Olympiads in Informatics, 2017, Vol. 11, 77–86

DOI: 10.15388/ioi.2017.06

An Introduction to Running Time Analysis for an SOI Workshop

Dennis KOMM, Tobias KOHN
Department of Computer Science, ETH Zürich
Universitätstrasse 6, 8092 Zürich, Switzerland
e-mail: {dennis.komm,tobias.kohn}@inf.ethz.ch

Abstract We organize an introductory course on algorithm design and complexity analysis for prospective participants of the Swiss Olympiad in Informatics and interested high school students. The students are assumed to have some background in programming, but no formal computer science education.

Our goal for the first lesson is to introduce them to the basic tools used in running time analysis, with a particular emphasis on worst-case versus best-case analysis, uniform versus logarithmic cost measurement, and big-O notation; however, we avoid being too formal and use the example of primality testing to introduce the concepts hands-on. In this paper, we describe our approach in detail.

Keywords: SOI workshop, running time analysis, worst-case analysis, cost measurement, big-O notation

1. Introduction

We organize an introductory course on algorithm design and analysis for high school students. The course was originally intended for participants of the SOI (Swiss Olympiad in Informatics). However, it has become an important part of our outreach program, and now, all interested high school students with basic programming skills are welcome. This also allows us to encourage gifted students in our course to participate in the SOI. Many prospective contestants primarily lack the necessary self-confidence rather than the skills to try and participate in an Olympic contest. Hence, opening the SOI classes to all interested students might lower the barrier for some students to take part in the SOI.

One particular challenge of the course is to give a proper introduction to algorithm analysis to an audience that has only basic knowledge in programming, and is not familiar with the formal analysis of algorithms and their running times. At the same time, we regard the issue of time complexity as a central aspect of algorithmic design and problem solving strategies.

The attending students are typically around 16 to 18 years old. In order to attend our course, we require the students to have some basic training and experience in programming. The material presented in this paper covers a class of 90 minutes. During class, the presentation of the material is interleaved with hands-on exercises. Students are given some time to work individually on the given tasks, using their own laptops.

Our approach is to use an easy computational problem to demonstrate principles such as time complexity, best- and worst-case analysis, uniform and logarithmic cost measurement, and big-O notation in the first lecture of the course. In what follows, we describe the steps taken in detail, based on the teaching material supplied.

The programming languages used are both Python and C++. Since all students have experience in programming with Python (a prerequisite of the course), we decided to start with that language and switch to C++ at a later lecture; in particular, we use TigerJython, which has been developed by one of the authors (Kohn, 2017).

1.1. *Overview*

In order to discuss the theoretical concepts mentioned above, we chose the example of determining whether a given integer is a prime number. With some simple steps, the basic algorithm of testing all possible divisors can easily be improved. First, we can move to testing odd numbers only (after an initial test if the given integer is even). Second, the number of divisors to test can be cut down further by observing that it suffices to test up to the square root of the given integer.

This task can be used to discuss measuring the time complexity of an algorithm with respect to the input length, and introduce big-O notation. Moreover, this example also allows us to discuss the notions of average-case and worst-case scenarios, giving more depth to the idea of measuring time complexity.

1.2. *Related Work*

While the IOI is clearly recognized for its contests, the goals and benefits of the IOI movement reach much further. Dagienė notes that "the high-level goal of the IOI is to promote computer science among the youth, and to stimulate their interest in programming and algorithms" (Dagienė, 2010). As such, a most important part are training classes and outreach programs, spurred by the IOI movement and its international contests. The high value of the IOI for CS education is also confirmed by, e.g., Sysło: "The Olympiad conducts intensive educational activities" (Sysło, 2011).

There is general agreement that programming is a good starting point for CS education, even though some noteworthy exceptions exist (e.g., Bell *et al.* presented sophisticated materials which do not require any prior programming skills (Bell *et al.*, 2009). However, programming by itself is not enough (Dagienė, 2010), but must be completed by farther-reaching topics.

When it comes to continuing CS education beyond programming, on one hand, problem solving skills are often seen as the core competence to be achieved by students (cf., e.g., Wing, 2006). On the other hand, though, the ability to evaluate a program's properties such as complexity/efficiency and correctness (cf., e.g., Sysło, 2011) is a prerequisite for improving algorithms and finding more efficient solutions. Hence, our starting point with a discussion of complexity is a first step to a more thorough discussion of algorithms and problem solving strategies in general.

For students, it is particularly important to clearly motivate the discussed topics, and build on examples and practical activities (Dagienė, 2010). A good "connection between practice and theoretical concepts" (Dagienė, 2010) is key in fostering the students' understanding of both theory itself and its relevance.

2. Explaining Complexity

We start our class by discussing a typical task of a computer scientist, which is to answer whether a given natural number is a prime number. It is easy to motivate the problem, as it is well known by most students, and some of them are usually already familiar with methods such as, e.g., the *sieve of Eratosthenes* (cf., e.g., O'Neill, 2009; Sedgewick, 1992). However, our aim is not to present the best solution for the problem (even if this were possible within the time boundaries of the course), but to improve the straightforward solution step by step, while introducing the aforementioned tools of algorithm analysis at the same time. Therefore, we start with the simplest possible way to determine whether a given natural number `x` is prime, i.e., we consider the algorithm shown in Algorithm 1, which "answers" whether `x` (which we assume to be at least 3) is a prime number. For such a given `x`, the algorithm simply checks whether it is divided by any number between 1 and `x`, i.e., whether it is divided by $2, 3, \ldots,$ or $\mathtt{x} - 1$. If such a number is found, it answers "Composite," otherwise it answers "Prime." Naturally, we want to assess how long the algorithm takes to compute a result; we call this the algorithm's *time complexity*. Thus, the first question we ask is how to quantify this time.

Algorithm 1. Determining whether a given number is prime

```
  def prime(x):
1.      divisor = 2
2.      while (divisor < x):
3.              rest = x % divisor
4.              if (rest == 0):
5.                      print "Composite."
6.                      return
7.              divisor += 1
8.      print "Prime."
```

A straightforward approach is to measure the absolute time taken by the program's execution in, say, milliseconds. However, a statement such as "this algorithm takes 15 milliseconds on this instance" is not very meaningful since this time obviously depends on the machine on which the algorithm is executed. It is rather undesirable to be limited to comparing two algorithms only if they are run on the same machine. This is easy to see for the students. Another point is that the programming language matters; if the students are already familiar with basic C++, they can verify this themselves by implementing the same algorithm in both Python and C++ and run it on the same input. We argue that, consequently, another more robust approach is needed, which gives better insight into an algorithm's performance. The key observation is that computational problems usually require more time to be solved if the input length increases. The main question is what this increase looks like; is it linear, quadratic, cubic, exponential, ...?

Now, we discuss with the students that, if we would invoke `prime(x)` with `x` being set to 100 003 (which is prime), the body of the `while`-loop is executed roughly 100 000 times; if `x` is increased and still prime, the algorithm surely takes longer and longer.

Next, we give a small introduction to encoding natural numbers in binary; to most students, this is already known and needs no detailed description. With n bits, we can encode a number between 0 and $2^n - 1$, which means that encoding the number `x` takes roughly $n \approx \log_2 \mathtt{x}$ bits. Therefore, executing Algorithm 1 with prime input `x` takes a time that grows with 2^n with n denoting the binary length of `x`. Of course, in each execution of the loop, there are a number of computational steps involved, which we need to account for, but for now we just focus on the number of times the body of the loop is executed.

It is easy to see that Algorithm 1 does the job of testing a number for primality, but it also does quite some unnecessary work. We first argue that it suffices to only test odd numbers between 1 and `x` (after testing whether `x` is divisible by 2), which reduces the number of loop executions by a factor of roughly 2.

While this improvement is easy to see, the next step, namely that we only need to test whether `x` is divisible by a number of at least 2 and at most $\lfloor\sqrt{\mathtt{x}}\rfloor$, needs a little more thinking. Most students are not familiar with the formal concept of a proof by contradiction. Yet it is possible to argue that there cannot be two divisors a and $b = \mathtt{x}/a$ that are both larger than $\lfloor\sqrt{\mathtt{x}}\rfloor$. We conclude that, if the input `x` is not a prime number, we will always find a number between 2 and $\lfloor\sqrt{\mathtt{x}}\rfloor$ that divides it. If we do not find such a number, then there is no other divisor, and `x` is therefore prime. This idea is incorporated in Algorithm 2, which we usually have the students discover themselves. A typical mistake is to use "<" instead of "<=" in line 2 of Algorithm 2. Note that this implementation does not only test divisibility by odd numbers. This is done on purpose, because we want to compare this improvement to the above constant-factor improvement with respect to the naive algorithm.

We also ask the students to reason about the time complexity of their new algorithm in terms of how often the loop is executed now, which is roughly $\sqrt{\mathtt{x}}$ times, and hence $\sqrt{2^n} = 2^{n/2} \approx 1.414^n$.

We discuss the implications of this improvement in class with our students. To this end, we give a demonstration that is inspired by the book of Cormen *et al.* (Cormen *et al.*, 2009). Assume we run both Algorithms 1 and 2 on a computer that is able to

Algorithm 2. Determining faster whether a given number is prime (I)

```
from math import sqrt

  def prime(x):
1.      divisor = 2
2.      while divisor <= sqrt(x):
3.              rest = x % divisor
4.              if (rest == 0):
5.                      print "Composite."
6.                      return
7.              divisor += 1
8.      print "Prime."
```

process $1\,000\,000$ executions of the `while`-loop per second. Suppose further that we execute Algorithm 1 with the prime number `x` $= 100\,000\,000\,000\,031$, which means that we need roughly

$$\frac{100\,000\,000\,000\,031 \text{ iterations}}{1\,000\,000\ \frac{\text{iterations}}{\text{second}}} \approx 100\,000\,000 \text{ seconds} \approx 3 \text{ years}$$

to get a result. Using Algorithm 2 instead yields a running time of roughly

$$\frac{\sqrt{100\,000\,000\,000\,031} \text{ iterations}}{1\,000\,000\ \frac{\text{iterations}}{\text{second}}} \approx 10 \text{ seconds.}$$

Why we should care so much about efficiency can be illustrated even more impressively as follows. Assume that we run Algorithm 1 on a computer that is, say, $1\,000$ times faster than before. Then we get a time of

$$\frac{100\,000\,000\,000\,031 \text{ iterations}}{1\,000\,000\,000\ \frac{\text{iterations}}{\text{second}}} \approx 100\,000 \text{ seconds} \approx 1 \text{ day and } 3 \text{ hours},$$

which is therefore still a lot more than what Algorithm 2 takes on the much slower computer.

We can also do it the other way around. Suppose we have 10 minutes to spend, and ask what the maximum size is of a number `y` we can test. Using Algorithm 1, we obtain

$$\frac{\mathtt{y} \text{ iterations}}{1\,000\,000\ \frac{\text{iterations}}{\text{second}}} \leq 600 \text{ seconds},$$

which, solving for `y`, gives `y` $\leq 600\,000\,000$. Using Algorithm 2, we get

$$\frac{\sqrt{\mathtt{y}} \text{ iterations}}{1\,000\,000\ \frac{\text{iterations}}{\text{second}}} \leq 600 \text{ seconds},$$

and hence we can test numbers y up to $600\,000\,000^2 = 3\,600\,000\,000\,000\,000\,000$. Given that we motivate Algorithm 2 through a time bound based on the number x itself rather than its length, it might not immediately be clear why one should use the input length as a measurement for the time complexity at all. Two related problems, however, can help in this matter. Determining whether a number is divisible by 2 has constant complexity as it does not depend on the length of the input at all. Divisibility by 3, on the other hand, is tested by summing over the digits of the input number, resulting in linear running time. In the context of these examples, time complexity based on the number of (binary) digits becomes quite natural.

3. Best-Case and Worst-Case Analysis

Next, we present Algorithm 3 to the students as a variation of Algorithm 2, but instead of terminating right after finding a divisor of x (performing an "early exit"), it sets a Boolean variable isprime to False, which was previously initialized with True. We ask which of these two algorithms is "better"? Although probably everyone would agree that Algorithm 2 is superior to Algorithm 3 since the latter may perform unnecessary operations, the answer to the question is actually not that easy. The reason is that there are different ways to analyze the time complexity of algorithms. If Algorithm 3 is executed with x = 100 000, its while-loop is executed roughly 100 000 times, whereas Algorithm 2 terminates at the very beginning, namely after finding out that x is even. However, if x is prime, both implementations take roughly the same time. Thus, we discuss with the students that, in order to answer whether one of the algorithms is "better," we first need to fix what kinds of inputs we look at, which we describe as follows.

- **Best-case analysis.** Here, we analyze the given algorithm's time complexity on inputs of given length that are in a sense as "favorable" as possible. For Algorithm 2,

Algorithm 3. Determining faster whether a given number is prime (II)

```
  def prime(x):
1.      divisor = 2
2.      isprime = True
3.      while divisor <= sqrt(x):
4.              rest = x % divisor
5.              if (rest == 0):
6.                      isprime = False
7.      if (isprime):
8.              print "Prime."
9.      else:
10.             print "Composite."
```

this would be numbers with a divisor of 2. If Algorithm 3 is given such a number of length n, it still executes the body of its `while`-loop roughly 1.414^n times. Actually, this is the case for every input of length n.

- **Worst-case analysis.** In this case, we analyze the "least favorable" instances, i.e., those that make the algorithm run as long as possible. For Algorithm 2, these are prime numbers or squares of prime numbers; in both cases, the `while`-loop is executed roughly 1.414^n times, which is now the same time Algorithm 3 takes.
- **Average-case analysis.** We can also analyze the behavior of the algorithms on average. Since, for Algorithm 3, the time complexity is the same for every input of a given length, this is both easy and not very meaningful. Conversely, the behavior of Algorithm 2 is different on different inputs. However, here, we would first have to fix what we mean by "average." We could, e.g., assume that all inputs appear with the same probability. Then again, from a practical perspective, some inputs may be more likely than others. Therefore, it depends on the concrete environment in which the algorithm is executed to determine what a typical input looks like.

We usually design algorithms with their worst-case time complexity in mind; as noted above, with respect to this measure, Algorithms 2 and 3 are "equally good." Of course, we stress that implementing Algorithm 3 instead of Algorithm 2 is still a bad idea since Algorithm 2 is clearly faster in case of non-worst-case instances.

4. Uniform and Logarithmic Measurement

So far, we have only considered the number of executions of the `while`-loop when speaking about the algorithm's time complexity. As in the example above, we are interested in a function that describes how the running time grows with the input length n. Now we take a closer look at the work carried out by the algorithm, i.e., the number of *elementary instructions* within the loop. By this we mean arithmetic operations (treating addition and multiplication equally as one operation each) and, e.g., comparing two numbers. We briefly introduce the following two measurements on an informal level.

- **The uniform cost measurement.** When applying this measurement, we account cost 1 to every elementary instruction carried out by the machine executing the algorithm. The above algorithms perform a number of instructions in every execution of the `while`-loop that does not depend on n; specifically, in every iteration, `divisor` is increased and compared with $\sqrt{\texttt{x}}$, `x` is divided by `divisor`, and then compared to 0. We can therefore say that the number of computational steps is roughly $4 \cdot 1.414^n$ (plus a constant number of instructions at the beginning and the end of the program).
- **The logarithmic cost measurement.** The obvious problem with the uniform cost measurement is that it does not account for the sizes of the numbers that are in-

volved. To allow for a more accurate measurement, we can thus account a number of computational steps to instructions involving this number that is equal to its length. Since `x` is represented with n bits and `divisor` increases until it is roughly $\sqrt{\texttt{x}}$ (which can be bounded by at most n bits), the analysis yields roughly $4n \cdot 1.414^n$.

We conclude that the logarithmic cost measurement makes sense if the numbers considered are unbounded and may possibly be many times as large as the registers of the computer. In most of the examples presented later in class, such as sorting a sequence of given numbers, the input numbers can be represented with a few bits, and we will thus use the uniform cost measurement.

5. Big-O Notation

Our last goal of the first lecture is to introduce big-O notation on an informal level; the aim is not to give a mathematical rigorous definition. We again consider the aforementioned time complexity of roughly $4 \cdot 1.414^n$, and argue that, with growing n, 1.414^n is the dominant factor of the expression, and the constant 4 becomes less and less significant with respect to its magnitude. The same is true for any other constant as well. What we do want to distinguish is whether the complexity grows, e.g., linearly, polynomially, or exponentially.

To this end, all functions that grow, say, almost quadratically, are contained in a set $\mathcal{O}(n^2)$. The important thing is for the students to really think of $\mathcal{O}(n^2)$ as an infinite set that contains, e.g., $1.52n^2$, $4n^2$, $20n^2$, or $1000n^2$. All these functions are contained in this set. Hence, we say "$56n^2$ is in $\mathcal{O}(n^2)$" and simply write $56n^2 \in \mathcal{O}(n^2)$, and so on. Furthermore, we observe that the function n^2+n can be bounded from above by $2n^2$, and thus also $n^2 + n \in \mathcal{O}(n^2)$; likewise, $17n \leq 17n^2$ and thus $17n \in \mathcal{O}(n^2)$. The same is true for $2n^2 + 6n^{1.5}$, or $2.75n^2 + \sqrt{2}n + 9$, and so on. We explain that it suffices if this is true for sufficiently large n. Finally, we discuss that sets as above can be defined for other functions (which we always assume to be positive and monotonically increasing), giving a few examples.

It follows that the worst-case time complexity of Algorithm 1 is in $\mathcal{O}(2^n)$ with respect to the uniform cost measurement, and the complexity of Algorithms 2 and 3 is in $\mathcal{O}(1.414^n)$. The intuition is finally underpinned with a few more examples of different algorithms and their time complexity.

One way to conclude this lecture is to discuss the existence of randomized primality testing such as the algorithm of Solovay and Strassen (Hromkovič, 2008), or the deterministic AKS algorithm (Agrawal *et al.*, 2004), which achieves a running time in $\mathcal{O}(n^d)$, for a constant d.

6. Conclusion

We described an example of how to introduce the concepts of running time, best- and worst-case analysis, uniform and logarithmic cost measurement, and big-O notation. To this end, we used the simple example of primality testing. We have so far tested this approach with different classes and settings, with the common property that the students had implemented algorithms before, but had no solid background in the formal analysis of algorithms.

Our experience is very positive. We found that, often, during presentation of our material, some students would quickly point out that the initial algorithm could be improved upon by focusing on odd numbers only, or even that testing possible divisors up to the square root would suffice. However, after having seen the context of the material, it became clear to these students that the goal of our class was not in finding the best solution for primality testing, but rather in discussing the basic concepts of complexity and what it means to improve an algorithm.

After this introductory lecture, the students were all able to use terms such as "asymptotic worst-case complexity" correctly on an intuitive level. Subsequent lectures discussed topics such as sorting or graph algorithms, with the students being able to understand and express the complexities of the different algorithms presented.

References

Agrawal, M., Kayal, N., Saxena, N. (2004). PRIMES is in $\mathcal{P}$. *Annals of Mathematics*, 160(2),781–793.

Bell, T., Alexander, J., Freeman, I., Grimley, M. (2009). Computer science unplugged: school students doing real computing without computers. *NZ J. Appl. Comput. Inf. Tech.* 13(1), 20–29.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2009). *Introduction to Algorithms*, 3rd edition. The Mit Press.

Dagienė, V. (2010). Sustaining informatics education by contests. In: *Proceedings of the 4th International Conference on Informatics in Secondary Schools – Evolution and Perspectives: Teaching Fundamentals Concepts of Informatics*. 1–12.

Hromkovič, J. (2008). *Design and Analysis of Randomized Algorithms: Introduction to Design Paradigms*. Springer.

Kohn, T. (2017). *Teaching Python Programming to Novices: Addressing Misconceptions and Creating a Development Environment*. PhD Thesis, ETH Zürich.

O'Neill, M.E. (2009). The genuine sieve of eratosthenes. *Journal of Functional Programming*, 19(1), 95–106.

Sedgewick, R. (1992). *Algorithms in C++*. Addison-Wesley.

Sysło, M. (2011). Outreach to prospective informatics students. In: *Proceedings of the 5th International Conference on Informatics in Schools: Situation, Evolution and Perspectives*. 56–70.

Wing, J.M. (2006). Computational thinking. *Commun. ACM* 49(3), 33–35.

D. Komm is lecturer at ETH Zurich and an external lecturer at University of Zurich. He studied computer science at RWTH Aachen University and Queensland University of Technology. He received his PhD from ETH Zurich in 2012. His research interests focus on algorithmics and advice complexity.

T. Kohn is a PostDoc researcher in computer science education at ETH Zurich. The focus of his research is programming education, particularly in high schools. He holds an MSc in mathematics, and a PhD in computer science from ETH, and has been teaching mathematics and computer science for 10 years.