Complexity Analysis (Big O)

- We want a way to characterize runtime or memory usage of an algorithm or data structure that's completely *platform-independent*.
    - i.e. doesn't depend on hardware, operating system, programming language, etc.
    - Algorithms themselves are platform independent.

- This will allow us to compare algorithms and data structures in the abstract and to evaluate how efficiently they will perform on increasingly large tasks.

- To do this, we describe how an algorithm or data structure's runtime or memory usage changes relative to a change in the input size.
    - Input size is usually denoted as the number of data elements n.

- Importantly, we want to describe how algorithms and data structures behave in the limit, as n approaches ∞.
    - An algorithm may have an expensive one-time startup cost but perform extremely efficiently afterwards.
        - e.g. a linear-time algorithm with an expensive startup vs. an exponential algorithm with inexpensive startup.

- Specifically, we describe algorithms in terms of *orders of growth*.
    - An algorithm grows *on the order of* some mathematical function if that function provides an upper-bound on the runtime beyond a certain input size n.

- An algorithm's order of growth is expressed with Big O notation.
    - e.g. the run time of insertion sort grows quadratically as the input size increasers, so we can say insertion sort is order

$O(n^2)$.

- Here's a simple example to consider, summing an array of *n* integers:

```
sum = 0;
for (i = 0; i < n; i++) {
    sum += array[i];
}
return sum;
```

    - The instruction `sum = 0` executes in some constant time c_1 independent of *n*.
    - Each iteration of the loop executes in some constant time c_2, and this happens *n* times.
    - The `return` statement executes in some constant time c_3 independent of *n*.
    - So runtime is $c_1 + c_2 n + c_3$.
    - c_1, c_2, and c_3 depend on the particular computer running this function, so we ignore them to figure out run-time complexity.
    - Thus, we say this sum function grows on the order of *n*, or, in other words that it is $O(n)$.
        - This is also known as a *linear-time* function.

- So, if our sum algorithm takes 32ms to sum 10,000 elements, how long will it take to sum 20,000?
    - For an $O(n)$ algorithm, if size doubles, execution time doubles.

- What about non-linear times?
    - e.g. bubble sort is $O(n^2)$. What happens if the size of the list to be sorted doubles?
    - It goes up by a factor of 4. Why?

Calculating wall clock time from Big O

- Let's keep looking at bubble sort, an $O(n^2)$ algorithm.

- $O(n^2)$ means that runtime is proportional to n^2.

- So, for two given sizes n_1 and n_2 and their runtimes t_1 and t_2, the ratio of Big O orders should equal the ratio of wall clock runtimes:

$$\frac{n_1^2}{n_2^2} = \frac{t_1}{t_2}$$

- Now double input size, so $n_2 = 2n_1$:

$$\frac{n_1^2}{(2n_1)^2} = \frac{t_1}{t_2}$$

- If we solve for t_2, we see that $t_2 = 4t_1$.

A more challenging wall clock time calculation

- Let's say you're investigating the performance of the mergesort algorithm on a particular kind of `struct`.
  - Merge sort is $O(n \log n)$.

- You've found that it takes 96ms to sort an array of 4000 of your `struct`s.

- About how long will it take to sort 1,000,000 `struct`s?

- Let's make some approximations to make the math a little easier, since we're working with base-2 logarithms:
    - 4000 ≈ 4096 = 2^{12}
    - 1,000,000 ≈ 1,048,576 = 2^{20}

- Now let's set the problem up as before, setting the ratio of Big O orders equal to the ratio of wall clock runtimes:

$$\frac{n_1 \ log \ n_1}{n_2 \ log \ n_2} = \frac{t_1}{t_2}$$

$$\frac{2^{12} log \ 2^{12}}{2^{20} \ log \ 2^{20}} = \frac{96ms}{t_2}$$

Remember that *log* $2^x = x$. Applying that formula and doing some cancelling, we get:

$$\frac{12}{2^8 \cdot 20} = \frac{96ms}{t_2}$$

Then doing some more cancelling and rearranging:

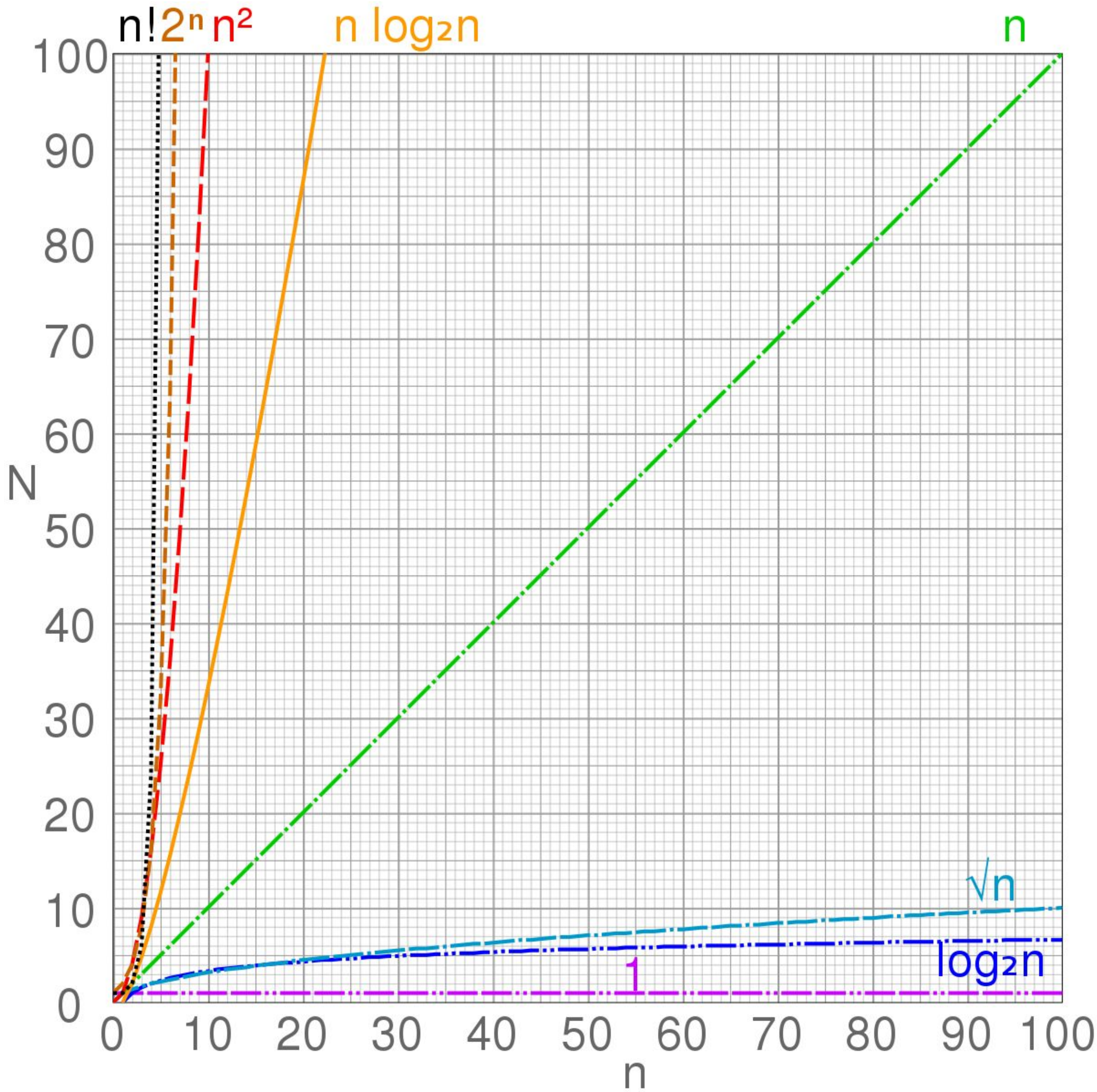
$$t_2 = 32ms \cdot 5 \cdot 2^8$$

And, multiplying out, we get:

$$t_2 = 40,960ms \approx 41s$$

Common growth order functions

- Below is a plot showing how several common growth order functions compare to each other in terms of growth rate:



By [Cmglee on Wikipedia](), [CC BY-SA 4.0]()

- These common growth order functions are referred to as follows:
  - *O(1)* – constant complexity
  - *O(log n)* – log-n complexity
  - *O(√n)* – root-n complexity
  - *O(n)* – linear complexity
  - *O(n log n)* – n-log-n complexity
  - *$O(n^2)$* – quadratic complexity
  - *$O(n^3)$* – cubic complexity
  - *$O(2^n)$* – exponential complexity
  - *O(n!)* – factorial complexity

Determining a program's complexity: nested loops

- Loops are one of the main determinants of a program's complexity.

- Here are the loop signatures for some common growth order functions:

  - *O(n):*

```
for (int i = 0; i < n; i++) {
    ...
}
```

  - *O(√n):*

```
for (int i = 0; i * i < n; i++) {
    ...
}
```

  - *O(log n):*

```
for (int i = n; i > 0; i /= 2) {
```

```
        ...
}
```

or

```
for (int i = 0; i < n; i *= 2) {
        ...
}
```

- Another common loop structure you'll see, which is $O(n^2)$ is:

```
for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
                ...
        }
}
```

  - Why is this $O(n^2)$?
  - Remember this from discrete math?

$$1 + 2 + 3 + ... + n = \frac{1}{2}n(n+1)$$

- When loops are nested, their individual growth orders are multiplied to compute the function's overall complexity.

- So, for example, when an $O(n)$ loop is nested within another $O(n)$ loop, the total complexity is $O(n^2)$. Similarly:
  - $O(\sqrt{n})$ loop inside $O(n)$ loop (and vice versa) → $O(n\sqrt{n})$.
  - $O(\log n)$ loop inside $O(n)$ loop (and vice versa) → $O(n \log n)$.
  - $O(\sqrt{n})$ loop inside $O(\sqrt{n})$ loop → $O(n)$.
  - $O(\log n)$ loop inside $O(\log n)$ loop → $O(\log^2 n)$.
  - $O(n)$ loop inside $O(n)$ loop inside $O(n)$ loop → $O(n^3)$.

Dominant components of growth order functions

- When a growth order function has additive terms, one of those terms will dominate the others.
  - Specifically, function $f(n)$ dominates $g(n)$ if

    $$\exists\, n_0 : \forall\, n > n_0,\ f(n) > g(n)$$

- In these cases, we simply ignore the non-dominant terms in our expression of the algorithm's complexity.

- For example, let's say we've analyzed a particular algorithm and found that it grows on the order of $n^2 + n + 1$.
  - In this case, the n^2 term dominates the others, so we say the algorithm's complexity is simply $O(n^2)$.

Worst case, best case, and average case

- It is important to note that the worst-case, best-case, and average-case complexities of an algorithm can differ.

- For example, consider a linear search algorithm:

```
int linear_search(int q, int* array, int n) {
    for (int i = 0; i < n; i++) {
        if (array[i] == q) {
            return i;
        }
    }
    return -1;
}
```

- Worst case: *O(n)*.
- Best case: *O(1)*.
- Average case: probably *O(n)*, but depends on data distribution.

- Quicksort also has different worst-, best-, and average-case complexities:
  - Worst case: $O(n^2)$.
  - Best case: *O(n log n)*.
  - Average case: *O(n log n)*.
  - Quicksort is still widely used because the worst case is very rare and the ignored constant factors make it typically faster than other *O(n log n)* sorts.

Constant factors can still be important in practice

- What if we're trying to choose between two algorithms whose growth orders we've computed:
  - *Algorithm 1* grows on the order of $1{,}000{,}000n \rightarrow O(n)$.
  - *Algorithm 2* grows on the order of $2n^2 \rightarrow O(n^2)$.

- Which algorithm do we choose?

- Just comparing computational complexities would lead us to choose *Algorithm 1*.

- However, for $n < 500{,}000$, *Algorithm 2* will actually run faster.

- In this case, it's important to *know your data* in addition to knowing the algorithms.
  - If your data will mainly have $n > 500{,}000$, then choose *Algorithm 1*. Otherwise, choose *Algorithm 2*.

- Quicksort is another good example

Empirical analysis: making a final choice

- Algorithmic complexity analysis can be a great aid in helping to compare algorithms at a high level and choose between them.

- Sometimes a comparison based on algorithmic complexity analysis is all we need to choose an algorithm.

- Often, though, algorithmic complexity analysis can only help us narrow the field of possible algorithms from which to choose.

- In these cases, an empirical performance analysis can help us choose the right algorithm.

- In a rigorous empirical analysis, we run all candidate algorithms on the same data on the same machine(s).
    - Many runs over the same data are made with each algorithm to eliminate possible outlier performances.
    - Ideally, the testing data is drawn from the same distribution as the data the algorithms will see when deployed.