complexity_analysis

August 28, 2021

1 The Analysis of Algorithims

- Primary analysis tool
  - characterizing the running times of algorithms and data structure operations
  - Space usage

- Running time is a natural measure of "goodness,"
  - time is a precious resource
  - computer solutions should run as fast as possible

Motivation

- Sometimes it is necessary to have a precise understanding of the complexity of an algorithm.
- In order to obtain this understanding we could proceed as follows:

  We implement the algorithm in a given programming language.

  We count how many additions, multiplications, assignments, etc.~are needed for an input of a given size.

  Additionally, we have to count all storage accesses.

  We look up the amount of time that is needed for the different operations in the processor handbook.

  Using the information discovered in the previous two steps we predict the running time of our algorithm for given input.

1.0.1 Experimental Studies

```
from time import time

start time = time( ) # record the starting time
run algorithm
end time = time( )   # record the ending time
elapsed = end time - start time # compute the elapsed time
```

1.0.2 Naive Approach

$S = 1 + 2 + 3 + ... + n$

[32]:
```
import time


def sum_of_n_2(n):
    start = time.time()

    the_sum = 0
    for i in range(1, n + 1):
        the_sum = the_sum + i

    end = time.time()

    return the_sum, end - start

for i in range(5):
    print("Sum is %d required %10.6f seconds" % sum_of_n_2(10000))
```

```
Sum is 50005000 required   0.000954 seconds
Sum is 50005000 required   0.000987 seconds
Sum is 50005000 required   0.000973 seconds
Sum is 50005000 required   0.000983 seconds
Sum is 50005000 required   0.001019 seconds
```

1.0.3 Alternative Approach

$S = 1 + 2 + 3 + ... + n$

$S = n + (n-1) + (n-2) + ... + 1$

$2S = (n+1) + (n+1) + ... + (n+1) = n * (n+1)$

$S = \frac{n*(n+1)}{2}$

[33]:
```
def sum_of_n_3(n):
    start = time.time()

    the_sum = ((n+1)*n)/2.0

    end = time.time()

    return the_sum, end - start

for i in range(5):
    print("V2: Sum is %d required %10.6f seconds" % sum_of_n_3(10000))
```

```
V2: Sum is 50005000 required   0.000002 seconds
V2: Sum is 50005000 required   0.000001 seconds
V2: Sum is 50005000 required   0.000000 seconds
```

```
V2: Sum is 50005000 required   0.000001 seconds
V2: Sum is 50005000 required   0.000001 seconds
```

1.1 Moving Beyond Experimental Analysis

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

1.1.1 Counting Primitive Operations

- Perform analysis directly on a high-level description of the algorithm
- Identify and analyse primitive operations - **a primitive operation is low-level instruction with an execution time that is constant** e.g.,
  - 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.
- A primitive operation is basic operation that is executed by the hardware
- Many of our primitive operations may be translated to a small number of instructions.
- Instead of trying to determine the specific execution time of each primitive operation, we will count
  - how many primitive operations are executed
  - use this number t as a measure of the running time of the algorithm.

1.1.2 Measuring Operations as a Function of Input Size

- Capture the order of growth of an algorithm's running time
- A function $f(n)$ that characterizes the number of primitive operations that are performed as a function of the input size n

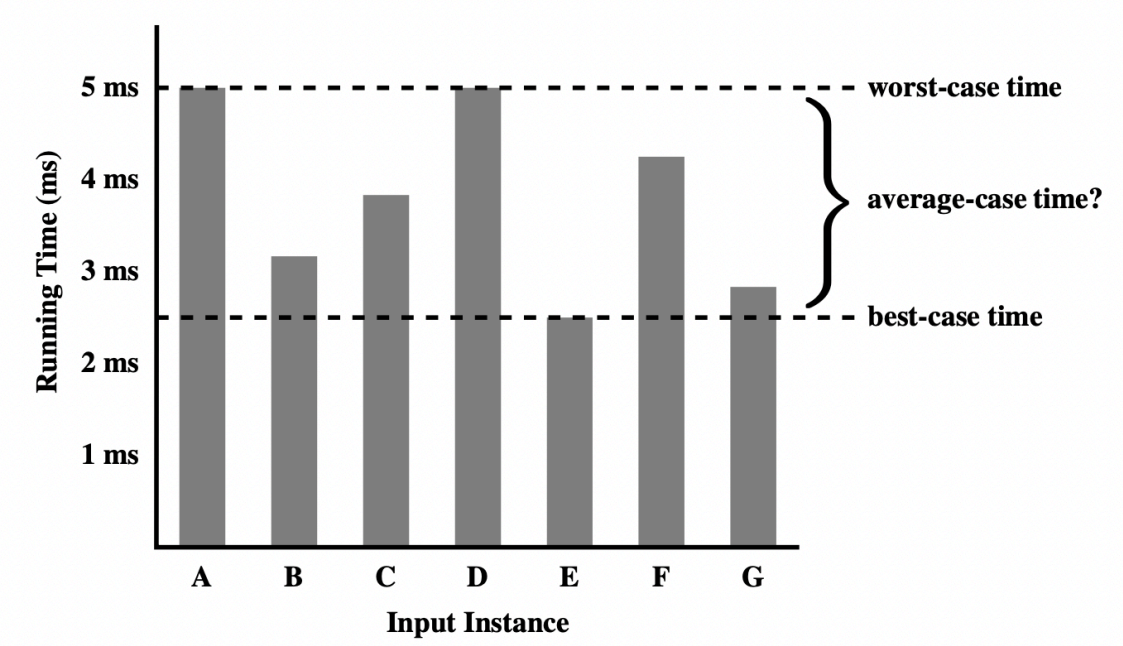
1.1.3 Focusing on the Worst-Case Input

- **Worst-case:** $T(n)$ The maximum steps/time/memory taken by an algorithm to complete the solution for inputs of size n
- **Average-case:** The expected time over all inputs with the same characteristics (e.g. size).
- **Best-case:** The performance in ideal situations. Rarely used.

Worst-case analysis is much easier than average-case analysis, as it requires only the ability to identify the worst-case input, which is often simple.

When analysing algorithmic performance, we ignore computational resources (e.g. CPU speed); we only care about the growth of $T(n)$ as $n \to \infty$

This is called the **asymptotic behaviour**



1.2 Growth Function

1. $f(x) = n$ (constant function)

- for any argument n, the constant function $f(n)$ assigns the value c.

2. $f(n) = log_b(n)$ (logarithmic function)

- $x = \log_b n$ if and only if $b^x = n$.

3. $f(x) = sqrt(x)$ (square root function)
4. $f(x) = x$ (linear function)
5. $f(x) = nlog(n)$ (log-linear function)
6. $f(x) = x^2$ (quadratic function)
7. $f(x) = x^3$ (cubic function)
8. $f(x) = a^x, \forall a > 1$ (exponential function)

1.3 Math Review

1.3.1 Exponentiation

Exponents have a number of important properties, including:

$$b^0 = 1 \tag{1}$$
$$b^1 = b \tag{2}$$
$$b^{1/2} = b^{0.5} = \sqrt{b} \tag{3}$$
$$b^{-1} = \frac{1}{b} \tag{4}$$
$$b^a \cdot b^c = b^{a+c} \tag{5}$$
$$(b^a)^c = b^{ac} \tag{6}$$

1.3.2 Logarithms

Definition: $\log_b a = c$ if and only if $a = b^c$. If $b = 2$, we write $\lg a$.

There are a number of log identities you should be aware of:

$$\log_b(a \cdot c) = \log_b a + \log_b c \tag{7}$$
$$\log\left(\frac{a}{c}\right) = \log_b a - \log_b c \tag{8}$$
$$\log(a^c) = c \cdot \log_b a \tag{9}$$
$$b^{\log_c a} = a^{\log_c b} \tag{10}$$
$$\log_b a = \frac{\log_c a}{\log_c b} \tag{11}$$

1.3.3 Recursive Definitions

Factorial - The **factorial** of a number n is represented by $n!$. Informally, $n! = 1{\cdot}2{\cdot}3{\cdot}\ldots{\cdot}n$. More formally:

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

1.3.4 Recursive Definitions

Fibonacci numbers The **fibonacci numbers** are defined by:

$$F_i = \begin{cases} 0 & i = 0 \\ 1 & i = 1 \\ F_{i-2} + F_{i-1} & i > 1 \end{cases}$$

1.3.5 Summations

Arithmetic Series

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$
$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$
$$\sum_{i=1}^{n} i^k \approx \frac{n^{k+1}}{k+1} = \Theta(n^{k+1})$$

1.3.6 Summations

- **Geometric series**

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + a^3 + ... + a^n = \frac{a^{n+1} - 1}{a - 1}$$
$$\sum_{i=0}^{\infty} a^i = \frac{1}{1 - a} \text{ where } a < 1$$

1.3.7 Polynomials

A polynomial function has the form

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdot \cdot \cdot + a_d n^d$$

where a_0 , a_1 , . . . , a_d are constants, called the **coefficients** of the polynomial, and $a^d \neq 0$.

Integer d, which indicates the highest power in the polynomial, is called the **degree** of the polynomial.

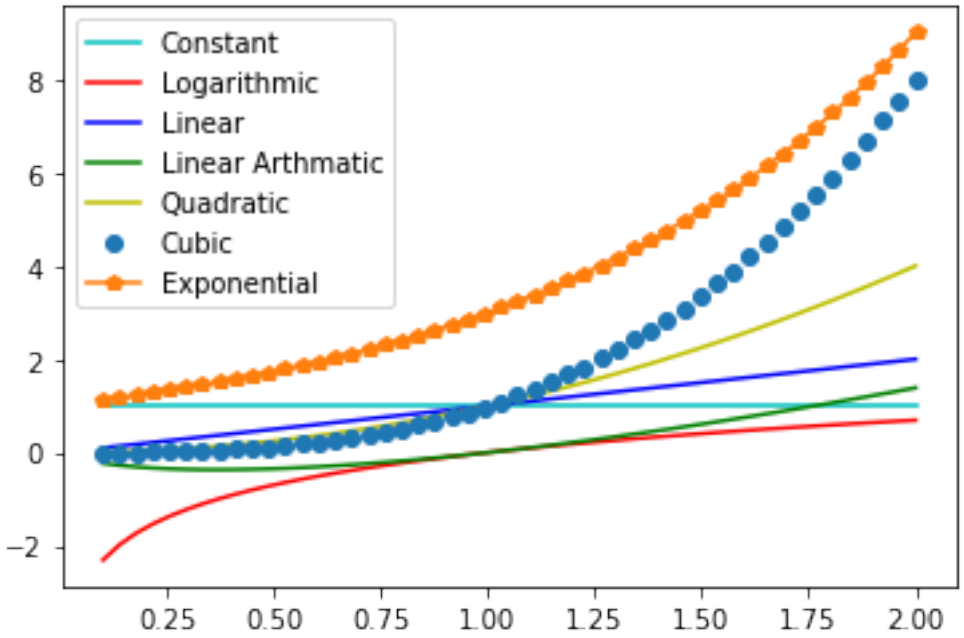
1.3.8 Comparing Growth Rates

[34]:
```
# remember to evaluate this cell first!
import matplotlib.pylab as plt
import numpy as np
import math
import timeit
import random
%matplotlib inline
```

[41]:
```
count = 50
xs = np.linspace(0.1, 2, count)
ys_const        = [1] * count
ys_log          = [math.log(x) for x in xs]
```

```
ys_linear       = [x for x in xs]
ys_linearithmic = [x * math.log(x) for x in xs]
ys_quadratic    = [x**2 for x in xs]
ys_cubic    = [x**3 for x in xs]
ys_exp    = [3**x for x in xs]
plt.plot(xs, ys_const, 'c', label='Constant')
plt.plot(xs, ys_log, 'r', label = 'Logarithmic')
plt.plot(xs, ys_linear, 'b', label='Linear')
plt.plot(xs, ys_linearithmic, 'g', label='Linear Arthmatic')
plt.plot(xs, ys_quadratic, 'y', label='Quadratic');
plt.plot(xs, ys_cubic, 'o', label='Cubic');
plt.plot(xs, ys_exp, '-p', label='Exponential');
plt.legend()
```

[41]: <matplotlib.legend.Legend at 0x10fdeca30>



constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	$log(n)$	n	$nlogn$	n^2	n^3	a^n

The Ceiling and Floor Functions

- $\lceil x \rceil$ = the largest integer less than or equal to x.
- $\lfloor x \rfloor$ = the smallest integer greater than or equal to x.

2 3.3 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.
- For example, it is often enough just to know that the running time of an algorithm **grows proportionally** to n.

A function that returns the maximum value of a Python list.

```
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
```

f(n) = 3n^3+2n+10

This is a classic example of an algorithm with a running time that grows **proportional to** n, as the loop executes once for each data element, with some fixed number of primitive operations executing for each pass.

2.1 3.3.1 The "Big-Oh" Notation

- 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

$$f(n) \leq cg(n)$$

for $n \geq n_0$.

Example : The function $8n + 5$ is $O(n)$. Justification: - By the big-Oh definition, we need to find a real constant $c > 0$ and an integer constant $n0 \geq 1$ such that $8n+5 \leq cn$ for every integer $n \geq n_0$. - It is easy to see that a possible choice is $c = 9$ and $n_0 = 5$. - Indeed, this is one of infinitely many choices available because there is a trade-off between c and n_0. - For example, we could rely on constants $c = 13$ and $n_0 = 1$.

Proposition: The algorithm, find max, for computing the maximum element of a list of n numbers, runs in O(n) time. Justification

- The initialization before the loop begins requires only a constant number of primitive operations.
- Each iteration of the loop also requires only a con- stant number of primitive operations.
- The loop executes n times.
- The number of primitive operations being $c' + c".n$ for appropriate constants c' and $c"$

Example: $5n^4+3n^3+2n^2+4n+1$ **is** $O(n4)$. **Justification**: Note that $5n^4+3n^3+2n^2+4n+1 \leq (5+3+2+4+1)n^4 = cn^4$, for $c = 15$,when $n \geq n_0 = 1$.

Thus, the highest-degree term in a polynomial is the term that determines the asymptotic growth rate of that polynomial

Example: $5n^2 + 3nlogn + 2n + 5$ is $O(n^2)$.

Example: $20n^3 + 10nlogn + 5$ is $O(n^3)$.

Example: $2^n + 2$ is $O(2^n)$.

2.1.1 3.3.2 Big-Omega Notation

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

$$f(n) \geq cg(n)$$

for $n \geq n_0$.

Example: $3nlogn-2n$ **is** $\Omega(nlogn)$**.** **Justification**: $3nlogn-2n = nlogn + 2n(logn-1) \geq nlogn$ for $n \geq 2$; hence, we can take $c = 1$ and $n_0 = 2$ in this case.

2.1.2 3.3.3 Big-Theta Notation

- Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers.
- We say that $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$
- if there is two real constant $c_1 > 0$ and $c_2 > 0$ and
- an integer constant $n_0 \geq 1$ such that

$$c_1g(n) \leq f(n) \geq c_2g(n)$$

for $n \geq n_0$.

Example: $3nlogn + 4n + 5logn$ **is** $\Theta(nlogn)$**.** **Justification**: $3nlogn \leq 3nlogn + 4n + 5logn \leq (3 + 4 + 5)nlogn$ for $n \geq 2$

2.1.3 Examples of Algorithm Analysis

Element Uniqueness: We are given a single sequence S with n elements and asked whether all elements of that collection are 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
```

[36]:
```
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
```

[37]:
```
S1=[19, 2,3,4,12,34]
S2=[18,23,78,23,100]

print(unique1(S1))
print(unique1(S2))
```

```
True
False
```

```
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 temp[j-1] == temp[j]:
      return False                # found duplicate pair
  return True                     # if we reach this, elements were unique
```

[38]:
```
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 temp[j-1] == temp[j]:
      return False                # found duplicate pair
  return True                     # if we reach this, elements were unique
```

[39]:
```
S1=[19, 2,3,4,12,34]
S2=[18,23,78,23,100]

print(unique2(S1))
print(unique2(S2))
```

```
True
False
```

[40]:
```
sorted(S1)
```

[40]:
```
[2, 3, 4, 12, 19, 34]
```

2.2 Simple Justification Techniques

1. **Counter Example**: - Some claims are of the generic form, “There is an element x in a set S that has property P.” - To justify such a claim, we only need to produce a particular x in S

that has property P. - Likewise, some hard-to-believe claims are of the generic form, *"Every element x in a set S has property P."* - To justify that such a claim is false, we only need to produce a particular x from S that does not have property P. - Such an instance is called a counterexample.

Simple Justification Techniques 2. **Proof by Contradiction** - In applying the justification by con- tradiction technique, we establish that a statement q is true by first supposing that q is false and then showing that this assumption leads to a contradiction

Simple Justification Techniques 3. **Proof by Induction** - Most of these above claims are equivalent to saying some statement $q(n)$ is true "for all $n \geq 1$." - The **Induction** technique amounts to showing that, for any particular $n \geq 1$, there is a finite sequence of implications that starts with something known to be true and ultimately leads to showing that $q(n)$ is true.

[ ]: