THE UNIVERSITY OF VERMONT
COLLEGE OF ENGINEERING &
MATHEMATICAL SCIENCES

COMPLEXITY

EXAMPLES and CALCULATIONS

CS 124 / Department of Computer Science

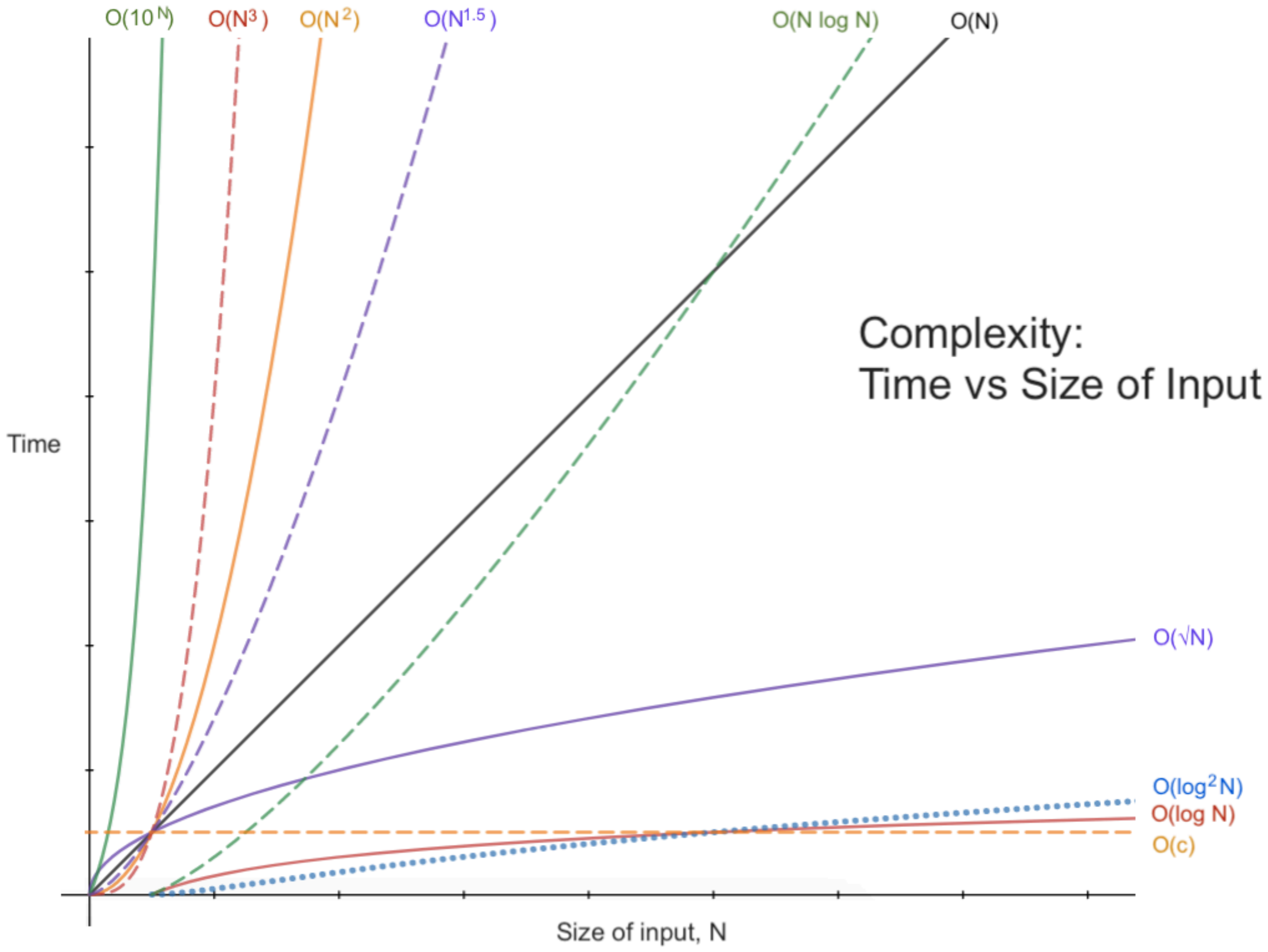
In this video...

In this video lecture we'll present

- concrete examples of different algorithms with different run-time complexities,
- rules of thumb for calculating Big *O* for a given algorithm, and
- rules for combining bounds.

But first a quick review.

Complexity measures



Describing bounds

Some common terminology

<table>
<tr><td>2^N</td><td colspan="2">Exponential</td></tr>
<tr><td>N^3</td><td>Cubic</td><td rowspan="2">Polynomial</td></tr>
<tr><td>N^2</td><td>Quadratic</td></tr>
<tr><td>N log N</td><td colspan="2"></td></tr>
<tr><td>N</td><td colspan="2">Linear</td></tr>
<tr><td>log2 N</td><td>Log squared</td><td rowspan="3">Sublinear</td></tr>
<tr><td>log N</td><td>Log</td></tr>
<tr><td>c</td><td>Constant</td></tr>
</table>

Measures of complexity

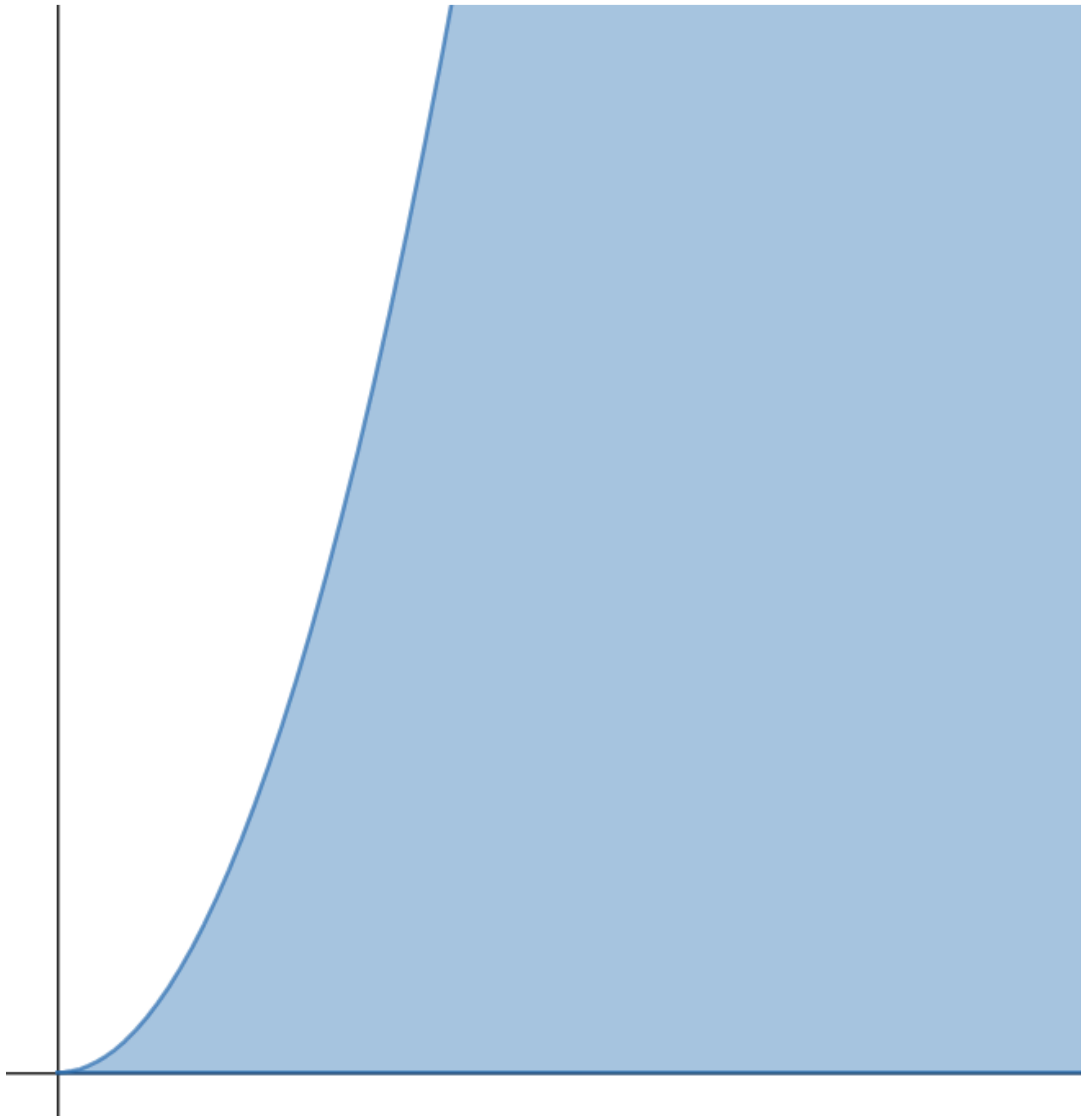
$T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

$T(N) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) \geq cg(N)$ when $N \geq n_0$.

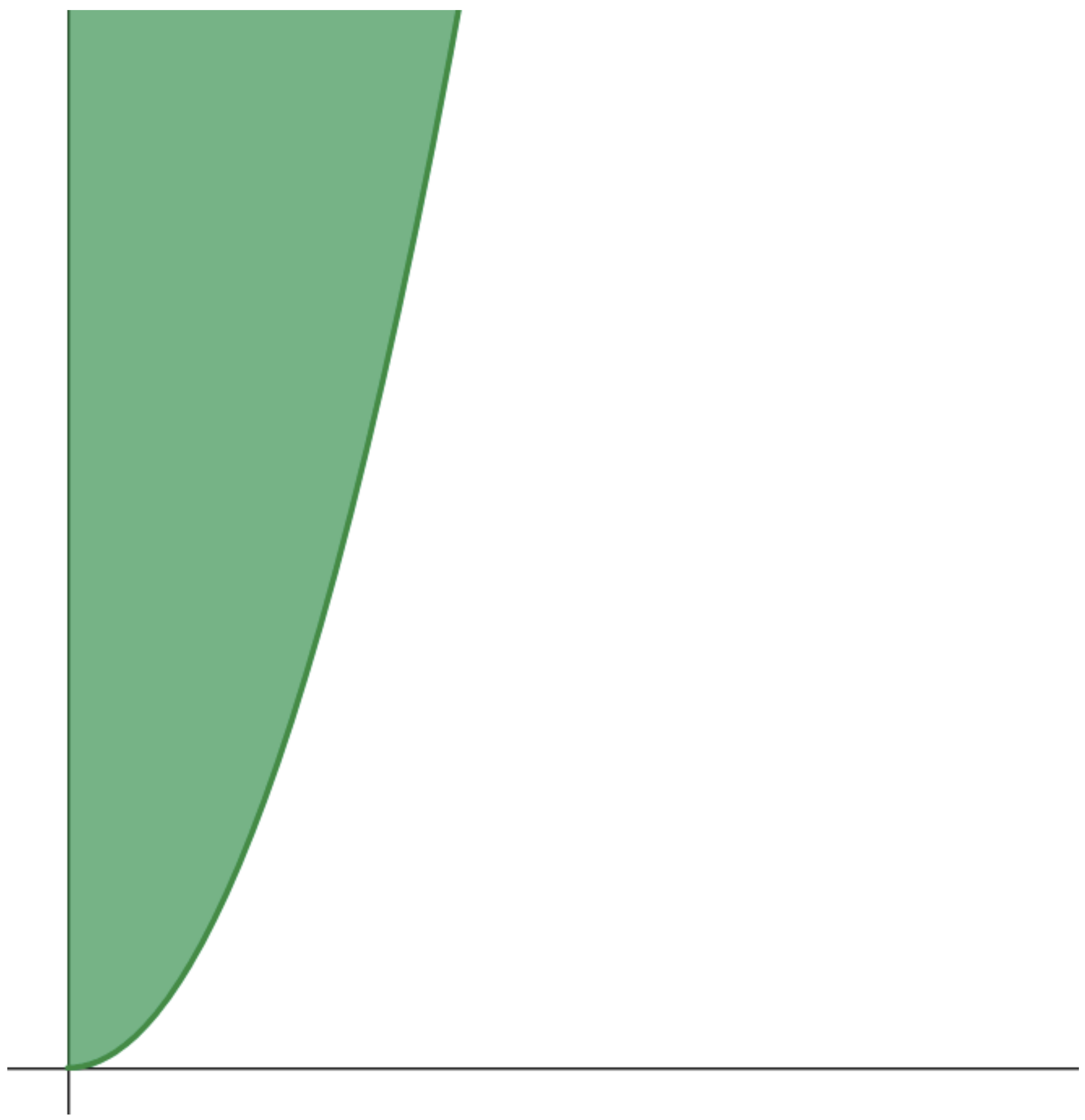
$T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.

$T(N) = o(p(N))$ if for all positive constants c there exists some n_0 such that $T(N) < cp(N)$ when $N > n_0$.

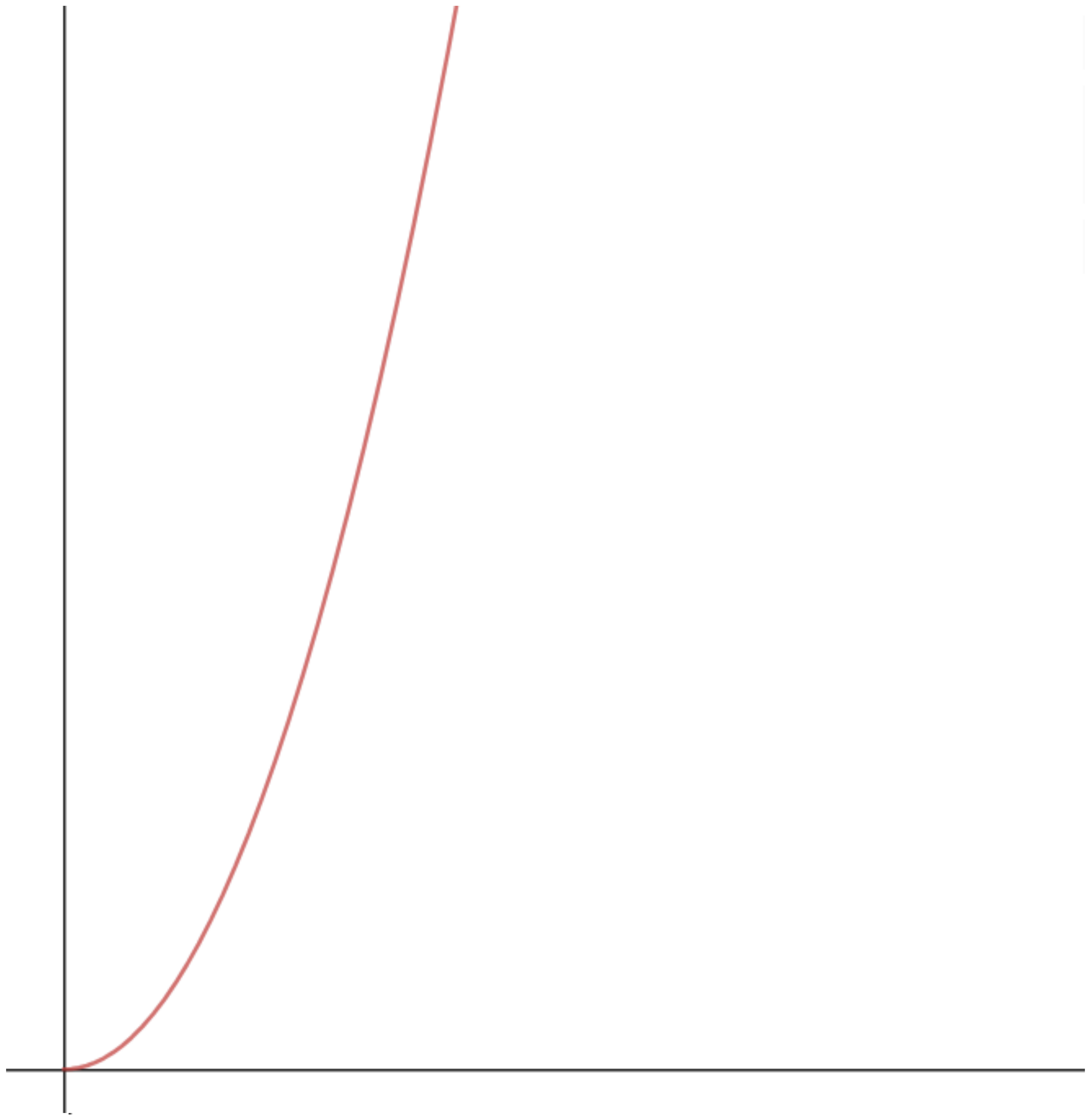
Big O is an upper bound



Ω is a lower bound



Θ is a tight bound



Dominating terms and ignoring constants

$$n^3 + 2n + 5 \rightarrow n^3$$

$$5n^2 \rightarrow n^2$$

$$n + 3 \rightarrow n$$

$$\log n + 1 \rightarrow \log n$$

Dominating terms and ignoring constants

$$n^3 + 2n + 5 \rightarrow n^3$$

$$5n^2 \rightarrow n^2$$

$$n + 3 \rightarrow n$$

$$\log n + 1 \rightarrow \log n$$

Dominating terms and ignoring constants

$$n^3 + 2n + 5 \rightarrow n^3$$

$$5n^2 \rightarrow n^2$$

$$n + 3 \rightarrow n$$

$$\log n + 1 \rightarrow \log n$$

Dominating terms and ignoring constants

$$n^3 + 2n + 5 \to n^3$$

$$5n^2 \to n^2$$

$$n + 3 \to n$$

$$\log n + 1 \to \log n$$

Examples

O(1): Constant time

Some functions or algorithms run in constant time, *O*(1). This means that their run time does not vary with the size of the input or that the size of the input is fixed.

For example:

- Node::getItem() method (seen in earlier lectures)
- Retrieving a value from an array by its index.
- Most arithmetic calculations, *e.g.,* computing the average of two doubles.

Examples

O(log *N*): Logarithmic time

Some functions or algorithms run in logarithmic time, $O(\log N)$. This means that their run time increases slowly, with the log of the size of the input.

For example:

- Binary search of a sorted list or binary search tree (BST)

Here, the number of steps required to complete a search increases by one with each doubling of the size of the input.

Examples

O(log *N*): Binary search of sorted list

Let's say we wanted to find if a number exists in this sorted list.

7	19	23	30	42	43	49	55

We could perform a linear search, checking each element one at a time from left to right. But that might take eight comparisons. There are eight elements in the list, and each comparison counts for a step in our algorithm, therefore linear search has complexity $O(N)$.

Examples

O(log *N*): Binary search of sorted list

But we can do better. Say we're looking to see if 42 is in this list. First we check to see if it's in the first half.

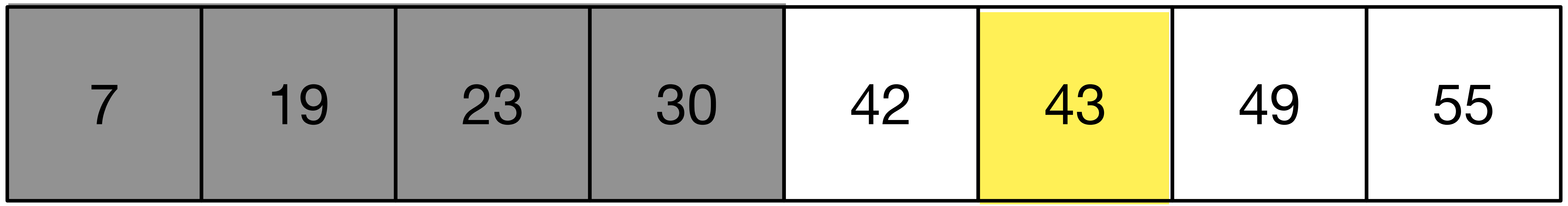
7	19	23	30	42	43	49	55

Then we find it's not in the first half, so we know if it's in the list it must be in the second half. So next we can search the first half of the second half.

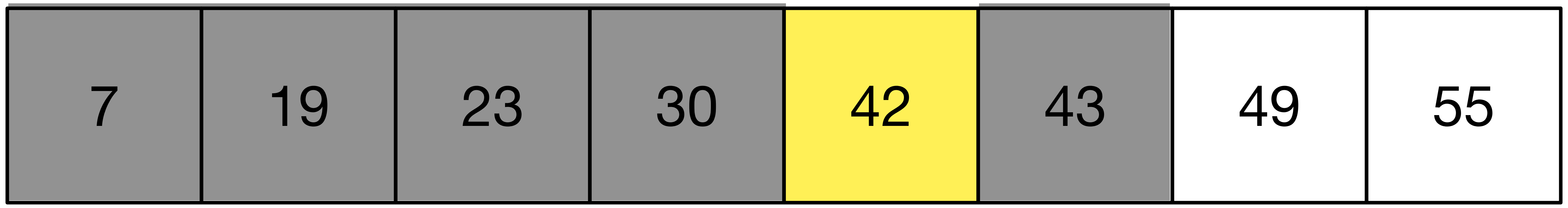
Examples

O(log *N*): Binary search of sorted list

Checking the first half of the second half, we find that this contains values greater than 30 and less than or equal to 43.



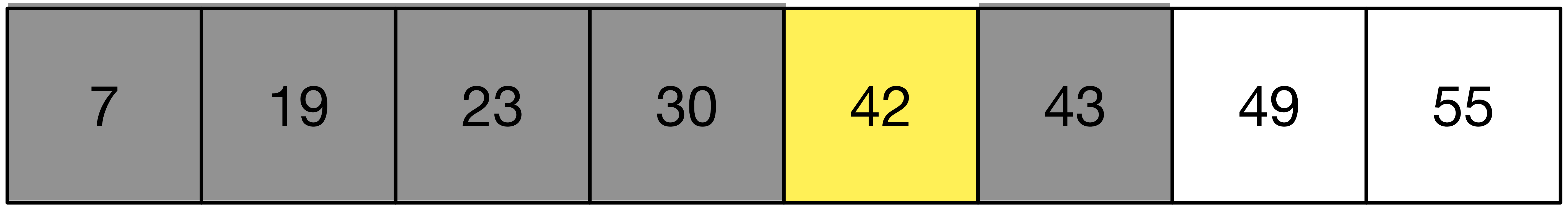
So we halve our search space again, and check.



Examples

O(log *N*): Binary search of sorted list

We found the target in the list in three steps. This should come as no surprise, since we started with eight sorted elements and halved the search space each time. Note that $8 = 2^3$.



In the worst case this search might have taken (log *N*) + 1 steps (searching for the value 55). But we've seen that extra step doesn't really matter, and that we ignore such constants. So in this case, complexity is *O*(log *N*).

Examples

O(log *N*): Binary search of sorted list

If we were to double the size of the input, our search would take just one more step. So again, our run time complexity is *O*(log *N*).

In general, any algorithm that continually halves its search or working space will run in *O*(log *N*) time.

Just as an aside, what do you think would be the *best case* for binary search of a sorted list?

Examples

O(log *N*): Binary search of sorted list

The best case is where the search target is at index $\lfloor N / 2 \rfloor$ within the list.

This is the first element to be checked in a binary search.

Examples

O(log *N*): Binary search of sorted list

The best case is where the search target is at index $\lfloor N / 2 \rfloor$ within the list.

This is the first element to be checked in a binary search.

So in the best case, the search time is $O(1)$ even though the average and worst cases are of order $O(\log N)$!

We'll revisit best case analysis at various points during the course. But for now, let's move on and look at some other algorithms.

Examples

$O(N)$: Summing an array / looping over an array

Some functions run in $O(N)$ or what we call linear time. In these cases, the run time varies directly with the size of the input.

A typical example of an $O(N)$ function is summing all the values in an integer array.

In general, this will hold for any algorithm that has to loop through the length of an array and perform some simple calculation.

Examples

$O(N \log N)$: Divide and conquer algorithms

A typical example of a class of algorithms with complexity $O(N \log N)$ is "divide and conquer" algorithms. These algorithms split a problem instance into two halves, solve each half, and then combine the results.

We'll defer discussion and analysis of this important class of algorithms until a little later in the course.

Examples

O(*N*²): Quadratic time

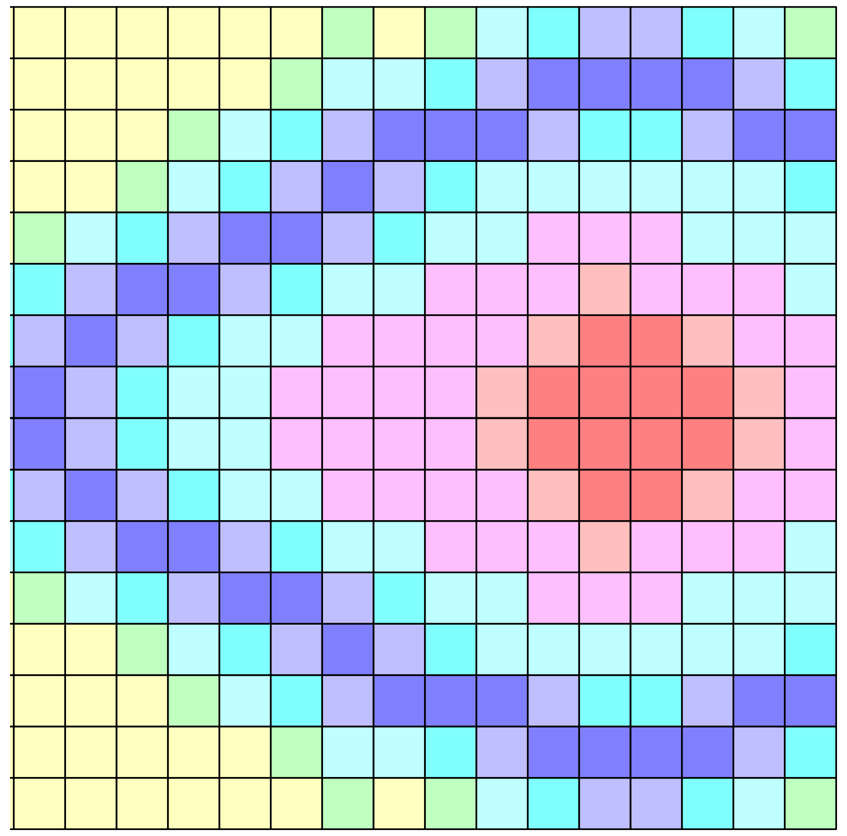
In general, any algorithm with two nested loops will run in quadratic time. That is, it will have complexity of $O(N^2)$.

Let's look at an example to see why this is so.

Examples

$O(N^2)$: Quadratic time

Let's say we wanted to calculate the average color of a collection of pixels.



The pixels would be stored in a two-dimensional array. To perform the calculation, we'd have to iterate through each row, and then through each column within each row. With M rows and N columns we'd need to check $M \times N$ pixels. Assuming a square array this would be N^2 pixels.

Examples

$O(N^2)$: Quadratic time

So for each pixel, we'd sum the RGB values, and then divide by the total number of pixels. Calculating the sum would take constant time, calculating the average from the sum would take constant time.

So what varies with the input is the number of pixels we need to check. Hence, this executes in $O(N^2)$ time.

Examples

Exponential time: recursive Fibonacci

The example of recursive calculation of a Fibonacci number we saw in an earlier lecture runs in exponential time.

(We warned you about inappropriate use of recursion!)

Now this is a little different. What does it mean for this to be exponential? There's no data being processed.

Examples

Exponential time: recursive Fibonacci

Here's the recursive Fibonacci algorithm we saw earlier:

```
int fibonacci(int x) {
    if (x == 0) || (x == 1) {
        return(x);
    } else {
        return(fibonacci(x - 2) + fibonacci(x - 1));
    }
}
```

Examples

Exponential time: recursive Fibonacci

Now let's count the number of recursive calls needed to calculate the n^{th} Fibonacci number.

F_0 and F_1 require zero recursive calls. These just return a value right away. Calculating F_2 requires two recursive calls, one to get the value of F_0 and another to get the value of F_1.

Let's keep track with a table.

Examples

Exponential time: recursive Fibonacci

n	number of recursive calls
0	0
1	0
2	2
3	4
4	8
5	14
6	24
7	40
8	66

Examples

Exponential time: recursive Fibonacci

n	number of recursive calls
9	108
10	176
11	286
12	464
13	752
14	1,218
15	1,972
16	3,192
17	5,166

Examples

Exponential time: recursive Fibonacci

We can see that the number of recursive calls needed to calculate the n^{th} Fibonacci number explodes exponentially. In fact, using the algorithm shown it would take 331,160,280 recursive calls to calculate F_{40}!

Since the amount of work done increases exponentially with the input n, we say this is an exponential time algorithm.

Combining bounds

If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then

$$T_1(N) + T_2(N) = O(f(N) + g(N)) = O(\max(f(N), g(N)))$$

and

$$T_1(N) \times T_2(N) = O(f(N) \times g(N))$$