# Pigeons in the Holes and Bugs in the Code

This month’s column was meant to be an additional exploration of the A\* algorithm but, as the old saying goes, the best laid plans of mice and men… and so on. So that exploration is going to have to wait until a later column. But deadlines being deadlines, and commitments being commitments, I thought I would offer a tongue-in-cheek discussion about two serious subjects.

The first subject is the logically based, mathematically provable theorem called the [pigeonhole principle](https://en.wikipedia.org/wiki/Pigeonhole_principle). On its surface, the pigeonhole principle says an entirely obvious fact that if one has $$n$$ objects into $$m$$ holes, where $$n>m$$, then you must, necessarily, have at least one hole with more than one object in it.

<>

When first encountering this statement, everyone’s initial reaction is “Of course! That’s obvious!” but the full implication of the statement often takes some time to get used to and some people never grasp the them.

One of the more interesting blind-spots concerns file compression. An amazingly large number of people, spread across the entire societal cross-section, believe that any file can be losslessly compressed. They may not assert that categorically but they are often surprised when they find out that a lossless compression algorithm can actually make the ‘compressed’ size bigger (for example consider [this post](https://superuser.com/questions/464315/why-is-a-7zipped-file-larger-than-the-raw-file/464381)).

By definition, lossless compression is a process that replaces the bit pattern in a file with another bit pattern such in such a way that the original can be recovered from the new pattern by inverting the process. When a losslessly compressed file is smaller than the original, it means that blocks of the information in the file is redundant. In simplistic terms, there are patterns of bits that are sufficient large that they can be replaced with a smaller pattern of bits.

Ignoring for the moment the restriction that a file must be represented by bits (equivalently, ignoring exactly how to implement the following) a lossless compression could take the bit field ‘100000010111111’ and replace it with ‘16x0106x1’, where the string ‘6x0’ means ‘000000’. This process is completely invertible and, for a field of even modest size, can make the resulting file smaller.

<>

Unfortunately, the pigeonhole principle guarantees that the process can’t be repeated indefinitely. If the bit pattern is sufficiently random, the tokenization process can actually result in a bit pattern larger than the original file. This is typically encountered when trying to compress an already compressed file.

The mathematical proof of this is not something worth presenting. Rather, there is a slick reductio ad absurdum argument that is easier to understand, a lot more satisfying, and leaves a punch line in its wake.

Imagine the same file as before and further imagine that lossless compression results in a file smaller by some factor (for convenience a factor of 4 was used in the following figure) after each application. The first compressed file then becomes the source for the second, the second for the third, and so on. Eventually, the original file should be able to be characterized by a single very interesting bit. Since we reject that idea, we must conclude that at some stage the compression fails to compress.

<>

So, it is clear that lossless compression can’t make every file smaller. Now call that file where the lossless compression fails to compress the base file (or the devil’s file, if you like to be dramatic). Since that bit pattern clearly contains all the information we want (either it is the original or it is one the files that has been losslessly compressed) in a pattern that can’t be made smaller, then the only file that can be as small as it is one with the exact same bit pattern; any other file must be bigger. Since the lossless compression produces a new bit pattern, it must necessarily be bigger.

Now onto the next principle. There is no official name for it nor is it a principle on par with the pigeonhole principle. It is not based on logic or mathematics or computer science but on common sense and experience with human nature. For the sake of argument in this blog, it will be called the no-bug principle.

Quite simply stated, the no-bug principle asserts that no piece of code of sufficient size/complexity is truly free of bugs no matter how many people or how much money and time a group spends on eliminating them.

Like its more rigorously proven cousin, the no-bug principle strikes disbelief into the hearts of people from all walks of life. The usual manifestation of this disbelief is the eye-rolling and snark-filled rants that come from otherwise rational people when a glitch is encountered in a favorite app or program. The duration of the rant and its intensity are inversely proportional to the amount of coding experience the ranter has. Typical utterances include things like “I can’t believe they didn’t test this before they shipped” or “I could do better than this $%@^$” and similar statements vomited out under duress.

Ironically, these ranters are also typically the people who sue for pardon in their own work and defend themselves with statements like “well, if I had more time” or “nobody’s perfect”.

Interestingly, the two principles can be combined in comic fashion if we reject the first and accept the second. Recognizing that any executable is simply a pattern of bits, we can then comfort ourselves in believing that every executable can be reduced to a single very interesting bit that is filled to the brim with bugs.