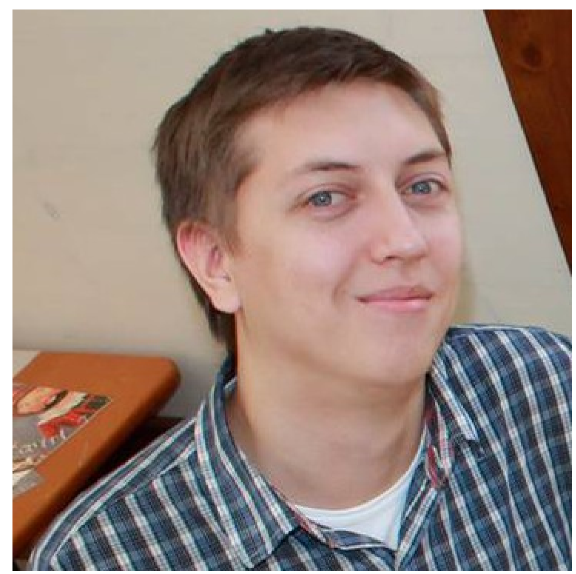
Introduction
Data Structures

This course

Lecturer: Nick Smallbone (me)

- nicsma@chalmers.se, room 5463

Assistant: Evgeny Kotelnikov



All info on the website!

Lectures

Twice a week:

- Wednesday 13-15
- Friday 13-15

Almost always in EL41
(but next Wednesday in EB!)

Labs

Four labs and one hand-in

Do them in pairs if at all possible

Lab supervision:

- Tuesday 13-15
- Tuesday 15-17
- Friday 10-12

All in 3354/3358, starting next Tuesday

Exercises

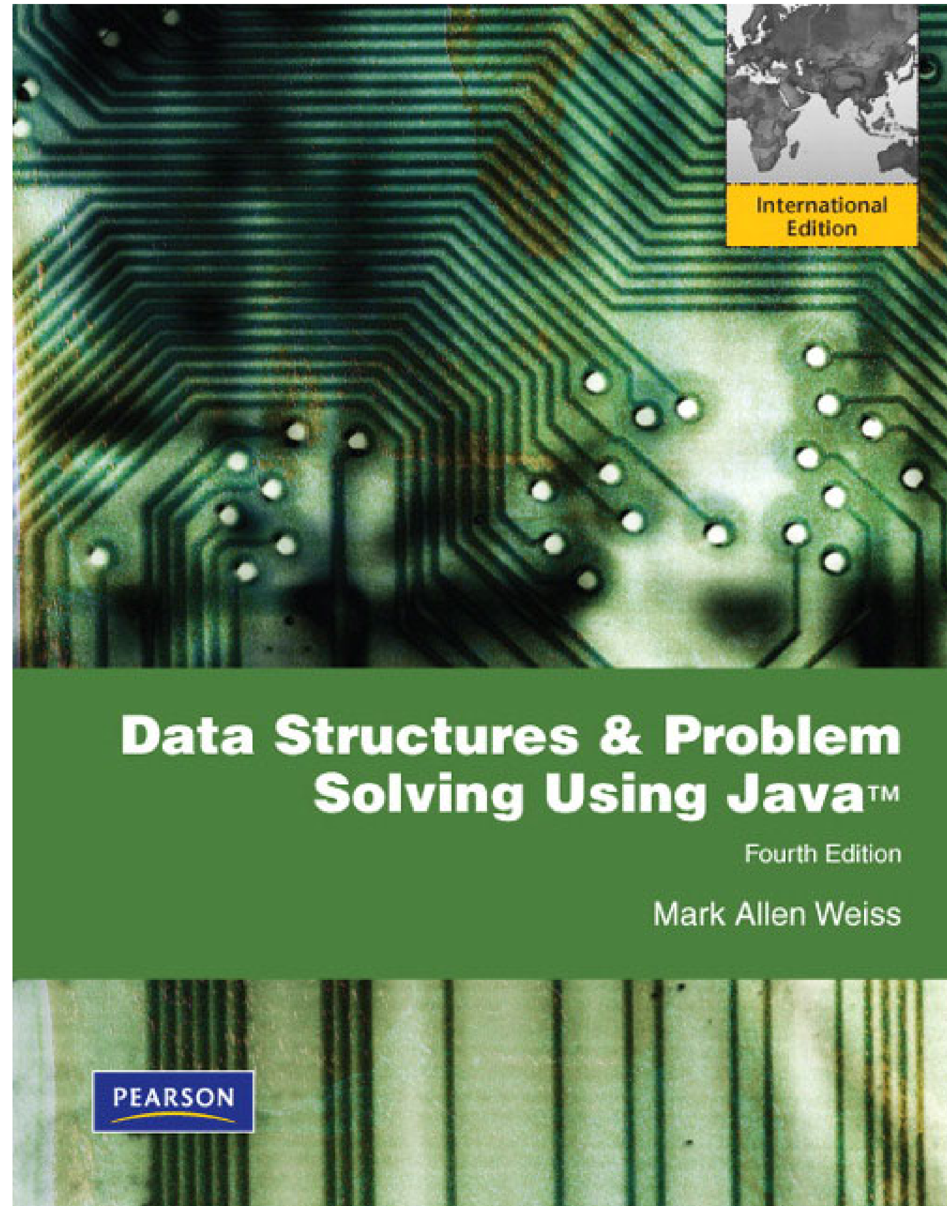
Optional (but helpful) exercises

One set a week – answers go up following week

No formal exercise sessions, but you can ask Evgeny for help at the lab sessions

Course book

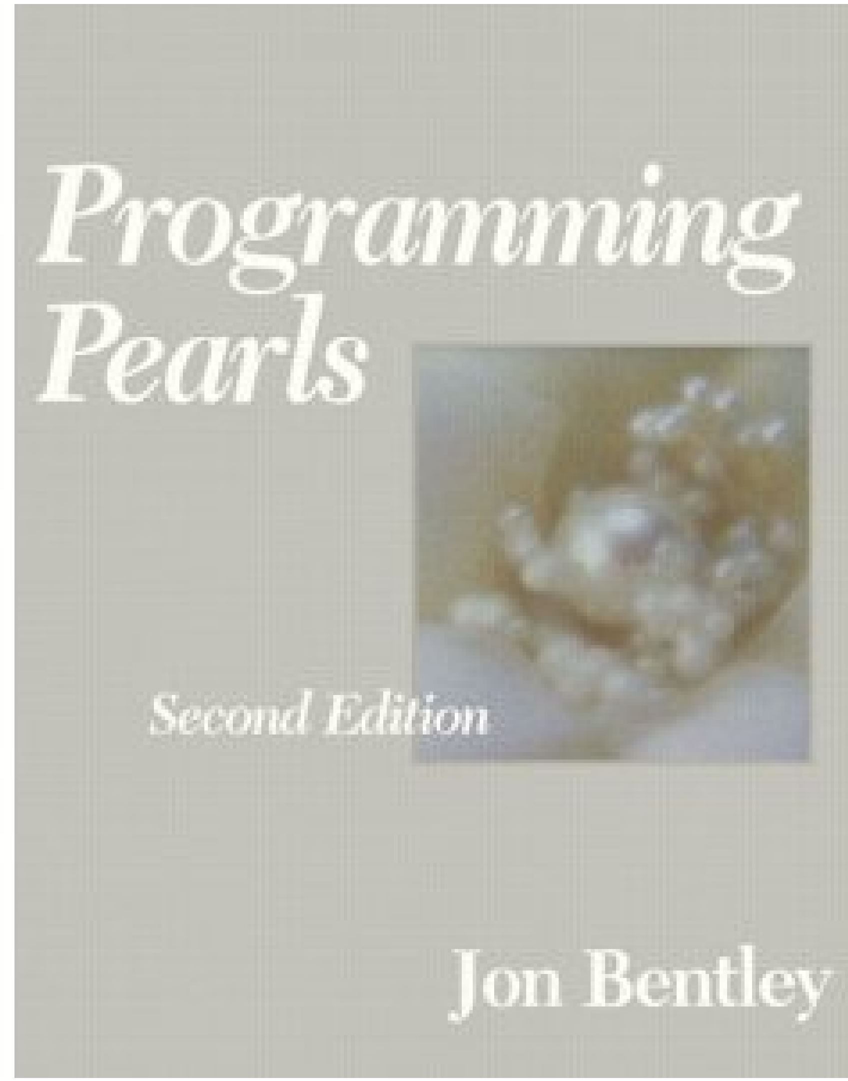
- Mark Weiss: Data Structures and Problem Solving Using Java, 4th ed.
- Order from e.g. Adlibris
- May be able to manage without it



Not the course book

- Jon Bentley: Programming Pearls
- A classic computer science book – imaginative solutions to various programming problems
- Not the course book, but excellent extra reading

  (Also has the advantage of being short and cheap!)



A simple problem

Suppose we want to write a program that reads a file, and then outputs it, twice

Idea: read the file into a string

```
String result = “”;
Character c = readChar();
while(c != null) {
    result += c;
    c = readChar();
}
System.out.print(result);
System.out.print(result);
```

A simple problem

Suppose we want ... e a program that reads a file, ... ts it twice

Idea: re... ng

This program is *amazingly slow*!

```
String res
Charac
while(c !
    resul
    c = read      );
}
System.out.print(result);
System.out.print(result);
```

The right way to solve it?

Use a StringBuilder instead

```
StringBuilder result = new StringBuilder();
Character c = readChar();
while(c != null) {
    result.append(c);
    c = readChar();
}
System.out.print(result);
System.out.print(result);
```

...but: **why is there a difference?**

Behind the scenes

A string is basically an *array of characters*

- String s = "hello" ↔ char[] s = {'h','e','l','l','o'}

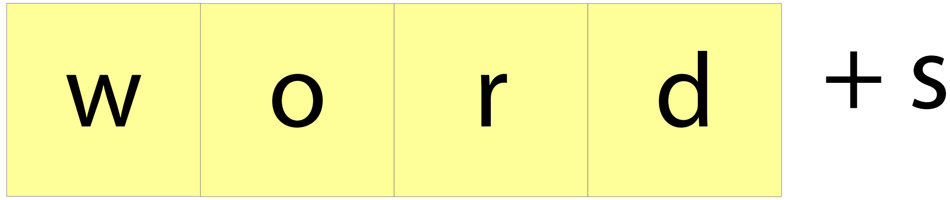
This little line of code…

```
result = result + c;
```

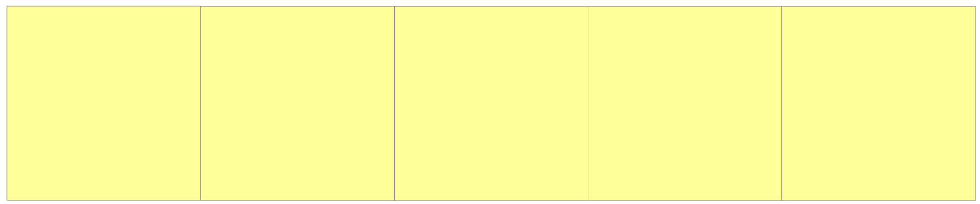
is:

- Creating a new array one character longer than before
- Copying the original string into the array, one character at a time
- Storing the new character at the end

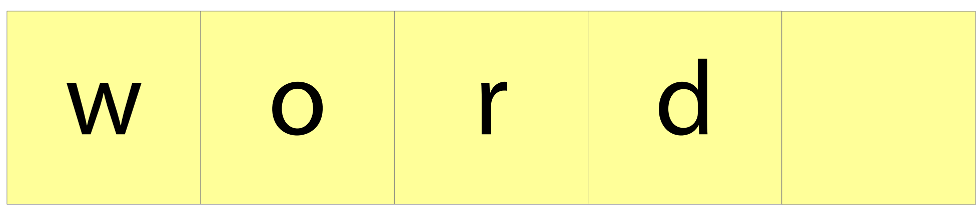
(See CopyNaive.java)



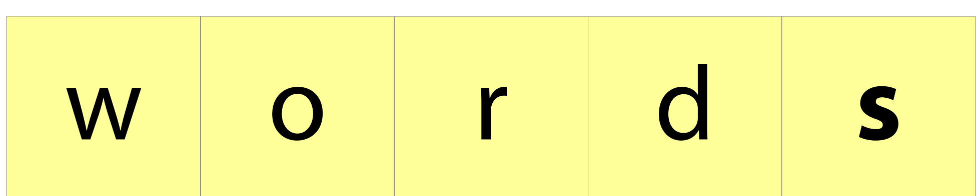
1. Make a new array



2. Copy the old array there



3. Add the new element



Well, is it really so bad?

Appending a single character to an string of length n needs to copy n characters

Imagine we are reading a file of length n

- ...we append a character n times
- ...the string starts off at length 0, finishes at length n
- ...so average length throughout is $n/2$
- total: $n \times n/2 = n^2/2$ characters copied

For “War and Peace”, n = 3600000

so 1800000 × 3600000 = **6,480,000,000,000** characters copied!

No wonder it's slow!

Improving it (take 1)

It's a bit silly to copy the whole array every time we append a character

Idea: add some slack to the array

- Whenever the array gets full, make a new array that's (say) 100 characters bigger
- Then we can add another 99 characters before we need to copy anything!
- Implementation: array+variable giving size of *currently used* part of array

(See Copy100.java)

h	e	l	l	o		w	o	r	l

Add an element:

h	e	l	l	o		w	o	r	l
d									

Add an element:

h	e	l	l	o		w	o	r	l
d	**!**								

Improving it (take 1)

Does this idea help?

We will avoid copying the array 99 appends out of 100

In other words, we will copy the array **1/100th** as often...

...so instead of copying **6,480,000,000,000** characters, we will copy only **64,800,000,000!**
(Oh. That's still not so good.)

Improving it (take 2)

The trick: as the array gets bigger, have more and more slack space

- Whenever the array gets full, **double** its size

So we need to copy the array *less and less often* as it gets bigger

This works – and is what StringBuilder does!

See CopyDouble.java

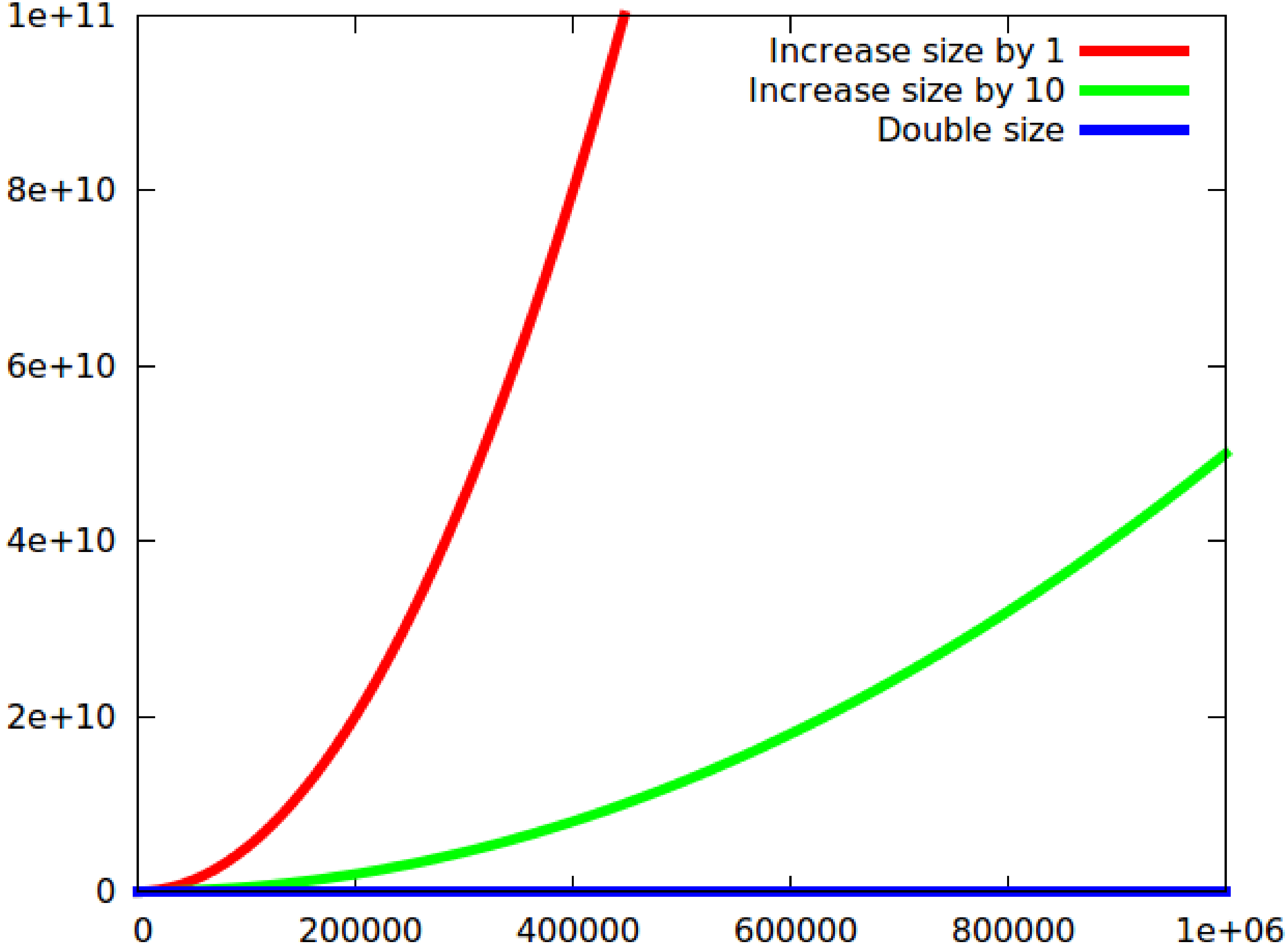
Improving it (take 2)

Why does it work?

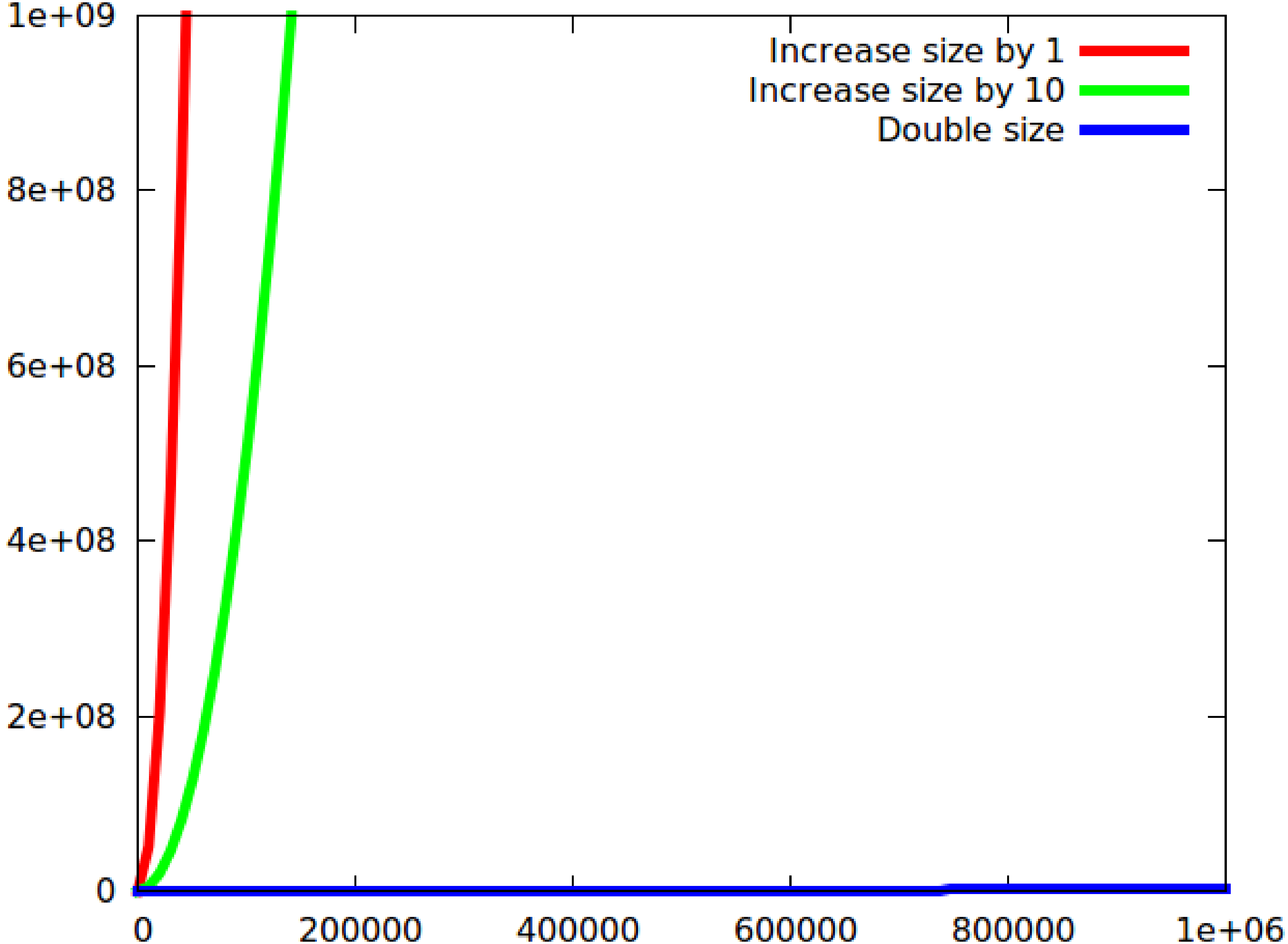
- Imagine the array is currently full, e.g., size 1024, and we append a character
- This means we create a new array of size 2048
- After 1024 appends, the array will be full again and we will have to copy 2048 characters
- In general, if we have just copied *2n* characters, we have previously added *n* characters without copying
- This "averages out" at 2 characters copied per append

For "War and Peace", we copy **~7,200,000** characters. A million times less than the first version!

Performance – a graph

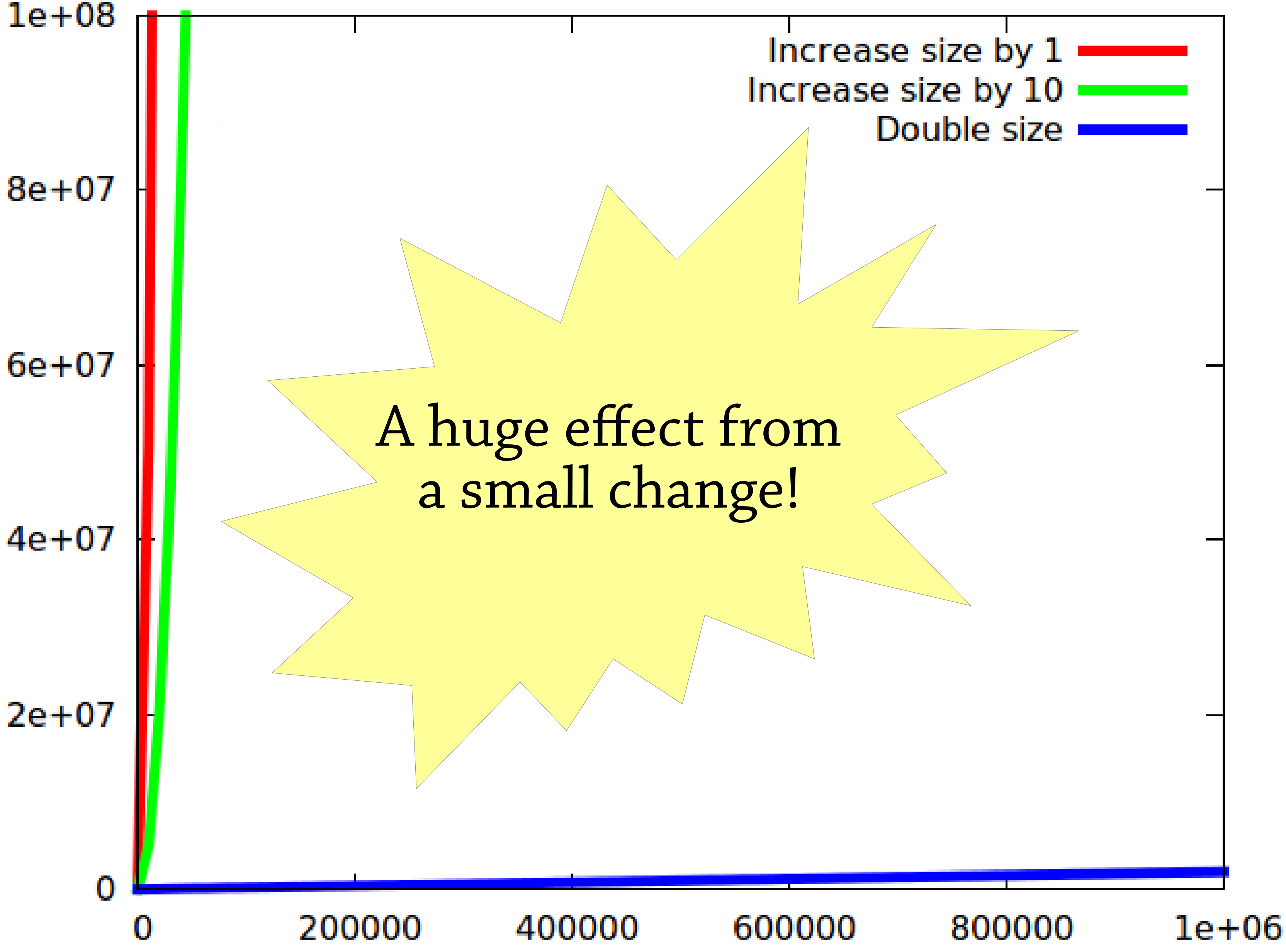


Zoom in!



1e+09
8e+08
6e+08
4e+08
2e+08
0
0
200000
400000
600000
800000
1e+06
Increase size by 1
Increase size by 10
Double size

Zoom in!


1e+08
8e+07
6e+07
4e+07
2e+07
0
0
200000
400000
600000
800000
1e+06
Increase size by 1
Increase size by 10
Double size
A huge effect from
a small change!

Dynamic arrays

A dynamic array is like an array, but can be resized – very useful data structure:

- `E get(int i);`
- `void set(int i, E e);`
- `void add(E e);`

Implementation is just as in our file-reading an example:

- An array
- A variable storing the size of the used part of the array
- `add` copies the array when it gets full, but doubles the size of the array each time

Called `ArrayList` in Java

About strings and StringBuilder

String: array of characters

- Fixed size
- Immutable (can't modify once created)

StringBuilder: *dynamic* array of characters

- Can be resized and modified efficiently

Why can't the String class use a dynamic array?

The moral of the story

It's often tempting to program using "brute force", using just arrays, strings, etc.

But by choosing the right data structure:

- The code becomes simpler (compare `arrayList.add(e)` against our array-copying dance from earlier)
- Hence it's easier to avoid mistakes
- You can get whopping performance improvements!

So what is a data structure anyway?

Vague answer: any way of organising the data in your program

A data structure always supports a particular set of *operations*:

- Arrays: get (`a[i]`), set (`a[i]=x`), create (`new int[10]`)
- Dynamic arrays: same as arrays plus `add`
- Haskell lists: `cons`, `head`, `tail`
- Many, many more…

kittens

kittens
kittens **tumblr**
kittens **playing**
kittens **that look like hitler**
kittens **mittens**
kittens **meowing**
kittens **fighting**
kittens **inspired by kittens youtube**
kittens **im in love zippy**
kittens **game**

Sök på Google | Jag har tur

Prefix tree – return all strings starting with a particular sequence

Interface vs implementation

As a user, you are mostly interested in *what operations* the data structure supports, not how it works

Terminology:

- The set of operations is an *abstract data type (ADT)*
- The data structure *implements* the ADT
- Example: *map* is an ADT which can be implemented by a binary search tree, a 2-3 tree, a hash table, … (we will come across all these later)

Interface vs implementation

Why study how data structures work inside? Can't we just use them?

- As computer scientists, you ought to understand how things work inside
- Sometimes you need to *adapt* an existing data structure, which you can only do if you understand it
- The best way to learn how to *design your own* data structures is to study lots of existing ones

This course

- *How to design* data structures
  - Lectures and exercises
- *How to reason* about their performance
  - Lectures, exercises, hand-in
- *How to use them* and pick the right one
  - Labs and exercises

Binary search

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*.

You might have to look at every element before you find the right one.

Searching

But what if the array is sorted?

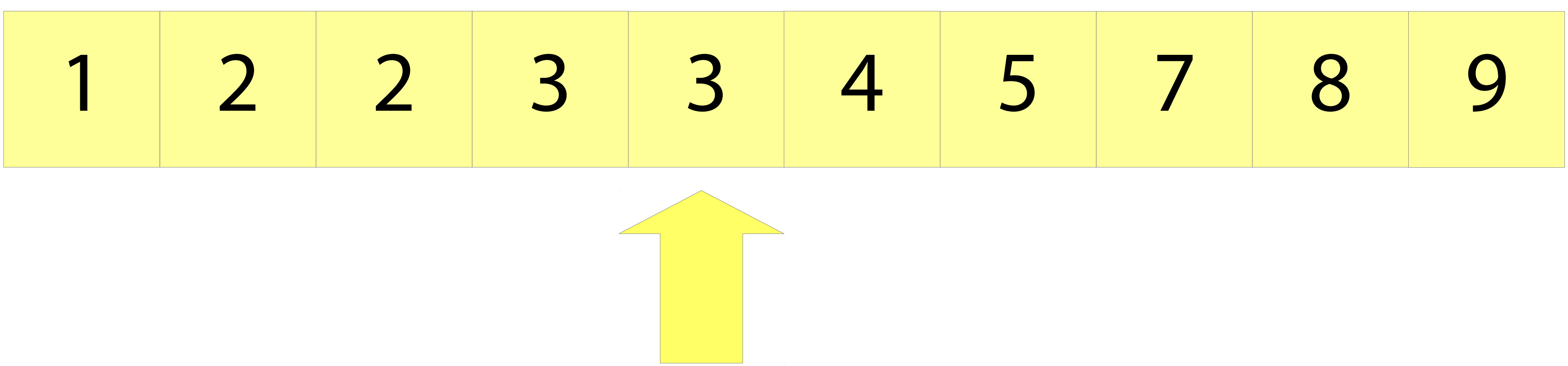
1	2	2	3	3	4	5	7	8	9

Then we can use *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.



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.

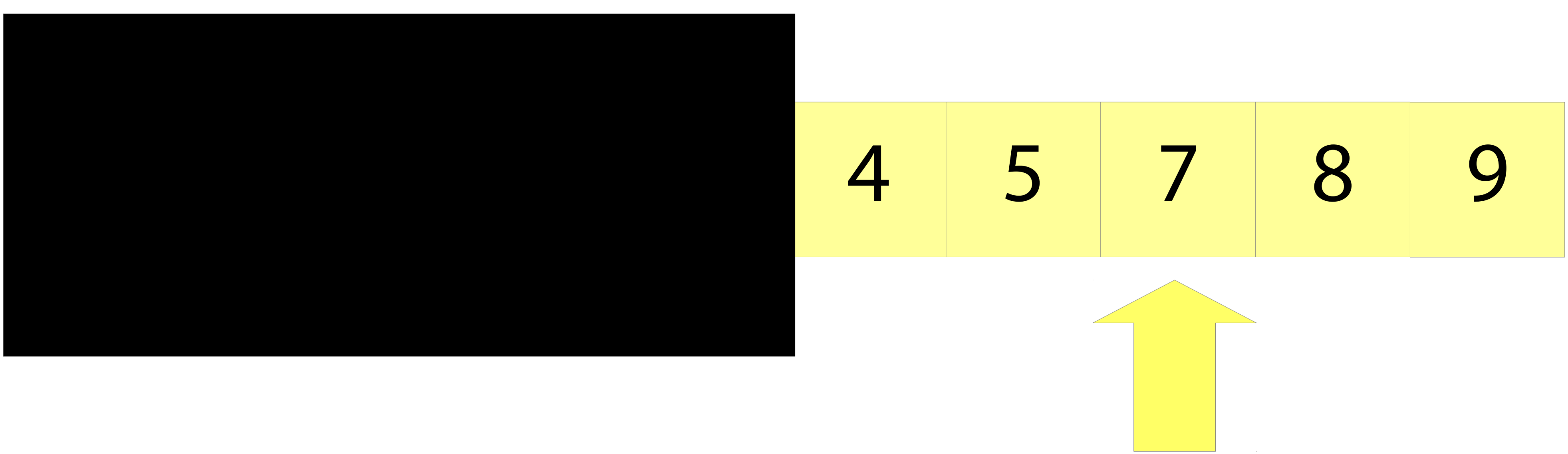
[redacted]

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.

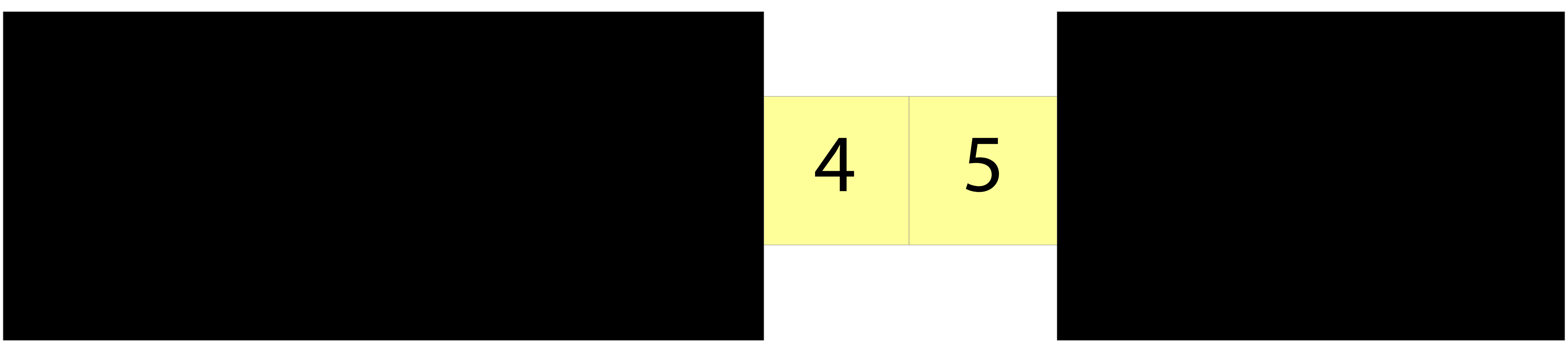


Binary search

7 is greater than 4.

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

We can ignore everything after 7.

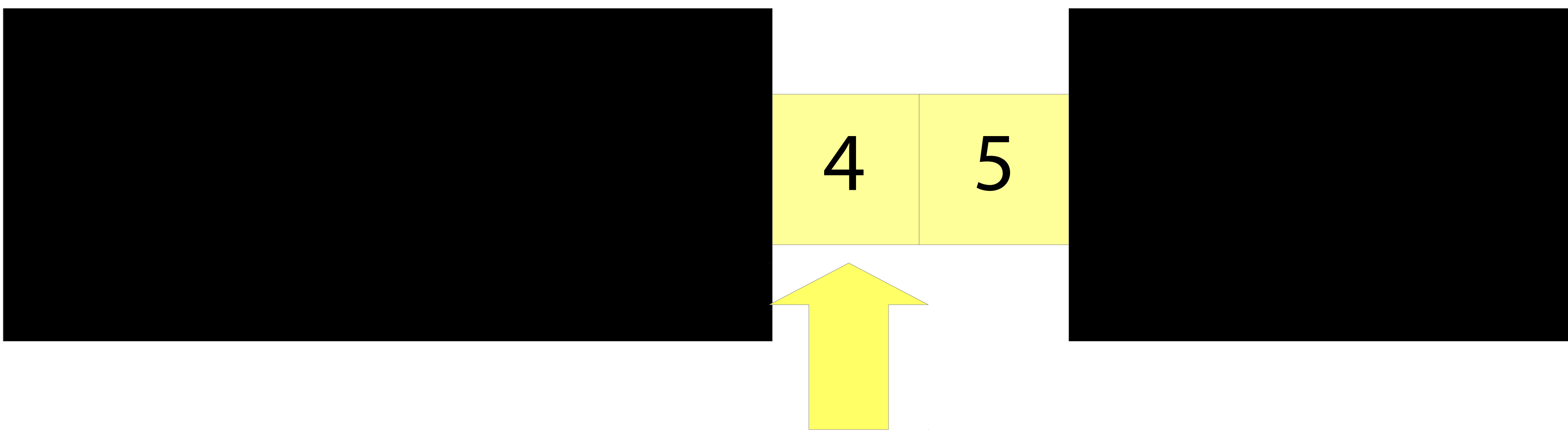


Binary search

We repeat the process.

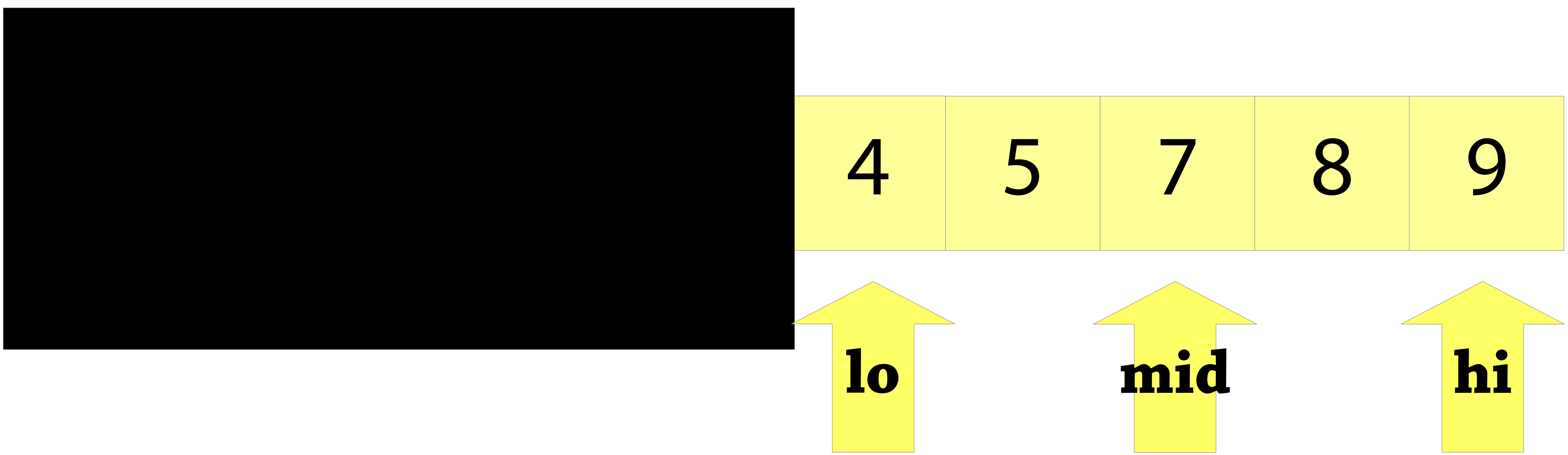
We look half way along the array again.

We find 4!



Implementing binary search

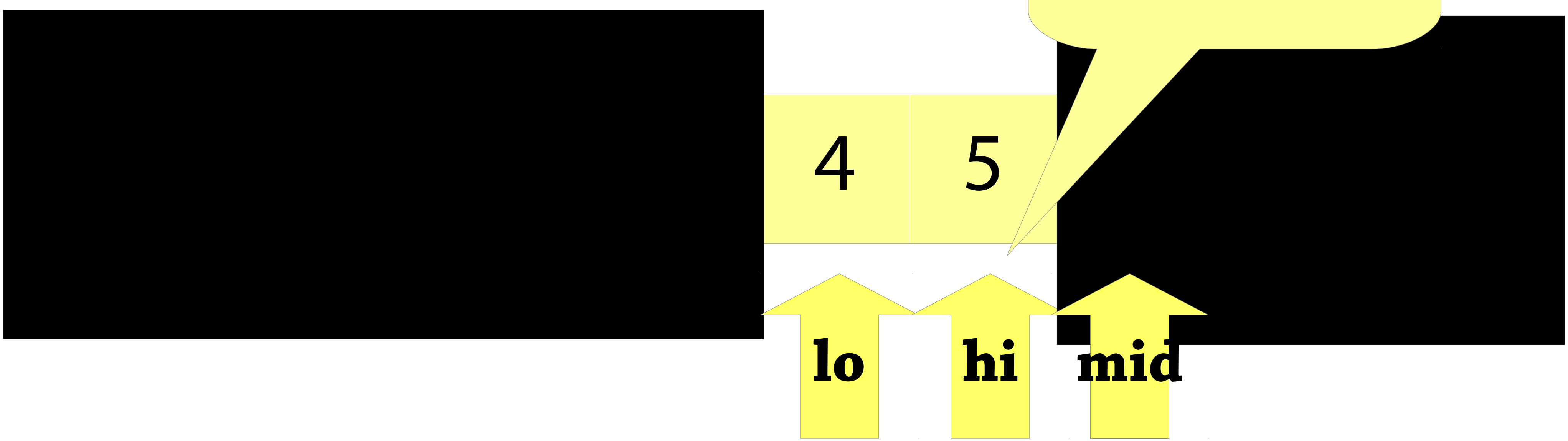
Keep two indices `lo` and `hi`. They represent the part of the array to search.



Let `mid = (lo + hi) / 2` and look at `a[mid]` – then either set `lo = mid+1`, or `hi = mid-1`, depending on the value of `a[mid]`

Implementing binary search

Keep two indices lo and hi. T t
the part of the array to search



Let mid = (lo + hi) / 2 and look at a[mid] – then either set lo = mid+1, or hi = mid-1, depending on the value of a[mid]

Performance of binary search

In binary search, we repeatedly:

- Look at one element
- Then halve the part of the array we have to search

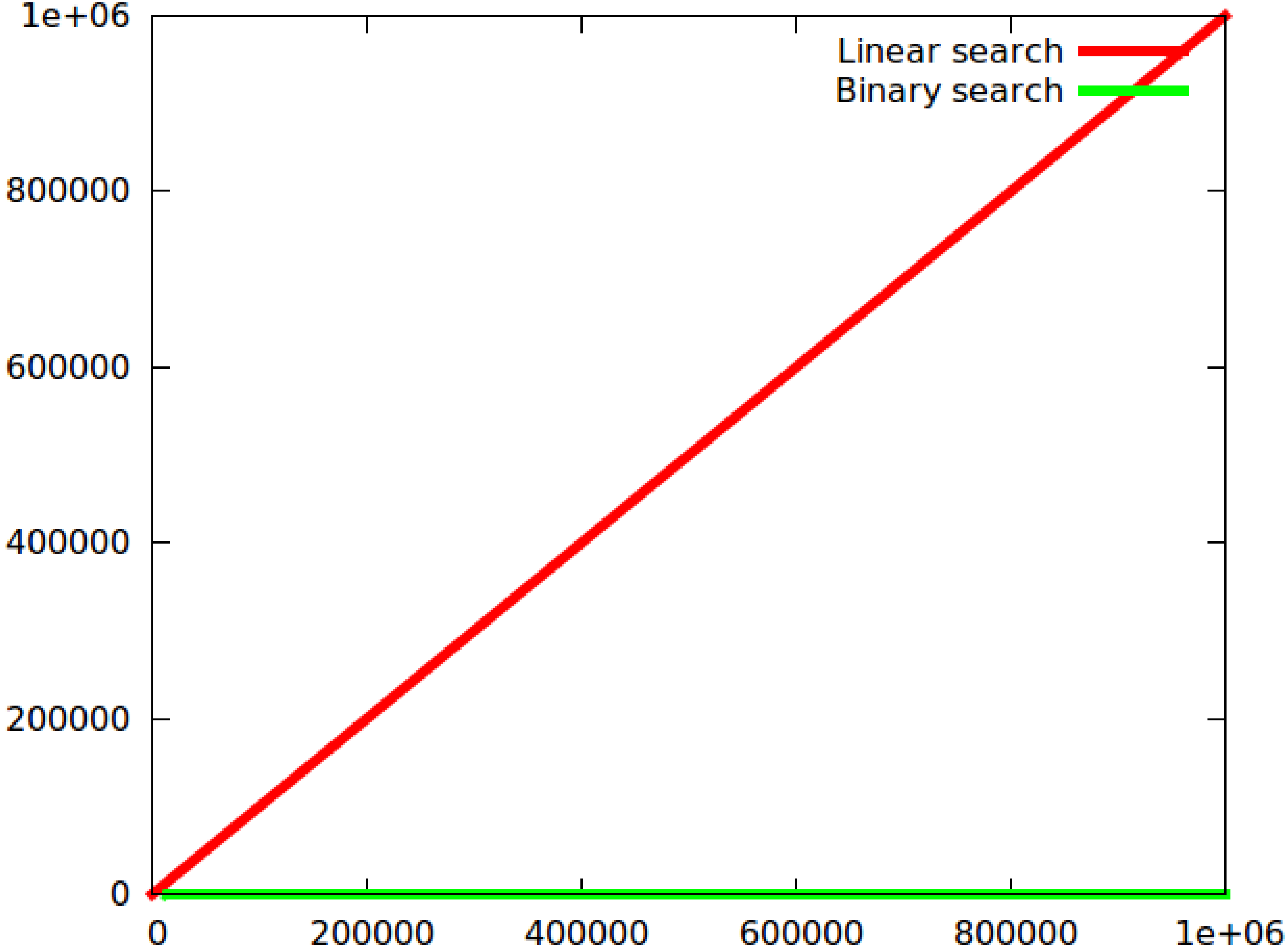
With an array of size 2^n, after n tries, we are down to 1 element

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

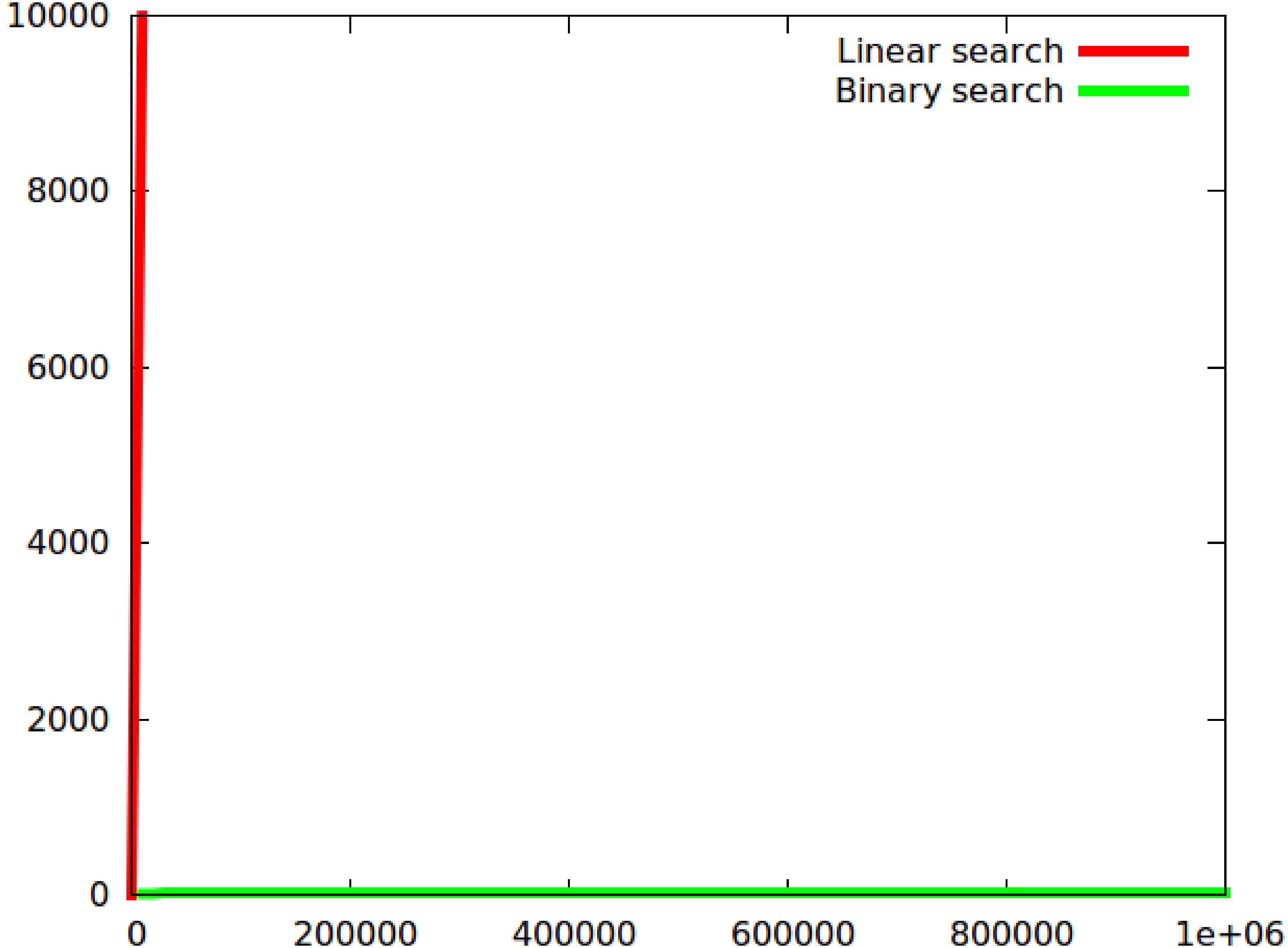
$\log_2 1000000000$ is about 30: 30 tries are enough to find any item in a sorted array of a billion elements

(compared to a billion tries for linear search!)

Performance – a graph



Zoom in!



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

Big points

Using data structures correctly *simplifies* your program and makes it *faster*

- Simpler: by using appropriate operations, e.g., "add element" (dynamic array) instead of "create new array, copy old array to new array, store element in new array" (plain array)
- Faster: the data structure can do whatever tricks are needed to make the operations it provides fast (e.g. dynamic arrays – doubling size)

Most data structures are based on some simple idea

- Dynamic arrays: keep some slack in the array
- Binary search: halve the search space every time

We can use maths to *predict* the performance of our algorithms (more of this next time)

Reading for today: Weiss 2.4.2-2.4.3 (dynamic arrays), 5.6 (binary search)