CSE 143

Lecture 4: testing and complexity

reading: 13.1-13.2

http://www.alexsweet.co.uk/comics.php?comic=2
Tips for testing

- You cannot test every possible input, parameter value, etc.
  - Think of a limited set of tests likely to expose bugs.

- Think about boundary cases
  - Positive; zero; negative numbers
  - Right at the edge of an array or collection's size

- Think about empty cases and error cases
  - 0, -1, null; an empty list or array

- test behavior in combination
  - Maybe `add` usually works, but fails after you call `remove`
  - Make multiple calls; maybe `size` fails the second time only
Interfaces

- **interface**: A list of methods that a class can promise to implement.
  - Inheritance gives you an is-a relationship *and* code sharing.
    - A Lawyer can be treated as an Employee and inherits its code.
  - Interfaces give you an is-a relationship *without* code sharing.
    - A Rectangle object can be treated as a Shape but inherits no code.
  - Always declare variables using the **interface** type.

```java
List<String> list = new ArrayList<String>();
```
Runtime Efficiency (13.2)

- **efficiency**: measure of computing resources used by code.
  - can be relative to speed (time), memory (space), etc.
  - most commonly refers to run time

Assume the following:
- Any single Java statement takes same amount of time to run.
- A method call's runtime is measured by the total of the statements inside the method's body.
- A loop's runtime, if the loop repeats N times, is N times the runtime of the statements in its body.
Efficiency examples

statement1;
statement2;
statement3;

\[
\begin{align*}
&\text{for (int } i = 1; i \leq N; i++) \{ \\
&\quad \text{statement4;}
\}
\end{align*}
\]

\[
\begin{align*}
&\text{for (int } i = 1; i \leq N; i++) \{ \\
&\quad \text{statement5;}
&\quad \text{statement6;}
&\quad \text{statement7;}
\}
\end{align*}
\]

\[
\begin{align*}
&\{3 \} \\
&\{N \} \\
&\{3N \} \\
&\{4N + 3\}
\end{align*}
\]
Efficiency examples 2

```c
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        statement1;
    }
}
```

```c
for (int i = 1; i <= N; i++) {
    statement2;
    statement3;
    statement4;
    statement5;
}
```

- How many statements will execute if N = 10? If N = 1000?
Algorithm growth rates (13.2)

- We measure runtime in proportion to the input data size, N.
  - **growth rate**: Change in runtime as N changes.

- Say an algorithm runs $0.4N^3 + 25N^2 + 8N + 17$ statements.
  - Consider the runtime when N is *extremely large*.
  - We ignore constants like 25 because they are tiny next to N.
  - The highest-order term ($N^3$) dominates the overall runtime.

- We say that this algorithm runs "on the order of" $N^3$.
- or $O(N^3)$ for short ("Big-Oh of N cubed")
**Complexity classes**

- **complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size $N$.

<table>
<thead>
<tr>
<th>Class</th>
<th>Big-Oh</th>
<th>If you double $N$, ...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$O(1)$</td>
<td>unchanged</td>
<td>10ms</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log_2 N)$</td>
<td>increases slightly</td>
<td>175ms</td>
</tr>
<tr>
<td>linear</td>
<td>$O(N)$</td>
<td>doubles</td>
<td>3.2 sec</td>
</tr>
<tr>
<td>log-linear</td>
<td>$O(N \log_2 N)$</td>
<td>slightly more than doubles</td>
<td>6 sec</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(N^2)$</td>
<td>quadruples</td>
<td>1 min 42 sec</td>
</tr>
<tr>
<td>cubic</td>
<td>$O(N^3)$</td>
<td>multiplies by 8</td>
<td>55 min</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^N)$</td>
<td>multiplies drastically</td>
<td>$5 \times 10^{61}$ years</td>
</tr>
</tbody>
</table>

...
Complexity classes

Collection efficiency

- Efficiency of our `ArrayIntList` or Java's `ArrayList`:

<table>
<thead>
<tr>
<th>Method</th>
<th>ArrayList</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>O(1)</td>
</tr>
<tr>
<td>add(index, value)</td>
<td>O(N)</td>
</tr>
<tr>
<td>get</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove</td>
<td>O(N)</td>
</tr>
<tr>
<td>set</td>
<td>O(1)</td>
</tr>
<tr>
<td>size</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Max subsequence sum

• Write a method $\text{maxSum}$ to find the largest sum of any contiguous subsequence in an array of integers.
  • Easy for all positives: include the whole array.
  • What if there are negatives?

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>1</td>
<td>-4</td>
<td>10</td>
<td>15</td>
<td>-2</td>
<td>22</td>
<td>-8</td>
<td>5</td>
</tr>
</tbody>
</table>

Largest sum: $10 + 15 + -2 + 22 = 45$

• (Let's define the max to be 0 if the array is entirely negative.)

• Ideas for algorithms?
Algorithm 1 pseudocode

\textbf{maxSum}(a):
\begin{align*}
\text{max} & \; = \; 0. \\
\text{for each starting index } & \; i: \\
\text{for each ending index } & \; j: \\
\text{sum} & \; = \; \text{add the elements from } a[i] \text{ to } a[j]. \\
\text{if } & \; \text{sum} \; > \; \text{max}, \\
\text{max} & \; = \; \text{sum}.
\end{align*}

\text{return } \text{max}.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>1</td>
<td>-4</td>
<td>10</td>
<td>15</td>
<td>-2</td>
<td>22</td>
<td>-8</td>
<td>5</td>
</tr>
</tbody>
</table>
Algorithm 1 code

- What complexity class is this algorithm?
  - $O(N^3)$. Takes a few seconds to process 2000 elements.

```java
public static int maxSum1(int[] a) {
    int max = 0;
    for (int i = 0; i < a.length; i++) {
        for (int j = i; j < a.length; j++) {
            // sum = add the elements from a[i] to a[j].
            int sum = 0;
            for (int k = i; k <= j; k++) {
                sum += a[k];
            }
            if (sum > max) {
                max = sum;
            }
        }
    }
    return max;
}
```
Flaws in algorithm 1

- Observation: We are redundantly re-computing sums.
  
  
  - We already had computed the sum of 2-5, but we compute it again as part of the 2-6 computation.

- Let's write an improved version that avoids this flaw.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>1</td>
<td>-4</td>
<td>10</td>
<td>15</td>
<td>-2</td>
<td>22</td>
<td>-8</td>
<td>5</td>
</tr>
</tbody>
</table>
Algorithm 2 code

- What complexity class is this algorithm?
  - $O(N^2)$. Can process tens of thousands of elements per second.

```java
public static int maxSum2(int[] a) {
    int max = 0;
    for (int i = 0; i < a.length; i++) {
        int sum = 0;
        for (int j = i; j < a.length; j++) {
            sum += a[j];
            if (sum > max) {
                max = sum;
            }
        }
    }
    return max;
}
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>1</td>
<td>-4</td>
<td>10</td>
<td>15</td>
<td>-2</td>
<td>22</td>
<td>-8</td>
<td>5</td>
</tr>
</tbody>
</table>
A clever solution

- **Claim 1**: A max range cannot start with a negative-sum range.

  
<table>
<thead>
<tr>
<th>i</th>
<th>...</th>
<th>j</th>
<th>j+1</th>
<th>...</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sum(j+1, k)</td>
</tr>
<tr>
<td>sum(i, k)</td>
<td>&lt; sum(j+1, k)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Claim 2**: If \( \text{sum}(i, j-1) \leq 0 \) and \( \text{sum}(i, j) < 0 \), any max range that ends at \( j+1 \) or higher cannot start at any of \( i \) through \( j \).

  
<table>
<thead>
<tr>
<th>i</th>
<th>...</th>
<th>j-1</th>
<th>j</th>
<th>j+1</th>
<th>...</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>\geq 0</td>
<td>&lt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sum(j+1, k)</td>
</tr>
<tr>
<td>&lt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sum(j+1, k)</td>
</tr>
<tr>
<td>sum(?, k)</td>
<td>&lt; sum(j+1, k)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Together, these observations lead to a very clever algorithm...
Algorithm 3 code

- What complexity class is this algorithm?
  - \( \mathcal{O}(N) \). Handles many millions of elements per second!

```java
public static int maxSum3(int[] a) {
    int max = 0;
    int sum = 0;
    int i = 0;
    for (int j = 0; j < a.length; j++) {
        if (sum < 0) { // if sum becomes negative, max range
            i = j;       // cannot start with any of i - j-1
            sum = 0;     // (Claim 2)
        }
        sum += a[j];
        if (sum > max) {
            max = sum;
        }
    }
    return max;
}
```
Runtime of first 2 versions

• Version 1:

<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>15</td>
</tr>
<tr>
<td>2000</td>
<td>47</td>
</tr>
<tr>
<td>4000</td>
<td>203</td>
</tr>
<tr>
<td>8000</td>
<td>781</td>
</tr>
<tr>
<td>16000</td>
<td>3110</td>
</tr>
<tr>
<td>32000</td>
<td>12563</td>
</tr>
<tr>
<td>64000</td>
<td>49937</td>
</tr>
</tbody>
</table>

• Version 2:

<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>16</td>
</tr>
<tr>
<td>2000</td>
<td>16</td>
</tr>
<tr>
<td>4000</td>
<td>110</td>
</tr>
<tr>
<td>8000</td>
<td>406</td>
</tr>
<tr>
<td>16000</td>
<td>1578</td>
</tr>
<tr>
<td>32000</td>
<td>6265</td>
</tr>
<tr>
<td>64000</td>
<td>25031</td>
</tr>
</tbody>
</table>
## Runtime of 3rd version

- Version 3:

<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
</tr>
<tr>
<td>8000</td>
<td>0</td>
</tr>
<tr>
<td>16000</td>
<td>0</td>
</tr>
<tr>
<td>32000</td>
<td>0</td>
</tr>
<tr>
<td>64000</td>
<td>0</td>
</tr>
<tr>
<td>128000</td>
<td>0</td>
</tr>
<tr>
<td>256000</td>
<td>0</td>
</tr>
<tr>
<td>512000</td>
<td>0</td>
</tr>
<tr>
<td>1e6</td>
<td>0</td>
</tr>
<tr>
<td>2e6</td>
<td>16</td>
</tr>
<tr>
<td>4e6</td>
<td>31</td>
</tr>
<tr>
<td>8e6</td>
<td>47</td>
</tr>
<tr>
<td>1.67e7</td>
<td>94</td>
</tr>
<tr>
<td>3.3e7</td>
<td>188</td>
</tr>
<tr>
<td>6.5e7</td>
<td>453</td>
</tr>
<tr>
<td>1.3e8</td>
<td>797</td>
</tr>
<tr>
<td>2.6e8</td>
<td>1578</td>
</tr>
</tbody>
</table>

**Input size (N)**