CSE 101
Algorithms and Abstract Data Types
Introduction to Algorithm Runtime and Efficiency

How shall we measure the runtime of an algorithm? A standard approach is to identify some *basic operation* within the algorithm, then count the number of times that operation is performed on particular inputs. Runtime is thus measured not in seconds, but in our intger count of the number of executions of that basic operation. Care must therefore be taken in choosing which operation to count, so as not to obtain a runtime that is artificially low. We thus seek an operation that will be executed as many, or more, times than any other operation in the algorithm.

This is not as difficult as it may sound. For iterative algorithms, we choose an operation within the body of the innermost loop.

Example
A simple algorithm that performs some generic operations.

```
op1
for i=1 to 10
   op2
   for j=1 to 10
      op3
      for k=1 to 10
         op4
```

Examination of this pseudo-code reveals the number of executions of each operation to be those given in the following table.

operation	# executions
`op1`	1
`op2`	10
`op3`	100
`op4`	1000

Assuming that the costs of ops 1-4 are (at least approximately) equal, the better choice of basic operation is `op4`, since doing something 1000 times is worse than doing some (roughly equivalent) thing 1, or 10, or 100 times. Two obvious questions arise though.

(1) What if ops 1-4 are not of equal cost

and

(2) Why not count all operations?

To address (1), we would need to look inside ops 1-4. Consider the case for instance when ops 1-4 are themselves function calls. We would examine the code for each function and attempt to analyze its runtime in terms of some more basic operation(s). Ultimately, we must deconstruct our pseudo-code to the point where each basic operation is of approximately equal constant cost, and measure runtime in those constant units. Let us assume we've already done this, so that ops 1-4 may well be the same operation.

This brings up question (2). The total number of executions of all operations (now assumed to have equivalent cost) in the above example is $1000 + 100 + 10 + 1 = 1{,}111$. We could justifiably take this as the cost, but very few iterative algorithms (that do anything interesting) iterate a fixed number of times, as above. A more realistic situation would be an algorithm taking an integer n as input, and having the number of iterations depend on that input.

Example
Another simple algorithm performing a single operation `op` inside nested loops, whose number of iterations depend on n.

```
op
for i=1 to n
   op
   for j=1 to n
      op
      for k=1 to n
         op
```

The number of executions of our basic operation is now a function $T(n) = n^3 + n^2 + n + 1$ of the input value n. The question is whether we should ignore the lower order terms in this function, and instead consider the runtime to be the simpler function n^3. The answer follows when we realize that the point of our analysis is not to determine the runtime for a fixed value of n, but for *all* values of n, and especially for arbitrarily large such values. In other words, we wish to determine how the runtime grows with the parameter n. But both functions, the accurate $T(n)$ and the estimate n^3, grow without bound as n becomes arbitrarily large. To compare them, we consider their quotient $T(n)/n^3$, and analyze its behavior in the limit as n approaches ∞. We see that

$$\lim_{n\to\infty} \frac{T(n)}{n^3} = \lim_{n\to\infty} \frac{n^3 + n^2 + n + 1}{n^3} = \lim_{n\to\infty} \left(1 + \frac{1}{n} + \frac{1}{n^2} + \frac{1}{n^3}\right) = 1$$

The limit 1 indicates that the two functions $T(n)$ and n^3, although they both have limit ∞ as $n \to \infty$, approach ∞ *at the same rate*. We therefore deem the runtime, measured as the number of basic operations performed, to be n^3. Equivalently, we can simply refrain from counting those operations not within the body of the innermost loop, making the above algorithm equivalent, as far as runtime is concerned, to

```
for i=1 to n
   for j=1 to n
      for k=1 to n
         op
```

Suppose we analyze a second algorithm that performs 2 executions of `op` within its innermost loop.

```
for i=1 to n
   for j=1 to n
      for k=1 to n
         op
         op
```

For the sake of discussion, let us also suppose that these two algorithms (somehow) accomplish the same overall task. The runtime of the second algorithm is easily seen to be $2n^3$. It would appear then that the first algorithm is twice as efficient as the second, since it produces the same result at half the cost. However, the goal of algorithm analysis is to measure the efficiency of algorithms as a technology, indepent of the computing devices on which they are implemented and run. Since we can equalize the two algorithms by running the second on a machine that executes `op` in half the time, we should not deem the two runtimes, n^3 and $2n^3$, to be essentially different.

Our discussion has led us to the following informal procedure for analyzing iterative algorithms.

(1) Choose a basic operation that appears within the body of the innermost loop.
(2) Count the number of executions of this operation as a function of the input size n.
(3) Simplify the function found in (2) by dropping all lower order terms, and replacing the coefficient of the highest order term by 1.

The simplified function found in (3), or rather its rate of growth, is then deemed to be our algorithm runtime. If we wish to compare two algorithms, we must compare the growth rate of two functions.

Asymptotic Growth of Functions

Let $f(n)$ and $g(n)$ be functions. We define

(1) $f(n) = O(g(n))$ if and only if there exist $B > 0$ and $n_0 > 0$ such that $\frac{f(n)}{g(n)} \leq B$ for all $n \geq n_0$. We say $g(n)$ is an *asymptotic upper bound* for $f(n)$.

(2) $f(n) = \Omega(g(n))$ if and only if there exist $b > 0$ and $n_0 > 0$ such that $b \leq \frac{f(n)}{g(n)}$ for all $n \geq n_0$. We say $g(n)$ is an *asymptotic lower bound* for $f(n)$.

(3) $f(n) = \Theta(g(n))$ if and only if there exist $b > 0$, $B > 0$ and $n_0 > 0$ such that $b \leq \frac{f(n)}{g(n)} \leq B$ for all $n \geq n_0$. We say $g(n)$ is a *tight asymptotic bound* for $f(n)$.

(4) $f(n) = o(g(n))$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. We say $g(n)$ is a *strict asymptotic upper bound* for $f(n)$.

(5) $f(n) = \omega(g(n))$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$. We say $g(n)$ is a *strict asymptotic lower bound* for $f(n)$.