Eidgenössische
Technische Hochschule
Zürich

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

Department of Computer Science
Markus Püschel
Peter Widmayer
Thomas Tschager
Tobias Pröger

13th October 2016

Data Structures & Algorithm — Solutions to Sheet 3 — AS 16

Solution 3.1 *Estimating Asymptotic Running Time I.*

a) To determine the number of calls of f, we first estimate the number of calls upwards. The outer loop will be traversed less than n times. The inner loop will be traversed exactly i times. As i is never greater than $n/3$, the inner loop will be traversed at most $n/3$ times. The number of calls of f is therefore at most $n \cdot n/3 = n^2/3 \in \mathcal{O}(n^2)$.

On the other hand, we note that the outer loop is traversed at least $n/9$ times. In approximately half of the cases it holds that $i > n/6$. Therefore, the inner loop will be traversed at least $n/6$ times. The total number of calls of f is therefore at least $(n/18) \cdot (n/6) = n^2/108 \in \Omega(n^2)$.

As the number of calls of f is both in $\mathcal{O}(n^2)$ and in $\Omega(n^2)$, it follows that f is called asymptotically $\Theta(n^2)$ times.

b) The outer loop is traversed exactly n times. The first inner loop starts with $j = 100$, decrements j in every step by 1, and terminates if $j^2 < 1$, i.e. $j < 1$. The loop is therefore traversed only a constant number of times! The second inner loop doubles the value of k in every step. Hence, it is traversed only $\Theta(\log(n))$ times. The function f is therefore called asymptotically $\Theta(n \cdot (1 + \log(n))) = \Theta(n \log n)$ times.

c) The outer loop is traversed exactly n times. To determine the running time of the middle loop, we note that it terminates if $j^2 > n$, or equivalently if $j > \sqrt{n}$. Therefore, it is traversed only $\Theta(\sqrt{n})$ times. The inner loop is traversed $n-1$ times. Therefore, f is called asymptotically $\Theta(n \cdot \sqrt{n} \cdot n) = \Theta(n^{5/2})$ times.

Solution 3.2 *Estimating Asymptotic Running Time II.*

a) First, we note:

- For $n = 0$ the function g is called exactly twice, hence $c(0) = 2$.
- For $n = 1$ the function g is called exactly once, hence $c(1) = 1$.
- For $n \geq 2$ there is a call of g in step 9. Moreover, we call the main function f with the input $n-1$, and a second time with the input $n-2$. Step 6 leads to $c(n-1)$ calls of g, the steps 7 and 8 each to $c(n-2)$ calls of g. Therefore, $c(n) = 1 + c(n-1) + 2 \cdot c(n-2)$.

We show using mathematical induction over n that $c(n) \geq 2^{n-1}$ holds. As $c(n)$ has two bases, we have to use *two* base cases. Analogously, we have to use general induction instead of mathematical induction, as we need induction hypotheses for $c(n-2)$ *and* for $c(n-1)$ (the classical mathematical induction only allows for one induction hypothesis for *a single* n).

Base case I ($n = 0$): It holds that $c(0) = 2 \geq \frac{1}{2} = 2^{0-1}$.

Base case II ($n = 1$): It holds that $c(1) = 1 \geq 1 = 2^{1-1}$.

Induction hypothesis: There exists an $n \in \mathbb{N}$, such that *for all* $0 \leq n' \leq n$: $c(n') \geq 2^{n'-1}$.

Inductive step ($n \to n+1$):

$$\begin{aligned} c(n+1) &= 1 + c(n) + 2 \cdot c(n-1) \\ &\overset{I.H.}{\geq} 1 + 2^{n-1} + 2 \cdot 2^{(n-1)-1} \\ &= 1 + 2^{n-1} + 2^{n-1} > 2 \cdot 2^{n-1} = 2^n = 2^{(n+1)-1}. \end{aligned}$$ ∎

b) We solve the recursion by storing the intermediate results in an array $Z[0..n]$. In the entry $Z[i]$ we store the value for $f(i)$ (for every $i = 0, \ldots, n$). That leads to the following code fragment:

Function $f'(n)$

1 Set $Z[0] \leftarrow 2 \cdot g(0)$.
2 Set $Z[1] \leftarrow g(1)$.
3 For $i = 2, \ldots, n$:
4 Compute $x \leftarrow Z[i-1] + 2 \cdot Z[i-2]$.
5 Set $Z[i] \leftarrow g(x)$.
6 Return $Z[n]$.

We note that the number of calls of g is only in $\mathcal{O}(n)$.

Lösung 3.3 *Algorithm Design: Divide-and-Conquer.*

a) A naive algorithm would compare for every $i = 1, \ldots, n$ the object O_i with all other objects O_j, $j \in \{1, \ldots, n\} \backslash \{i\}$ and check in this way if there exist at least $\lfloor n/2 \rfloor$ other objects O_j with $O_j = O_i$. The running time of this procedure is in $\mathcal{O}(n^2)$. One could argue, that it is sufficient to consider i only until $\lceil n/2 \rceil$, but that would not improve the quadratic running time in the worst case.

b) Using a divide-and-conquer approach we get the following solution: We divide the objects $O_1, \ldots, O_n$ into two sets M_1 and M_2 of the same size. We note: If the input $O_1, \ldots, O_n$ has a majority element, i.e. it occurs more than $n/2$ times, then this element has to occur more than $n/4$ times in one of the two sets (if it would occur at most $n/4$ in both sets, then it would occur at most $n/2$ times in $\{O_1, \ldots, O_n\}$). Therefore, it follows that the majority element of $O_1, \ldots, O_n$ is also a majority element of at least one of both sets M_1 and M_2.

We determine the majority element of the two sets recursively, if such an element exists. In this way, we get no, one or two candidates (i.e. possible objects) for the majority element of $O_1, \ldots, O_n$. We check for each of these candidates individually, if it is a majority element of $O_1, \ldots, O_n$ by iterating through the objects O_i and counting how often the candidate occurs.

Analysis: Let $T(n)$ be the running time of the algorithm for an input of n objects. To check if a candidate is a majority element we need $\mathcal{O}(n)$ step. Hence, we get $T(n) = 2T(n/2) + a \cdot n$

and $T(1) = c$, where a and c are constants. As seen in the lecture, the formula has a solution $c \cdot n + a \cdot \log(n) \cdot n$. Hence, the cost of the algorithm is in $\Theta(n \log n)$.

c) It is possible to find the majority element in linear time, i.e. in $\mathcal{O}(n)$: We consider the objects $O_1, \ldots, O_n$ in this order and store 1) the current candidate for the majority element, and 2) a counter. First, we set the counter to 0 and the candidate to an arbitrary object (it is not important as long as the counter is 0). We check the objects $O_1, \ldots, O_n$ and distinguish three cases:

Case 1: The current object is not equal to the candidate and the counter is 0: We set the candidate to the current object and the counter to 1

Case 2: The current object is not equal to the candidate and the counter is greater than 0: We decrease the counter by 1.

Case 3: The current object is equal to the candidate: We increase the counter by 1.

Now we show using mathematical induction over the number of objects n that the following statement holds: *If m is the majority element of $O_1, \ldots, O_n$, then our candidate will be m at the end of the algorithm and the counter is greater than 0.* Attention: This does not hold the other way around!

Base case ($n = 1$): If we are given only $n = 1$ object, the statement is trivially true.

Induction hypothesis: We assume that the statements holds for all inputs with at most n objects.

Inductive step ($n \to n + 1$): We use the induction hypothesis to show that the statement is true for inputs with $n + 1$ objects. We distinguish two cases. First, consider the case that the counter is not set to 0 after the first step. Then, the first object O_1 has to occur more often than all other objects together. As this object is the candidate at the beginning of our algorithm, the statement is true in this case. On the other hand, consider the case, where the counter is set to 0 after exactly k steps. Following the induction hypothesis, we know that no object is the majority element in the first k objects. If there exists a majority element m, it must be the majority element of the remaining $n - k$ objects. As our counter is set to the initial state after k steps, we can consider the remaining objects independently. By the induction hypothesis, the counter will be greater than 0 after traversing the remaining $n - k$ objects and m will be our candidate at the end. ∎

Is this helpful? We have seen that we can find a possible candidate for the majority element in $\mathcal{O}(n)$ time. It remains to iterate through the objects O_i in order to check if this candidate is the majority element of $O_1, \ldots, O_n$.