Not always the first things you think about when writing a program (or architecting a system). Not always of the utmost importance, but often can become important -- so be prepared. (At the very least, be prepared to explain why you made certain choices -- and make sure you are aware of their limitations, etc). Efficiency will generally directly influence how scalable something is. However, sometimes you'll have to pick a slightly-less efficient method of doing something, in order to ensure it scales well.

There are often many, many ways to perform a certain task, but some ways may be easier to implement than others, or make more sense than others. There is also often a large tradeoff when trying to make things scalable or efficient, versus "getting them done".

Consider calculating a number in the Fibonacci series (1, 1, 2, 3, 5, 8, 13, etc...). The two most apparent ways to do this are either via recursion, or iteration.

**Recursively:**

```c
int rec_fib(int n) {
    if (n<=0) return 0;  // Technically, <0 should be an error, but I'm being lazy ;)
    if (n<=2) return 1;
    return rec_fib(n-1) + rec_fib(n-2);
}
```

**Iteratively:**

```c
int iter_fib(int n) {
    int a = 1;
    int b = 1;
    int c;
    // Edge cases (no need to calculate 0, 1, 1)
    if (n <=0) return 0;
    if (n <=2) return 1;
    // Loop until (n-2) since we start off with a min. value of n=3
    for (int i=0; i<(n-2); ++i) {
        c = a+b;
        b = a;
        a = c;
    }
    return a; // (or c)
}
```

The recursive method is clearly shorter (code-wise), but is it "better"? How efficient is it in terms of big-O notation? Or memory? What's being wasted? What could you do to speed it up?

For the given iterative case, can you make it more efficient at all?

Going back to what was mentioned earlier: sometimes you have to pick a slightly less efficient method (or architecture) in order to ensure scalability. Case in point: early FB days. It was super-efficient to have the database living on the same machine as the web server, but this not only didn't scale, it made things rather non-fault-tolerant. The process of separating these two layers exposed a terrible inefficiency in the way database calls were being done, which caused several rounds of optimizations to happen. Separating the layers also allowed easy horizontal scaling of the web tier.
Bottleneck: Once scaled, where was the bottleneck? Could it be avoided or its effects lessened? Could it still be expanded/scaled?

When designing a new system, always at least think about how it will scale. Ask yourself at least these questions: Does the given dataset grow linearly with users? Exponentially? Logarithmically? Where will the bottlenecks be? CPU? RAM? Disk I/O? Network?

Sometimes, trying to make things REALLY scalable (or extensible) can cause severe efficiency issues. Case in point, at eBay (back in 2003-2004) trying to make an extensible object store database (using a relational database) added so much overhead that it became almost unusable.

It often really is about finding a balance between being efficient (optimization), and designing things such that they scale well. (Or perhaps re-designing it)

Some examples:

* **Globalcenter** (1997-1998): Data collection/reporting for switches/routers (via SNMP). Expanded from a handful of routers to hundreds of routers and switches (and therefore thousands and tens of thousands of ports). Redesign on both front-end (distribution) and back-end (tried a DB, stayed with files on faster storage)

* **eBay** (~2003): Database monitoring tool. Went from 4-5 hosts to almost 100, polling more and more elements. Mainly redesigned polling mechanism (ssh -> custom) - also parallelized poller.

* **Facebook** (~2005+): Photo storage: re-architected multiple times in order to cope with scaling.

* **Facebook** (~2007): New feed-related feature presented, questions of scale weren't well-addressed, system was NOT horizontally scalable.

* **Blizzard** (~2010): Facebook Friend-Finder designers didn't anticipate people having as many friends as I did. Also, built-in Battle.net friends list had a hard-coded limit.