Lecture

Ways of writing out algorithms

- Pseudo code
  - Main choice for this course
- Natural language
- Flowcharts
  - Think mermaidjs ?

First n integers sum

$$\sum_{i=0}^{N} i = \frac{n(n-1)}{2}$$

Quadratic

```
int sum1(n) {
        int sum = 0
        for int i = 0 to n
                for int j = i to n
                        sum += i
}
```

$$T(n) = 1 + 2n(2n+2) + 1$$
$$= 4n^2 + 4n + 2$$

Input	Cost
T(1)	10
T(2)	26
T(5)	122

Linear

```
int sum2(n) {
        int sum = 0
        for int i = 0 to n
                sum += i
}
```

$$T(n) = 1 + 2n(2) + 1$$
$$= 4n + 2$$

Input	Cost
T(1)	6
T(2)	10
T(5)	22

Constant

```
int sum(n) {
        return n * (n + 1) / 2
}
```

$$T(n) = 4$$

Input	Cost
T(1)	4
T(2)	4
T(5)	4

Strategies for analyzing runtime

- Follow program execution
  - Track executions and analyze how it scales with input size
  - Cost of individual operations is ignored
    - Runtime based on input size is what matters
  - We are testing for input sizes approaching infinity

Big-Oh (O)

- Upper bounded function
- Represents the worst case of an algorithm
  - Maximum Time
- Highest order polynomial is what is taken
  - Constants dropped

$$T(n) = O(f(n)) \quad T(n) \leq f(n) \text{ when } n \geq n_0$$

- In our quadratic example:

$$T(n) = 4n^2 + 4n + 2 \implies O(n^2)$$

- Common patterns:

- Single loop from 1 to $n \rightarrow O(n)$
- Nested loops of same size $\rightarrow O(n)$
- Loop with iterator doubling / halving $\rightarrow O(\log n)$
  - `for i = 0; i < n; i *= 2`
- Loop with iterator growing by constant $\rightarrow O(n)$

Big-Omega (Ω)

- Lower bounded function
- Represents the best case of an algorithm
  - Minimum Time
- Also specifies the lower limit of the algorithm
- Lowest order polynomial is what is taken

$$T(n) = \Omega(f(n)) \quad T(n) \geq f(n) \text{ when } n \geq n_0$$

- In our quadratic example:

$$T(n) = 4n^2 + 4n + 2 \implies \Omega(n^2)$$

Big-Theta (Θ)

- Tight bound of a function

$$\begin{gathered} T(n) = \Theta(fn) \\ \text{if } T(n) = O(f(n)) \\ \text{and } T(n) = \Omega(f(n)) \end{gathered}$$

Efficiency Classes

Efficiency Class	Time Complexity
Constant	1
Logarithmic	$\log n$
Linear	n
Linear Logarithmic	$n \log n$
Quadratic	n^2
Cubic	n^3
Exponential	2^n or n^n
Factorial	$n!$

Relationships

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$$

Types of Algorithms

Recursive

- Count recursive steps

Iterative

- Count operations
- Loops

Both

- Can solve the same problems, and be implemented in each type

Pseudo code to sum form

```
A(n) {
        int i;
        for i = 0 to n
                print("welcome")
}
```

$$T(n) = \sum_{i=1}^{N} 1 \implies O(n)$$

```
B(n) {
        int i, j;
        for i = 0 to n
                for j = 0 to n
                        print("nested")
}
```

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} 1 \implies O(n^2)$$