Algorithm Analysis

Data Structure & Algorithms with Python
Lecture 3

Overview

- Introduction - Motivation, definitions etc.
- Types of Analyses - Best case, worst case and average case
- Various functions used for analysing rate of growth
- Asymptotic analysis:
  - Big-Oh notation
  - Big-Omega notation
  - Big-Theta notation
- Comparison between various kinds of complexities
- Examples of how efficient algorithms can be modified to reduce computational complexity.
- Summary

Introduction

- *What is an Algorithm?*
  An algorithm is a step-by-step procedure with unambiguous instructions to solve a given problem.

- *What is a data structure?*
  A data structure is a systematic way of organizing and accessing data.

- *Why the analysis of Algorithm required?*
  There are multiple algorithms for doing things. Algorithm analysis helps us in selecting the most efficient algorithm in terms of time and space consumed.

- *What is the Goal of Algorithm Analysis?*
  To compare algorithms (or solutions) mainly in terms of running time, also other factors (memory requirement, developer effort etc.).

Experimental Analysis of Algorithms

Experimentally, algorithm is analysed using the following two quantities:

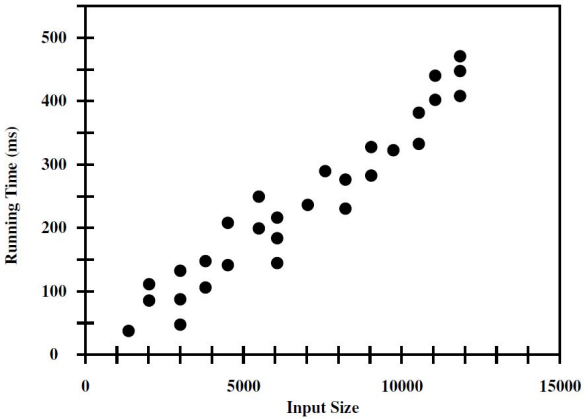
- Running Time t
- Static and Run-time memory requirement.

Run-time is usually found to be proportional to the size n of the input being processed.

$$t \propto n$$

Input size refers to

- Size of an array
- Polynomial degree
- Number of elements in a matrix
- Numbers of bits in a binary representation of input
- Vertices and edges in a graph



Challenges of Experimental Analysis

- Experimental running time depends on the hardware and software platform used for implementation.
- Experiments can be done only on a limited set of test inputs.
- An algorithm needs to be fully implemented for execution and study its running time experimental.

Objectives of Algorithm Analysis

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

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

Approach involves two steps:

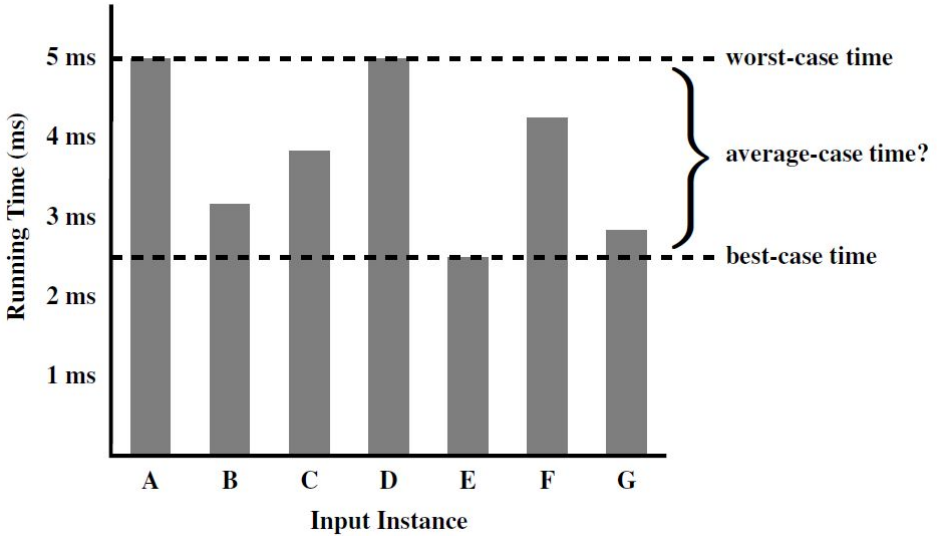
- Counting Primitive Operations (low-level instruction with fixed execution time). E.g:
  - Assigning an identifier to an object
  - Determining the object associated with an identifier
  - Arithmetic operations
  - Comparing two numbers
  - Accessing an element an array or a list
  - Calling a function
  - Returning from a function.
- Measuring operations as a function of Input size

$$t = f(n)$$

Rate of growth: The rate at which the running time increases as a function of input size.

Types of Analysis

- Worst Case
  - Defines the input for which the algorithm takes maximum time complete
- Best Case
  - Defines the input for which the algorithm takes minimum time to complete
- Average Case
  - Provides an average prediction about the running-time of the algorithm.
  - Needs to have understanding of probability distribution of input data.



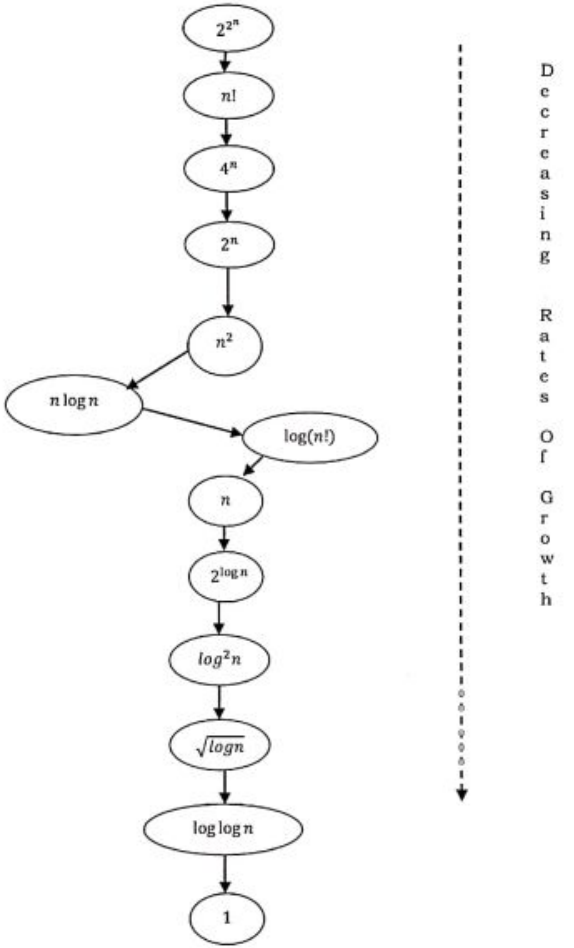
Seven Functions for analysing rate of growth

Approximation:

Total Cost = cost_of_car + cost_of_bicycle
Total Cost ≈ cost_of_car (approx)

$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer' - Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem



The Constant Function:
Computation time does not change with input size.

$$f(n) = c$$

Examples:

- Adding a number to the front of an array
- Adding two numbers
- Assigning a value to a variable
- Comparing two numbers

The Logarithmic function:
Computation time increases logarithmically with input size.

$$f(n) = \log_b n,\ b > 1$$

$$x = \log_b n \iff b^x = n$$

$$\log_b 1 = 0$$

In computer science, we consider base to be 2. By default, we will assume

$$\log n = \log_2 n$$

Example:

- Finding an element in a sorted array (Why?)

Logarithmic Rules

1. $\log_b(ac) = \log_b a + \log_b c$
2. $\log_b(a/c) = \log_b a - \log_b c$
3. $\log_b(a^c) = c \log_b a$
4. $\log_b a = \log_d a / \log_d b$
5. $b^{\log_d a} = a^{\log_d b}$

Examples

- $\log(2n) = \log 2 + \log n = 1 + \log n$, *by rule 1*
- $\log(n/2) = \log n - \log 2 = \log n - 1$, *by rule 2*
- $\log n^3 = 3 \log n$, *by rule 3*
- $\log 2^n = n \log 2 = n \cdot 1 = n$, *by rule 3*
- $\log_4 n = (\log n)/\log 4 = (\log n)/2$, *by rule 4*
- $2^{\log n} = n^{\log 2} = n^1 = n$, *by rule 5.*

The Linear Function:
The computation time is increases linearly with input size.

$$f(n) = n$$

Example:

- Finding an element in an unsorted array (Why?)

The N-Log-N function:

$$f(n) = n \log n$$

The function grows a little more rapidly than the linear function and a lot less rapidly than the quadratic function.

Example:

- Sorting n items by ‘divide-and-conquer’ (mergesort) algorithm - The fastest sorting algorithm possible.

The quadratic function:

$$f(n) = n^2$$

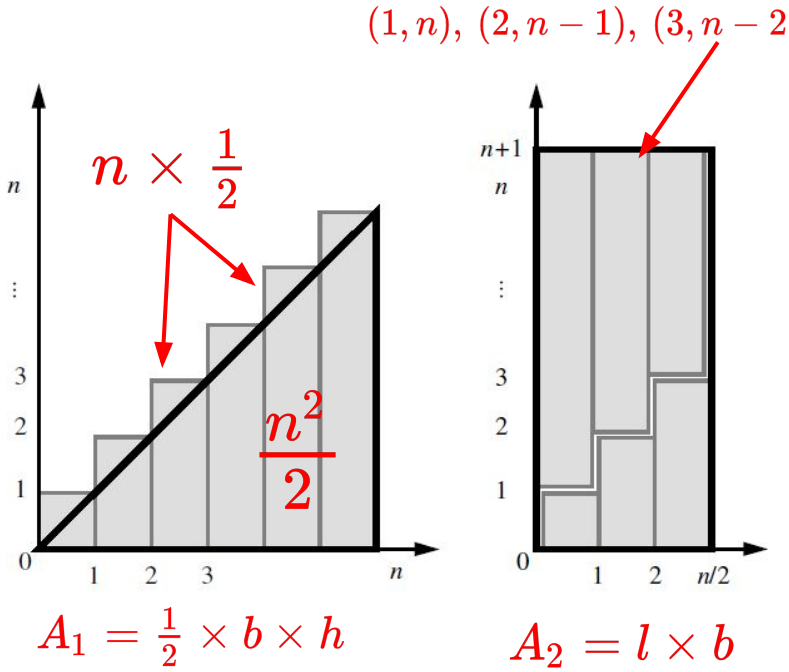
$$1 + 2 + 3 + \cdots + (n-1) + n = \frac{n(n+1)}{2};\ n \geq 1$$

$$[\{1+n\} + \{2+(n-1)\} + \{3+(n-2)\} + \cdots] = \frac{n(n+1)}{2}$$

Examples:

- Shortest path between two nodes in a graph.
- Multiplying two matrices by conventional means.
- Worst possible sorting algorithm.

- Algorithms using two nested loops will usually lead to quadratic rate of growth



Cubic Function & Other Polynomials

Cubic Function

$$f(n) = n^3$$

Polynomials:

$$f(n) = a_0 + a_1 n + a_1 n^2 + \cdots + a_d n^d$$

Where d is the degree of polynomial and $\{a_0, a_1, \cdots, a_d\}$ are the coefficients of the polynomial.

Summations:

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b);\ a \leq b$$

Examples:

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

Polynomial functions using this notation can be written as

$$f(n) = \sum_{i=0}^{d} a_i n^i$$

The Exponential function:

$$f(n) = b^n$$

Where b is the base and n is the exponent.

Exponent Rules:

1. $(b^a)^c = b^{ac}$
2. $b^a b^c = b^{a+c}$
3. $b^a / b^c = b^{a-c}$

Examples:

- $256 = 16^2 = (2^4)^2 = 2^{4 \cdot 2} = 2^8 = 256$ (Exponent Rule 1)
- $243 = 3^5 = 3^{2+3} = 3^2 3^3 = 9 \cdot 27 = 243$ (Exponent Rule 2)
- $16 = 1024/64 = 2^{10}/2^6 = 2^{10-6} = 2^4 = 16$ (Exponent Rule 3)

Geometric Sums:

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n$$
$$= \frac{a^{n+1} - 1}{a - 1};\ a \neq 1,\ n \geq 0$$

Examples:

Q. What is the largest number that can represented in binary notation using n bits?

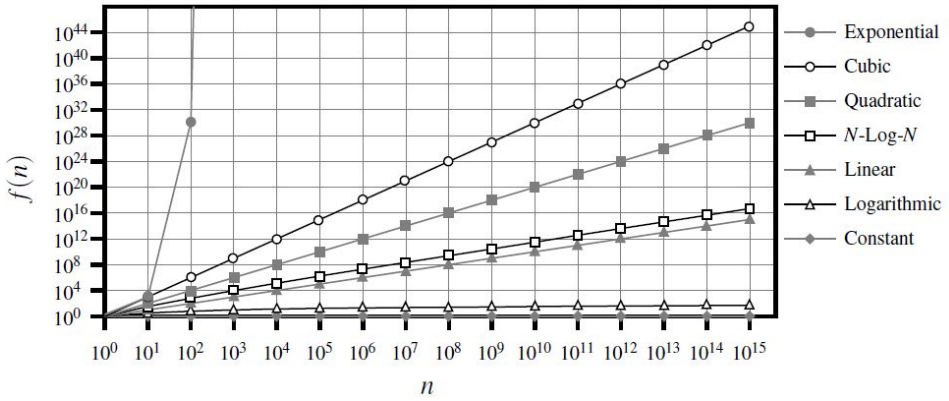
A. $$1 + 2 + 4 + 8 + \ldots + 2^{n-1} = 2^n - 1$$

$$11_b \rightarrow 2^0 + 2^1 = 1 + 2 = 3$$
$$111_b \rightarrow 2^0 + 2^1 + 2^2 = 7$$
$$1111_b \rightarrow 2^0 + 2^1 + 2^2 + 2^3 = 15$$

Comparing Growth Rates

Ceiling and Floor Function:

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



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

Increasing order of complexity

Increasing rate of growth

Comparative Analysis of various 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}

- It shows the importance of good algorithm design.
- The handicap of an asymptotically slower algorithm can not be overcome by using a dramatic speedup in hardware.
- An asymptotically slow algorithm is beaten in the long run by an asymptotically faster algorithm, even if the constant factor for the faster algorithm is worse.

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

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

m

Running the above algorithm on 256 times faster machine

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

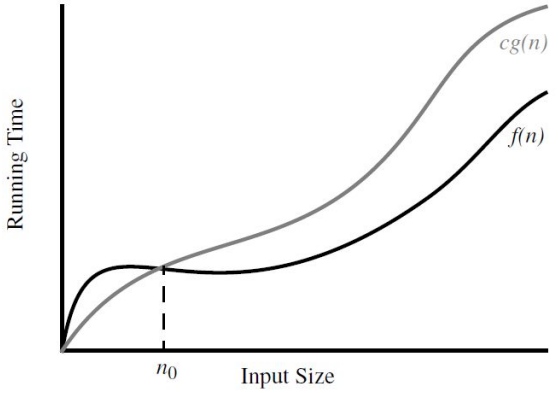
Note that the linearly growing runtime has a worse constant factor compared to quadratic and exponential runtime algorithms

Asymptotic Analysis

- It is an approximate analysis an algorithm complexity as n tends to infinity.
- We focus on the growth rate of the running time as a function of the input size n.
- The “Big-Oh” Notation:

$$f(n) \text{ is } O(g(n))$$
$$\text{if } f(n) \leq cg(n) \text{ for } n \geq n_0$$

- The big-Oh notation allows us to say that a function *f(n)* is “less than or equal to” another function *g(n)* up to a constant factor and in the ***asymptotic*** sense as *n* grows toward infinity.
- “Big-Oh” notations provides the ***tighter upper bound*** of a given function.




```
def find_max(data):
    '''
    return the max value element
     from a nonempty array
    '''
    biggest = data[0]                    C1
    for val in data:
        If val > biggest                 C2
            biggest = val                C1
    Return biggest
```

$$f(n) = c_1 + n \times (c_1 + c_2)$$
$$\text{find\_max is } O(n)$$

Some properties of Big-Oh notation

- If f(n), n>=1 is a polynomial of degree d,

  $f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d,\ n \geq 1$

  Then $f(n)$ is $O(n^d)$

  Justification: for n >= 1,

$$
\begin{aligned}
& f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d \\
\Rightarrow\ & f(n) \leq (|a_0| + |a_1| + \cdots + |a_d|) n^d \\
\Rightarrow\ & f(n) \leq c n^d \\
\Rightarrow\ & f(n) \sim O(n^d)
\end{aligned}
$$

Examples:

$$5n^2 + 3n \log n + 2n + 5 \text{ is } O(n^2)$$

$$20n^3 + 10n \log n + 5 \text{ is } O(n^3)$$

$$3 \log n + 2 \text{ is } O(\log n)$$

$$2^{n+2} + 2 \text{ is } O(2^n)$$

$$2n + 100 \log n \text{ is } O(n)$$

Asymptotic Analysis with Big-Omega Notation

If

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

Then

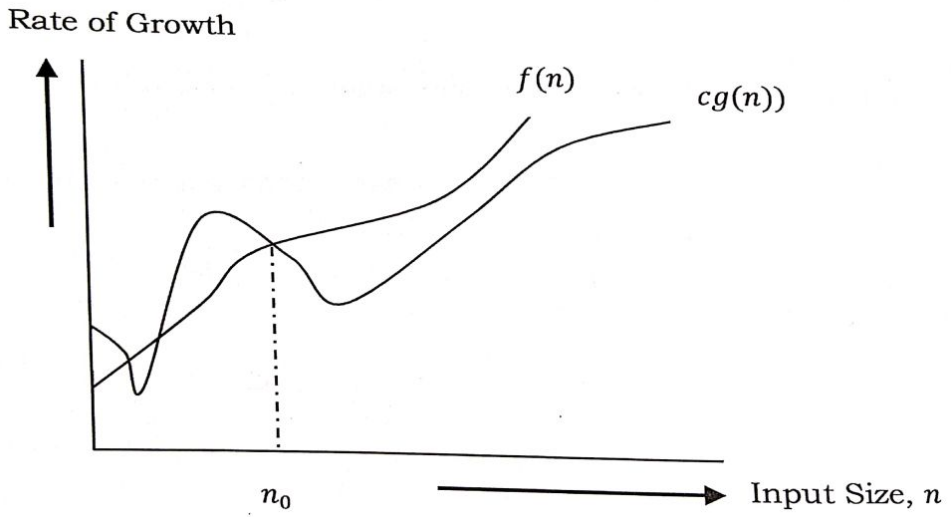
$$f(n) \text{ is } \Omega(g(n))$$

Big-omega notation provides ***tighter lower bound*** of the function.

Example:

$$\begin{aligned} f(n) &= 3n \log n - 2n \\ &= n \log n + 2n(\log n - 1) \\ &\geq n \log n, \forall n \geq 2 \end{aligned}$$

$$f(n) \sim \Omega(n \log n)$$



Asymptotic Analysis with Big-Theta Notation

We say $f(n)$ is $\Theta(g(n))$

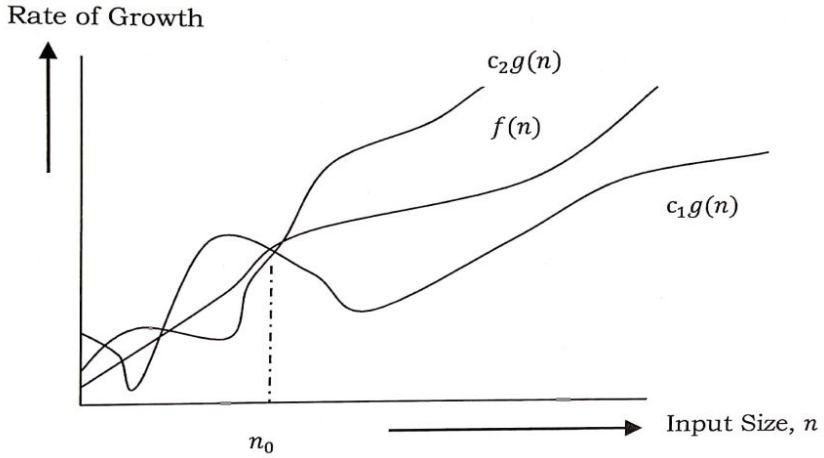
if $f(n) \sim O(g(n))$ & $f(n) \sim \Omega(g(n))$

Or

$$c_1 g(n) \leq f(n) \leq c_2 g(n),\ \forall n \geq n_0$$

Example:

$$f(n) = 3n \log n + 4n + 5 \log n$$
$$\Rightarrow 3n \log n \leq (3 + 4 + 5) n \log n,\ \forall n \geq 2$$
$$\Rightarrow f(n) \sim \Theta(n \log n)$$



- In this case, the upper and lower bound of a given function is same.
- The rate of growth in the best case and the worst case will be same and so will be the average case growth rate.

Some words of Caution

- Constant factors in Big-Oh notation should not be too large

$$O(n \log n) > O(n)$$

$$10n \log n \sim O(n \log n) < 10^{100} n \sim O(n)$$

- When using the big-Oh notation, we should at least be somewhat mindful of the constant factors and lower-order terms we are “hiding.”

- Similarly be careful of constants in exponentials:

$$O(n^c) < O(b^n),\ c > 1,\ n > 1$$

$$O(n^{100}) > O(2^n)$$

Commonly used summations

Arithmetic Series:

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

Geometric Series

$$\sum_{k=1}^{n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1};\ x \neq 1$$

Harmonic Series:

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \log n$$

Others:

$$\sum_{k=1}^{n} \log k \approx n \log n$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \cdots + n^p \approx \frac{1}{p+1} n^{p+1}$$

Few Examples of Asymptotic Analysis of Algorithms

Constant time operations:

Finding the length of a list

`len(data) ~ O(1)`

accessing an element in a list

`Data[j] ~ O(1);`

```
def find_max(data):
    biggest = 0
    for i in range(len(data)):
        If data[i] > biggest:
            biggest = data[i]
```

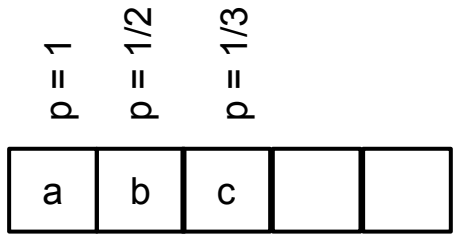
Finding the maximum of a sequence

`find_max(data) ~ O(n)`

How many times, we might update the “biggest” value?

Expected number of times we update the biggest value is a Harmonic number $H_n = \sum_{k=1}^{n} \frac{1}{k} \sim O(\log n)$

Probability of updating the value of biggest:
k = 1, p = 1, certainty that update will occur the first time
k = 2, p = 1/2 ,
k = 3, p = 1/3



Examples:

Ex 1:

$$\begin{aligned} f(n) &= n^4 + 100n^2 + 50 \\ &\not\leq n^4 \ \forall n \geq 10 \end{aligned}$$

Closest Upper Bound

$$f(n) \leq 2n^4 \ \forall n \geq 11$$

$$f(n = 10) = 10000 + 10000 + 50 = 20,050 \not\leq 10^4$$
$$f(n = 11) = 14641 + 12100 + 50 = 26,791 \leq 2 \times 11^4 = 29282$$

Hence, $f(n) \sim O(n^4)$ with $c = 2$ and $n_0 = 11$

Ex 2:

$$f(n) = 410 \leq 410, \ \forall n \geq 1$$
$$\therefore f(n) \sim O(1), \text{ with } c = 1 \text{ and } n_0 = 1$$

Ex 3:

$$f(n) = 2n^3 - 2n^2 \leq 2n^3 \ \forall n \geq 1$$
$$\Rightarrow f(n) \sim O(n^3) \text{ with } c = 2, \text{ and } n_0 = 1$$

Ex 4:

$$f(n) = n^2 + 1 \leq 2n^2 \ \forall n \geq 1$$
$$\text{therefore, } f(n) \sim O(n^2)$$
$$\text{with } c = 2 \text{ and } n_0 = 1$$

Running-time analysis for standard components

Type	Code	Time
Loops	`for i in range(0,n):` `  print(i)`	$T = c \times n = cn \sim O(n)$

Nested Loops	`for i in range(0,n):`
	`  for j in range(0, n):`
	`  print(i,j)`
		$T = c \times n \times n = cn^2 \sim O(n^2)$

Consecutive statements	`n = 100`
	`for i in range(0,n):`
	`  print(i)`
	`for i in range(0,n):`
	`  For j in range(0,n):`
	`  print(i,j)`
		$T = c_0 + c_1 n + c_2 n^2 \sim O(n^2)$

If-then-else	`if n == 1:`
	`  Print n`
	`else:`
	`  For i in range(0,n)`
	`  Print i`
		$T = c_0 + c_1 n \sim O(n)$

Algorithms with Logarithmic growth rate

- It takes constant time to cut the problem size by a fraction (usually ½).
- Assume that at step k,

$$
\begin{aligned}
2^k &= n \\
\Rightarrow \log(2^k) &= \log n \\
\Rightarrow k \log 2 &= \log n \\
\Rightarrow k &= \log n, \ (\text{base} = 2)
\end{aligned}
$$

Total time is O(log n).

```
1 # Functions with logarithmic growth rate
2
3 def logarithm1(n):
4   i = 1
5   while i <= n:
6     i = i * 2
7     print(i, end = ' ')
8
9 def logarithm2(n):
10  i = n
11  while i >= 1:
12    i = i// 2  # floor division
13    print(i, end =' ')
14
15 logarithm1(100)
16 print('\n')
17 logarithm2 (100)
```

```
2 4 8 16 32 64 128

50 25 12 6 3 1 0
```

Average Prefixes:

For a given sequence S of n elements, find another sequence A where A[j] is the average of elements from S[0] to S[j]

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

Analyzing computational complexity:

- n = len(s) ~ O(1)
- A = [0] * n ~ O(n)
- Outer loop (counter j) ~ O(n)
- Inner loop (counter i) is executed 1+2+3+ … + n times = n(n+1)/2 ~ O(n^2)
- Total Time = O(1) + O(n) + O(n^2) ~ O(n^2)

```python
# Average Prefixes
# 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 # 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


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

## test

S = [1, 5, 7, 8, 9, 15]
print(prefix_average1(S))
print(prefix_average2(S))
```

```
[1.0, 3.0, 4.333333333333333, 5.25, 6.0, 7.5]
[1.0, 3.0, 4.333333333333333, 5.25, 6.0, 7.5]
```

Two loops ~ $O(n^2)$

- Initializing variables n and total uses O(1) time
- Initializing the list A uses O(n) time
- Single for loop: counter j is updated in O(n) time.
- Body of the loop is executed n times ~ O(n)
- Total running time of prefix_average3() is in O(n) time.

```python
# 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

## test

S = [1, 5, 7, 8, 9, 15]
print(prefix_average1(S))
print(prefix_average2(S))
print(prefix_average3(S))
```

```
[1.0, 3.0, 4.333333333333333, 5.25, 6.0, 7.5]
[1.0, 3.0, 4.333333333333333, 5.25, 6.0, 7.5]
[1.0, 3.0, 4.333333333333333, 5.25, 6.0, 7.5]
```

Three-way Set Disjointness

- There are three sequences of numbers: A, B and C.
- No individual sequence contains duplicate values.
- 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, there is no element

$$x \text{ s. t. } x \in A, x \in B, \text{and } x \in C$$

```python
# Three-way set disjointness

def disjoint1(A, B, C):
  '''
  Return True if there is no element common to all the three lists
  '''
  for a in A:
    for b in B:
      for c in C:
        if a == b == c:
          return False
  return True

# test
A = [1, 3, 5, 7, 9]
B = [2, 4, 5, 7, 9]
C = [10, 9, 8, 6, 11]
print(disjoint1(A, B, C))
```

```
False
```

Worst-case run time ~ $O(n^3)$

- If there are no matching elements in A & B, there is no need to iterate over C.
- Test condition a == b is evaluated O(n^2) times.
- There can be maximum n matching pairs in A & B and hence the loop C will use O(n^2) time.
- Total time ~ O(n^2)

```python
def disjoint2(A, B, C):
  '''
  Return True if there is no element common to all the three lists
  '''
  for a in A:
    for b in B:
      if a == b:
        for c in C:
          if a == c:
            return False
  return True


# test
A = [1, 3, 5, 7, 9]
B = [2, 4, 5, 7, 9]
C = [10, 9, 8, 6, 11]
print(disjoint1(A, B, C))
print(disjoint2(A, B, C))
```

```
False
False
```

Element Uniqueness

Given a sequence of n numbers, return True if all the elements are distinct.

Example 1: Worst-case running time of this function is proportional to

$$(n-1)+(n-2)+\cdots+2+1 \sim O(n^2)$$

Example 2:

- sorted() function runs in O(n log n) time.
- The loop runs is O(n) time.
- Total time ~ O(n log n)

```
# 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

S = [1,2,3,4]
print(unique1(S))
```

```
True
```

```
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

T = [4, 3, 2, 1, 1]
print(unique2(T))


```

```
False
```

Simple Justification Techniques

- **By Example**

  “Every element x in a set S has property P”.

  To disprove such a generic claim, we only need to produce one particular x from S that does not have property P. Such an instance is called a ***counterexample***.

  Example: The statement

  “ $2^i - 1$ is a prime $\forall i > 1$ ”

  Is false because

  for $i = 4,\ 2^4 - 1 = 15$ (not prime)

- **The “Contra” Attack**

  Contrapositive: To justify “if p is true, then q is true”, we establish “if q is not true, then p is not true”.

  Example:
  Hypothesis: Let *a* and *b* be integers. If *ab* is even, then *a* is even **or** *b* is even.

  To justify the claim, consider the contrapositive: “If a is odd **and** b is odd, then ab is odd”. So, a = 2j + 1, b = 2k+1 for some integer j and k. Then ab = 4jk + 2j+2k+1 = 2(2jk+j+k)+1 which is odd. Hence, the statement is true.

  We apply ***De Morgan’s law*** here.

De Morgan's law

$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$
$$\overline{A \cap B} = \overline{A} \cup \overline{B}$$

Contradiction: 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.

By reaching such a contradiction, we show that no consistent situation exists with q being false, so q must be true.

Example:
Hypothesis: Let a and b be integers. If ab is odd, then a is odd **and** b is odd.

Let ab be odd. We wish to show that a is odd and b is odd.

Let us assume the opposite (De Morgan's law): a is even **or** b is even.

If a = 2j then ab = 2(jb), that is ab is even. This is a contradiction as we assumed ab to be odd. Hence, a is odd. Similarly, b is odd and the above statement is true.

- **Induction**
  Consider the statement where a claim is being made about an infinite set of numbers.

  “q(n) is true for all n >= 1”

  First we show
  q(n) is true for n = 1
  q(n) is true for n = 2,3, … k for some constant k
  We justify that the inductive step is true for for some j > k if q(j) is true for all j < n.

  Then, q(n) is true for all n.

Example: Fibonacci Function F(n)

$F(1) = 1, F(2) = 2 \text{ and } F(n) = F(n-2) + F(n-1) \forall n > 2$

We claim that $F(n) < 2^n$

Base cases:

(n <= 2), F(1) < 2, F(2) = 2 < (4=2^2)

Induction step:

Suppose the above step is true for all k < n.
Now Show that the hypothesis is true for some k < j < n.

Proof:

$$n-1 < n \Rightarrow F(n-1) < 2^{n-1}$$
$$n-2 < n \Rightarrow F(n-2) < 2^{n-2}$$
$$\Rightarrow F(n-2) + F(n-1) < 2^{n-2} + 2^{n-1}$$
$$F(n) < 2^{n-2} + 2^{n-1} < 2^{n-1} + 2^{n-1}$$
$$\Rightarrow F(n) < 2.2^{n-1} = 2^n$$
$$\Rightarrow F(n) < 2^n$$

Hence, F(n) is true for all n > 2

Another Example for Induction

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \; \forall n \geq 1$$

Base Case: for n = 1, sum = 1 = 1(1+1)/2

Induction step: n >= 2, Assume that the claim is true for all k < n. Let k = n-1.

$$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i$$

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

$$= \frac{2n+n^2-n}{2} = \frac{n(n+1)}{2}$$

Hence the above statement is true for all n.

Summary

- Absolute Running-time is a good metric for analyzing algorithm performance but is, hardware dependent and requires full implementation.
- Growth rate of running time with input size is used for analyzing algorithms.
- Asymptotic analysis is carried out without implementation by making asymptotic approximations as n → \infty.
- Worst-case, best-case and average case analysis.
- Different types of growth rate: constant, linear, quadratic, polynomial, exponential and logarithmic.
- Various notations for asymptotic analysis: Big-Oh, Omega and Theta
- Various justification techniques for proving or disproving a claim.