Data structure and algorithm in Python

Algorithm Analysis

Xiaoping Zhang

School of Mathematics and Statistics, Wuhan University

Table of contents

- **data structure** is a systematic way of organizing and accessing dat;
- **algorithm** is a step-by-step procedure for performing some task in a finite amout of time.

The primary analysis tool involves characterizing the running time of algorithms and data structure algorithms, with space usage also being of interest.

Running time is a natural measure of “goodness”, since time is a precious time - computer solutions should run as fast as possible.

- increases with the input size
- be affected by the hardware environment (e.g., the processor, clock rate, memory, disk) and software environment (e.g., the operating system, programming language).

Experimental studies

Experimental studies

Elapsed time

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

Experimental studies

- The `time` function in `time` module
- The `clock` function in `time` module
- The module `timeit`

Challenges of Experimental Analysis

- Experiments should be performed in the same hardware and software environments.
- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).
- An algorithm must be fully implemented in order to execute it to study its running time experimentally.

Experimental studies

Moving Beyond Experimental Analysis

Moving Beyond Experimental Analysis

Our goal is to develop an approach to analyzing the efficiency of algorithms that:

- Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.
- Is performed by studying a high-level description of the algorithm without need for implementation.
- Takes into account all possible inputs.

Counting Primitive Operations

Define a set of primitive operations:

- Assigning an identifier to an object
- Determining the object associated with an identifier
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Accessing a single element of a Python list by index
- Calling a function (excluding operations executed within the function)
- Returning from a function.

Formally, a primitive operation corresponds to a low-level instruction with an execution time that is constant.

Counting Primitive Operations

Instead of trying to determine the specific execution time of each primitive operation, we will simply count how many primitive operations are executed, and use this number t as a measure of the running time of the algorithm.

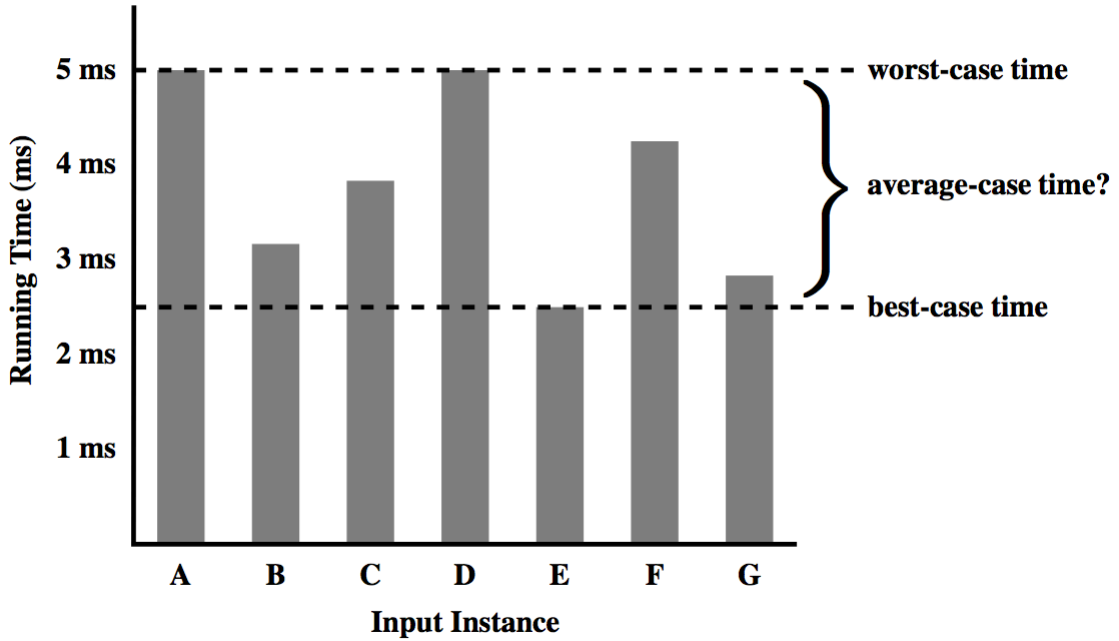
The implicit assumption of "operation count" is the running times of different primitive operations will be fairly similar. Thus, the number, t, of primitive operations an algorithm performs will be proportional to the actual running time of that algorithm.

Measuring Operations as a Function of Input Size

To capture the order of growth of an algorithm's running time, we will 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.

Focusing on the Worst-Case Input

- Average-case analysis
  - quite challenging
  - requires to define a probability distribution on the set of inputs
- Worst-case analysis
  - much easier
  - requires only the ability to identify the worst-case input, which is often simple
  - typically leads to better algorithms



The Seven Functions used in the analysis of algorithms

The Seven Functions used in the analysis of algorithms

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

The Seven Functions used in the analysis of algorithms

- Ideally, we would like data structure operations to run in times proportional to the constant or logarithm function, and we would like our algorithms to run in linear or n-log-n time;
- Algorithms with quadratic or cubic running times are less practical;
- Algorithms with exponential running times are infeasible for all but the smallest sized inputs.

The Seven Functions used in the analysis of algorithms

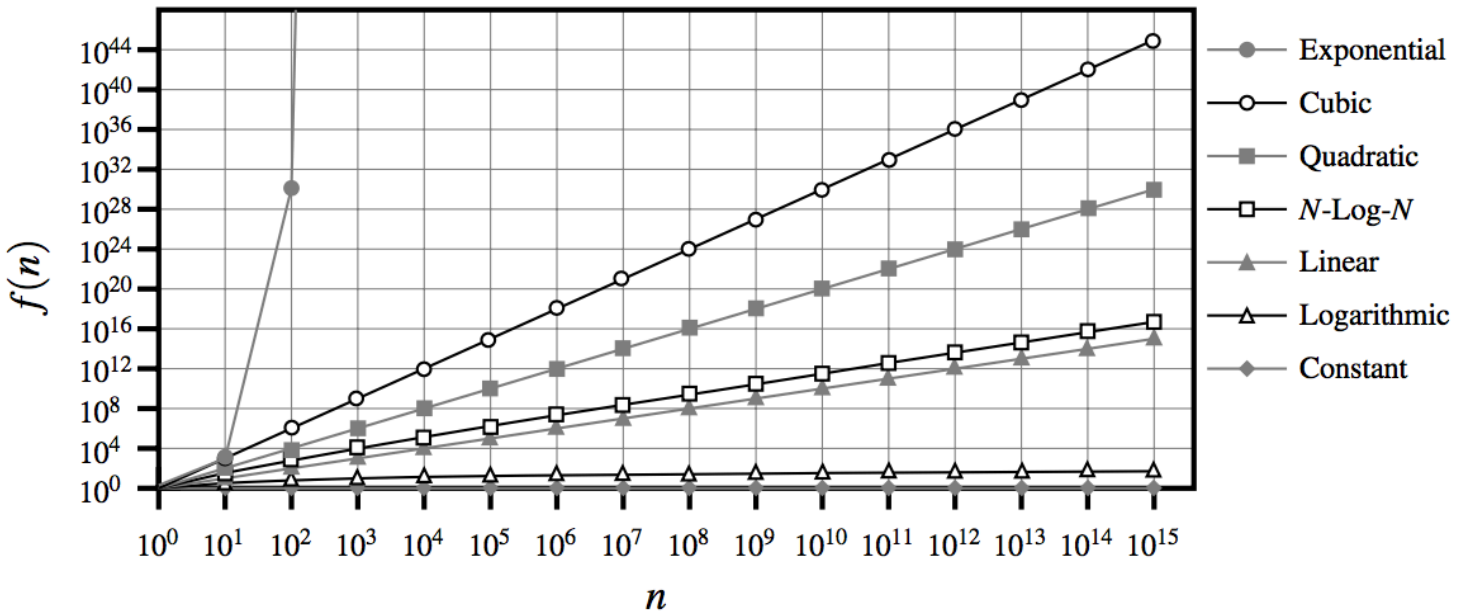


Figure 3.4: Growth rates for the seven fundamental functions used in algorithm analysis. We use base $a = 2$ for the exponential function. The functions are plotted on a log-log chart, to compare the growth rates primarily as slopes. Even so, the exponential function grows too fast to display all its values on the chart.

Asymptotic Analysis

Asymptotic Analysis

In algorithm analysis, we focus on the growth rate of the running time as a function of the input size n, taking a "big-picture" approach.

```
def find_max(data):
  """Return the maximum element from a nonempty
  Python list."""
  biggest = data[0]
  for val in data:
    if val > biggest
    biggest = val
  return biggest
```

Asymptotic Analysis

The “Big-Oh” Notation

The "Big-Oh" Notation

Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. 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

$$f(n) \leq cg(n), \quad \text{for} \quad n \geq n_0.$$

This definition is often referred to as the "big-Oh" notation, for it is sometimes pronounced as "$f(n)$ is big-Oh of $g(n)$".

The "Big-Oh" Notation

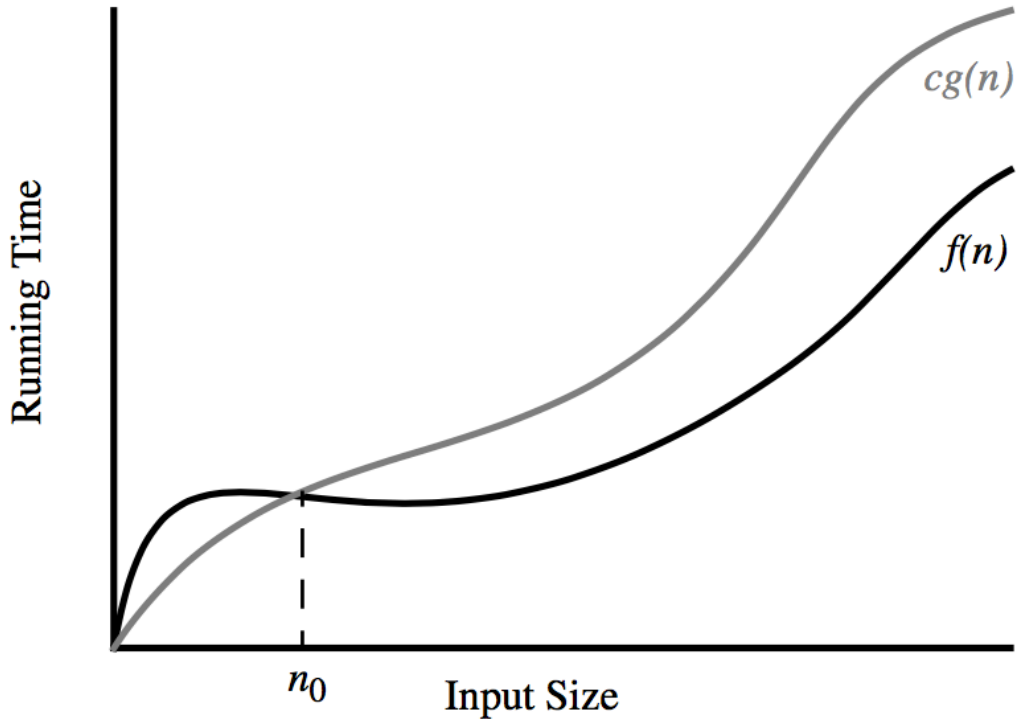


Figure 3.5: Illustrating the "big-Oh" notation. The function $f(n)$ is $O(g(n))$, since $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

Some Properties of the Big-Oh Notation

The big-Oh notation allows us to ignore constant factors and lower-order terms and focus on the main components of a function that affect its growth.

Example

$5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$.

Proposition

If $f(n)$ is a polynomial of degree d, that is,

$$f(n) = a_0 + a_1 n + \cdots + a_d n^d,$$

and $a_d > 0$, then $f(n)$ is $O(n^d)$.

Some Properties of the Big-Oh Notation

Example

$5n^2 + 3n\log n + 2n + 5$ is $O(n^2)$.

Example

$20n^3 + 10n\log n + 5$ is $O(n^3)$.

Example

$3\log n + 2$ is $O(\log n)$.

Example

2^{n+2} is $O(2^n)$.

Example

$2n + 100\log n$ is $O(n)$.

Asymptotic Analysis

Comparative Analysis

Comparative Analysis

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}

Table 3.2: Selected values of fundamental functions in algorithm analysis.

Comparative Analysis

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

Table 3.3: Maximum size of a problem that can be solved in 1 second, 1 minute, and 1 hour, for various running times measured in microseconds.

Asymptotic Analysis

Examples of Algorithm Analysis

Constant-Time Operations

Example

Given an instance, named data, of the Python list class, a call to the function, `len(data)`, is evaluated in constant time.

This is a very simple algorithm because the list class maintains, for each list, an instance variable that records the current length of the list.

Constant-Time Operations

Example

The expression `data[j]` is evaluated in $O(1)$ time for a Python list.

Because Python's lists are implemented as "array-based sequences", references to a list's elements are stored in a consecutive block of memory. The j^{th} element of the list can be found, not by iterating through the list one element at a time, but by validating the index, and using it as an offset into the underlying array. In turn, computer hardware supports constant-time access to an element based on its memory address.

Linear-Time Operations

the Problem of Finding the Maximum of a Sequence

```
def find_max(data):
  """Return the maximum element from a nonempty
  Python list."""
  biggest = data[0]
  for val in data:
    if val > biggest
    biggest = val
  return biggest
```

- Initialization uses $O(1)$ time
- The loop executes n times, and within each iteration, it performs one comparison and possibly one assignment statement (as well as maintenance of the loop variable).
- Enacting a return statement in Python uses $O(1)$ time.

Prefix Average

Example : Prefix Average

Given a sequence S consisting of n numbers, we want to compute a sequence A such that $A[j]$ is the average of elements $S[0],...,S[j]$, for $j = 0,...,n1$, that is,

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

Prefix Average: Quadratic-Time Algorithm

```
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
  for j in range(n):
    total = 0
    for i in range(j + 1):
      total += S[i]
    A[j] = total / (j+1)
  return A
```

The running time of `prefix_average1` is $O(n^2)$

Prefix Average: Another Quadratic-Time Algorithm

```
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
  for j in range(n):
    A[j] = sum(S[0:j+1]) / (j+1)
  return A
```

The running time of `prefix_average2` is $O(n^2)$

Prefix Average: A Linear-Time Algorithm

```
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
  total = 0
  for j in range(n):
    total += S[j]
    A[j] = total / (j+1)
  return A
```

The running time of `prefix_average3` is $O(n)$

Three-Way Set Disjointness

Definition : Three-Way Set Disjointness

Given three sequences of numbers: A, B, and C. Assume that no individual sequence contains duplicate values, but that there may be some numbers that are in two or three of the sequences. The three-way set disjointness problem is to determine if the intersection of the three sequences is empty, namely, that there is no element x such that $x \in A, x \in B$, and $x \in C$.

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
  return True
```

If each of the original sets has size n, then the worst-case running time of this function is $O(n^3)$.

Three-Way Set Disjointness

```
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:
        for c in C:
          if a == c
            return False
  return True
```

The running time of `disjoint2` is $O(n^2)$

Element Uniqueness

Definition : Element Uniqueness

Given a single sequence S with n elements, whether all elements of that collection are distinct from each other?

Element Uniqueness

```
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
  return True
```

The running time of `unique1` is $O(n^2)$

Element Uniqueness

```
def unique2(S):
  """Return True if there are no duplicate
  elements in sequence S."""
  temp = sorted(S)
  for j in range(1, len(temp)):
    if S[j-1] == S[j]:
      return False
  return True
```

The running time of unique2 is $O(n \log n)$