Algorithms and Data Structures
Jacobs University Bremen
Dr. Florian Rabe

Homework 3
given: 2017-02-21

# Homework 3

You have to submit your solutions as announced in the lecture.
**Unless mentioned otherwise, all problems are due 2017-03-02, 11:00.**
There will be no deadline extensions unless mentioned otherwise in the lecture.

---

## Problem 3.1 *Problem Complexity*

Points: 6+6+6

What is the complexity class of the worst-case time complexity $C(n)$ of the following *problems*:

1. Find the smallest element in a list of length $n$ of integers.
2. Find an unknown $x \in \mathbb{N}$ between 0 and $n$ by repeatedly asking yes/no questions about $x$.
   (Any question is allowed that has a well-defined answer. For example, you may ask "Is $x$ prime?" or "Is $x == 5$?" A solution must determine $x$ with perfect certainty in all cases.)
3. Crack an unknown password if the only prior knowledge you have is that it contains $n$ characters.

In each case, answer the question in two parts:

- Give an algorithm (in pseudo-code or a programming language) that is in the given complexity class. (3 points each)
- Argue informally (but convincingly) why there can be no better algorithm. (3 points each)

---

**Solution:** Solutions had to actually give the algorithms. Here the algorithms are only sketched.

1. $\Theta(n)$
   - It is straightforward to give a linear algorithm.
   - The smallest element can be any element in the list, and there is no relation between elements in different positions. Therefore, the optimal algorithm must inspect every element of the list, and thus be at least linear.
2. $\Theta(\log_2 n)$
   - We use give an algorithm such that every question cuts the range of possible values for $x$ in half. For example, we can ask "Is $x \leq n/2$?" If yes, we recurse for the lower half; if no, we recurse for the upper half. After $k$ questions, the range is reduced to $n/2^k$. Thus, we know $x$ after $\log_2 n$ questions.
   - There are $n$ possible solutions. Therefore, the optimal algorithm must be such that it can return $n$ different values. But the only data that can be used to return different values are the answers to the questions. After $k$ questions we have received $2^k$ possible sequences of answers. Therefore, we must have $2^k \geq n$ to be able to return $n$ different values. Thus, we have to ask at least $\log_2 n$ questions.
3. $\Theta(c^n)$ for a constant $c$. Note that there are $c^n$ possible passwords if $c$ is the number of characters allowed in the password.
   - A brute-force attack cracks the password by trying every possibility. That takes $c^n$ attempts in the worst-case.
   - Any password could be the right one. Because we have no other information, even the optimal algorithm might try the correct password last in the worst-case.

---

## Problem 3.2 *Complexity Analysis*

Points: 5

The following is a (highly simplified variant of an) example that came up last year in the instructor's research: Consider the following function in the Scala programming language:

```
def processString(s: String) {
  var rest: String = s
  while (rest != "") {
    if (rest.startsWith("foo")) {
      // does not matter, assume this takes O(1)
    } else {
      // does not matter, assume this takes O(1)
    }
    rest = rest.substring(1)
  }
}
```

Here `startsWith` and `substring` are methods on strings from the Java library (which Scala can call with only constant-time overhead).

Let $C(n)$ be the run time of this function where $n$ is the length of the input string. What is the complexity class of $C(n)$ and why?

You can try the program yourself on increasingly large input to find out. (If you prefer, you can use Java instead of Scala—the effect is the same.) In the instructor's case, the surprising effect was noticed by a student trying to process a 100 MB text file, e.g., when $n > 10^8$. The problem is already noticeable for smaller values of $n$.

---

**Solution:** The complexity depends on the implementation of `substring` in the Java standard library.

The simplest implementation always copies the string, `rest = rest.substring(1)` is linear in the length of `rest`. Then the algorithm is quadratic.

But it is possible to implement `substring` efficiently so that it always takes constant time. The trick is not to copy the original string and instead store every string in terms of its start and end position. Then the algorithm is linear.

It turns out an earlier version of the standard Java library had implemented `substring` efficiently. But the current version had switched to the simpler implementation. That confusion caused the instructor to mistakenly believe his implementation to be linear when it actually was quadratic.

We noticed it when trying to parse a 100 MB JSON file. Parsing JSON is a straightforward linear operation that should be faster than reading the file from the hard drive. But it took 30 minutes. After writing a data structure that allowed for constant-time substring operations, parse time went down to about a second.

---

## Problem 3.3 *Polynomial Algorithms* Points: $\infty$

Consider the following problem: Given a natural number $x$ that uses $n$ bits, find a non-trivial factor of $x$ (or say that $x$ is prime).

Here "non-trivial factor" means a number $p|x$ such that $1 < p < x$.

Give an algorithm (pseudo-code or programming language) with polynomial worst-case time complexity $C(n)$ or show that no such algorithm exists.

---

**Solution:** This was a bit of a trick question—the complexity of factorization is not known.

We know that factorization is at most exponential: we can simply test all numbers $p$ from 2 to $\sqrt{2^n}$. Moreover, given a potential factor $p$ of $x$, we can test $p|x$ in at most quadratic time by doing the division. Finally, we know that testing whether $x$ has any non-trivial factor is polynomial (using the advanced AKS algorithm discovered in 2002).

But we do not know if there is a polynomial algorithm that can find a non-trivial factor of $x$ (even if we know $x$ has one). It is generally assumed that there is none.

Finding such an algorithm might make many public key encryption schemes vulnerable to attacks. Their security is often based on the property that a brute-force attack is at least as complex as factorization. With factorization assumed to be super-polynomial, these attacks are assumed to be infeasible.

Showing that there is no such algorithm would immediately prove the $P \neq NP$ conjecture—one of the most famous open problems in computer science.

However, we know that any complexity analysis depends on assumptions about the underlying physical machines. In 1994, Shor found a polynomial factorization algorithm running on a then-hypothetical quantum computer. This is one of the motivations of researchers, industry, and military/intelligence agencies to build quantum computers. This has been accomplished by now at small scales: the current quantum computation record factored numbers for $n = 16$. But the physical difficult and financial cost of building and running quantum computers is still large.

---