Big-Oh Notation

Formal Definitions

A function $T(n)$ is in $O(f(n))$ ***(upper bound)***

iff there exist positive constants k and n_0

such that $|T(n)| \le k|f(n)|$ for all $n \ge n_0$.

A function $T(n)$ is in $\Omega(f(n))$ ***(lower bound)***

iff there exist positive constants k and n_0

such that $k|f(n)| \le |T(n)|$ for all $n \ge n_0$.

A function $T(n)$ is in $\Theta(f(n))$ ***(tight bound)***

iff it is in $O(f(n))$ and it is in $\Omega(f(n))$.

The definition for big-Oh was given on your lecture slides. the rest are presented here for your interest only. Not every function has a tight bound. For example, consider

$$f(n) = \begin{cases} n^3 & n \text{ even} \\ 1 & n \text{ odd} \end{cases}$$

In this case, we have $f(n) = O(n^3)$ and $f(n) = \Omega(1)$, but no $\Theta(\cdot)$ bound.

Useful Formulae

$$\sum_{i=1}^{n} i = \tfrac{1}{2}n(n+1) \qquad \sum_{i=1}^{n} i^2 = \tfrac{1}{6}n(n+1)(2n+1) \qquad \sum_{i=1}^{n} i^3 = \tfrac{1}{4}n^2(n+1)^2$$

You should know the first sum above. The rest will be given if you ever need them.

However, you should remember that $\sum_{i=1}^{n} i = O(n^2)$, $\sum_{i=1}^{n} i^2 = O(n^3)$, and $\sum_{i=1}^{n} i^3 = O(n^4)$.

Example 1

What is the big-O of $2n^2 + 1000n + 5$?

answer: $O(n^2)$

You can do this by inspection. To prove it formally, you must to find constants k and n_0 such that the definition given above holds:

$T(n) = 2n^2 + 1000n + 5$ $\qquad f(n) = n^2$

$T(n) \le k\, f(n)$?

$2n^2 + 1000n + 5 \le k\, n^2$?

Yes: for $k = 3$, $n_0 = 1001$

check: $2(1001^2) + 1000(1001) + 5 = 3005007$

$3(1001^2) = 3006003$

Example 2

Put these in order by big-O bound:

$4n^2 \quad \log_3 n \quad 20n \quad 2 \quad \log_2 n \quad n^n \quad 3^n \quad n\log n \quad 1000n^{2/3} \quad 2^n \quad 2^{n+1} \quad \log(n!)$

answer:

$2\,,\quad \log_2 n = \log_3 n\,,\quad 1000n^{2/3}\,,\quad 20n\,,\quad n\log n = \log(n!)\,,\quad 4n^2\,,\quad 2^n = 2^{n+1}\,,\quad 3^n\,,\quad n^n$

Some comments:

$\log_2 n = \log_3 n$: From highschool math, you should remember that $\log_b c = \dfrac{\log_a c}{\log_a b}$. Therefore, $\log_2 n = \dfrac{\log_3 n}{\log_3 2}$ which is a constant times $\log_3 n$. When looking at complexity classes, we ignore multiplicative constants.

$2^n = 2^{n+1}$: because $2^{n+1} = 2 \cdot 2^n$ which is a constant times 2^n.

$2^n \neq 3^n$: because they do *not* differ by a constant factor. Divide one by the other: $\dfrac{3^n}{2^n} = \left(\dfrac{3}{2}\right)^n$ which is a function of n – not a constant.

$n\log n = \log(n!)$: This is because of Stirling's approximation for the factorial: $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right)$. You can just remember the result that $\log(n!) = \Theta(n\log n)$.

Algorithm Analysis

Example 1

```
sum = 0;
for (i=0; i<3; i++)
     for (j=0; j<n; j++)
          sum++;
```

O(n) : outer loop is O(1), inner loop is O(n)

Example 2

```
sum = 0;
for (i=0; i<n*n; i++)
     sum++;
```

O(n^2) : loop is 1..n^2

Example 3

```
for (i=0; i<n; i++) {
     for (j=0; j<n; j++)
          A[i] = random(n);    // assume random() is O(1)
     sort(A, n);               // assume sort() is O(n log n)
}
```

O(n^2log n) : outer loop is O(n), inner loop is O(n), but sorting is O(n log n)
so, the complexity of the algorithm is n(n + n log n) = O(n^2log n)

Example 4

```
sum = 0;
for (i = 0; i < n; i++) {
     if (is_even(i)) {
          for (j = 0; j < n; j++)
          sum++;
     } else
          sum = sum + n;
}
```

O(n^2) : outer loop is O(n)
inside the loop: if "true" clause executed for half the values of n → O(n)
if "false" clause executed for other half → O(1)
the innermost loop is O(n)
so the complexity is n(n + 1) = O(n^2)

Example 5 (recursive)
```
List *SearchList(List *a, int key) { // The list has n elements
     if (a == NULL)
          return NULL; // not found
     else if (a->data == key)
          return a;
     else
          return SearchList(a->next, key);
}
```
O(n) : This is tail recursion, and it only calls itself once. Draw a picture of the recursive calls, and you will see that this is O(n).

Example 6 (recursive - from lecture slides)
```
int somefunc(int n) {
     if (n <= 1)
          return 1;
     else
          return somefunc(n-1) + somefunc(n-1);
}
```
$O(2^n)$: If you draw a picture of the recursive calls, you will get a full binary tree. The tree is of height n, with 2^i leaves at each level. The total number of recursive calls is the sum of the leaves at each level, which is $\sum_{i=1}^{n} 2^i = 2^{n+1} = O(2^n)$.

Example 7 (recursive - Fibonacci)
```
int Fibonacci(int n) {
     if (n <= 2)
          return 1;
     else
          return Fibonacci(n-1) + Fibonacci(n-2);
}
```
$O(2^n)$, $\Omega(2^{n/2} = \Omega(\left(\sqrt{2}\right)^n)$: A picture of the recursion tree is given in your textbook. If you draw the calls with the parameter (n-1) on the left, and (n-2) on the right, then the tree will be deepest on the left, with a height of n, and least deep on the right, with a height of n/2. Therefore, the size of the tree is greater than a full binary tree of height n/2, but less than a full binary tree of height n. This gives us both upper and lower bounds on the complexity of the function:

- left side is of height n → # leaves < 2^{n+1} → $O(2^n)$
- right side is of height n/2 → # leaves > $2^{(n+1)/2}$ → $\Omega(2^{n/2})$

Example 8 (recursive - Fibonacci)

A better way to write a function to calculate the Fibonacci series is to store the last two values. An O(n) iterative version is given in your text. Here is a recursive O(n) version:

```
int Fibonacci(int[] A, int i, int n) {
     if (i <= 2)
          A[i] = 1;
     else
          A[i] = A[i-1] + A[i-2];

     if (i == n)
          return A[i-1] + A[i-2];
     else
          return Fibonacci(A, i+1, n);
}

... = Fibonacci(A, 1, n);
```

O(n) : This is tail recursion again. Draw a picture of the recursion tree, and you'll see there are O(n) recursive calls.

(This version stores all the Fibonacci numbers in an array. If you only wanted the n^{th} Fibonacci number, then you only need to store the last two numbers in the series. You could easily re-write this function so that instead of the A array, it had two parameters for the previous and 2^{nd}-previous numbers.)