Characterizing Running Times and Algorithm Analysis

ITX2010, CSX3003, IT2230

Data Structures and Algorithms, Information Structures

Learning Objectives

Introduction

Students will be able to:

- Describe why algorithm analysis is important
- Explain what time complexity and growth rate is
- Identify “Big-O” to describe execution time
- Classify the pattern of running time
- Analyze the running time of an algorithm

Chapter Outline

Introduction

1. Algorithm Analysis
   1) Algorithm Analysis
   2) Algorithm Complexity
   3) Time Complexity
   4) Growth Rate
   5) Average Case Analysis
   6) Asymptotic Analysis
   7) Big-O Notation
2. Running Time Analysis
   1) Rules
   2) Patterns
   3) Constant Running Time
   4) Linear Running Time
   5) Logarithmic Running Time
   6) Linear Logarithmic Running Time
   7) Quadratic Running Time

3

Algorithm Analysis

Section 1: Algorithm Analysis

1) Algorithm Analysis
2) Algorithm Complexity
3) Time Complexity
4) Growth Rate
5) Average Case Analysis
6) Asymptotic Analysis
7) Big-O Notation

Algorithm Analysis

- A task to evaluate an algorithm efficiency based upon the amount of computing resources - such as memory space or running time that algorithm uses.
- It concentrate to analyze an algorithm in a way that is independent of the hardware and software requirements.
- It can analyze at the high level of abstraction (algorithm) till to the program code.

Algorithm Analysis

- It often used in comparing two pieces of program or algorithm.
- Ex: Analyze an algorithm to
  - Calculate the sum of n integers
  - Search a key in a given list
- It can be classified in two main stages:
  1. Posterior Analysis (Empirical Analysis):

     Actual statistic like running time space required are monitors.
  2. Prior Analysis (Theoretical Analysis):

     The efficiency is evaluated - excluding hardware and software condition, by considering variable, operations, etc.

Algorithm Complexity

The complexity of an algorithm with its n inputs can be evaluated in term of two traditional factors as follows:

1. **Space complexity**: Memory space reserved from:
   1. Variable
   2. Data structures
   3. Line of Code (LOC)
2. **Time complexity**: Evaluating the execution or running time spent for an outcome

Time Complexity

1. The first benchmark technique of Evaluating the execution or running time depends on the actual execution time.
    - **Ex**: Consider two algorithms of computing a summation of n integer
    - It has been found that the algorithm sum_of_n1 use for loop in the computation which not been found in the algorithm sum_of_n2.
    - Both return the correct result but the second algorithm spent less time than the first algorithm.

```python
def sum_of_n1(n):
  dblSum = 0
  for i in range(1, n+1):
    dblSum = dblSum+i
  return dblSum
```

```python
def sum_of_n2(n):
  return (n*(n+1))/2
```

Fig 3-1 Sum of n Algorithms [2]

Tutorail 1: Sum of N Algorithm

1.3

```python
#Algorithm Analysis: Sum of N
import time

def sum_of_n1(n):
  #Start timer
  start = time.time()
  dblSum = 0
  for i in range(1, n+1):
    dblSum = dblSum+i
  #Stop timer
  end = time.time()
  return dblSum, end-start

def sum_of_n2(n):
  #Start timer
  start = time.time()
  #Stop timer
  end = time.time()
  return (n*(n+1))/2, end-start

print("\nsum_of_n1(5):")
print("Sum of n is : %d spent: %10.7f seconds" % sum_of_n1(10))
print("\nsum_of_n2(5):")
print("Sum of n is : %d spent: %10.7f seconds" % sum_of_n2(10))
```

```
sum_of_n1(5):
Sum of n is : 55 spent:   0.0000031 seconds

sum_of_n2(5):
Sum of n is : 55 spent:   0.0000002 seconds
```

Time Complexity

- How do we justify if they will be run on the different language, OS, or computer?
  - The benchmark should be independent from the software and hardware specification in basis!

- How do we clarify the significant different?
  - The significant different time complexity should be clearly proof and easily justified.

Time Complexity

- If every line of code found in algorithm can affect to the running time, it is reasonably be a base line of evaluating time complexity rather than the actual time computation.
  - The line of code should be considered to evaluate the execution time!
- As the input is not only a single value but a data collection.

Growth Rates

- As the algorithm's input is a data list feed to the algorithm, the growth rate can be monitored in term of how sensitivity of the execution time requires when the list size is increased.

n	$2n^2$	$n^2 + n$
10	200	110
100	20,000	10,100
1000	2,000,000	1,001,000
10000	200,000,000	100,010,000
100000	20,000,000,000	10,000,100,000

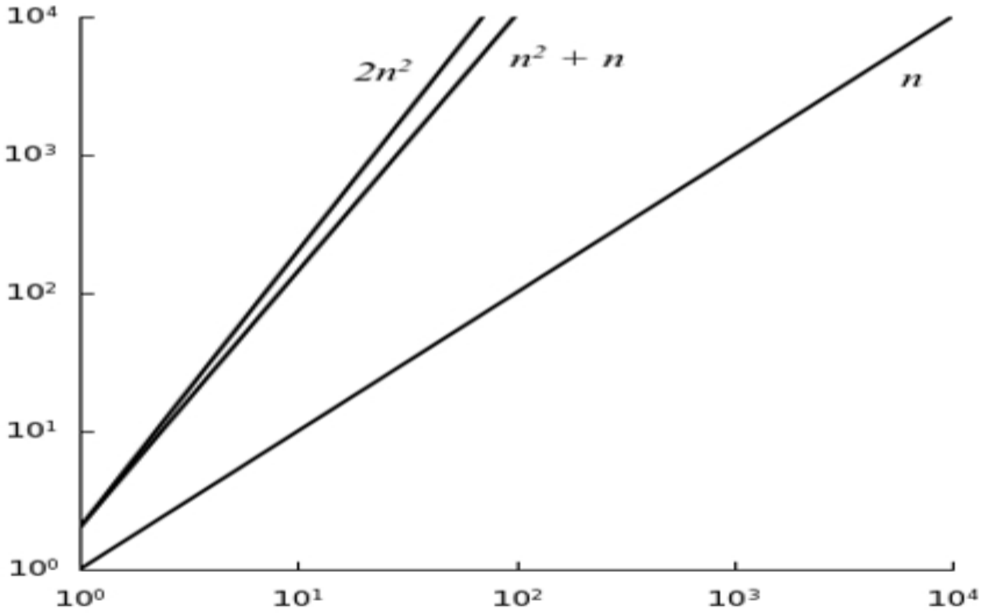


Fig 3-2 comparison of the growth rates for different list sizes [1]

Average Case Analysis

1. **Best case**
   - The best situation that returns the fastest execution time.
   - It seems unrealistic.
2. **Worst case**
   - A situation that consumes the execution time.
   - It gives the absolute guarantee or extreme case.
3. **Average case**
   - It is computed from an average between the Best case and the Worst case.
   - It is estimated to be a distribution.

Asymptotic Notations

3.1 O-notation, Ω-notation and Θ-notation

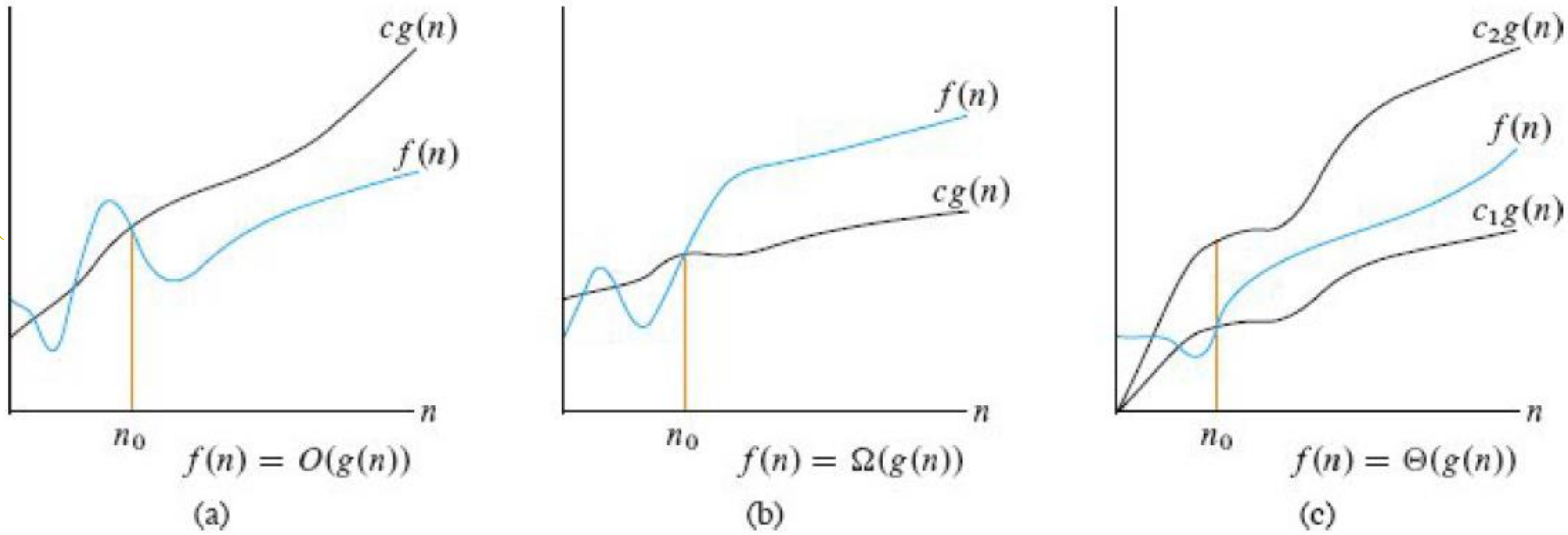
1. **Big-O (O-notation), “bounded above by → asymptotic upper bound”: O(f(n))**
   - For some c and N, T(n) ≤ c·f(n) whenever n > N. **[2] [7]**
   - It describes the upper bound of an algorithm’s growth rate.

2. **Big-Omega (Ω-notation), “bounded below by → asymptotic lower bound”: Ω**(T(n))
   - For some c>0 and N, T(n) ≥ c·f(n) whenever n > N. **[2] [7]**
   - It describes the lower bound of an algorithm’s growth rate.

3. **Big-Theta (Θ-notation), “bounded above and below → asymptotic tight bound”: Θ(f(n))**
   - T(n) = O(f(n)) and also T(n) = W(f(n)) **[2] [7]**
   - It describes both upper and lower bound of an algorithm’s growth rate.

Asymptotic Notations [7]

3.1 O-notation, Ω-notation and Θ-notation



Asymptotic Notations

1.6

[7]

3.1 O-notation, Ω-notation and Θ-notation

Figure 3.2 Graphic examples of the O, Ω, and Θ notations. In each part, the value of n_0 shown is the minimum possible value, but any greater value also works. **(a)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(b)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$. **(c)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that at and to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ inclusive.

Asymptotic Notations

[7]

3.1 O-notation, Ω-notation and Θ-notation

1. **Big-O (O-notation), "bounded above by → asymptotic upper bound": O(f(n))**
   - A function grows no faster than (or <u>at most as slow as</u>) a certain rate, based on the highest-order term.

<u>Ex</u>: Given $f(n) = 7n^3 + 100n^2 - 20n + 6$

- Its highest-order term is $7n^3$
- This function's rate of growth is n^3 → it is $O(n^3)$ #
- Generally, $O(n^c)$ for any **$c \geq 3$** #

Asymptotic Notations

[7]

3.1 O-notation, Ω-notation and Θ-notation

2. **Big-Omega (Ω-notation), "bounded below by → asymptotic lower bound": Ω(f(n))**
   - A function grows at least as fast as a certain rate.

   Ex: Given $f(n) = 7n^3 + 100n^2 - 20n + 6$
   - Its highest-order term is $7n^3$
   - It grows at least as fast as n^3 → it is $\Omega(n^3)$ #
   - Generally, $\Omega(n^c)$ for any $\mathbf{c \leq 3}$ #

Asymptotic Notations

[7]

3.1 O-notation, Ω-notation and Θ-notation

3. Big-Theta (Θ-notation), "bounded above and below → asymptotic tight bound": Θ(f(n))

- A function grows precisely at a certain rate.
- If you can show that a function is both O(f(n)) and Ω(f(n) for the function f(n), → then you have shown that the function is Θ(f(n)).

Ex: Given $f(n) = 7n^3 + 100n^2 - 20n + 6$

- Its highest-order term is $7n^3$
- Since this function's rate of growth is n^3 → it is $O(n^3)$ and grows at least as fast as n^3 → it is $\Omega(n^3)$,
- → it is also Θ(f(n)) #

Asymptotic Notations

1.6

[7]

3.1 O-notation, Ω-notation and Θ-notation

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i – 1].
    j = i – 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j – 1
    A[j + 1] = key
```

Asymptotic Notations

[7]

3.1 O-notation, Ω-notation and Θ-notation

```
INSERTION-SORT(A, n)
1 for i = 2 to n
2     key = A[i]
3     // Insert A[i] into the sorted subarray A[1 : i – 1].
4     j = i – 1
5     while j > 0 and A[j] > key
6         A[j + 1] = A[j]
7         j = j – 1
8     A[j + 1] = key
```

1. **Big-O (O-notation), "bounded above by → asymptotic upper bound": O(f(n))**
   - The running time is dominated by the inner loop.
   - Each iteration of the inner loop takes a constant time → 1 (line 6-7).
   - The total time spent in the inner loop is at most a constant times n^2, or $O(n^2)$

[7]

Asymptotic Notations

3.1 O-notation, Ω-notation and Θ-notation

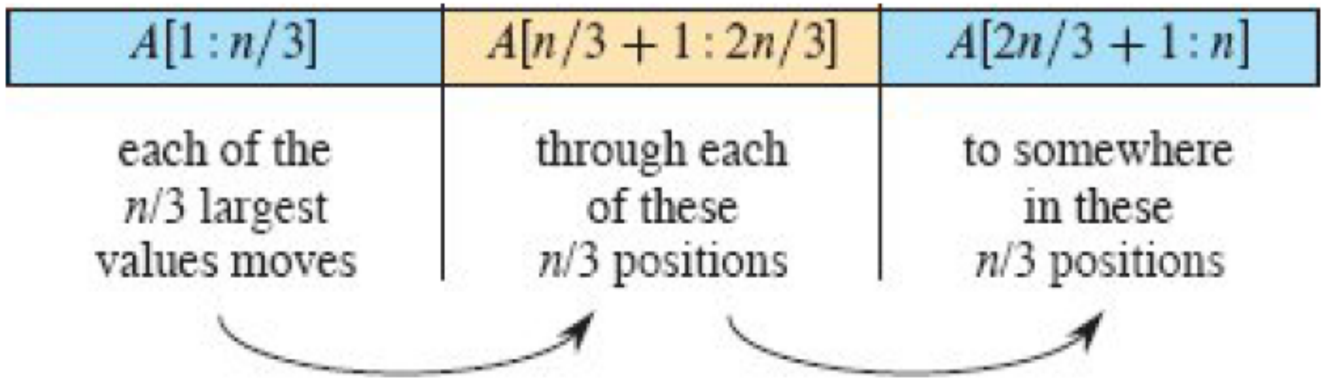


Figure 3.1 The $\Omega(n^2)$ lower bound for insertion sort. If the first $n/3$ positions contain the $n/3$ largest values, each of these values must move through each of the middle $n/3$ positions, one position at a time, to end up somewhere in the last $n/3$ positions. Since each of $n/3$ values moves through at least each of $n/3$ positions, the time taken in this case is at least proportional to $(n/3)(n/3) = n^2/9$, or $\Omega(n^2)$.

2. **Big-Omega (Ω-notation), "bounded below by → asymptotic lower bound": Ω(f(n))**
   - The total time spent if the first n/3 position contain the n/3 largest values →$\Omega(n^2)$

Asymptotic Notations

[7]

3.1 O-notation, Ω-notation and Θ-notation

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i – 1].
    j = i – 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j – 1
    A[j + 1] = key
```

3. **Big-Theta (Θ-notation), “bounded above and below → asymptotic tight bound”: Θ(f(n))**
   - Since the insertion sort runs in O(n2)time in all cases
   - And there is an input that makes it take $\Omega(n^2)$,
   - So, we can conclude that it is also $\Theta(n^2)$.

Asymptotic Notations

[7]

3.1 O-notation, Ω-notation and Θ-notation

```
INSERTION-SORT(A, n)
for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1 : i – 1].
    j = i – 1
    while j > 0 and A[j] > key
        A[j + 1] = A[j]
        j = j – 1
    A[j + 1] = key
```

3. **Big-Theta (Θ-notation), "bounded above and below → asymptotic tight bound": Θ(f(n))**
   - Actually, we cannot say that it is $\Theta(n^2)$ because it does not run for all case.
   - But we should say its running time to be $O(n^2)$ or $\Omega(n^2)$ #

Asymptotic Notations

3.1 O-notation, Ω-notation and Θ-notation

1.6

[7]

FAQ 1: Don’t conflate O-notation with Θ-notation !

- “An O(n lg n)-time algorithm runs faster than an O(n^2)-time algorithm”
  - May be yes or no:
  - Since O(n^2) algorithm might actually run in Θ(n) !
- You should be careful to choose the appropriate asymptotic notation.
- If you want to indicate an asymptotically tight bound, use Θ-notation.

Asymptotic Notations

[7]

3.1 O-notation, Ω-notation and Θ-notation

FAQ 2: Use asymptotic notation to provide the simplest and most precise bounds possible!

- Ex: $f(n) = 3n^2 + 20n$
- Its running time is $\Theta(n^2)$ or $\Theta(3n^2+20n)$, $\Theta(n^2)$ or $O(n^3)$ #
- However, $O(n^3)$ is less precise than $\Theta(n^2)$

Asymptotic Notations

1.6

[7]

3.1 O-notation, Ω-notation and Θ-notation

FAQ 3: Asymptotic notation in equations and inequalities

- Ex1: $4n^2 + 100n + 500 = O(n^2)$ → How do I interpret this formula?
- → $4n^2 + 100n + 500 \in O(n^2)$ #

- Ex2: $T(n) = 2\,T(n/2) + \Theta(n)$ → What is its running time?
- → $\Theta(n)$ #

Asymptotic Notations

[7]

3.1 O-notation, Ω-notation and Θ-notation

Ex3: $\sum_{i=1}^{n} O(i)$,

there is only a single anonymous function (a function of i). This expression is thus *not* the same as $O(1) + O(2) + \cdots + O(n)$, which doesn't really have a clean interpretation.

$$\sum_{k=1}^{n} \Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right) .$$

Asymptotic Notations

[7]

3.1 O-notation, Ω-notation and Θ-notation

Ex3.1: $\sum_{k=1}^{n} \Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right)$.

Series Type	f(k) =	Θ(f(n)) → Θ (n)=
Arithmetic series	$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n$,	$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} = \Theta(n^2)$.
	$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1)$	$= \frac{n(n+1)}{2} + (n+1) = \frac{n^2+n+2n+2}{2} = \frac{(n+1)(n+2)}{2}$. $= \Theta(n^2)$.
General - Arithmetic series	$\sum_{k=1}^{n}(a+bk)$	$\sum_{k=1}^{n}(a+bk) = \Theta(n^2)$

Asymptotic Notations

[7]

3.1 O-notation, Ω-notation and Θ-notation

Ex3.1: $$\sum_{k=1}^{n} \Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right).$$

Series Type	f(k) =	Θ(f(n)) → Θ (n)=
Sum of squares	$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$	$\Theta(n^2)$
Sum of cube	$\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4}$.	$\Theta(n^3)$

[7]

Asymptotic Notations

3.1 O-notation, Ω-notation and Θ-notation

Ex3.3: **Products**

The finite product $a_1 a_2 \ldots a_n$ can be expressed as

$$\prod_{k=1}^{n} a_k .$$

If $n = 0$, the value of the product is defined to be 1. You can convert a formula with a product to a formula with a summation by using the identity

$$\lg\left(\prod_{k=1}^{n} a_k\right) = \sum_{k=1}^{n} \lg a_k$$

Asymptotic Notations

[7]

3.1 O-notation, Ω-notation and Θ-notation

In some cases, asymptotic notation appears on the left-hand side of

Ex4: an equation, as in

$$2n^2 + \Theta(n) = \Theta(n^2).$$

Interpret such equations using the following rule: *No matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid*. Thus, our example means that for *any* function $f(n) \in \Theta(n)$, there is *some* function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$ for all n. In other words, the right-hand side of an equation provides a coarser level of detail than the left-hand side.

We can chain together a number of such relationships, as in

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2). \end{aligned}$$

Asymptotic Notations

[7]

3.1 O-notation, Ω-notation and Θ-notation

FAQ 4: Proper abuses of asymptotic notation

For example, when we say $O(g(n))$, we can assume that we're interested in the growth of $g(n)$ as n grows, and if we say $O(g(m))$ we're talking about the growth of $g(m)$ as m grows. The free variable in the expression indicates what variable is going to ∞.

The most common situation requiring contextual knowledge of which variable tends to ∞ occurs when the function inside the asymptotic notation is a constant, as in the expression $O(1)$.

Θ-Notation

[7]

We use Θ-notation for ***asymptotically tight bounds***. For a given function $g(n)$, we denote by $\Theta(g(n))$ ("theta of g of n") the set of functions

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.

Ω-Notation

[7]

Just as O-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$

Big-O Notation

[7]

Here is the formal definition of O-notation. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the *set of functions*

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$ [1]

Big-O Notation

1.7

[7]

1. Big-O is a factor that determines the magnitude => Don’t need to determine the computer measure of efficiency.

2. O(n) is called as “on-the-order-of n”

Ex: $O[n^2]$ = > Its efficiency is on-the-order of n-squared.

Big-O Notation derived from f(n)

Step of works:

1. Keep the largest term in the function and discard the others.
2. In each term, set or drop the coefficient of the term to one.

Ex1: $f(n) = 5n^4 + 7n^3 + 15n^2 + n$

step 1 => $n^4 + n^3 + n^2 + n$,

step 2 => n^4 =>

Big-O notation = $O[f(n)] = O(n^4)$

Big-O Notation derived from f(n)

Ex2: $f(n) = a_j n^k + a_{j-1} n^{k-1} + \dots + a_2 n^2 + a_1 n + a_0$

step 1 => $n^k + n^{k-1} + \dots + n^2 + n + 1$

step 2 => n^k

Big-O notation = $O[f(n)] = O(n^k)$

Big-O Notation derived from T(n)

T(n)	keep one	drop coef
$3n^2+4n+1$	$3\ n^2$	n^2
$101\ n^2+102$	$101\ n^2$	n^2
$15\ n^2+6n$	$15\ n^2$	n^2
$a\ n^2+bn+c$	$a\ n^2$	n^2

Fig 3-3 comparison of the growth rates for different list sizes [2]

Efficiency	Big-O	Iterations	Est. Time *
Logarithmic	O(log n)	14	microseconds
Linear	O(n)	10,000	0.1 seconds
Linear Logarithm	O(n log n)	140,000	2 seconds
Quadratic	$O(n^2)$	$10{,}000^2$	15-20 min.
Polynomial	$O(n^c)$	$10{,}000^k$	Hours
Exponential	$O(c^n)$	$2^{10{,}000}$	intractable
Factorial	O(n!)	10,000!	intractable
*Assumes instruction speed of one microsecond and 10 instructions in loop.			

Table 3-1 Measure Efficiency [2]

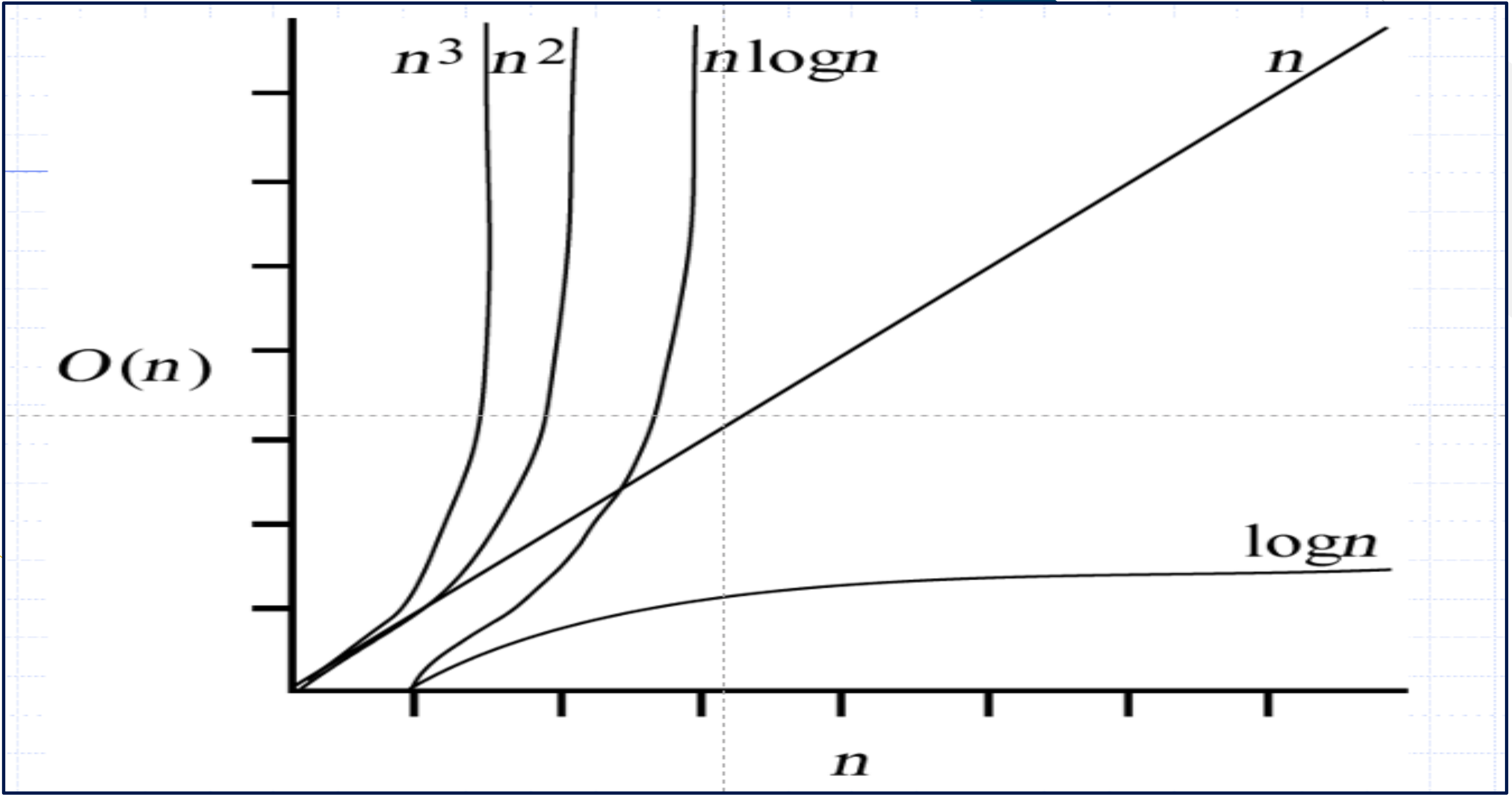


Fig 3-5 comparison of the growth rates for different list sizes [2]

3

Algorithm Analysis

Section 2: Running Time Analysis

1) Rules
2) Patterns
3) Constant Running Time
4) Linear Running Time
5) Logarithmic Running Time
6) Linear Logarithmic Running Time
7) Quadratic Running Time

Running Time Analysis

Rules

- Two important rules [2]

1. **Rule of sums**

   If you do a number of operations in sequence, the runtime is dominated by the most expensive operation.

2. **Rule of products**

   If you repeat an operation a number of times, the total runtime is the runtime of the operation multiplied by the iteration count.

- A sequence of operations when call sequences are flattened

$$T(n) = \max(T_A(n), T_B(n), T_C(n))$$

Running Time Analysis

[2]

Traditional Patterns

- Constants: O(1)
  - Each statements takes an constant execution.
- IF-Then-Else Statement:
  - Considerer the maximum order between True condition (Then case) and False condition (Else case).
- Logarithmic Statement O(log n):
  - If each iteration the input size of decreases by a constant multiple factors.

Running Time Analysis

Traditional Patterns

- Loop: O(n)
  - The running time of a loop is a product of running time of the statement inside a loop and number of iterations in the loop. [3]
- Nested Loop: $O(n^c)$
  - The running time The running time of a loop is a product of running time of the statement inside loop multiplied by a product of the size of all the loops. [3]

Constant Running Time O(1), Θ(1)

- An algorithm spent a constant running times as independent from the input size. [4]

- **Ex:**
  - Directly access the n^{th} element of an array
  - Push and Pop the top element of stack
  - Hash search of a home key

Constant Running Time O(1), Θ(1)

```
#Algorithm Analysis: Constant running time
def sum_of_n2(n):
  return (n*(n+1))/2
```

Linear Running Time O(n), Θ(n)

- An algorithm spent a linear running times that depends on the input size.

- **Ex:**
  - Search a key from an array or a list
  - Find a minimum or maximum key from an array or a list
  - Display all element in an array or a list

2.4

Linear Running Time O(n) , Θ(n)

```
#Algorithm Analysis: Linear running time
def sum_of_n1(n):
  dblSum = 0
  for i in range(1, n+1):
    dblSum = dblSum+i
  return dblSum, end-start
```

Logarithmic Running Time $O(\log_n n)$, $\Theta(\log_n n)$

- An algorithm spent a linear running times that depends on the proportional to the logarithm of the input size.

- **Ex:** Binary search

2.5

Logarithmic Running Time $O(\log_n n)$, $\Theta(\log_n n)$

```
i = 1
sum = 1
while sum < 10:
  print("Round : ", i, ", sum = ", sum)
  sum = sum * 2
  i+=1
```

```
Round : 1 , sum = 1
Round : 2 , sum = 2
Round : 3 , sum = 4
Round : 4 , sum = 8
```

Linear Logarithmic Running Time $O(n \log_n n)$, $\Theta(n\log_n n)$

- An algorithm spent a linear running times that depends on the proportional to n log n production of the input size.

- **<u>Ex</u>:** Heap sort, Quick sort (Average case)

2.6

Linear Logarithmic Running Time $O(n\log_n n)$, $\Theta(n\log_n n)$

```python
i = 1
for i in range(3):
  j=1
  sum = 1
  print("\nRound I : ", i, ", sum = ", sum)
  while sum < 10:
    print("Round J : ", j, ", sum = ", sum)
    sum = sum * 2
    j+=1
```

```
Round I :  0 , sum =  1
Round J :  1 , sum =  1
Round J :  2 , sum =  2
Round J :  3 , sum =  4
Round J :  4 , sum =  8

Round I :  1 , sum =  1
Round J :  1 , sum =  1
Round J :  2 , sum =  2
Round J :  3 , sum =  4
Round J :  4 , sum =  8

Round I :  2 , sum =  1
Round J :  1 , sum =  1
Round J :  2 , sum =  2
Round J :  3 , sum =  4
Round J :  4 , sum =  8
```

Quadratic Running Time $O(n^2)$, $\Theta(n)$

- An algorithm spent a linear running times that depends on the square of the input size.

- **Ex:** Insertion sort, Selection sort Bubble sort

Quadratic Running Time $O(n^2)$, $\Theta(n)$

```python
i = 1
for i in range(3):
  j=1
  sum = 1
  print("\nRound I : ", i, ", sum = ", sum)
  while sum < 10:
    print("Round J : ", j, ", sum = ", sum)
    sum = sum + 1
    j+=1
```

```
Round I :  0 , sum =  1
Round J :  1 , sum =  1
Round J :  2 , sum =  2
Round J :  3 , sum =  3
Round J :  4 , sum =  4
Round J :  5 , sum =  5
Round J :  6 , sum =  6
Round J :  7 , sum =  7
Round J :  8 , sum =  8
Round J :  9 , sum =  9

Round I :  1 , sum =  1
Round J :  1 , sum =  1
Round J :  2 , sum =  2
Round J :  3 , sum =  3
Round J :  4 , sum =  4
Round J :  5 , sum =  5
Round J :  6 , sum =  6
Round J :  7 , sum =  7
Round J :  8 , sum =  8
Round J :  9 , sum =  9

Round I :  2 , sum =  1
Round J :  1 , sum =  1
Round J :  2 , sum =  2
Round J :  3 , sum =  3
Round J :  4 , sum =  4
Round J :  5 , sum =  5
Round J :  6 , sum =  6
Round J :  7 , sum =  7
Round J :  8 , sum =  8
Round J :  9 , sum =  9
```

References

Texts | Integrated Development Environment (IDE)

[1] Data Structures and Algorithms Using Python, Rance D. Necaise, John Wiley & Sons, Inc., 2011

[2] Data Structures: A Pseudocode Approach with C++, Richard F. Gilberg and Behrouz A. Forouzan, Brooks/Cole, 2001

[3] Problem Solving in Data Structures & Algorithms Using Python: Programming Interview Guide, 1st Edition, Hermant Jain, Thiftbooks, March 2017

[4] Problem Solving in Data Structures & Algorithms Using Python: Programming Interview Guide, 1st Edition, Hermant Jain, Thiftbooks, March 2017

[5] https://trinket.io/features/python3

[6] https://colab.research.google.com/

[7] Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein,, Fourth Edition, The MIT Press, 2022.