

Algorithm Quality in A Nutshell

Terry Yin

October 26, 2014

How do you put an elephant into a refrigerator?

- step 1: Open the refrigerator;
- step 2: put in the elephant;
- step 3: close the door.

We just created an algorithm for putting an elephant into a refrigerator. But, somehow, it feels not very practical. How do we determine the quality of an algorithm?

1 Does it work

Correctness is for sure an important part of the algorithm quality. How can we verify if (the representation of) an algorithm is correct?


Accounting is called the “language of business”, and is known to be precise. How do they ensure the correctness of their bookkeeping? Accountants usually use an approach called **double-entry bookkeeping**. For instance, recording earnings of \$100 would require making two entries: a debit entry of \$100 to an account called “Cash” and a credit entry to an account called “Income” [Wikipedia \(2014a\)](#). In the end, everything should be balanced. If not, they immediately know there’s a problem.

We should have this kind of **redundancy** for the algorithm as well. Besides the representation of the algorithm, which is the code, we should also have a representation for our intent by the algorithm.

Traditional, the representation of the intent of the algorithm is often in the form of a **requirement specification** or **test plan**. But the problem of static documentation is they often “lie”. It’s hard to check if they are documenting the actual thing. And they tend to become outdated very soon.

Therefore, we need a representation of the algorithm’s intent that doesn’t lie. That could be **unit testing**, which is some computer program checking the correctness of another computer program [Wikipedia \(2014e\)](#).

Below is an example of the algorithm of determining the **leap year**:



```
def leap_year(year):  
    if year % 4 != 0: return False  
    if year % 100 != 0: return True  
    if year % 400 != 0: return False  
    return True
```

Below is some unit test to check the intent of the algorithm of the leap year:



```
assert not leap_year(1999)
assert leap_year(2008)
assert leap_year(2000)
assert not leap_year(2100)
```

2 Internal quality of an algorithm

“An algorithm is an ordered set of unambiguous, executable steps that define a terminating process” (Brookshear 2011, p. 189). The representation of an algorithm also needs some abstraction to avoid too much detail.

A generally good abstraction might not necessarily be a part of a good algorithm. Abstraction helps people understand ideas. Sometimes, it doesn't even have to be right. For example, the arrival flight sign in most airport looks like this (The symbol next to the arrow on the left):



Figure 1: Signboard for arrival flight. September 2014, Suvarnabhumi Airport, Bangkok by Terry Yin

This is a good abstraction. Everybody will understand what does it mean immediately. But the fact is no airplane lands like that. It looks more like crashing. It's unimaginable if anybody implements landing according to that abstraction. To make the representation more precise, computer science established the basic building blocks called **primitive** (Brookshear 2011, p. 190).

But too much detail will also prevent people from understanding the algorithm. Luckily, any programming language has some ability of abstraction, e.g. extra the details into a sub-routine. So a good algorithm should be **detailed appropriately**. Or following the **Single Level of Abstraction Principle**.

Probably we should try to avoid using pseudo-code to represent an algorithm, when it's possible to use a real programming language. The algorithm is part of the software design. As Jack Reeves pointed out:

“After reviewing the software development lifecycle as I understood it, I concluded that the only software documentation that actually seems to satisfy the criteria of an engineering design is the source code listings.” Reeves (1992)

3 Efficiency of an algorithm

Our algorithm might be able to provide the functionality we need at a small scale. But will it work efficient enough when the input scales up? The efficiency is often also a part of our intent for an algorithm, which is often called **non-functional requirement** Wikipedia (2014c). There are many aspects in non-functional. Here we will only look at the **time complexity** and the **space complexity**.

3.1 How long will it run before getting the result?

An obvious quality of the algorithm is how fast it is. That is determined by the time complexity of the algorithm. Computer has been doubling its speed every 18 months, [Wikipedia \(2014b\)](#). But that's still often not enough to make an algorithm does what it couldn't within a reasonable amount of time. Because the time complexity of the algorithm could increase very fast as the amount it's dealing scales up. The time complexity of the algorithm is often identified by the **big-theta notation** like $\Theta(n^2)$ ([Brookshear 2011](#), p.226), or **big-O notation** like $O(n)$ [Wikipedia \(2014d\)](#).

Say, you come to a typical American family, it's important to remember everybody's *name*. You will need to call those people by name in the future. So the complexity is $O(n)$. For a family with n people, the time you spend on remembering the names will be proportional to n .

Say, you come to a typical Chinese family, it's important to know the *relationship* between each two of them. Because, well, one day you will need to say "oh, you are Zhangsan's 3rd cousin and son of Zhangsan's mother's 2nd sister." This time, the complexity is $O(n^2)$. As you can see, the time you spend on remembering the relationship of a Chinese family increase much faster than just remembering the names as the number of family member increases.

But, if you study the Chinese family deep enough, you will find that there are tricks to help you remember the relationships between each family members. You don't have to remember each relationship, you only need to remember the **family tree**. Now the complexity of remembering the relationship is $O(n)$ again. And when you need to recall the relationship between any two family members, the complexity is $O(\lg n)$.

The interesting finding here is, $O(\lg n)$ increases slower than $O(n)$. So that means getting to know a large Chinese family is actually easier than getting to know a large American family.

3.2 How much space will it take

Space complexity is often another concern of algorithm, because computers have limited memory (and other resources). Time complexity and space complexity are sometime tradeable. For example, in the above Chinese family example, we use extra space in our memory to remember the relation names in a full family tree. So that we can reduce the complexity of searching the relationship between any two members from $O(n^2)$ to $O(\lg n)$.

4 Conclusion

As Kent Beck pointed out, there are three steps in designing a piece of software [c2.com \(2014\)](#):

- Make it work
- Make it right
- Make it fast

"Make it work" means to make it work according to the functional requirement. "Make it right" means to have the proper internal structure. "Make it fast" means to make it work according the non-functional requirement. And the order is very important. Trying to make it fast before it has the right structure will only lead to something that doesn't work and not fast, and of course, have a messy internal structure.

Regarding the elephant story: I didn't make the story. But I heard it too long ago, and I don't know who's the author of it. There's a second part (and more) of the story. How do you put a giraffe into a freezer?

References

Brookshear, J. G. (2011), *Computer science: an overview*, Paul Muljadi.

c2.com (2014), ‘Make it work make it right make it fast’. [Online; accessed 26-October-2014].

URL: http://en.wikipedia.org/w/index.php?title=The_Elements_of_Programming_Style&oldid=614175936

Reeves, J. W. (1992), ‘What is software design’, *C++ Journal* **2**(2), 14–12.

Wikipedia (2014a), ‘Double-entry bookkeeping system — wikipedia, the free encyclopedia’. [Online; accessed 26-October-2014].

URL: http://en.wikipedia.org/w/index.php?title=Double-entry_bookkeeping_system&oldid=629384284

Wikipedia (2014b), ‘Moore’s law — wikipedia, the free encyclopedia’. [Online; accessed 20-September-2014].

URL: http://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=626293387

Wikipedia (2014c), ‘Non-functional requirement — wikipedia, the free encyclopedia’. [Online; accessed 26-October-2014].

URL: http://en.wikipedia.org/w/index.php?title=Non-functional_requirement&oldid=627025279

Wikipedia (2014d), ‘Time complexity — wikipedia, the free encyclopedia’. [Online; accessed 26-October-2014].

URL: http://en.wikipedia.org/w/index.php?title=Time_complexity&oldid=630268634

Wikipedia (2014e), ‘Unit testing — wikipedia, the free encyclopedia’. [Online; accessed 26-October-2014].

URL: http://en.wikipedia.org/w/index.php?title=Unit_testing&oldid=624378949