So far...

• We’ve been mainly looking at
  – the syntax of the Java language.
  – key ideas in object-oriented programming: objects, inheritance, polymorphism, and dynamic binding, access privileges.
  – mechanisms provided in Java for organization of abstractions: abstracted classes, interfaces, packages.
  – Head First Java Chapter 1 - 11
Now...

• We are going to continue our voyage with Java, but with a different focus.
• We are going to examine
  – a range of canonical data structures.
  – algorithms often associated with these data structures.
  – the focus is on the efficiency and cost of these algorithms and data structures.
Measuring Cost
What Does Cost Mean?

• Cost can mean
  – Development costs: How much engineering time? When delivered?
  – Costs of failure: How robust? How safe?
  – Operational cost (for programs, time to run, space requirements).

• Is this program fast enough? Depends on:
  – What purpose.
  – What input data.
Cost Measures (Time)

• Cost Measures (Time)
  – Wall-clock or execution time
  – You can do this at home:
    
    time java FindPrimes 1000
  – Advantages: easy to measure, meaning is obvious. Appropriate where time is critical (real-time systems, e.g.).
  – Disadvantages: applies only to specific data set, compiler, machine, etc.

• Number of times certain statements are executed:
  – Advantages: more general (not sensitive to speed of machine).
  – Disadvantages: still applies only to specific data sets.

• Symbolic execution times:
  – Formulas for execution times or statement counts in terms of input size.
  – Advantages: applies to all inputs, makes scaling clear.
  – Disadvantage: practical formula must be approximate, may tell very little about actual time.
Example

• An algorithm for processing a retail store's inventory takes:
  – $10,000 \text{ ms}$ to read the initial inventory from disk, and then
  – $10 \text{ ms}$ to process each transaction
  – Processing $n$ transactions takes $(10,000 + 10 \times n)$ ms.
  – $10 \times n$ is more important if $n$ is very large.
Asymptotic Cost

• The constant coefficients can change:
  – If we had a faster computer or,
  – Use a different language or compiler.

• The constant factors can get smaller as technology improves.

• We want to express the speed of an algorithm independently of a specific implementation on a specific machine.

• We examine the cost of the algorithms for large input sets i.e. the asymptotic cost.
Big-Oh

• Specify bounding from above:
  – Big-Oh says how slowly code might run as its input size grows.

• Let $n$ be the size of a program's input. Let $T(n)$ be a function that equals to the algorithm's running time, given an input of size $n$.

• Let $f(n)$ be another function. We say that $T(n) \in O(f(n))$ if and only if $T(n) \leq c \times f(n)$ whenever $n$ is big, and for some constant $c$. 
Example

- Consider the cost function: \( T(n) = 10,000 + 10 \times n \),

- Let's try out \( f(n) = n \). We can choose \( c \) as large as we want:

\[
T(n) = 10000 + 10 \times n
\]

- As these functions extend forever to the right (infinity), their lines will never cross again.
  - For any \( n \) bigger than 1000, \( T(n) \leq c \times f(n) \).
Definition

• $O(f(n))$ is the set of all functions $T(n)$ that satisfy:
  – There exist positive constants $c$ and $N$ such that, for all $n \geq N, T(n) \leq c \times f(n)$

• In the example above: $c = 20$, and $N = 1000$. 
• $T(n) \leq 2f(n)$ whenever, $n \geq 1$

• So: $T(n)$ is in $O(f(n))$

• Notice $T(n) > f(n)$ everywhere.
Examples

• If $T(n) = 1,000,000 \times n$, $T(n)$ is in $O(n)$.
  
  – Let $f(n) = n$, set $c = 1,000,000$, $N = 1$:
    $$T(n) = 1,000,000 \times n \leq c \times f(n)$$
  
  – Big-Oh notation doesn't care about (most) constant factors.
  
  – Generally leave constants out; it's unnecessary to write $O(2n)$.

• If $T(n) = n$, $T(n)$ is in $O(n^3)$.
  
  – Let $f(n) = n^3$, set $c = 1$, $N = 1$:
    $$T(n) = n \leq n^3 = f(n)$$
  
  – Big-Oh notation can be misleading. Just because an algorithm's running time is in $O(n^3)$ doesn't mean it's slow; it might also be in $O(n)$. 
More Examples

• If $T(n) = n^3 + n^2 + n$, then $T(n)$ is in $O(n^3)$.
  
  – Let $T(n) = n^3$, set $c = 3$, $N = 1$.
  
  
  
  
  
  
  
  $T(n) = n^3 + n^2 + n \leq 3 \times n^3 = c \times f(n)$

  – Big-Oh notation is usually used only to indicate the dominating term in the function. The other terms become insignificant when $n$ is really big.
<table>
<thead>
<tr>
<th>Function</th>
<th>Common Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>$O(\log^2 n)$</td>
<td>Log-squared</td>
</tr>
<tr>
<td>$O(\sqrt{n})$</td>
<td>Root-n</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>$n \log n$</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>Cubic</td>
</tr>
<tr>
<td>$O(n^4)$</td>
<td>Quartic</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>Exponential</td>
</tr>
<tr>
<td>$O(e^3)$</td>
<td>Bigger exponential</td>
</tr>
</tbody>
</table>
Example

```java
/** Find position of X in list Return -1 if not found */
class List {
    ...
    int find (Object X) {
        int c = 0;
        ListNode cur = head;
        for (; cur != null; cur = cur.next, c++) {
            if (cur.item.equals(X))
                return c;
        }
        return -1;
    }
}
```

• Choose representative operation: number of `.equals` tests.

• If $N$ is length of the list, then loop does at most $N$ tests: worst-case time is $N$ tests.

• Worst-case time is $O(N)$;
Caveats About Constants

• $n^2$ is in $O(n)$ \quad \text{WRONG!}
  
  – Justification: if we choose $c = n$, we get $n^2 \leq n^2$.
  
  – $c$ must be a \textbf{constant}; it cannot depend on $n$. 
Caveats About Constants

• $e^{3n}$ is in $O(e^n)$ because the constant factor 3 don't matter.  \textbf{WRONG!}

• $10^n$ is in $O(2^n)$ because the constant factor 10 don't matter.  \textbf{WRONG!}

• Big-Oh notation doesn't care about most constant factors. But...

• Constant factor in an exponent is not the same as a constant factor in front of a term.
  – $e^{3n}$ is bigger than $e^n$ by a factor of $e^{2n}$
  – $10^n$ is bigger than $2^n$ by a factor of $5^n$. 
Caveats About Constants

• Problem actual size does matter in practice.
• Example:
  – An algorithm runs in time $T(n) = n \log n$, and another algorithm runs in time $U(n) = 100 \times n$
  – Big-Oh notation suggests you should use $U(n)$, because $T(n)$ dominates $U(n)$ asymptotically.
  – In practice, $U(n)$ is only faster than $T(n)$ if your input size is greater than current estimates of the number of subatomic particles in the universe.
  – For $U(n) \leq T(n)$, $100 \leq \log n$, $2^{100} \leq n$
Omega

• Big-Oh is an upper bound, it says, "Your algorithm is at least this good."
• Omega gives us a lower bound, it says, "Your algorithm is at least this bad."
• Omega is the reverse of Big-Oh:
  – If $T(n)$ is in $O(f(n))$, $f(n)$ is in $\Omega(T(n))$. 
Examples

• $2n$ is in $\Omega(n)$ because $n$ is in $O(2n)$.
• $n^2$ is in $\Omega(n)$ because $n$ is in $O(n^2)$.
• $n^2$ is in $\Omega(3n^2 + n \log n)$ because $3n^2 + n \log n$ is in $O(n^2)$. 
Definition

- $\Omega(f(n))$ is the **set of all functions** $T(n)$ that satisfy:
  - There exist positive constants $d$ and $N$ such that, for all $n \geq N$, $T(n) \geq d \cdot f(n)$
• \( T(n) \geq 0.5f(n) \) whenever \( n \geq 1 \)

• So \( T(n) \) is in \( \Omega(f(n)) \)

• Notice \( T(x) < f(n) \) everywhere.
Theta

• If we have: \( T(n) \) is in \( O(f(n)) \) and is also in \( \Omega(g(n)) \) then \( T(n) \) is effectively sandwiched between \( c \times f(n) \) and \( d \times g(n) \).

• When \( f(n) = g(n) \) we say that \( T(n) \) is in \( \Theta(g(n)) \).

• But how can a function be sandwiched between \( f(n) \) and \( f(n) \)?
  – we choose different constants (\( c \) and \( d \)) for the upper bound and lower bound.
Let $c = 1$, $d = 0.5$, $c \cdot f(n) \geq T(n) \geq 0.5f(n)$ whenever $x \geq 1$

So $f'(x)$ is in $\Theta(g(n))$
Interpreting

• Choice of $O$, $\Omega$, or $\Theta$ is independent of whether we're talking about worst-case running time, best-case running time, average-case running time, memory use, or some other function.

• "Big-Oh" is NOT a synonym for "worst-case running time," and Omega is not a synonym for "best-case running time."
Analysis Example 1

Problem #1: Given a set of p points, find the pair closest to each other.
Algorithm #1: Calculate the distance between each pair; return the minimum.

double minDistance = point[0].distance(point[1]);

/* Visit a pair (i, j) of points. */
for (int i = 0; i < numPoints; i++) {
    /* We require that j > i so that each pair is visited only once. */
    for (int j = i + 1; j < numPoints; j++) {
        double thisDistance =
            point[i].distance(point[j]);
        if (thisDistance < minDistance) {
            minDistance = thisDistance;
        }
    }
}

There are p (p - 1) / 2 pairs, and each pair takes constant time to examine. Therefore, worst- and best-case running times are in \( \Theta(p^2) \).
Analysis Example 2

Problem #2: Smooshing an array called "ints" to remove consecutive duplicates

Algorithm #2:

```java
int i = 0, j = 0;
while (i < ints.length) {
    ints[j] = ints[i];
    do {
        i++;
    } while ((i < ints.length) && (ints[i] == ints[j]));
    j++;
}
```

The outer loop can iterate up to ints.length times, and so can the inner loop. But the index "i" advances on every iteration of the inner loop. It can't advance more than ints.length times before both loops end.

So the worst-case running time of this algorithm is Theta(p) time.
/** True iff X is a substring of S */
boolean occurs (String S, String X) {
    if (S.equals (X)) return true;
    if (S.length () <= X.length () return false;
    return
    occurs (S.substring (1), X) ||
    occurs (S.substring (0, S.length ()-1), X);
}