Series of Constants

- **Sum of constants**
  (when the body of the series doesn't contain the counter variable such as $i$)

$$\sum_{i=a}^{b} k = k \sum_{i=a}^{b} 1 = k(b-a+1)$$

- **Example:**

$$\sum_{i=4}^{10} 5 = 5 \sum_{i=4}^{10} 1 = 5(10-4+1) = 35$$
Splitting Series

For any constant $k$,

- splitting a sum with addition

\[ \sum_{i=a}^{b} (i + k) = \sum_{i=a}^{b} i + \sum_{i=a}^{b} k \]

- moving out a constant multiple

\[ \sum_{i=a}^{b} ki = k \sum_{i=a}^{b} i \]
Series of Powers

- **Sum of powers of 2**

\[ \sum_{i=0}^{N} 2^i = 2^{N+1} - 1 \]

\[ 1 + 2 + 4 + 8 + 16 + 32 = 64 - 1 = 63 \]

Think about binary representation of numbers:

\[ \begin{array}{cccccc}
1 & 1 & 1 & 1 & 1 & 1 \\
\end{array} \quad (63) \\
+ \phantom{1}1 \\
\hline \\
1 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \quad (64) \]

(and now a crash course on binary numbers…)
More Series Identities

- Sum from $a$ through $N$ inclusive (when the series doesn't start at 1)

$$\sum_{i=a}^{N} i = \sum_{i=1}^{N} i - \sum_{i=1}^{a-1} i$$

- Is there an intuition for this identity?

- Can apply same idea if you want the split series to start from 0

$$\sum_{i=a}^{N} 2^i = \sum_{i=0}^{N} 2^i - \sum_{i=0}^{a-1} 2^i$$
Series Practice Problems

- Give a closed form expression for the following summation.
  - A closed form expression is one without the $\Sigma$ or "…".

$$\sum_{i=0}^{N-2} 2i$$

- Give a closed form expression for the following summation.

$$\sum_{i=10}^{N-1} (i - 5)$$
Efficiency examples 6 (revisited)

```c
int sum = 0;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i / 2; j += 2) {
        sum++;
    }
}
```

- Compute the value of the variable sum after the following code fragment, as a closed-form expression in terms of input size n.
  - Ignore small errors caused by i not being evenly divisible by 2 and 4.
Growth Rate Terminology (recap)

- \( f(n) = O(g(N)) \)
  - \( g(n) \) is an **upper bound** on \( f(n) \)
  - \( f(n) \) **grows no faster** than \( g(n) \)

- \( f(n) = \Omega(g(N)) \)
  - \( g(N) \) is a **lower bound** on \( f(n) \)
  - \( f(n) \) grows at least as fast as \( g(N) \)

- \( f(n) = \Theta(g(N)) \)
  - \( f(n) \) grows at the same rate as \( g(N) \)
Facts About Big-Oh

- If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then
  - $T_1(N) + T_2(N) = O(f(N) + g(N))$
  - $T_1(N) \times T_2(N) = O(f(N) \times g(N))$

- If $T(N)$ is a polynomial of degree $k$, then:
  - $T(N) = \Theta(N^k)$
  - Example: $17n^3 + 2n^2 + 4n + 1 = \Theta(n^3)$

- $\log^k N = O(N)$, for any constant $k$ (for us, $k$ will generally be 1)
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>$5 \times 10^{61}$ years</td>
</tr>
</tbody>
</table>
Complexity cases

- **Worst-case**
  - “most challenging” input of size n

- **Best-case**
  - “easiest” input of size n

- **Average-case**
  - random inputs of size n

- **Amortized**
  - $m$ “most challenging” consecutive inputs of size n, divided by $m$
Bounds vs. Cases

Two orthogonal axes:

- **Bound**
  - Upper bound ($O$)
  - Lower bound ($\Omega$)
  - Asymptotically tight ($\Theta$)

- **Analysis Case**
  - Worst Case (Adversary), $T_{\text{worst}}(n)$
  - Average Case, $T_{\text{avg}}(n)$
  - Best Case, $T_{\text{best}}(n)$
  - Amortized, $T_{\text{amort}}(n)$

- One can estimate the bounds for any given case.
Example

`List.contains(Object o)`

- **returns** `true` if the list contains `o`; `false` otherwise

- **Input size:** `n` (the length of the `List`)

- **f(n) = “running time for size n”**

- **But f(n) needs clarification:**
  - **Worst case f(n):** it runs in at most `f(n)` time
  - **Best case f(n):** it takes at least `f(n)` time
  - **Average case f(n):** average time
A method in Java can call itself; if written that way, it is called a *recursive method*.

The code of a recursive method should be written to handle the problem in one of two ways:

- **base case**: a simple case of the problem that can be answered directly; does not use recursion.
- **recursive case**: a more complicated case of the problem, that isn't easy to answer directly, but can be expressed elegantly with recursion; makes a recursive call to help compute the overall answer.
Defining powers recursively:

\[
\begin{align*}
\text{pow}(x, 0) &= 1 \\
\text{pow}(x, y) &= x \times \text{pow}(x, y-1), \quad y > 0
\end{align*}
\]

// recursive implementation
public static int pow(int x, int y) {
    if (y == 0) {
        return 1;
    } else {
        return x * pow(x, y - 1);
    }
}