today’s outline

➔ formal definition of $O$, $\Omega$
➔ proving algorithm complexity
➔ problem solving session
formal definitions of $\Theta$ and $\Omega$
recap $O(n^2)$

set of functions that \textbf{grow no faster} than $n^2$

$\rightarrow$ count the number of steps
$\rightarrow$ constant factors don’t matter
$\rightarrow$ only highest-order term matter

These functions are in $O(n^2)$

$$n^2 \quad 2n^2 + 3n \quad \frac{n^2}{165} + 1130n + 3.14159$$
the formal definition of $O(n^2)$

a function $f(n)$ is in $O(n^2)$ iff

$$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \text{ such that } \forall n \in \mathbb{N}, n \geq B \implies f(n) \leq cn^2$$

Beyond **breakpoint** $B$, $f(n)$ is upper-bounded by $cn^2$, where $c$ is some wisely chosen constant multiplier.
the formal definition of $O(n^2)$

A function $f(n)$ is in $O(n^2)$ if

$$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \text{such that } \forall n \in \mathbb{N}, n \geq B \implies f(n) \leq cn^2$$

**Simple example:** prove $700n^2 \in O(n^2)$
the formal definition of $\Omega(n^2)$

a function $f(n)$ is in $O(n^2)$ iff

$$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \text{ such that } \forall n \in \mathbb{N}, n \geq B \Rightarrow f(n) \leq cn^2$$

a function $f(n)$ is in $\Omega(n^2)$ iff

$$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \text{ such that } \forall n \in \mathbb{N}, n \geq B \Rightarrow f(n) \geq cn^2$$

$O(n^2)$: set of functions that **grow no faster** than $n^2$

$\Omega(n^2)$: set of functions that **grow no slower** than $n^2$

$\Theta(n^2)$: set of functions that are in **both** $O(n^2)$ and $\Omega(n^2)$

(functions growing **as fast as** $n^2$)
growth rate ranking of typical functions

\[
\begin{align*}
    f(n) &= n^n \\
    f(n) &= 2^n \\
    f(n) &= n^3 \\
    f(n) &= n^2 \\
    f(n) &= n \log n \\
    f(n) &= n \\
    f(n) &= \sqrt{n} \\
    f(n) &= \log n \\
    f(n) &= 1
\end{align*}
\]
analyse a sorting algorithm
insertion sort

- grow a sorted list inside an unsorted list
- in each iteration
  - remove an element from the unsorted part
  - insert it into the position it belongs to in the sorted part

see animation at: http://en.wikipedia.org/wiki/Insertion_sort
insertion sort

```python
def IS(A):
    '''sort the elements in A in non-decreasing order'''
    n: size of A

    1. i = 1
    2. while i < len(A):
    3.     t = A[i]  # take red square out
    4.     j = i
    5.     while j > 0 and A[j-1] > t:
    7.         j = j - 1
    8.     A[j] = t  # put red square in
    9.     i = i + 1  # next element to be red-squared
```
insert sort worst case runtime

\[ W_{IS}(n) = \]
Prove the worst case complexity of insertion sort is $O(n^2)$

$$W_{IS}(n) = \frac{3}{2}n^2 + \frac{9}{2}n - 4 \in O(n^2)$$

$$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq B \Rightarrow W_{IS}(n) \leq cn^2$$

**Proof:**
Prove $W_{IS}(n) = \frac{3}{2}n^2 + \frac{9}{2}n - 4 \in \Omega(n)$

$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq B \Rightarrow W_{IS}(n) \geq cn^2$
analyse another algorithm
maximum slice

→ input: \( \textbf{L} \), a list of numbers

→ output: the maximum sum over slices of \( \textbf{L} \)

\[ \textbf{L} = [-2, -3, 4, -1, 6, -3] \]

\[ \text{max} = 4 + (-1) + 6 = 9 \]
def max_sum(L):
    ''' maximum sum over slices of L'''

    max = 0
    i = 0
    while i < len(L):
        j = i + 1
        while j <= len(L):
            sum = 0
            k = i
            while k < j:
                sum = sum + L[k]
                k = k + 1
            if sum > max:
                max = sum
            j = j + 1
        i = i + 1
    return max
def max_sum(L):

    ''' maximum sum over slices of L'''

    max = 0
    i = 0
    while i < len(L):
        j = i + 1
        while j <= len(L):
            sum = 0
            k = i
            while k < j:
                sum = sum + L[k]
                k = k + 1
            if sum > max:
                max = sum
                j = j + 1
        i = i + 1

    return max
def max_sum(L):
    ''' maximum sum over slices of L''''

    max = 0
    i = 0
    while i < len(L):
        j = i + 1
        while j <= len(L):
            sum = 0
            k = i
            while k < j:
                sum = sum + L[k]
                k = k + 1
            if sum > max:
                max = sum
            j = j + 1
        i = i + 1

    return max
takeaway

➔ when finding **upper-bound**, it is OK to **over-estimate** the number of steps, as long as the over-estimated number is upper bounded.

➔ when finding **lower-bound**, it is OK to **under-estimate** the number of steps, as long as the under-estimated number is lower-bounded.
problem solving: penny piles