CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Data structures

Introduction and dynamic arrays

Dr. Alex Gerdes

DIT961 - VT 2018

Introduction

- Teaching team:
  - Alex Gerdes, lecturer, examiner and course responsible
    - alexg@chalmers.se, room EDIT 6479
  - Sarosh Nasir, course assistant
  - Adi Hrustic, course assistant
- All information on the course website!
- Student representatives! Any takers?

Learning outcomes

- Explain some basic abstract data types and data structures, including lists, queues, tables, trees, and graphs
- Explain some of the algorithms used to manipulate and query these data structures in an efficient way, and explain why they are correct
- Apply basic abstract data types and data structures and algorithms related to these
- Implement and use abstract data types as interfaces and data structures as classes in an object-oriented programming language
- Implement and use abstract data types in a functional programming language
- Make informed choices between different data structures and algorithms for different applications
- Analyse the efficiency of some algorithms

Course organisation

- 9 weeks in total
- Two lectures per week, which cover the theoretical part
- One exercise session per week, where you can get personal help to understand the material; these sessions are not obligatory, but recommended
- 2 to 3 lab sessions per week, where you can get help from the course assistants
- one hand-in and 3 lab assignments, which all need to be completed in order to complete the course
- One written exam at the end of the course; this is obligatory and done individually
- Your final grade will be determined by your grade on the written exam only
- *Loaded with holidays -> irregular schedule -> check TimeEdit*

Course website

- The course website contains updated and relevant information:
  - Latests news (also announced via a Google-group, subscribe!)
  - Schedule
  - Slides (contains last year’s slides now, will be updated after/just before lecture)
  - Lab assignments
  - Exercises
  - …
- Check it regularly!

Labs

- One hand-in and three labs, description on website
- Do them in pairs, search suitable partner
- Lab supervision in 3354/3358, check TimeEdit
- Submit files via Fire (link on website)
- The deadlines are on the website and Fire
- Electronic queueing system:

  http://www.waglys.com/is5Zbj

Exercises

- Optional (but helpful) exercises
- One set a week - answers go up following week
- One exercise session per week, teacher is (often) present

Ask questions!

Reflect!

It is going to be fun!!!

A simple problem

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);
```

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;
```

See `CopyNaive.java`

is:

1. Creating a new array one character longer than before
2. Copying the original string into the array, one character at a time
3. Storing the new character at the end

Behind the scenes

1. Make new array

2. Copy the old array

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* = 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
  - en 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`

Improving it (take 1)

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)

- 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!

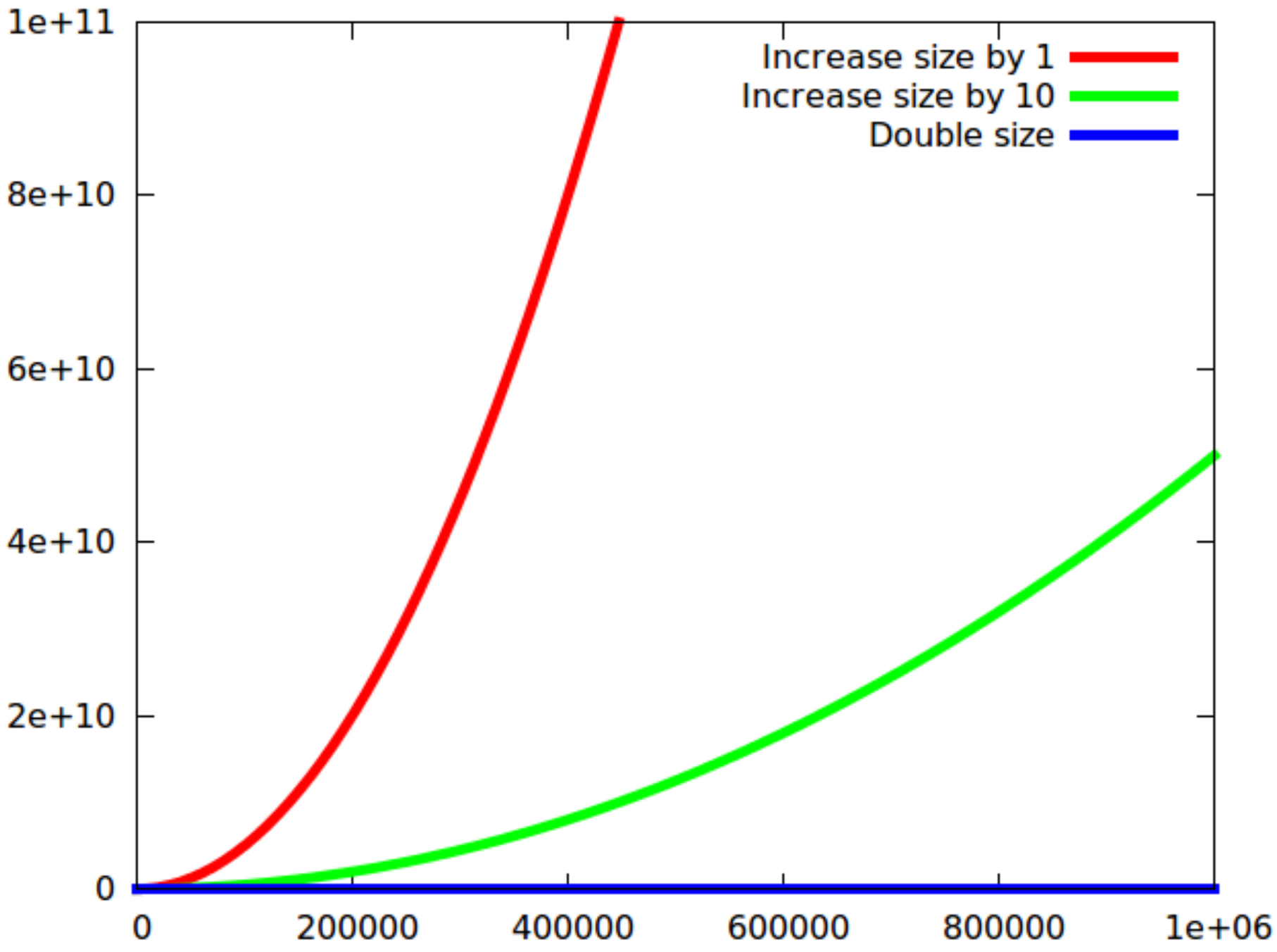
See `CopyDouble.java`

Improving it (take 2)

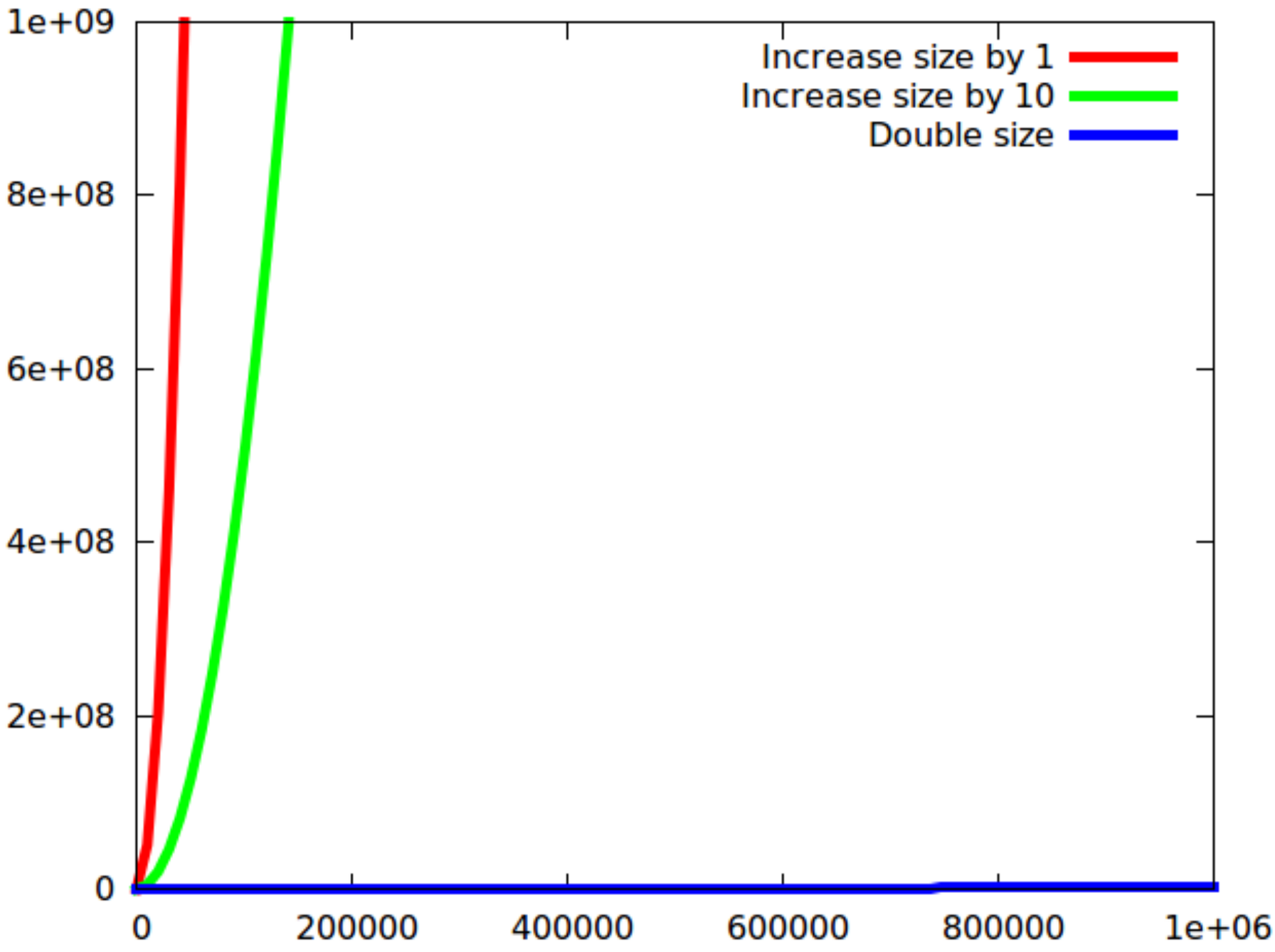
- Why does it work?
  - Imagine the array is currently full, e.g., size 1024, and we append a character
  - This means we create a new array of size 2048
  - After 1024 appends, the array will be full again and we will have to copy 2048 characters
  - In general, if we have just copied 2*n* characters,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

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

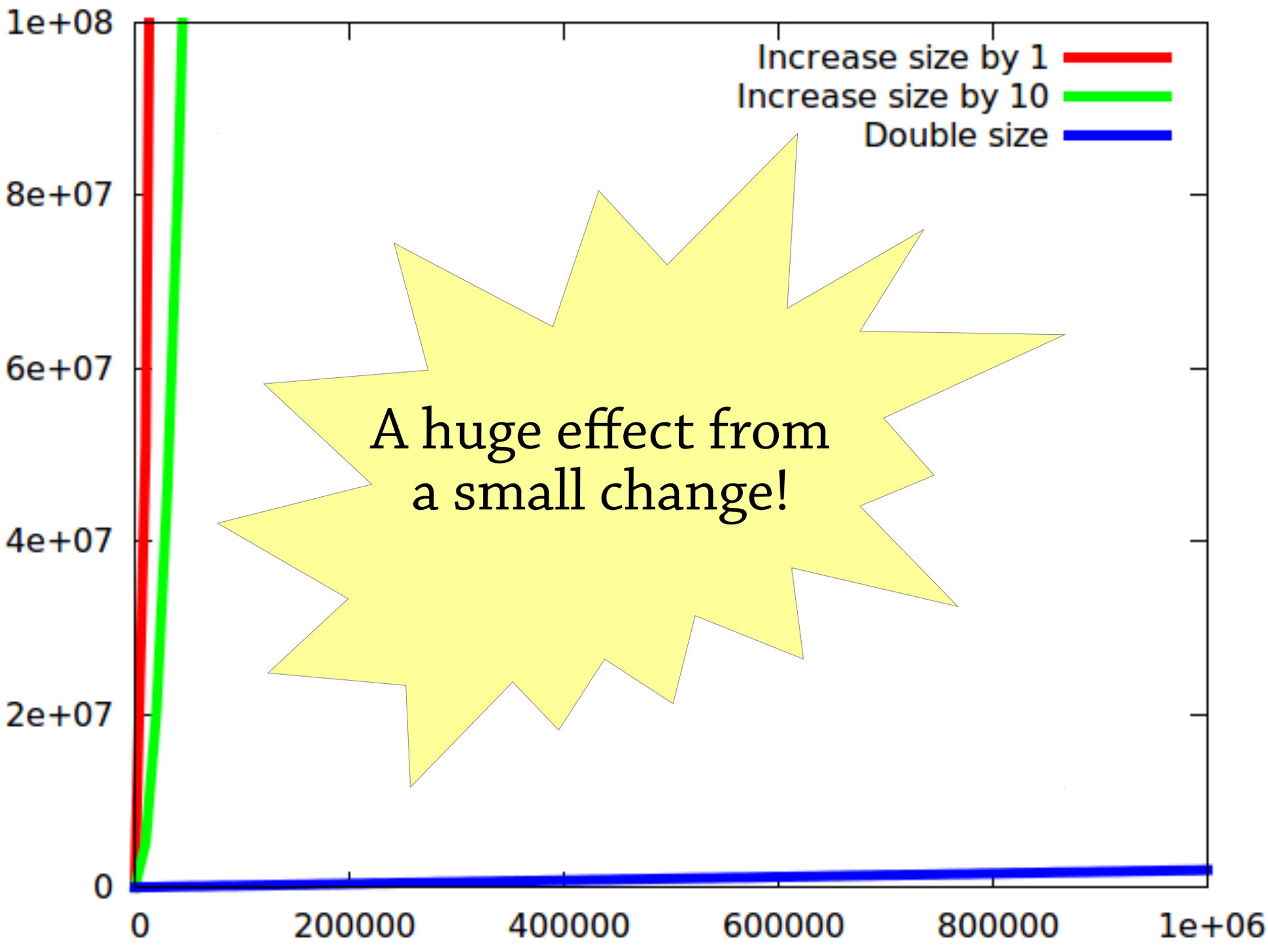
Performance - a graph



Zoom in!


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 `String` 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

- (is there a tiny catch?)

The moral of the story

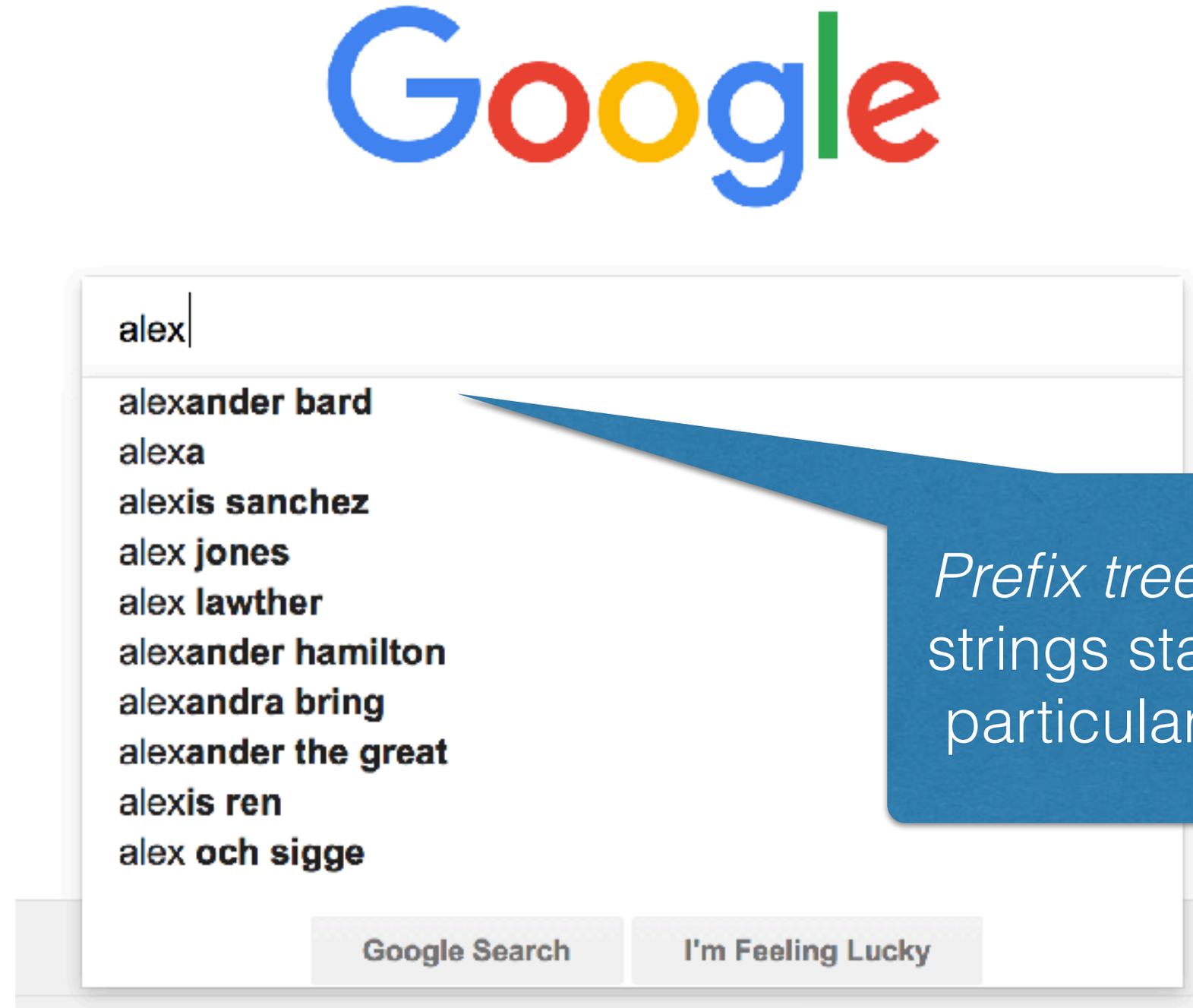
- It's often tempting to program using “brute force”, using just arrays, strings, etc.

- 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...

Real life applications



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* 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

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