Dropping Glass Balls

- Tower with N Floors
- Given 2 glass balls
- Want to determine the *lowest* floor from which a ball can be dropped and will break
- How?

- What is the most efficient algorithm?
- How many drops will it take for such an algorithm (as a function of N)?

Glass balls revisited (more balls)

- Assume the number of floors is 100
- In the best case how many balls will I have to drop to determine the lowest floor where a ball will break?
  1. 1  
  2. 2  
  3. 10  
  4. 16  
  5. 17  
  6. 18  
  7. 20  
  8. 21  
  9. 51  
  10. 100

In the worst case, how many balls will I have to drop?

- 1  
- 2  
- 10  
- 16  
- 17  
- 18  
- 20  
- 21  
- 51  
- 100

If there are \( n \) floors, how many balls will you have to drop? (roughly)

What is big-Oh about? (preview)

- Intuition: avoid details when they don’t matter, and they don’t matter when input size \( N \) is big enough
  - For polynomials, use only leading term, ignore coefficients
  
\[
\begin{align*}
y &= 3x & y &= 6x-2 & y &= 15x + 44 \\
y &= x^2 & y &= x^2-6x+9 & y &= 3x^2+4x
\end{align*}
\]

- The first family is \( O(n) \), the second is \( O(n^2) \)
  - Intuition: family of curves, generally the same shape
  - More formally: \( O(f(n)) \) is an upper-bound, when \( n \) is large enough the expression \( cf(n) \) is larger
  - Intuition: linear function: double input, double time, quadratic function: double input, quadruple the time

More on O-notation, big-Oh

- Big-Oh hides/obscures some empirical analysis, but is good for general description of algorithm
  - Allows us to compare algorithms *in the limit*
    - 20N hours vs \( N^2 \) microseconds: which is better?
  - O-notation is an upper-bound, this means that \( N \) is \( O(n) \), but it is also \( O(n^2) \); we try to provide tight bounds. Formally:
    
    A function \( g(n) \) is \( O(f(n)) \) if there exist constants \( c \) and \( n \) such that \( g(n) < cf(n) \) for all \( n > n \)

\[
x = n \\
\]
Which graph is “best” performance?

Big-Oh calculations from code

- Search for element in an array:
  - What is complexity of code (using O-notation)?
  - What if array doubles, what happens to time?

```java
for(int k=0; k < a.length; k++) {
    if (a[k].equals(target)) return true;
}
return false;
```

- Complexity if we call N times on M-element vector?
  - What about best case? Average case? Worst case?

Some helpful mathematics

- $1 + 2 + 3 + 4 + ... + N$
  - $N(N+1)/2$, exactly = $N^2/2 + N/2$ which is $O(N^2)$ why?
- $N + N + N + ... + N$ (total of N times)
  - $N^2$ which is $O(N^2)$
- $N + N + N + ... + N + ... + N + ... + N$ (total of 3N times)
  - $3N^2$ which is $O(N^2)$
- $1 + 2 + 4 + ... + 2^N$
  - $2^{N+1} - 1 = 2 \times 2^N - 1$ which is $O(2^N)$

- Impact of last statement on adding $2^N+1$ elements to a vector
  - $1 + 2 + ... + 2^N + 2^{N+1} = 2^{N+2} - 1 = 4 \times 2^N - 1$ which is $O(2^N)$

Running times @ $10^6$ instructions/sec

<table>
<thead>
<tr>
<th>N</th>
<th>$O(\log N)$</th>
<th>$O(N)$</th>
<th>$O(N \log N)$</th>
<th>$O(N^2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000003</td>
<td>0.00001</td>
<td>0.000033</td>
<td>0.0001</td>
</tr>
<tr>
<td>100</td>
<td>0.000007</td>
<td>0.00010</td>
<td>0.000664</td>
<td>0.1000</td>
</tr>
<tr>
<td>1,000</td>
<td>0.00010</td>
<td>0.01000</td>
<td>0.10000</td>
<td>1.0</td>
</tr>
<tr>
<td>10,000</td>
<td>0.000013</td>
<td>0.01000</td>
<td>0.132900</td>
<td>1.7 min</td>
</tr>
<tr>
<td>100,000</td>
<td>0.000017</td>
<td>0.10000</td>
<td>1.661000</td>
<td>2.78 hr</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.000020</td>
<td>1.0</td>
<td>19.9</td>
<td>11.6 day</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>0.000030</td>
<td>16.7 min</td>
<td>18.3 hr</td>
<td>318 centuries</td>
</tr>
</tbody>
</table>