The National School of Artificial Intelligence
المدرسة الوطنية العليا للذكاء الاصطناعي

Data Structures and Algorithms 2

Chapter 2
Algorithm Analysis

Dr. Fouzia ANGUEL

2nd year / S3

September 2024 – January 2025

Course Outline

- ❖ Algorithms efficiency
  - - Machine-dependent vs Machine-independent
- ❖ Function ordering
  - - Order of growth
  - - Weak Order;
  - - Landau symbols Big-Oh ; Big-omega ; big theta and Little-oh .
- ❖ Algorithm complexity analysis
  - – Rules for complexity analysis
  - – Analysis of various types of algorithms
  - – Master Theorem

Algorithm Efficiency

Example : Shortest path problem

- A city has *n* view points
- Buses move from one view point to another
- A bus driver wishes to follow the shortest path (travel time wise).
- Every view point is connected to another by a road.
- However, some roads are less congested than others.
- Also, roads are one-way, i.e., the road from view point 1 to 2, is different from that from view point 2 to 1.

Algorithm Efficiency

Example : Shortest path problem

How to find the shortest path between any two pairs?

➔ **Naïve approach**

- ◆ List all the paths between a given pair of view points
- ◆ Compute the travel time for each.
- ◆ Choose the shortest one.

How many paths are there between any two view points (without revisits)?

$$\mathbf{n! \cong (n/e)^n}$$

➔ It will be impossible to run the algorithm for n = 30

Algorithm efficiency

- Run time in the computer is Machine dependent

Example : Need to multiply two positive integers a and b

Subroutine 1: Multiply a and b

Subroutine 2:
```
V = a,    W = b
While W > 1
 V →V + a; W →W-1
 Output V
```

Algorithm efficiency

First subroutine has 1 multiplication.

Second has b additions and subtractions.

For some architectures, 1 multiplication is more expensive than b additions and subtractions.

Ideally, we would like to program all choices and run all of them in the machine we are going to use and find which is efficient!

Machine Independent Analysis

We assume that every basic operation takes constant time

Example **Basic** Operations:

Addition, Subtraction, Multiplication, Memory Access

Non-basic Operations:

Sorting, Searching

Efficiency of an algorithm is the number of basic operations it performs

We do not distinguish between the basic operations.

Subroutine 1 uses 1 basic operation (*)

Subroutine 2 uses 2b basic operations (+, -)

Subroutine **1** is more efficient.

This measure is good for all large input sizes

In fact, we will not worry about the exact values, but will look at "broad classes" of values.

Let there be **n inputs**.

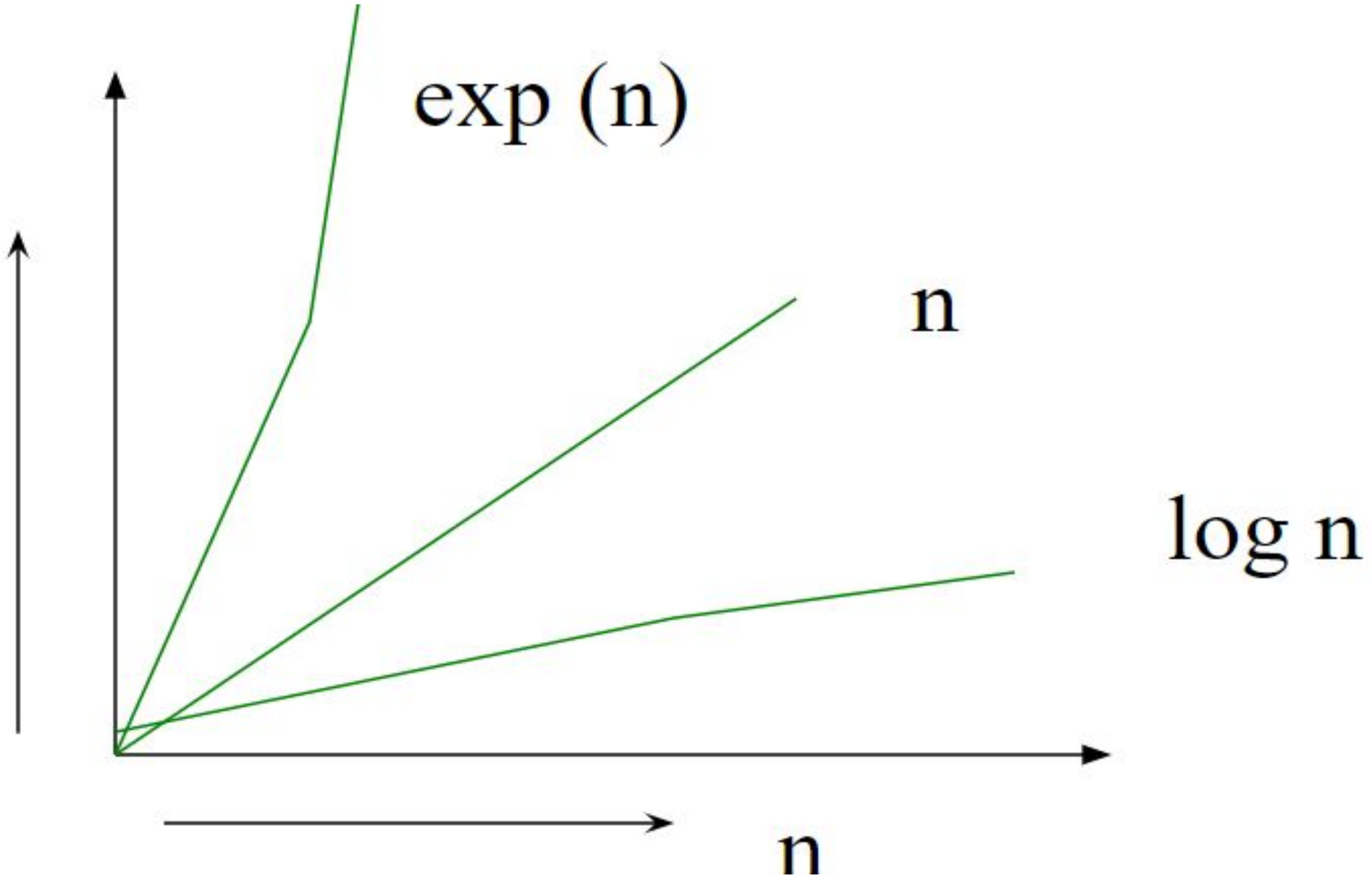
If an algorithm needs **n** basic operations and another needs **2n** basic operations, we will consider them to be in the same efficiency category.

However, we distinguish between exp(n), n, log(n)

Function Ordering

Order of Increase(order of growth)

We worry about the speed of our algorithms for large input sizes.

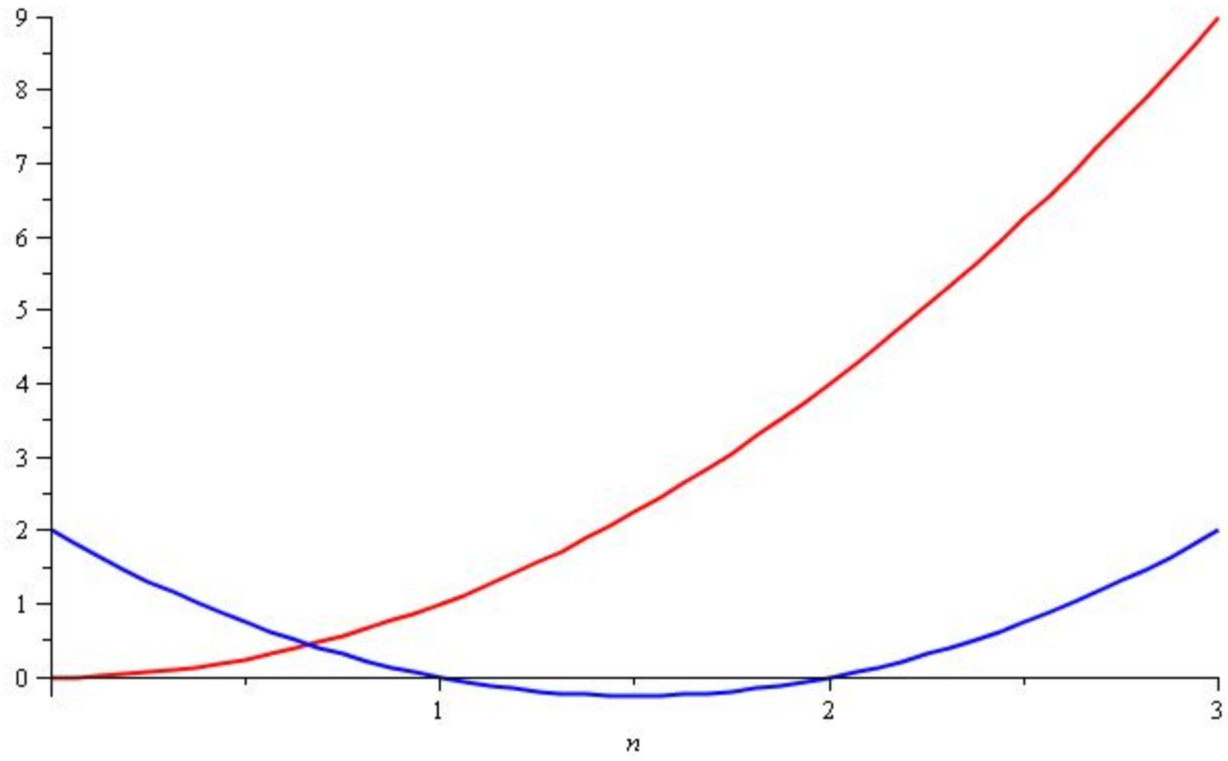


Quadratic Growth

Consider the two functions

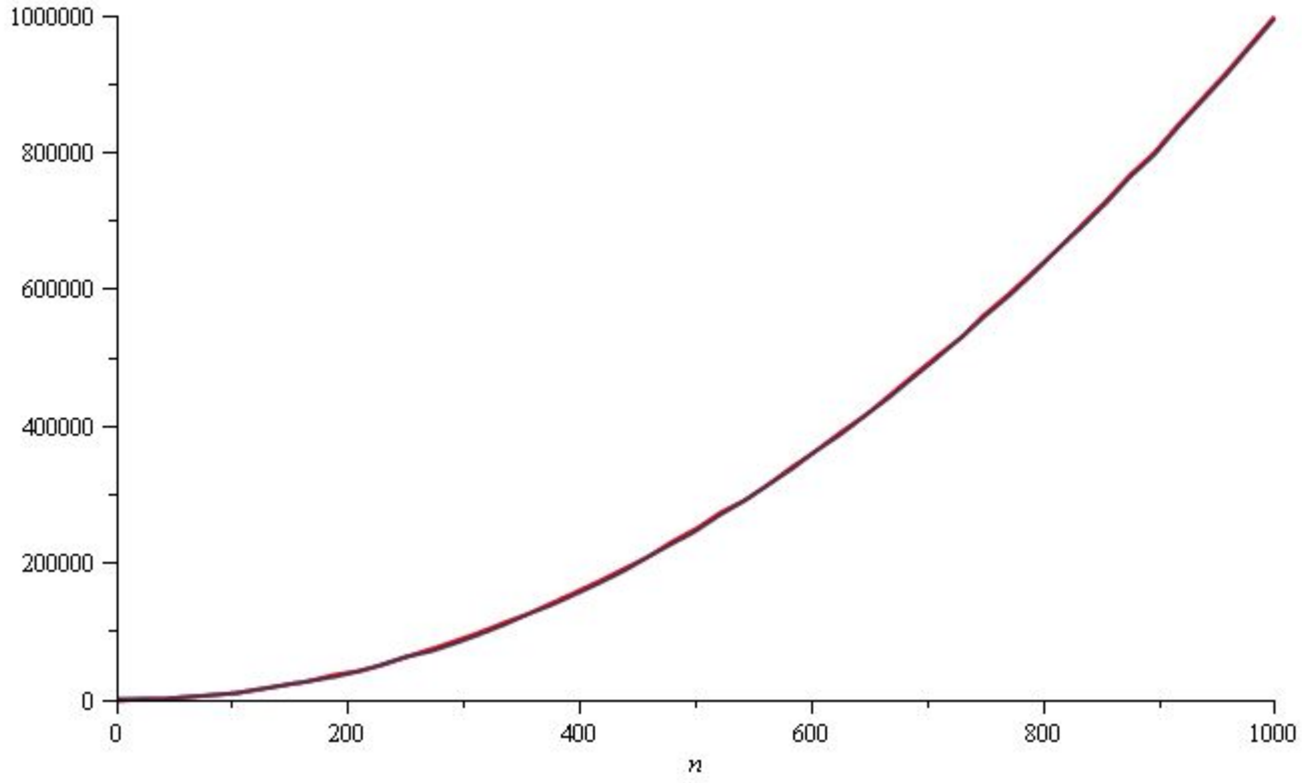
$f(n) = n^2$ and $g(n) = n^2 - 3n + 2$

Around $n = 0$, they look very different



Quadratic Growth

Yet on the range $n = [0, 1000]$, they are (relatively) indistinguishable:



Quadratic Growth

The absolute difference is large, for example,

f(1000) = 1 000 000

g(1000) = 997 002

but the relative difference is very small

$$\left|\frac{f(1000) - g(1000)}{f(1000)}\right| = 0.002998 < 0.3\%$$

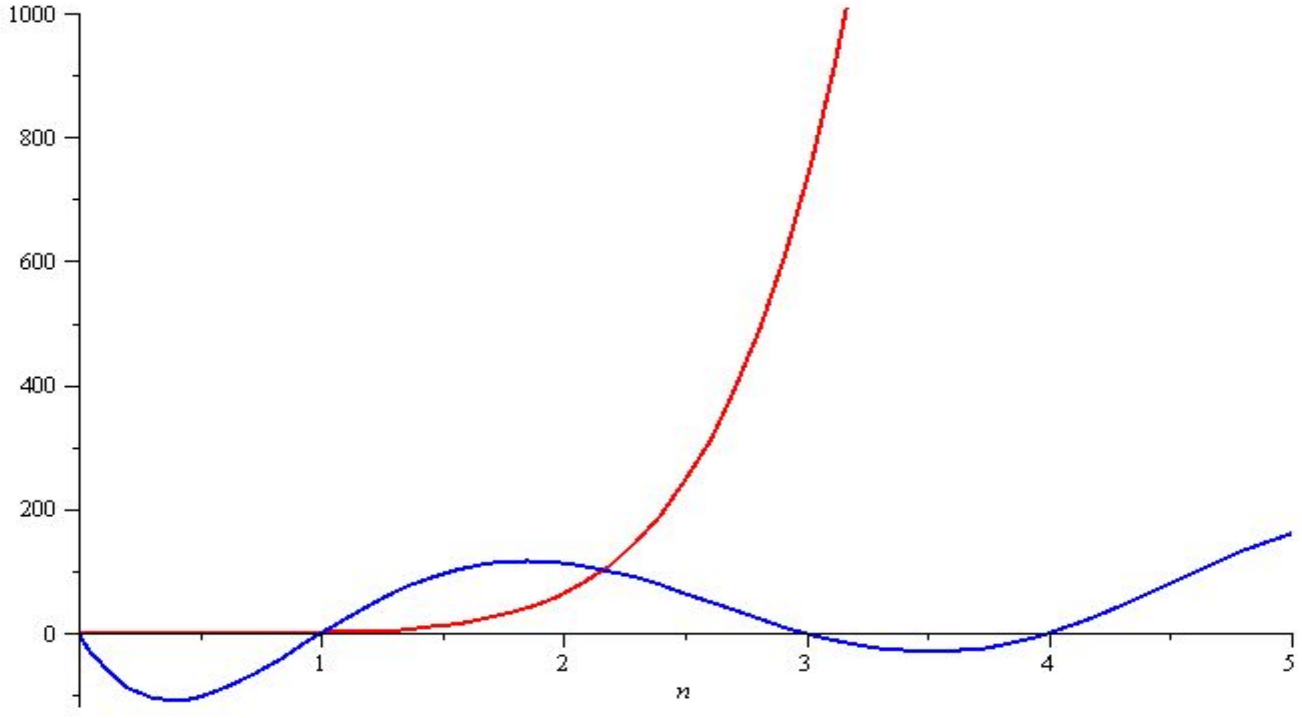
and this difference goes to zero as $n \to \infty$

Polynomial Growth

To demonstrate with another example,

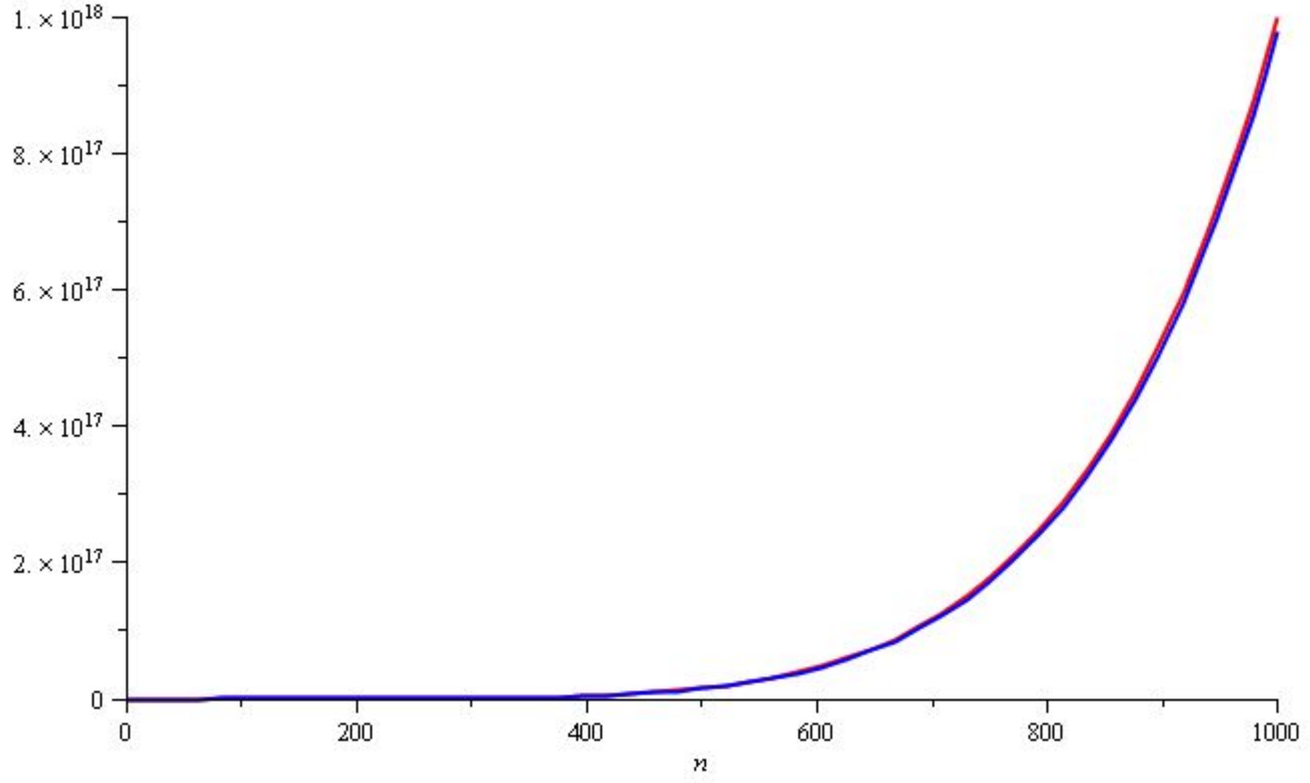
$f(n) = n^6$ and $g(n) = n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$

Around $n = 0$, they are very different



Polynomial Growth

Still, around $n = 1000$, the relative difference is less than 3%



Polynomial Growth

The justification for both pairs of polynomials being similar is that, in both cases, they each had the same leading term:

$\boldsymbol{n^2}$ in the first case, $\boldsymbol{n^6}$ in the second

Suppose however, that the coefficients of the leading terms were different

- In this case, both functions would exhibit the same rate of growth, however, one would always be proportionally larger

Weak ordering

Consider the following definitions:

- We will consider two functions to be equivalent, ***f* ~ *g***, if

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=c \qquad \text{where} \qquad 0<c<\infty$$

- We will state that ***f* < *g*** if $$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$$

For functions we are interested in, these define a *weak ordering*

Weak ordering

f and g are functions from the set of natural numbers to itself.

Let $f(n)$ and $g(n)$ describe the run-time of two algorithms

- If $\boldsymbol{f(n) \sim g(n)}$, then it is always possible to improve the performance of one function over the other by purchasing a faster computer

- If $\boldsymbol{f(n) < g(n)}$, then you can **never** purchase a computer fast enough so that the second function always runs in less time than the first

Note that for small values of n, it may be reasonable to use an algorithm that is asymptotically more expensive, but we will consider these on a one-by-one basis

Function orders "Landau Symbols"

we will make some assumptions:

- Our functions will describe the time or memory required to solve a problem of size n
- We are restricting to certain functions :
  - They are defined for $n \geq 0$
  - They are strictly positive for all n
    - In fact, $f(n) > c$ for some value $c > 0$
    - That is, any problem requires at least one instruction and byte
  - They are increasing (monotonic increasing)

Function orders "Landau Symbols"

Big Oh Notation

A function **f(n)** is **O(g(n))** if the rate of growth of f(n) is not greater (not faster) than that of g(n).

Definition 1

f(n) = O(g(n)) if there are a number $\mathbf{n_o}$ and a nonnegative **c** such that

$$\text{for all } n \geq n_o, \quad 0 \leq f(n) \leq cg(n).$$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists and is finite, then f(n) is O(g(n)) .

Intuitively, (not exactly) f(n) is O(g(n)) means f(n) $\leq$ g(n) for all n beyond some value n_o; i.e. g(n) is an upper bound for f(n).

Function orders “Landau Symbols”

Example Functions

sqrt(n) , n, 2n, ln n, exp(n), n + sqrt(n) , n + n^2

$\lim_{n\to\propto}$ sqrt(n) /n = 0,	sqrt(n) is O(n)
$\lim_{n\to\propto}$ n/sqrt(n) = infinity,	n is not O(sqrt(n))
$\lim_{n\to\propto}$ n /2n = 1/2,	n is O(2n)
$\lim_{n\to\propto}$ 2n /n = 2,	2n is O(n)

$\lim_{n\to\propto} \ln(n)/n = 0,$	$\ln(n)$ is $O(n)$
$\lim_{n\to\propto} n/\ln(n) =$ infinity,	n is not $O(\ln(n))$
$\lim_{n\to\propto} \exp(n)/n =$ infinity,	$\exp(n)$ is not $O(n)$
$\lim_{n\to\propto} n/\exp(n) = 0,$	n is $O(\exp(n))$
$\lim_{n\to\propto} (n+\mathrm{sqrt}(n))/n = 1,$	$n + \mathrm{sqrt}(n)$ is $O(n)$
$\lim_{n\to\propto} n/(\mathrm{sqrt}(n)+n) = 1,$	n is $O(n+\mathrm{sqrt}(n))$
$\lim_{n\to\propto} (n + n^2)/n =$ infinity,	$n + n^2$ is not $O(n)$
$\lim_{n\to\propto} n/(n + n^2) = 0,$	n is $O(n + n^2)$

Implication of big-Oh notation

Suppose we know that our algorithm uses at most O(f(n)) basic steps for any n inputs, and n is sufficiently large,

- then we know that our algorithm will terminate after executing at most f(n) basic steps.

- We know that a basic step takes a constant time in a machine.

Hence, our algorithm will terminate in a constant time f(n) units of time, for all large n.

Function orders "Landau Symbols"

Ω "Omega" Notation

Now a lower bound notation, **Ω**

Definition 2

f(n) = **Ω(g(n))** if there are a number $\mathbf{n_o}$ and a nonnegative **c** such that

for all $n \geq n_o$, $f(n) \geq cg(n)$.

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ exists

We say g(n) is a lower bound on f(n),
i.e. no matter what specific inputs we have, the algorithm will not run faster than this lower bound.

Suppose, an algorithm has complexity Ω(f(n)) . This means that there exists a positive constant c such that for all sufficiently large n, there exists at least one input for which the algorithm consumes at least c*f(n) steps.

Function orders "Landau Symbols"

θ "theta " Notation

Definition 3

f(n) = **θ(g(n))** if and only if f(n) is O(g(n)) and Ω(g(n))

f(n) = **θ(g(n))** if there exist positive n_o, c_1, and c_2 such that

$$c_1\, g(n) \le f(n) \le c_2\, g(n) \qquad \text{whenever } n \ge n_o$$

- θ(g(n)) is "`asymptotic equality"
- $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)}$ is a finite, positive constant, if it exists

A function f(n) is θ(g(n)) if The function f(n) has a rate of growth **equal** to that of g(n) . **Θ** represents a **tight bound** in asymptotic analysis, which means it captures both the **upper** and **lower** bounds of a function's growth.

Function orders "Landau Symbols"

Little-oh Notation

Definition 4
f(n) = o(g(n)) if for all positive constant c, there exists an n_o such that :

$$f(n) < cg(n) \text{ when } n > n_o$$

Less formally , **f(n) = o(g(n))** if f(n) =O(g(n)) and f(n) ≠ θ(g(n)).

"asymptotic strict inequality"

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \quad \text{if} \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} \quad \text{exists}$$

Function orders “Landau Symbols”

Suppose that f(n) and g(n) satisfy $\lim_{n\to\infty}\frac{f(n)}{g(n)} = c$

If $\lim_{n\to\infty}\frac{f(n)}{g(n)} = c$ where $0 < c < \infty$, it follows that f(n) = $\mathbf{\Theta}$(g(n))

If $\lim_{n\to\infty}\frac{f(n)}{g(n)} = c$ where $0 \le c < \infty$, it follows that f(n) = $\mathbf{O}$(g(n))

If $\lim_{n\to\infty}\frac{f(n)}{g(n)} = 0$, it follows that f(n) = o(g(n))

Function orders "Landau Symbols"

$$\mathrm{f}(n) = \mathbf{o}(\mathrm{g}(n)) \qquad \lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} = 0$$

$$\mathrm{f}(n) = \mathbf{O}(\mathrm{g}(n)) \qquad \lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} < \infty$$

$$\mathrm{f}(n) = \mathbf{\Theta}(\mathrm{g}(n)) \qquad 0 < \lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} < \infty$$

$$\mathrm{f}(n) = \mathbf{\Omega}(\mathrm{g}(n)) \qquad \lim_{n\to\infty} \frac{\mathrm{f}(n)}{\mathrm{g}(n)} > 0$$

Function orders “Landau Symbols”

Terminology

Asymptotically less than or equal to	**O**
Asymptotically greater than or equal to	**Ω**
Asymptotically equal to	**θ**
Asymptotically strictly less	**o**

Little-o as a Weak Ordering

We can show that, for example

$\ln(n) = \mathbf{o}(n^p)$ for any $p > 0$

Proof: Using l'Hôpital's rule.

If you are attempting to determine $\lim_{n\to\infty} \frac{f(n)}{g(n)}$

but both $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$, it follows

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f^{(1)}(n)}{g^{(1)}(n)}$$

Repeat as necessary…

Note: the *k*th derivative will always be shown as $f^{(k)}(n)$

$$\lim_{n\to\infty} \frac{\ln(n)}{n^p} = \lim_{n\to\infty} \frac{1/n}{pn^{p-1}} = \lim_{n\to\infty} \frac{1}{pn^p} = \frac{1}{p} \lim_{n\to\infty} n^{-p} = 0$$

Big-Θ as an Equivalence Relation

If we look at the first relationship, we notice that $f(n) = \Theta(g(n))$ seems to describe an equivalence relation:

1. $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
2. $f(n) = \Theta(f(n))$
3. If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, it follows that $f(n) = \Theta(h(n))$

Consequently, we can group all functions into equivalence classes, where all functions within one class are big-theta Θ of each other

Big-Θ as an Equivalence Relation

For example, all of

n^2 $\quad$ $100000\,n^2 - 4\,n + 19$ $\quad$ $n^2 + 1000000$

$323\,n^2 - 4\,n\ln(n) + 43\,n + 10$ $\quad$ $42n^2 + 32$

$n^2 + 61\,n\ln^2(n) + 7n + 14\ln^3(n) + \ln(n)$

are big-**Θ** of each other

E.g., $42n^2 + 32 = \boldsymbol{\Theta}(\,323\,n^2 - 4\,n\ln(n) + 43\,n + 10\,)$

We will select just one element to represent the entire class of these functions: $\boldsymbol{n^2}$

- We could choose any function, but this is the simplest

Function orders "Landau Symbols"

Terminology

The most common classes are given names:

$\Theta(1)$	constant
$\Theta(\ln(n))$ or $\Theta(\log(n))$	logarithmic
$\Theta(n)$	linear
$\Theta(n \ln(n))$	"n log n"
$\Theta(n^2)$	quadratic
$\Theta(n^3)$	cubic
2^n, e^n, 4^n, …	exponential

Recall that all logarithms are scalar multiples of each other

Therefore $\log_b(n) = \Theta(\ln(n))$ for any base b

Function orders “Landau Symbols”

Example Functions

sqrt(n) , n, 2n, ln n, exp(n), n + sqrt(n) , n + n^2

$\lim_{n \rightarrow \propto}$ sqrt(n) /n = 0,	sqrt(n) is o(n) and O(n)
$\lim_{n \rightarrow \propto}$ n/sqrt(n) = infinity,	n is Ω(sqrt(n))
$\lim_{n \rightarrow \propto}$ n /2n = 1/2,	n is θ(2n)
$\lim_{n \rightarrow \propto}$ 2n /n = 2,	2n is θ(n)

$\lim_{n\to\propto} \ln(n)/n = 0,$	$\ln(n)$ is $o(n)$
$\lim_{n\to\propto} n/\ln(n) = \text{infinity},$	n is $\Omega(\ln(n))$
$\lim_{n\to\propto} \exp(n)/n = \text{infinity},$	$\exp(n)$ is $\Omega(n)$
$\lim_{n\to\propto} n/\exp(n) = 0,$	n is $o(\exp(n))$
$\lim_{n\to\propto} (n+\text{sqrt}(n))/n = 1,$	$n + \text{sqrt}(n)$ is $\theta(n)$
$\lim_{n\to\propto} n/(\text{sqrt}(n)+n) = 1,$	n is $\theta(n+\text{sqrt}(n)),$
$\lim_{n\to\propto} (n + n^2)/n = \text{infinity},$	$n + n^2$ is $\Omega(n)$
$\lim_{n\to\propto} n/(n + n^2) = 0,$	n is $o(n + n^2)$

Algorithms Analysis

An algorithm is said to have ***polynomial*** *time complexity* if its run-time may be described by **$O(n^d)$** for some fixed $d \geq 0$

- We will consider such algorithms to be ***efficient***

Problems that have no known polynomial-time algorithms are said to be ***intractable***

- *Traveling salesman problem*: find the shortest path that visits n cities
- Best run time: $\Theta(n^2 2^n)$

Complexity of a Problem Vs Algorithm

A **problem** is $O(f(n))$ means there is some $O(f(n))$ algorithm to solve the problem.

A **problem** is $\Omega(f(n))$ means every algorithm that can solve the problem is $\Omega(f(n))$

Rules for arithmetic with big-O symbols

Rule 1

If T1(n) = O(f (n)) and T2(n) = O(g(n)), then

(a) T1(n) + T2(n) = O(f (n) + g(n)) (intuitively and less formally it is O(max(f (n), g(n)))),

(b) T1(n) $*$ T2(n) = O(f (n) $*$ g(n)).

Rule 2

If T(n) is a polynomial of degree k, then $T(n) = \theta(n^k)$.

Rule 3

- $\log^k n = O(n)$ for any constant k.

This tells us that logarithms grow very slowly.

Rules for arithmetic with big-O symbols

Rule 4

If $f(n) = \mathbf{O}(g(n))$, then

$c * f(n) = \mathbf{O}(g(n))$ for any constant c.

Rule 5

If $f1(n) = \mathbf{O}(g(n))$ but $f2(n) = \mathbf{o}(g(n))$, then

$f1(n) + f2(n) = \mathbf{O}(g(n))$.

Rule 6

If $f(n) = \mathbf{O}(g(n))$, and $g(n) = \mathbf{o}(h(n))$, then

$f(n) = \mathbf{o}(h(n))$. (complexity of $f \circ g$)

These are not all of the rules, but they're enough for most purposes.

Algorithm Complexity Analysis

- Three cases for which the efficiency of algorithms has to be determined :
  - *worst case* : is when an algorithm requires a maximum number of steps,
  - the *best case* : is when the number of steps is the smallest, and
  - the *average case* falls between these extremes.
- We define *T*avg(*N*) and *T*worst(*N*), as the average and worst-case running time, resp., used by an algorithm on input of size *N*. Clearly, *T*avg(*N*) ≤ *T*worst(*N*).
- Average-case performance often reflects *typical behavior*
- Worst-case performance represents a guarantee for performance on any possible input.
- The best-case performance of an algorithm is of little interest: does not represent the typical behavior.It is occasionally analyzed.

Algorithm Complexity Analysis

Example

1. diff = sum = 0;	● Line 1 takes 2 basic steps
2. for (k=0: k < N; k++)	● in every iteration of first loop
3. sum → sum + 1;	Line 3 takes 2 basic steps.
4. diff → diff - 1;	Line 4 takes 2 basic steps
	First loop runs N times
5. for (k=0: k < 3N; k++)	● in every iteration of second
6. sum → sum - 1;	Line 6 loop takes 2 basic step
	● Second loop runs for 3N times

Overall, 2 + 4N + 6N steps (without counting the test and increment operations for each iteration in the two loops)

This is O(N)

Algorithm Complexity Analysis

General Rules

Rule 1- **Consecutive Statements:**
This just add , which means that the maximum is that counts .

Rule 2- Complexity of a loop:
The running time of a loop is at most the running time of the statements inside the loop (including tests) times the number of iterations.

O(Number of iterations in a loop * maximum complexity of each iteration)

Rule 3- Nested Loops:
The running time of a group of nested loops is the running time inside a group of nested loops multiplied by the product of the sizes of all the loops .

Complexity of an outer loop = number of iterations in this loop * complexity of inner loop, etc.

Algorithm Complexity Analysis

Example

`sum = 0;`
`for (i=0; i < N; i++)`
`    for (j=0; j < N; j++)`
`        sum → sum + 1;`
	Outer loop: N iterations
	Inner loop: O(N)
	Overall: O(N²)

`for (i=0; i < N; i++)`
`    a[i] = 0;`
`for (i=0; i < N; i++)`
`    for (j=0; j < N; j++)`
`        a[i] = a[j] + i+j ;`
	First loop O(N)
	Inner loop: O(N)
	Outer loop: N iterations
	Overall: O(N) + O(N²) So
	O(N²)

Algorithm Complexity Analysis
General Rules

Rule 3- If else

For the fragment

```
If (Condition)
    S1
Else
    S2
```

Maximum of the two complexities

The running time of an if/else statement is never more than the running time of the test plus the larger of the running times of S& and S2

```
If (yes)
        print(1,2,….1000N)
else  print(1,2,….N²)
```

overall **O(N^2)**

the basic strategy is analyzing from the inside (or deepest part) out . If there are function calls , these must be analyzed first .

Algorithm Complexity Analysis

Analysis of recursion

- If the recursion is really just a for loop , the analysis is usually trivial .

```
 Long factorial  (int n) {
        if (n <= 1)
               return 1;
        else
               return n*factorial(n – 1);
}
```

O(N)

- However, if the recursion is properly used . The analysis will involve a recurrence relation.

Algorithm Complexity Analysis

Analysis of recursion

- Suppose we have the following code :

```
Long fib (int n) {
1.   if (n <= 1)
2.              return 1;
     else
3.        return fib(n – 1) + fib(n – 2);
}
```

Let T(N) be the running time for the function call fib(n)

if N= 0 or N=1 $T(0) = T(1) = O(1)$

if n>=2

$T(n)$ = cost of constant op at line 1 + cost of line 3 work

$T(n)$ = 1 op + (addition + 2 function calls)

Algorithm Complexity Analysis

Analysis of recursion

T(n) = 1 op + (addition + cost of fib(n-1) + cost fib(n-2))

Thus ,

$$T(n) = T(n-1) + T(n-2) + 2$$

Since fib(n) = fib(n-1) + fib(n-2)

it is easy to show by induction that :

$$T(n) >= fib(n)$$

- we have showed (in chapter 1) that fib(n) $<(5/3)^n$
- a similar proof shows for n>4, fib(n) >= $(3/2)^n$

thus T(n) >= $(3/2)^n$ and so

the running time of the programme grows exponentially.
This program is slow because there is a huge amount of redundant work being performed.

By using an array and a for loop,
the programme running time can be reduced substantially.

Algorithm Complexity Analysis

Maximum Subsequence Problem

Given an array of N elements $A_1, A_2, A_3, \ldots, A_N$, (possibly negative)

find the maximum value of $\sum_{k=i}^{j} A_k$

Need to find i, j such that the sum of all elements between the ith and jth positions is maximum for all such sums

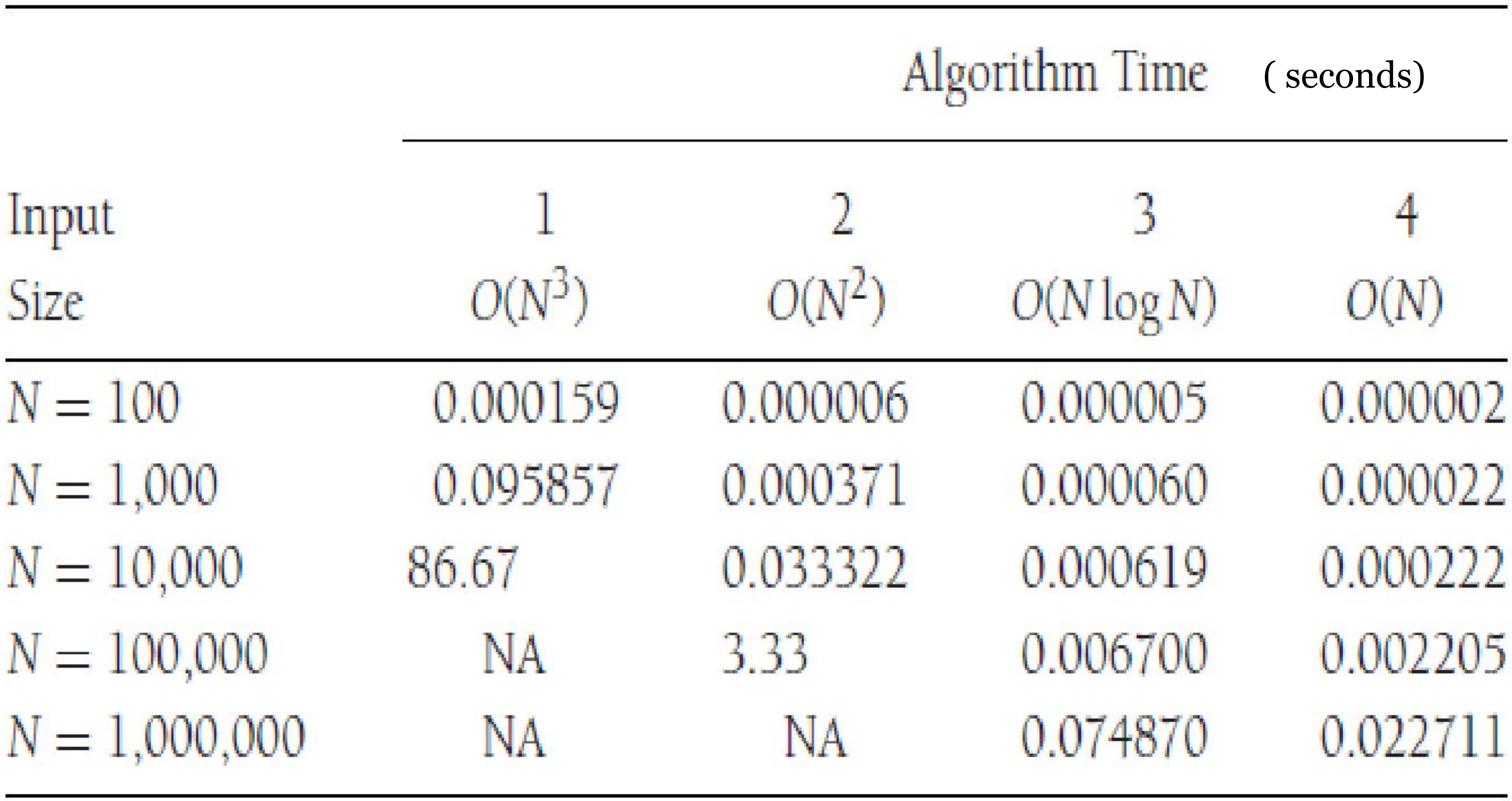
(for convenience, the maximum subsequence sums is 0 if all integers are negative)

Example

for the input -2, 11,-4,13,-5,-2 the answer is 20

We will discuss four algorithms to solve it, their performance varies : $O(N)$, $O(N\log N)$, $O(N^2)$, $O(N^3)$

Running time of 4 algorithms for max subsequence sum

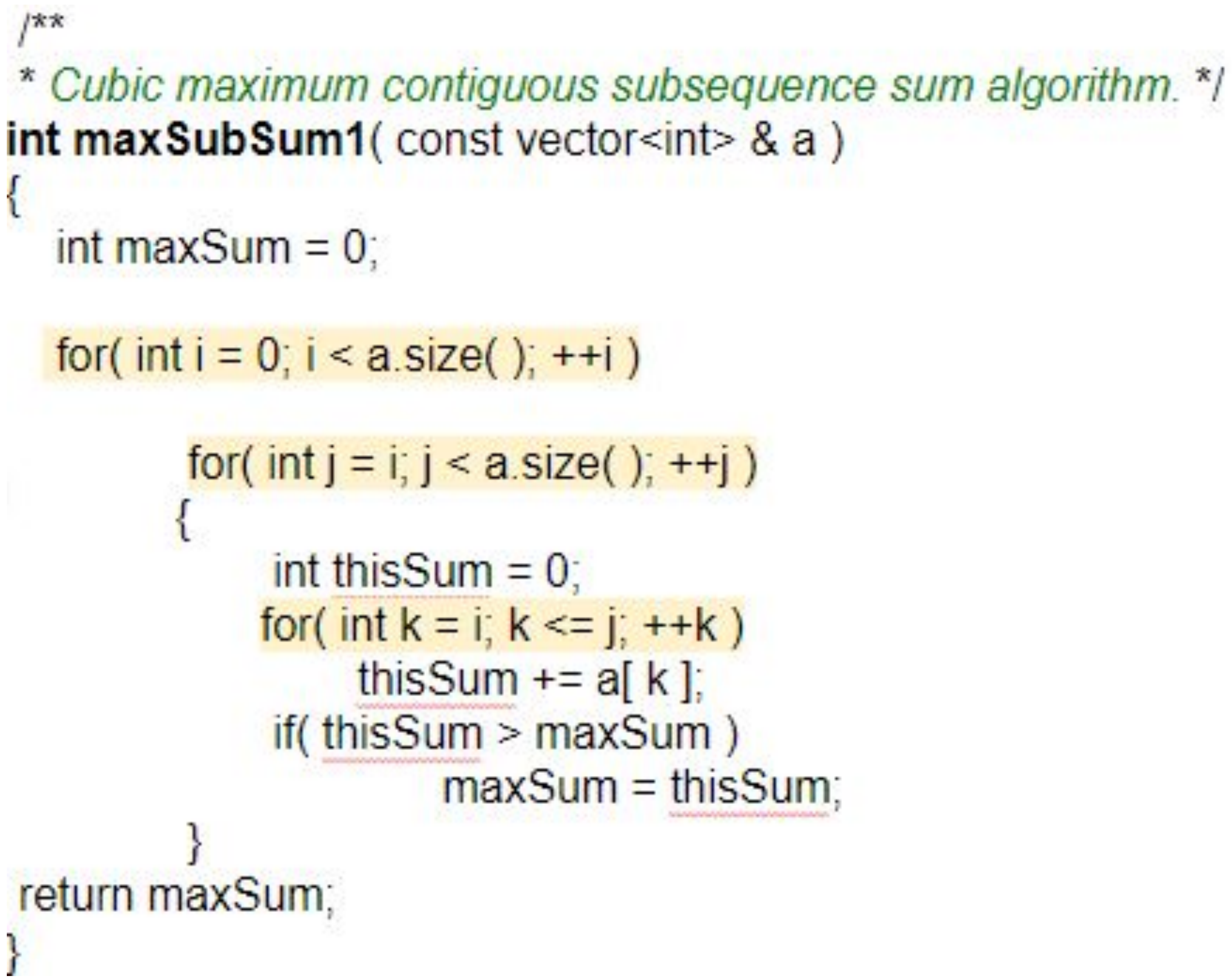


	Algorithm Time (seconds)			
Input	1	2	3	4
Size	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
$N = 100$	0.000159	0.000006	0.000005	0.000002
$N = 1{,}000$	0.095857	0.000371	0.000060	0.000022
$N = 10{,}000$	86.67	0.033322	0.000619	0.000222
$N = 100{,}000$	NA	3.33	0.006700	0.002205
$N = 1{,}000{,}000$	NA	NA	0.074870	0.022711

Figure [textbook Weiss, Figure 2.2]

Maximum Subsequence Problem

Algorithm 1



```
/**
 * Cubic maximum contiguous subsequence sum algorithm. */
int maxSubSum1( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); ++i )

            for( int j = i; j < a.size( ); ++j )
            {
                int thisSum = 0;
                for( int k = i; k <= j; ++k )
                    thisSum += a[ k ];
                if( thisSum > maxSum )
                        maxSum = thisSum;
            }
return maxSum;
}
```

Complexity of Algorithm 1

Because constants do not matter, the runtime is obtained from the sum :

We have $$\sum_{k=0}^{N-1} \sum_{k=j}^{N-1} \sum_{k=i}^{j} 1$$

inner loop $$\sum_{k=i}^{j} 1 = j - i + 1$$

Outer Loop $$\sum_{j=i}^{N-1} (j - i + 1) = \frac{(N-i+1)(N-i)}{2}$$

$$\sum_{i=0}^{N-1} \frac{(N-i+1)(N-i)}{2} = \frac{N^3 + 3N^2 + 2N}{6}$$

Analysis of Algorithm 1

in Algorithm 1 can be made more efficient leading to O(N^2).
Thus , the cubic running time can be avoid by removing the innermost for loop, because :

$$\sum_{K=i}^{j} A_k = A_j + \sum_{K=i}^{j-1} A_k$$

Maximum Subsequence Problem

Algorithm 2

```
/**
* Quadratic maximum contiguous subsequence sum algorithm.
 */
int maxSubSum2( const vector<int> & a )
  {
  int maxSum = 0;
  for( int i = 0; i < a.size( ); ++i )
    {
          int thisSum = 0;
           for( int j = i; j < a.size( ); ++j )
            {
                  thisSum += a[ j ];

                  if( thisSum > maxSum )
                          maxSum = thisSum;
          }
     }
return maxSum;}
```

Complexity of Algorithm 2

the runtime of algorithm2 is obtained from the two for loops :

$$\sum_{i=0}^{N-1} \sum_{j=i}^{N-1} 1 = \sum_{i=0}^{N-1} (N - i)$$

$$\sum_{i=0}^{N-1}(N - i) = N^2 - \frac{(N-1)N}{2} = \frac{N(N+1)}{2} = \frac{N^2 + N}{2}$$

$O(N^2)$

Maximum Subsequence Problem

Algorithm 3

Divide and Conquer

Divide-and- conquer strategy :

- ❖ Split the big problem into "two" small sub-problems,
- ❖ Solve each of them efficiently,
- ❖ Combine the "two" solutions.

Maximum Subsequence Problem

Algorithm 3

Divide and Conquer

❖ Divide the array into two parts: left part, right part each to be solved recursively

❖ The maximum subsequence can be in one of three places :

- completely in the left half ,
- or completely in right half
- or it crosses the middle and is both halves.

➔ The first two cases can be solved recursively

➔ The last case, can be obtained by finding the max subsequence in the left ending at the last element and the max subsequence in the right starting from the center (i.e. the first element in the second half)

Maximum Subsequence Problem

Algorithm 3

Example

First half				Second half			
4	−3	5	−2	-1	2	6	-2

Max subsequence sum for first half = 6 (elements A_1–A_3)

second half = 8 (elements A_5–A_7)

Max subsequence sum for first half ending at the last element (4th elements included) is 4 (elements A_1–A_4)

Max subsequence sum for second half starting at the first element (5th element included) is 7 (elements A_5–A_7)

Max subsequence sum spanning the middle is 4 + 7 = 11

Max subsequence spans the middle

Maximum Subsequence Problem

Algorithm 3 : divide and conquer

```
int maxSumRec( const vector<int> & a, int left, int right )
{
    if( left == right )  // Base case
        if( a[ left ] > 0 )
            return a[ left ];
        else
            return 0;

    int center = ( left + right ) / 2;
    int maxLeftSum  = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    int maxLeftBorderSum = 0, leftBorderSum = 0;
    for( int i = center; i >= left; --i )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }
```

```
        int maxRightBorderSum = 0, rightBorderSum = 0;
        for( int j = center + 1; j <= right; ++j )
        {
            rightBorderSum += a[ j ];
            if( rightBorderSum > maxRightBorderSum )
                maxRightBorderSum = rightBorderSum;
        }

        return max3( maxLeftSum, maxRightSum,
                         maxLeftBorderSum + maxRightBorderSum );
    }

    /**
     * Driver for divide-and-conquer maximum contiguous
     * subsequence sum algorithm.
     */
    int maxSubSum3( const vector<int> & a )
    {
        return maxSumRec( a, 0, a.size( ) - 1 );
    }
```

Complexity analysis
Algorithm 3

Let T(N) be the time it takes to solve a maximum subsequence sum problem of size N.

- If N=1; lines 8 to 12 executed; taken to be one unit : T(1) = 1
- N>1: 2 recursive calls, 2 for loops, some bookkeeping ops (e.g. lines 14, 34)
  - The 2 for loops (lines 19 to 32): clearly O(N)
  - Lines 8, 14, 18, 26, 34: constant time; ignored compared to O(N)
  - Recursive calls made on half the array size each :

$$2 * T(N/2)$$

SO: programme time is :

$$2 * T(N/2) + O(N) \text{ with } T(1) = 1$$

Complexity analysis for Algorithm 3

$T(1)=1$

$T(n) = 2T(n/2) + cn$

$= 2.(cn/2 + 2T(n/4))+ cn$

$= 4T(n/4) + 2cn$

$= 8T(n/8) + 3cn$

$=$…………..

$= 2^iT(n/2^i) + icn$

$=$………………… (reach a point when $n = 2^i$ $i=\log n$

$= n.T(1) + c\, n \log n$

$= n + c\, n \log n = $ **$O(n \log n)$**

Complexity analysis

Algorithm 4

```
/**
 * Linear-time maximum contiguous subsequence sum algorithm.
 */
int maxSubSum4( const vector<int> & a )
{
    int maxSum = 0, thisSum = 0;

    for( int j = 0; j < a.size( ); ++j )
    {
        thisSum += a[ j ];

        if( thisSum > maxSum )
            maxSum = thisSum;
        else if( thisSum < 0 )
            thisSum = 0;
    }

    return maxSum;
}
```

$T(N) = O(N)$ Obvious!

but the logic of the algorithm.
is not obvious ???

Complexity analysis

Binary Search

- Given an integer X and integers $A_0, A_1, A_2, \ldots\ldots, A_{n-1}$ which are presorted.
- find i such that $A_i = X$, or
- return i= -1 if X is not in the input.

Solution 1

➔ Scanning through the list from left to right. Runs in linear time .

➔ this algorithm does not take advantage of the fact that the list is sorted .

Solution 2 (better)

➔ Check if X is the middle. If so, the answer is found .

➔ If X < the middle , we can apply the same strategy to the sorted subarray to the left;

➔ likewise, if X > middle, we look to the right half.

Complexity analysis
Binary Search

Algorithm 1

```
/**
 * Performs the standard binary search using two comparisons per level.
 * Returns index where item is found or -1 if not found.
 */
template <typename Comparable>
int binarySearch( const vector<Comparable> & a, const Comparable & x )
{
    int low = 0, high = a.size( ) - 1;

    while( low <= high )
    {
        int mid = ( low + high ) / 2;

        if( a[ mid ] < x )
            low = mid + 1;
        else if( a[ mid ] > x )
            high = mid - 1;
        else
            return mid;   // Found
    }
    return NOT_FOUND;     // NOT_FOUND is defined as -1
}
```

Algorithm 2

```
Search(num, A[],left, right)
  {
        if (left = right)
         {
              if (A[left ]=num)    return(left) and exit;
              else conclude NOT PRESENT and exit;
          }
      center =⌊ (left + right)/2⌋;
       If (A[center] < num)
          Search(num, A[], center + 1, right);
      If (A[center]>num)
          Search(num, A[], left, center );
      If (A[center]=num) return(center) and exit;
}
```

Complexity analysis
Binary Search

Algorithm 1

work done inside the loop takes O(1) per iteration

number of iterations ?

The number of iterations continues until the search space is reduced to 1 (or the target is found). The relationship can be described by:

$$n, n/2, n/4, \ldots, 1$$

The number of iterations needed to reduce n to 1 is $\log_2 n$.

thus, the running time of Algo 1 is O(log n)

Algorithm 2

$T(n) = T(n/2) + C$

the running time of Algorithm 2 is O(log n)

Complexity analysis
divide and conquer
Master Theorem

Used to calculate time complexity of divide-and-conquer algorithms.

It applies to recurrence relations of the form:

$$T(n) = aT(n/b) + f(n)$$

where

– n is the size of the input;

– a is the number of subproblems in the recursion;

– n/b is the size of each subproblem (all assumed to have the same size);

– f(n): cost of work done outside recursive calls.

n/b might not be an integer, but replacing T(n/b) with ⌈ T(n/b) or ⌊T(n/b)⌋ does not affect the asymptotic behavior of the recurrence.

Complexity analysis

divide and conquer

Master Theorem "Basic Form"

The master theorem compares the function $n^{\log_b a}$ to the function **f(n)**.

➔ Intuitively, if $n^{\log_b a}$ is larger (by a polynomial factor), then the solution is $T(n) = \Theta(n^{\log_b a})$

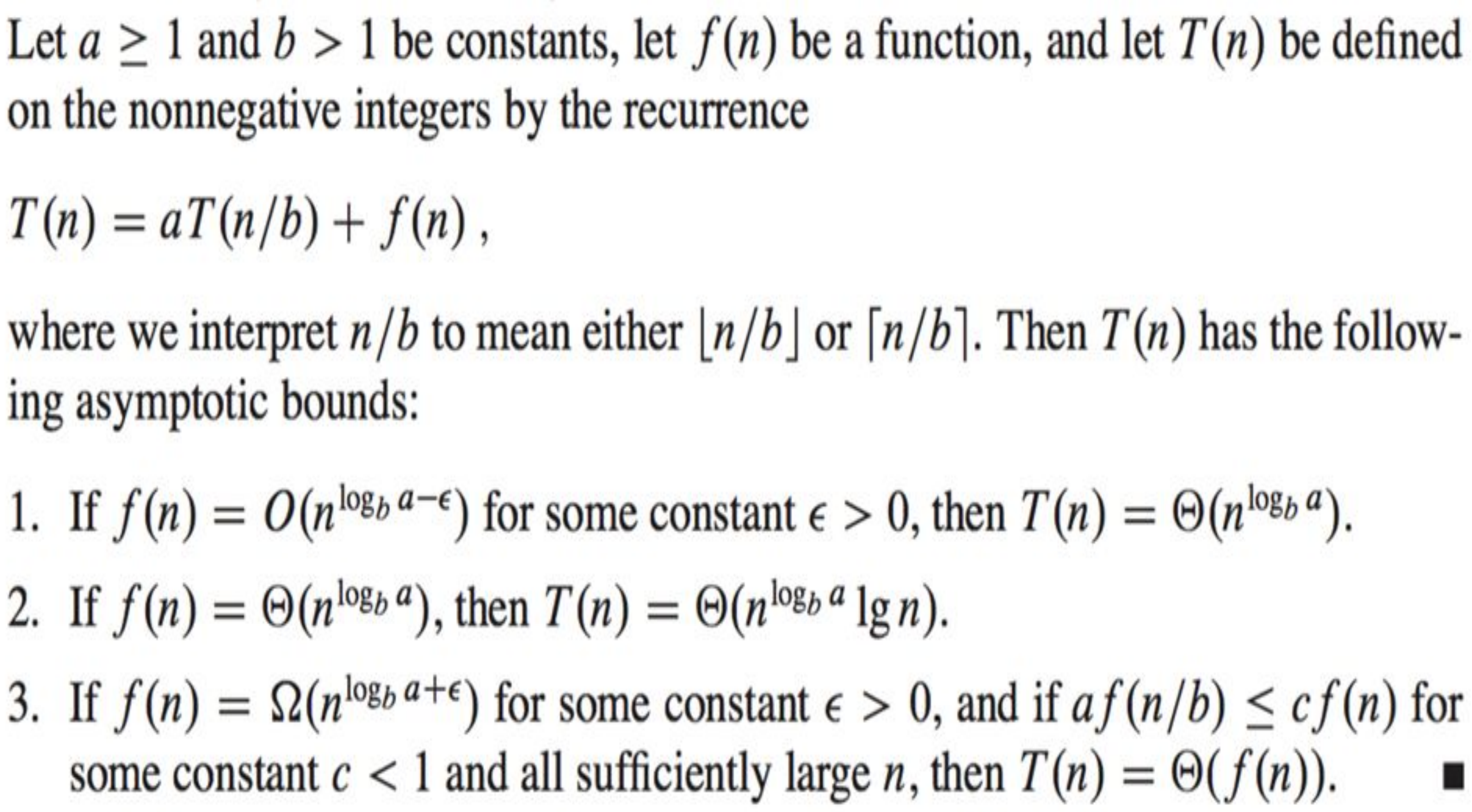
➔ if f(n) is larger (by a polynomial factor), then the solution is $T(n) = \Theta(f(n))$

➔ If they are the same size, then we multiply by a logarithmic factor. $T(n) = \Theta(n^{\log_b a} \log n)$

Complexity analysis

divide and conquer

Master Theorem “Basic Form”



Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Master Theorem

These cases are not exhaustive–

➔ it is possible for f(n) to be asymptotically larger than $n^{\log_b a}$, but not larger by a polynomial factor (no matter how small the exponent in the polynomial is).

For example, this is true when :

$$f(n) = n^{\log_b a} \log n$$

➔ In this situation, the basic master theorem **would not apply**. If you need to solve this recurrence, you'd either have to use an the **advanced version of the Master Theorem**, or apply another method such as the **recursion tree** or **substitution method**

Complexity analysis
divide and conquer
Basic Form of Master Theorem

Examples

$$T(n) = 9\,T(\tfrac{n}{3}) + n$$

$$T(n) = T(\tfrac{2n}{3}) + 1$$

$$T(n) = 3\,T(\tfrac{n}{4}) + n \log n$$

$$T(n) = 2\,T(\tfrac{n}{2}) + n \log n$$

Complexity analysis

divide and conquer

Master Theorem

Example 1

T(n) = 9 T($\frac{n}{3}$) + n.

Here a = 9, b = 3, f(n) = n, and

$$n^{\log_b a} = n^{\log_3 9} = \theta(n^2)$$

Since f(n) = O($n^{\log_3 9-\varepsilon}$) for ε = 1,

case 1 of the Master Theorem applies, so T(n) = $\theta(n^2)$

Complexity analysis
divide and conquer
Basic Form of the Master Theorem

Example 2

$$T(n) = T(\frac{2n}{3}) + 1$$

Here a = 1, b = 3/2, f(n) = 1,

$$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$$

Since f(n) = $\theta(n^{\log_b a})$,

case 2 of the master theorem applies, so T(n) = θ(log n).

Complexity analysis
divide and conquer
Master Theorem

Example 3

$$T(n) = 3\,T(\tfrac{n}{4}) + n \log n$$

Here a = 3, b = 4, f(n) = n log n,

$$n^{\log_b a} = n^{\log_4 3} = n^{0{,}793}$$

For ε = 0.2, we have f(n) = $\Omega(n^{\log_4 3+\varepsilon})$.

So case 3 applies if we can show that

$a\,.\,f(\frac{n}{b}) \le cf(n)$ for some c < 1 and all sufficiently large n.

3. $\frac{n}{4}\log\frac{n}{4} \le c\,n\log n$. Setting $c = \frac{3}{4}$ would cause this condition to be satisfied.

so T(n) = θ(n log n).

Complexity analysis

divide and conquer

Basic Form of Master Theorem

Example 4

$$T(n) = 2T(\tfrac{n}{2}) + n \log n$$

Here a = 2, b = 2, f(n) = n log n,

$$n^{\log_b a} = n^{\log_2 2} = n$$

Case 3 does not apply because even though $n \log n$ is asymptotically larger than n, it is not polynomially larger. That is, the ratio $\frac{f(n)}{n^{\log_b a}} = \log n$ is asymptotically less than n^{ε} for all positive constants ε.

Complexity analysis
Recursion

There are three methods for solving recurrences—that is, for obtaining asymptotic "Θ" or "O" bounds on the solution:

- In the substitution method, we guess a bound and then use mathematical induction to prove our guess correct.

- The recursion-tree method converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.

- The basic master theorem used to solve three cases of recurrences. In addition, the advanced master theorem which extends the basic version to handle more complex recurrences that may involve multiple terms or non-polynomial functions. This version allows for more flexibility in analyzing algorithms that do not fit neatly into the basic cases.