LARGE Asymptotic Notation

Asymptotic notation is notation that describes the approximate limiting behavior of functions, and includes "big O" and "little o" notation, Ω notation, and Θ notation,

The complexity of a computation is a measure of the resources used to make the computation. There is time complexity, a measure of the time used, and space complexity, a measure of how much memory is needed.

Since machines operate at different speeds, a measure of computation time in seconds, minutes, or hours, is dependent on the machine, and is thus not a useful measure of the time complexity of the algorithm used. Asymptotic notation is used to abstract from the technology of the machine. For example, Euclid's algorithm for finding the greatest common divisor of two integers, worked on a modern computer, takes less time than on an older computer, or by hand. But the true efficiency of the algorithm has not changed for 2300 years.

Complexity of Functions. For now, we consider only functions of non-negative integers to non-negative integers. If we write $\sqrt{13}$, for example, we mean an integer close to the real square root of 13, and if we write 13/2, we mean 6. When we write $\log_2 n$, we mean an integer which is close to the real value.

O Notation. If we write $f(n) = O(g(n))$, where f and g are functions, we mean that eventually $f(n)$ does not exceed some constant multiple of $g(n)$. More formally, there exist positive constants C and N such that $f(n) \leq C\, g(n)$ for all $n > N$.

The Right Side is as Simple as Possible. Typically, when we write $f(n) = O(g(n))$ (or Θ or Ω) the expression we use for g is simple, while the expression for f may be complicated. For example we write $3n^2 - 5n + 11 = O(n^2)$, but we would never write $n^2 = O(3n^2 - 5n + 11)$, even though it's true.

The right side is always written as simply as possible, and thus would never be $O(2n)$ or $O(n+3)$, because $O(n)$ is the same thing; and we would never write $O(\log_2 n)$, because the asymptotic class $O(\log n)$ does not depend on the choice of base.

We can ignore the values of $g(n)$ for the first few values of n, if necessary. For example, we can write $O(\log\log n)$ even though the function $\log\log n$ is undefined for $n < 2$.

Examples.

- $5n + 17 = O(n)$. Proof: Pick $C = 6$ and $N = 17$. Then $5n + 17 \leq 6n$ for all $n \geq 17$.
- $f(n) = n^2$. Then $f(n) \neq O(n)$. Proof: Assume $f(n) = O(n)$. Then there exist constants C, N such that $n^2 \leq Cn$ for all $n \geq N$. Let $n = \max\{N, C+1\}$. Then $n \geq N$, but $f(n) = n^2 \geq (C+1)\,n > C\,n$, contradiction.

Alternative Definition of O. $f(n) = O(g(n))$ if and only if there are constants C and K such that $f(n) \leq C\,g(n) + K$ for all n.

Using Limits. If $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} < \infty$ then $f(n) = O(g(n))$

Ω Notation. The inverse of O-notation. $f(n) = \Omega(g(n))$ means $g(n) = O(f(n))$. An alternative definition is that $f(n) = \Omega(g(n))$ if and only if there is some positive real number C and integer N such that $f(n) \geq C\,g(n)$ for all $n \geq N$.

Limits. If $\lim_{n\to\infty} \dfrac{f(n)}{g(n)} > 0$ then $f(n) = \Omega(g(n))$. The limit could be ∞.

Θ Notation. $f(n) = \Theta(g(n))$ means that both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Logarithms. Let's review some facts about logarithms that you learned in high school. For this section, we do not insist that everything is an integer, instead we use real numbers. For real numbers x and b such that $x > 0$ and $b > 1$, we define $\log_b x$ to be an exponent, the unique real number such that $b^{\log_b x} = x$.

1. $\log_b x > 0$ if $x > 1$.
2. $\log_b x < 0$ if $0 < x < 1$.
3. $\log_b 1 = 0$
4. $\log_b x$ is undefined if $x \leq 0$.
5. If $x < y$, then $\log_b x < \log_b y$.
6. $\log_b(xy) = \log_b x + \log_b y$.
7. $\log_b \left(\frac{x}{y}\right) = \log_b x - \log_b y$.
8. For any number n, positive, negative, or zero, $\log_b (x^n) = n \log_b x$
9. For any $c > 1$, $\log_c x = \frac{\log_b x}{\log_b c}$

Asymtotic Analysis of Code. We allow one time unit for each step. In the examples below, we assume that it takes one step to assign a value to a variable, one step to perform any arithmetic operation, one step to compare two values, and one step to write a line of output. Thus, in the first example, each iteration takes finitely many steps, hence $\Omega(1)$ time altogether.

In each example below, we assume that n is an integer whose value is given outside the code shown.

1.
   ```
   for(int i = 0; i < n; i++)
     cout << "i = " << i << endl;
   ```
   The time complexity of this code is $\Theta(n)$

2.
   ```
   for(int i = 1; i < n; i=i*2)
     cout << "i = " << i << endl;
   ```

The time complexity of this code is $\Theta(\log n)$

Proof: Assume $n > 1$. Let t be the number of iterations of the loop. During the last iteration of the loop, $i = 2^{t-1} < n$ Since the loop does not iterate again, we know that $i * 2 = 2^t \geq n$. Thus $2^{t-1} < n \leq 2^t$. Taking the base 2 logarithm for each of those three terms, we have $t - 1 < \log_2 n \leq t$. It follows that $t = \Theta(\log n)$. Since there are finitely many steps during each iteration, the time complexity of the code is $\Theta(\log n)$. ▮

3.
```
for(int i = n; i > 1; i=i/2)
 cout << "i = " << i << endl;
```
The time complexity is also $\Theta(\log n)$. Do you see why?

4.
```
for(int i = 1; i < n; i++)
 for(int j = 1; j < i; j = j*2)
  cout << "i = " << i << " j = " << j << endl;
```
The time complexity is $\Theta(n \log n)$.

Proof: The inner loop iterates $\Theta(\log i)$ times during iteration i of the outer loop. The number of iterations of the inner loop can be approximated by the definite integral $\int_{x=1}^{n} \ln x \, dx$. Using integration by parts, we get $n \ln n - n + 1 = \Theta(n \log n)$. ▮

5.
```
for(int i = 2; i < n; i=i*i)
 cout << "i = " << i << endl;
```
The time complexity is $\Theta(\log \log n)$.

Proof: By substitution. Let $j = \log_2 i$ and let $m = \log_2 n$. Taking the base 2 logarithm of both sides of the assignment $i = i * i$, we obtain the assignment $j = j * 2$. Finally, since $\log_2 2 = 1$, the code is equivalent to

```
for(int j = 1; i < m; j=j*2)
 cout << "i = " << i << endl;
```

By the result given in problem 2, the time complexity is $\Theta(\log m) = \Theta(\log \log n)$ ∎