The big-Oh notation in terms of limits

Given two functions $f(n)$ and $g(n)$ from natural numbers to natural numbers, we say $f(n)$ is $O(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

Note that in practice, all you need is to show

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$
Examples of the big-Oh notation

- Every constant in $O(1)$
- $n^k$ is $O(2^n)$ for any natural number $k$.
- $\log(n)$ is $O(n^\epsilon)$ for any $\epsilon > 0$.
- $O(f_1(n)) + O(f_2(n))$ is $O(\max(f_1(n), f_2(n)))$.
- $O(f_1(n)) \cdot O(f_2(n))$ is $O(f_1(n) \cdot f_2(n))$ if 1 is both $O(f_1(n))$ and $O(f_2(n))$.

In practice, if we write $O(f(n))$ then $f(n)$ is almost always an increasing function (when $n$ is large enough).
Examples of algorithm analysis

```c
int sumarr1 (int A[], int start, int finish) {
    if (start < finish) {
        return A[start] + sumarr1 (A, start+1, finish) ;
    } // end of [if]
    return 0 ;
} // end of [sumarr1]
```

Assume that $T(n)$ is the time-complexity of `sumarr1`, where $n$ is the difference between `start` and `finish`. Then we have the following recurrence equation for $T(n)$:

$$T(n) = T(n-1) + O(1) \quad \text{if } n > 0$$

So we know that $T(n)$ is $O(n)$. 

Lecturer: Hongwei Xi
int sumarr2 (int A[], int start, int finish) {
    if (start < finish) {
        int m = start + (finish - start) / 2;
        return A[m] +
        sumarr2 (A, start, m-1) + sumarr2 (A, m+1, finish);
    } // end of [if]
    return 0;
} // end of [sumarr2]

Assume that $T(n)$ is the time-complexity of $sumarr2$, where $n$ is the difference between $start$ and $finish$. Then we have the following recurrence equation for $T(n)$:

$$T(n) = 2 \cdot T(n/2) + O(1) \text{ if } n > 0$$

So we know that $T(n)$ is $O(n)$. 
public class Node {
    int item;
    Node next;
    Node (int x) { item = x; }
    Node (int x, Node xs) { item = x; next = xs; }
} // end of [Node]
public static int list_length (Node xs) {
    int res = 0 ;
    while (xs != null) {
        xs = xs.next ; res = res + 1 ;
    }
    return res ;
} // end of [list_length]
public static Node list_copy (Node xs) {
    Node ys0 = null;
    if (xs == null) return ys0;
    ys0 = new Node (xs.item); xs = xs.next;
    Node ys = ys0;
    // copying xs into ys.next
    while (xs != null) {
        ys.next = new Node (xs.item);
        xs = xs.next; ys = ys.next;
    } // end of [while]
    ys.next = null;
    return ys0;
} // end of [list_copy]
public static
Node list_append (Node xs0, Node ys) {
    if (xs0 == null) return ys ;
    Node xs = xs0; Node xs_next = xs.next ;
    while (xs_next != null) {
        xs = xs_next ; xs_next = xs_next.next ;
    }
    xs.next = ys ;
    return xs0 ;
} // end of [list_append]
public static
Node list_reverse (Node xs) {
    Node ys = null ;
    while (xs != null) {
        Node xs_next = xs.next ;
        xs.next = ys ; ys = xs ; xs = xs_next ;
    }
    return ys ;
} // end of [list_reverse]
public static
Node list_insert (Node xs0, Node x0) {
    if (xs0 == null || x0.item <= xs0.item) {
        x0.next = xs0 ; return x0 ;
    }
    Node xs = xs0 ;
    Node xs_next = xs.next ;
    while (true) {
        if (xs_next == null || x0.item <= xs_next.item) {
            xs.next = x0 ; x0.next = xs_next ; break ;
        }
        xs = xs_next ; xs_next = xs_next.next ;
    } // end of [while]
    return xs0 ;
} // end of [list_insert]
End of the slides for lecture 4

End