Hands-On: Efficiency and Empirical Analysis

This activity focuses on empirically measuring a program's running time. Before attempting this activity, you should complete the Efficiency note set and the *Empirical Analysis* section of the Algorithm Analysis note set, as well as the associated videos available in the instructional resources

After completing this activity you should

- Be able to measure the running time of programs.
- Be able to describe factors that affect running time.
- Be able to to characterize a program's time complexity.

You will need the following source code files to complete this activity.

- EarlyExit.java
- TimingCode.java
- TimeComplexity.java

Measuring running time

Java provides the following two methods for measuring time: `System.nanoTime()` and `System.currentTimeMillis()` . We could use either for our purposes in this lab, but we will use `nanoTime()` since it is expressly designed to measure *elapsed time*. Note that the `currentTimeMillis()` method is designed to measure "wall-clock" time. For more information on these methods, see the following links.

- `System.nanoTime()`
- `System.currentTimeMillis()`

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. (Of course `nanoTime` requires some time itself and the subtraction takes time, so `elapsedTime` isn't *strictly* the running time of `foo` .) 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.

1. Open `TimingCode.java` and compile it.
2. Run the program and observe the output.
3. 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 is known.

The two search methods below illustrate this idea.

```
/** A linear search that exits only after scanning 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;
}

/** A linear search that exits as soon as the result of the search is known. */
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.

1. Open `EarlyExit.java` and compile it.
2. Run this program and observe its output.
3. 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 than 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. (Strictly speaking, the notion of time makes sense for an *implemented* algorithm - a program - but not for an *algorithm*. We could perhaps speak in terms of the "work" or "computational steps" required by an algorithm, but "time" only makes sense when referring to a running program. Even so, the terms *time complexity* and *time* are used in the context of algorithms and we are expected to implicitly understand the distinction.)

The time complexity of many algorithms can described by a polynomial function. That is, the time complexity function *T(N)* is proportional to some polynomial function *f(N) = N^k*. The function *f* is called the *order of growth* of the algorithm's time complexity since the amount of work required by the algorithm as the problem size increases grows in proportion to the function *f*. For example, *f(N) = N^2* would characterize a *quadratic* growth rate.

If we were to implement an algorithm with polynomial time complexity in a Java method, we could empirically discover its *order of growth* by recording the running time of the method as we systematically increase the problem size (*N*) by a factor of two, as shown in the following table.

N	T(N)	Ratio
N	T(N)	
2N	T(2N)	T(2N) / T(N)
4N	T(4N)	T(4N) / T(2N)
8N	T(8N)	T(8N) / T(4N)
...	...	...
2^k x N	T(2^k x N)	T(2^k x N) / T(2^(k-1) x N)

Since we know that the time complexity *T(N)* is proportional to a polynomial function and since we're doubling *N* on each step, the values in the **Ratio** column will approach a constant value approximately equal to *2^k*. Solving for *k* would then give us the degree of the polynomial that characterizes this algorithm's time complexity. For example, the following table suggests that the underlying algorithm being timed has time complexity proportional to *N^2*.

N	T(N) (sec.)	Ratio
250	0.061	
500	0.042	1.35
1000	0.112	2.67
2000	0.340	3.04
4000	1.298	3.82
8000	5.334	4.11
16000	21.880	4.10
32000	85.763	3.92
64000	345.634	4.03

Of course not all algorithms have polynomial time complexity. Common orders of growth that we will see in this course include *log N*, *N*, *N log N*, *N^2*, *N^3*, *2^N*, and *N!*. For the purposes of this lab, however, we will restrict ourselves to talking about algorithms with time complexity proportional to a *polynomial*. (*f(N) = N^k* for *k* > 0)

The `TimeComplexity` class demonstrates how you can generate data that will allow you to characterize polynomial time complexity.

1. Open `TimeComplexity.java` and compile it.
2. Run this program and observe its output.

3. Interact with this code and make sure that you understand everything that it does and how you might apply it for your own purposes.

Submission

The submission page for this activity asks you to apply your understanding of empirical analysis of time complexity to a problem and then submit it for a grade.