Exercise Sheet 7: Answers
COMS10017 Algorithms 2023/2024

Reminder: $\log n$ denotes the binary logarithm, i.e., $\log n = \log_2 n$.

1 Countingsort and Radixsort

1. We use Countingsort to sort the following array A:

4	2	2	0	1	4	2

   Answer the following questions:

   (a) What is the state of the auxiliary array C after the second loop of the algorithm?

   Solution.

   $$C = 1 \quad 2 \quad 5 \quad 5 \quad 7$$

   Remark: $C[i]$ indicates how many elements in A have a value less or equal to i. ✓

   (b) What is the state of C after each iteration i of the third loop?

   Solution.

i	$C[0]$	$C[1]$	$C[2]$	$C[3]$	$C[4]$
initial	1	2	5	5	7
$i=6$	1	2	**4**	5	7
$i=5$	1	2	4	5	**6**
$i=4$	1	**1**	4	5	6
$i=3$	**0**	1	4	5	6
$i=2$	0	1	**3**	5	6
$i=1$	0	1	**2**	5	6
$i=0$	0	1	2	5	**5**

   Remark: Observe that the highlighted numbers are all different. Is this a coincidence or is this necessarily always the case?

   ✓

2. Illustrate how Radixsort sorts the following binary numbers:

   100110 101010 001010 010111 100000 000101

Solution.

100110		10011**0**		1000**0**0		100**0**00		10**0**000		1**0**0000		**0**00101
101010		10101**0**		0001**0**1		101**0**10		00**0**101		0**0**0101		**0**01010
001010	→	00101**0**	→	1001**1**0	→	001**0**10	→	10**0**110	→	1**0**0110	→	**0**10111
010111		10000**0**		1010**1**0		000**1**01		01**0**111		1**0**1010		**1**00000
100000		01011**1**		0010**1**0		100**1**10		10**1**010		0**0**1010		**1**00110
000101		00010**1**		0101**1**1		010**1**11		00**1**010		0**1**0111		**1**01010

✓

3. Radixsort sorts an array A of length n consisting of d-digit numbers where each digit is from the set $\{0, 1, \dots, b\}$ in time $O(d(n+b))$.

   We are given an array A of n integers where each integer is *polynomially bounded*, i.e., each integer is from the range $\{0, 1, \dots, n^c\}$, for some constant c. Argue that Radixsort can be used to sort A in time $O(n)$.

   Hint: Find a suitable representation of the numbers in $\{0, 1, \dots, n^c\}$ as d-digit numbers where each digit comes from a set $\{0, 1, \dots, b\}$ so that Radixsort runs in time $O(n)$. How do you chose d and b?

Solution. We encode the numbers in A using digits from the set $\{0, 1, \dots, n-1\}$, i.e., we set $b = n - 1$. To be able to encode all numbers in the range $\{0, 1, \dots, n^c\}$ it is required that $(b+1)^d \geq n^c + 1$ (we can encode $(b+1)^d$ different numbers using d digits where each digit comes from a set of cardinality $b + 1$, and the cardinality of the set $\{0, 1, \dots, n^c\}$ is $n^c + 1$). Since $(b+1)^d = n^d$, we can set $d = c + 1$, since

$$n^{c+1} \geq n^c + 1$$

holds for every $n \geq 2$ (assuming that $c \geq 1$). The runtime then is

$$O(d(n+b)) = O((c+1)(n + (n-1))) = O((c+1)2n) = O(n) ,$$

since 2 and $c + 1$ are both constants. ✓

2 Loop Invariant for Radixsort

Radixsort is defined as follows:

Require: Array A of length n consisting of d-digit numbers where each digit is taken from the set $\{0, 1, \dots, b\}$

```
1: for i = 1, ..., d do
2:     Use a stable sort algorithm to sort array A on digit i
3: end for
```

(least significant digit is digit 1)

In this exercise we prove correctness of Radixsort via the following loop invariant:

> At the beginning of iteration i of the for-loop, i.e., after i has been updated in Line 1 but Line 2 has not yet been executed, the following holds:
>
> The integers in A are sorted with respect to their last $i - 1$ digits.

1. *Initialization:* Argue that the loop-invariant holds for $i = 1$.

Solution. In the beginning of the iteration with $i = 1$ the loop-invariant states that the integers in A are sorted with respect to their last $i - 1 = 0$ digits. This is trivially true. ✓

2. *Maintenance:* Suppose that the loop-invariant is true for some i. Show that it then also holds for $i + 1$.

   Hint: You need to use the fact that the employed sorting algorithm as a subroutine is stable.

   Solution. Suppose that the integers in A are sorted with respect to their last $i-1$ digits at the beginning of iteration i. We will show that at the beginning of iteration $i + 1$ the intergers are sorted with respect to their last i digits.

   Let A_{i+1} be the state of A in the beginning of iteration $i+1$. For an integer x, let $x^{(i)}$ be the integer obtained by removing all but the last i digits from x. Suppose for the sake of a contradiction that there are indices j, k with $j < k$ such that $(A_{i+1}[j])^{(i)} > (A_{i+1}[k])^{(i)}$. If such integers exist then the loop invariant would not hold. We will show that assuming that these integers exist leads to a contradiction.

   First, suppose that digit i of $(A_{i+1}[j])^{(i)}$ and digit i of $(A_{i+1}[k])^{(i)}$ are identical. Note that this implies $(A_{i+1}[j])^{(i-1)} > (A_{i+1}[k])^{(i-1)}$. Observe that in iteration i, the digits are sorted with respect to digit i. Since the subroutine employed in Radixsort is a stable sort algorithm, the relative order of the two numbers has not changed since their ith digits are identical. This implies that the relative order of the two numbers was the same at the beginning of iteration i. This is a contradiction, since the loop invariant at the beginning of iteration i states that the digits are sorted with respect to their $i-1$ last digits, however, $(A_{i+1}[j])^{(i-1)} > (A_{i+1}[k])^{(i-1)}$ holds.

   Next, suppose that digit i of $(A_{i+1}[j])^{(i)}$ and digit i of $(A_{i+1}[k])^{(i)}$ are different. Then, since $(A_{i+1}[j])^{(i)} > (A_{i+1}[k])^{(i)}$ we have that digit i of $(A_{i+1}[j])^{(i)}$ is necessarily larger than digit i of $(A_{i+1}[k])^{(i)}$. This however is a contradiction to the fact that the numbers were sorted with respect to their ith digit in iteration i.

   Hence, the assumption that there are indices j, k such that $(A_{i+1}[j])^{(i)} > (A_{i+1}[k])^{(i)}$ is wrong. If no such indices exist then the integers in A are sorted with respect to their last i digits at the beginning of iteration $i + 1$. ✓

3. *Termination:* Use the loop-invariant to conclude that A is sorted after the execution of the algorithm.

   Solution. After iteration d (or before iteration $d + 1$, which is never executed), the invariant states that the numbers in A are sorted with respect to their last d digits, which simply means that all numbers are now sorted with regards to all their digits. ✓

3 Recurrences: Substitution Method

1. Consider the following recurrence:

$$T(1) = 1 \text{ and } T(n) = T(n-1) + n$$

   Show that $T(n) \in O(n^2)$ using the substitution method.

Solution. We need to show that $T(n) \leq C \cdot n^2$, for some suitable constant C. To this end, we first plug our guess into the recurrence:

$$T(n) = T(n-1) + n \leq C(n-1)^2 + n \ .$$

It is required that $C(n-1)^2 + n \leq Cn^2$:

$$\begin{aligned}
C(n-1)^2 + n &\leq Cn^2 \\
C(n^2 - 2n + 1) + n &\leq Cn^2 \\
C - 2Cn + n &\leq 0 \\
C(1 - 2n) &\leq -n \\
C &\geq \frac{n}{2n-1} \ .
\end{aligned}$$

Observe that $\frac{n}{2n-1} \leq 1$ holds for every $n \geq 1$. Our guess thus holds for every $C \geq 1$.

It remains to verify the base case. We have $T(1) = 1$ and $C1^2 = C$. Hence, $C1^2 \leq T(1)$ holds for every $C \geq 1$. We thus choose $C = 1$.

We have shown that $T(n) \leq Cn^2 = n^2$ holds for every $n \geq 1$. This implies that $T(n) = O(n^2)$. ✓

2. Consider the following recurrence:

$$T(1) = 1 \text{ and } T(n) = T(\lceil n/2 \rceil) + 1$$

Show that $T(n) \in O(\log n)$ using the substitution method.

Hint: Use the inequality $\lceil n/2 \rceil \leq \frac{n}{\sqrt{2}} = \frac{n}{2^{\frac{1}{2}}}$, which holds for all $n \geq 2$. Use $n = 2$ as your base case.

Solution. We need to show that $T(n) \leq C \cdot \log n$, for a suitable constant C. To this end, we plug our guess into the recurrence:

$$\begin{aligned}
T(n) &= T(\lceil n/2 \rceil) + 1 \\
&\leq C \cdot \log\left(\lceil n/2 \rceil\right) + 1 \\
&\leq C \cdot \log\left(\frac{n}{\sqrt{2}}\right) + 1 \\
&= C\log(n) - C \cdot \frac{1}{2}\log(2) + 1 \\
&= C\log(n) - \frac{1}{2}C + 1 \ ,
\end{aligned}$$

where we used the inequality $\lceil n/2 \rceil \leq \frac{n}{\sqrt{2}}$. It is required that $C\log(n) - \frac{1}{2}C + 1 \leq C\log(n)$:

$$\begin{aligned}
C\log(n) - \frac{1}{2}C + 1 &\leq C\log(n) \\
1 &\leq \frac{1}{2}C \\
2 &\leq C \ .
\end{aligned}$$

The "induction step" part of the proof thus works for any $C \geq 2$. Regarding the base case, we will consider $n = 2$. We have:

$$T(2) = T(1) + 1 = 2 \ .$$

We thus need to show that $2 \leq C \log 2$. This holds for every $C \geq 2$. We can thus pick the value $C = 2$. This proves that $T(n) \in O(\log n)$. ✓

4 Optional and Difficult Questions

Exercises in this section are intentionally more difficult and are there to challenge yourself.

4.1 Algorithmic Puzzle: Maxima of Windows of length $n/2$

We are given an array A of n positive integers, where n is even. Give an algorithm that outputs an array B of length $n/2$ such that $B[i] = \max\{A[j]\ ,\ i \leq j \leq i + n/2 - 1\}$. Can you find an algorithm that runs in time $O(n)$?

Solution. Let $C[i]$ and $D[i]$ be new arrays of lengths $n/2$. We first observe that we can rewrite $B[i]$ as the maximum of two maxima:

$$\begin{aligned} B[i] &= \max\{C[i], D[i]\}\ , \text{ where} \\ C[i] &= \max\{A[j]\ :\ i \leq j \leq n/2 - 1\}\ , \text{ and} \\ D[i] &= \max\left(\{A[j]\ :\ n/2 \leq j \leq i + n/2 - 1\} \cup \{0\}\right)\ . \end{aligned}$$

Suppose we already computed the tables C and D. Then in $O(n)$ time, we can compute the table B by computing the maxima $\max\{C[i], D[i]\}$ for every $0 \leq i \leq n/2 - 1$. It thus remains to compute tables C and D. To this end, observe that $C[n/2-1] = A[n/2-1]$, and for every $k < n/2 - 1$, we have $C[k] = \max\{A[k], C[k+1]\}$. We thus obtain the following algorithm for computing the table C:

Algorithm 1 Computing table C

```
C[n/2 − 1] ← A[n/2 − 1]
for i = n/2 − 2 . . . 0 do
    C[i] ← max{A[i], C[i + 1]}
end for
```

Similarly, observe that $D[0] = 0$, and for every $k > 0$, we have $D[k] = \max\{D[k-1], A[k + n/2]\}$. We thus obtain the following algorithm for computing table D:

Algorithm 2 Computing table D

```
D[0] ← 0
for i = 1 . . . n/2 − 1 do
    D[i] ← max{A[i + n/2], D[i − 1]}
end for
```

Computing tables C and D takes $O(n)$ time. The total runtime is therefore $O(n)$. ✓