Evaluating an algorithm

• How to know whether a given algorithm is good, efficient, etc.?

• One idea: *Implement it, run it, time it / measure it (averaging trials)*
  - Pros?
    • Find out how the system effects performance
    • Stress testing – how does it perform in dynamic environment
    • No math!
  - Cons?
    • Need to implement code (takes time)
    • Can be hard to estimate performance
    • When comparing two algorithms, all other factors need to be held constant (e.g., same computer, OS, processor, load)
How efficient is this algorithm? Can it be improved?

```java
// returns the range of values in the given array;
// the difference between elements furthest apart
// example: range({17, 29, 11, 4, 20, 8}) is 25
public static int range(int[] numbers) {
    int maxDiff = 0; // look at each pair of values
    for (int i = 0; i < numbers.length; i++) {
        for (int j = 0; j < numbers.length; j++) {
            int diff = Math.abs(numbers[j] - numbers[i]);
            if (diff > maxDiff) {
                maxDiff = diff;
            }
        }
    }
    return maxDiff;
}
```
A slightly better version:

```java
// returns the range of values in the given array;
// the difference between elements furthest apart
// example: range({17, 29, 11, 4, 20, 8}) is 25
public static int range(int[] numbers) {
    int maxDiff = 0; // look at each pair of values
    for (int i = 0; i < numbers.length; i++) {
        for (int j = i + 1; j < numbers.length; j++) {
            int diff = Math.abs(numbers[j] - numbers[i]);
            if (diff > maxDiff) {
                maxDiff = diff;
            }
        }
    }
    return diff;
}
```
Range algorithm 3

A MUCH faster version. Why is it so much better?

```java
// returns the range of values in the given array;
// example: range({17, 29, 11, 4, 20, 8}) is 25
public static int range(int[] numbers) {
    int max = numbers[0];  // find max/min values
    int min = max;
    for (int i = 1; i < numbers.length; i++) {
        if (numbers[i] < min) {
            min = numbers[i];
        }
        if (numbers[i] > max) {
            max = numbers[i];
        }
    }
    return max - min;
}
```
Runtime of each version

- **Version 1:**
  - N | Runtime (ms) |
  - 1000 | 15 |
  - 2000 | 47 |
  - 4000 | 203 |
  - 8000 | 781 |
  - 16000 | 3110 |
  - 32000 | 12563 |
  - 64000 | 49937 |

- **Version 2:**
  - N | Runtime (ms) |
  - 1000 | 16 |
  - 2000 | 16 |
  - 4000 | 110 |
  - 8000 | 406 |
  - 16000 | 1578 |
  - 32000 | 6265 |
  - 64000 | 25031 |

- **Version 3:**
  - N | Runtime (ms) |
  - 1000 | 0 |
  - 2000 | 0 |
  - 4000 | 0 |
  - 8000 | 0 |
  - 16000 | 0 |
  - 32000 | 0 |
  - 64000 | 0 |
  - 128000 | 0 |
  - 256000 | 0 |
  - 512000 | 0 |
  - 1e6 | 0 |
  - 2e6 | 16 |
  - 4e6 | 31 |
  - 8e6 | 47 |
  - 1.67e7 | 94 |
  - 3.3e7 | 188 |
  - 6.5e7 | 453 |
  - 1.3e8 | 797 |
  - 2.6e8 | 1578 |
Max subsequence sum

• Write a method `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

maxSum(a):
    max = 0.
    for each starting index i:
        for each ending index j:
            sum = add the elements from a[i] to 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>
• How efficient is this algorithm?
  - Poor. It 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.
Algorithm 2 code

• How efficient is this algorithm?
  ▪ Mediocre. It can process 10,000s 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;
}
```
A clever solution

• Claim 1: The max range cannot start with a negative-sum range.

\begin{align*}
\text{i} & \quad \ldots \quad \text{j} \quad \text{j+1} \quad \ldots \quad \text{k} \\
< 0 & \quad \text{sum}(j+1, k) \\
\text{sum}(i, k) & < \text{sum}(j+1, k)
\end{align*}

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

\begin{align*}
\text{i} & \quad \ldots \quad \text{j-1} \quad \text{j} \quad \text{j+1} \quad \ldots \quad \text{k} \\
\geq 0 & \quad < 0 \quad \text{sum}(j+1, k) \\
< 0 & \quad \text{sum}(j+1, k) \\
\text{sum}(?, k) & < \text{sum}(j+1, k)
\end{align*}

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

• How efficient is this algorithm?
  ▪ Excellent. It can handle 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) so move i up to j
        }
        sum += a[j];
        if (sum > max) {
            max = sum;
        }
    }
    return max;
}
```
Analyzing efficiency

**efficiency**: A measure of the use of computing resources by code.
- most commonly refers to run time; but could be memory, etc.

Rather than writing and timing algorithms, let's *analyze* them. Code is hard to analyze, so let's make the following assumptions:
- Any *single Java statement* takes a constant amount of time to run.
- The runtime of a *sequence* of statements is the sum of their runtimes.
- An *if/else*'s runtime is the runtime of the if test, plus the runtime of whichever branch of code is chosen.
- A *loop*'s runtime, if the loop repeats *N* times, is *N* times the runtime of the statements in its body.
- A *method call*'s runtime is measured by the total of the statements inside the method's body.
Runtime example

statement1;
statement2;

\[ 2 \]

\[
\text{for (int } i = 1; i \leq N; i++) \{ \\
\quad \text{statement3;} \\
\quad \text{statement4;} \\
\quad \text{statement5;} \\
\quad \text{statement6;} \\
\}\]

\[
4N \\
\frac{1}{2} N^2 + 4N + 2
\]

\[
\text{for (int } i = 1; i \leq N; i++) \{ \\
\quad \text{for (int } j = 1; j \leq N/2; j++) \{ \\
\quad \quad \text{statement7;} \\
\quad \}\}
\]

\[
\frac{1}{2} N^2
\]

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

- 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*.
    (Almost any algorithm is fine if $N$ is small.)
  - 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")
Growth rate example

Consider these graphs of functions. Perhaps each one represents an algorithm:

\[ N^3 + 2N^2 \]
\[ 100N^2 + 1000 \]

• Which is better?
Growth rate example

• How about now, at large values of $N$?
**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>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^N)$</td>
<td>multiplies drastically</td>
<td>$5 * 10^{61}$ years</td>
</tr>
</tbody>
</table>
## Java collection efficiency

<table>
<thead>
<tr>
<th>Method</th>
<th>Array List</th>
<th>Linked List</th>
<th>Stack</th>
<th>Queue</th>
<th>TreeSet /Map</th>
<th>[Linked] HashSet /Map</th>
<th>Priority Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add or put</strong></td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)*</td>
<td>O(1)*</td>
<td>O(log N)</td>
<td>O(1)</td>
<td>O(log N)*</td>
</tr>
<tr>
<td><strong>add at index</strong></td>
<td>O(N)</td>
<td>O(N)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>contains/indexOf</strong></td>
<td>O(N)</td>
<td>O(N)</td>
<td>-</td>
<td>-</td>
<td>O(log N)</td>
<td>O(1)</td>
<td>-</td>
</tr>
<tr>
<td><strong>get/set</strong></td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(1)*</td>
<td>O(1)*</td>
<td>-</td>
<td>-</td>
<td>O(1)*</td>
</tr>
<tr>
<td><strong>remove</strong></td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(1)*</td>
<td>O(1)*</td>
<td>O(log N)</td>
<td>O(1)</td>
<td>O(log N)*</td>
</tr>
<tr>
<td><strong>size</strong></td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

• * = operation can only be applied to certain element(s) / places
Big-Oh defined

- Big-Oh is about finding an *asymptotic upper bound*.

- Formal definition of Big-Oh:
  \[ f(N) = O(g(N)), \text{ if there exists positive constants } c, N_0 \text{ such that } f(N) \leq c \cdot g(N) \text{ for all } N \geq N_0. \]
  - We are concerned with how \( f \) grows when \( N \) is large.
    - not concerned with small \( N \) or constant factors
  - Lingo: "\( f(N) \) grows no faster than \( g(N) \)."

![Graph showing Big-Oh notation](image-url)
Big-Oh questions

- \( N + 2 = O(N) \)?
  - yes
- \( 2N = O(N) \)?
  - yes
- \( N = O(N^2) \)?
  - yes
- \( N^2 = O(N) \)?
  - no
- \( 100 = O(N) \)?
  - yes
- \( N = O(1) \)?
  - no
- \( 214N + 34 = O(N^2) \)?
  - yes
Preferred Big-Oh usage

• Pick the tightest bound. If \( f(N) = 5N \), then:
  \[
  f(N) = O(N^5) \\
  f(N) = O(N^3) \\
  f(N) = O(N \log N) \\
  f(N) = O(N) \quad \leftarrow \text{preferred}
  \]

• Ignore constant factors and low order terms:
  \[
  f(N) = O(N), \quad \text{not} \quad f(N) = O(5N) \\
  f(N) = O(N^3), \quad \text{not} \quad f(N) = O(N^3 + N^2 + 15)
  \]

- Wrong: \( f(N) \leq O(g(N)) \)
- Wrong: \( f(N) \geq O(g(N)) \)
- Right: \( f(N) = O(g(N)) \)
A basic Big-Oh proof

- **Claim**: \(2N + 6 = O(N)\).

- **To prove**: Must find \(c, N_0\) such that for all \(N \geq N_0\),
  \[
  2N + 6 \leq c \cdot N
  \]

- **Proof**: Let \(c = 3, N_0 = 6\).
  \[
  2N + 6 \leq 3 \cdot N
  \]
  \[
  6 \leq N
  \]
Math background: Exponents

• Exponents:
  - \( X^Y \), or "X to the Y\(^{th}\) power";
  - \( X \) multiplied by itself \( Y \) times

• Some useful identities:
  - \( X^A \cdot X^B = X^{A+B} \)
  - \( X^A / X^B = X^{A-B} \)
  - \((X^A)^B = X^{AB}\)
  - \( X^N + X^N = 2X^N \)
  - \( 2^N + 2^N = 2^{N+1} \)
Math background: Logarithms

• Logarithms
  ▪ definition: \( X^A = B \) if and only if \( \log_X B = A \)
  ▪ intuition: \( \log_X B \) means:
    "the power \( X \) must be raised to, to get \( B \)"
  ▪ In this course, a logarithm with no base implies base 2.
    \( \log B \) means \( \log_2 B \)

• Examples
  ▪ \( \log_2 16 = 4 \) (because \( 2^4 = 16 \))
  ▪ \( \log_{10} 1000 = 3 \) (because \( 10^3 = 1000 \))
Logarithm bases

• Identities for logs with addition, multiplication, powers:
  - \( \log (A \cdot B) = \log A + \log B \)
  - \( \log (A/B) = \log A − \log B \)
  - \( \log (A^B) = B \log A \)

• Identity for converting bases of a logarithm:

\[
\log_A B = \frac{\log_C B}{\log_C A}
\]

- example:
  \( \log_4 32 = (\log_2 32) / (\log_2 4) \)
  = 5 / 2

- Practically speaking, this means all \( \log_c \) are a constant factor away from \( \log_2 \), so we can think of them as equivalent to \( \log_2 \) in Big-Oh analysis.
More runtime examples

• What is the exact runtime and complexity class (Big-Oh)?

```java
int sum = 0;
for (int i = 1; i <= N; i += c) {
  sum++;
}
```
- Runtime = $N / c = O(N)$.

```java
int sum = 0;
for (int i = 1; i <= N; i *= c) {
  sum++;
}
```
- Runtime = $\log_c N = O(\log N)$.
Binary search

- **binary search** successively eliminates half of the elements.

  - **Algorithm**: Examine the middle element of the array.
    - If it is too big, eliminate the right half of the array and repeat.
    - If it is too small, eliminate the left half of the array and repeat.
    - Else it is the value we're searching for, so stop.

  - Which indexes does the algorithm examine to find value 42?
  - What is the runtime complexity class of binary search?

<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>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>2</td>
<td>7</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>22</td>
<td>25</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>56</td>
<td>68</td>
<td>85</td>
<td>92</td>
<td>103</td>
</tr>
</tbody>
</table>

```
min
mid
max
```
Binary search runtime

• For an array of size $N$, it eliminates $\frac{1}{2}$ until 1 element remains.
  $N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \ldots, 4, 2, 1$
  ▪ How many divisions does it take?

• Think of it from the other direction:
  ▪ How many times do I have to multiply by 2 to reach $N$?
    $1, 2, 4, 8, \ldots, \frac{N}{4}, \frac{N}{2}, N$
  ▪ Call this number of multiplications "$x$".
    $2^x = N$
    $x = \log_2 N$

• Binary search is in the logarithmic ($O(\log N)$) complexity class.
**Math: Arithmetic series**

- Arithmetic series notation (*useful for analyzing runtime of loops*):
  \[ \sum_{i=j}^{k} \text{Expr} \]
  - the sum of all values of \( \text{Expr} \) with each value of \( i \) between \( j--k \)

- Example:
  \[ \sum_{i=0}^{4} 2i + 1 \]
  \[ = (2(0) + 1) + (2(1) + 1) + (2(2) + 1) + (2(3) + 1) + (2(4) + 1) \]
  \[ = 1 + 3 + 5 + 7 + 9 \]
  \[ = 25 \]
Arithmetic series identities

- sum from 1 through \( N \) inclusive:

\[
\sum_{i=1}^{N} i = \frac{N(N+1)}{2} = O(N^2)
\]

- Intuition:
  - sum = 1 + 2 + 3 + ... + (\( N-2 \)) + (\( N-1 \)) + \( N \)
  - sum = (1 + \( N \)) + (2 + \( N-1 \)) + (3 + \( N-2 \)) + ... // rearranged
  - // \( N/2 \) pairs total

- sum of squares:

\[
\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} = O(N^3)
\]
Series runtime examples

- What is the exact runtime and complexity class (Big-Oh)?

```java
int sum = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N * 2; j++) {
        sum++;
    }
}
```

- Runtime = $N \cdot 2N = O(N^2)$.

```java
int sum = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= i; j++) {
        sum++;
    }
}
```

- Runtime = $N(N + 1) / 2 = O(N^2)$. 