EBERHARD KARLS UNIVERSITÄT TÜBINGEN

FACULTY OF HUMANITIES

Department of General and Computational Linguistics

Analysis of Algorithms

Data Structures and Algorithms for CL III, WS 2019-2020

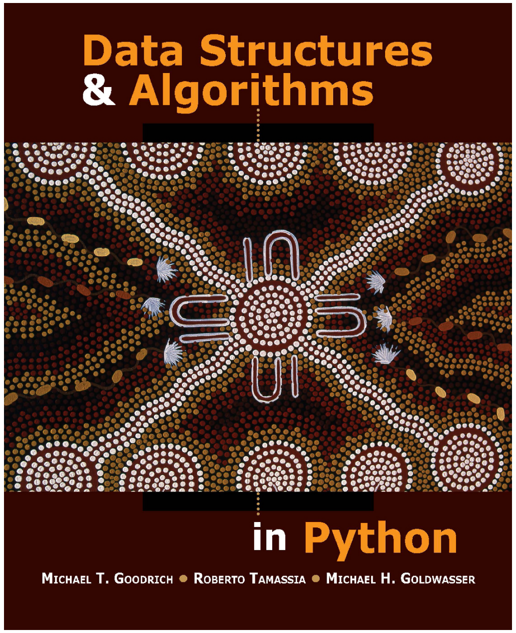
Corina Dima

corina.dima@uni-tuebingen.de

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

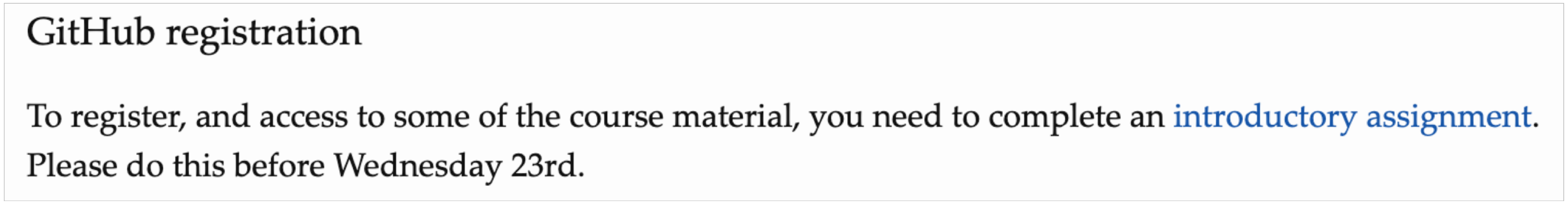
Data Structures & Algorithms in Python

MICHAEL GOODRICH
ROBERTO TAMASSIA
MICHAEL GOLDWASSER



1. Python Primer
2. Object-Oriented Programming

Don’t forget to register – registration closes tonight!

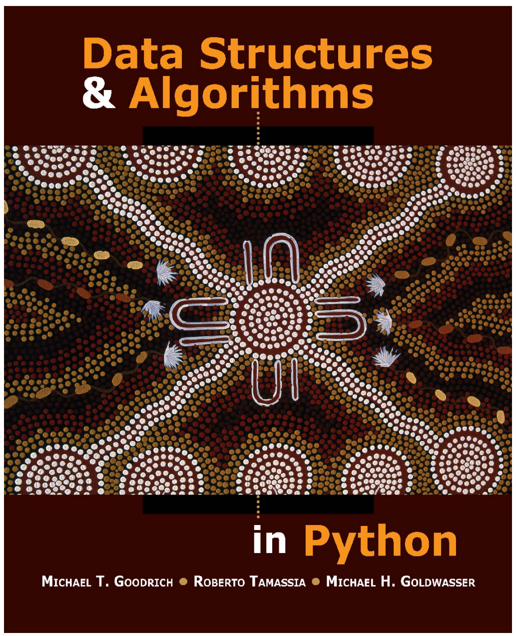

GitHub registration

To register, and access to some of the course material, you need to complete an introductory assignment.
Please do this before Wednesday 23rd.

https://dsacl3-2019.github.io/

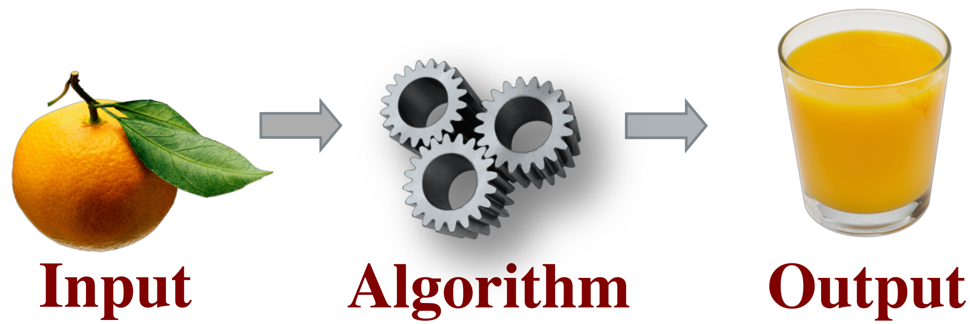
Data Structures & Algorithms in Python

MICHAEL GOODRICH
ROBERTO TAMASSIA
MICHAEL GOLDWASSER

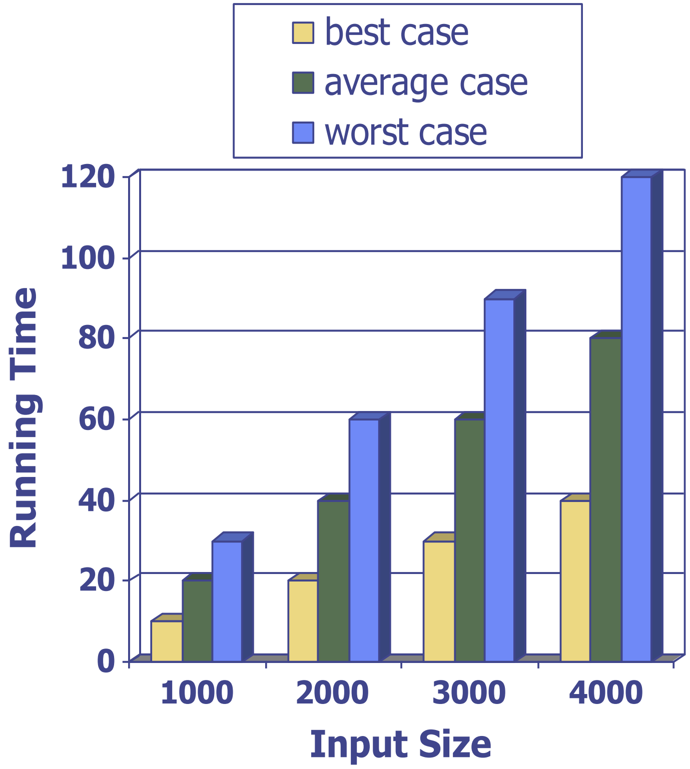


3. Algorithm Analysis

- experimental studies
- seven functions
- asymptotic analysis


Input
Algorithm
Output

Running Time



- The running time of an algorithm typically grows with the input size.
- But may also vary for inputs of the same size
- Running time is influenced by the hardware and software environment

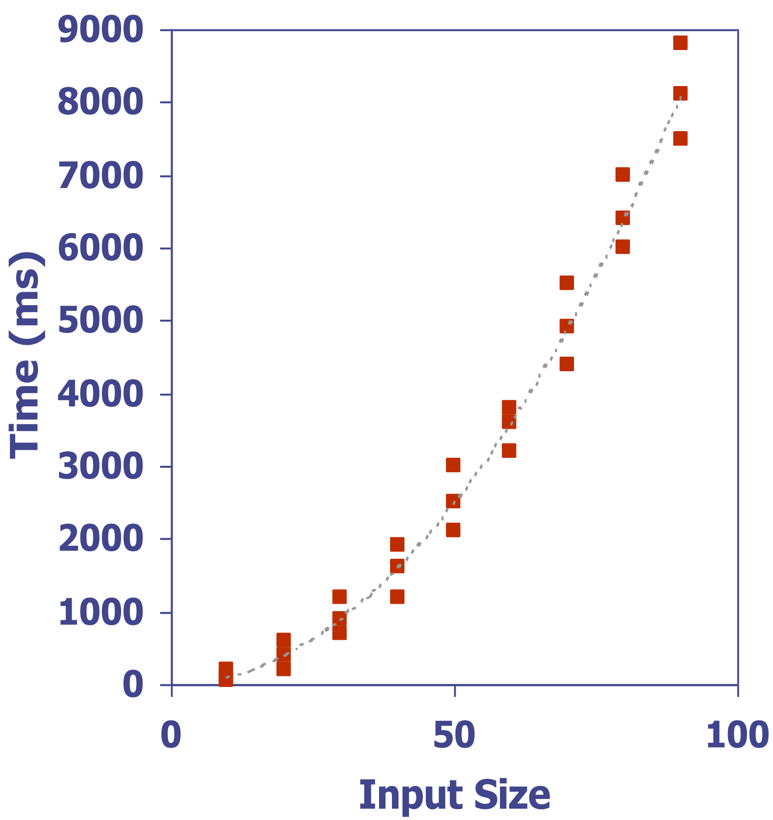
EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Experimental Study

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition, recording the time needed

```
from time import time
start_time = time( )
run algorithm
end_time = time( )
elapsed = end_time − start_time
```

- Analyze the results



or using `clock()` or the `timeit` module

Limitations of Experiments

Action | **Challenge**

Limitations of Experiments

Action	Challenge
Write a program implementing the algorithm	Algorithm must be fully implemented before performing an experimental study

Limitations of Experiments

Action	Challenge
Write a program implementing the algorithm	Algorithm must be fully implemented before performing an experimental study
Run the program with inputs of varying size and composition, recording the time needed	Experiments can only be done on a limited set of inputs

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Limitations of Experiments

Action	Challenge
Write a program implementing the algorithm	Algorithm must be fully implemented before performing an experimental study
Run the program with inputs of varying size and composition, recording the time needed	Experiments can only be done on a limited set of inputs
Analyze the results	Experimental runs of two different algorithms are difficult to compare directly unless the experiments are performed in the same hardware and software environments

Beyond Experimental Analysis

- An **approach** to **analyzing the efficiency of algorithms** that:
  1. Can be used to evaluate the relative efficiency of two algorithms independently of the hardware and software environment
  2. Can be performed by studying a high-level description of the algorithm (pseudocode), without actually implementing it
  3. Takes into account all possible inputs
  4. Characterizes running time as a function of the input size, n

Theoretical Analysis

- Perform the analysis directly on a high-level description of the algorithm
- **Count the number of primitive operations** that are executed, and use this number, *t*, as a measure of the running time of the algorithm

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Assumed to take a constant amount of time in the RAM model

Examples of Primitive Operations

- Assigning an identifier to an object
- Determining the object associated with an identifier
- Performing an arithmetic operation (e.g. adding two numbers)
- Comparing two numbers
- Accessing a single element of a list by index
- Calling a function
- Returning from a function

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Focusing on Worst-Case Input

- An algorithm might run faster on some inputs that it does on others of the same size
- Express the running time of an algorithm as a function of the input size obtained by taking the average over all possible inputs of the same size
- Challenging: requires defining a probability distribution over the set of inputs
- Solution: characterize running times in terms of the worst case, as a function of the input size, n, of the algorithm
- Easier: only need to identify the worst-case input
- Plus: performing well on the worst-case input means that the algorithm needs to do well on every input

- Associate, with each algorithm, a function *f(n)* that characterizes the number of primitive operations that are performed as a function of the input size *n*

Seven Important Functions in Algorithm Analysis

1. Constant	$f(n) = c$
2. Logarithmic	$f(n) = log_b n, \mathrm{b} > 1$
3. Linear	$f(n) = n$
4. N-log-N	$f(n) = n \log n$
5. Quadratic	$f(n) = n^2$
6. Cubic, other polynomials	$f(n) = n^3$
7. Exponential	$f(n) = b^n, b > 0$

The Constant Function

- $\boldsymbol{f(n) = c}, for\ some\ fixed\ constant\ c$
- No matter the n, the function assigns the value c
- c is a constant, e.g. $c = 5$, $c = 27$, $c = 2^{10}$
- But will use typically $g(n) = 1$, given that any other constant function $f(n) = c$ can be written as $f(n) = cg(n)$
- Simple, but helps characterize the number of steps needed to do a basic operation like adding or comparing two numbers

The Logarithm Function

- $\boldsymbol{f(n) = log_b n}$, $b > 1$
- Defined as: $x = log_b n \; if \; and \; only \; if \; b^x = n$
- By definition, $log_b 1 = 0$
- b is called the base of the logarithm
- The most commonly used base is 2: a common operation is to repeatedly divide the input in half

The Linear Function

- $\boldsymbol{f(n) = n}$
- Given an input value *n*, assigns the value itself
- Arises in algorithm analysis any time we have to do a single operation for each of n elements, e.g.
  - Comparing a number x to each element of a sequence of size n
  - Counting the number of elements in a sequence

The N-log-N Function

- $f(n) = n \log n$
- Base 2 logarithm
- Also called the linearithmic function (Sedgewick & Wayne, 2011)
- Grows a little more rapidly than the linear function, and a lot less rapidly than the quadratic function
- An n-log-n algorithm is usually preferable to a quadratic algorithm

The Quadratic Function

- $\boldsymbol{f(n) = n^2}$
- Given an input the function assigns the product of *n* with itself
- Appears in the analysis of algorithms because of nested loops, where the inner loop performs a linear number of operations, and the outer loop is performed a linear number of times
- Also appears in nested loops where the first iteration uses one operation, the second two operations, the third three operations etc., where the number of operations is

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + (n-2) + (n-1) + n =$$

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

The Quadratic Function

- $\boldsymbol{f(n) = n^2}$
- Given an input the function assigns the product of *n* with itself
- Appears in the analysis of algorithms because of nested loops, where the inner loop performs a linear number of operations, and the outer loop is performed a linear number of times
- Also appears in nested loops where the first iteration uses one operation, the second two operations, the third three operations etc., where the number of operations is

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + (n-2) + (n-1) + n = \frac{n(n+1)}{2}$$

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

The Quadratic Function

- $f(n) = n^2$
- Given an input the function assigns the product of n with itself
- Appears in the analysis of algorithms because of nested loops, where the inner loop performs a linear number of operations, and the outer loop is performed a linear number of times
- Also appears in nested loops where the first iteration uses one operation, the second two operations, the third three operations etc., where the number of operations is

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + (n-2) + (n-1) + n = \frac{n(n+1)}{2}$$

Card Friedrich Gauss, 1777 - 1855

The Cubic Function and Other Polynomials

- $\boldsymbol{f(n) = n^3}$
- $\boldsymbol{f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ \ldots + a_d n^d}$, where $a_0, a_1, a_2, a_3, \ldots, a_d$ are constants called the coefficients of the polynomial, and $a_d \neq 0$.
- d indicates the highest power of the polynomial and is called the degree of the polynomial
- Examples
  - $f(n) = 2 + 5n + n^2$
  - $f(n) = 1 + n^3$

The Exponential Function

- $\boldsymbol{f(n) = b^n, b > 0}$
- b is called the base, n is called the exponent
- $f(n)$ assigns to the input *n* the value obtained by multiplying the base b a total number of n times
- Appears in the analysis of algorithms where we have a loop that starts by performing one operation and then e.g. doubles the number of operations performed with each iteration – at the *n*^*th*^ iteration the number of operations performed is 2^n.

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \ldots + a^n$$

The Exponential Function

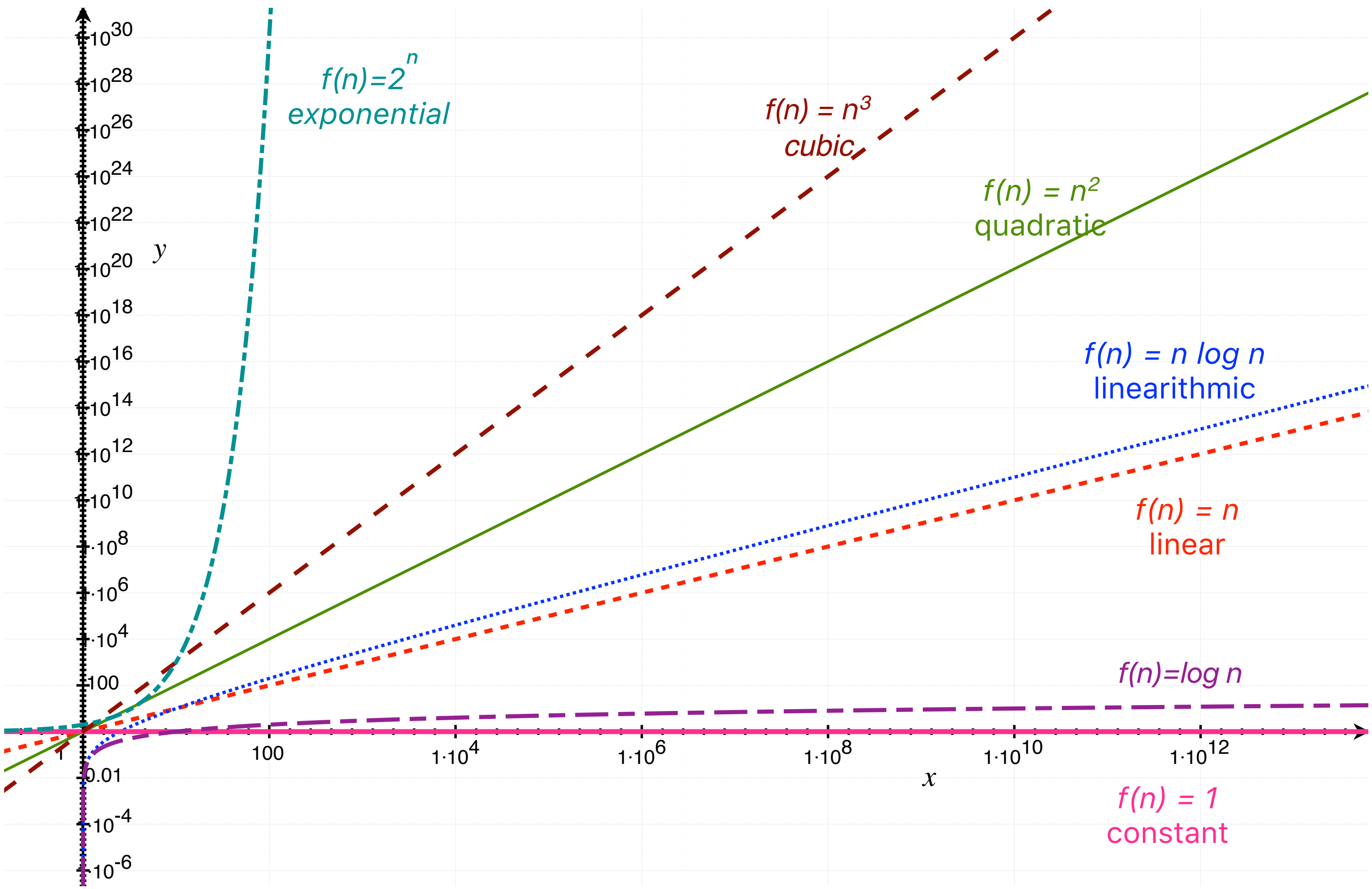
- $\boldsymbol{f(n) = b^n, b > 0}$
- b is called the base, n is called the exponent
- $f(n)$ assigns to the input *n* the value obtained by multiplying the base b a total number of n times
- Appears in the analysis of algorithms where we have a loop that starts by performing one operation and then e.g. doubles the number of operations performed with each iteration – at the *n*^th^ iteration the number of operations performed is 2^n.

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \ldots + a^n = \frac{a^{n+1} - 1}{a - 1}$$

Comparing Growth Rates

constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	2^n

Comparing Growth Rates



Comparing Growth Rates

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Comparing Growth Rates

Running Time (μs)	Maximum Problem Size (n) 1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$2n^2$	707	5,477	42,426
2^n	19	25	31

Better Hardware?

- The importance of a good algorithm goes beyond what can be solved effectively on a given computer
- Suppose a hardware speedup of 256 times – algorithm with given running times run 256 times faster on the new computer
- m is the size of the previous maximum problem size

Running Time	New Maximum Problem Size
$400n$	$256m$
$2n^2$	$16m$, because $16^2 = 256$
2^n	$m + 8$, because $2^8 = 256$

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Asymptotic Algorithm Analysis

- "big-picture approach": it is often enough just to know that the running time of an algorithm grows proportionally to n
- Analyze algorithms using a mathematical notation for functions that disregard constant factors
- Characterize running times of algorithms by using functions that map the size of the input, n, to values that correspond to the main factor that determines the growth rate in terms of n
- Analyze an algorithm by estimating the number of primitive operations executed up to a constant factor

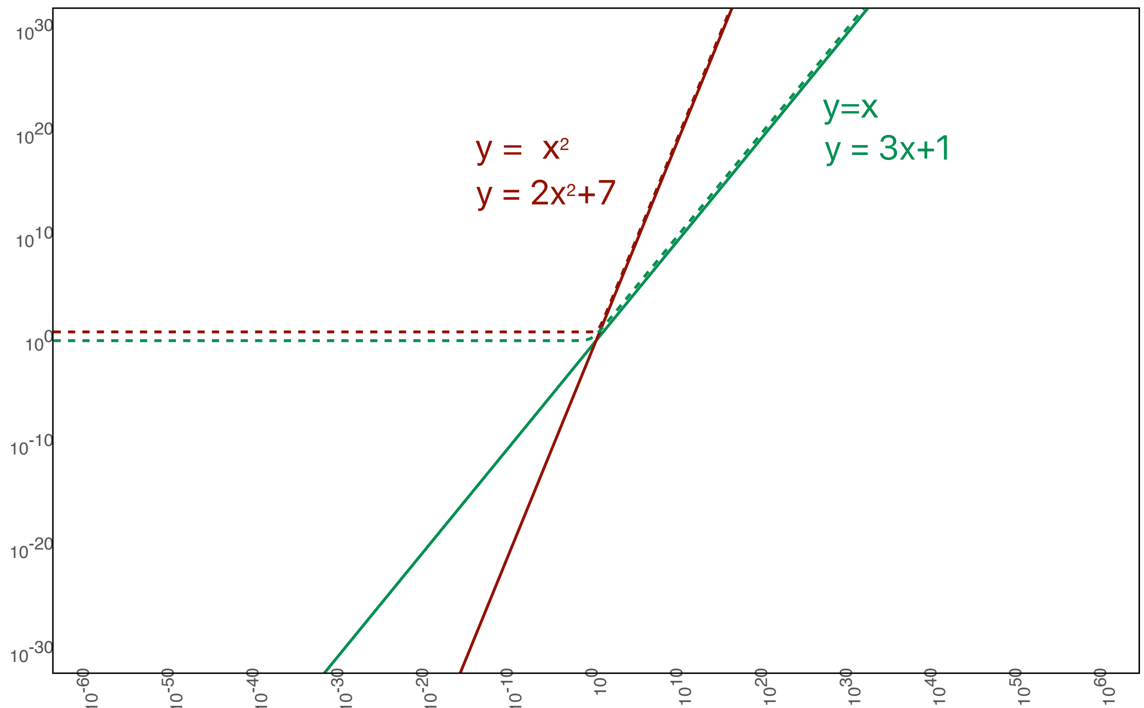
Counting Primitive Operations

```
1 def find_max(data):
2   """Return the maximum element from a nonempty Python list."""
3   biggest = data[0]                # The initial value to beat
4   for val in data:                 # For each value:
5     if val > biggest               # if it is greater than the best so far,
6       biggest = val                # we have found a new best (so far)
7   return biggest                   # When loop ends, biggest is the max
```

Step 1	Step 3	Step 4	Step 5	Step 6	Step 7
2 ops	2 ops	2n ops	2n ops	0 to n ops	1 op

Constant Factors

- The growth rate is not affected by
  - Constant factors
  - Lower-order terms

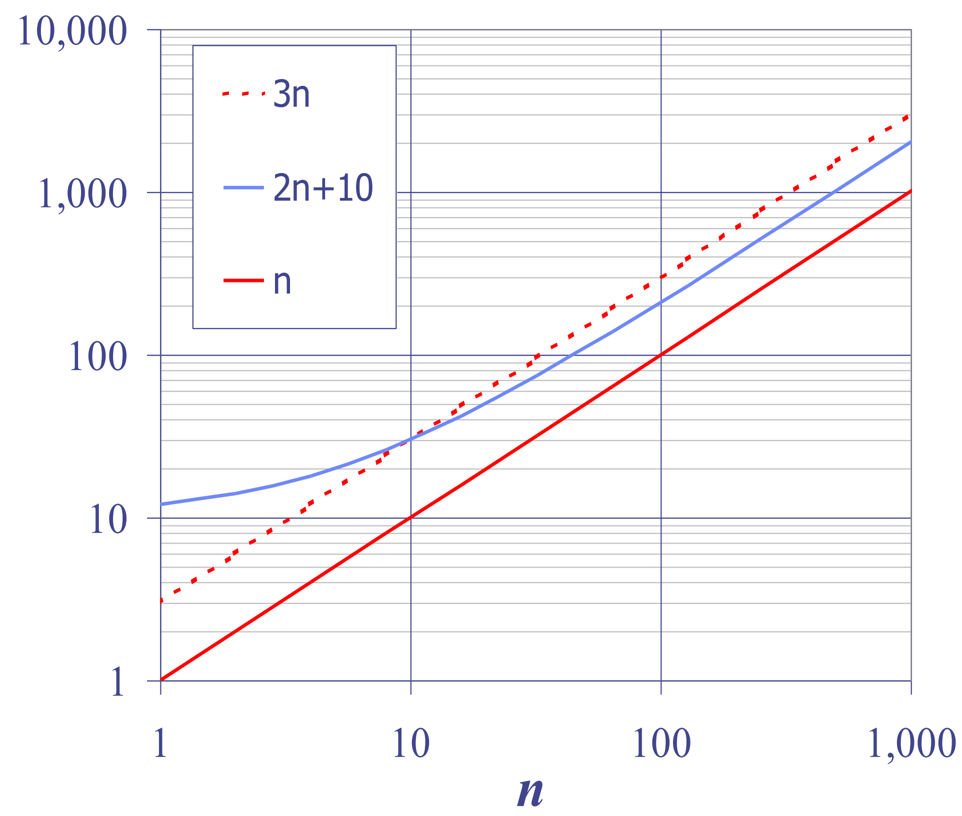


Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

  $$f(n) \leq c\, g(n) \text{ for } n \geq n_0$$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$

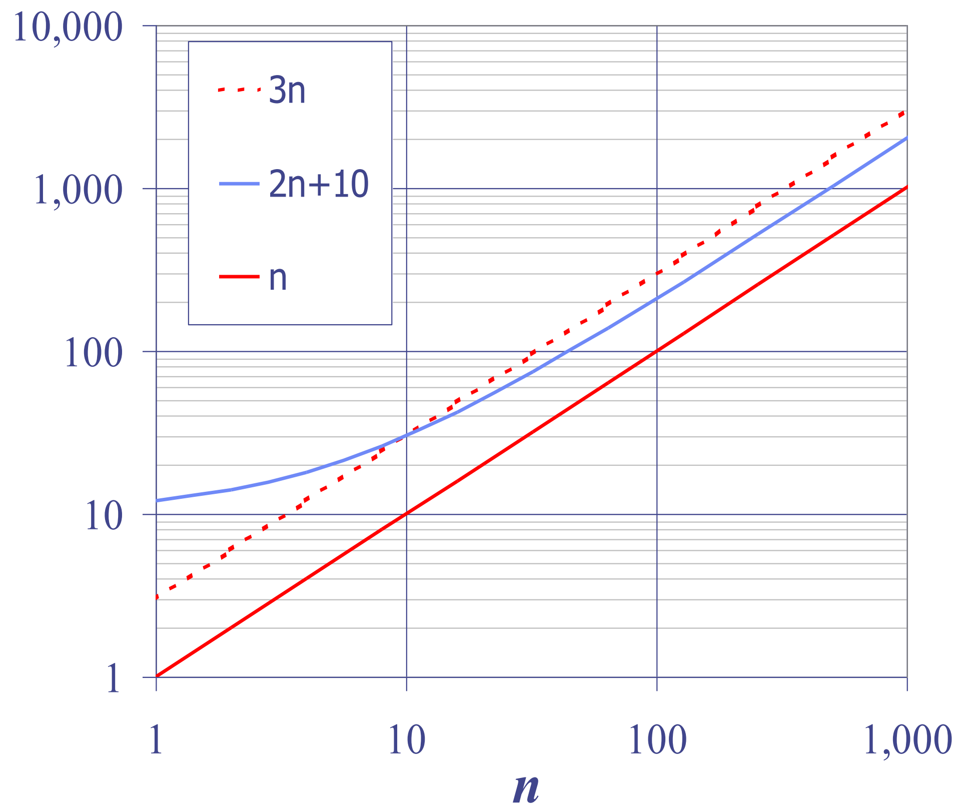


Big-Oh Notation

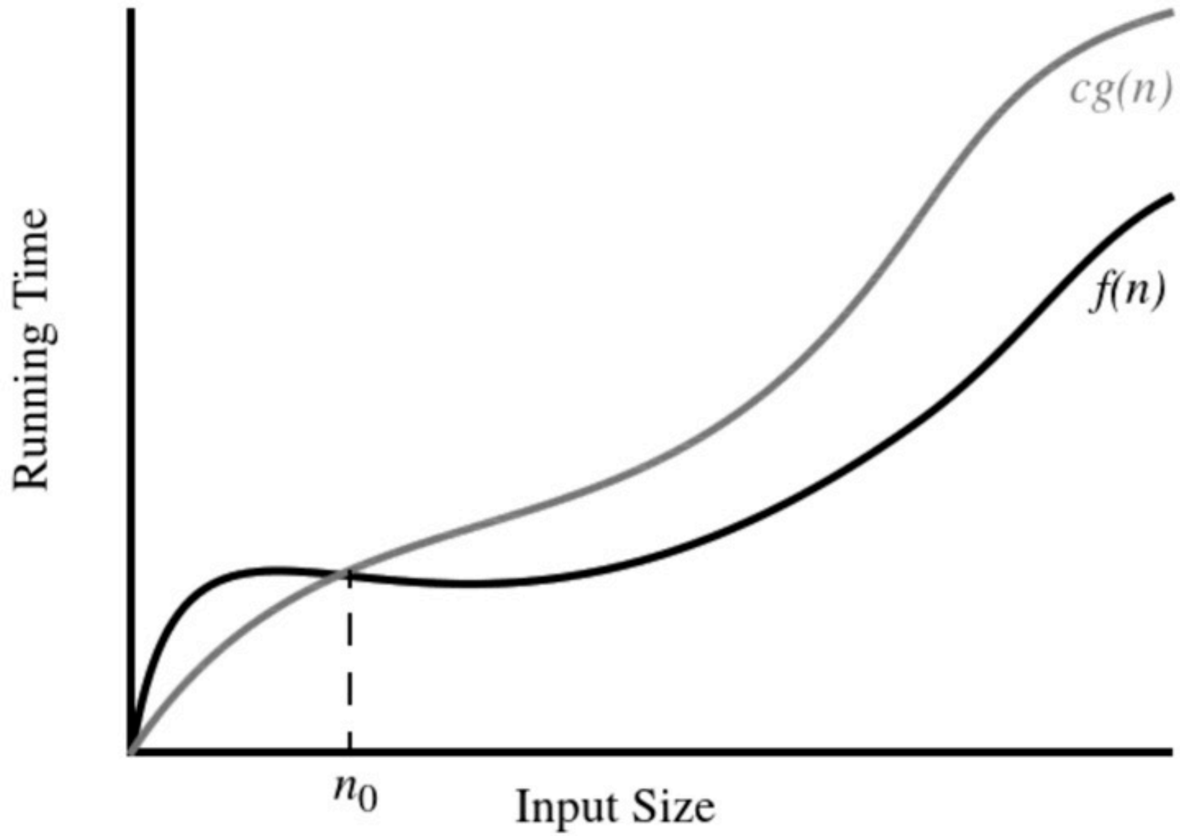
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

  $$f(n) \leq c\, g(n) \text{ for } n \geq n_0$$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)n \geq 10$
  - $n \geq \frac{10}{c-2}$
  - Pick $c = 3$ and $n_0 = 10$

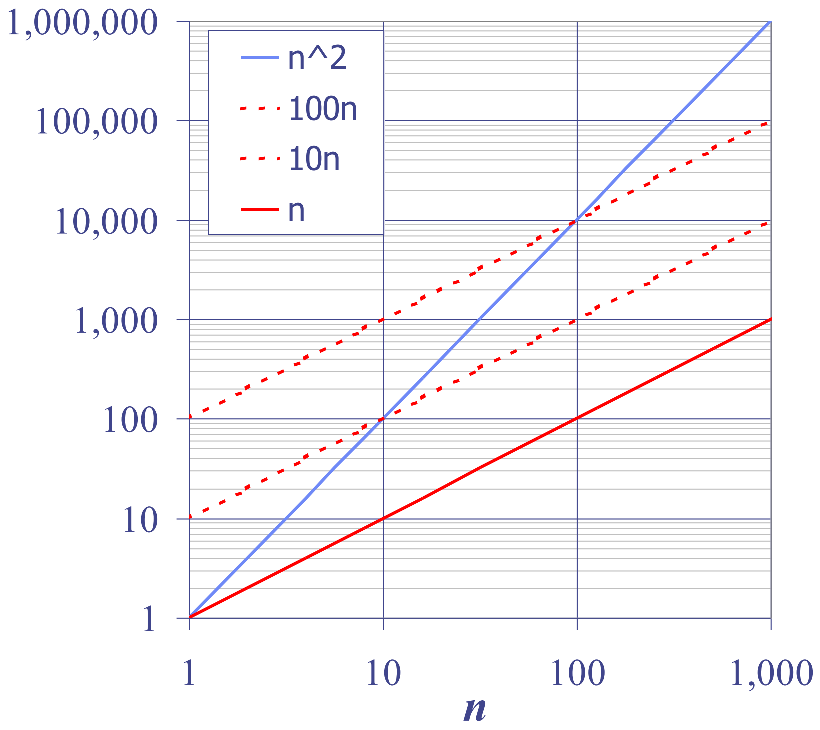


Big-Oh Notation



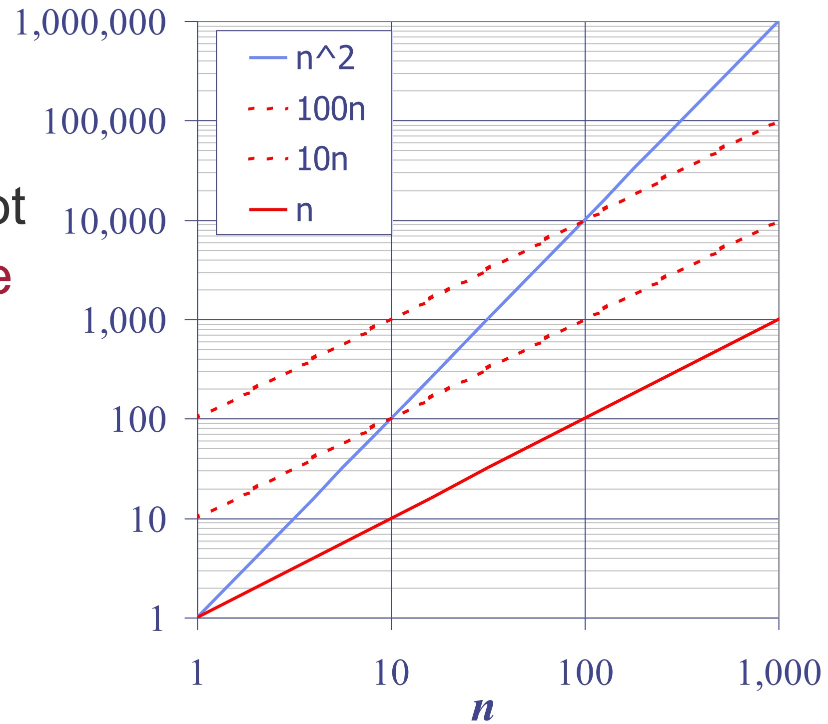
Big-Oh Example

- Example: n^2 is not $O(n)$



Big-Oh Example

- Example: n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant



More Big-Oh Examples

- $7n - 2$ is $O(n)$
- $3n^3 + 20n^2 + 5$ is $O(n^3)$
- $3 \log n + 5$ is $O(\log n)$

More Big-Oh Examples

- $7n - 2$ is $O(n)$
  - Need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq cn$ for $n \geq n_0$.
  - $7n - 2 \leq 7n - 2n \leq 5n$; this is true for $c = 5$ and $n_0 = 1$.

- $3n^3 + 20n^2 + 5$ is $O(n^3)$
  - Need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$
  - $3n^3 + 20n^2 + 5 \leq 3n^3 + 20n^3 + 5n^3 \leq (3 + 20 + 5)n^3$; this is true for $c = 28$ and $n_0 = 1$.

- $3 \log n + 5$ is $O(\log n)$
  - Need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \log n$ for $n \geq n_0$
  - $3 \log n + 5 \leq 8 \log n$; this is true for $c = 8$ and $n_0 = 2$ $(\log 1 = 0)$

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules

- If $f(n)$ is a polynomial of degree d, $f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ldots + a_d n^d$, then $f(n)$ is $O(n^d)$, i.e.
  - Drop lower-order terms
  - Drop constant factors

- Use the smallest possible class of functions
  - $2n$ is $O(n)$ instead of $2n$ is $O(n^2)$

- Use the simplest expression of the class
  - $3n + 5$ is $O(n)$ instead of $3n + 5$ is $O(3n)$

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We say that algorithm **find_max** "runs in $O(n)$ time"

```
def find_max(data):
  """Return the maximum element from a nonempty Python list."""
  biggest = data[0]                 # The initial value to beat
  for val in data:                  # For each value:
    if val > biggest                # if it is greater than the best so far,
      biggest = val                 # we have found a new best (so far)
  return biggest                    # When loop ends, biggest is the max
```

Example: Computing Prefix Averages

- Given a sequence S consisting of n numbers, compute a sequence A such that A$[j]$ is the average of elements $S[0], \dots, S[j]$, for $j = 0, \dots, n-1$:

$$A[j] = \frac{\sum_{i=0}^{j} S[i]}{j+1} = \frac{S[0] + S[1] + \cdots + S[j]}{j+1}$$

- $A[j]$ is the j-th prefix average of S

	0	1	2	3	4	5
S	20	10	3	3	14	4
A	20	15	11	9	10	9

Prefix Averages 1

- What is the running time of the following algorithm for computing prefix averages?

```
def prefix_average1(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                    # create new list of n zeros
  for j in range(n):
    total = 0                    # begin computing S[0] + ... + S[j]
    for i in range(j + 1):
      total += S[i]
    A[j] = total / (j+1)         # record the average
  return A
```

Prefix Averages 1: Analysis

	0	1	2	3	4	5
S	20	10	3	3	14	4
sum over how many elements?	1	2	3	4	5	6

- The running time of the algorithm is $O(1 + 2 + 3 + \cdots + n)$
- The sum of the first n integers is $\frac{n(n+1)}{2} = \frac{n^2+n}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$
- prefix averages 1 runs in $O(n^2)$ time

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Prefix Averages 2: Using sum()

- Use a Python function to simplify the code

```
def prefix_average2(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                           # create new list of n zeros
  for j in range(n):
    A[j] = sum(S[0:j+1]) / (j+1)        # record the average
  return A
```

Prefix Averages 3: Linear Time

- The following algorithm computes prefix averages by keeping a running sum

```
def prefix_average3(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                  # create new list of n zeros
  total = 0                    # compute prefix sum as S[0] + S[1] + ...
  for j in range(n):
    total += S[j]              # update prefix sum to include S[j]
    A[j] = total / (j+1)       # compute average based on current sum
  return A
```

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Prefix Averages 3: Linear Time

- The following algorithm computes prefix averages by keeping a running sum

```
def prefix_average3(S):
  """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
  n = len(S)
  A = [0] * n                    # create new list of n zeros
  total = 0                      # compute prefix sum as S[0] + S[1] + ...
  for j in range(n):
    total += S[j]                # update prefix sum to include S[j]
    A[j] = total / (j+1)         # compute average based on current sum
  return A
```

- This algorithm runs in $O(n)$ time

Relatives of Big-Oh

- big-Oh notation (O)
  - Provides an asymptotic way of saying that a function is “less than or equal to” another function
- big-Omega notation (Ω)
  - Provides an asymptotic way of saying that a function grows at a rate that is “greater than or equal to” that of another.
- big-Theta notation (Θ)
  - Allows us to say that two functions “grow at the same rate” up to constant factors

Big-Omega (Ω)

- Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers
- $f(n)$ is $\Omega(g(n))$ if $g(n)$ is $O(f(n))$, that is, there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \geq c\big(g(n)\big) \text{ for } n \geq n_0$$

- Example: Show that $3n \log n - 2n$ is $\Omega(n \log n)$.

Big-Omega (Ω)

- Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers
- $f(n)$ is $\Omega(g(n))$ if $g(n)$ is $O(f(n))$, that is, there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \geq c\big(g(n)\big) \text{ for } n \geq n_0$$

- Example: $3n \log n - 2n$ is $\Omega(n \log n)$
  - $3n \log n - 2n = n \log n + 2\, n \log n - 2n = n \log n + 2n(\log n - 1) \geq n \log n$ for $n \geq 2$; hence $c = 1$ and $n_0 = 2$.

Big-Theta (Θ)

- $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$, that is, there are real constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that

$$c' g(n) \leq f(n) \leq c'' g(n), \text{ for } n \geq n_0$$

- Example: Show that $3n \log n + 4n + 5 \log n$ is $\Theta(n \log n)$.

Big-Theta (Θ)

- $f(n)$ is $\Theta\big(g(n)\big)$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$, that is, there are real constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that

$$c'g(n) \leq f(n) \leq c''g(n), \text{ for } n \geq n_0$$

- Example: $3n \log n + 4n + 5 \log n$ is $\Theta(n \log n)$
  - $3n \log n \ \leq 3n \log n + 4n + 5 \log n \ \leq (3 + 4 + 5)n \log n$, for $n \geq 2$, hence $c' = 3, c'' = 12, n_0 = 2$.

Intuition for Asymptotic Notation

- big-Oh
  - $f(n)$ is $O(g(n))$ if $f(n)$ is asymptotically less than or equal to $g(n)$
- big-Omega
  - $f(n)$ is $\Omega(g(n))$ if $f(n)$ is asymptotically greater than or equal to g(n)
- big-Theta
  - $f(n)$ is $\Theta(g(n))$ if $f(n)$ is asymptotically equal to $g(n)$

Beware of Large Constants

- The function $f(n) = 10^{100}n$ is $O(n)$
- If we were to compare it to $10n \log n$, we should prefer the $O(n \log n)$-time algorithm, although the linear time algorithm is asymptotically faster
- 10^{100}= one googol
- If the asymptotic notations hide very large constants, they can be misleading

Is It Efficient?

- Any algorithm running in $O(n \log n)$ time (with a reasonable constant factor) should be considered efficient
- An $O(n^2)$ algorithm may be fast in some contexts
- An algorithm running in $O(2^n)$ time should never be considered efficient

More Examples of Algorithm Analysis

- `len(data), data[j]` – where data is an instance of Python's `list` class - constant-time operations, both run in $O(1)$ time

Three Way Disjointness

- Suppose three sequences of numbers, A, B and C;
- no individual sequence contains duplicate values – but there may be some numbers that are in two or three of the sequences
- Determine if the intersection of the three sequences in empty – namely - that there is no element x such that $x \in A, x \in B$ and $x \in C$

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Three-Way Set Disjointness

```
def disjoint1(A, B, C):
  """Return True if there is no element common to all three lists."""
  for a in A:
    for b in B:
      for c in C:
        if a == b == c:
          return False          # we found a common value
  return True                   # if we reach this, sets are disjoint
```

Three-Way Set Disjointness

```
def disjoint1(A, B, C):
  """Return True if there is no element common to all three lists."""
  for a in A:
    for b in B:
      for c in C:
        if a == b == c:
          return False        # we found a common value
  return True                 # if we reach this, sets are disjoint
```

- Worst-case running time is $O(n^3)$, because it loops through each possible triple of values from the three sets to see if the values are equivalent

Three-Way Set Disjointness: Take 2

- Observation: once inside the body of the loop over B, if selected elements a and b do not match each other, it don't make sense to iterate through the values of C looking for a matching triple

```
def disjoint2(A, B, C):
  """Return True if there is no element common to all three lists."""
  for a in A:
    for b in B:
      if a == b:              # only check C if we found match from A and B
        for c in C:
          if a == c           # (and thus a == b == c)
            return False      # we found a common value
  return True                 # if we reach this, sets are disjoint
```

Three-Way Set Disjointness: Take 2

- Observation: once inside the body of the loop over B, if selected elements a and b do not match each other, it don't make sense to iterate through the values of C looking for a matching triple

```
def disjoint2(A, B, C):
  """Return True if there is no element common to all three lists."""
  for a in A:
    for b in B:
      if a == b:              # only check C if we found match from A and B
        for c in C:
          if a == c           # (and thus a == b == c)
            return False      # we found a common value
  return True                 # if we reach this, sets are disjoint
```

- Worst-case running time is $O(n^2)$

Element Uniqueness

- Given a sequence S with n elements, are all elements distinct from each other?

```
def unique1(S):
  """Return True if there are no duplicate elements in sequence S."""
  for j in range(len(S)):
    for k in range(j+1, len(S)):
      if S[j] == S[k]:
        return False                    # found duplicate pair
  return True                           # if we reach this, elements were unique
```

Element Uniqueness

- Given a sequence S with n elements, are all elements distinct from each other?

```
def unique1(S):
  """Return True if there are no duplicate elements in sequence S."""
  for j in range(len(S)):
    for k in range(j+1, len(S)):
      if S[j] == S[k]:
        return False                    # found duplicate pair
  return True                           # if we reach this, elements were unique
```

outer loop, j	0	1	2	…	n-2	n-1
inner loop, k	n-1	n-2	n-3		1	0

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Element Uniqueness

- Given a sequence S with n elements, are all elements distinct from each other?

```
def unique1(S):
  """Return True if there are no duplicate elements in sequence S."""
  for j in range(len(S)):
    for k in range(j+1, len(S)):
      if S[j] == S[k]:
        return False              # found duplicate pair
  return True                     # if we reach this, elements were unique
```

outer loop, j	0	1	2	…	n-2	n-1
inner loop, k	n-1	n-2	n-3		1	0

- $(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}$;
- worst-case running time proportional to $O(n^2)$

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Element Uniqueness: Using Sorting

- Idea: sort the sequence first; any duplicates are then guaranteed to be next to each other

```
def unique2(S):
  """Return True if there are no duplicate elements in sequence S."""
  temp = sorted(S)                    # create a sorted copy of S
  for j in range(1, len(temp)):
    if S[j−1] == S[j]:
      return False                    # found duplicate pair
  return True                         # if we reach this, elements were unique
```

Element Uniqueness: Using Sorting

- Idea: sort the sequence first; any duplicates are then guaranteed to be next to each other

```
def unique2(S):
  """Return True if there are no duplicate elements in sequence S."""
  temp = sorted(S)                # create a sorted copy of S
  for j in range(1, len(temp)):
    if S[j−1] == S[j]:
      return False                # found duplicate pair
  return True                     # if we reach this, elements were unique
```

- Sorting: $O(n \log n)$ - details next week
- Once the sequence is sorted, a single loop is needed to find duplicates – which runs in $O(n)$ time
- Therefore the entire algorithm runs in $O(n \log n)$. Better?

$O(n \log n)$ better than $O(n^2)$

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Binary Search (review from Java 2)

- One of the most important computer algorithms
- Locate a target value within a sorted sequence of n elements
- If the sequence is unsorted, the standard approach is to use a loop to examine each element – sequential search, linear time, $O(n)$
- If the sequence is sorted and indexable, there is a much more efficient algorithm
- Intuition: think of how you look up a word in a dictionary
  - Open at a certain page; if the word is on that page, stop
  - if word should be before in lexicographic order, continue looking in the first half
  - Otherwise continue looking in the second half

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Binary Search

```
def binary_search(data, target, low, high):
  """Return True if target is found in indicated portion of a Python list.

  The search only considers the portion from data[low] to data[high] inclusive.
  """
  if low > high:
    return False                          # interval is empty; no match
  else:
    mid = (low + high) // 2
    if target == data[mid]:               # found a match
      return True
    elif target < data[mid]:
      # recur on the portion left of the middle
      return binary_search(data, target, low, mid − 1)
    else:
      # recur on the portion right of the middle
      return binary_search(data, target, mid + 1, high)
```

Binary Search: Analysis

- Proposition: The binary search algorithm runs in $O(\log n)$ time for a sorted sequence with n elements.
- Justification
  - With each recursive call the number of candidate entries still to be searched is given by the value $high - low + 1$
  - The number of remaining candidates is reduced by at least one half with each recursive call
  - Initially, $low = 0,\ high = n - 1, mid = \lfloor (low + high)/2 \rfloor$
  - The number of candidates to be searched at the next recursive call is either
    - $(mid - 1) - low + 1 = \left\lfloor \frac{low+high}{2} \right\rfloor - \mathrm{low} \leq \frac{high - low + 1}{2}$

    or
    - $high - (mid + 1) + 1 = \mathrm{high} - \left\lfloor \frac{low+high}{2} \right\rfloor \leq \frac{high - low + 1}{2}$

Binary Search: Analysis (cont'd)

- The initial number of candidates is n;
- After the 1st call in a binary search, it is at most $\frac{n}{2} = \frac{n}{2^1}$
- After the 2nd call, it is at most $\frac{n}{4} = \frac{n}{2^2}$
- In general, after the jth call, it is at most $\frac{n}{2^j}$
- In the worst case (target not found), binary search stops when there are no more candidate entries
- The maximum number of recursive calls is the smallest integer such that $\frac{n}{2^r} < 1$, therefore $r > \log_2 n$
- Thus $r = \lfloor \log_2 n \rfloor + 1$, so binary search runs in $O(\log_2 n)$ time.

Binary Search: Analysis (cont'd)

- $O(\log n)$ binary search - much better than $O(n)$ sequential search
- Think for $n = 1{,}000{,}000{,}000$
- $O(\log n\ ) \approx 29.897$

Math You May Need to Review

- Summations
- Logarithms and Exponents
- See Appendix B.
- Extra resource:
  - https://www.khanacademy.org/math/algebra2/x2ec2f6f830c9fb89:logs

- Properties of logarithms
  - $\log_b xy = \log_b x + \log_b y$
  - $\log_b \frac{x}{y} = \log_b x - \log_b y$
  - $\log_b x^a = a \log_b x$
  - $\log_b a = \frac{\log_x a}{\log_x b}$

- Properties of exponentials
  - $a^{b+c} = a^b a^c$
  - $a^{bc} = (a^b)^c$
  - $\frac{a^b}{a^c} = a^{b-c}$
  - $b^{\log_c a} = a^{\log_c b}$

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Thank you.