Analyzing Algorithms

- **Remember SortByFreqs APT Problem:**
  - Start with array of words (Strings)
  - Find frequency of each word
  - Return array of words ordered from most frequent to least
  - (In case of a tie, return in alphabetical order)

```java
public class SortByFreqs {
    public String[] sort(String[] data) {
        // fill in code here
    }
}
```

- There are several approaches to a solution
  - Are they all equivalent?
Analyzing Algorithms

- Consider three solutions to `SortByFreqs`, also code used in Anagram discussion
  - Sort, then scan looking for changes
  - Insert into Set, then count each unique string
  - Find unique elements without sorting, sort these, then count each unique string

- We want to discuss trade-offs of these solutions
  - Ease to develop, debug, verify
  - Runtime efficiency
  - Vocabulary for discussion
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

\[
\begin{align*}
  t &= 3n \\
  t &= 6n-2 \\
  t &= 15n + 44 \\
  t &= n^2 \\
  t &= n^2-6n+9 \\
  t &= 3n^2+4n
\end{align*}
\]

- 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 \( c*f(n) \) 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
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 vectors?**
  - What about best case? Average case? Worst case?
Big-Oh calculations again

- Alcohol APT: first string to occur 3 times
  - What is complexity of code (using O-notation)?

```java
for (int k=0; k < a.length; k++) {
    int count = 0;
    for (int j=0; j <= k; k++) {
        if (a[j].equals(a[k])) count++;
    }
    if (count >= 3) return a[k];
}
return ""; // nothing occurs three times
```

- What happens to time if array doubles in size?
- $1 + 2 + 3 + \ldots + n-1$, why and what’s O-notation?
Amortization: Expanding ArrayLists

- Expand capacity of list when `add()` called
- Calling `add N` times, doubling capacity as needed

<table>
<thead>
<tr>
<th>Item #</th>
<th>Resizing cost</th>
<th>Cumulative cost</th>
<th>Resizing Cost per item</th>
<th>Capacity After add</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3-4</td>
<td>4</td>
<td>6</td>
<td>1.5</td>
<td>4</td>
</tr>
<tr>
<td>5-8</td>
<td>8</td>
<td>14</td>
<td>1.75</td>
<td>8</td>
</tr>
</tbody>
</table>

\[2^{m+1} - 2^{m+1}\] \[2^{m+1}\] \[2^{m+2} - 2\] \[\text{around 2}\] \[2^{m+1}\]

- What if we grow size by one each time?
Some helpful mathematics

- $1 + 2 + 3 + 4 + \ldots + N$
  - $N(N+1)/2$, exactly $= N^2/2 + N/2$ which is $O(N^2)$ why?

- $N + N + N + \ldots + N$ (total of $N$ times)
  - $N \times N = N^2$ which is $O(N^2)$

- $N + N + N + \ldots + N + \ldots + N$ (total of $3N$ times)
  - $3N \times N = 3N^2$ which is $O(N^2)$

- $1 + 2 + 4 + \ldots + 2^N$
  - $2^{N+1} - 1 = 2 \times 2^N - 1$ which is $O(2^N)$

- Impact of last statement on adding $2^N+1$ elements to a vector
  - $1 + 2 + \ldots + 2^N + 2^N+1 = 2^{N+2} - 1 = 4 \times 2^N - 1$ 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>0.000010</td>
<td>0.00100</td>
<td>0.010000</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>