Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Markus Püschel, David Steurer
François Hublet, Goran Zuzic, Tommaso d'Orsi, Jingqiu Ding

17 October 2022

Algorithms & Data Structures Exercise sheet 4 HS 22

The solutions for this sheet are submitted at the beginning of the exercise class on 24 October 2022.

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.

Master Theorem. The following theorem is very useful for running-time analysis of divide-and-conquer algorithms.

Theorem 1 (Master theorem). *Let $a, C > 0$ and $b \geq 0$ be constants and $T : \mathbb{N} \to \mathbb{R}^+$ a function such that for all even $n \in \mathbb{N}$,*

$$T(n) \leq aT(n/2) + Cn^b. \tag{1}$$

Then for all $n = 2^k$, $k \in \mathbb{N}$,

- *If $b > \log_2 a$, $T(n) \leq O(n^b)$.*
- *If $b = \log_2 a$, $T(n) \leq O(n^{\log_2 a} \cdot \log n)$.*
- *If $b < \log_2 a$, $T(n) \leq O(n^{\log_2 a})$.*

If the function T is increasing, then the condition $n = 2^k$ can be dropped. If (1) holds with "$=$", then we may replace O with Θ in the conclusion.

This generalizes some results that you have already seen in this course. For example, the (worst-case) running time of Karatsuba algorithm satisfies $T(n) \leq 3T(n/2) + 100n$, so $a = 3$ and $b = 1 < \log_2 3$, hence $T(n) \leq O(n^{\log_2 3})$. Another example is binary search: its running time satisfies $T(n) \leq T(n/2) + 100$, so $a = 1$ and $b = 0 = \log_2 1$, hence $T(n) \leq O(\log n)$.

Exercise 4.1 *Applying Master theorem.*

For this exercise, assume that n is a power of two (that is, $n = 2^k$, where $k \in \{0, 1, 2, 3, 4, \ldots\}$).

a) Let $T(1) = 1$, $T(n) = 4T(n/2) + 100n$ for $n > 1$. Using Master theorem, show that $T(n) \leq O(n^2)$.

Solution:

We can apply Theorem 1 with $a = 4$, $b = 1$ and $C = 100$. In this case, $b < \log_2 a$, and therefore the by the Master theorem we have $T(n) \leq O(n^{\log_2 a}) = O(n^2)$.

b) Let $T(1) = 5$, $T(n) = T(n/2) + \frac{3}{2}n$ for $n > 1$. Using Master theorem, show that $T(n) \leq O(n)$.

Solution:

We can apply Theorem 1 with $a = 1$, $b = 1$ and $C = \frac{3}{2}$. In this case, $b > \log_2 a$, and therefore the by the Master theorem we have $T(n) \leq O(n^b) = O(n)$.

c) Let $T(1) = 4$, $T(n) = 4T(n/2) + \frac{7}{2}n^2$ for $n > 1$. Using Master theorem, show that $T(n) \leq O(n^2 \log n)$.

Solution:

We can apply Theorem 1 with $a = 4$, $b = 2$ and $C = \frac{7}{2}$. In this case, $b = \log_2 a$, and therefore the by the Master theorem we have $T(n) \leq O(n^{\log_2 a} \cdot \log n) = O(n^2 \log n)$.

The following definitions are closely related to O-Notation and are also useful in running time analysis of algorithms.

Definition 1 (Ω-Notation)**.** Let $n_0 \in \mathbb{N}$, $N := \{n_0, n_0 + 1, \ldots\}$ and let $f : N \to \mathbb{R}^+$. $\Omega(f)$ is the set of all functions $g : N \to \mathbb{R}^+$ such that $f \in O(g)$. One often writes $g \geq \Omega(f)$ instead of $g \in \Omega(f)$.

Definition 2 (Θ-Notation)**.** Let $n_0 \in \mathbb{N}$, $N := \{n_0, n_0 + 1, \ldots\}$ and let $f : N \to \mathbb{R}^+$. $\Theta(f)$ is the set of all functions $g : N \to \mathbb{R}^+$ such that $f \in O(g)$ and $g \in O(f)$. One often writes $g = \Theta(f)$ instead of $g \in \Theta(f)$.

Exercise 4.2 *Asymptotic notations.*

a) Give the (worst-case) running time of the following algorithms in Θ-Notation.

1) Karatsuba algorithm.

   Solution:

   $\Theta(n^{\log_2(3)})$

2) Binary Search.

   Solution:

   $\Theta(\log_2(n))$

3) Bubble Sort.

   Solution:

   $\Theta(n^2)$

b) **(This subtask is from January 2019 exam).** For each of the following claims, state whether it is true or false. You don't need to justify your answers.

claim	true	false
$\frac{n}{\log n} \leq O(\sqrt{n})$	☐	☐
$\log(n!) \geq \Omega(n^2)$	☐	☐
$n^k \geq \Omega(k^n)$, if $1 < k \leq O(1)$	☐	☐
$\log_3 n^4 = \Theta(\log_7 n^8)$	☐	☐

Solution:

claim	true	false
$\frac{n}{\log n} \le O(\sqrt{n})$	☐	☒
$\log n! \ge \Omega(n^2)$	☐	☒
$n^k \ge \Omega(k^n)$	☐	☒
$\log_3 n^4 = \Theta(\log_7 n^8)$	☒	☐

c) **(This subtask is from August 2019 exam).** For each of the following claims, state whether it is true or false. You don't need to justify your answers.

claim	true	false
$\frac{n}{\log n} \ge \Omega(n^{1/2})$	☐	☐
$\log_7(n^8) = \Theta(\log_3(n^{\sqrt{n}}))$	☐	☐
$3n^4 + n^2 + n \ge \Omega(n^2)$	☐	☐
$(*) \quad n! \le O(n^{n/2})$	☐	☐

Solution:

claim	true	false
$\frac{n}{\log n} \ge \Omega(n^{1/2})$	☒	☐
$\log_7(n^8) = \Theta(\log_3(n^{\sqrt{n}}))$	☐	☒
$3n^4 + n^2 + n \ge \Omega(n^2)$	☒	☐
$(*) \quad n! \le O(n^{n/2})$	☐	☒

Note that the last claim is challenge. It was one of the hardest tasks of the exam. If you want a 6 grade, you should be able to solve such exercises.

Solution:

All claims except for the last one are easy to verify using either the theorem about the limit of $\frac{f(n)}{g(n)}$ or simply the definitions of O, Ω and Θ. Thus, we only present the solution for the last one.

Note that for all $n \ge 1$,

$$n! \ge 1 \cdot 2 \cdots n \ge \lceil n/10 \rceil \cdots n \ge \lceil n/10 \rceil^{0.9n} \ge (n/10)^{0.9n} .$$

Let's show that $(n/10)^{0.9n}$ grows asymptotically faster than $n^{n/2}$.

$$\lim_{n\to\infty} \frac{n^{n/2}}{(n/10)^{0.9n}} = \lim_{n\to\infty} 10^{0.9n} \cdot n^{-0.4n} = \lim_{n\to\infty} (10^{9/4}/n)^{0.4n} = 0 .$$

Hence it is not true that $(n/10)^{0.9n} \leq O(n^{n/2})$ and so it is not true that $n! \leq O(n^{n/2})$.

Sorting and Searching.

Exercise 4.3 *One-Looped Sort* **(1 point)**.

Consider the following pseudocode whose goal is to sort an array A containing n integers.

Algorithm 1 Input: array $A[0 \dots n-1]$.

```
i ← 0
while i < n do
    if i = 0 or A[i] ≥ A[i − 1] then:
        i ← i + 1
    else
        swap A[i] and A[i − 1]
        i ← i − 1
```

(a) Show the steps of the algorithm on the input $A = [10, 20, 30, 40, 50, 25]$ until termination. Specifically, give the contents of the array A and the value of i after each iteration of the while loop.

Solution:

The initial state of the algorithm is:

$$A = [\mathbf{10}, 20, 30, 40, 50, 25] \qquad i = 0$$

We bolded the element $A[i]$ for convenience. In the first 5 steps, the algorithm executes $i \leftarrow i + 1$ and gets to the state $i = 5$ without changing the array.

$$A = [10, \mathbf{20}, 30, 40, 50, 25] \qquad i = 1$$
$$A = [10, 20, \mathbf{30}, 40, 50, 25] \qquad i = 2$$
$$A = [10, 20, 30, \mathbf{40}, 50, 25] \qquad i = 3$$
$$A = [10, 20, 30, 40, \mathbf{50}, 25] \qquad i = 4$$
$$A = [10, 20, 30, 40, 50, \mathbf{25}] \qquad i = 5$$

Then, in the next 3 steps, the algorithm moves the element 25 into its correct sorted position in the array:

$$A = [10, 20, 30, 40, \mathbf{25}, 50] \qquad i = 4$$
$$A = [10, 20, 30, \mathbf{25}, 40, 50] \qquad i = 3$$
$$A = [10, 20, \mathbf{25}, 30, 40, 50] \qquad i = 2$$

After that, in the next 4 steps, the algorithm again executes $i \leftarrow i + 1$ until $i = n$ and we are done.

$$
\begin{aligned}
A &= [10, 20, 25, \mathbf{30}, 40, 50] & i &= 3 \\
A &= [10, 20, 25, 30, \mathbf{40}, 50] & i &= 4 \\
A &= [10, 20, 25, 30, 40, \mathbf{50}] & i &= 5 \\
A &= [10, 20, 25, 30, 40, 50] & i &= 6
\end{aligned}
$$

(b) Explain why the algorithm correctly sorts any input array. Formulate a reasonable loop invariant, prove it (e.g., using induction), and then conclude using invariant that the algorithm correctly sorts the array.

Hint: *Use the invariant "at the moment when the variable i gets incremented to a new value $i = k$ for the first time, the first k elements of the array are sorted in increasing order".*

Solution:

We prove the hinted loop invariant by induction.

- **Base Case.**
  After the first while-loop iteration we always have $i = 1$, and the first element is trivially sorted.
- **Induction Hypothesis.**
  Assume now that the hypothesis for $1 \le k \le n$: assume that the variable i is, for the first time, equal to k, and the first k elements are sorted in increasing order.
- **Inductive Step.**
  We must show that the property holds when i becomes $k + 1$ for the first time.

  Suppose $i = k$ for the first time. Examining the algorithm, we see that the algorithm inserts $A[k]$ into $A[0 \dots k]$ by moving $A[i]$ to the left until it is in its correct place (i.e., its left neighbor is not larger). This phase is the same method as in a single phase of the InsertionSort algorithm. This makes the first $k+1$ elements sorted, as required. Then, the algorithm increments i until $i = k + 1$ (for the first time), proving the claim.

Proving this loop invariant immediately implies that, at termination when $i = n$, the first n elements are sorted, meaning that the entire array is sorted.

(c) Give a reasonable running-time upper bound, expressed in O-notation.

Solution:

Consider the above loop invariant for $i = 1, 2, \dots, n$. For each value $k \ge 1$, between the first time $i = k$ and the first time $i = k + 1$ there are $O(k)$ in-between steps. Since the algorithm terminates when $i = n$, the number of steps required is $O(1) + O(2) + O(3) + \ldots + O(n - 1) = O(n^2)$. The final running time is upper-bounded $O(n^2)$.

Remark: On a reverse-sorted array, it can be shown that the algorithm takes $\Omega(n^2)$ steps, hence the above $O(n^2)$-bound cannot be improved.

Exercise 4.4 *Searching for the summit* **(1 point)**.

Suppose we are given an array $A[1 \dots n]$ with n **unique** integers that satisfies the following property. There exists an integer $k \in [1, n]$, called the *summit index*, such that $A[1 \dots k]$ is a strictly increasing array and $A[k \dots n]$ is a strictly decreasing array. We say an array is **valid** is if satisfies the above properties.

(a) Provide an algorithm that find this k with worst-case running time $O(\log n)$. Give the pseudocode and give an argument why its worst-case running time is $O(\log n)$.

Note: Be careful about edge-cases! It could happen that $k = 1$ or $k = n$, and you don't want to peek outside of array bounds without taking due care.

Solution:

The summit index can be found using the following algorithm:

Algorithm 2 Find the summit

```
function FINDSUMMITINDEX(T, i, j)
    m ← ⌊(i + j)/2⌋
    if j = i then
        return i
    if T[m + 1] < T[m] then              ▷ m is right of the summit (or is the summit)
        return FINDSUMMITINDEX(T, i, m)  ▷ keep searching in the left half
    else                                 ▷ m is strictly left of the summit
        return FINDSUMMITINDEX(T, m + 1, j)  ▷ keep searching in the right half
Input: Valid array T of length n with unique elements
Output: FINDSUMMITINDEX(T, 1, n)
```

Let $A(n)$ be the worst-case running time of this algorithm on an input array of length n. Then, $A(n)$ is such that $A(n) \leq A(n/2) + C$ where C is a constant, since a constant number of operations are performed before a recursive call is performed on an array twice smaller. This is $A(n) \leq 1 \cdot A(n/2) + Cn^0$, hence, by the Master theorem, we have $A(n) = O(\log n)$ (case $\log a = \log 1 = 0 = b$, yielding $A(n) = O(n^b \log n) = O(n^0 \log n) = O(\log n)$).

(b) Given an integer x, provide an algorithm with running time $O(\log n)$ that checks if x appears in the array of not. Describe the algorithm either in words or pseudocode and argue about its worst-case running time.

Solution:

Consider the binary search algorithm for sorted integer arrays from the lecture. More precisely, let the binary search algorithm for arrays sorted in ascending order be denoted by $\text{BS}^{\uparrow}$, while the binary search for arrays sorted in descending order is $\text{BS}^{\downarrow}$. Assume that for $c \in \{\uparrow, \downarrow\}$, $\text{BS}^c(T, x)$ returns true if x is in T, and false otherwise. These two algorithms have running times $O(\log n)$. We can now use $\text{BS}^{\uparrow}$, $\text{BS}^{\downarrow}$, and FINDSUMMITINDEX as subroutines to find our element:

Algorithm 3 Search in a valid array

```
Input: Valid integer array T of length n with unique elements, integer x
k ← FINDSUMMITINDEX(T, 1, n)
k1 ← BS↑(T[1..k], x)          ▷ search in array T[1..k], sorted in ascending order
k2 ← BS↓(T[k + 1..n], x)      ▷ search in array T[k + 1..n], sorted in descending order
Output: k1 or k2
```

This algorithm runs in time $O(\log n) + O(\log n) + O(\log n) = O(\log n)$, since every of the three subroutines has $O(\log n)$ running times.

Exercise 4.5 *Counting function calls in loops (cont'd)* **(1 point)**.

For each of the following code snippets, compute the number of calls to f as a function of n. Provide **both** the exact number of calls and a maximally simplified, tight asymptotic bound in big-O notation.

(a)

Algorithm 4

```
i ← 0
while 2^i < n do
    j ← i
    while j < n do
        f()
        j ← j + 1
    i ← i + 1
```

Solution:

Given i, the inner loop performs $\sum_{j=i}^{n-1} 1 = (n-1) - i + 1 = n - i$ calls to f. The full algorithm thus performs $\sum_{i=0}^{\lceil \log_2 n \rceil - 1}(n - i) = n\lceil \log_2 n \rceil - \sum_{i=0}^{\lceil \log_2 n \rceil - 1} i = n\lceil \log_2 n \rceil - \frac{(\lceil \log_2 n \rceil - 1)\lceil \log_2 n \rceil}{2} = O(n \log n)$ calls to f.

(b)

Algorithm 5

```
i ← n
while i > 0 do
    j ← 0
    f()
    while j < n do
        f()
        k ← j
        while k < n do
            f()
            k ← k + 1
        j ← j + 1
    i ← ⌊i/2⌋
```

Solution:

Given i and j, the innermost loop performs $\sum_{k=j}^{n-1} 1 = n - j$ calls to f. Hence, the second loop (guarded by $j < n$) performs $1 + \sum_{j=0}^{n-1}(1 + (n - j)) = 1 + \sum_{j=0}^{n-1}((n+1) - j) = 1 + \sum_{j=2}^{n+1} j = \frac{(n+1)(n+2)}{2}$ calls to f. Finally, we observe that, if $n > 1$, the outermost loop performs exactly $\lfloor \log_2 n \rfloor + 1$ iterations: writing $n = \overline{b_\ell \dots b_0}^2$ in binary notation with $\ell = \lfloor \log_2 n \rfloor$ and $b_\ell = 1$, the variable i contains exactly $\overline{b_\ell \dots b_i}^2$ after i iterations, and is zero after exactly $\ell + 1$ of them. Hence, the full algorithm performs $(\lfloor \log_2 n \rfloor + 1)\frac{(n+1)(n+2)}{2} = O(n^2 \log n)$ calls to f.