Performance of dynamic arrays - simpler

Suppose the array has capacity 2^n

It must have been expanded n times: $1 \rightarrow 2 \rightarrow 4 \rightarrow \ldots \rightarrow 2^{n-1} \rightarrow 2^n$

The total number of copied elements is $1 + 2 + 4 + \ldots + 2^{n-1} = 2^n - 1$

If the array has size m, its capacity is at most 2m, so the number of copied elements is at most 2m-1

Binary search

Complexity

Weiss chapter 5

Searching

Suppose I give you an array, and ask you to find a particular value in it, say 4.

5	3	9	2	8	7	3	2	1	4

The only way is to look at each element in turn.

This is called *linear search*.

Performance of linear search

If we are unlucky, the item we are looking for will be the last one in the array

So, if the array has size n, we might need to look at n elements

Searching

But what if the array is sorted?

1	2	2	3	3	4	5	7	8	9

There is a better way, called *binary search*.

Binary search

Suppose we want to look for 4.

We start by looking at the element half way along the array, which happens to be 3.

1	2	2	3	3	4	5	7	8	9

Binary search

3 is less than 4.

Since the array is sorted, we know that 4 must come after 3.

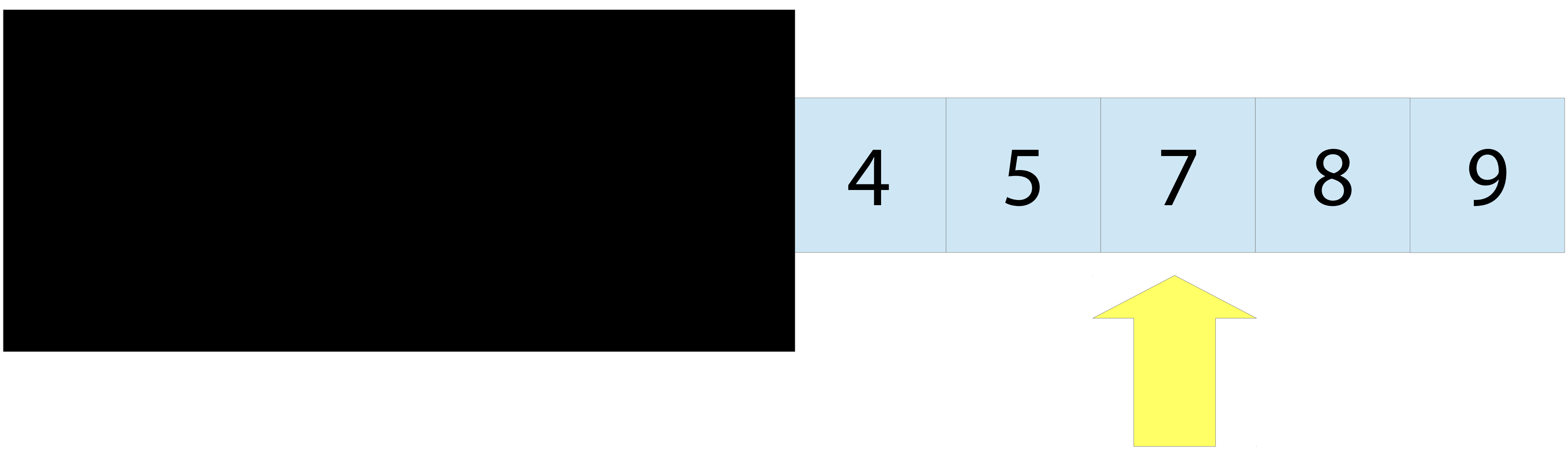
We can ignore everything before 3.

[redacted]	4	5	7	8	9

Binary search

Now we repeat the process.

We look at the element half way along what's left of the array. This happens to be 7.

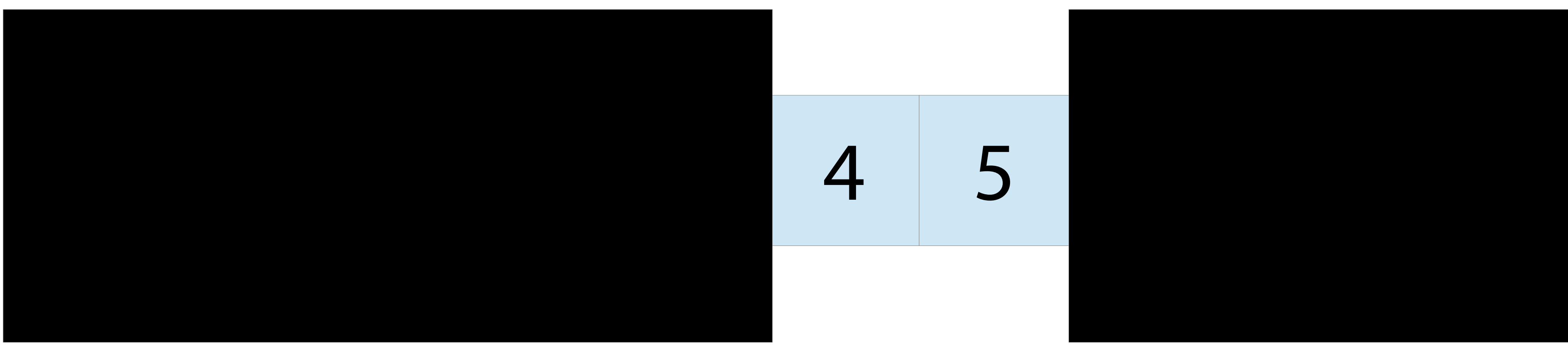


Binary search

7 is greater than 4.

Since the array is sorted, we know that 4 must come before 7.

We can ignore everything after 7.

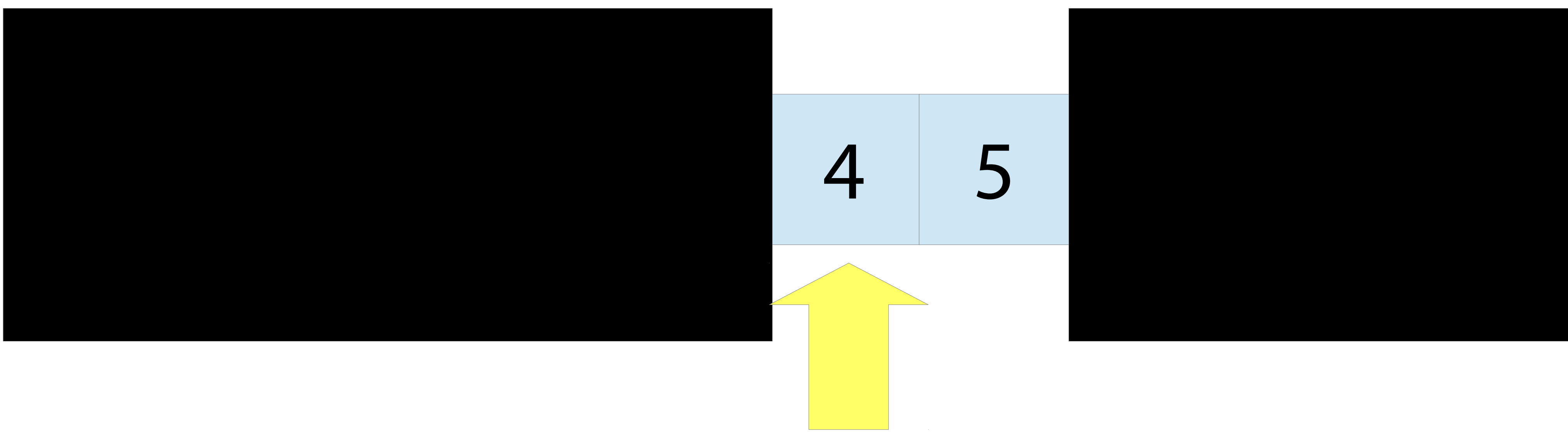


Binary search

We repeat the process.

We look half way along the array again.

We find 4!



Implementing binary search

Keep two variables `low` and `high`, representing the part of the array to search

Let `mid = (low + high) / 2` and look at `a[mid]`

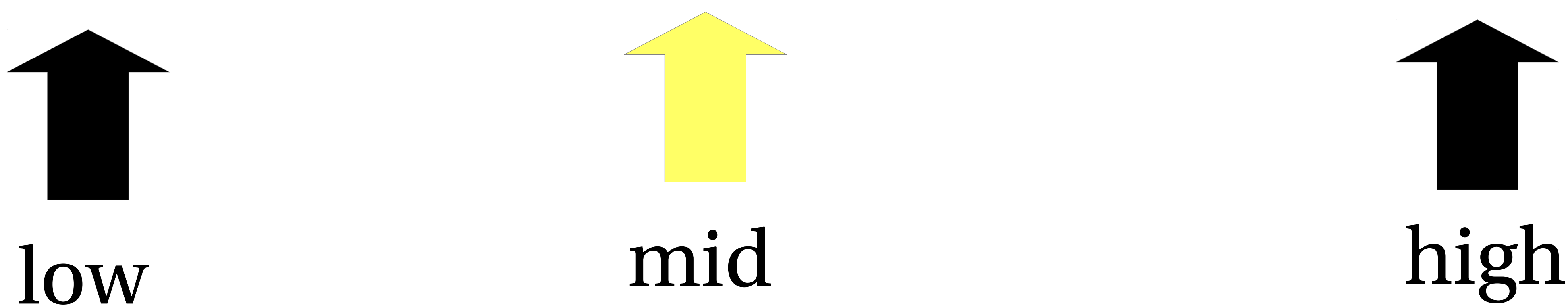
Depending on the answer, cut off parts of the array by adjusting `low` and `high`

Binary search

Looking for 4 again:

```
mid = (low + high)/2
```

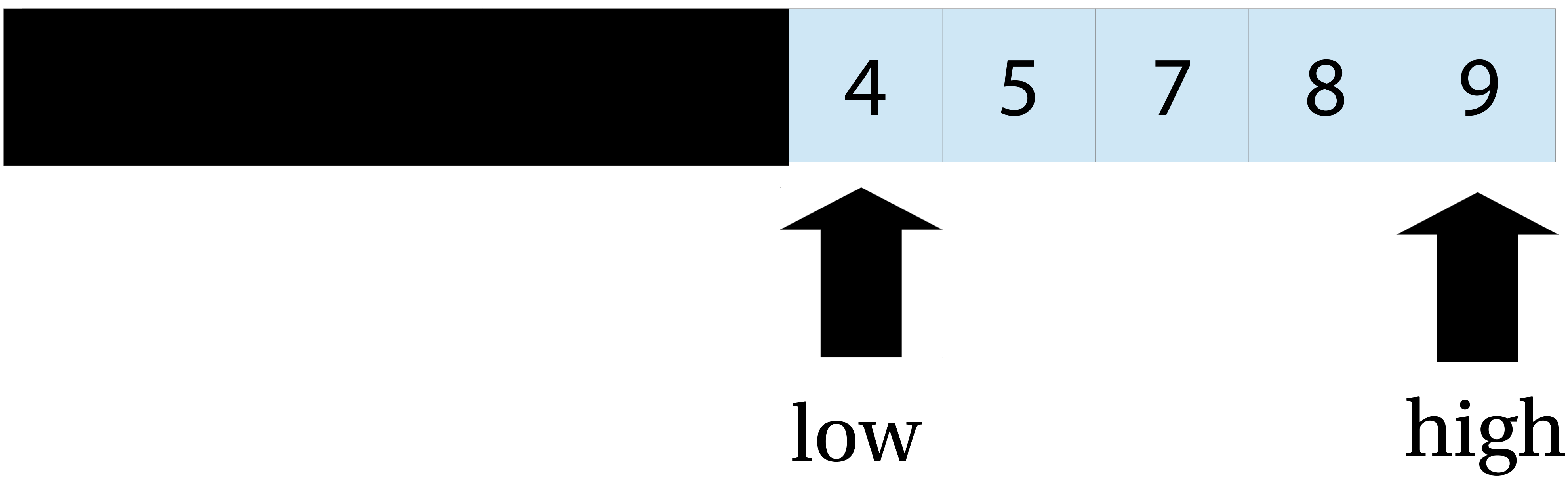
1	2	2	3	3	4	5	7	8	9



Binary search

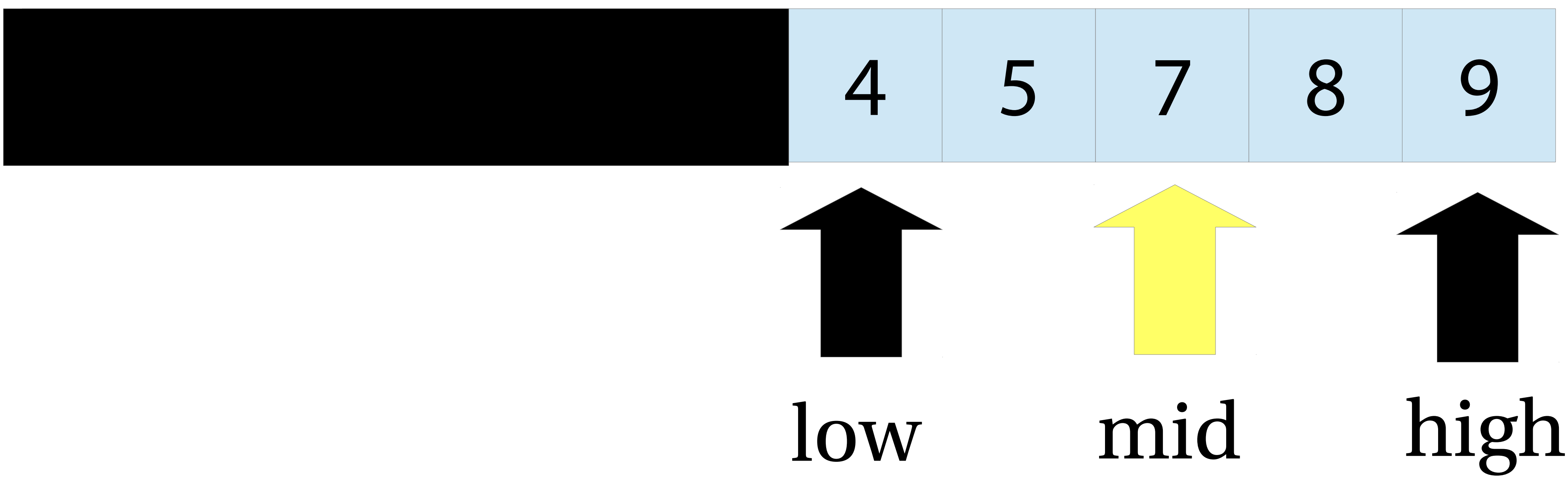
Cut off everything below mid:

```
low = mid + 1
```



Binary search

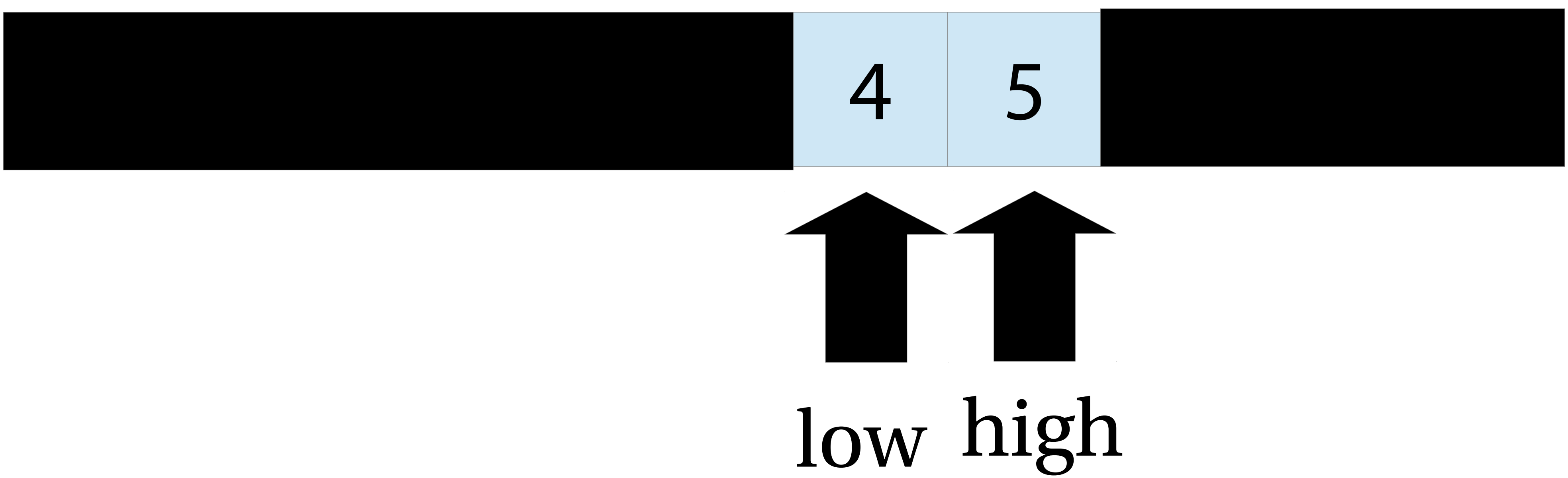
```
mid = (low + high)/2
```



Binary search

Cut off everything above mid:

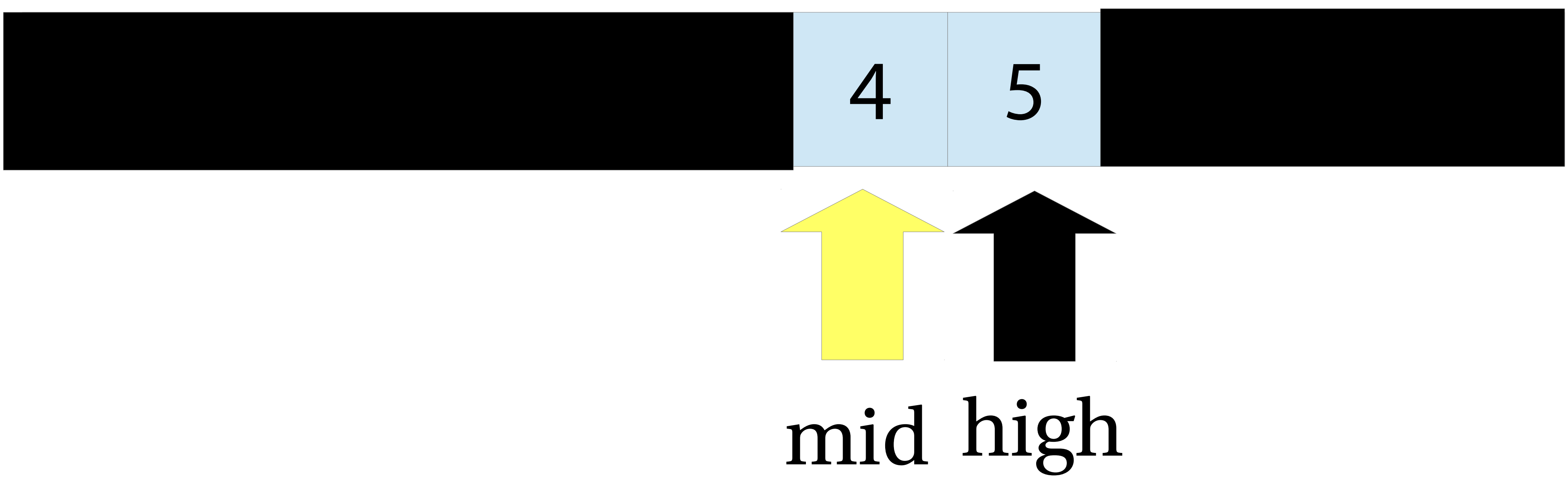
```
high = mid - 1
```



Binary search

Found it!

```
mid = (low + high)/2
```



```
public static <E extends Comparable<? super E>>
E binarySearch(E[] a, E x)
{
  int low = 0;
  int high = a.length – 1;
  int mid;
  while (low <= high) {
    mid = (low + high)/2;
    if (a[mid].compareTo(x) < 0)
      low = mid + 1;
    else if (a[mid].compareTo(x) > 0)
      high = mid – 1;
    else
      return a[mid];
  }
  return null;
}
```

Weiss
section 4.7

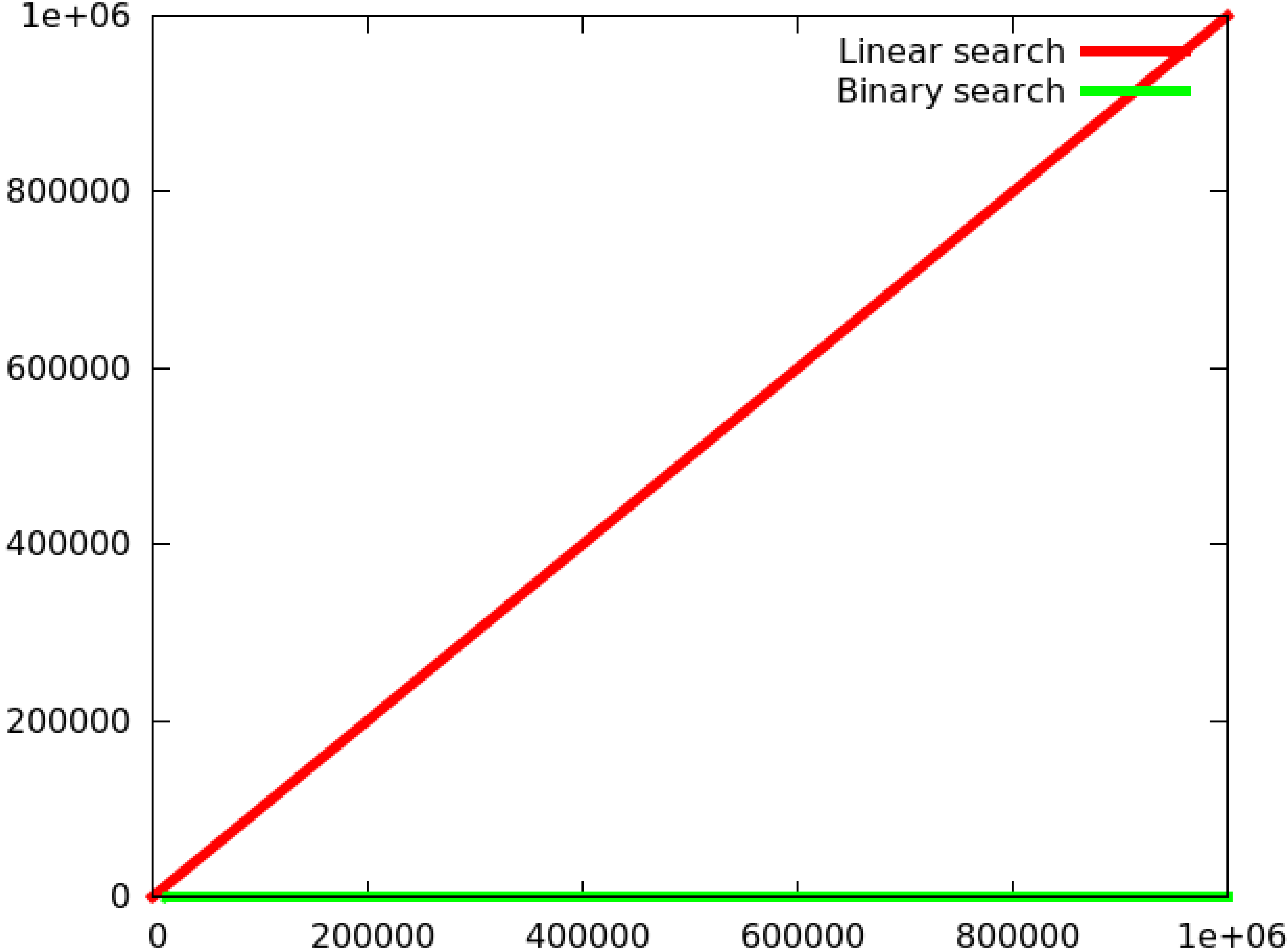
Performance of binary search

Every time we look at an element, we cut `high - low` in half

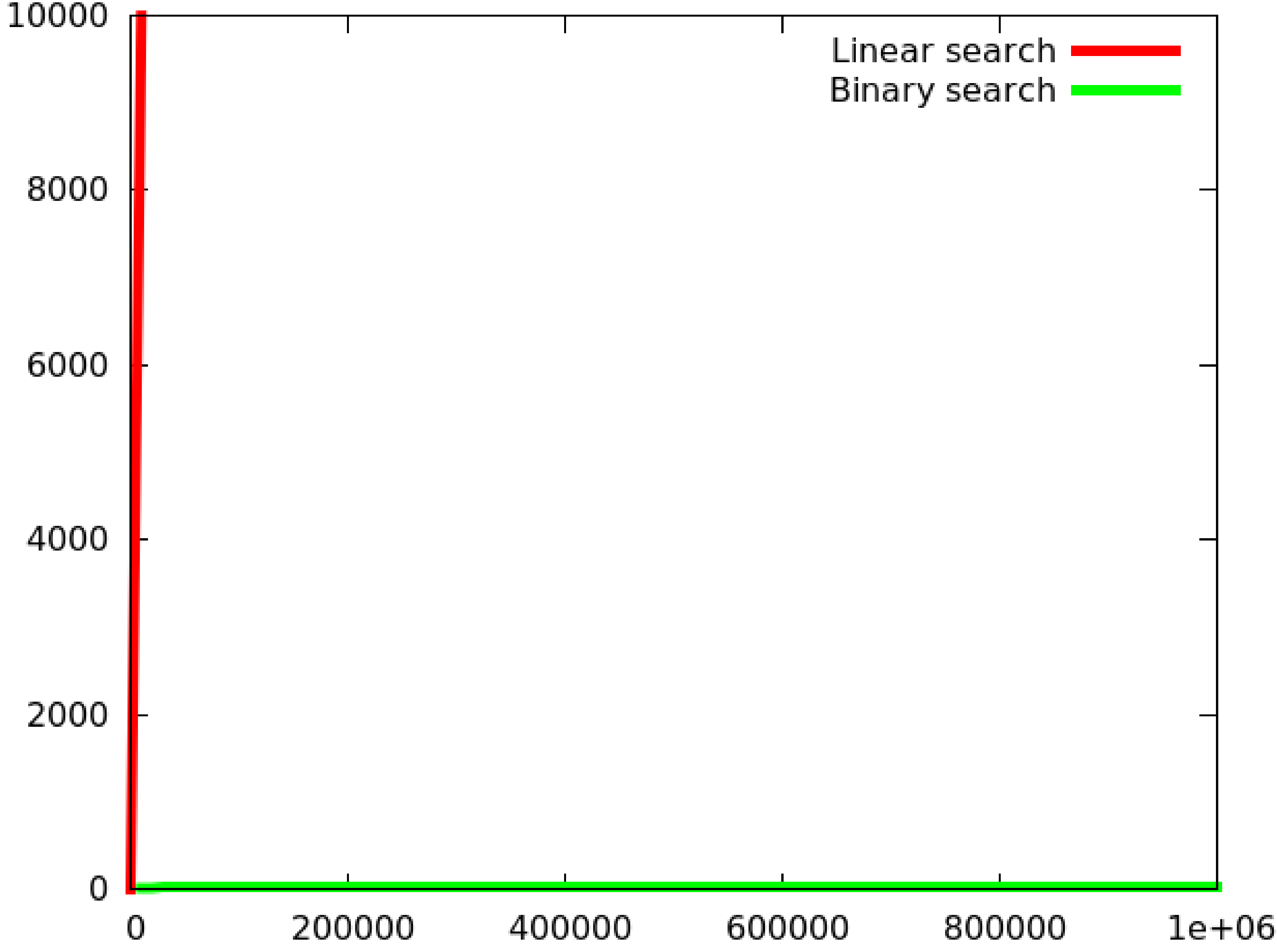
With an array of size 2^n, after n searches, we are down to 1 element

On an array of size n, need to look at **$\log_2 n$** elements!

Performance – a graph



Zoom in!



10000
8000
6000
4000
2000
0
0
200000
400000
600000
800000
1e+06
Linear search
Binary search

Binary search needs to look at only *20 elements* for an array of size one million!
30 for an array of size *one billion*!

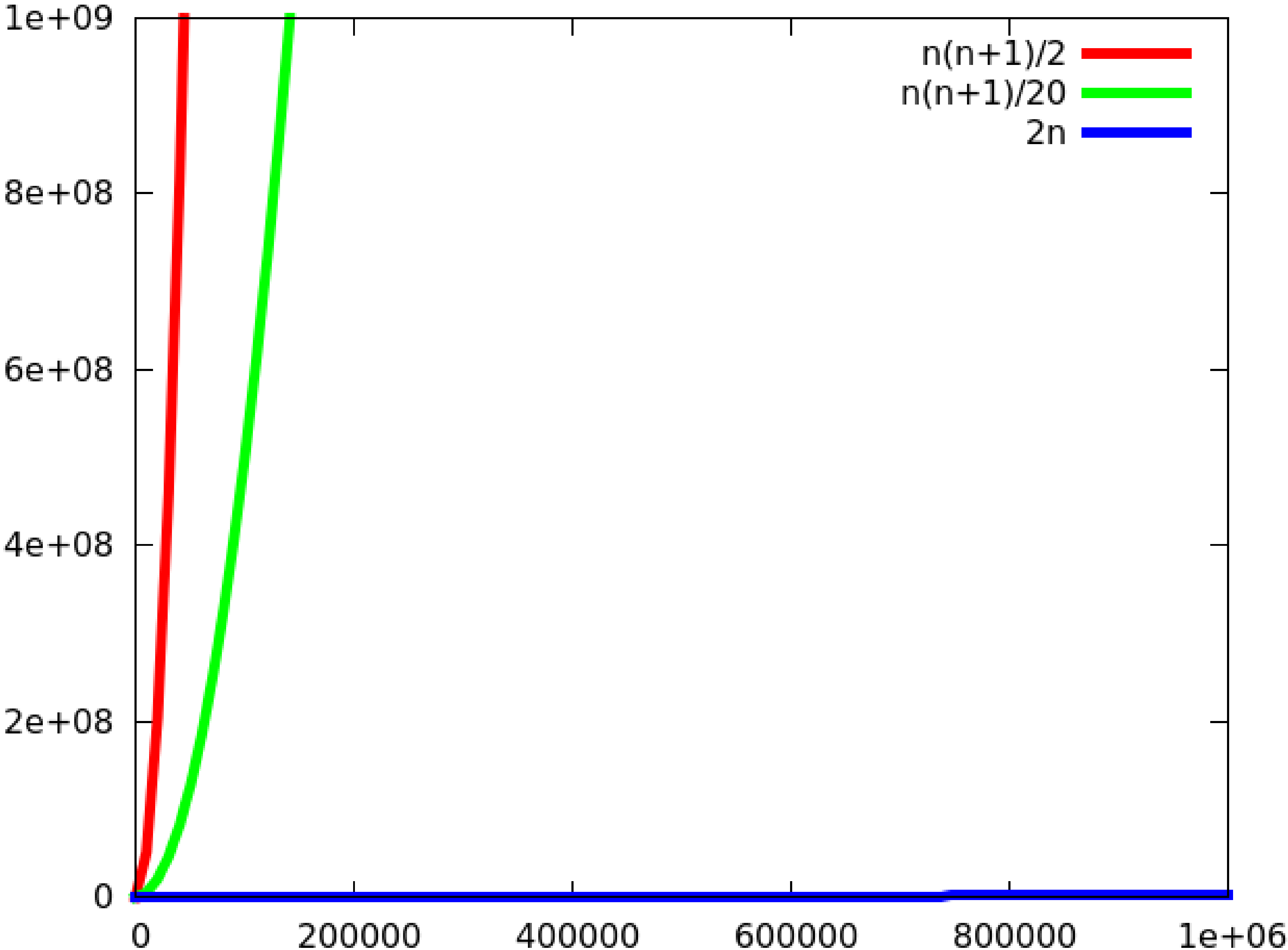
Arrays.binarySearch

You can find this in `java.util.Arrays`:

```
int binarySearch
  (Object[] a, Object key)
```

- Returns an *index*, not a value, and returns a negative number if not found

Complexity
(reasoning about performance)


1e+09
8e+08
6e+08
4e+08
2e+08
0
0
200000
400000
600000
800000
1e+06
n(n+1)/2
n(n+1)/20
2n

Big idea:
Let's ignore constant factors!

When n is 1000000…

- $\log_2 n$ is 20
- n is 1000000
- n^2 is 1000000000000
- 2^n is a number with 300,000 digits…

An algorithm that takes 1000n steps trounces one that takes n^2 steps

A corollary:
the speed of the computer
doesn't matter
(count *number of steps*
instead of *amount of time*)

How many steps?

```
Object search(Object[] a, Object x) {
  for(int i = 0; i < a.length; i++) {
    if (a[i].equals(target))
      return a[i];
  }
  return null;
}
```

Assume that loop body takes 1 step

Linear search is **O(n)**:
amount of time taken is proportional to **n**,
where n is `a.length`

("linear complexity")

Big-O complexity

"The time taken is proportional to…"

- $O(n)$: time is proportional to input size
- $O(n^2)$: time is proportional to square of input size
- $O(\log n)$: time is proportional to log of input size ("logarithmic complexity")
- $O(1)$: takes constant time

Complexity is also called *growth rate*

Can measure things other than time too

Dynamic arrays: consumes O(n) space for n adds

Binary search: does O(log n) comparisons for an array of size n

How many steps?

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < a.length; j++)
      if (a[i].equals(a[j]) && i != j)
        return false;
  return true;
}
```

Assume that loop body takes 1 step

$O(n^2)$, where $n =$ `a.length`
("quadratic complexity")

(outer loop runs n times,
inner loop runs n times for each run
of the outer loop, giving $n \times n$)

How many steps?

```
boolean disjoint(Object[] a, Object[] b) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < b.length; j++)
      if (a[i].equals(b[j]))
        return false;
  return true;
}
```

O(mn), where

m = `a.length`

n = `b.length`

Big O, formally

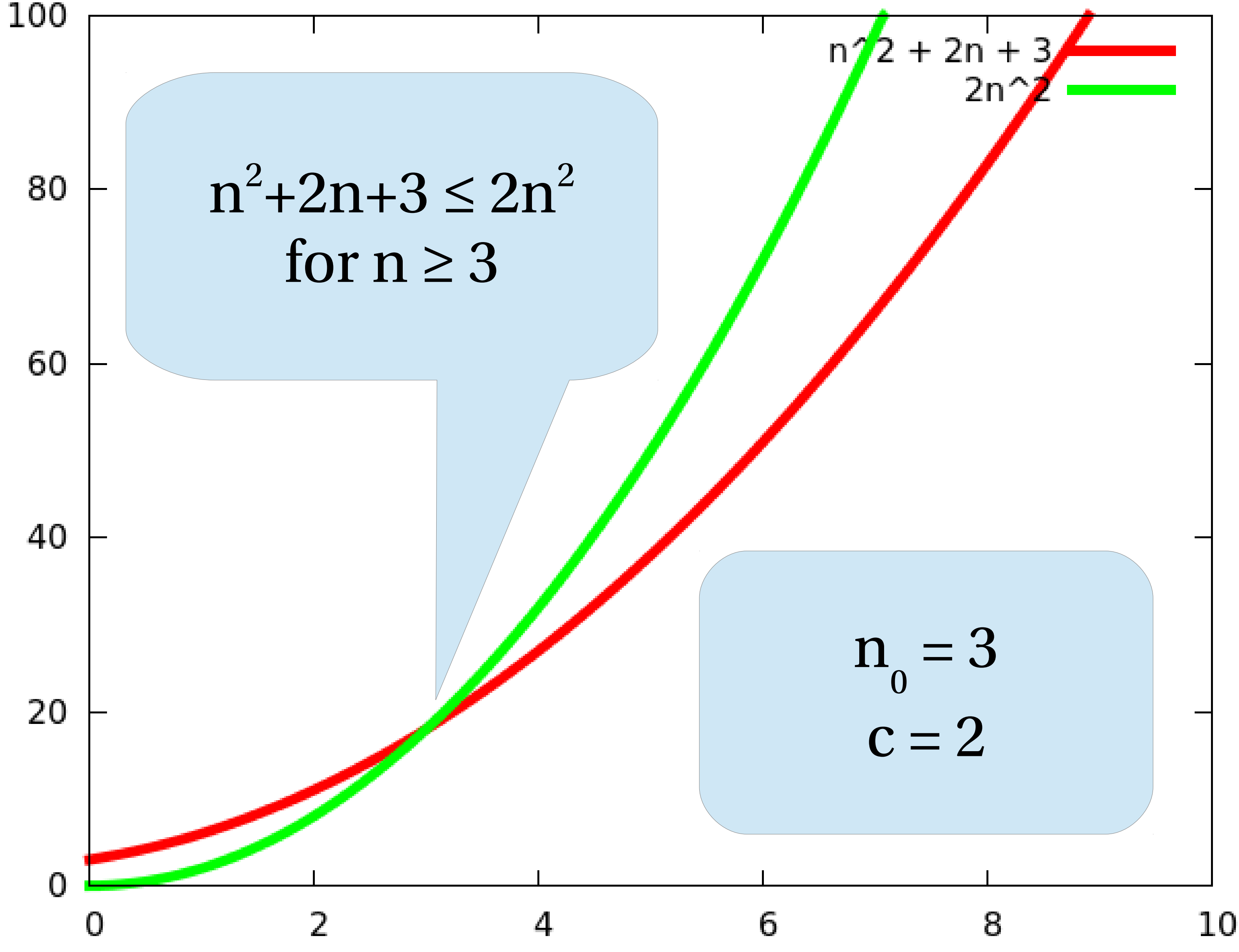
T(n) is O(f(n)) if for sufficiently large n, T(n) is at most proportional to f(n)

- there are two constants n_0 and c
- such that whenever $n \geq n_0$, $T(n) \leq c \times f(n)$

"For sufficiently large n, $T(n) \leq c \times f(n)$"

T(n) is typically "the time that algorithm X takes on an input of size n"

An example: $n^2 + 2n + 3$ is $O(n^2)$



Multiplying Big O

$O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$

$k \times O(f(n)) = O(f(n))$, if k is constant

(Exercise: show that these are true)

- You can drop constant factors when calculating Big O

e.g. $2 \times O(n) = O(2n) = O(n)$

e.g. $O(n^2) \times O(n^3) = O(n^5)$

Big-O	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

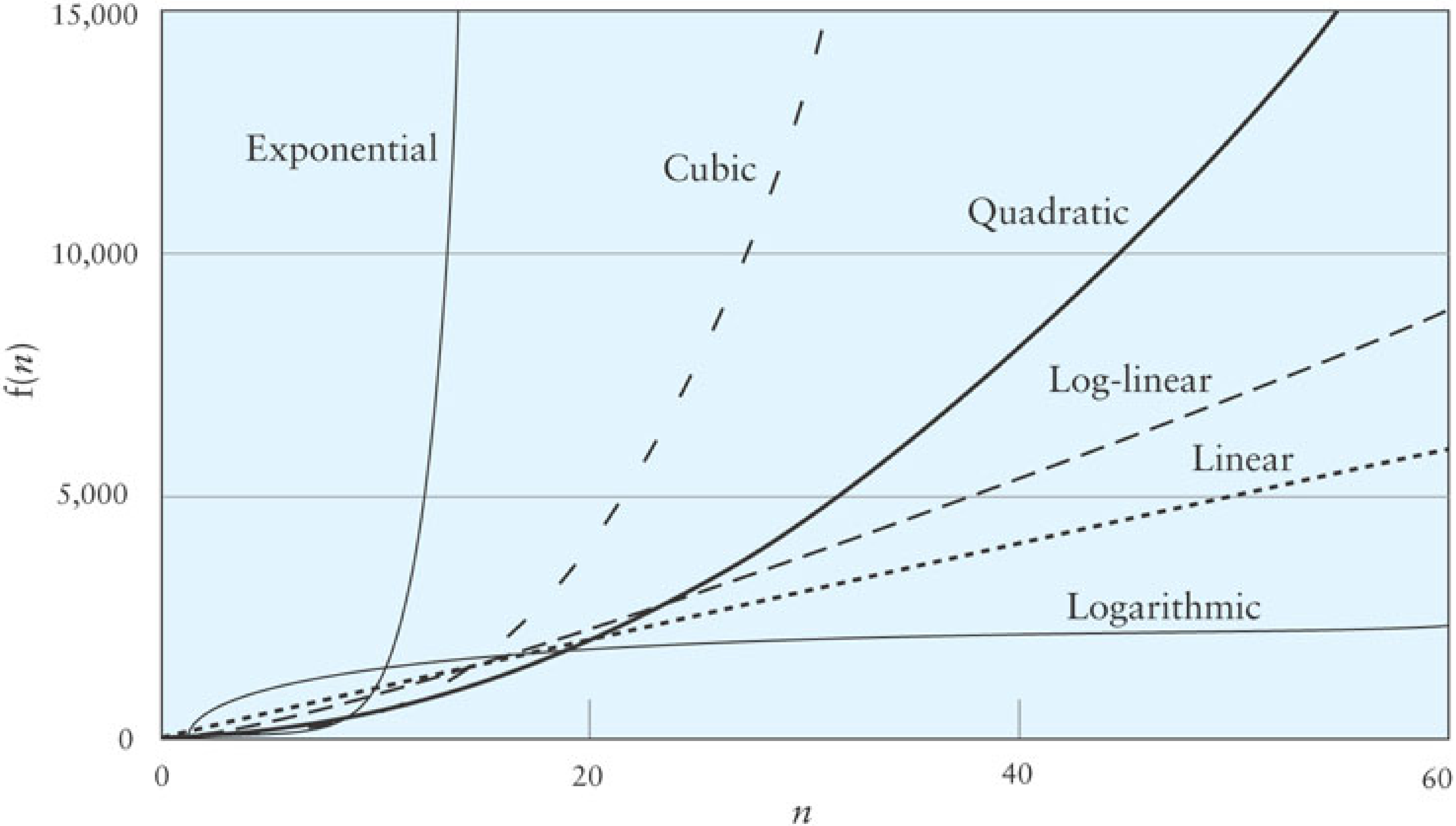
Growth rates

Imagine that we double the input size from n to 2n.

If an algorithm is...

- $O(1)$, then it takes the same time as before
- $O(\log n)$, then it takes a constant amount more
- $O(n)$, then it takes twice as long
- $O(n \log n)$, then it takes twice as long plus a little bit more
- $O(n^2)$, then it takes four times as long

If an algorithm is $O(2^n)$, then adding *one element* makes it take twice as long


15,000
10,000
5,000
0
f(n)
Exponential
Cubic
Quadratic
Log-linear
Linear
Logarithmic
0
20
40
60
n

A hierarchy

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

When adding a term lower in the hierarchy to one higher in the hierarchy, the lower-complexity term disappears:

$O(1) + O(\log n) = O(\log n)$

$O(\log n) + O(n^k) = O(n^k)$ (if $k \geq 0$)

$O(n^j) + O(n^k) = O(n^k)$, if $j \leq k$

$O(n^k) + O(2^n) = O(2^n)$

Hierarchy examples

$O(n) + O(2^n) = 2^n$

$O(n^3) + O(n^2 \log n)$
$= O(n^2) \times O(n + \log n)$
$= O(n^2) \times O(n)$
$= O(n^3)$

The second one uses the multiplication rule $O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$ with $f(n) = n^2$, $g(n) = n + \log n$

Quiz

What are these in Big O notation?

- $n^2 + 11$
- $2n^3 + 3n - 1$
- $n^4 + 2^n$
- $(n^2 + 3)(2^n \times n) + \log_{10} n$

Big O notation is very succinct!

Just use hierarchy and multiplication rules!

$n^2 + 11 = O(n^2) + O(1) = O(n^2)$

$2n^3 + 3n - 1 = O(n^3) + O(n) + O(1) = O(n^3)$

$n^4 + 2^n = O(n^4) + O(2^n) = O(2^n)$

$(n^2 + 3)(2^n \times n) + \log_{10} n =$
$O(n^2) \times O(2^n \times n) + O(\log n) =$
$O(2^n \times n^3) + O(\log n) = O(2^n \times n^3)$

The complexity of a loop

The running time of a loop
is the number of times it runs
times the running time of the body

Or:
If a loop runs m times
and the body takes $O(f(n))$ time
then the loop takes $O(m \times f(n))$

What's the complexity?

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < a.length; j++)
      if (a[i].equals(a[j]) && i != j)
        return false;
  return true;
}
```

What's the complexity?

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < a.length; j++)
      if (a[i].equals(a[j]) && i != j)
        return false;
  return true;
}
```

Body is O(1)

What's the complexity?

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < a.length; j++)
      if (a[i].equals(a[j]) && i != j)
        return false;
  return true;
}
```

Inner loop is O(n)

Body is O(1)

What's the complexity?

Outer loop is $O(n^2)$

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j < a.length; j++)
      if (a[i].equals(a[j]) && i != j)
        return false;
  return true;
}
```

Inner loop is $O(n)$

Body is $O(1)$

What's the complexity?

```
void something(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 1; j < a.length; j *= 2)
      … // something taking O(1) time
}
```

What's the complexity?

```
void something(Object[] a) {
  for(int i = 0; i < a.length; i+
    for (int j = 1; j < a.length; j *= 2)
      … // something taking O(1) time
}
```

Inner loop is O(log n)

What's the complexity?

Outer loop is O(n log n)

Inner loop is O(log n)

```
void something(Object[] a) {
  for(int i = 0; i < a.length; i+
    for (int j = 1; j < a.length; j *= 2)
      … // something taking O(1) time
}
```

What's the complexity?

```
long squareRoot(long n) {
    long i = 0;
    long j = n+1;
    while (i + 1 != j) {
        long k = (i + j) / 2;
        if (k*k <= n) i = k;
        else j = k;
    }
    return i;
}
```

What's the complexity?

```
long squareRoot(long n) {
    long i = 0;
    long j = n+1;
    while (i + 1 != j) {
        long k = (i + j) / 2;
        if (k*k <= n) i = k;
        else j = k;
    }
    return i;
}
```

Each iteration takes O(1) time

What's the complexity?

```
long squareRoot(long n) {
    long i = 0;
    long j = n+1;
    while (i + 1 != j) {
        long k = (i + j) / 2;
        if (k*k <= n) i = k;
        else j = k;
    }
    return i;
}
```

Each iteration takes O(1) time

...and halves j-i, so O(log n)

A downside to Big O

Big O gives an *upper bound* of runtime! Note the "≤" in the formal definition.

Binary search is $O(\log n)$, but it is also $O(n)$, $O(n^2)$, $O(2^n)$, …

When calculating big O, you may occasionally get *too big* answers – then you just have to do the maths by hand (exercise 5.21 in Weiss)

What's the complexity?

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j <= i; j++)
      if (a[i].equals(a[j]))
        return false;
  return true;
}
```

What's the complexity?

```
boolean unique(Object[] a) {
  for(int i = 0; i < a.length; i++)
    for (int j = 0; j <= i; j++)
      if (a[i].equ
        return fa
  return true;
}
```

i ≤ n, so this loop runs
O(n) times,
so O(n^2) in total?

The good news

In this kind of loop...

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

...you can say that the inner loop runs O(n) times, without messing up the answer. The complexity is the same as if we had `j <= n` as the loop condition!

Analysis of `unique`

Outermost loop runs n times

Innermost loop runs i times

Innermost loop takes c×i steps (O(i))

$$\sum_{i=0}^{n-1} c \times i = c\left(\sum_{i=0}^{n-1} i\right) = c\left(\frac{n(n-1)}{2}\right) = O(n^2)$$

So $O(n^2)$ was correct!

Running statements in sequence

What's the complexity of this program?

```
for (int i = 0; i < n; i++) …
for (int i = 1; i < n; i *= 2) …
```

Running statements in sequence

What's the complexity of this program?

```
for (int i = 0; i < n; i++) …
for (int i = 1; i < n; i *= 2) …
```

The first line is O(n), the second is O(log n)

So total is O(n + log n) = O(n)

For statements in sequence, add their complexities!

A bigger example

```
for (int i = 1; i <= n; i *= 2) {
    for (int j = 0; j < n*n; j++)
        for (int k = 0; k <= j; k++)
            …
    for (int j = 0; j < n; j++)
        …
}
```

A bigger example

```
for (int i = 1; i <= n; i *= 2) {
    for (int j = 0; j < n*n; j++)
        for (int k = 0; k <= j; k++)
            …
    for (int j = 0; j < n; j++)
        …
}
```

This loop is O(n)

A bigger example

The `j`-loop runs n^2 times

```
for (int i = 1; i <= n; i *= 2) {
   for (int j = 0; j < n*n; j++)
      for (int k = 0; k <= j; k++)
         …
   for (int j = 0; j < n; j++)
      …
}
```

This loop is O(n)

A bigger example

```
for (int i = 1; i <= n; i *= 2) {
    for (int j = 0; j < n*n; j++)
        for (int k = 0; k <= j; k++)
            …
    for (int j = 0; j < n; j++)
        …
}
```

The `j`-loop runs n^2 times

`k <= j < n*n` so this loop is $O(n^2)$

This loop is $O(n)$

A bigger example

The outer loop runs O(log n) times

The j-loop runs n^2 times

```
for (int i = 1; i <= n; i *= 2) {
    for (int j = 0; j < n*n; j++)
        for (int k = 0; k <= j; k++)
            …
    for (int j = 0; j < n; j++)
        …
}
```

`k <= j < n*n` so this loop is $O(n^2)$

This loop is O(n)

A bigger example

The outer loop runs O(log n) times

The j-loop runs n^2 times

```
for (int i = 1; i <= n; i *= 2) {
    for (int j = 0; j < n*n; j++)
        for (int k = 0; k <= j; k++)
            …
    for (int j = 0; j < n; j++)
        …
}
```

`k <= j < n*n` so this loop is $O(n^2)$

This loop is $O(n)$

Total: $O(\log n) \times (O(n^2) \times O(n^2) + O(n))$
$= O(n^4 \log n)$

Summary

Binary search – an $O(\log n)$ algorithm

Big O complexity and why we use it

Rules for manipulating big O

Finding complexity of an algorithm

See you after Easter!

P.S. Try to read Weiss chapter 5!