Lecture 6

Complexity

- Lab 1 – Preview
- Time-Complexity
- Big-Oh
- Addendum: Arrays vs Linked Lists

Preview of Lab 1

Computer Science CS136 (Fall 2021)
Duane Bailey & Aaron Williams
Laboratory 1
The Silver Dollar Game

Objective. To implement a simple game using `Vectors` or arrays.

Discussion. The Silver Dollar Game is played between two players. An arbitrarily long strip of paper is marked off into squares:

THE "COIN STRIP" USED IN THE SILVER DOLLAR GAME.

The game begins by placing silver dollars in a few of the squares. Each square holds at most one coin. Interesting games begin with some pairs of coins separated by one or more empty squares.

A POSSIBLE STARTING POSITION FOR THE SILVER DOLLAR GAME.

The goal is to move all the n coins to the leftmost n squares of the paper. This is accomplished by players alternately moving a single coin, constrained by the following rules:

1. Coins move only to the left.
2. No coin may pass another.
3. No square may hold more than one coin.

If multiple people are playing, then the last person to move is the winner.

This week we will be thinking about how to represent the "coin strip" at the center of this challenge. There are many choices of how we might represent the state of the coin strip; your job is to identify one that seems appealing and make it the basis of your implementation. When your implementation is finished, you can test it using two existing applications—a solitare "puzzle", and a two-person "game".

The *Coin Strip Lab* involves designing and implementing a data structure for the Silver Dollar Game.

```java
 * The game is finished when the n coins occupy the leftmost n squares.
 */
public class Game
{
    public static void main(String[] args)
    {
        CoinStrip s = new CoinStrip();  // construct a coinstrip with coins
        Scanner input = new Scanner(System.in); // prepare to read moves
        String player0, player1;
        if (args.length == 2) {
            player0 = args[0];
            player1 = args[1];
        } else {
            player0 = "Alice";
            player1 = "Bob";
        }
        boolean player0Playing = true;

        // print out the strip of coins
        System.out.println(s);
        // play the game
        while (!s.gameOver()) {
            // there are moves left to go: read in a coin number and distance
            if (player0Playing) {
                System.out.println(player0 + ": enter a coin number and a distance.");
            } else {
                System.out.println(player1 + ": enter a coin number and a distance.");
            }
            boolean valid = true; // assume the move will be valid...
            do {
                int coin = input.nextInt(); // coin number
                int dist = input.nextInt(); // square count
```

```java
 * In a game, the winner is the person moving last.
 */
public class CoinStrip
{
    // TBS: private instance variables

    /**
     * A constructor to generate a random Silver Dollar Game.
     * @post A CoinStrip with at least three coins is constructed.
     */
    public CoinStrip()
    {
        // TBS: initialize variables
    }

    /**
     * Access the number of coins.
     * @post Returns the number of coins.
     * @return Number of coins in play.
     */
    public int numCoins()
        // post: returns number of coins in CoinStrip
    {
        // TBS: determine the number of coins
        return 0; // not correct
    }
```

Code for playing a 1-Player Puzzle or a 2-Player Game is given to you (e.g. `Game.java` on left).
But it won't run until you design and implement the `CoinStrip` data structure (start code on right).

Time-Complexity

Live Coding: Time Complexity

We'll discuss the basics of time counting during the live coding of the `Count.java` file.

- What if we change the static values `CAPACITY` and/or `MAXVALUE`? What if we make `CAPACITY` ten times larger? How much longer will the program run?
- Does second implementation of the loop save a *significant* amount of time? What exactly would we mean by significant?
- If `// do something` was simple (e.g., `print(i)`), then how much would the `if` statement contribute to the overall run-time? What if it was more complicated?
- How much time and space does `new int[CAPACITY]` take?

```
  GNU nano 4.8                          Count.java
import java.util.Random;

public class Count {
    public static void main(String[] args) {
        final int CAPACITY = 100;
        final int MAXVALUE = 1000;

        int[] A = new int[CAPACITY];
        Random rand = new Random();

        // How much time does this loop take?
        for (int i = 0; i < CAPACITY; i++) {
            A[i] = rand.nextInt(MAXVALUE);
        }

        // How much time does this loop take?
        // How much faster is this loop?

        // How much time does this if-statement take?

        // How much time  did the whole program take?
        // How much space did the whole program take?
    }
}
```

Starting code for `Count.java`.

```java
import java.util.Random;

public class Count {
    public static void main(String[] args) {
        final int CAPACITY = 100;
        final int MAXVALUE = 1000;

        int[] A = new int[CAPACITY];
        Random rand = new Random();

        // How much time does this loop take?
        for (int i = 0; i < CAPACITY; i++) {
            A[i] = rand.nextInt(MAXVALUE);
        }

        // How much time does this loop take?
        for (int i = 0; i < CAPACITY; i++) {
            for (int j = 0; j < CAPACITY; j++) {
                if (i != j && A[i] == A[j]) {
                    System.out.printf("A[%d] = A[%d] = %d%n", i,j,A[i]);
                }
            }
        }
        System.out.println("");

        // How much faster is this loop?
        for (int i = 0; i < CAPACITY-1; i++) {
            for (int j = i+1; j < CAPACITY; j++) {
                if (i != j && A[i] == A[j]) {
                    System.out.printf("A[%d] = A[%d] = %d%n", i,j,A[i]);
                }
            }
        }
        System.out.println("");
```

The loop runs `CAPACITY` times.
Each iteration takes the same time.

The outer loop runs `CAPACITY` times. On each iteration, the inner loop runs `CAPACITY` times. So the `if` statement runs `CAPACITY*CAPACITY` times.

Intuitively, this runs about half as long as the previous one since it only generates `i` and `j` pairs with `i < j`.
Alternatively, let's add the time for each iteration of the `i` loop.
The first iteration (i.e., `i = 0`) runs `CAPACITY-1` times.
The second (i.e., `i = 1`) runs `CAPACITY-2` times, etc.
Recall that $1+2+\ldots+n = n(n+1)/2$.
So the `if` statement runs `(CAPACITY-1)*CAPACITY/2` times in total.

Finished `Count.java`.

This will be true 50% of the time.

A pessimistic view would take the maximum of the two branches: max(`CAPACITY`, `MAXVALUE`).

```
        // How much time does this if-statement take?
        if (rand.nextInt(1) == 2) {
            for (int i = 0; i < CAPACITY; i++) {
                // do something
            }
        }
        else {
            for (int i = 0; i < MAXVALUE; i++) {
                // do something
            }
        }

        // How much time  did the whole program take?
        // How much space did the whole program take?
    }
}
```

In this branch,
the loop runs `CAPACITY` times.

In this branch,
the loop runs `MAXVALUE` times.

Finished `Count.java`.

Run-Time versus Time-Complexity

Run-Time

The time that a program takes when it is run.

Measured in a time unit (e.g. milliseconds, days).

- Consider the unix command `time`.

We can estimate the run-time using basic rules.

- How long does each instruction take? Often we estimate each to be one unit of time.
- Sequential instructions are added together.
- A loop is estimated by multiplying the number of loops by the time taken for each loop.
- Take the maximum of the two branches in an if-statement.

The amount of time is often *parameterized* based on certain values.

Time-Complexity

A more abstract measurement of the time complexity.

The focus is on how much time is taken relative to the size of the problem, or more specifically, the size of the input denoted `n`.

- Larger problems take longer to solve.

We use big-oh to provide more useful and concise measurements, and the analysis may involve simple proofs.

Time-complexity is often pessimistic (i.e., it considers worst-case performance).

Uses the same principles as run-time counting.

Big-Oh

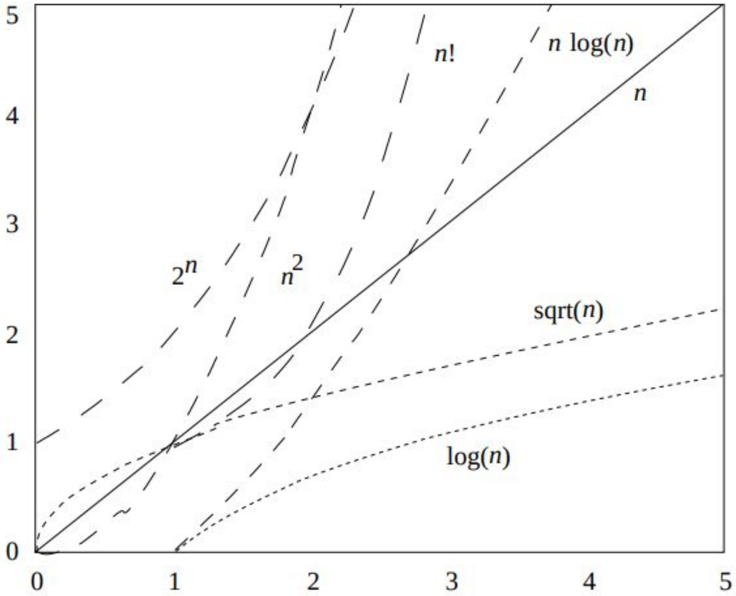


Figure 5.2 Near-origin details of common curves. Compare with Figure 5.3.

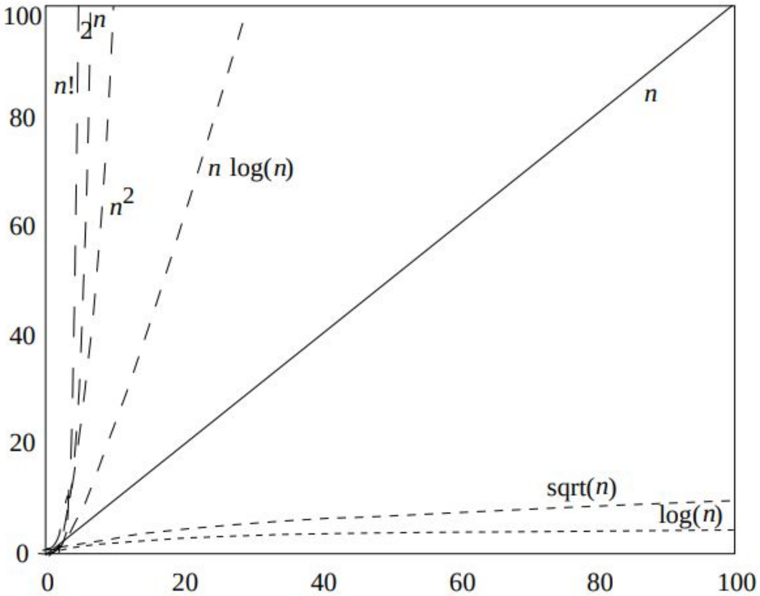


Figure 5.3 Long-range trends of common curves. Compare with Figure 5.2.

From the CSCI 136 textbook *Java Structures*.

We are interested in how the run-time of an algorithm f(n) grows as n → ∞.

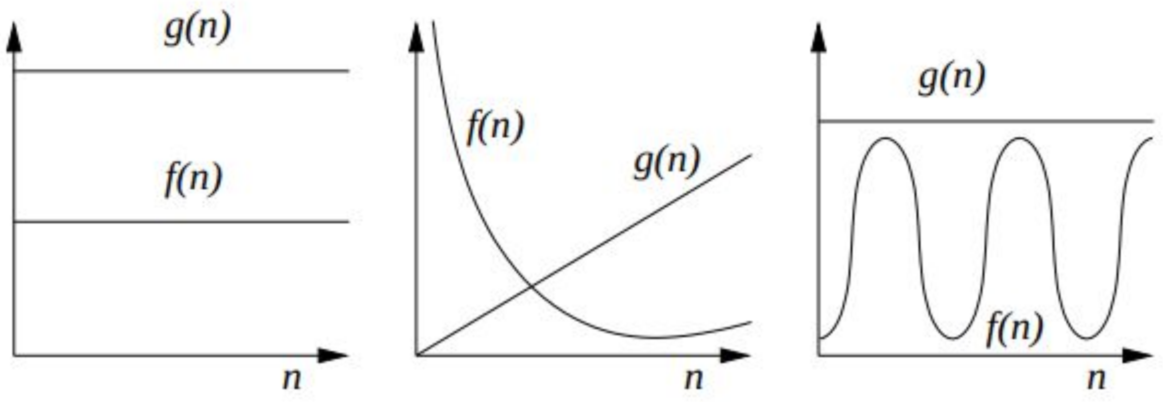


Figure 5.1 Examples of functions, $f(n)$, that are $O(g(n))$.

From the CSCI 136 textbook *Java Structures*.
We bound the run-time f(n) using a simpler function g(n).

Big-Oh

Exact runtime formulae are overly complicated.

We instead use big-oh notation as an estimate.

Big-oh is an upper bound that does two things:

- Remove lower order (ie slower growing) terms.
- Remove constant factors.

When measuring time we count each individual simple step counts as 1.

- Use addition for consecutive operations.
- Use multiplication for loops.

It's more accurate to refer to O(f(n)) as a set, e.g. $10n^2 + 3n \in O(n^2)$ or $10n^2 + 3n \in O(n^3)$.

Definition 5.1 *A function $f(n)$ is $O(g(n))$ (read "order g" or "big-O of g"), if and only if there exist two positive constants, c and n_0, such that*

$$|f(n)| \leq c \cdot g(n)$$

for all $n \geq n_0$.

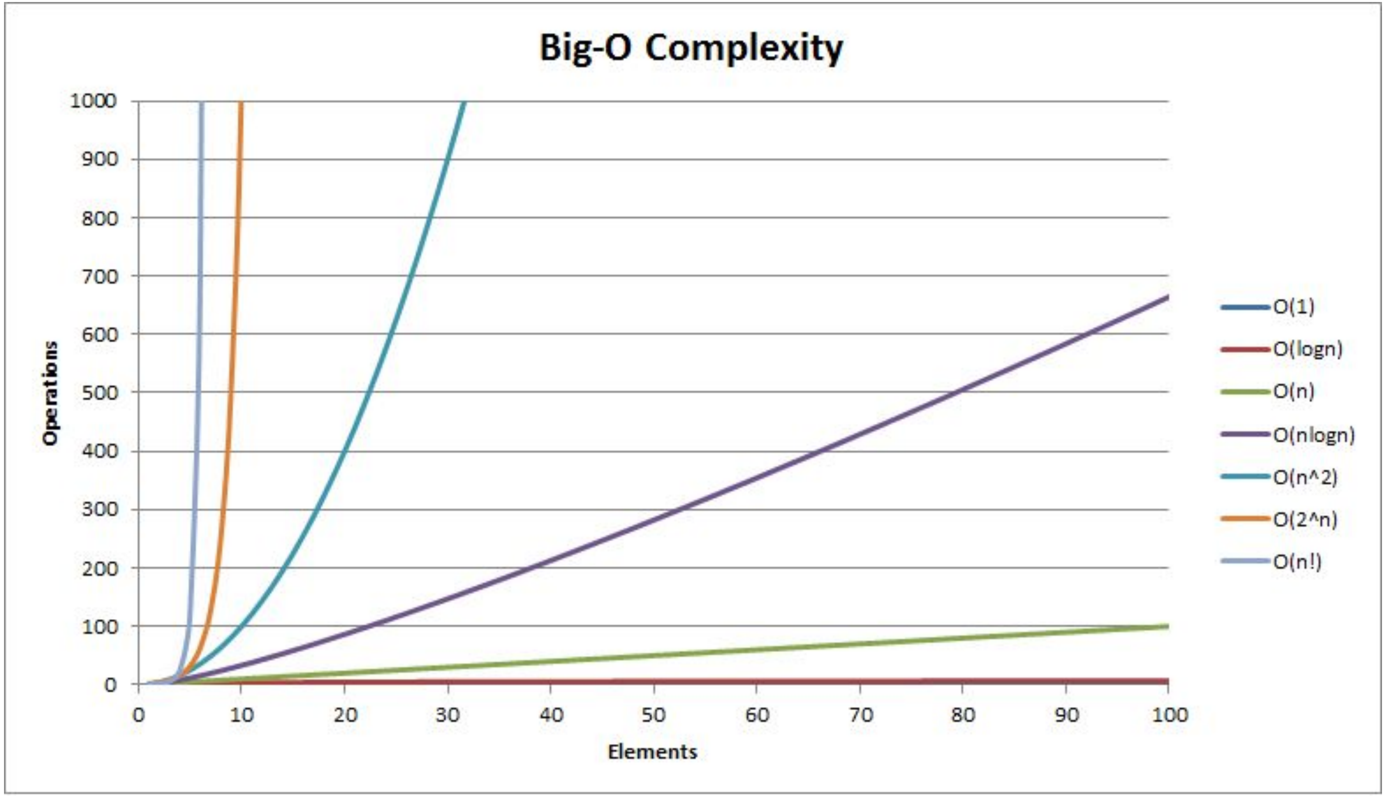
DEFINITION 7.2

Let f and g be functions $f, g \colon \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$,

$$f(n) \leq c\, g(n).$$

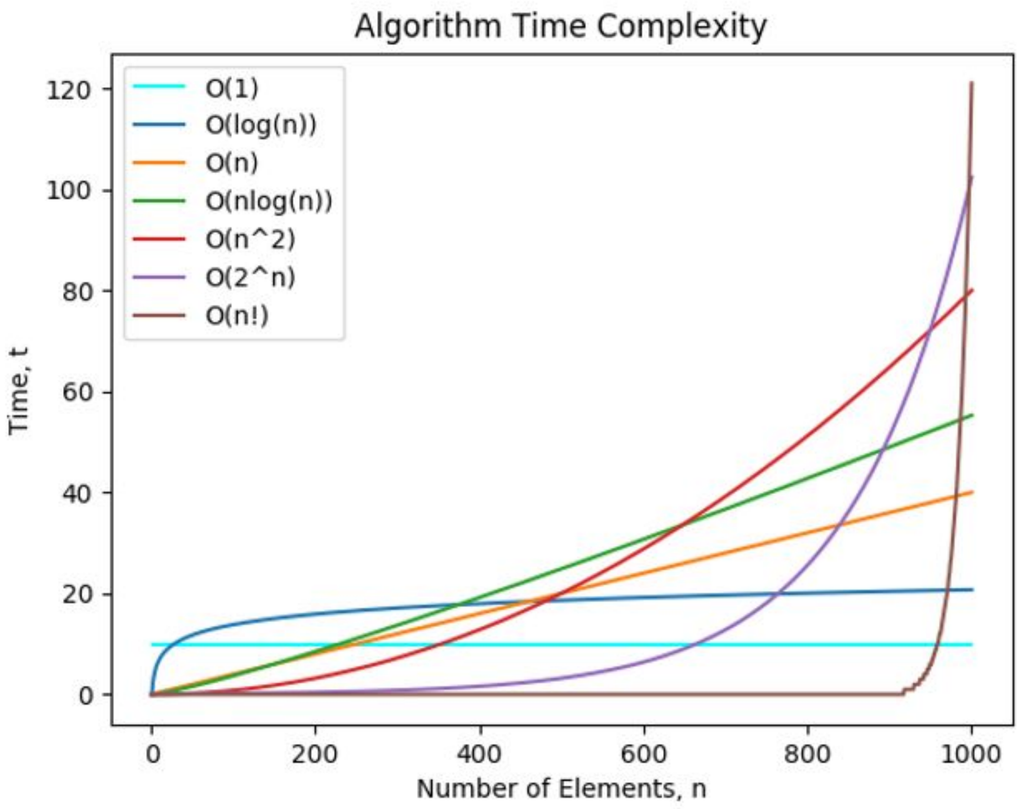
Asymptotic Upper Bounds Let $T(n)$ be a function—say, the worst-case running time of a certain algorithm on an input of size n. (We will assume that all the functions we talk about here take nonnegative values.) Given another function $f(n)$, we say that $T(n)$ *is* $O(f(n))$ (read as "$T(n)$ is order $f(n)$") if, for sufficiently large n, the function $T(n)$ is bounded above by a constant multiple of $f(n)$. We will also sometimes write this as $T(n) = O(f(n))$. More precisely, $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ so that for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$. In this case, we will say that T is *asymptotically upper-bounded by* f. It is important to note that this definition requires a constant c to exist that works for *all* n; in particular, c cannot depend on n.

Formal definitions of big-oh from various sources.



Growth of various functions (scale: n = 0 – 100).

- Notice that n^2 and 2^n don't look too different on this scale.



Growth of various functions (scale: n = 0 – 1000).

- Notice that n^2 and 2^n are starting to look very different on this scale.

Variants of Big-Oh

We also use the following variants of Big-Oh.

Bound	Notation	Name
$\leq$	O	big oh
$<$	o	little oh
$\geq$	Ω	big omega
$>$	ω	little omega
$=$	Θ	big theta

Variants of big-oh

For example, we'll argue that comparison-based sorting takes $\Omega(n \log(n))$-time later in the course.

Examples

Polynomials

Polynomials have the form $c_0 + c_1 n + c_2 n^2 + \ldots + c_k n^k$ where the c's are constant and k is a constant.

Note: A constant function is a (special type of) polynomial function.

Polynomials have excellent closure properties.

If p(x) and q(x) are both polynomials:

- $c \cdot p(x)$ is a polynomial for any constant c.
- $p(x) + q(x)$ is a polynomial.
- $p(x) \cdot q(x)$ is a polynomial.
- $p(q(x))$ is a polynomial (known as composition).

The last point implies that you can call a polynomial-time subroutine a polynomial number of times, and the resulting run-time will still be polynomial.

Note: Terms of the form $c_k n^k \log(n)$ are *polylogarithmic* with n log(n) arising frequently (e.g. merge sort).

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

How many ordered pairs of distinct values?

How many unordered pairs of distinct values?

Answers: n(n-1) and n(n-1)/2.

Exponential Functions

Exponential terms have the form $c \cdot b^{e(n)}$ where e(n) is some polynomial in n. When comparing these to polynomials, the variable n appears in the exponent rather than in the base.

Notes:

- The base of the exponent is important. For example, 3^n grows much faster than 2^n or 2^{10n}.
- If e(n) is logarithmic, then $c \cdot b^{e(n)}$ is polynomial.
- If e(n) is exponential, then $c \cdot b^{e(n)}$ is doubly exponential.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

How many subsets of numbers are there?

How many permutations of numbers?

Answers: 2^n and n!

Logarithms

Logarithms commonly arise in computer science when we repeatedly halve the search space.
(In some cases the halving is obtained using the help of a data structure.)

Notes:

- log log(x) is not the same as $\log^2(x)$.
- By default the base is assumed to be 2.
- We typically don't care about the base of the logarithm. This is due to the change of base formula which results in a constant factor.

$$\log_a n = \frac{\log_b n}{\log_b a}$$

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Guess my number between 1 - 100.
I will tell you lower or higher.
How many guesses?

Constant Functions

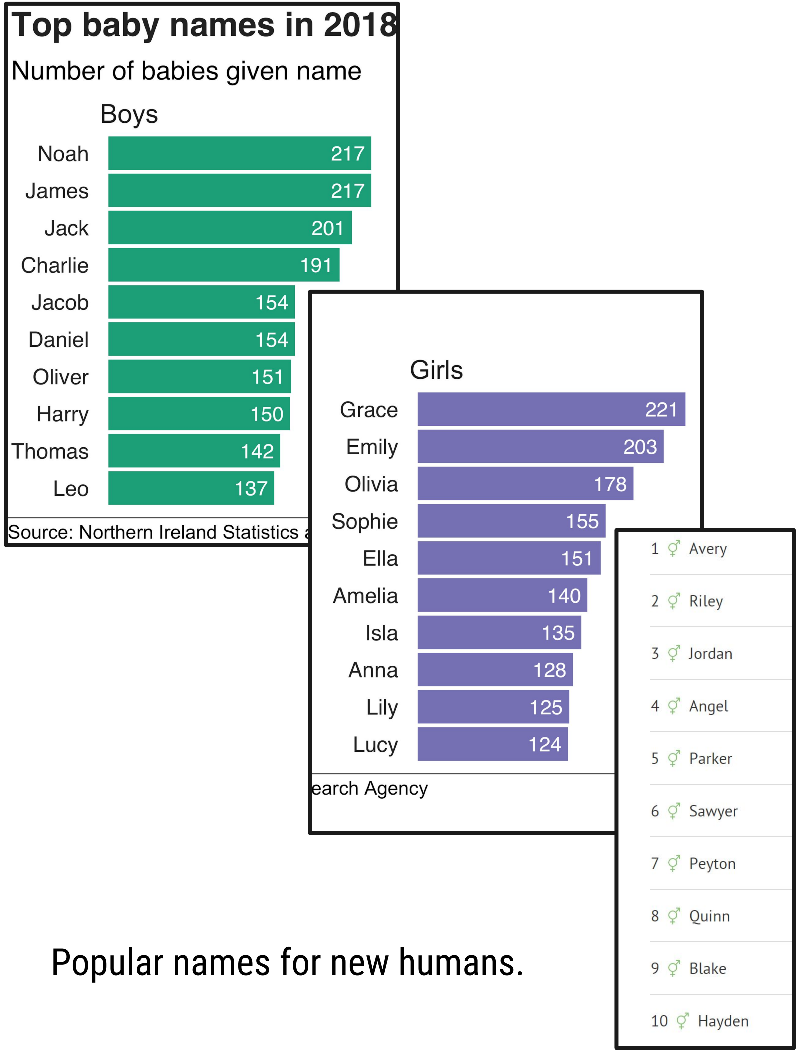
There are relatively few problems that can be answered in constant time. One example is below.

FIRST	
Input:	A non-empty list of integers L, and an integer k.
Output:	yes, if the first integer in L is k. Otherwise, no.

Constant functions also arise when analyzing problems whose input/instance size is bounded.

For example, the number of humans is $n \leq 8{,}000{,}000{,}000$ which is constant, so sorting human names can be done in constant-time. In particular, $n \log n \cong 32{,}000{,}000{,}000$.

This limitation is inherent to the way that we analyze algorithms. Claiming constant-time for every real-world instance is a great way to annoy computer scientists!



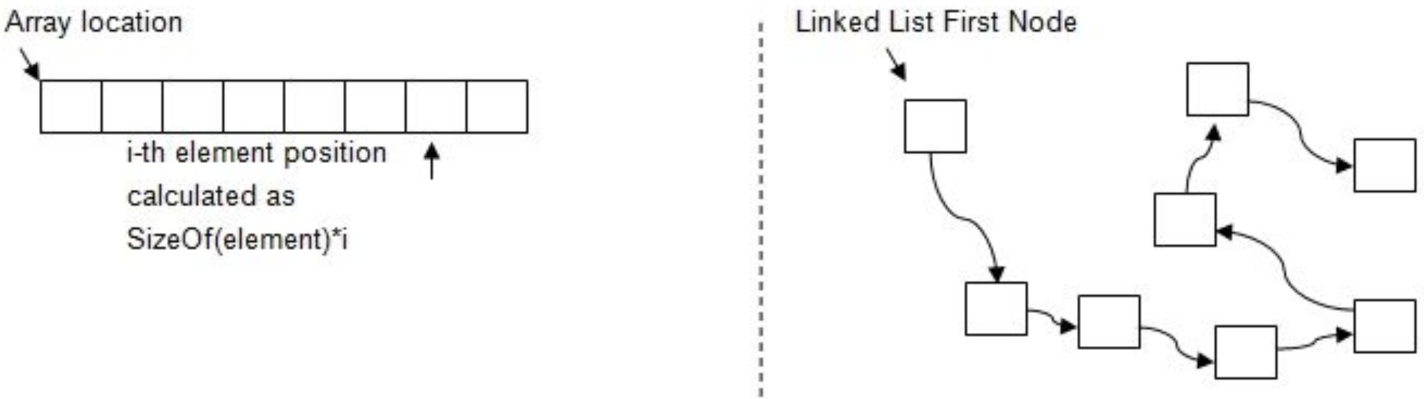
Popular names for new humans.

Addendum: Arrays vs Linked Lists

Sequential Data

There are two fundamental ways in which sequential data is stored in a computer.

1. Arrays. The values are positioned sequentially within memory.
   - Pros: Fast access based on position.
   - Cons: Homogeneous (data points have same size); fixed predetermined size; slow insert / delete.
2. Linked Lists. The values are chained together using pointers.
   - Pros: Easy to resize; fast insert / delete.
   - Cons. Slow access based on position; node types.



Linked lists have variations including (a) doubly linked, (b) circularly linked, (c) tail pointers.

We'll look at these in more detail later in the course.