CHAPTER 1: ALGORITHMIC COMPLEXITY

Contents

1. Reminders

Intuitively, an algorithm is any unambiguous computational procedure that can take one or more input values and produce one or more output values. It is a tool for solving a well-specified problem whose statement indicates the relationship between inputs and outputs. The algorithm can also be defined more simply as a finite sequence of unambiguous instructions that can be executed automatically. Some classic examples of algorithms include:

- Matrix multiplication
- Solution of linear equations system
- Shortest path (Dijkstra algorithm)
- Sorting
- Searching a word in a dictionary
- Web search

As an example, we formally define the sorting algorithm as follows:

Input: A sequence of n numbers $\{a_1, a_2, \dots, a_n\}$.

Output: A permutation $\{a'_1, a'_2, \dots, a'_n\}$ such as $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

For example, the sequence {1,18,5,4,33} must be transformed into {1,4,5,18,33} by the sorting algorithm. The input sequence is called **instance** of the sorting problem.

- **Algorithm description.** There are 3 ways to describe an algorithm:

  a- Operating principle (using natural language)
  Example: sequential (or linear) search for an item in an unsorted list. We successively compare all the elements of the list with the target element until finding this one or arriving at the end of the list.

  b- Pseudo code or formalism which can express:
  - Assignments: ←
  - Conditional statements: **If** … **Then** … **Else**… **End if**
  - Loops: **While** condition …..**End loop**

Example: Sequential search in an Array

```
// Input: n items stored in an array T.
// The target item noted key.
// We use a Boolean variable found and an integer i.

i ← 1;
found ← false;
While ((not found) and (i <= n))
          If (T[i] = key) Then found ← true;
          Else i ← i+1;
End loop
Return found
```

c- Programming Language such as Python, C, C++ or Java.

Example:

Insertion sort with Java:

```
public void insertionSort()
{ int i, j;
  for (j=1; j<nElems; j++)
  {
    long temp = A[j];
    i = j;
    while (i>0 && A[i-1]>=temp)
    {
      A[i] = A[i-1];
      --i;
    }
    A[i] = temp;
  }
}
```

with Python:

```
def insertionSort():
   for j in range(1,nElems):
      temp = A[j]
      i = j
      while i>0 and A[i-1]>=temp:
         A[i] = A[i-1]
         i -= 1
      A[i] = temp
```

Notes

- The three previous methods of describing an algorithm are increasingly precise and therefore less and less ambiguous.
- A good way to solve a problem consists in linking the three methods, that is: before programming, we must give the principle and then express it in pseudo-code.
- It is necessary to prove the correctness of an algorithm, i.e. to ensure that it is correct in all cases. There are algorithm proof techniques but are beyond the level of this course.

- **Algorithm implementation**

The algorithm implementation consists of translating it into a programming language with the aim of executing it on a computer. The implementation requires the choice of data representation or data structure.

2. Algorithm quality and features

Solving a problem can often be performed by several algorithms and for each algorithm there are several data structures. The algorithm as well as the data structure associated with its implementation is characterized by:

- Its **simplicity**: a simple algorithm is easy to understand, to implement and generally to prove.

- Its **running time** (time complexity or temporal complexity): We prefer an algorithm that runs in one second to another that takes an hour on the same machine. Similarly, we reject any algorithm whose execution time exceeds a century!!

- Its **memory requirement** (space complexity or spatial complexity): this memory depends on both the algorithm and the structures of the chosen data. The memory requirement must never exceed a certain limit which depends on the machine used. This has the effect of putting a limit on the size of the problems that can be solved.

Data structure

Data structures are a main feature of algorithms and can have a direct impact on their complexities. Data structures specify how to **represent problem data** that can be solved by computer using an algorithm. The data structure used must be economic in term of memory space and easy to access. A data structure can be chosen independently of the programming language which is used for writing the program manipulating its data. This language is supposed to provide, in one way or another, the necessary mechanisms to define and manipulate these data structures. Modern programming languages can manipulate computer memory in the form of isolated variables, tables and pointers indicating the data location in memory or more complex predefined structures.
In the case of isolated variables or static arrays, the place in memory is allocated by the compiler and cannot be modified during program execution. However in the case of dynamic arrays or linked lists, the memory allocation is done during execution and therefore it can increase or decrease as needed.

Classical data structures can be classified into three categories:

- **Linear or sequential structures**: Data is organized in the form of a list sequentially. These are structures that can be represented by linear lists such as arrays or linked lists having a single dimension. We find also in this category stacks and queues. Stacks follow a particular order in which the operations are performed; the order is described by LIFO (Last In First Out) which implies that the element that is inserted last, comes out first, while queues follow FIFO order (First In First Out) which implies that the element that is inserted first, comes out first.

- **Trees** and in particular binary trees.

- **Relational structures** which take into account existing relations between entities which they describe.

3. Definition of algorithmic complexity

The algorithm complexity analysis aims to estimate the quality of an algorithm (associated with a data structure). This measure allows the comparison of two algorithms without having to implement them. Complexity can be studied in two aspects: temporal (running time) and spatial (memory space).

- **Temporal complexity**

If we take into account for the estimation of the complexity the computer resources such as the clock frequency, the number of processors, the disk access time etc., we immediately realize the complication of such a task. It's why we often estimate **the relationship between the data size and the running time independently of the architecture used.**

We define algorithmic complexity as a measure of how long an algorithm would take to compute given an input of size n. More precisely, it's a measure of the number of elementary operations it performs on an input of size n.

If an algorithm has to scale, it should compute the result within a finite and practical time bound even for large values of n. For this reason, complexity is calculated **asymptotically** as n approaches infinity. Analysis of complexity is helpful when comparing algorithms or seeking improvements. Algorithmic complexity falls within a branch of theoretical computer science called computational complexity theory. It's important to note that we're concerned about the order of an algorithm's complexity, not the actual execution time in terms of milliseconds.

In a formal way, we can also define the complexity of an algorithm $\mathcal{A}$ as being any **order of growth** of the number of **elementary machine operations** carried out during the execution of $\mathcal{A}$. These notions will be defined in the following section.

- **Space complexity**

While complexity is usually in terms of time, sometimes complexity is also analyzed in terms of memory space, which translates to the algorithm's memory requirements. The space complexity will be depicted in section 6.

In what follows, we will mainly address temporal or time complexity.

4. Complexity calculation

4.1. Complexity analysis model

It is a simplified model that takes into account technological resources and their associated costs. We will take as a reference a model of random access machine with a single processor where the operations are executed one after the other without simultaneous operations. Depending on the type of instructions, the complexity is evaluated as follows:

Sequence

$\mathcal{A}$: | $\mathcal{I}$;
| $\mathcal{K}$; $\Rightarrow T_{\mathcal{A}}(N) = T_{\mathcal{I}}(N) + T_{\mathcal{K}}(N)$

Alternative

$\mathcal{A}$: | **If** $\mathcal{C}$ **Then** $\mathcal{I}$ **Else** $\mathcal{K}$ $\Rightarrow T_{\mathcal{A}}(N) = T_{\mathcal{C}}(N) + max\{T_{\mathcal{I}}(N), T_{\mathcal{K}}(N)\}$: Worst case
$\Rightarrow T_{\mathcal{A}}(N) = T_{\mathcal{C}}(N) + min\{T_{\mathcal{I}}(N), T_{\mathcal{K}}(N)\}$: Best case

Loops

$\mathcal{A}$: | **nb** $\circlearrowleft$ $\mathcal{B}$ $\Rightarrow T_{\mathcal{A}}(N) = nb \times T_{\mathcal{B}}(N)$

Recursivity

Write the recurrence formula and deduce the complexity (see section 4.5)

4.2. Elementary machine operations

The following operations are called elementary machine operations:

- memory access to read or write the value of a variable or an array element;
- an arithmetic operation between integers or reals such as addition, subtraction, multiplication, division or calculation of the remainder of an integer division;
- a comparison between two integers or reals.

Example:

The instruction c ← a + b; requires four elementary machine operations:

1- a memory access for reading the value of a,
2- a memory access for reading the value of b,
3- an addition of a and b,
4- a memory access for writing the new value of c.

Notes

- It is recommended to identify the fundamental operations of an algorithm. Their number is mainly involved in complexity analysis. Here are some examples:

Algorithm	Main Operations
Search	Comparisons
Sort	Comparisons and permutations
Matrix multiplication	Additions and multiplications

- Focus on loops and recursivity; the final complexity of an algorithm generally comes from these two types of instructions.

4.3. Illustration example: Insertion sort

As an illustration example, we consider insertion sort algorithm of an array A of size N. We associate to each line i of the code its cost (or execution time) Ci. The analysis can be done as follows:

```
void insertionSort (int *A, int N)          // cost        Number of times
   {                                        // ==========================
   int j, temp, i;
   for (j=2 ; j<=N ; j++)                   // cost C1 ........ N
      {
        temp = A[j];                        // cost C2 ........ N-1
        i = j-1;                            // cost C3 ........ N-1
        while (i>0 && A[i]>temp)            // cost C4 ........ 2+3+...+N
          {
            A[i+1]=A[i];                    // cost C5 ........ 1+2+...N-1
            i=i-1;                          // cost C6 ........ 1+2+...N-1
          }
        A[i+1]=temp;                        // cost C7 ........ N-1
      }
   }
```

To finalize the analysis of insertion sort complexity, mathematical notions are necessary, in particular numerical sequences and the notion of order of growth.

4.4. Mathematical reminders

Numerical sequences

- **Arithmetic sequence:**

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

- **Geometric sequence:**

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$$

- **Harmonic sequence:**

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

The complexity of insertion sort algorithm is therefore:

$$T(N) = (\frac{C_4 + C_5 + C_6}{2})N^2 + (C_1 + C_2 + C_3 + \frac{C_4 - C_5 - C_6}{2} + C_7)N - (C_2 + C_3 + C_4 + C_7)$$

And since complexity is calculated **asymptotically** as N approaches infinity, we can note the complexity with the Landau notation or Big O notation (See reminder below) as follows:

$$\mathbf{T(N) = O(N^2)}$$

Note

T(N) represents the complexity in the **worst case** because we did not take into account in our calculation the value of the logical expression of the condition (A[i] > temp) which determines the number of executions of the inner loop. The insertion sort algorithm can therefore take less time for arrays with a particular structure and therefore complexity is a random variable. This is why we consider the complexity in **average case** that can be calculated as the expectation of this random variable.

Orders of growth

- Θ (theta) Notation: let g be a positive function of an integer variable n. **Θ(g(n))** denotes the set of positive functions of the variable n, for which there exist two constants c1, c2 and an integer n_0, satisfying the relation:

$$\mathbf{0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \forall\ n \geq n_0}$$

We write roughly f(n) = Θ(g(n)) to express that f ∈ Θ(g(n)) (although Θ(g(n)) is a set and f is an element of this set).

Example:

Let $g(n) = n^2$ and $f(n) = 50n^2+10n$.

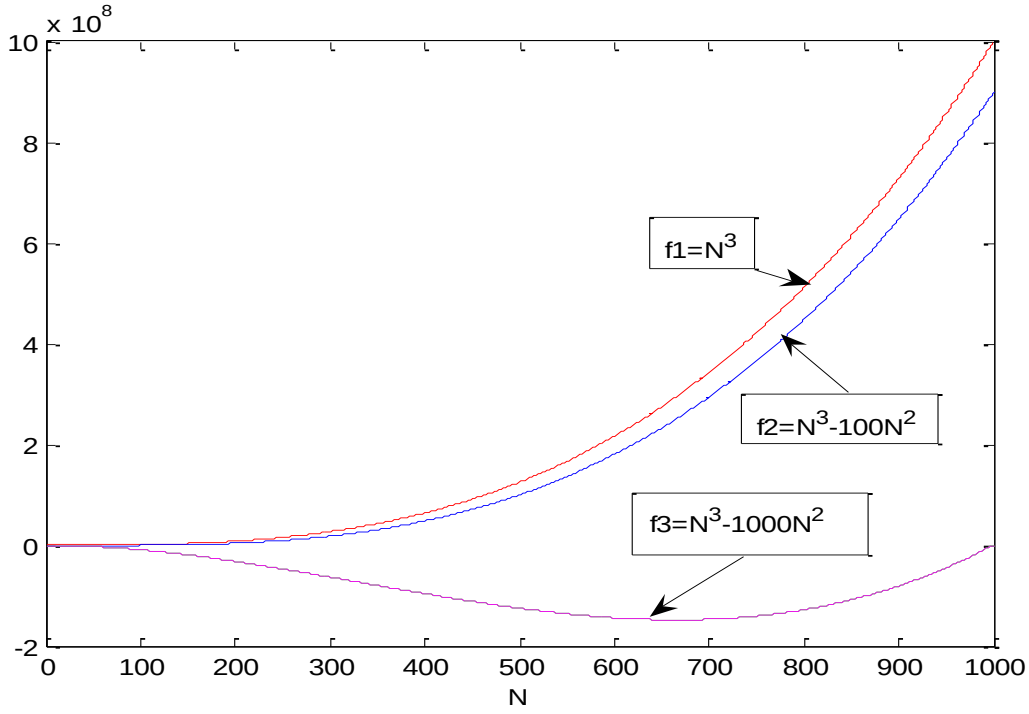
We need to find c_1, c_2 and n_0 such as: $c_1n^2 \leq 50n^2+10n \Rightarrow c_1 \leq 50$,

$50n^2+10n \leq c_2n^2 \Rightarrow c_2 \geq 50$.

So we have f(n) = Θ(g(n)) = $\Theta(n^2)$

The following figure shows the graphs of functions f1, f2 et f3. We can easily verify that f2 = Θ(f1(N)) and also f3 = Θ(f1(N)).

We can interpret the relation f(n)=Θ(g(n)) as follows: for n large enough, the function f is both upper and lower bounded by the function g. We can say that the functions f and g are equal, up to a constant.



- Big O Notation: When the function f is only upper bounded by g we use the notation Big O. **O(g(n))** denotes the set of positive functions of the variable n, for which there exists a constant c and an integer n_0, satisfying the relation: $\mathbf{0 \leq f(n) \leq c\, g(n)} \quad \forall\ \mathbf{n \geq n_0}$

The relation f(n) = O(g(n)) indicates that the function f is upper bounded by the function g for sufficiently large values of the argument n. So f(n) = Θ(g(n)) implies that f(n) = O(g(n)). Formally, we can write $\Theta(g(n)) \subseteq O(g(n))$.

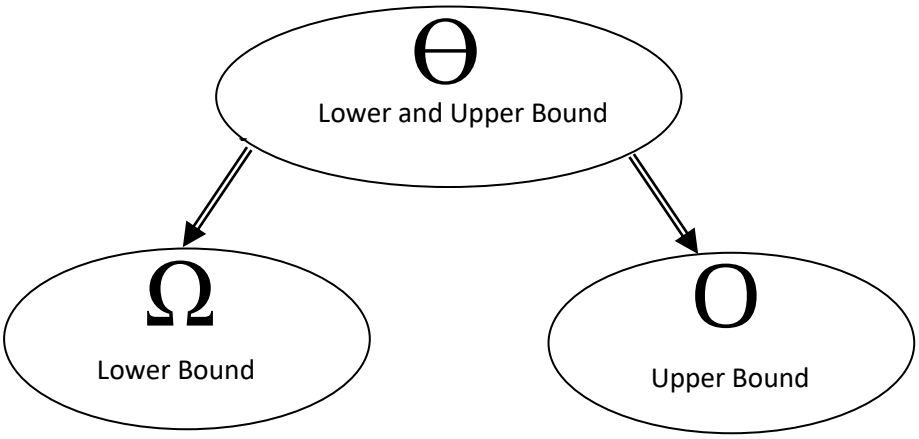
Example: $100n^2 + 10n = \mathbf{O}(n^2)$ but we can also write $\mathbf{100n = O(n^2)}$ **!** Because 100n is also upper bounded by n^2.

- Ω (Big omega) Notation: Ω(g(n)) denotes the set of positive functions of the variable n, for which there exists a constant c and an integer n_0, satisfying the relation : $\mathbf{0 \leq cg(n) \leq f(n)} \quad \forall\ \mathbf{n \geq n_0}$

The relation f(n) = Ω(g(n)) indicates that the function f is lower bounded by the function g for sufficiently large values of the argument n. Therefore, **f(n) = Θ(g(n)) implies that f(n) = Ω(g(n))**. Formally, we can write: $\mathbf{\Theta(g(n)) \subseteq \Omega(g(n))}$. The following theorem follows immediately from the given definitions:

Theorem: f(n) = **Θ**(g(n)) if and only if: f(n) = **Ω**(g(n)) and f(n) = **O**(g(n))

The three previous notations can be summarized by the following diagram:



Some properties of Big O notation

1. Constant factor can be ignored (e.g. : $3x^2 = O(x^2)$)
2. A large power of n grows faster than a smaller power ($n^r = O(n^s)$ if $0 \leq r \leq s$)
3. The growth rate of a sum of terms is the growth rate of the fastest growing term. (e.g. : $an^3 + bn^2 = O(n^3)$)
4. Exponential functions are faster growing than polynomials.
5. All logarithms have the same growth rate.

IMPORTANT

The complexity of an algorithm can in some cases be easily found if we analyze the algorithm and identify its fundamental operations. Thus for insertion sort, we can analyze only comparisons as follows:

The number of comparisons performed at iteration i is at most i. The total number of comparisons is:

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

4.5. Complexity and recursivity

To calculate the complexity of a recursive algorithm, it is necessary to determine the recurrence formula corresponding to the complexity and then to develop this formula. Let us illustrate this with the calculation of the complexity of the following recursive function (factorial of an integer n):

```
int fact(int n) {
 if (n==0) return 1;
 else return n * fact(n-1);
}
```

In this case, the complexity parameter (size of the problem) is the value of n. Also there is only one execution case (best and worst case are the same). We therefore notice that:

- If $n \neq 0$, computing factorial of n costs a memory access, an integer comparison, two arithmetic operations (subtraction to compute $n-1$ and multiplication) and the calculation of the factorial of $n-1$, that is calculating factorial of $n-1$ and 4 elementary machine operations.

- If $n = 0$, the cost is only one memory access and one comparison (two elementary machine operations).

We therefore set the following recurrence formula; let's consider T(n) the complexity of the function fact:

$$T(n) = \begin{cases} 2 & if\ n = 0 \\ T(n-1) + 4 & if\ n \neq 0 \end{cases}$$

The development of this formula can be done as follows:

$T(n) = T(n-1) + 4 = T(n-2) + 8 = \cdots = T(n-p) + 4p$

Then we set $n - p = 0$ and we find: $T(n) = 4n + 2 = O(n)$

There are several techniques for solving recurrence equations (generating function, rational fractions, Z transform, etc.). The following theorem is the solution of the main recurrence equations.

Theorem:
Let $T(n)$ a function defined by the following recurrence equation, where $b \geq 2$, $k \geq 0$, $a > 0$, $c > 0$:

$$T(n) = aT(n/b) + cn^k$$

If $a > b^k$ then $T(n) = \Theta(nlog_b(a))$
If $a = b^k$ then $T(n) = \Theta(n^k log(n))$
If $a < b^k$ then $T(n) = \Theta(n^k)$

The following table summarizes the solution of the main recurrence equations frequently encountered in the complexity of algorithms:

Recurrence equation	Solution
$T(n) = T(n-1) + b$	$T(n) = O(n)$
$T(n) = aT(n-1) + b\ , a \neq 1$	$T(n) = O(a^n)$
$T(n) = T(n-1) + an + b$	$T(n) = O(n^2)$
$T(n) = T(n/2) + b$	$T(n) = O(logn)$
$T(n) = aT(n/2) + b\ , a \neq 1$	$T(n) = O(a^{\log(n)})$
$T(n) = T(n/2) + an + b$	$T(n) = O(n)$
$T(n) = 2T(n/2) + an + b$	$T(n) = O(nlogn)$

5. Examples of complexity calculation

Linear complexity O(N)

- Sequential search: if the element doesn't exist, all the N elements are explored, resulting in a worst case complexity of O(N).
- Factorial calculation (iterative and recursive).

Constant complexity O(1)

- Loop with constant number of iterations: **`for (i = 1 ; i <= 100 ; i++)`**
- Hash coding without collisions.

Logarithmic complexity O(logN)

- Binary search

```
istart = 1;
iend = n;
found = false;
while (istart<=iend && !found) {
 imedian = (istart+iend)/2;
 if (A[imedian]==e) found = true;
 else if (A[imedian]>e) iend = imedian-1; else istart = imedian+1;
}
```

The number of iterations in the worst case is equal to $log_2(n)$; indeed the search space evolves as follows: n, $n/2$, $n/4$, …, $n/2^p$, until only one cell remains. If we set $n/2^p = 1$, we deduce the number of iterations p equal to $log_2(n)$.

Quadratic complexity O(N²)

- Two nested linear loops
- Insertion sort (worst case)

Exponential complexity O(2^N)

- Hanoi tower game

```
Procedure Hanoi(N, A, C, B)
{ if N≠0 then
    Hanoi(N-1,A,B,C);
    Move disk from A to C
    Hanoi(N-1,B,C,A);
  endif
}
```

The complexity is obtained by developing the formula: $T(N) = 2T(N-1) + O(1)$ and $T(0) = O(1)$

6. Space complexity

So far, we have approached algorithmic complexity only through the prism of computation time. For a calculation to be usable, it is indeed necessary that it be executed quickly. But there is another critical resource: memory. We are therefore interested in this section in the spatial complexity, in other words, the memory that the algorithm will require. This can also depend, as for the time complexity, on the size of the instance to be processed by the algorithm.

Let's take as an example, an algorithm performing n^4 operations which can possibly still be considered reasonably efficient in terms of running time since the time required for a current machine performing 10^{11} operations per second to run the algorithm on an entry of size 10 000 is about a day. But if at each step it uses a new memory space then it will need 10^{15} memory spaces to perform its calculation. This is currently prohibitive, at least if we limit ourselves to RAM in order to avoid slow disk access. However, memory is no longer a really limiting parameter these days, the price of RAM having dropped enormously in twenty years (moreover virtual memory is another answer to this problem).

There is an (unofficial) rule that says that to save computing time, one must use more memory space. This is illustrated by the classic example of swapping two integer variables x and y :

```
// swapping by using an auxiliary variable z
z = x;
x = y;
y = z;

// swapping without auxiliary variable
x = y - x;
y = y - x;
x = y + x;
```

The first method use an auxiliary variable and make 3 assignments (space complexity = 1, time complexity = 3) while the second method use only the 2 variables x and y, but make 3 assignments and 3 operations (space complexity = 0, time complexity = 6).

Measuring spatial complexity is usually done by estimating the amount of memory used by the program as illustrated in the very simple example of a function to determine the index of the first negative element in an array of n integers.

```
int firstNegative(vector <int> T)        O(n)
{  int n = T.size();                     O(1)
   int i = 0;                            O(1)
   while (i < n)
   {  if (T[i]<0) return i;
      i++;
   }
   return -1;
}
```

The space complexity of firstNegative function is in fact O(1), because the array T already exists as input.

7. Different forms of complexity

It is obvious that the complexity of an algorithm may not be the same for two different data sets. This can have consequences on the choice of the best algorithm for a given problem. In practice, we are interested in the following forms of complexity:

7.1. Worst case complexity

We consider in this case the greatest number of elementary operations carried out on **the set of all problem instances**; we are looking for an upper bound that is reached even for instances with a very low, or even zero, probability.
This form of complexity is easier to calculate but can lead to an erroneous choice of an algorithm. Indeed the worst case can be a very rare case!

7.2. Best case complexity

We consider in the best case complexity the smallest number of elementary operations; i.e we are looking for a lower bound of the complexity instead of an upper bound.
This form of complexity can be considered as a complement to worst case complexity but does not offer any guarantee to the user.

7.3. Average case complexity

This is the true complexity of an algorithm. This involves calculating the average (mathematical expectation) of the number of elementary operations performed on all problem instances. This calculation is generally difficult and often delicate to implement because it is necessary to know the probability of each of the data sets in order to be able to calculate the average complexity.
This form of analysis has been the subject of numerous research works and can explain, on one hand, the behavior of certain algorithms in practice; and on the other hand, the choice of algorithms for huge size problems such as machine learning problems in artificial intelligence.

8. Polynomial complexity and exponential complexity

An algorithm is considered practical, relative to a class of problems, if it is able to solve any instance in the worst case (or average case) in polynomial time, i.e., its complexity is $O(N^k)$, where k is an integer.

In reality, a complexity is said to be polynomial if it can be bounded by a polynomial, such as:

$O(1)$	(Constant complexity)
$O(log(N))$	(Logarithmic complexity)
$O(\sqrt{N})$	(Root complexity)
$O(N)$	(Linear complexity)
$O(NlogN)$	(Loglinear complexity)
$O(N^2)$	(Quadratic complexity)
$O(N^3)$	(Cubic complexity)
…	

On the other hand, a complexity is said to be exponential ($O(k^N)$ in general), if it cannot be bounded by a polynomial, such as:

O(N^{logN})	(Sub-exponential complexity)
$O(2^N), O(3^N)$	(Exponential complexity)
$O(N!)$	(Factorial complexity)
$O(2^{2^N})$	(Double exponential complexity)

A practical algorithm is polynomial. This efficiency criteria is confirmed by practice:

- An exponential exceeds any polynomial for large values of n. For example, 1.1^n first grows slowly, but ultimately exceeds n^{100}. Further, it is rare to find polynomial complexities of order higher than n^5 in practice.

- The set of polynomials has interesting closure properties. Adding, multiplying and composing polynomials gives polynomials. We can thus build large polynomial algorithms from smaller ones.

It's hard for the human mind to estimate exponential complexities. The following table illustrates theses complexities compared to polynomial complexities. Calculation times assume 0.1 nanoseconds per instruction (equivalent to a processor running at over 6 GHZ). Unfilled cells have durations greater than 1000 billion years (greater than the estimated age of the universe!).

Complexity \ Size	20	50	100	200	500	1000
$10^7 nlog_2(n)$	0.09 s	0.3 s	0.7 s	1.5 s	4.5 s	10 s
$10^6 n^2$	0.04 s	0.25 s	1 s	4 s	25 s	100 s
$10^5\ n^3$	0.08 s	1.25 s	10 s	80 s	21 min	2.7 h
$n^{\log n}$	0.04 ms	0.4 s	32 min	1.2 years	5. 10^7 years	--
2^n	100 µs	31 h	--	--	--	--
$n!$	7.7 years	--	--	--	--	--

These calculations show that exponential growth quickly makes the use of an exponential algorithm prohibitive if large data are to be processed. It also suggests that technological progress that would make our current computers faster would not qualitatively change the conclusion of the previous sentence unless quantum computers see the light of day in the future!