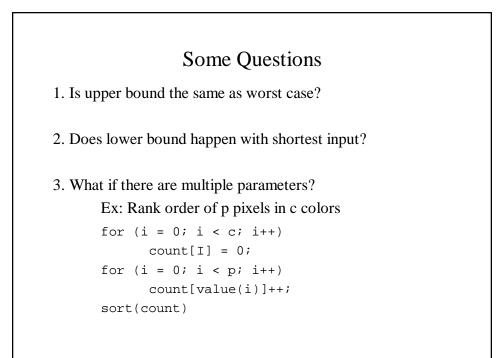


```
Example
Code:
    sum1 = 0;
    for (k=1; k<=n; k*=2)
        for (j=1; j<=n; j++)
            sum1++;
Complexity:
Code:
    sum2 = 0;
    for (k=1; k<=n; k*=2)
        for (j=1; j<=k; j++)
            sum2++;
Complexity:</pre>
```



# Space Complexity

Does it matter?

What determines space complexity?

How can you reduce it?

What tradeoffs are involved?

### Constants in Bounds

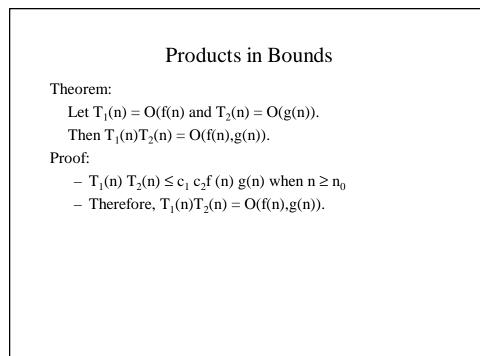
Theorem:

O(cf(x) = O(f(x)))

Proof:

- T(x) = O(cf(x)) implies that there are constants  $c_0$  and  $n_0$  such that  $T(x) \le c_0(cf(x))$  when  $x \ge n_0$
- Therefore,  $T(x) \le c_1(f(x))$  when  $x \ge n_0$  where  $c_1 = c_0 c$
- Therefore, T(x) = O(f(x))

# $\begin{array}{l} \textbf{Sum in Bounds} \\ \textbf{Theorem:} \\ \texttt{Let } T_1(n) = O(f(n) \text{ and } T_2(n) = O(g(n)). \\ \texttt{Then } T_1(n) + T_2(n) = O(\max(f(n),g(n)). \\ \textbf{Proof:} \\ & \quad - \text{ From the definition of O, } T_1(n) \leq c_1 f(n) \text{ for } n \geq n_1 \text{ and } \\ T_2(n) \leq c_2 g(n) \text{ for } n \geq n_2 \\ & \quad - \text{ Let } n_0 = \max(n_1, n_2). \\ & \quad - \text{ Then, for } n \geq n_0, T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n) \\ & \quad - \text{ Let } c_3 = \max(c_1, c_2). \\ & \quad - \text{ Then, } T_1(n) + T_2(n) \leq c_3 f(n) + c_3 g(n) \\ & \quad \leq 2c_3 \max(f(n), g(n)) \\ & \quad \leq c \max(f(n), g(n)) \end{array}$



# Polynomials in Bounds

Theorem:

If T(n) is a polynomial of degree x, then  $T(n) = O(n^x)$ .

Proof:

- $T(n) = n^{x+}n^{x-1} + \ldots + k$  is a polynomial of degree x.
- By the sum rule, the largest term dominates.
- Therefore,  $T(n) = O(n^x)$ .

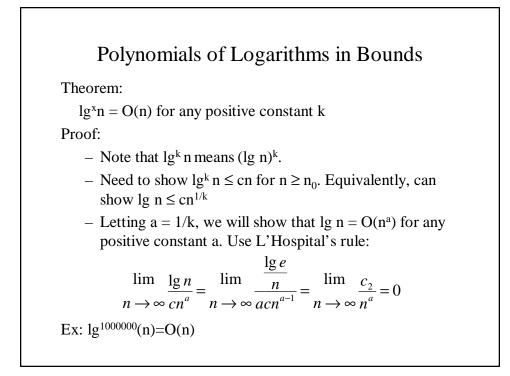
# L'Hospital's Rule

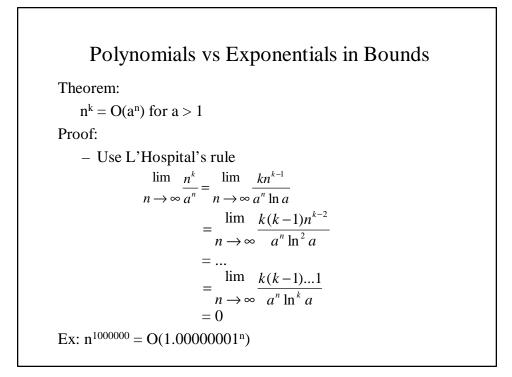
Finding limit of ratio of functions as variable approaches  $\infty$  $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$ 

Use to determine O or  $\Omega$  ordering of two functions

$$f(x = O(g(x)) \text{ if } \lim_{x \to \infty} \frac{f(x)}{g(x)} = 0$$

$$f(x) = \Omega(g(x))$$
 if  $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$ 



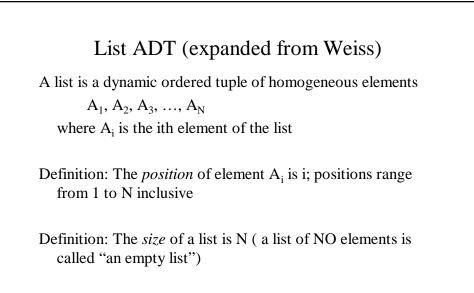


# Relative Orders of Growth

n (linear) log<sup>k</sup>n for k < 1 constant  $n^{1+k}$  for k > 0 (polynomial)  $2^n$  (exponential) n log n log<sup>k</sup>n for k > 1 n<sup>k</sup> for k < 1 log n

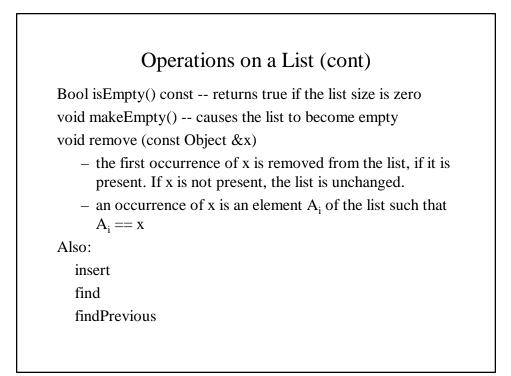
# Relative Orders of Growth

$$\label{eq:stant} \begin{split} & \text{constant} \\ & \text{log}\,^k n \; \text{for} \; k > 1 \\ & \text{log} \; n \\ & n^k \; \text{for} \; k < 1 \\ & n \; (\text{linear}) \\ & n \; \log \; n \\ & n^{1+k} \; \text{for} \; k > 0 \; (\text{polynomial}) \\ & 2^n \; (\text{exponential}) \end{split}$$



### Operations on a List

List() -- construct an empty list
List(const List &rhs) -- construct a list as a copy of rhs
~List() -- destroy the list
const List &operator=(const List &rhs)
make this list contain copies of the elements of rhs in the same order
elements are deep copied from rhs, not used directly. If L<sub>1</sub> = (A<sub>1</sub>, A<sub>2</sub>, A<sub>3</sub>) and L<sub>2</sub> = (B<sub>1</sub>, B<sub>2</sub>) before the assignment, then L<sub>1</sub> = L<sub>2</sub> causes L<sub>2</sub> = (A<sub>1</sub>, A<sub>2</sub>, A<sub>3</sub>)



### Iterators

An *iterator* is an object that provides access to the elements of a collection (in a specified order) without exposing the underlying structure of the collection.

- order dictated by the iterator
- collection provides iterators on demand
- each iterator on a collection is independent
- iterator operations are generic

# **Iterator Operations**

Bool isPastEnd() -- returns true if the iterator is past the end of the list

void advance() -- advances the iterator to the next position in the list. If iterator already past the end, no change.

const Object &retrieve() -- returns the element in the list at the current position of the iterator. It is an error to invoke "retrieve" on an iterator that isPastEnd

# List Operations

- ListIter<Object> first() -- returns an iterator representing the first element on the list
- List Iter<Object> zeroth() -- returns an iterator representing the header of a list
- ListIter<Object> find(const Object &x) -- returns an iterator representing the first occurrence of x in the list. If x not present, the iterator isPastEnd.
- ListIter<Object> findPrevious(const Object &x) -- returns an iterator representing the element before x in the list. If x is not in the list, the iterator represents the last element in the list. If x is first element (or list is empty), the iterator returned is equal to the one returned by zeroth().

# List Operators (cont)

void insert (const Object &x, const listIter<Object> &p)

- inserts a copy of x in the list after the element referred to by p
- if p isPastEnd, the insertion fails without an indication of failure.

# Ex: Building a List

```
List<int> list; // empty list of int
ListIter<int> iter = list.zeroth();
for (int i=0; i < 5; i++) {
    list.insert(iter);
    iter.advance();
  }</pre>
```

# Ex: Building a List #2

```
List<int> list; // empty list of int
ListIter<int> iter = list.zeroth();
for (int i=0; i < 5; i++) {
   list.insert(iter);
  }</pre>
```

Ex: