Mohammed Hayat May 1, 2019

Code is at: https://github.com/mkh3/scratch/tree/master/cgi_exercise

Problem 1:

Assumptions:

- I have assumed that by 'number', you mean natural numbers.

Methodology:

The naive approach would be to iterate overall numbers from 1 to x, and check if each is a divisor. But the highest factor that any number can have (apart from itself) is 'half' of that number. (Since apart from 1, 2 is the lowest factor a number can have)

So the search space can be pruned by finding the 'half' of N, and if x happens to be higher than that, then iterate fromthe half-way point, rather than X.

Example:

If N = 72 and x= 50, i.e. we need to find all factors of 72 that are less than or equal to 50. Naively we would want to iterate down from 50 to 1, checking if each was a factor. But we can prune the space by iterating from 36 (72/2) as there won't be any factors of 72 between 36 and 50.

Problem 2:

Assumptions:

- The implementation assumes that the balloons are stationary, otherwise the code would need to implement trajectories. So all balloons that could collide have *already collided*, or more strictly, occupy the same position in the 1-d space.
- I have assumed that the *Balloon* objects are present in an ordered collection. In the code this is an ArrayList.

Approach:

- We are checking for combinations of balloons that occupy the same point.
- If there were just 2 balloons, just 1 check would be needed: take the 2^{nd} balloon and check if it has collided with the 1^{st} balloon.
- If there were 3 balloons, then take the 3^{rd} one, check if it has collided with the 2^{nd} and then the 1^{st} one. Then take the 2^{nd} balloon and check it against the 1^{st}. That's it. 2+1 comparisons needed.
- Extrapolating for n balloons, for the n^{th} balloon, we check it against all n-1 other balloons; then for the $n\text{-}2^{th}$ balloon, it is checked against the 'lower' (this is why an ordered collection is needed) n-3 balloons.
- A map from Location to lists of Balloons is maintained.

Problem 3:

My approach here is naïve. I am simply sweeping the array for the largest element and then removing it. To find the k highest elements, I simply perform such sweeps k times.

This works when k<<N. If k is comparable to N, then the time complexity approaches $O(N^2)$, which for a large N could be prohibitively expensive. In that case a parallel data structure that allows for constant time addition, lookup and deletion would be needed, like a SkipList.

Problem 4:

I am assuming that I do not need to provide any code to actually do the check, simply the approach to finding out which one is the fastest.

It is a little like measuring the thickness of a sheet of paper by measuring how thick a stack of 100 is, and then dividing that number by 100. Even if this overestimates the thickness (by measuring the interstitial air for example), it is a good functional number, because the thickness of paper is likely to matter when it is in reams anyway.

Similarly when comparing algorithms, I would gather a set of say, 100 numbers: 50 palindromes and 50 non-palindromes.

Then to test each algorithm, every number would be **repeatedly** fed to the algorithm, a fixed number of times, to then determine how long it takes to decide. For example, if the number is 28382, it would be fed to the algorithm a 100 times, to determine how much time, on average the algorithm takes to determine that 28382 is a palindrome.

The above is repeated for all 50 palindromes and non-palindromes, for each algorithm in consideration.

We can then get a single, average 'decision-time' figure for each algorithm.

However more interestingly, this method can show more subtle differences between algorithms, like:

A 'worse' algorithm being actually better at weeding out non-palindromes

	Palindrome decision time (ms)	Non palindrome decision time (ms)	Average decision time (ms)
A	60	50	55
B	100	30	65

Here, even though B has worse overall performance, it is quicker at weeding out non-palindromes. So if the dataset is mostly non-palindromes, B might be a better choice.

It can show how some algorithms are better at certain 'kinds' of numbers

Depending upon the design, some algorithms might be better at numbers that have certain properties, for example numbers that are multiples of 10, like 1000000 and are not palindromes, or that have repeating characters like 666666, that are.

Again if the dataset has a lot of numbers that exhibit these properties, it might make sense to pick one of these 'highly-specialised' algorithms.

Problem 5:

Naming conventions:

I read somewhere that reading and understanding someone else's code is harder than writing new code.

Naming conventions, meaningless in themselves, offer the consistency that makes the job of reading code a lot easier. If you can clearly see that if something is named like a constant, then it probably is a constant and that means you can get to the hard part of understanding what the code does quicker without having to decipher another layer of language.

You wouldn't want to read, say Java code, where all the variable names were in Russian if you did not know Russian. Yes, it is the same language, 'public' is still 'public', a 'class' is still a 'class', and you would eventually understand the code, but it would take you longer.

We need naming conventions for the same reason we use indentation or place code on separate lines (we have to in Python but the point stands). The computer does not care, but they make life easier for everyone else.

Design Patterns:

In the past year, I can recall using the following OO design patterns:

Singleton + Factory pattern:

Very commonly used to manage resources that are limited, (either by hardware or by policy), especially connections to remote devices, databases etc. There usually is a singleton factory that creates device-access or data-access objects (often themselves singletons) which are in turn used by the 'client code' to do its work.

Prototype:

In UI work (with a Java-based web framework called Vaadin), having the library create a prototypical UI page, which can then be later customised by whoever is using it. The standard UI page contains basic 'Click Here' controls, a background thread to asynchronously fetch data and a grid to display the results. Developers normally control how the grid and buttons are shown and sometimes how data is being fetched.

Façade/Proxy:

For communicating with a remote device, allowing a 'representation' of that device to present an interface to that device.

Architects are not needed, they are an over-head (in favour):

Software development increasingly relies upon re-use of existing libraries, heavily-opinionated frameworks and 'X as a service' offerings.

This means that for a vast majority of software projects, once the developers settle on one of these 'families' (of language+frameworks+DB+platform), highly complex, scalable and available applications can be built without much architectural input at all. One might argue that all software projects need customisation, but that may not necessarily need special 'architecture' experience and could be achievable by consensus among developers themselves.

However, architectural input may be needed in areas that require a high-degree of domain expertise (like medical technology or aviation).

Mohammed Hayat

mohd.zaryat@gmail.com

0742828 1276