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)?

$$n! \cong (n/e)^n$$

➔ 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: $V = a, \quad W = b$

While $W > 1$

$V \rightarrow V + a$; $W \rightarrow W-1$

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**.

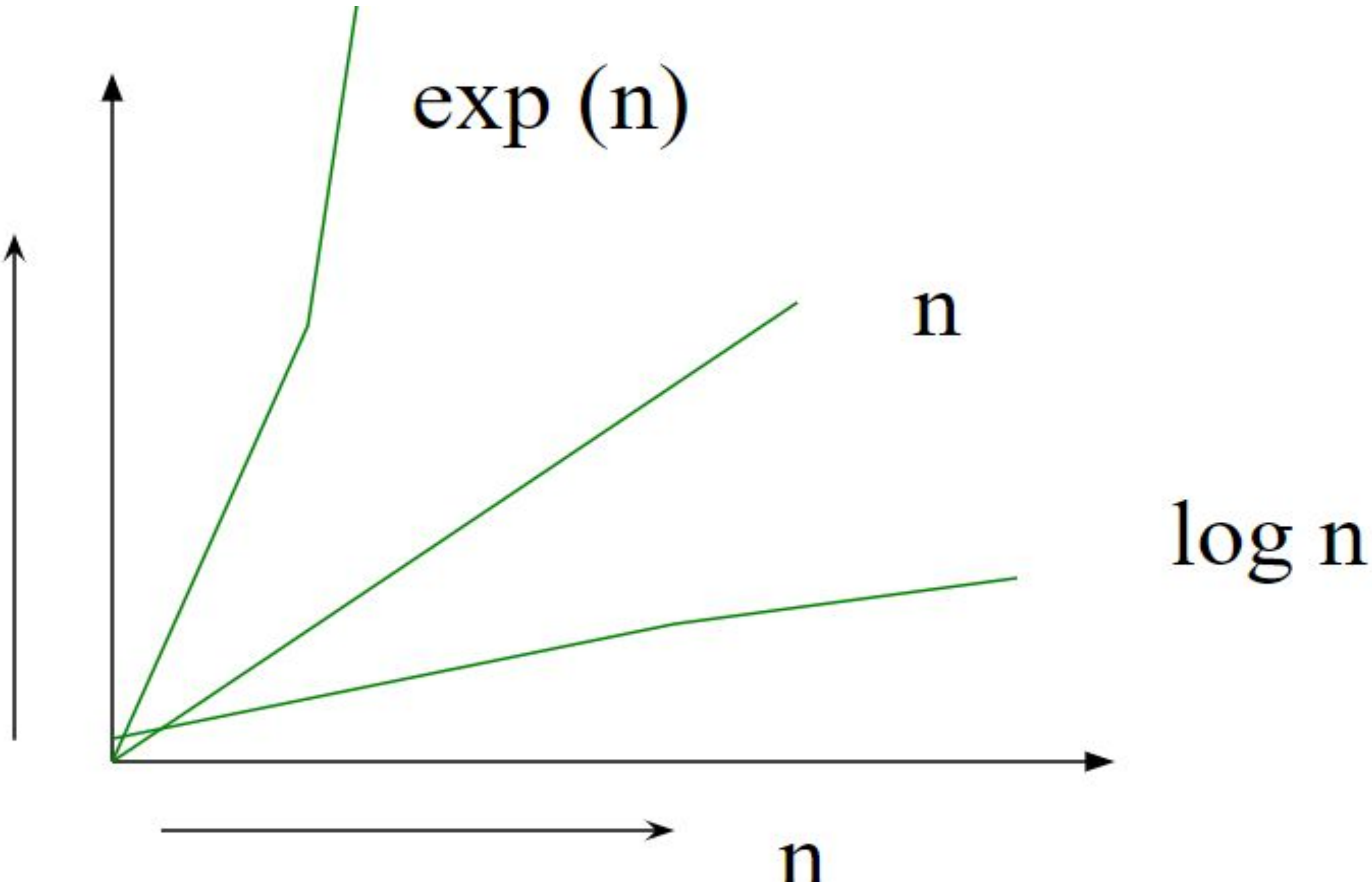
If an algorithm needs **n** basic operations and another needs **2n** basic operations, we will consider them to be in the same efficiency category.

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.

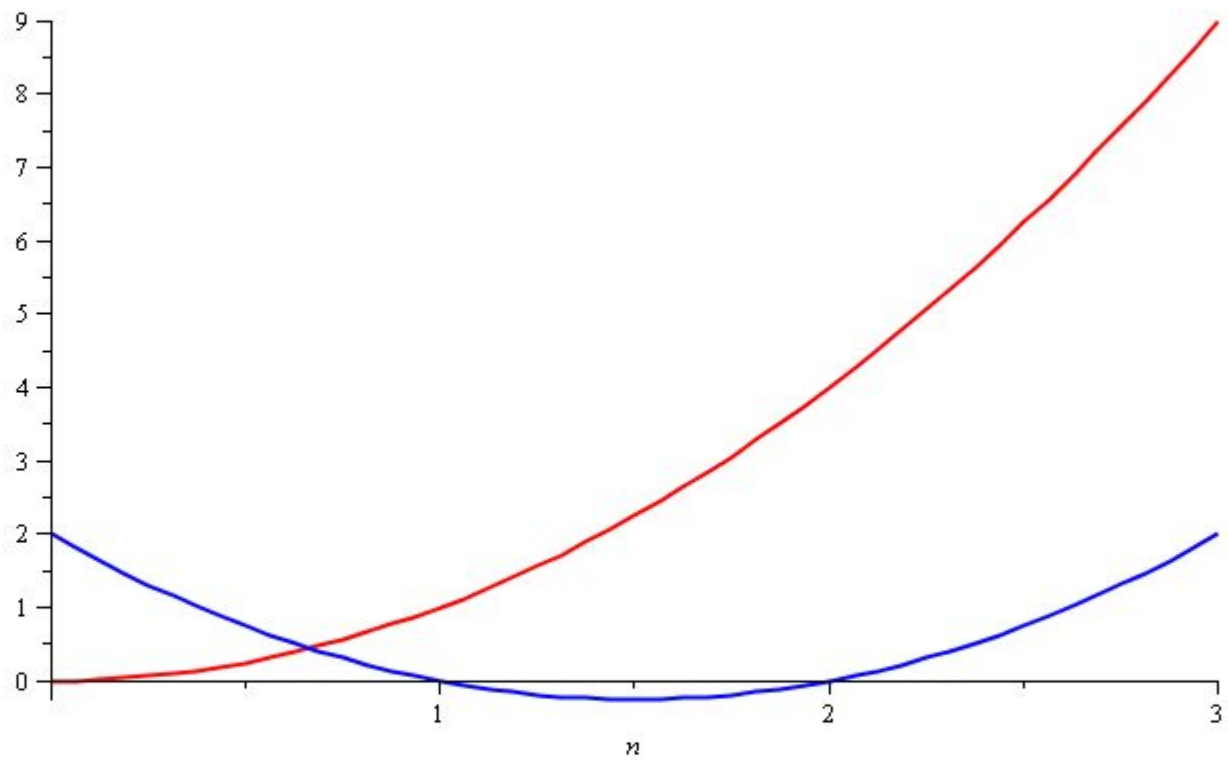


Quadratic Growth

Consider the two functions

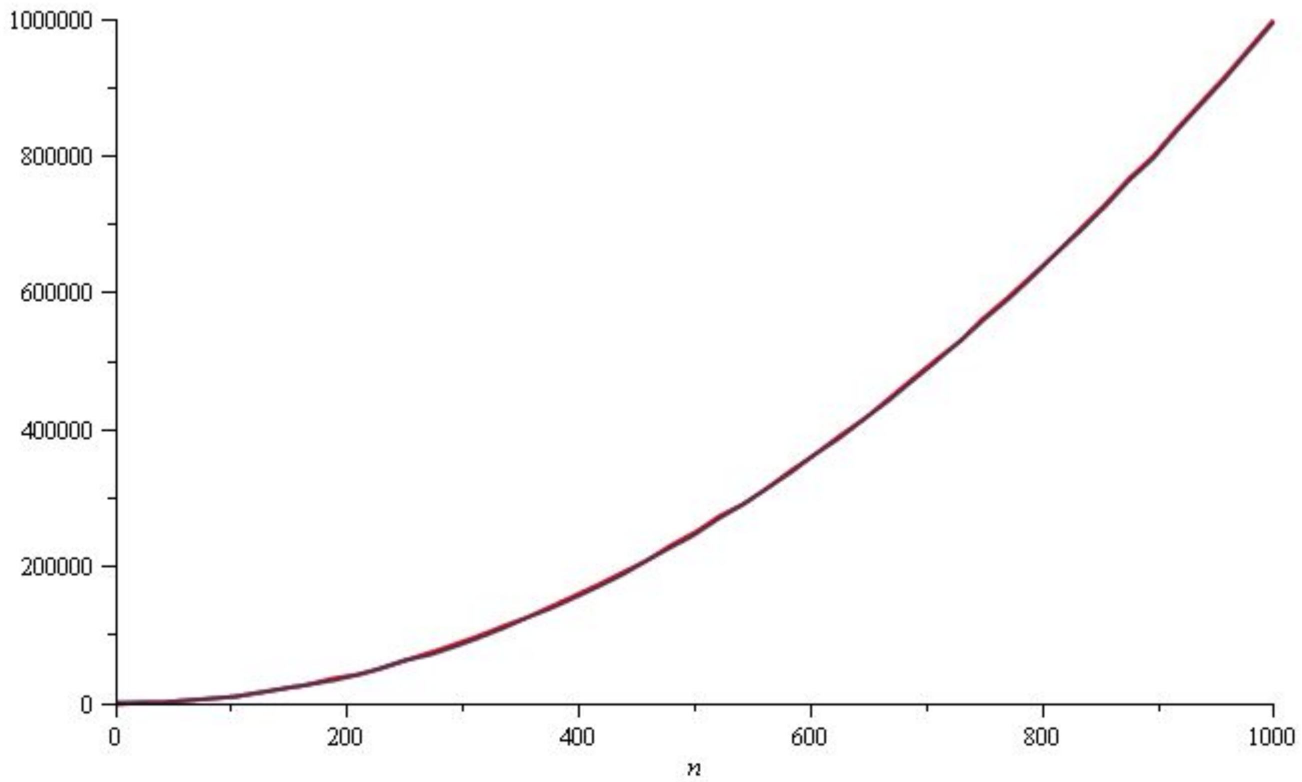
$f(n) = n^2$ and $g(n) = n^2 - 3n + 2$

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,

$f(1000) = 1\,000\,000$

$g(1000) = \quad 997\,002$

but the relative difference is very small

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

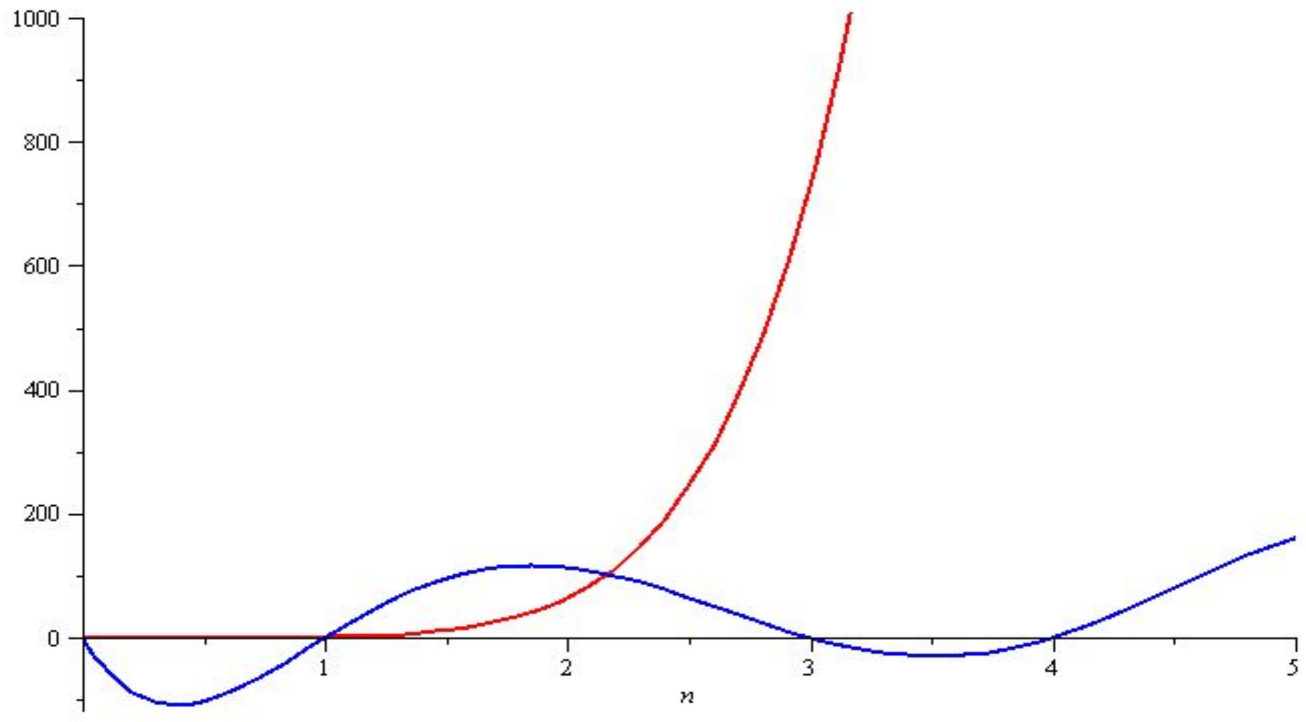
and this difference goes to zero as $n \to \infty$

Polynomial Growth

To demonstrate with another example,

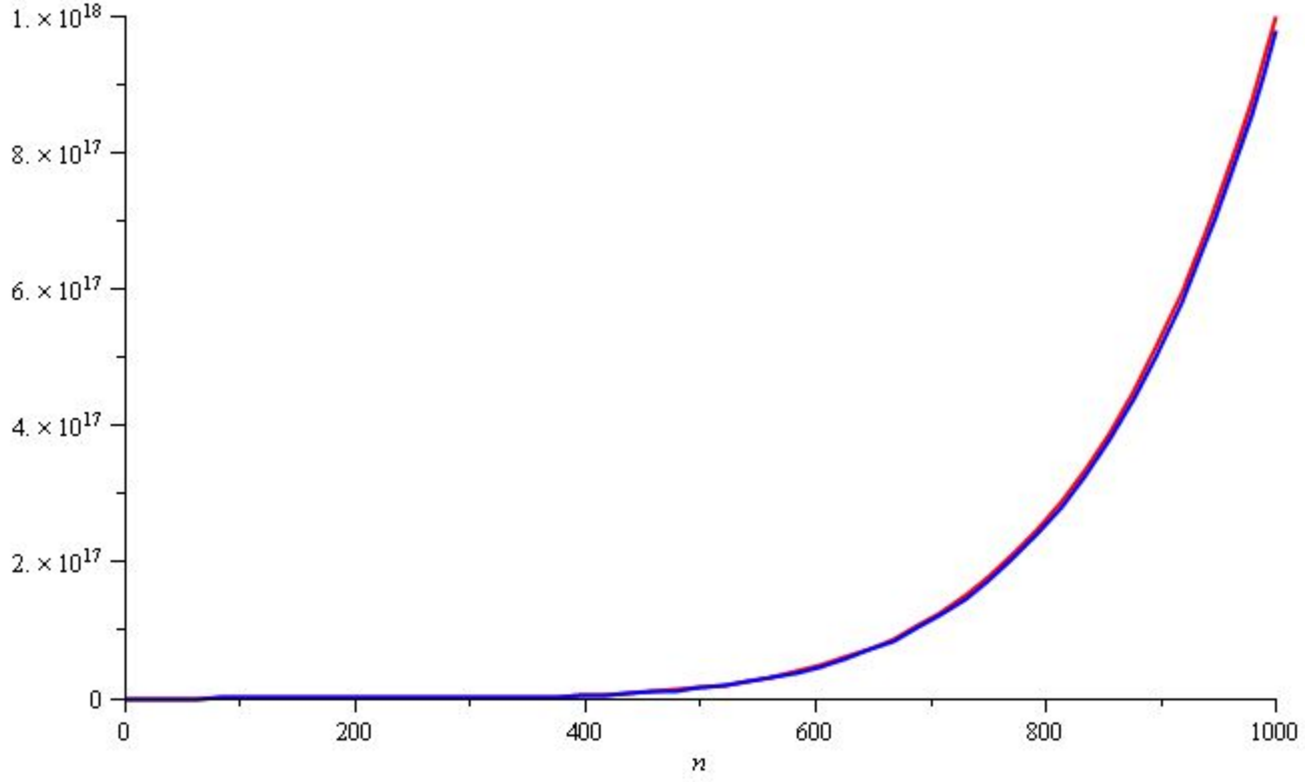
$f(n) = n^6$ and $g(n) = n^6 - 23n^5 + 193n^4 - 729n^3 + 1206n^2 - 648n$

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, $f \sim g$, if

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

- We will state that $\boldsymbol{f < g}$ 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

$\lim_{n \to \infty}$ sqrt(n) /n = 0, sqrt(n) is O(n)

$\lim_{n \to \infty}$ n/sqrt(n) = infinity, n is not O(sqrt(n))

$\lim_{n \to \infty}$ n /2n = 1/2, n is O(2n)

$\lim_{n \to \infty}$ 2n /n = 2, 2n is O(n)

$\lim_{n\to\propto} \ln(n)/n = 0,$ ln(n) is O(n)

$\lim_{n\to\propto} n/\ln(n) = \text{infinity},$ n is not O(ln(n))

$\lim_{n\to\propto} \exp(n)/n = \text{infinity},$ 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 + sqrt(n) is O(n)

$\lim_{n\to\propto} n/(\mathrm{sqrt}(n)+n) = 1,$ n is O(n+sqrt(n))

$\lim_{n\to\propto} (n + n^2)/n = \text{infinity},$ $n + n^2$ 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

$$\text{for all } n \geq n_o, \quad f(n) \geq c g(n).$$

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"
- $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ 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 $f(n) \neq \theta(g(n))$.

"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) = **Θ**(g(n))

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

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

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

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

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

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

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

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 $f(n) = \mathbf{\Theta}(g(n))$ seems to describe an equivalence relation:

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

Consequently, we can group all functions into equivalence classes, where all functions within one class are big-theta $\mathbf{\Theta}$ of each other

Big-Θ as an Equivalence Relation

For example, all of

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

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

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

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
$2^n, e^n, 4^n, \ldots$	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)

$\lim_{n\to\infty} \ln(n)/n = 0$, $\ln(n)$ is $o(n)$

$\lim_{n\to\infty} n/\ln(n) = \text{infinity}$, n is $\Omega(\ln(n))$

$\lim_{n\to\infty} \exp(n)/n = \text{infinity}$, $\exp(n)$ is $\Omega(n)$

$\lim_{n\to\infty} n/\exp(n) = 0$, n is $o(\exp(n))$

$\lim_{n\to\infty} (n+\text{sqrt}(n))/n = 1$, $n + \text{sqrt}(n)$ is $\theta(n)$

$\lim_{n\to\infty} n/(\text{sqrt}(n)+n) = 1$, n is $\theta(n+\text{sqrt}(n))$,

$\lim_{n\to\infty} (n + n^2)/n = \text{infinity}$, $n + n^2$ is $\Omega(n)$

$\lim_{n\to\infty} n/(n + n^2) = 0$, 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 $\mathbf{O}(n^d)$ 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

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

Rule 5

If $f_1(n) = \mathbf{O}(g(n))$ but $f_2(n) = \mathbf{o}(g(n))$, then

$$f_1(n) + f_2(n) = \mathbf{O}(g(n)).$$

Rule 6

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

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

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 $T\text{avg}(N)$ and $T\text{worst}(N)$, as the average and worst-case running time, resp., used by an algorithm on input of size N. Clearly, $T\text{avg}(N) \leq T\text{worst}(N)$.
- 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++)
3.   sum → sum + 1;
4.   diff → diff - 1;
	• in every iteration of first loop
	    Line 3 takes 2 basic steps.
	    Line 4 takes 2 basic steps
	First loop runs N times

5. for (k=0: k < 3N; k++) 6. sum → sum - 1;	• in every iteration of second 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 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

1. sum = 0; 2. for (i=0; i < N; i++) 3. for (j=0; j < N; j++) 4. sum → sum + 1;	Outer loop: N iterations Inner loop: O(N) **Overall: $O(N^2)$**
1. for (i=0; i < N; i++) 2. a[i] = 0; 3. for (i=0; i < N; i++) 4. for (j=0; j < N; j++) 5. a[i] = a[j] + i+j ;	First loop O(N) Inner loop: O(N) Outer loop: N iterations **Overall: $O(N) + O(N^2)$ So** **$O(N^2)$**

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, …., 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 i^{th} and j^{th} 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.