DIT181: Data Structures and Algorithms

Lecture 2: Algorithms and complexity

Michał Pałka, Gül Calikli

Email: michal.palka@chalmers.se, calikli@chalmers.se

Students that took LAD (DIT725)

- If you took Logic, Algorithms and Data structures
- If you passed the assignments
- **Then**
  - You only need to pass the exam of DIT181
  - Register as an ‘observer’ to the course
  - Register for the exam (when it’s open)
  - Do not join the group

This lecture

- Algorithms
- Complexity
- Big-O notation

Algorithms

Maximum of a sequence of integers

```
private static int maximum (int[] array) {
   if (array.length == 0)
     throw new IllegalArgumentException
                ("maximum requires a non-empty array");
   int cur = array[0];
   for (int i = 1; i < array.length; ++i) {
     cur = Math.max(cur, array[i]);
   }
   return cur;
 }
```

Maximum of a sequence of integers

Empty sequences do not have a maximum element

```
private static int maximum (int[] array) {
    if (array.length == 0)
      throw new IllegalArgumentException
                  ("maximum requires a non-empty array");
    int cur = array[0];
    for (int i = 1; i < array.length; ++i) {
      cur = Math.max(cur, array[i]);
    }
    return cur;
  }
```

Maximum of a sequence of integers

```
private static int maximum (int[] array) {
   if (array.length == 0)
     throw new IllegalArgumentException
                ("maximum requires a non-empty array");
   int cur = array[0];
   for (int i = 1; i < array.length; ++i) {
     cur = Math.max(cur, array[i]);
   }
   return cur;
 }
```

The largest element so far is initialised as the first one. Indices are zero-based.

Compute the maximum of the largest element so far and element i, and assign it to the variable holding the largest element so far

Maximum of a sequence of integers (Python)

‘Lists’ are actually arrays in Python

```
def maximum(list):
    if len(list) == 0:
        raise ValueError
                ('maximum() requires a non-empty list')
    cur = list[0]
    for x in list[1:]:
        cur = max(cur, x)
    return cur
```

Maximum of a sequence of integers (Go)

```
func maximum(array []int) int {
    if len(array) == 0 {
        panic("maximum() requires a non-empty array")
    }
    var cur = array[0]
    for i := 1; i < len(array); i++ {
        cur = max(cur, array[i])
    }
    return cur
}
```

Maximum of a sequence of integers

- All the above programs implement finding the maximum element of a sequence
- All the programs are very **similar** to each other (language-specific details)
- All of them perform similarly
- All of them implement the same **algorithm**

Algorithms

Informally, an ***algorithm*** *is any well-defined computational procedure that takes some value, or set of values, as* ***input*** *and produces some value, or set of values, as* ***outpu****t. An algorithm is thus a sequence of computational steps that transform the input into the output.*

–*Introduction to Algorithms*, T. H. Cormen et. al.

- Description of an algorithm omits some **programming-language-specific details** (they are unimportant)
- We can **reason** about some **properties** of an algorithm regardless of the language it will be implemented in

This course

- In this course we will use Java
- A lot of algorithm implementations will be very similar in Java and in other languages (Python, Go, etc)
- The knowledge of algorithms is transferable to other languages

An analysis problem

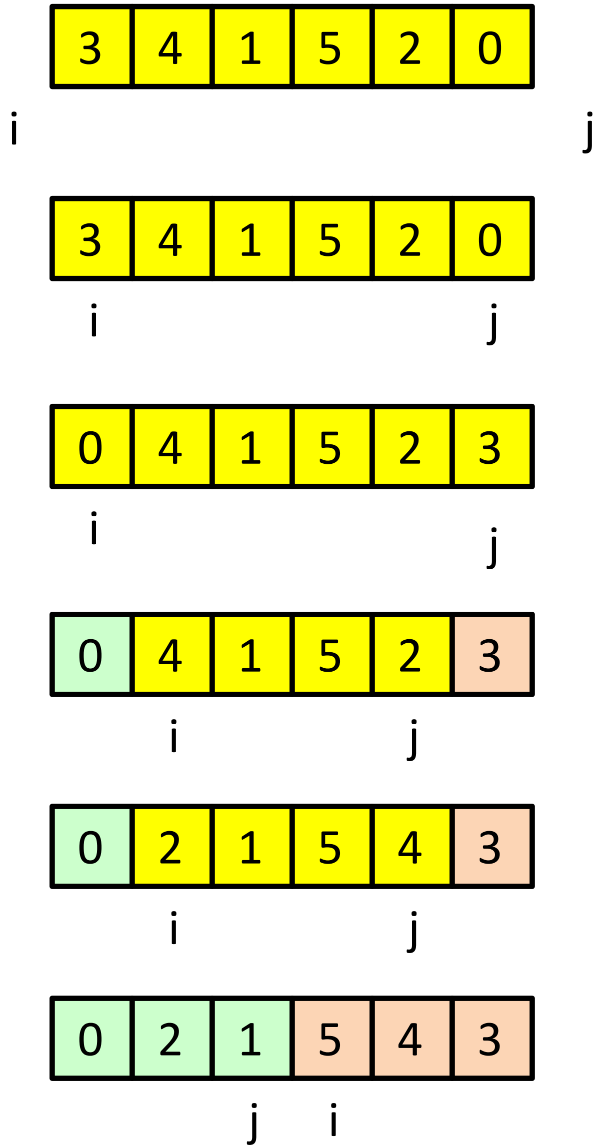
- **In-class exercise 2.1:**
- How many swaps (at most) are performed by the method below? How many array accesses (at most)?

```
private static int partition (int[] array) {
    if (array.length == 0) return 0;

    int pivot = array[0];
    int i = -1;
    int j = array.length;

    while(true) {
      do { ++i; } while (array[i] < pivot);
      do { --j; } while (array[j] > pivot);
      if (i >= j) return j + 1;
      swap(array, i, j);
    }
  }
```

Example execution




```
private static int
        partition (int[] array) {
    if (array.length == 0)
      return 0;

    int pivot = array[0];
    int i = -1;
    int j = array.length;

    while(true)  {
      do { ++i; }
      while (array[i] < pivot);
      do { --j; }
      while (array[j] > pivot);
      if (i >= j) return j + 1;
      swap(array, i, j);
    }
  }
```

Complexity

Reading a file

- Recall the code from the last lecture

```
String result = “”;
int num_chars = 0;
Character c = readChar();
while(c != null) {
        result += c;
        num_characters += 1;
        c = readChar();
}
System.out.println(num_chars);
System.out.print(result);
```

Reading a file

- The same in Python

```
import sys

s = ''
num_chars = 0

while True:
    c = sys.stdin.read(1)
    s2 = s
    if c == '':
        break
    s += c
    num_chars += 1

print num_chars
print s,
```

Reading a file

- The same in Go

```
 var in = bufio.NewReader(os.Stdin)
 var s = ""
 var num_chars = 0
 for {
    var c, err = in.ReadByte()
    if err == io.EOF {
        break
    }
    s += string(c)
    num_chars += 1
}
fmt.Println(num_chars)
fmt.Print(s)
```

Reading a file

- Fast Java version

```
StringBuilder result = new StringBuilder();
int num_chars = 0;
Character c = readChar();
while(c != null) {
        result.append(c);
        num_chars += 1;
        c = readChar();
}
System.out.println(num_chars);
System.out.print(result);
```

Reading a file

- Fast Python version

```
import sys

s = bytearray('')
n = 0

while True:
    c = sys.stdin.read(1)
    s2 = s
    if c == '':
        break
    s += c
    n += 1

print n
print s,
```

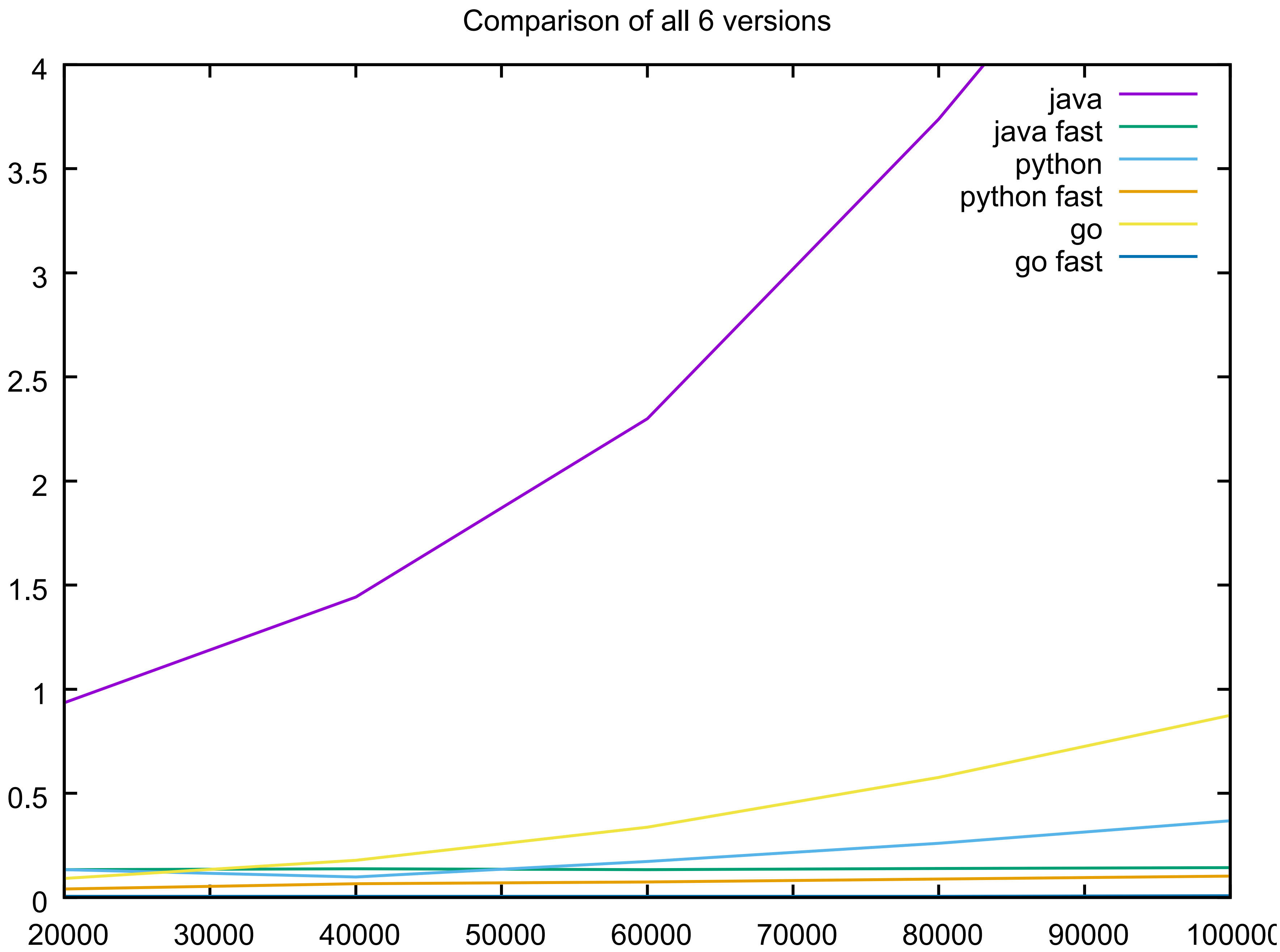
Mutable byte array

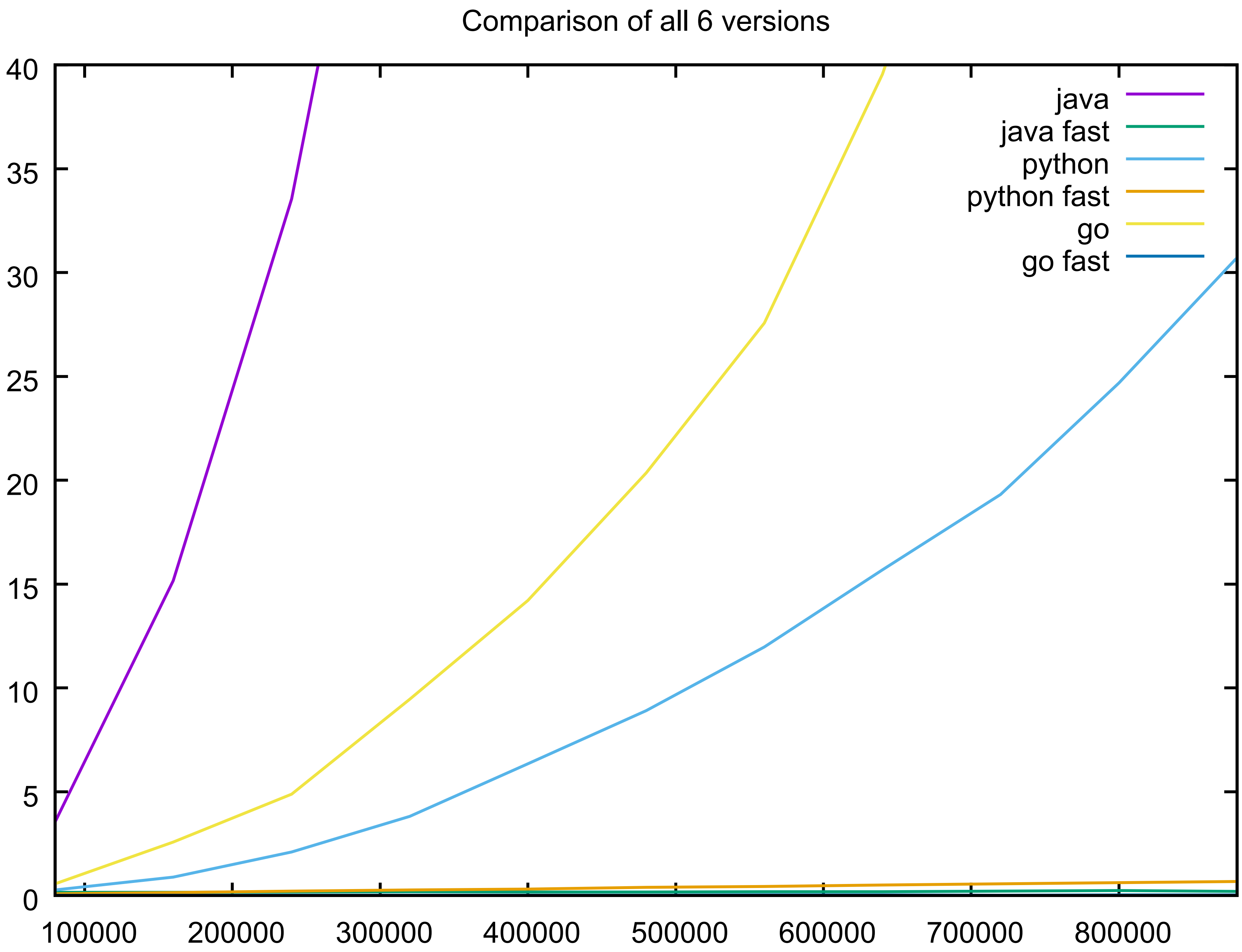
Reading a file

- Fast Go version

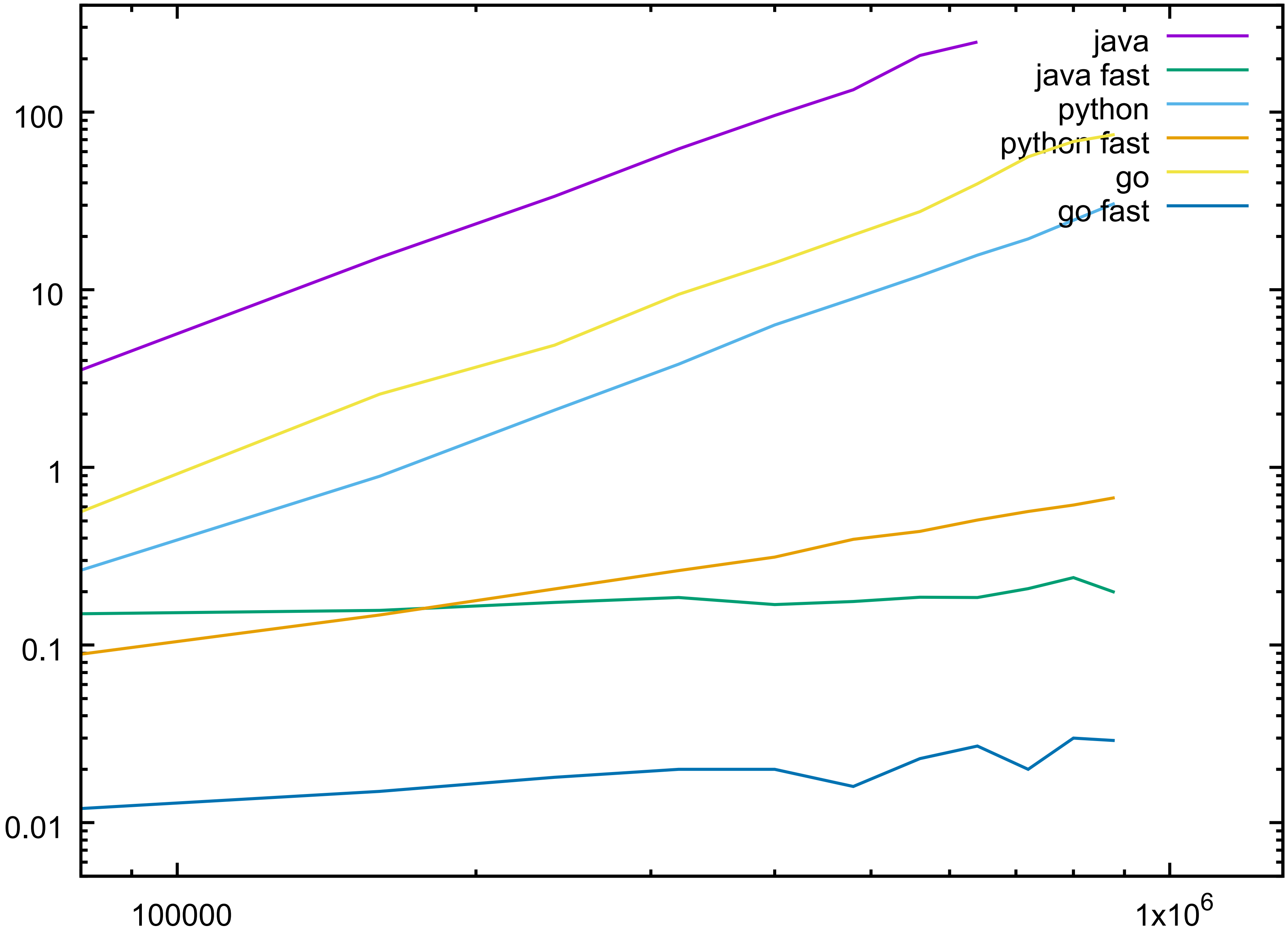
Mutable byte array

```
var in = bufio.NewReader(os.Stdin)
var s []byte
var num_chars = 0
for {
    var c, err = in.ReadByte()
    if err == io.EOF {
        break
    }
    s = append(s, c)
    num_chars += 1
}
fmt.Println(num_chars)
fmt.Printf("%s", s)
```


Comparison of all 6 versions
4
3.5
3
2.5
2
1.5
1
0.5
0
20000
30000
40000
50000
60000
70000
80000
90000
100000
java
java fast
python
python fast
go
go fast


Comparison of all 6 versions
java
java fast
python
python fast
go
go fast
0
5
10
15
20
25
30
35
40
100000
200000
300000
400000
500000
600000
700000
800000

Comparison of all 6 versions, log-log scale



Performance

- The ‘fast’ algorithm performs a lot better than the ‘slow’ algorithm
- The programming language makes a difference, but not nearly as much as the choice of the algorithm
- For file size 880k, the fastest of the ‘slow’ versions (Python) took 30s
- For the same file, the slowest of the ‘fast’ versions (again Python) took 0.7s

Performance, cont

- Choosing the right **algorithm** is often more important than the **implementation details**
- Studying properties of algorithms regardless of the implementation details is useful

Cost of running a program

- Time
  - The CPU time to execute the operations
  - Depends on a number of factors
  - Often *worst case* is considered
- Space
  - Maximum memory needed
  - Add the space needed to store all values existing at a given moment during program execution
- IO, interaction with external services
  - In this course we will largely ignore it
  - Often important in practice

Maximum: cost

```
private static int maximum (int[] array) {
    if (array.length == 0)
      throw new IllegalArgumentException
                  ("maximum requires a non-empty array");
    int cur = array[0];
    for (int i = 1; i < array.length; ++i) {
      cur = Math.max(cur, array[i]);
    }
    return cur;
  }
```

We need to count all the operations

Problems with cost

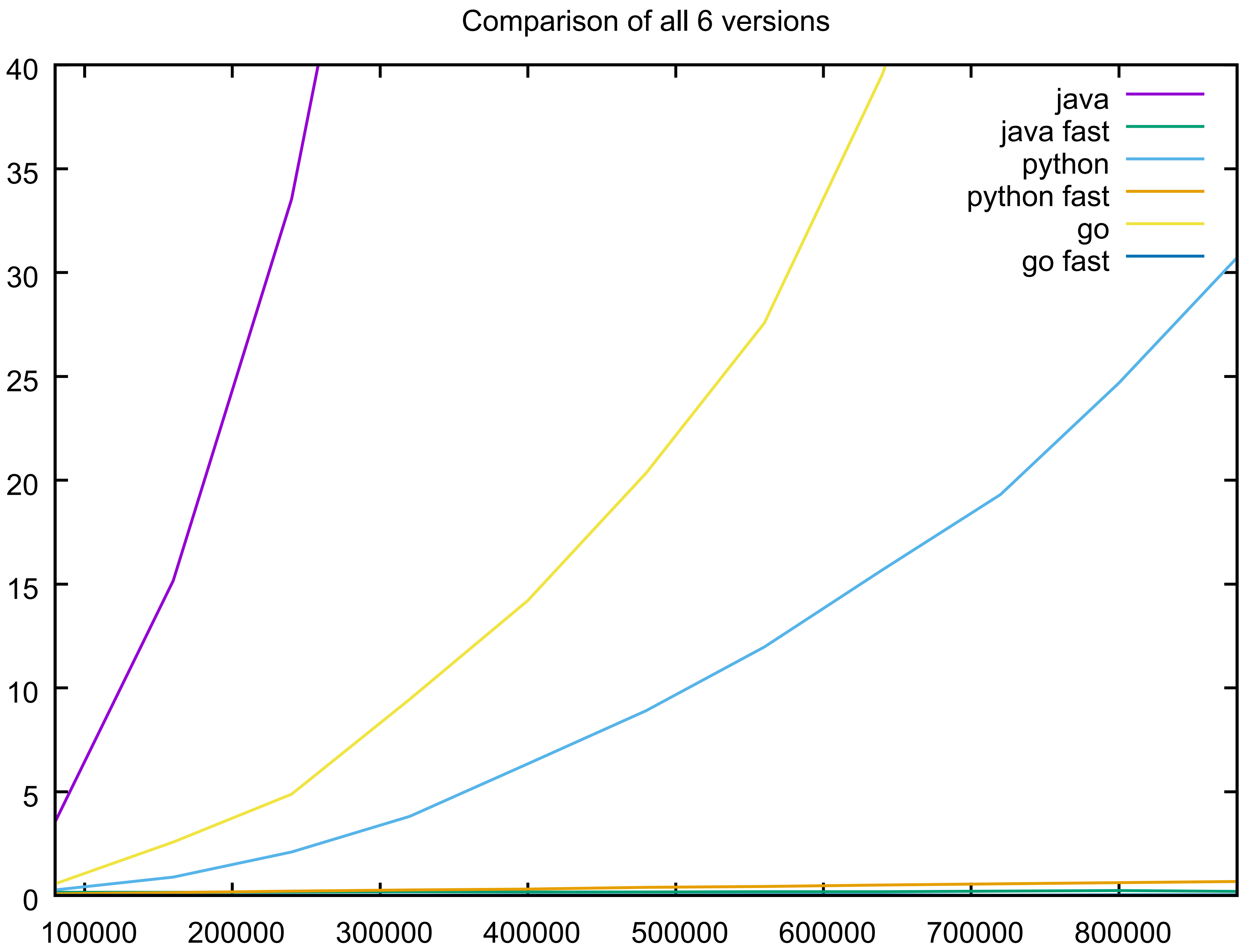
- We need to know the cost of every basic operation (addition, comparison, memory access, etc.)
  - This depends on the particular computer,
  - … on the programming language
  - … on the compiler
  - … on the operating system
  - The same operation might run slower or faster at different times (CPU non-determinism)
- Keeping track of all the different kinds of operations is difficult

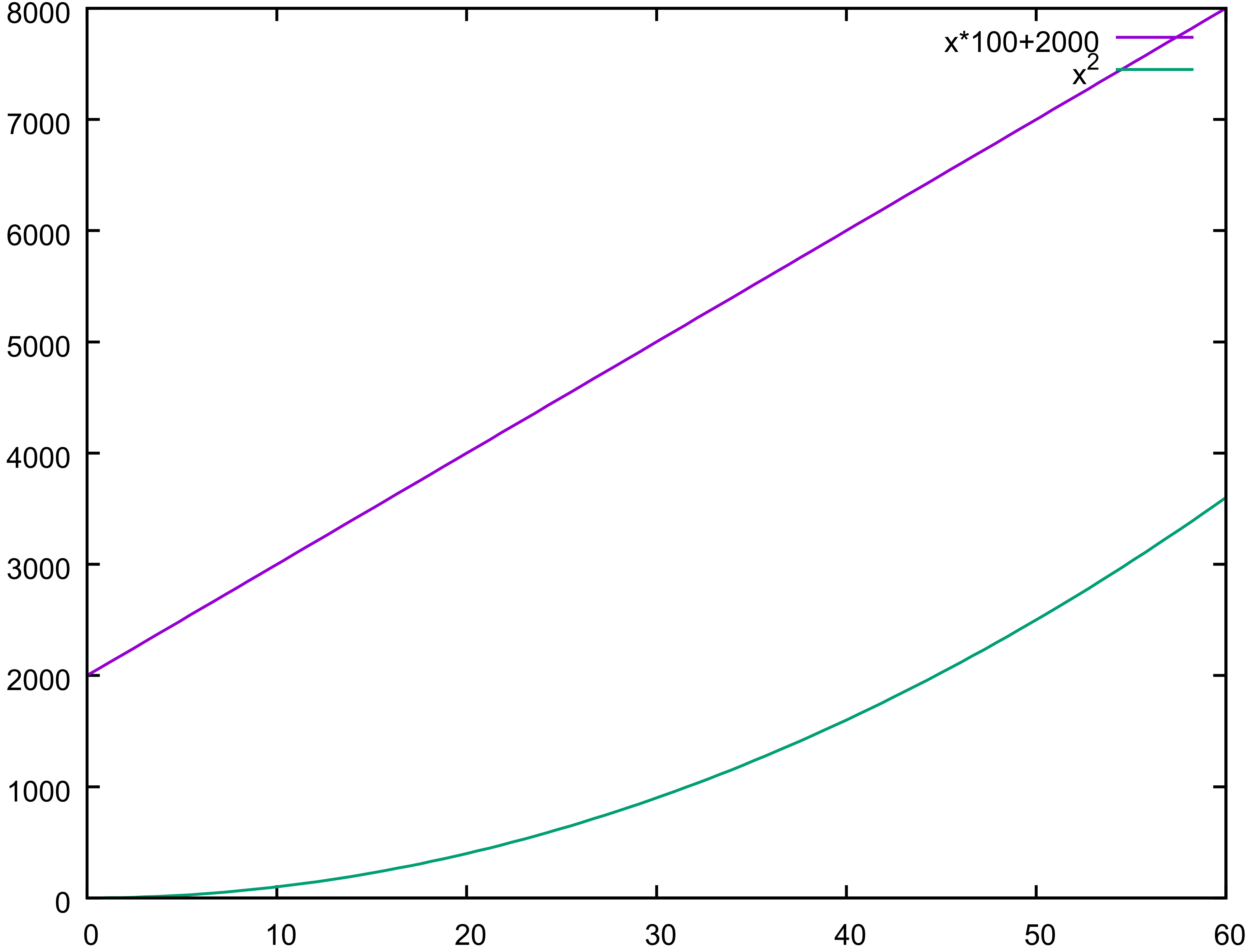
Exact counting is not feasible

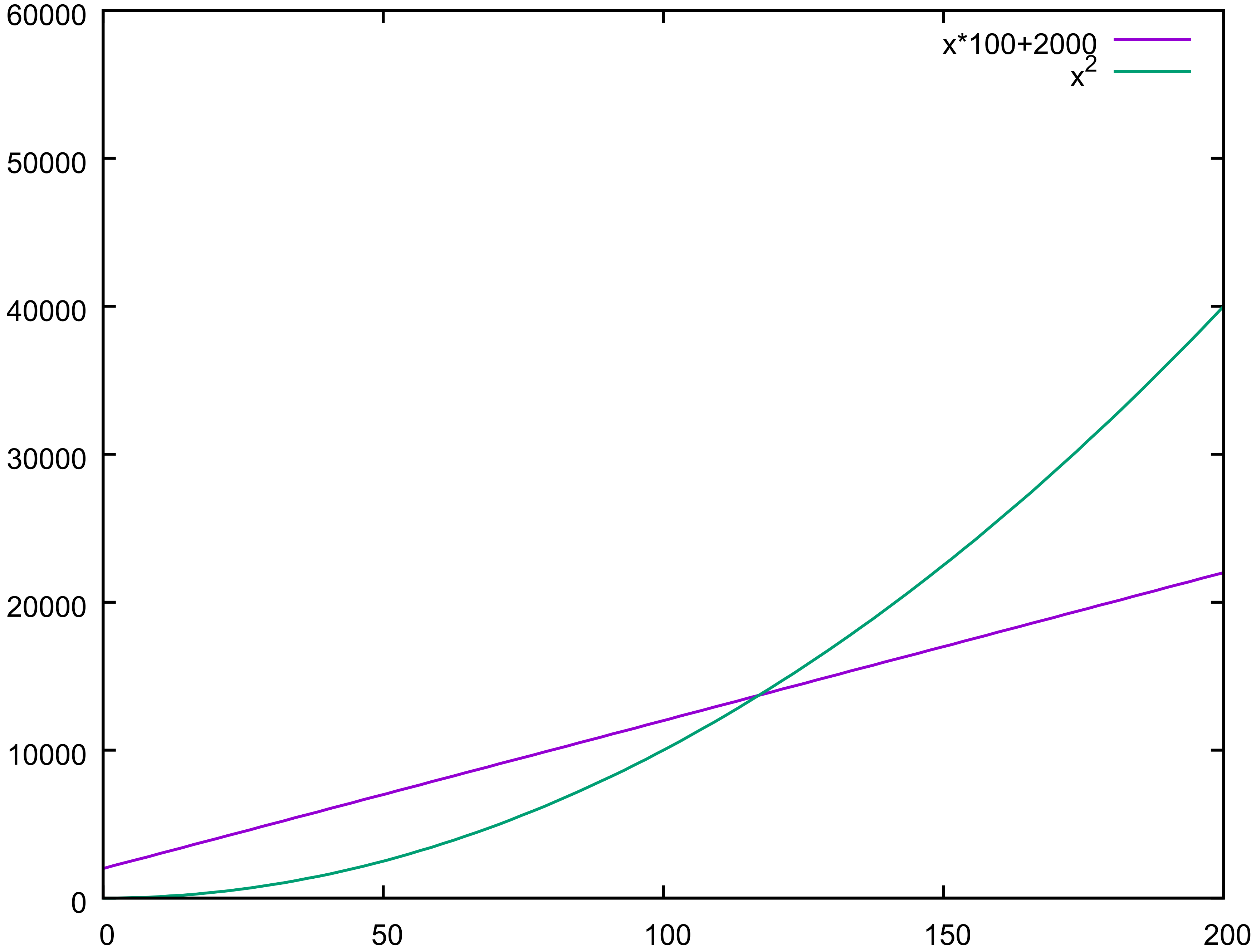
How should we count costs?

- Exact counting is difficult
- We want to distinguish the ‘slow’ algorithms from the ‘fast’ algorithms without considering specifics of programming languages/implementations

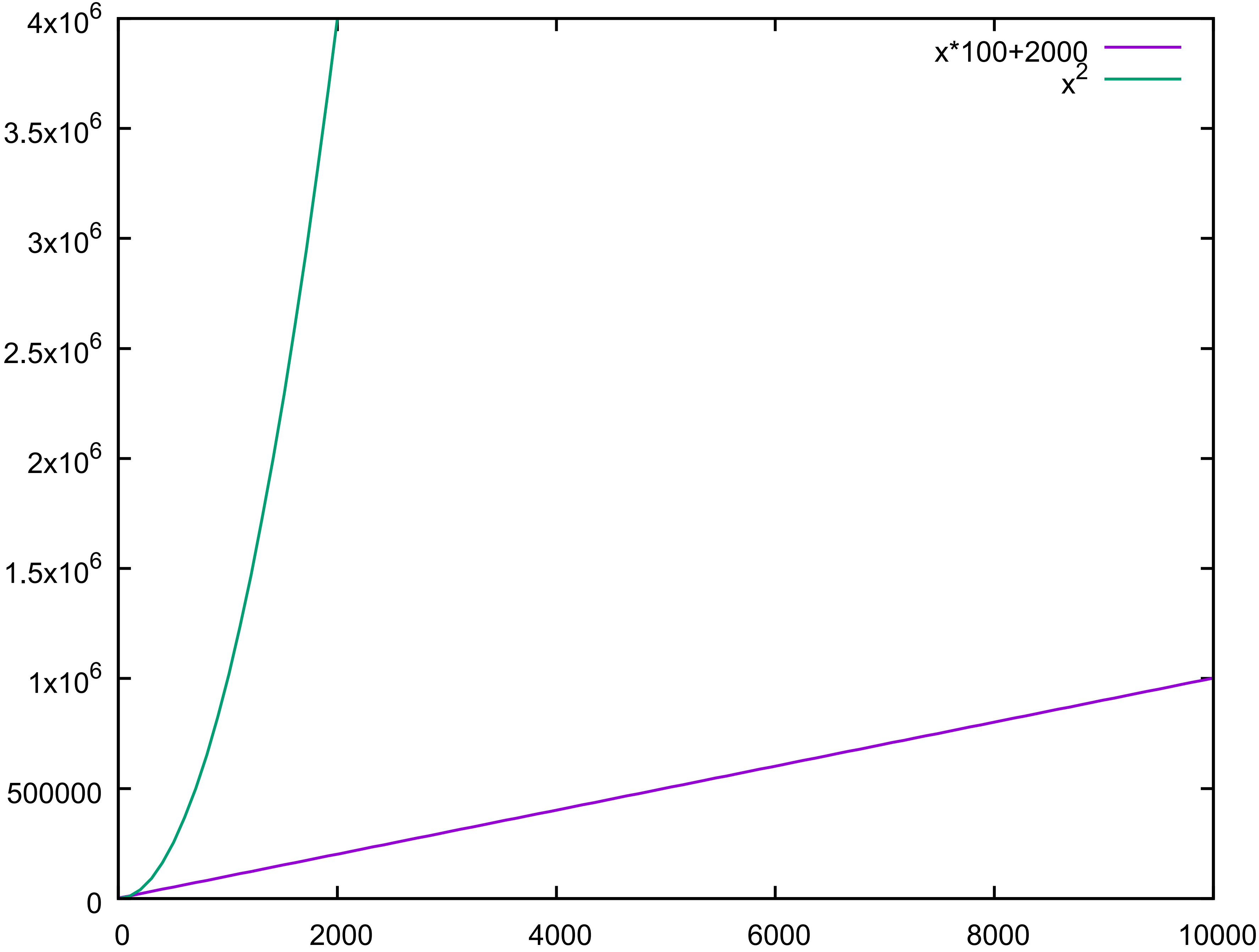
- Idea 1: Count every basic operation as 1
- Idea 2: Instead of comparing numbers, focus on the orders of growth of the functions


Comparison of all 6 versions
40
35
30
25
20
15
10
5
0
100000
200000
300000
400000
500000
600000
700000
800000
java
java fast
python
python fast
go
go fast


x*100+2000
x^2
8000
7000
6000
5000
4000
3000
2000
1000
0
0
10
20
30
40
50
60



60000
50000
40000
30000
20000
10000
0
0
50
100
150
200
x*100+2000
x^2



4x10^6
3.5x10^6
3x10^6
2.5x10^6
2x10^6
1.5x10^6
1x10^6
500000
0
0
2000
4000
6000
8000
10000
x*100+2000
x^2

Big-O notation

Big-O notation

- The big O notation expresses the upper bound for asymptotic growth of a function
- If $\boldsymbol{f(n)}$ and $\boldsymbol{g(n)}$ are functions, we say that $\boldsymbol{f(n)} \in \boldsymbol{O(g(n))}$ if there exist positive constants $\boldsymbol{c}$ and $\boldsymbol{n_0}$ such that
  $\boldsymbol{0 \leq f(n) \leq cg(n)}$ for all $\boldsymbol{n \geq n_0}$

- **For example** $\boldsymbol{f(n) = 3n + 5}$ is in $\boldsymbol{O(n^2)}$
- In other words, $\boldsymbol{f(n)} \in \boldsymbol{O(g(n))}$ if $\boldsymbol{g(n)}$ grows at least as fast (asymptotycally) as $\boldsymbol{g(n)}$

Maximum: big O

```
private static int maximum (int[] array) {
    if (array.length == 0)
      throw new IllegalArgumentException
                 ("maximum requires a non-empty array");
    int cur = array[0];
    for (int i = 1; i < array.length; ++i) {
      cur = Math.max(cur, array[i]);
    }
    return cur;
 }
```

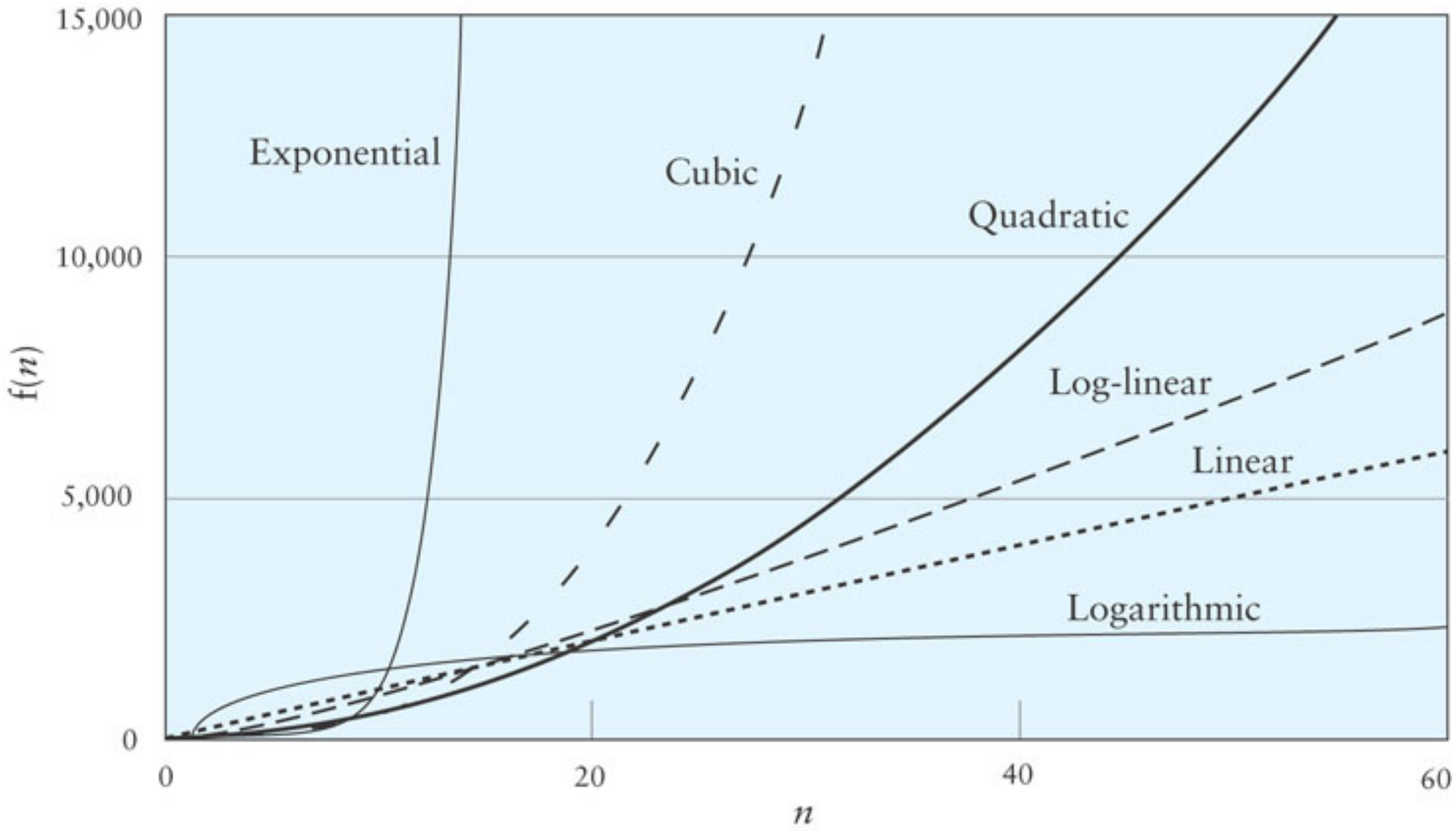
- The worst case cost: $f(n) = c_1 + c_2 n$ of time;
- $g(n) = c_3$ of memory where n is the size of the array, and c_1, c_2 and c_3 are some constants

Maximum: big O, cont.

- $\boldsymbol{f(n) = c_1 + c_2 n}$ is in $\boldsymbol{O(n)}$
- $\boldsymbol{g(n) = c_3}$ is in $\boldsymbol{O(1)}$, and also in $\boldsymbol{O(n)}$, $\boldsymbol{O(\log n)}$, etc.
- The growth rate of the cost of the algorithm is called its (time- or space-) *complexity*

Hierarchy of complexity classes

- $\boldsymbol{O}(\mathbf{1}) \subset \boldsymbol{O}(\log \boldsymbol{n}) \subset \boldsymbol{O}(\boldsymbol{n}) \subset \boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n}) \subset \boldsymbol{O}(\boldsymbol{n}^{2}) \subset \boldsymbol{O}(\boldsymbol{n}^{2} \log \boldsymbol{n}) \subset \boldsymbol{O}(\boldsymbol{n}^{3}) \subset \boldsymbol{O}(\mathbf{2}^{n})$



Big-O notation

- Allows us to forget about details that we are unsure about (how much each individual operation costs exactly)
- Also lets us calculate the complexity faster

Complexity of a loop

```
for (int i = 0; i < n; ++i) {
  // something of complexity O(f(n))
}
```

- What is the complexity of a loop containing statements of known complexities?

- The complexity of the whole loop is $\boldsymbol{O(nf(n))}$

Complexity of a sequence

```
// something of complexity O(f(n))
// something of complexity O(g(n))
```

- What is the complexity of a sequence of statements of known complexities?
- The complexity of the sequence is $\boldsymbol{O(f(n) + g(n))}$

Complexity of a sequence, cont.

```
for(int i = 0; i < n; i++) array[i] = 0;
for(int i = 0; i < n; i++) {
  for(int j = 0; j < n; j++)
    array[i] += array2[j];
}
```

- **In-class exercise 2.2:**
- What is the complexity of the code above?
- The complexity of the code is $\boldsymbol{O(n^2)}$

Appending to a string

```
String result = "";
int num_chars = 0;
Character c = readChar();
while(c != null) {
        result += c;
        num_characters += 1;
        c = readChar();
}
System.out.println(num_chars);
System.out.print(result);
```

```
StringBuilder result =
  new StringBuilder();
int num_chars = 0;
Character c = readChar();
while(c != null) {
        result.append(c);
        num_characters += 1;
        c = readChar();
}
System.out.println(num_chars);
System.out.print(result);
```

Naïve appending

```
int[] array = {};
for (int i = 0; i < n; i++) {
  int[] newArray = new int[array.length+1];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  newArray = array;
}
```

- The code above executes a similar number of operations as the naïve appending code
- What is the complexity of the code above?

Appending with doubling

```
int[] array = {0};
for (int i = 1; i <= n; i*=2) {
  int[] newArray = new int[array.length*2];
  for (int j = 0; j < i; j++)
    newArray[j] = array[j];
  newArray = array;
}
```

- **In-class exercise 2.3:**
- The code above executes a similar number of operations as the faster appending code
- What is the complexity of the code above?

Estimated complexity bounds

- The naïve version is $\boldsymbol{O(n^2)}$ – estimated correctly
- The fast version was estimated as $\boldsymbol{O(n \log n)}$, which is slower than it's actual growth rate of $\boldsymbol{O(n)}$
- To get a correct estimate, we need to realise that $\mathbf{1 + 2 + 4 + \cdots + 2^{\lfloor \log n \rfloor}}$ is $\boldsymbol{O(n)}$

Big-Theta notation

- Big-O allows us to state the one function grows asymptotically no faster than another function
- To state that two functions grow asymptotically equally fast, we may use big-Theta
- We say that $\boldsymbol{f(n) \in \Theta(g(n))}$ iff $\boldsymbol{f(n) \in O(g(n))}$ and $\boldsymbol{g(n) \in O(f(n))}$

Conclusion

- The **implementations** of the same algorithm in different programming languages are similar
- The general performance often affected more by what **algorithm** is implemented than by **implementation** details
- It is useful to study properties of algorithms that are **independent** of a particular implementation
- The big-O notation lets us differentiate between algorithms that have substantially different performance by looking at their **asymptotic performance**