ETH zürich

Departement Informatik

Programming and Problem-Solving

Complexity and Primality Testing

Dennis Komm

Spring 2021 – March 25, 2021

Time Complexity of Algorithms

Primality Testing

Exercise – Primality Testing

Write a function that

- takes an integer x as parameter
- calculates whether x is prime
- uses the % operator
- depending on that either returns `True` or `False`

Primality Test

```
def primetest(x):
    if x < 2:
        return False
    d = 2
    while d < x:
        if x % d == 0:
            return False
        d += 1
    return True
```

Primality Test

How long does it take the algorithm to produce the output?

- What is its **time complexity**?
- This depends on the number of loop iterations
- An absolute value does not make sense here
- The loop is iterated (roughly) `x` times (if `x` is prime)

⇨ Time complexity grows with `x` ... **but how fast?**

Time Complexity – Function of input size

- We measure the time complexity as a function of the input size
- The input of our algorithm is a single number `x`
- In our computer, numbers are represented in binary
- Ignoring leading zeros, for n bits we obtain

2^{n-1} is $\underbrace{10\ldots00}_{n}$, $2^{n-1}+1$ is $\underbrace{10\ldots01}_{n}$, $\ldots,$ and 2^n-1 is $\underbrace{11\ldots11}_{n}$

A number that is encoded with n bits has size around 2^n

Time Complexity – Technology Model

Random Access Machine

- **Execution model:** Instructions are executed one after the other (on one processor core)
- **Memory model:** Constant access time
- **Fundamental operations:** Computations $(+, -, \cdot, \ldots)$ comparisons, assignment / copy, flow control (jumps)
- **Unit cost model:** Fundamental operations provide a cost of 1

Time Complexity – Note

We are not completely accurate here

- Numbers can have arbitrarily large values
- We assume that arithmetic operations can be done in constant time
- The time needed to add two n-bit numbers depends on n
- Encoding of a floating point number does not directly correspond to its size
- Surely an addition is faster than a multiplication
- Logarithmic cost model takes this into account, but we also won't use it here

Time Complexity of Our Primality Test

- Suppose `x` is a prime number, encoded using n bits
- Number of loop iterations grows with size of `x` $\approx 2^n$
- Loop is iterated around 2^n times
- We would like to count the **fundamental operations**
- Algorithm executes five operations per iteration
- In total roughly $5 \cdot 2^n$ operations
- We would like to know how time complexity behaves when n grows
- Ignore constant 5

Time Complexity of Algorithms

Asymptotic Upper Bounds

Asymptotic Upper Bounds

The exact time complexity can usually not be predicted even for small inputs

- We are interested in **upper bounds**
- We consider the asymptotic behavior of the algorithm
- And ignore all constant factors

Example

- Linear growth with gradient 5 is as good as linear growth with gradient 1
- Quadratic growth with coefficient 10 is as good as quadratic growth with coefficient 1

Asymptotic Upper Bounds

Big-$\mathcal{O}$ Notation

The set $\mathcal{O}(2^n)$ contains all functions that do not grow faster than $c \cdot 2^n$ for some constant c

The set $\mathcal{O}(g(n))$ contains all functions $f(n)$ that do not grow faster than $c \cdot g(n)$ for some constant c, where f and g are positive

- Use asymptotic notation to specify the time complexity of algorithms
- We write $\mathcal{O}(n^2)$ and mean that the algorithm behaves for large n like n^2: when the input length is doubled, the time taken multiplies by four (at most)

Asymptotic Upper Bounds – Formal Definition

$\mathcal{O}$ Notation

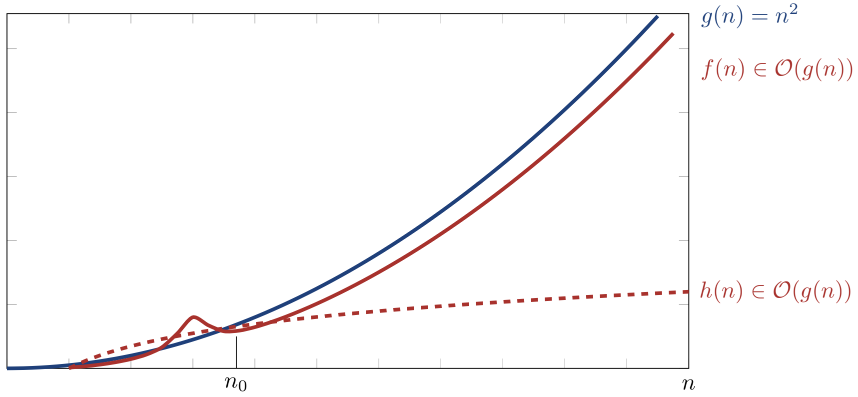
The set $\mathcal{O}(g(n))$ contains all functions $f(n)$ that do not grow faster than $c \cdot g(n)$ for some constant c, where f and g are positive

$$f(n) \in \mathcal{O}(g(n))$$

$$\Longleftrightarrow$$

$$\exists c > 0, n_0 \in \mathbb{N} \text{ such that } \forall n \geq n_0 \colon f(n) \leq c \cdot g(n)$$

Asymptotic Upper Bounds – Illustration



Asymptotic Upper Bounds – Examples

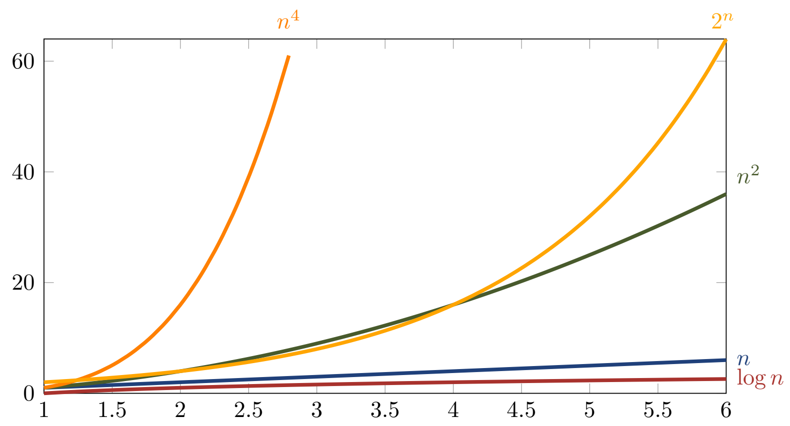
$$\mathcal{O}(g(n)) = \{f \colon \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0, n_0 \in \mathbb{N} \colon \forall n \geq n_0 \colon f(n) \leq c \cdot g(n)\}$$

$f(n)$	$f \in \mathcal{O}(?)$	Example
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$	$\mathcal{O}(n)$	$c = 2, n_0 = 0$
$n^2 + 100n$	$\mathcal{O}(n^2)$	$c = 2, n_0 = 100$
$n + \sqrt{n}$	$\mathcal{O}(n)$	$c = 2, n_0 = 1$

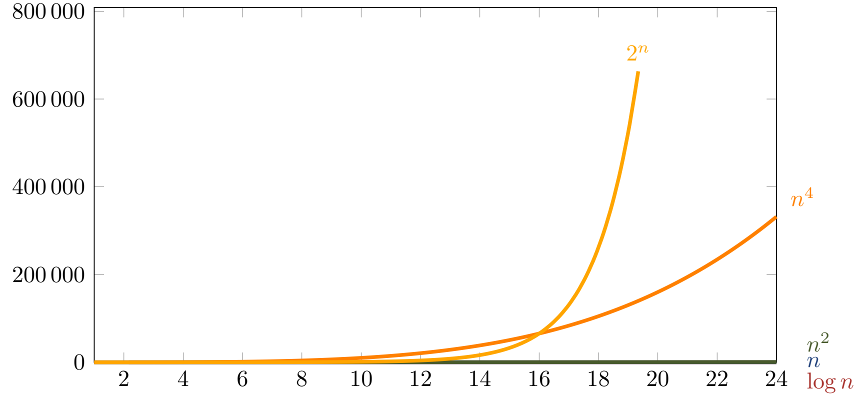
Time Complexity of Algorithms

Time Complexity Analysis

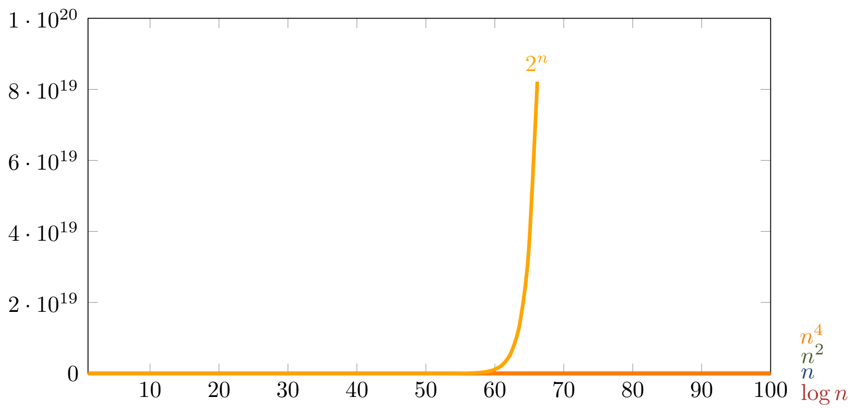
Small n



Larger n



“Large” n



Faster Primality Testing

First Attempt

Faster Primality Testing

Goal

Time complexity better than $\Omega(2^n)$

Observation

- If `x` is not divisible by 2, then it also is not divisible by 4, 6, 8, etc.
- We then only have to check odd numbers
- Algorithm only has to test half the numbers
- Loop is only iterated around $\mathtt{x}/2$ times

Faster Primality Testing

```
def primetest2(x):
    if x < 2 or (x > 2 and x % 2 == 0):
        return False

    d = 3d = 3
    while d < x:
        if x % d == 0:
            return False
        d += 2d += 2

    return True
```


Faster Primality Testing

What is the gain?

- Loop is iterated roughly $\mathtt{x}/2$ times instead of `x` times
- Time complexity improves by a factor of 2
- Again assume `x` is encoded using n bits
- Around $5 \cdot 2^n/2 = 2.5 \cdot 2^n$ fundamental operations in total
- Time complexity is still in $\mathcal{O}(2^n)$

⇨ No asymptotic improvement

Faster Primality Testing

Second Attempt

Faster Primality Testing

Observation

- If x with $\mathtt{x} > 2$ is not a prime number, then x is divisible by a number a with

$$1 < a < \mathtt{x}$$

- Then x is also divisible by a number b with

$$a \cdot b = \mathtt{x} \quad \text{and} \quad 1 < b < \mathtt{x}$$

- It cannot be the case that

$$a > \sqrt{\mathtt{x}} \quad \text{and} \quad b > \sqrt{\mathtt{x}},$$

since then

$$a \cdot b > \mathtt{x}$$

Faster Primality Testing

Including Modules

Including Modules

So far all functions have been defined in a single file

Modules

- Distribute functions over multiple files
- Files cannot "see" each other
- Functions can be **imported**
- Structured code

Including Modules

File `functions.py`

```
def square_root(n):
    i = 1
    while i * i < n: # Computer root of next larger square number
        i += 1
    return i
```

File `applications.py`

```
from functions import *

print(square_root(81))
```

Including Modules

- A large number of modules already exists
- For instance, there is a module `math` which includes a function `sqrt()` to compute square roots

```
from math import sqrt

print(sqrt(9))
```

Output: 3

Faster Primality Testing

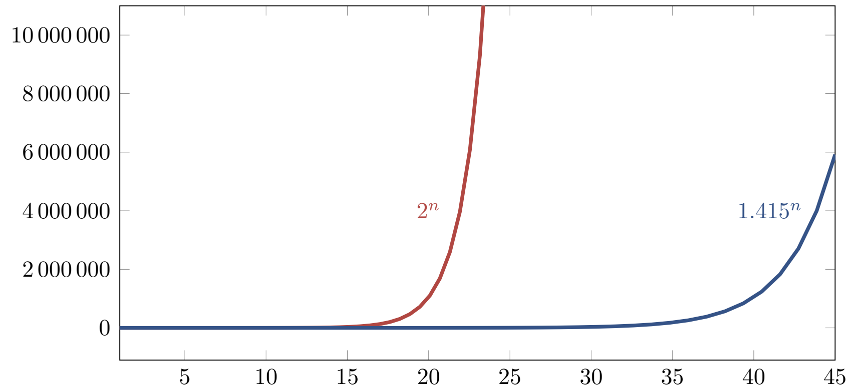
```
from math import sqrt

def primetest3(x):
    if x < 2 or (x > 2 and x % 2 == 0):
        return False

    d = 3
    while d < x<= sqrt(x):
        if x % d == 0:
            return False
        d += 2
    return True
```


Faster Primality Testing

What is the gain this time?

- What is the time complexity of this algorithm?
- Loop is iterated $\sqrt{\mathtt{x}}/2$ times
- Time complexity “grows” with $\sqrt{\mathtt{x}}$
- Time complexity is in $\mathcal{O}(\sqrt{2^n}) = \mathcal{O}(2^{n/2}) = \mathcal{O}(1.415^n)$

Faster Primality Testing



Faster Primality Testing

Suppose our computer can do 1000 iterations of the loop per second; for $x = 100\,000\,000\,000\,031$ this means:

`... d < x ...`

$$\frac{100\,000\,000\,000\,031 \text{ iterations}}{1000 \,\frac{\text{iterations}}{\text{second}}} > 100\,000\,000\,000 \text{ seconds} > 3100 \text{ years}$$

`... d <= sqrt(x) ...`

$$\frac{\sqrt{100\,000\,000\,000\,031} \text{ iterations}}{1000 \,\frac{\text{iterations}}{\text{second}}} < \frac{10\,000\,000 \text{ iterations}}{1000 \,\frac{\text{iterations}}{\text{second}}} < 3 \text{ hours}$$

Even if the computer that runs the slower program is 100 time faster, it still needs 31 years

Faster Primality Testing

Or the other way around...

Suppose we want to spend 10 minutes

Then there are at most "testable" primes in the magnitude of:

`... d < x ...`

$$\frac{x \text{ iterations}}{1000 \,\frac{\text{iterations}}{\text{second}}} = 600 \text{ seconds}$$

$$\Longleftrightarrow x = 600\,000$$

`... d <= sqrt(x) ...`

$$\frac{\sqrt{x} \text{ iterations}}{1000 \,\frac{\text{iterations}}{\text{second}}} = 600 \text{ seconds}$$

$$\Longleftrightarrow x = 600\,000^2$$

$$\Longleftrightarrow x = 360\,000\,000\,000$$

Faster Primality Testing

Best and Worst Case Analysis

Best and Worst Case Analysis

Which algorithm is faster?

```
def primetest3(x):
    if x < 2 or (x > 2 and x % 2 == 0):
        return False

    d = 3
    while d <= sqrt(x):
        if x % d == 0:
            return False
        d += 2
    return True
```

```
def primetest4(x):
    if x < 2 or (x > 2 and x % 2 == 0):
        return False

    d = 3
    isprime = True
    while d <= sqrt(x):
        if x % d == 0:
            isprime = False
        d += 2
    return isprime
```

Best and Worst Case Analysis

Suppose $\mathtt{x}$ is a multiple of 3

- Then the left algorithm is faster
- ⇨ Loop is left after first iteration
- "'Early Exit"'
- Right algorithm makes roughly $1.415^n/2$ comparisons

Suppose $\mathtt{x}$ is prime

- Then both algorithms make $1.415^n/2$ comparisons
- (Of course, still the left one should be implemented)

What else can we do?

Primality test

Test every number between 1 and $\mathtt{x}$

Randomized Monte Carlo algorithm

Test every second number

Polynomial AKS algorithm

Monte-Carlo Algorithm

Monte-Carlo Algorithm – Basic Idea

Randomized Algorithms make random decisions

- Input x does not "determine" output anymore
- The same x may result in different outputs
- **Monte-Carlo Algorithm** (MC Algorithm) has bounded error probability
- For `True`/`False` problems (primality test etc.) there are MC algorithms with one-sided error (1MC algorithms)
- **Las Vegas Algorithm** has error probability 0

Monte-Carlo Algorithm (1MC) – Example

Consider urn with 10^{100} balls colored white (and possibly red)

- **Claim:** Not all balls in the urn are white
- How to test?
- Random sample
- ⇨ If there is a red ball in the sample ⇨ Claim proven
- ⇨ If there is **no** red ball in the sample ⇨ Claim possibly false
- One-sided error

Red balls are **witnesses** for claim

Simplified Solovay-Strassen Algorithm

Simplified Solovay-Strassen Algorithm (1MC)

- Test whether $\mathtt{x}$ is a prime
- **Claim:** $\mathtt{x}$ is not a prime
- Consider set $\{2, \ldots, \mathtt{x}-1\}$ as urn
- Divisor of $\mathtt{x}$ is witness for the claim
- Random sample
- ⇨ If there is a divisor of $\mathtt{x}$ in sample ⇨ Claim proven
- ⇨ If there are **no** divisors of $\mathtt{x}$ in sample ⇨ Claim possibly false
- One-sided error

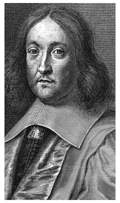
For $\mathtt{x} = p \cdot q$ with p and q being primes, probability to find a witness is

$$\frac{2}{\mathtt{x}-2}$$

Simplified Solovay-Strassen Algorithm (1MC)

- Find "better witnesses"
- (Not exactly trivial number theory)
- **Fermat's little theorem**

If $\mathtt{x}$ is prime $\Rightarrow a^{\mathtt{x}-1} \equiv 1 \pmod{\mathtt{x}} \quad \forall a \in \{2, \ldots, \mathtt{x}-1\}$



Pierre de Fermat (1607–1665)

Simplified Solovay-Strassen Algorithm (1MC)

- If $\mathtt{x}$ is prime $\Rightarrow \quad a^{\mathtt{x}-1} \mod \mathtt{x} = 1 \quad \forall a \in \{2, \ldots, \mathtt{x}-1\}$

$\mathtt{x} = 3{:}\quad 2^2 \equiv 1 \pmod{3}$
$\mathtt{x} = 5{:}\quad 2^4 \equiv 3^4 \equiv 1 \pmod{5}$

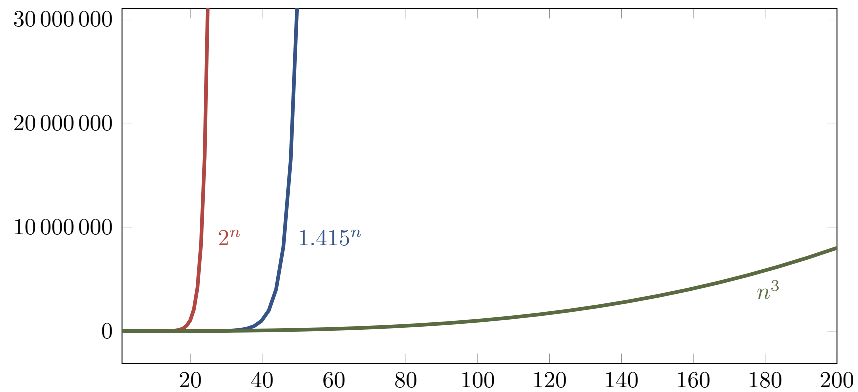
- If for one a we have: $\quad a^{\mathtt{x}-1} \mod \mathtt{x} \neq 1$
  - $\mathtt{x}$ is **definitely** no prime
  - a is witness that $\mathtt{x}$ is no prime
  - It can be proven that there are $> (\mathtt{x}-2)/2$ witnesses
- Otherwise $\mathtt{x}$ is possibly a prime

Simplified Solovay-Strassen Algorithm (1MC)

- **Input:** Number $\mathtt{x}$
- Choose a randomly from $\in \{2, \ldots, \mathtt{x}-1\}$
- Compute $z = a^{\mathtt{x}-1} \mod \mathtt{x}$
- If $z \neq 1$: **Output** "$\mathtt{x}$ is no prime"
- Otherwise: **Output** "$\mathtt{x}$ is possibly prime"

- Can be computed in polynomial time
- Time complexity $\mathcal{O}(n^3)$ instead of $\mathcal{O}(1.415^n)$
- Efficient algorithm

Simplified Solovay-Strassen Algorithm (1MC)



Simplified Solovay-Strassen Algorithm (1MC)

Algorithm has one-sided error

- Suppose $\mathtt{x}$ is a prime
- According to Fermat's little therom there is no witness in $\{2, \ldots, \mathtt{x} - 1\}$
- Correct output with probability 1
- Suppose $\mathtt{x}$ is no prime
- At least half of $\{2, \ldots, \mathtt{x} - 1\}$ are witnesses
- Correct output with probability $1/2$

Simplified Solovay-Strassen Algorithm (1MC)

Probability amplification by repeated execution each with an independent choice of a

- Run algorithm k times on the same $\mathtt{x}$
- **if** $\mathtt{x}$ is a prime, then error probability is 0
- **Else** only one witness has to be found
- Probability $< 1/2$ that no witness it found in 1. run
- Probability $< 1/4$ that no witness is found in 1. and 2. run
- Probability $< 1/k$ that no witness i found in all k runs