Introduction, dynamic arrays

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 w wan te program that

reads a file, t, twice

Idea: rea g

```
String
Characte
while(c !=
    resu
    c = read  r( );
}
System.out.print(result);
System.out.print(result);
```

This program is *amazingly slow*!

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`)

w	o	r	d	+ s

1. Make a new array

2. Copy the old array there

w	o	r	d	

3. Add the new element

w	o	r	d	**s**

Well, is it really so bad?

Appending a single character to an string of length *i* needs to copy *i* 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 = 3200000

so 1600000 × 3200000 = **5,120,000,000,000** 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 **5,120,000,000,000** characters, we will copy only **51,200,000,000!**
(Oh. That's still not so good.)

Improving it (take 2)

Another idea: whenever the array gets full, **double** its size

That way, we need to copy the array *less and less often* as it gets bigger

Does this work?

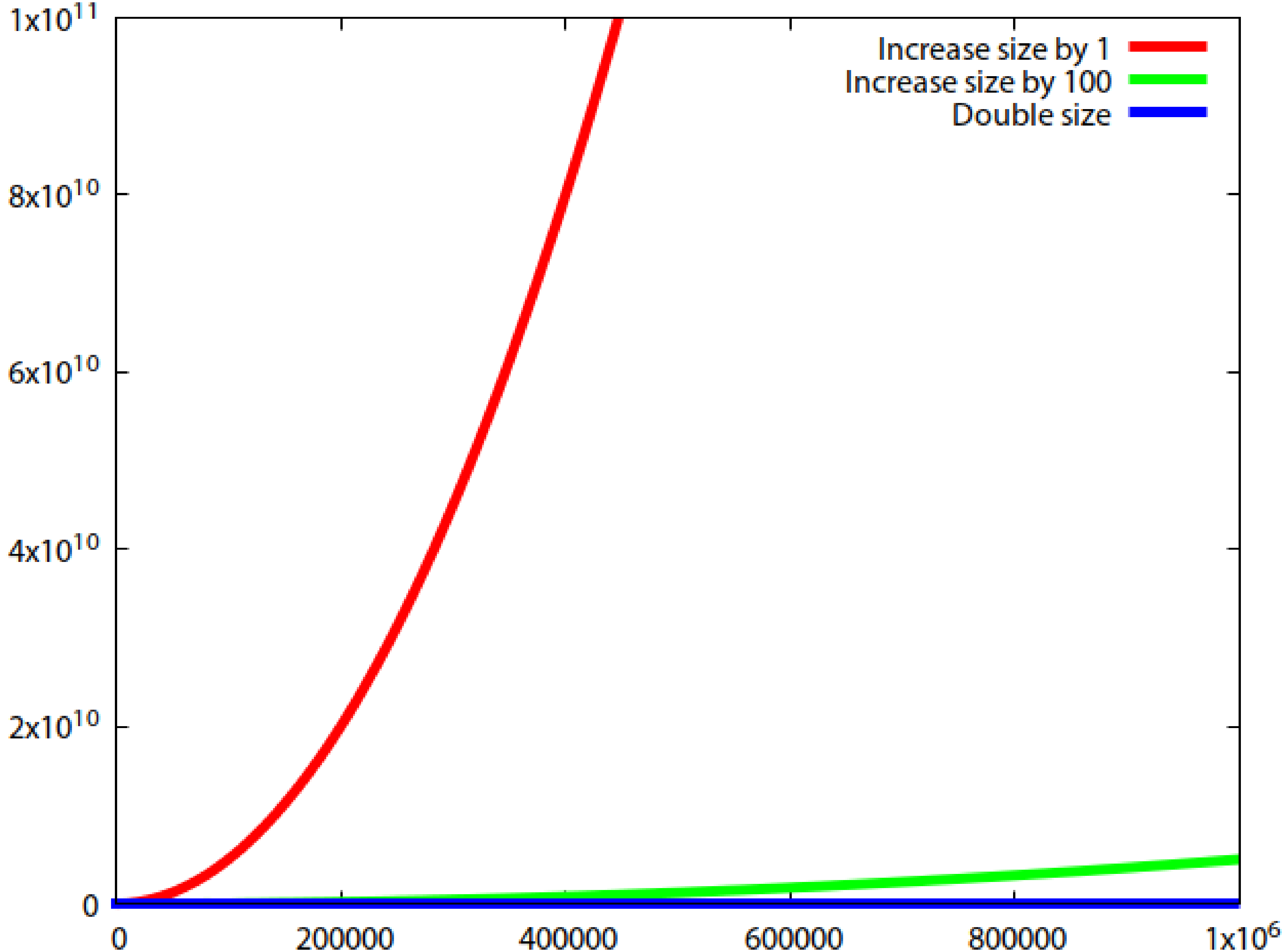
Improving it (take 2)

Let's calculate *how many characters are copied per character appended to the string*

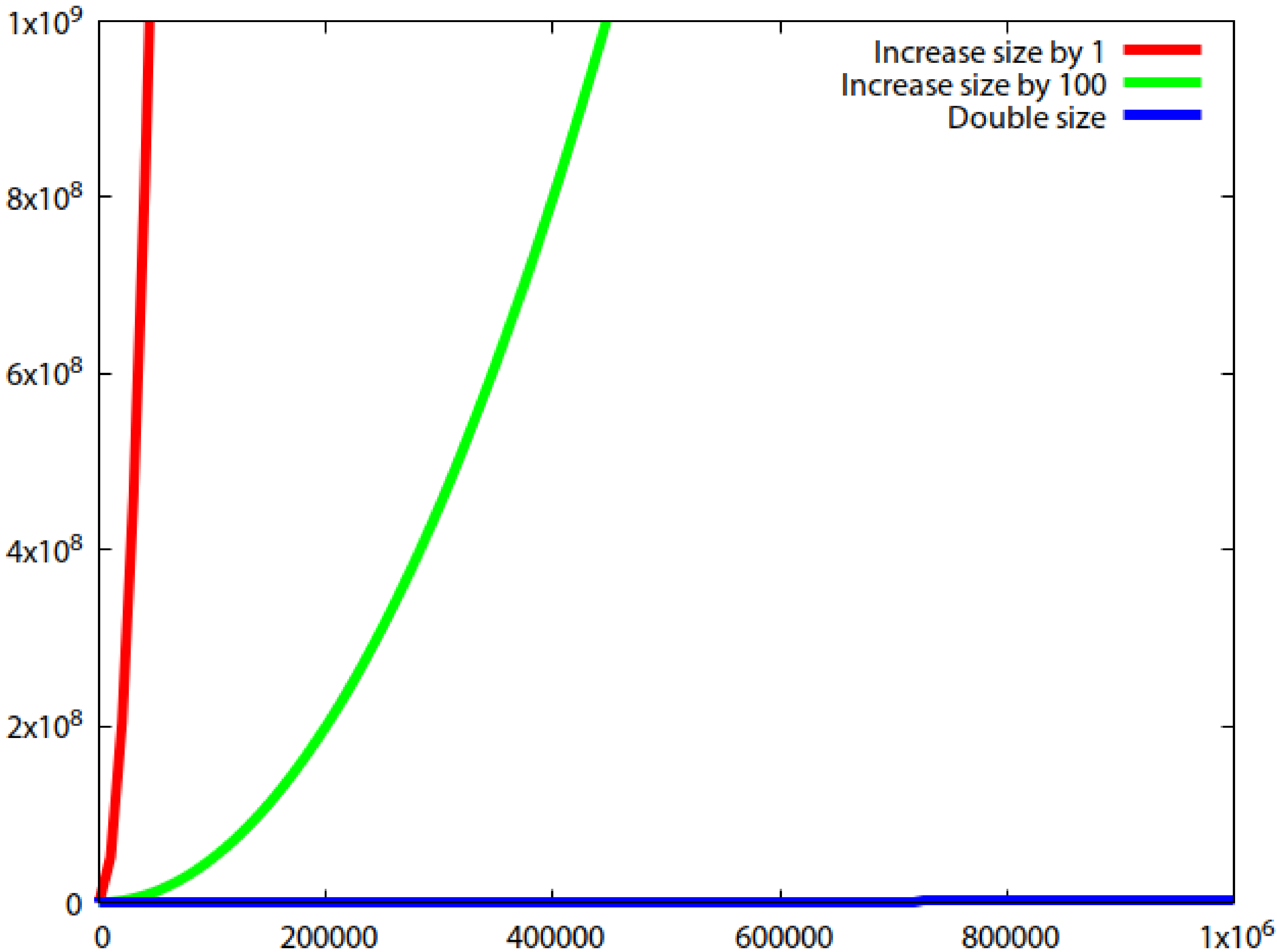
- Imagine we have just expanded the array
- It must have size *2n* and contain *n+1* characters
- The next *n-1* appends don't copy anything
- The next append after that copies *2n* characters
- *n* characters appended, *2n* characters copied: average of 2 characters copied per append

For “War and Peace”, we copy **~6,400,000** characters. A million times less than the first version!

Performance – a graph

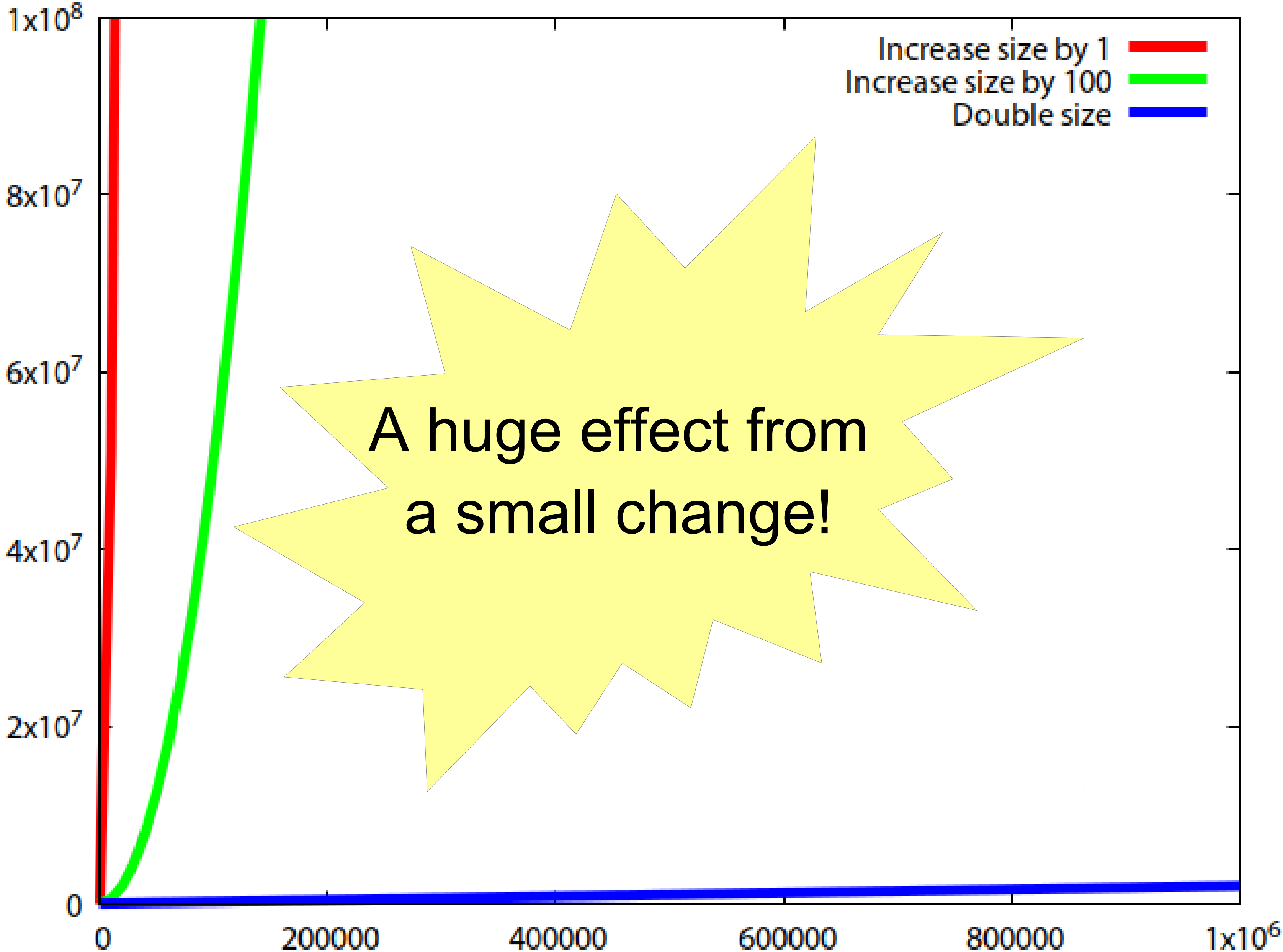


Zoom in!



Increase size by 1
Increase size by 100
Double size
$1x10^9$
$8x10^8$
$6x10^8$
$4x10^8$
$2x10^8$
0
0
200000
400000
600000
800000
$1x10^6$

Zoom in!



1x10^8
8x10^7
6x10^7
4x10^7
2x10^7
0
0
200000
400000
600000
800000
1x10^6
Increase size by 1
Increase size by 100
Double size
A huge effect from a small change!

Why does it work really?

The important property:

- After resizing the array, the new array is no more than half full
- For every “expensive” step of copying *2n* characters, there are *n* “cheap” steps with no copying => constant cost of 2 characters copied per step

Also works if we e.g. increase array size by 50% instead of doubling!

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 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

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/remove
- Haskell lists: cons, head, tail
- Many, many more...

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
- In order to choose the most suitable existing implementation of an ADT you need to known how they work to some extent.
- 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 and implement* data structures
- *How to reason* about them
- *How to use them* and pick the right one

Big points

“Brute force” programming works up to a point

- After that you need to *think*!
- Using the right data structures makes your program *simpler* and *faster*

Most data structures are based on some simple idea

Reasoning helps to get things right

- Dynamic arrays work because the array is always half empty after resizing