# Lab: Measuring efficiency

This lab focuses on empirically measuring a program's running time.

Learning objectives for this lab:

- Gain ability to measure the running time of programs
- Gain understanding of factors that affect running time
- Gain ability to characterize a program's time complexity

## Set-up

1. Open the `COMP2210\labs` directory on your Engineering H: drive. (If you didn't complete the previous lab where you created this directory structure, do so now.)
2. Download the zip file associated with this lab, store it in the `COMP2210\labs` directory, and unzip the file.
   - `lab05.zip`
3. Open jGRASP to the `lab05` directory.

## Measuring running time

Java provides two methods for measuring time: `System.nanoTime()` and `System.currentTimeMillis()`. Although both could be used for our purposes, `nanoTime()` is the best choice since it is expressly designed to measure *elapsed time*. The `currentTimeMillis()` method is designed to measure "wall-clock" time.

To measure how long method `foo` takes to run we could do the following:

```
long start = System.nanoTime();
foo();
long elapsedTime = System.nanoTime() - start;
```

The value in `elapsedTime` represents the number of nanoseconds that method `foo` required, according to the JVM's internal time source. For a more useful display value we could provide this time estimate in seconds.

```
double time = elapsedTime / 1_000_000_000d;
System.out.printf("%4.3f", time);
```

`TimingCode.java` illustrates timing multiple runs of method `foo` and using an average value as its running time in seconds. Interact with this code and make sure that you understand everything that it does and how you might apply it for your own purposes.

## Improving performance: avoid unnecessary work

The crux of making code more efficient is avoiding unnecessary work. Sometimes this can mean being more careful in how the code is written; for example, making sure a search algorithm terminates as soon as the result of the search can be known.

The two search methods below illustrate this idea.

```
// exits only after examining the entire list
private static <T> boolean searchA(List<T> list, T target){
   boolean found = false;
   for (T element : list) {
      if (element.equals(target)) {
         found = true;
      }
   }
   return found;
}

// exits as soon as it knows the status of the search
private static <T> boolean searchB(List<T> list, T target){
   for (T element : list) {
      if (element.equals(target)) {
         return true;
      }
   }
   return false;
}
```

On average, we would expect `searchB` to perform better than `searchA` since it exits the loop and returns `true` as soon as `target` is found. This is an example of efficiency improvements we should always make, mainly because the code is just ... *better.* That's a subjective judgment, but I think most of us would agree that `searchB` is more efficient *and* more appealing.

Note that the worst-case performance of both `searchA` and `searchB` is the same - both will examine every element of the list before terminating. But on "average-case" searches we should see a performance difference.

The `EarlyExit` class demonstrates how you can measure this performance difference by building large arrays and repeatedly timing average-case searches. Interact with this code and make sure that you understand everything that it does and how you might apply it for your own purposes.

## Characterizing time complexity

More important that measuring running time *per se* is being able to characterize an algorithm's *time complexity.* The time complexity of an algorithm dictates how scalable the algorithm is; that is, whether or not the algorithm's running time will be practical as the problem size increases.

A concrete way to think about scalability is to ask the question "What happens to running time if the problem becomes twice as large as it currently is?" Is our algorithm still useful if the problem size we expect suddenly doubles? What if it doubles again? Characterizing the algorithm's time complexity is the first step in understanding the algorithm's scalability.

One approach to characterizing time complexity is to empirically answer the doubling questions above by timing the implemented algorithm on increasing problem sizes. By making observations of how the running time is affected by doubling the size of the input, we can predict what the algorithm's time complexity is.

The running time of many algorithms is proportional to some function. Specifically, the running time satisfies the relationship $T(N) \propto cf(N)$, where $c$ is a constant and $f(N)$ is a function called the *order of growth* of the running time. For typical algorithms that we will see, this function is usually one of a small set of functions: $logN$, $N$, $NlogN$, $N^2$, $N^3$, or in general $N^k$ for positive $k$.

Let's restrict ourselves to talking about algorithms with time complexity proportional to a *polynomial.* Thus, $T(N) \propto cN^k$ for positive $k$. Timing a such a program as its input increases by a factor of two allows us to generate data like the following.

| $N$ | $T(N)$ | $R$ |
|---|---|---|
| $N_0$ | $T(N_0)$ | – |
| $N_1$ | $T(N_1)$ | $T(N_1)/T(N_0)$ |
| $N_2$ | $T(N_2)$ | $T(N_2)/T(N_1)$ |
| ... | ... | ... |

| $N$ | $T(N)$ | $R$ |
|---|---|---|
| $N_i$ | $T(N_i)$ | $T(N_i)/T(N_{i-1})$ |
| $\ldots$ | $\ldots$ | $\ldots$ |
| $N_m$ | $T(N_m)$ | $T(N_m)/T(N_{m-1})$ |

Since $T(N) \propto cN^k$, then $T(2N_i)/T(N_i) \propto (2N_i)^k/N_i^k = 2^k$. So by observing the value that the ratio of $T(2Ni)/T(N_i)$ converges toward, we can set this value equal to $2^k$ and simply solve for $k$, which is the power (order) of the polynomial that describes the time complexity of our algorithm.

The `TimeComplexity` class demonstrates how you can generate data that will allow you to characterize polynomial time complexity. Interact with this code and make sure that you understand everything that it does and how you might apply it for your own purposes.