrel{n \geq 8}{\geq} 4^{\frac{n}{2}} = 2^n$$

(c) False. Let  $c > 0$ . We have

$$\begin{aligned} 2 \log n &\geq c(\log n)^2 \\ \Leftrightarrow 2 &\geq c \log n \\ \Leftrightarrow \frac{2}{c} &\geq \log n \\ \Leftrightarrow 4^{\frac{1}{c}} &\geq n \end{aligned}$$

So for a given  $n_0 \geq 1$  choose  $n = \max\{n_0, 4^{\frac{1}{c}}\} + 1$ . For this  $n$  we have  $n > n_0$  and  $2 \log n < c(\log n)^2$ .

(d) True. Choose  $n_0 = 1$ ,  $c_1 = \frac{1}{2}$  and  $c_2 = 1$ . For  $n \geq n_0$  we have

$$c_1 \cdot (f(n) + g(n)) \leq \max\{f(n), g(n)\} \stackrel{f, g \geq 0}{\leq} c_2 (f(n) + g(n))$$

### Exercise 2: Sorting by asymptotic growth

Sort the following functions by their asymptotic growth. Write  $g <_{\mathcal{O}} f$  if  $g \in \mathcal{O}(f)$  and  $f \notin \mathcal{O}(g)$ . Write  $g =_{\mathcal{O}} f$  if  $f \in \mathcal{O}(g)$  and  $g \in \mathcal{O}(f)$  (no proof needed).

$\sqrt{n}$	$2^n$	$n!$	$\log(n^3)$
$3^n$	$n^{100}$	$\log(\sqrt{n})$	$(\log n)^2$
$\log n$	$10^{100}n$	$(n+1)!$	$n \log n$
$2^{(n^2)}$	$n^n$	$\sqrt{\log n}$	$(2^n)^2$

## Sample Solution

$$\begin{aligned} \sqrt{\log n} &<_{\mathcal{O}} \log(\sqrt{n}) =_{\mathcal{O}} \log n =_{\mathcal{O}} \log(n^3) <_{\mathcal{O}} (\log n)^2 <_{\mathcal{O}} \sqrt{n} <_{\mathcal{O}} 10^{100}n <_{\mathcal{O}} n \log n \\ &<_{\mathcal{O}} n^{100} <_{\mathcal{O}} 2^n <_{\mathcal{O}} 3^n <_{\mathcal{O}} (2^n)^2 <_{\mathcal{O}} n! <_{\mathcal{O}} (n+1)! <_{\mathcal{O}} n^n <_{\mathcal{O}} 2^{(n^2)} \end{aligned}$$

## Exercise 3: Stable Sorting

A sorting algorithm is called stable if elements with the same key remain in the same order. E.g., assume you want to sort the following strings where the sorting key is *the first letter by alphabetic order*:

["tuv", "adr", "bbc", "tag", "taa", "abc", "sru", "bcb"]

A *stable* sorting algorithm must generate the following output:

["adr", "abc", "bbc", "bcb", "sru", "tuv", "tag", "taa"]

A sorting algorithm is not stable (with respect to the sorting key) if it outputs, e.g., the following:

["abc", "adr", "bbc", "bcb", "sru", "taa", "tag", "tuv"]

- Which sorting algorithms from the lecture (except **CountingSort**) are *not* stable? Prove your statement by giving an appropriate example.
- Describe a method to make any sorting algorithm stable, without changing the *asymptotic* runtime. Explain.

## Sample Solution

- Selection Sort is not stable. Consider as input the array  $[x, y, z]$  with  $x.key = y.key = 1$  and  $z.key = 0$ . In the first step,  $x$  and  $z$  are swapped, because  $z$  has the smallest key in the array. So we get  $[z, y, x]$ . This array will not be altered in the second step (as  $y.key = x.key$ ), i.e., it equals the output of Selection Sort. So  $x$  and  $y$  have been swapped.
  - Quicksort is not stable. Consider as input the array  $[x, y, z, w]$  with  $x.key = 1$ ,  $y.key = z.key = 2$  and  $w.key = 0$  and assume  $x$  is taken as pivot. In the first divide step,  $y$  and  $w$  are swapped (i.e., we get  $[x, w, z, y]$ ) and the array is divided into  $[x, w]$  and  $[z, y]$ . Recursive sorting yields  $[w, x]$  and  $[z, y]$  and thus  $[w, x, z, y]$  will be returned. So  $y$  and  $z$  have been swapped.
  - Mergesort: If you implement Mergesort according to the pseudocode on page 26 of lecture 01, Mergesort is not stable. The reason is the condition  $A[i] < A[j]$  in line 7 of the code which may cause elements with the same key to change order. If we instead use the condition  $A[i] \leq A[j]$ , we make the algorithm stable.
- Add the number  $i$  to the key of the  $i$ -th element in the array (i.e., set  $A[i].key = (A[i].key, i)$ ). Now run the given (non-stable) sorting algorithm according to the lexicographic ordering<sup>1</sup> on this new set of keys. That is, we sort according to the original keys and use the index in  $A$  as tie breaker.

Changing the keys takes time  $O(n)$ . Additionally, each comparison between two elements is prolonged by an additional  $O(1)$  steps. As any sorting algorithm takes  $\Omega(n)$ , the asymptotic runtime does not change.

---

<sup>1</sup>Let  $(A, <_A)$  and  $(B, <_B)$  be ordered sets. The lexicographic ordering  $<_{lex}$  on  $A \times B$  is defined by  $(a, b) <_{lex} (a', b') \Leftrightarrow a <_A a' \vee (a = a' \wedge b <_B b')$

## Exercise 4: Running time

Give an asymptotically tight upper bound for the running time of the following algorithm as a function of  $n$ .

---

```
s ← 0
for i = 1 to n do
  j = 1
  while j < i do
    s ← s + i · j
    j ← 2 · j
```

---

## Sample Solution

For each  $i$ , the running time of the internal while is proportional to  $\log_2 i$ . Hence, the total running time is proportional to  $\sum_{i=1}^n \log_2 i$ . For an upper bound, note that this sum is upper bounded by  $\sum_{i=1}^n \log_2 n = n \log_2 n = O(n \log n)$ . In order to show that it is tight, note that the sum is lower bounded by  $\sum_{i=n/2}^n \log_2(n/2) = (n/2) \log(n/2) = \Omega(n \log n)$ . Hence the running time is  $\Theta(n \log n)$ .