Complexity

Faraaz Sareshwala

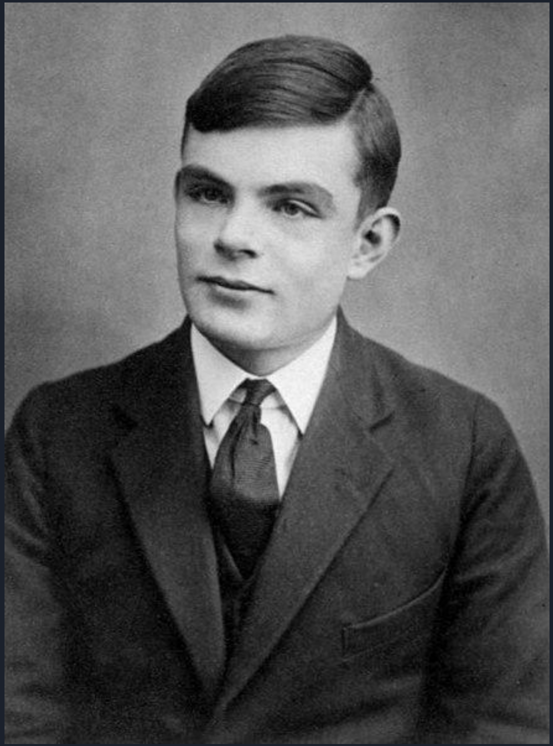
Why Study Algorithms?

Their impact is broad and far-reaching.

- **Internet:** web search, packet routing, distributed file sharing, etc
- **Biology:** human genome project, protein folding, etc
- **Computers:** circuit layout, file systems, compilers
- **Computer Graphics:** movies, video games, virtual reality, etc
- **Security:** cell phones, e-commerce, electronic voting machines, etc
- **Multimedia:** MP3, JPG, DivX, HDTV, facial recognition, etc
- **Social Networks:** recommendations, news feeds, advertisements, etc

Old Roots, New Opportunities

- Study of algorithms dates back to Euclid
- Formalized by Church and Turing in the 1930s
    - Check out The Imitation Game on Netflix
- Some important algorithms were discovered by undergraduates in a course like this!



Why Study Algorithms?

- Intellectual Stimulation
  - “An algorithm must be seen to be believed” -- Donald Knuth

- Become a More Proficient Programmer
  - “The difference between a bad programmer and a good one is whether they consider their code or their data structures more important. Bad programmers worry about the code. Good programmers worry about the data structures and their relationships.” -- Linus Torvalds

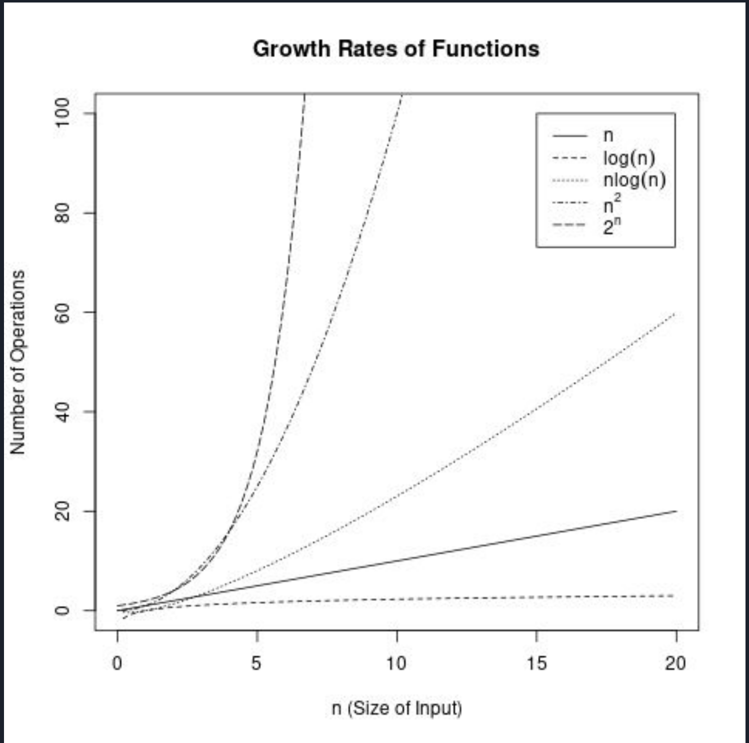
Data Structures and Algorithms

- **Data Structure:** A method to organize your data
- **Algorithm:** A method to solve a problem using a data structure

"Given an appropriate data structure, the algorithm will present itself" -- Donald Knuth

Growth of Functions

- Growth Rate
  - n
  - log n
  - n log n
  - n^2
  - 2^n



Complexity

- As the input size increases, how does the algorithm perform in terms of **time** taken to complete the task and the amount of memory (**space**) required to do so?
- **Time Complexity:** Roughly the growth rate of the number of CPU operations
- **Space Complexity:** The growth rate of the amount of memory required

- Why do we care?
  - Should I choose algorithm a or algorithm b?
  - What makes the algorithm faster?
  - Are some algorithms just always faster?
  - How can I make this algorithm faster?

Big-O Notation

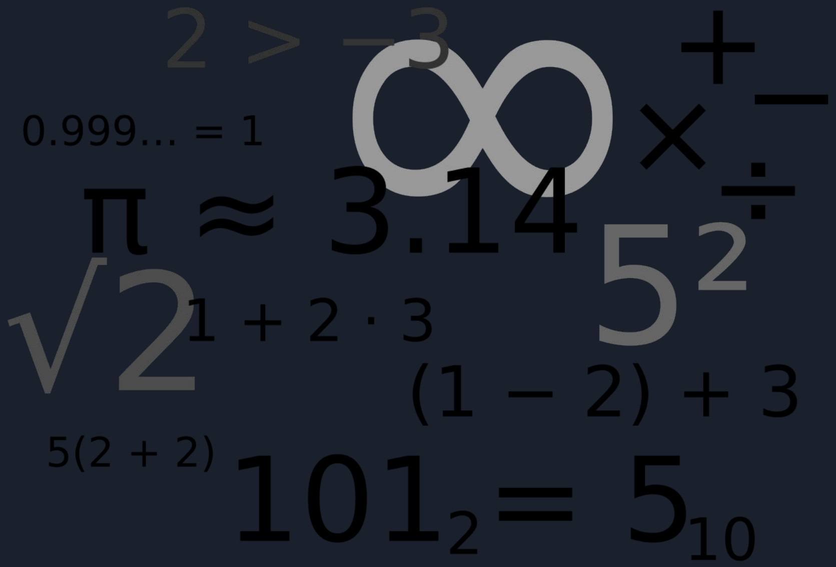
- Big-O notation allows us to standardize how we talk about algorithm complexity

Growth Rate	Time Complexity
n	O(n)
log n	O(log n)
n log n	O(n log n)
n^2	$O(n^2)$
2^n	$O(2^n)$

- Why just add an O(...) around the growth rate?
    - O(...) (Big Omicron): upper bound (worst case performance)
    - Θ(...) (Big Theta): absolute bound
    - Ω(...) (Big Omega): lower bound
- In this class, we only care about O(...) worst case time complexity

Calculating Time Complexity

- Mathematical methods
  - Master's Theorem
  - Recurrence relations
  - Recurrence tree
  - More
- Heuristics
  - Analyze directly from code



Calculation Heuristics: Consecutive Statements

```
log_n_function(); // O(log n)

n_function(); // O(n)

n_squared_function(); // O(n²)
```

- Maximum is all that counts
- Time complexity: $O(n^2)$
- Space complexity: ?

Calculation Heuristics: If/Else Statements

```
if (n_function()) {
    log_n_function();
}
else {
    n_squared_function();
}
```

- Total complexity is complexity of the test plus maximum of the two alternatives
- Time complexity:
    - $O(n + \max(\log(n), n^2))$
    - $O(n + n^2)$
    - $O(n^2)$
- Space complexity: ?

Calculation Heuristics: Loops

```
for (int i = 0; i < n; i++) {
    if (i % 2 == 0) {
        System.out.println(i);
    }
}
```

- Total complexity is complexity of statements inside the for loop times number of iterations
- Time complexity: O(n)
- Space complexity: O(1)

Calculation Heuristics: Nested Loops

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (array[i][j] < 100) {
            return false;
        }
    }
}
```

- Analyze these inside out. Total complexity is complexity of statement multiplied by product of the sizes of all loops.
- Time complexity: $O(n^2)$
- Space complexity: $O(1)$

Rules for Working with Big-O Notation

- Drop lower order polynomial terms
  - $O(n^2 + n + 1)$
  - $O(n^2)$
- Addition
  - If $T(n) = O(f(n))$ and $V(n) = O(g(n))$
  - $T(n) + V(n) = \max(O(f(n)), O(g(n)))$
  - In English: sequential statements, each with unique Big-O time complexities, have a combined Big-O time complexity of the most complex statement
- Drop constants
  - $O(3n^3)$
  - $O(n^3)$
- Multiplication
  - If $T(n) = O(f(n))$ and $V(n) = O(g(n))$
  - $T(n) \times V(n) = O(f(n) \times g(n))$
  - In English: dependent statements (e.g. nested for loops), each with unique Big-O time complexities, have a combined Big-O time complexity of each multiplied together

Intuitive Interpretations of Growth Rate Functions

- Constant: $O(1)$
  - Algorithm independent of input size
- Logarithmic: $O(\log n)$
  - Algorithm cuts size of problem by some fraction (usually ½)
- Linear: $O(n)$
  - Time increases directly with input size
- Linearithmic: $O(n \log n)$
  - Algorithm is logarithmic but also has a linear component
- Quadratic: $O(n^2)$
  - Algorithm has full input dependency per input element
- Exponential: $O(2^n)$
  - Combinatorial algorithm (e.g. NP problems)
  - Searching problems trying to find the optimal solution (e.g. Traveling Salesman)

Amortized Complexity

- Some operations have a high cost only sometimes
- Example: insertion into a C++ vector

```
void insert(int value) {
  if (!has_capacity()) {
    double_size();
  }
  insert_at_end(value);
}
```

- Cost to insert on average is O(1)
- Sometimes, we have to double the vector to create more capacity
- Time complexity: O(n)
- Amortized time complexity: O(1)
- Average case complexity over long period of time is still O(1)