TEMASEK JUNIOR COLLEGE

Temasek Junior College
JC H2 Computing
Problem Solving & Algorithm Design 13
Algorithm Complexity

1 Measuring the Efficiency of Algorithms

Selection of algorithms: settle for a space/time trade-off.

- An algorithm can be designed to gain faster run times at the cost of using extra space (memory) or the other way around.
- Some users might be willing to pay for more memory to get a faster algorithm, whereas others would rather settle for a slower algorithm that economizes on memory.
- Although memory is now quite inexpensive for desktop and laptop computers, the space/time trade-off.

1.1 Measuring the Run Time of Algorithms

- Measuring the time cost of an algorithm is to use the computer's clock to obtain an actual run time.
- This process, called *benchmarking* or *profiling*, starts by determining the time for several different data sets of the same size and then calculates the average time. Next, similar data are gathered for larger and larger data sets. After several such tests, enough data are available to predict how the algorithm will behave for a data set of any size.

Example 1 Counts from 1 to a given number (N : `Problem Size`).

- `N` starts from 1000 or 10,000,000, time the algorithm, and output the running time to the terminal window.
- Double the `N` and repeat this process.
- Repeat five such increases, a set of results can be generalized.

Code for the tester program:

```
"""
File: AD13_TC_Exp1_Times.py
Prints the running times for problem sizes that double, using a single
loop.
"""
import time
N = 10000000    # or 1000 Problem size
print("%12s%16s" % ("Problem Size N", "Seconds"))
for count in range(5):
    start = time.time()
    # The start of the algorithm
    work = 1
    for k in range(N):
        work += 1
        work -= 1
    # The end of the algorithm
    elapsed = time.time() - start
    print("%12d%16.12f" % (N, elapsed))
    N *= 2
```

Problem Size N	Seconds
10000000	1.4844105
20000000	3.5066576
40000000	11.638667
80000000	21.087813
160000000	39.211929

Figure 1 The output of Example 1

- `time()` function in the time module to track the running time. This function returns the number of seconds that have elapsed between the current time on the computer's clock and January 1, 1970. Thus, the difference between the results of two calls of `time.time()` represents the elapsed time in seconds.
- The program does a constant amount of work, in the form of two extended assignment statements, on each pass through the loop. This work consumes enough time on each iteration so that the total running time is significant but has no other impact on the results.

A quick glance at the results reveals that the running time more or less doubles when the size of the problem doubles. The running time prediction for a problem of size 32,000,000 would be approximately **80** seconds.

Example 2 Assignments move into a nested loop

The extended assignments have been moved into a **nested loop**. This loop iterates through the size of the problem within another loop that also iterates through the size of the problem. This program was left running overnight. By morning it had processed only the first data set, 1,000. The program was then terminated and run again with a smaller problem size of 10,000,000.

```
#File: AD13_TC_Exp2_Nested_Times.py
import time
N = 1000 # or 10000000  Problem size
print("%12s%16s" % ("Problem Size N", "Seconds"))
for j in range(N):
    start = time.time()
    # The start of the algorithm
    work = 1
    # Nested loops
    for j in range(N):
        for k in range(N):
            work += 1
            work -= 1
    # The end of the algorithm
    elapsed = time.time() - start
    print("%12d%16.12f" % (N, elapsed))
    N *= 2
```

Problem Size N	Seconds
1000	0.171882390976
2000	0.841088533401
4000	2.933720827103
8000	11.471257686615
16000	45.683874607086

Figure 2 The output of Example 2

- Note that when the problem size doubles, the number of seconds of running time more or less quadruples. At this rate, it would take 100+ days to process the largest number in the previous data set!

1.1.1 This method *permits accurate predictions* of the **running times** of many algorithms. However, there are two major **problems** with this technique:

- Different hardware platforms have different processing speeds, so the running times of an algorithm differ from machine to machine. Also, the running time of a program varies with the type of operating system that lies between it and the hardware. Finally, different programming languages and compilers produce code whose performance varies. For example, an algorithm coded in C usually runs slightly faster than the same algorithm in Python byte code. Thus, predictions of performance generated from the results of timing on one hardware or software platform generally cannot be used to predict potential performance on other platforms.

- It is impractical to determine the running time for some algorithms with very large data sets. For some algorithms, it doesn't matter how fast the compiled code or the hardware processor is. They are impractical to run with very large data sets on any computer.

- Timing algorithms may in some cases be a helpful form of testing that is hardware and software dependent,

Requirement: an *estimate of the efficiency* of an algorithm that is *independent* of a particular hardware or software platform.

1.2 Counting Instructions

- Technique which used to *estimate the efficiency* of an algorithm is to *count the instructions* executed with different problem sizes.
- These counts provide a *good predictor of the amount of abstract work* an algorithm performs, no matter what platform the algorithm runs on.
- Count instructions technique: *counting the instructions in the high-level code* in which the algorithm is written, not instructions in the executable machine language program.
- When analyzing an algorithm in this way, must distinguish between two classes of instructions:
  - Instructions that execute the same number of times regardless of the problem size
    - This class instructions can be ignored, because they do not figure significantly in this kind of analysis.
  - Instructions whose execution count varies with the problem size
    - The instructions in this class normally are found in *loops or recursive functions.*
    - In the case of loops, zero in on instructions performed in any nested loops or, more simply, just the number of iterations that a nested loop performs.

Example 3 Tracking and displaying the number of iterations the loop executes with the different data sets:

Pseudocode:

```
//N : Problem size
FOR j ← 1 TO N
    number ← 0
    work ← 1
     FOR k ← 1 TO N
          number ← number + 1
          work ← work + 1
          work ← work - 1
     ENDFOR
ENDFOR
```

Problem Size N	Iterations
1000	1000
2000	2000
4000	4000
8000	8000
16000	16000
32000	32000
64000	64000
128000	128000
256000	256000
512000	512000

Figure 3 The output of Example 3 that counts iterations.

Python:

```
#File: AD13_TC_Exp3_Iterations.py
N = 1000 # N : Problem size
print("%12s%16s" % ("Problem Size N", "Iterations"))
for j in range(N):
    number = 0
    # The start of the algorithm
    work = 1
    for k in range(N):
        number += 1
        work += 1
        work -= 1
    # The end of the algorithm
    print("%12d%12d" % (N, number))
    N *= 2
```

Example 4 Tracking and displaying the number of iterations the **inner** loop executes with the different data sets:

Pseudocode:

```
//N : Problem size
N ← 1000
FOR count ← 1 TO 5
      number ← 0
      work ← 1
      FOR J ← 1 TO N
            FOR k ← 1 TO N
                  number ← number + 1
                  work ← work + 1
                  work ← work - 1
            ENDFOR
      ENDFOR
ENDFOR
```

Python:

```
"""
File: AD13_TC_Exp4_Nested_Iterations.py
Prints the number of iterations for problem sizes
that double, using a nested loop.
"""
N = 1000 # Problem size
print("%12s%15s" % ("Problem Size N", "Iterations"))
for count in range(5):
    number = 0
    # The start of the algorithm
    work = 1
    for j in range(N):
          for k in range(N):
                number += 1
                work += 1
                work -= 1
    # The end of the algorithm
    print("%12d%15d" % (N, number))
    N *= 2
```

Problem Size N	Iterations
1000	1000000
2000	4000000
4000	16000000
8000	64000000
16000	256000000

Figure 4 The output of Example 4 that counts iterations with nested loops

From the results, the number of iterations is the square of the problem size (Figure 4).

Example 5 Tracking the number of calls of a recursive Fibonacci function for several problem sizes. Note that the function using a `global counter` variable. Each time the function is called at the top level, the counter is incremented.

```
"""
File: AD13_TC_Exp5_countFibonacci.py
Prints the number of calls of a recursive Fibonacci function with
problem sizes that double.
"""
counter = 0      #Global variable

def fib(n):
    """
    Count the number of calls of the
    Fibonacci function.
    """
    global counter
    counter += 1
    if n < 3:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

N = 2 # Problem Size
print("%12s%15s" % ("Problem Size N", "Calls"))
for count in range(5):
    counter = 0
    # The start of the algorithm
    fib(N)
    # The end of the algorithm
    print("%12d%15s" % (N, counter))
    N *= 2
```

Problem Size N	Calls
2	1
4	5
8	41
16	1973
32	4356617

Figure 5 The output of Example 5 that runs the Fibonacci function.

As the problem size doubles, the instruction count (number of recursive calls) grows slowly at first and then quite rapidly. At first, the instruction count is less than the square of the problem size. However, when the problem size reaches 16, the instruction count of 1973 is significantly larger than 256, or $\mathbf{16^2}$.

The problem with tracking counts in this way is that, with some algorithms, the computer still cannot run fast enough to show the counts for very large problem sizes.

- Counting instructions is the right idea, but need to turn to logic and mathematical reasoning for a complete method of analysis. The only tools need for this type of analysis are paper and pencil.

1.3 Measuring the Memory Used by an Algorithm [not in syllabus]

A complete analysis of the resources used by an algorithm includes the amount of memory required. Once again, focus on rates of potential growth. Some algorithms require the same amount of memory to solve any problem. Other algorithms require more memory as the problem size gets larger.

2 Complexity Analysis

Developing a method of determining the efficiency of algorithms that allows to rate them independently of platform-dependent timings or impractical instruction counts.

This method, called *complexity analysis*, entails reading the algorithm and using pencil and paper to work out some simple algebra.

2.1 Orders of Complexity

Consider the two counting loops discussed in **Example 3** and **4**. The first loop executes n times for a problem of size *n*. The second loop contains a nested loop that iterates n^2 times. The amount of work done by these two algorithms is similar for small values of *n* but is very different for large values of *n*. Figure 5 and Table 1 illustrate this divergence. Note that "work" in this case refers to the number of iterations of the most deeply nested loop.

Example 3
Pseudocode:

```
FOR k ← 1 TO N
    number ← number + 1
ENDFOR
```

Python:

```
for k in range(N):
    number += 1
```

Example 4
Pseudocode:

```
For J ← 1 TO N
    FOR k ← 1 TO N
        number ← number + 1
    ENDFOR
ENDFOR
```

Python:

```
for j in range(N):
    for k in range(N):
        number += 1
```

Figure 5
A graph of the amount of work done in the tester programs

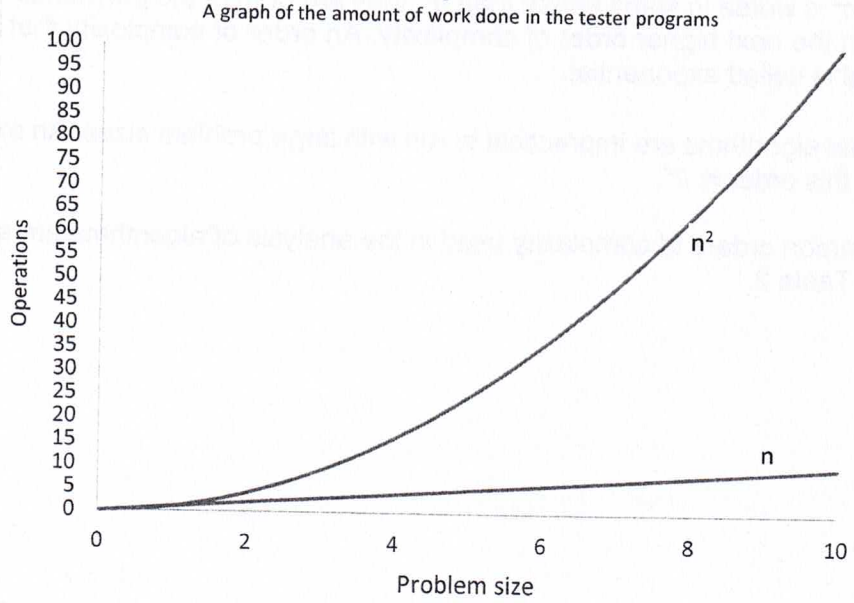


Table 1 The Amounts of Work in the algorithms

Problem Size	Work of the Example 3	Work of the Example 4
2	2	4 (2^2)
10	10	100 (10^2)
1,000	1,000	1,000,000 (1000^2)

- The performances of these algorithms differ by an *order of complexity*.
    - The performance of the first algorithm is *linear* in that its work grows in direct proportion to the size of the problem (problem *size* of 10, work of 10; 20 and 20; and so on).
        - The basic rule is easy: The complexities of code blocks executed one after the other are just added.
    - The behavior of the second algorithm is *quadratic* in that its work grows as a function of the square of the problem size (problem size of 10, work of 100).
        - The complexities of nested loops are multiplied.
        - The reasoning is simple: For each round of the outer loop, the inner one is executed in full. In this case, that means "linear times linear," which is quadratic.
    - Algorithms with linear behavior do *less* work than algorithms with quadratic behavior for most problem sizes n. In fact, as the problem size gets larger, the performance of an algorithm with the higher order of complexity becomes *worse* more quickly.

- Several other orders of complexity are commonly used in the analysis of algorithms. An algorithm has *constant* performance if it requires the same number of operations for any problem size. Array (Lists in Python) indexing is a good example of a constant-time algorithm. This is clearly the best kind of performance to have.

- Another order of complexity that is better than linear but worse than constant is called *logarithmic*. The amount of work of a logarithmic algorithm is proportional to the $\log_2$ of the problem size. Thus, when the problem doubles in size, the amount of work only increases by 1 (that is, just add 1).

- The work of a *polynomial time algorithm* grows at a rate of n^k, where k is a constant greater than 1. Examples are n^2, n^3, and n^{10}.

  Although n^3 is worse in some sense than n^2, they are both of the polynomial order and are better than the next higher order of complexity. An order of complexity that is worse than polynomial is called exponential.

- Exponential algorithms are impractical to run with large problem sizes. An example rate of growth of this order is 2^n.

The most common orders of complexity used in the analysis of algorithms are summarized in Figure 6 and Table 2.

Figure 6 A graph of some sample orders of complexity.

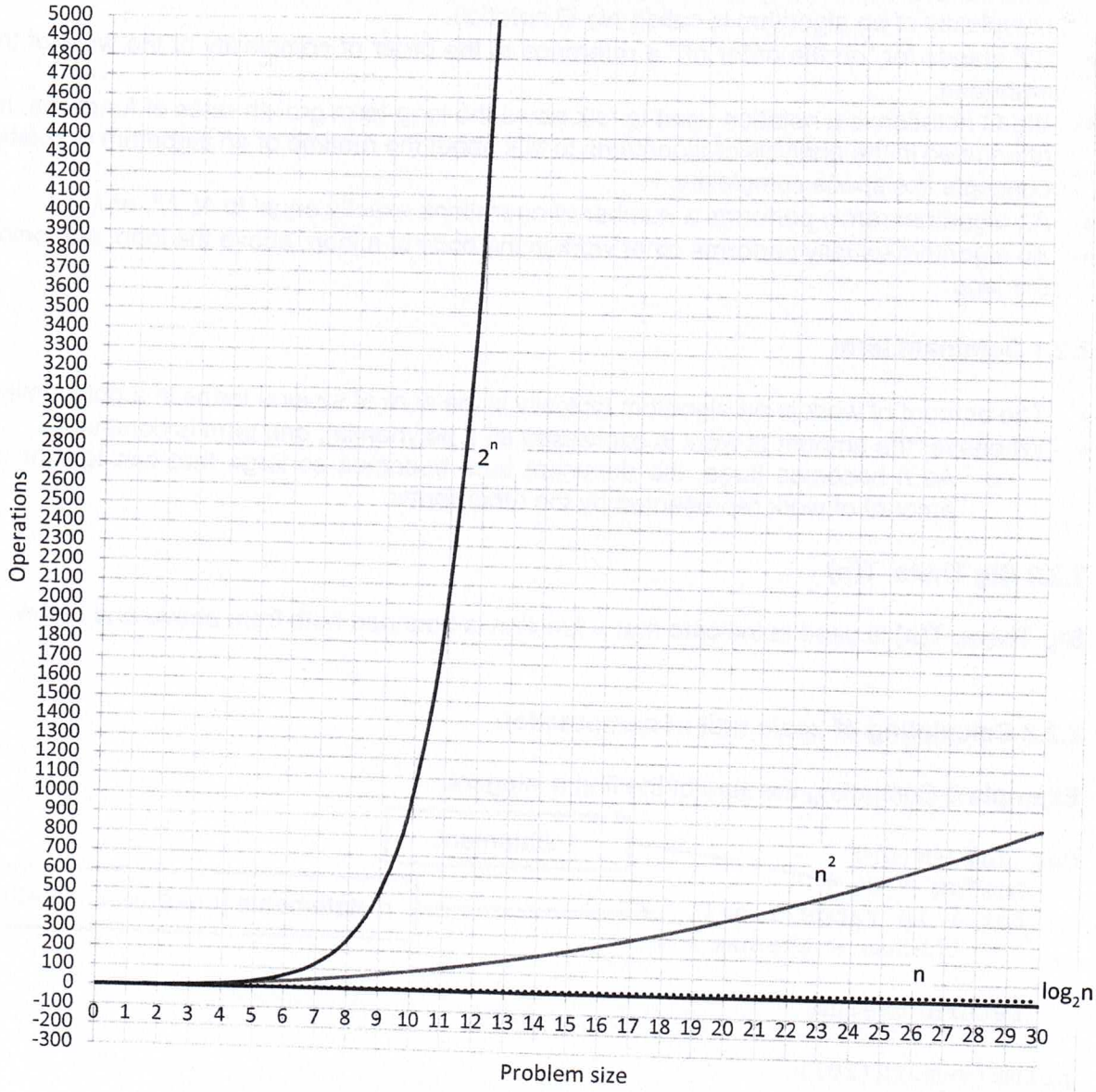


Table 2 Some Sample Orders of Complexity

n	Logarithmic ($\log_2 n$)	Linear (n)	Quadratic (n^2)	Exponential (2^n)
100	7	100	10,000	Off the chart
1,000	10	1,000	1,000,000	Off the chart
1,000,000	20	1,000,000	1,000,000,000,000	Really off the chart

2.2 Big-O Notation

- One notation that computer scientists use to express the efficiency or computational complexity of an algorithm is called *big-O notation*.
- "*O*" stands for "on the order of," a reference to the order of complexity of the work of the algorithm.
- Big-O notation is a notation used to talk about the long-term growth rates of functions. It's often used in the analysis of algorithms to talk about the runtime of an algorithm or related concepts like space complexity.
- An algorithm rarely performs a number of operations exactly equal to N, N^2, or k^N.
- An algorithm usually performs other work in the body of a loop, above the loop, and below the loop.

2.2.1 Dominant term

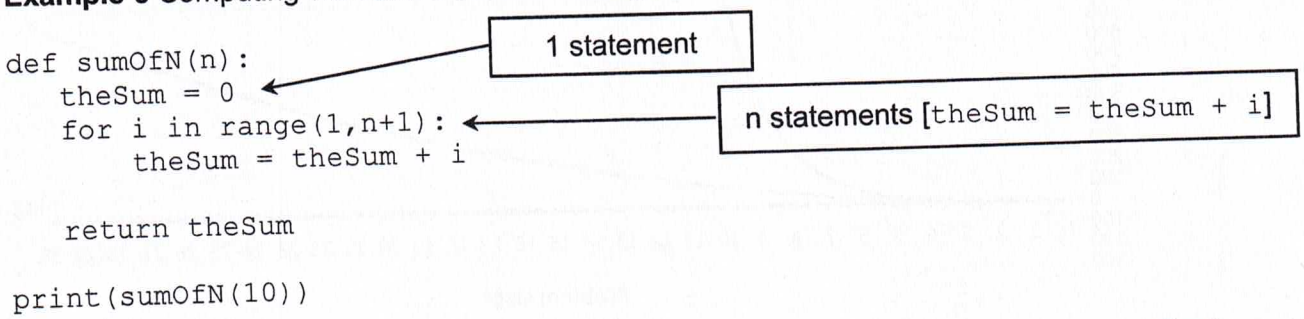
- The amount of work in an algorithm typically is the sum of several terms in a polynomial.
- Whenever the amount of work is expressed as a polynomial, one term is dominant.
    - As *n* becomes large, the dominant term becomes so large that can **ignore** the amount of work represented by the other terms.

2.2.2 Big Theta, T(n)

Big Theta, T(n) is used to indicate that a function is bounded both from above and below.

2.2.3 Calculating of basic unit of computation

Example 6 Computing the sum of the first *n* integers.




```
def sumOfN(n):
    theSum = 0
    for i in range(1,n+1):
        theSum = theSum + i

    return theSum

print(sumOfN(10))
```

When trying to characterize an algorithm's efficiency in terms of execution time, independent of any particular program or computer, it is important to quantify the number of operations or steps that the algorithm will require. If each of these steps is considered to be a basic unit of computation, then the execution time for an algorithm can be expressed as the number of steps required to solve the problem. Deciding on an appropriate basic unit of computation can be a complicated problem and will depend on how the algorithm is implemented.

A good basic unit of computation for comparing the summation algorithms shown in Example 6 might be to count the number of assignment statements performed to compute the sum. In the function **`sumOfN`**, the number of assignment statements is 1 (**`theSum = 0`**) plus the value of *n* (the number of times performing **`theSum = theSum + i`**). Denote this by a function, call it T, where T(n) = 1 + n. The parameter *n* is often referred to as the "size of the problem," and we can read this as "T(n) is the time it takes to solve a problem of size n, namely 1+n steps."

In the summation functions given above, it makes sense to use the number of terms in the summation to denote the size of the problem. The sum of the first 100,000 integers is a bigger instance of the summation problem than the sum of the first 1,000. Because of this, it might seem reasonable that the time required to solve the larger case would be greater than for the smaller case. Goal then is to show how the algorithm's execution time changes with respect to the size of the problem.

Computer scientists prefer to take this analysis technique one step further. It turns out that the exact number of operations is not as important as determining the most dominant part of the T(n) function. In other words, as the problem gets larger, some portion of the T(n) function tends to overpower the rest. This **dominant term** is what, in the end, is used for comparison. The **order of magnitude** function describes the part of T(n) that increases the fastest as the value of *n* increases. Order of magnitude is often called **Big-O** notation (for "order") and written as O(f(n)). It provides a useful approximation to the actual number of steps in the computation. The function ***f(n)*** provides a ***simple representation of the dominant part*** of the original T(n).

In the above example, T(n) = 1 + n. As *n* gets large, the constant 1 will become less and less significant to the final result. If we are looking for an approximation for T(n), then we can drop the 1 and simply say that the running time is O(n). It is important to note that the 1 is certainly significant for T(n). However, as *n* gets large, our approximation will be just as accurate without it.

2.2.3.1 Asymptotic Analysis

In the polynomial $\frac{1}{2} n^2 - \frac{1}{2} n$

- focus on the quadratic term, $\frac{1}{2} n^2$,
- in effect dropping the linear term, $\frac{1}{2} n$, from consideration. [$\frac{1}{2} n^2 - \cancel{\frac{1}{2} n}$]
- also drop the coefficient $\frac{1}{2}$ because the ratio between $\frac{1}{2} n^2$ and n^2 does not change as n grows. [$\cancel{\frac{1}{2}} n^2$]
    - For example, if double the problem size, the run times of algorithms that are $\frac{1}{2} n^2$ and n^2 increase by a factor of 4.

This type of analysis is sometimes called *asymptotic analysis* because the value of a polynomial asymptotically approaches or approximates the value of its largest term as *n* becomes very large.

Example 7

a) Explain $T(n) = 5n^2 + 27n + 1005$ is of complexity of $O(n^2)$.

When *n* is small, say 1 or 2, the constant 1005 seems to be the dominant part of the function.

However, as *n* gets larger, the n^2 term becomes the most important.

In fact, when *n* is really large, the other two terms become insignificant in the role that they play in determining the final result.

Again, to approximate T(n) as *n* gets large, we can ignore the other terms and focus on $5n^2$.

In addition, the coefficient 5 becomes insignificant as *n* gets large. We would say then that the function T(n) has an order of magnitude $f(n) = n^2$, or simply that it is $O(n^2)$.

b) Prove that $T(n) = 3n^3 + 2n + 7 \in O(n^3)$

- If $n \geq 1$, then $T(n) = 3n^3 + 2n + 7 \leq 3n^3 + 2n^3 + 7n^3 = 12n^3$. Hence $T(n) \in O(n^3)$.
- On the other hand, $T(n) = 3n^3 + 2n + 7 > n^3$ for all positive n.

  Therefore $T(n) \in \Omega(n^3)$.
  - **Big Omega** is used to give a **lower bound** for the growth of a function.
- And consequently $T(n) \in O(n^3)$.

2.2.4 Common Big-O functions

2.2.4.1 The order of complexity of a Linear-time algorithm is *O(n)*.

2.2.4.1.1 *O(1)* describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

- Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

For example SWAP() procedure has O(1) time complexity.

```
PROCEDURE SWAP(X ByVal, Y ByVal: INTEGER)
    DECLARE TEMP: INTEGER
    TEMP = X
    X = Y
    Y = TEMP
ENDPROCEDURE
```

- A loop or recursion that runs a constant number of times is also considered as O(1).

  For example the following loop is O(1).

```
// Here C is a constant
FOR I = 1 TO C STEP 1
    // some O(1) expressions
ENDFOR
```

Example 8 Check whether first element in an array is Null

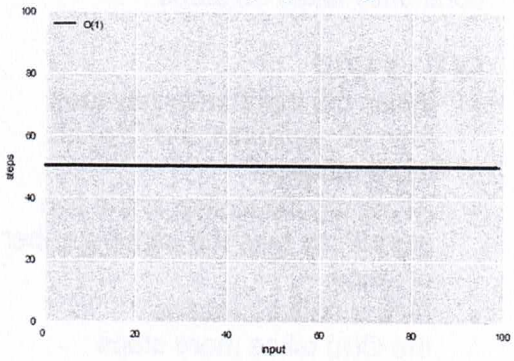
Access it directly by using the index operator array[0]

```
FUNCTION IsFirstElementNull(ARR) RETURNS BOOLEAN
    RETURN ARR [0] = NULL
ENDFUNCTION

DECLARE CODE : ARRAY[1:100] OF STRING
DECLARE ANS : BOOLEAN

ANS ← IsFirstElementNull(CODE)
```

Example 9 Determine if the **i-th element** of an array is an odd number.

To access the 1st or 2nd or millionth item it doesn't matter. Access it directly by using the index operator array[i]

```
# AD13_TC_Exp9_isNthElementOdd
array = [0,3,4,7,8,5,5,2]

def isNthElementOdd(array, n):
    return bool(array[n] % 2)

isOdd = isNthElementOdd(array, 3)
print(isOdd)
```



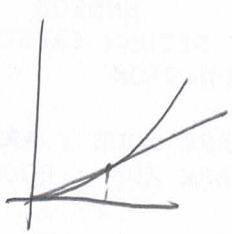
O(1)

Note:
Assume that the *O(1)* algorithm constantly takes 50 steps [Example 9].
To represent the number of steps (y-axis) vs the number of input elements (x-axis), *O*(1) is a perfect horizontal line, since the number of steps in the algorithm remains constant no matter how much data there is. This is why it is called *constant time*.

2.2.4.1.2 *O*(n) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.

- An algorithm that is O(n) will take as many steps as there are elements of data.
- Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example following functions have O(n) time complexity.

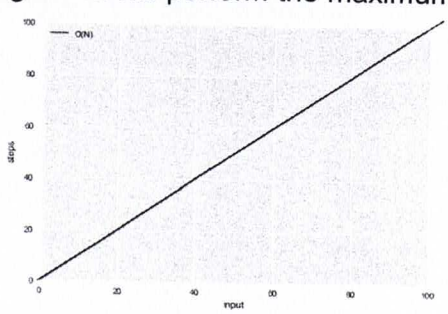
```
// Here C is a positive integer constant
FOR I = 1 TO n STEP C
    // some O(1) expressions
ENDFOR
FOR I = n TO -1 STEP -C
    // some O(1) expressions
ENDFOR
```



Example 10 Traverse an array and print each element.

When an array increases in size by one element, an O(n) algorithm will increase by one step Need to access all the elements one by one, so the calculation time increases at the same pace as the input.
Here demonstrates how Big O favours the worst-case performance scenario; a matching string could be found during any iteration of the for loop and the function would return early, but Big *O* notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.

```
# AD13_TC_Exp10_displayAllItems.py
array = [0,3,4,7,8,5,5,2]
def displayAllItems (array):
    for i in array:
        print (i, end = ' ')
displayAllItems (array)
```



O(N)

Note:
O(n) is a perfect diagonal line, as for every additional piece of data, the algorithm takes one additional step. This is why it is also referred to as *linear time*.

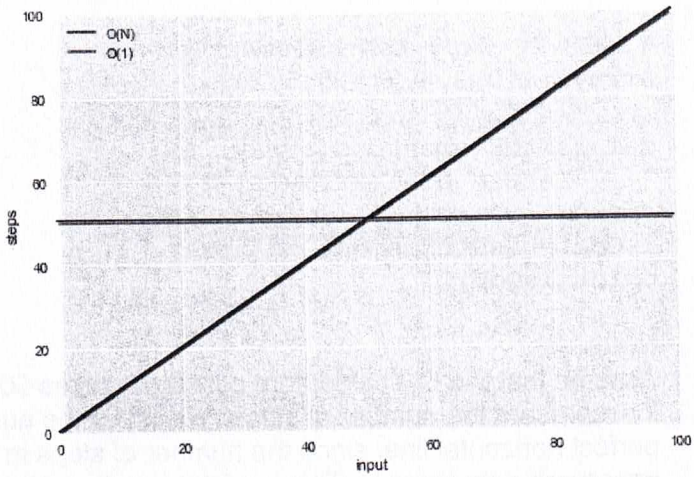
2.2.4.1.3 Comparing O(1) and O(n) [Example 9 and Example 10]

Plot the O(1) and O(n) algorithms in the same graph and let's assume that the O(1) algorithm constantly takes 50 steps.

O(1) vs O(n)

- When the input array has less than 50 elements, the O(n) is more efficient.
- At exactly 50 elements the two algorithms take the same number of steps.
- As the data increases the O(n) takes more steps.

Since the Big-O notation looks at how the algorithm performs as the data grows to **infinity**, this is why O(n) is considered to be less efficient than O(1).



O(1) vs O(n)

Example 11 Traverse an array and compare element.

```
FUNCTION ContainsValue(ARR, Value) RETURNS BOOLEAN
      DECLARE I : INTEGER
          FOR I ← 1 TO LENGTH(ARR)
              IF (ARR[I] = Value) THEN
                    RETURN TRUE
              ENDIF
          ENDFOR
      RETURN FALSE
ENDFUNCTION

DECLARE CODE : ARRAY[1:100] OF STRING
DECLARE ANS : BOOLEAN

ANS ← IsFirstElementNull(CODE, "A9800")
```

2.2.4.2 The order of complexity of a Quadratic is $O(n \times n)$, $O(n^2)$

$O(n^2)$ represents the complexity of an algorithm, whose performance is proportional to the square of the size of the input elements. It is generally quite slow: If the input array has 1 element it will do 1 operation, if it has 10 elements it will do 100 operations, and so on.

Time complexity of nested loops is equal to the number of times the innermost statement is executed.

```
FOR I = 1 TO n STEP C
  FOR J = 1 TO n STEP C
          // some O(1) expressions
  ENDFOR
ENDFOR

FOR I = n TO 1 STEP -C
  FOR J = I + 1 TO n STEP C
          // some O(1) expressions
      ENDFOR
ENDFOR
```

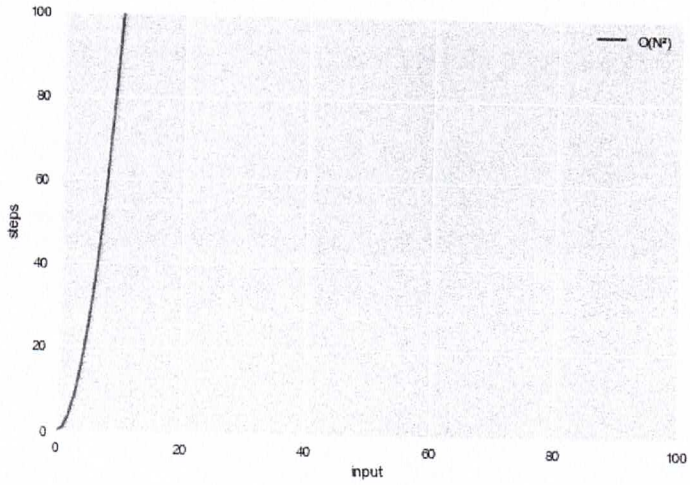
For example Insertion Sort have $O(n^2)$ time complexity.

Example 12 Find **duplicates** in an array

```
#AD13_TC_Exp12_containDuplicate
array = [0,3,4,7,8,5,5,2]
def containDuplicate (array):
    for i, outerEle in enumerate(array):
        for j, innerEle in enumerate(array):
            if (i == j):
                continue
            if (outerEle == innerEle):
                return True
    return False

duplicate = containDuplicate (array)
print (duplicate)
```

Note:
Adding more nested iterations through the input will increase the algorithm's complexity: e.g. if the number of iterations is 3 then its complexity will be $O(n^3)$ and so forth. Usually, we want to stay away from *polynomial* running times (quadratic, cubic, n^x, etc).

Refer to **Example 4**



$O(n^2)$

2.2.4.3 The order of complexity of a Logarithmic is $O(log_2N)$

$O(log_2N)$ describes an algorithm that its number of operations increases by one each time the data is doubled.

Exponents and base 2 logarithm

$8 = 2 * 2 * 2 = 2^3$
$log_2 8 = log_2 2^3$
$log_2 8 = 3$

Keep dividing 8 by 2 until we end up to 1, the number of 2s = 8, ((8 / **2**) / **2**) / **2** = 1. The answer is 3 again

Time Complexity of a loop is considered as $O(Log_2N)$ if the loop variables is divided / multiplied by a constant amount.

For O(Log$_2$N)

```
FOR I = 1 TO N STEP  I * 2
          // some O(1) expressions
ENDFOR

FOR I = 1 TO N STEP I / 2
          // some O(1) expressions
ENDFOR
```

For O(Log$_c$N)

```
FOR I = 1 TO N STEP  I * C
          // some O(1) expressions
ENDFOR

FOR I = 1 TO N STEP I / C
          // some O(1) expressions
ENDFOR
```

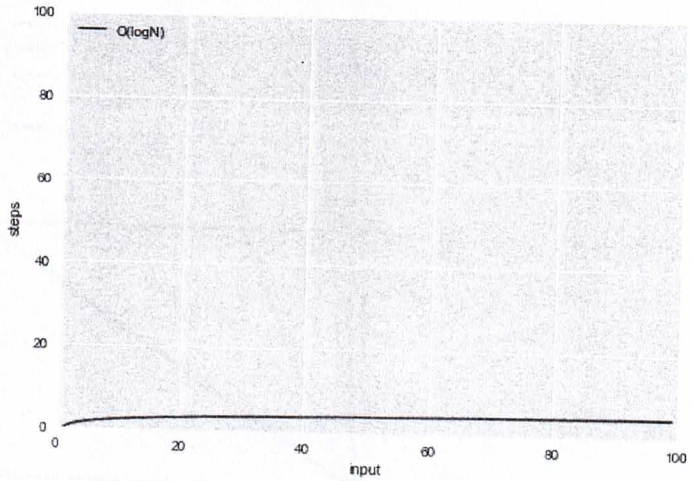
Example 13 Dictionary lookup (**binary search - Searching**)

01 Open the dictionary in the middle and check the first word.
02 If our word is alphabetically more significant, look in the right half, else look in the left half.
03 Divide the remainder in half again, and repeat steps 2 and 3 until we find our word.

```
#AD13_TC_Exp13_binary_search.py
Array = ["Apple","Bear","Carrot","Dumpling","Elephant","Fox","Goat"]
def binarySearch(array, element):
    first = 0
    last = len(array) - 1
    index = -1

    while (first <= last) and (index == -1):
        mid = (first + last) // 2
        if array[mid] == element:
            return True
        elif element < array[mid]:
            last = mid - 1
        else:
            first = mid + 1
    return False

found = binarySearch(Array, 'Dumpling')

print('Result:', found)
```

Note: Only pick one possibility per iteration, and pool of possible matches gets divided by two in each iteration. This makes the time complexity of binary search $O(\log_2 N)$.

The number of steps barely increase, as the input grows (i.e. it takes just one additional step each time the data **doubles**)



$O(\log_2 N)$

2.2.4.4 O(NlogN) — Log-linear

An algorithm of this complexity class is doing log(N) work N times and therefore its performance is slightly worse than O(N). Many practical algorithms belong in this category (from sorting, to pathfinding, to compression), so we are mentioning it for completeness.

Example 14 Merge Sort — it is a 'Divide and Conquer' algorithm: it divides the input array in two halves, calls itself for each one and then merges the two sorted halves.

- **Scalability**: Average.

2.2.4.5 The order of complexity of an Exponential is $O(2^n)$

Exponential growth means that the algorithm takes twice as long for every new element added.

Example 15 Find all subsets in a dataset.

- Scalability: Poor.

2.2.4.6 O(N!) — Factorial [Not in syllabus]

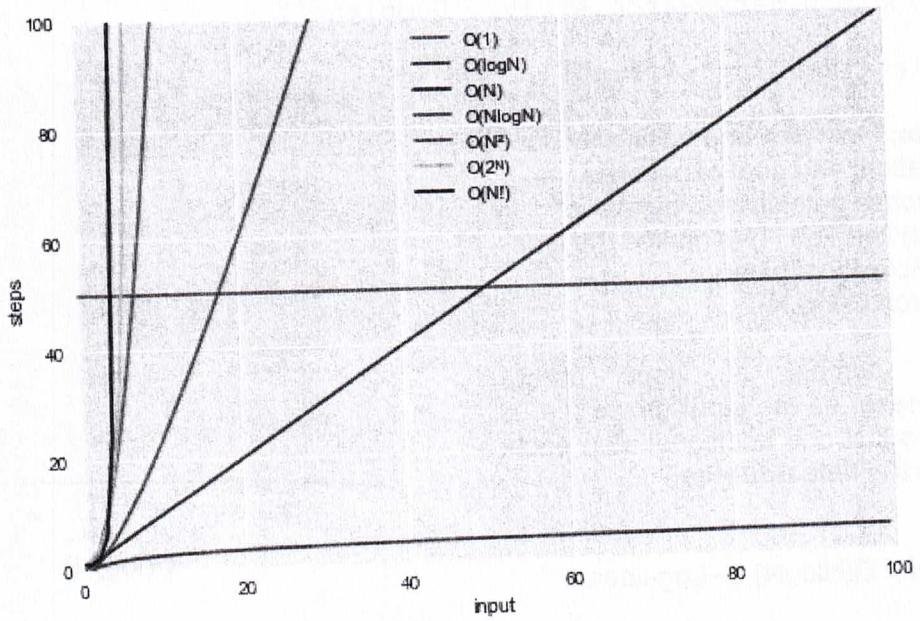
This class of algorithms has a run time proportional to the factorial of the input size: N! = N * (N-1) * (N-2) * (N-3) * . . . * 1.

Example 16 Find all different permutations in a dataset.

- **Scalability**: Very Poor.

2.2.4.7 Growth Hierarchy

The Big-O notation offers a consistent mechanism to compare any two algorithms and hence help make code faster and more scalable. Putting all of the complexities in a single graph, compare in terms of performance:



Big-O Complexity Classes

2.3 The Role of the Constant of Proportionality

The *constant of proportionality* involves the terms and coefficients that are usually ignored during big-*O* analysis. However, when these items are large, they may affect the algorithm, particularly for small and medium-sized data sets.

Note: No one can ignore the difference between *n* and *n/2*, when *n* is $1,000,000.

- In algorithms of Example 3 and 4, the instructions that execute within a loop are part of the constant of proportionality, as are the instructions that initialize the variables before the loops are entered.

Example 3
Pseudocode:

```
N : Problem Size
work ← 1
FOR k ← 1 TO N
      number ← number + 1
      work ← work + 1
      work ← work - 1
ENDFOR
```

Example 4
Pseudocode:

```
N : Problem Size
For J ← 1 TO N
      work ← 1
      FOR k ← 1 TO N
            number ← number + 1
            work ← work + 1
            work ← work - 1
      ENDFOR
ENDFOR
```

When analyzing an algorithm, must be careful to determine that any instructions do not hide a loop that depends on a variable problem size. If that is the case, then the analysis must move down into the nested loop.

Example 17 Determine the constant of proportionality for Example 3. Here is the code:

```
work = 1                        | 1
for x in range(n):              | n
   work += 1                    | n
   work -= 1                    | n
```

Note that, aside from the loop itself, there are three lines of code, each of them assignment statements. Each of these three statements runs in constant time. Also assume that on each iteration, the overhead of managing the loop, which is hidden in the loop header, runs an instruction that requires constant time. Thus, the amount of abstract work performed by this algorithm is $3n + 1$. Although this number is greater than just n, the running times for the two amounts of work, n and $3n + 1$, increase at the same rate, $O(n)$.

Example 18 Find T(n), O(n) of the following Python program.

The number of assignment operations is the sum of four terms.
The first term is the constant 3, representing the three assignment statements at the start of the fragment.
The second term is $3n^2$, since there are three statements that are performed n^2 times due to the nested iteration.
The third term is 2n, two statements iterated *n* times.
Finally, the fourth term is the constant 1, representing the final assignment statement.

`a=5`	1		
`b=6`	1		
`c=10`	1		
`for i in range(n):`	n		
`  for j in range(n):`		n	
`    x = i * i`			1
`    y = j * j`			1
`    z = i * j`			1
`for k in range(n):`	n		
`  w = a*k + 45`		1	
`  v = b*b`		1	
`d = 33`	1		

This gives us

$$T(n)= 3 + 3n^2 + 2n + 1 =3n^2 + 2n + 4.$$

By looking at the exponents, we can easily see that the n^2 term will be dominant and therefore this fragment of code is $O(n^2)$. Note that all of the other terms as well as the coefficient on the dominant term can be ignored as *n* grows larger.

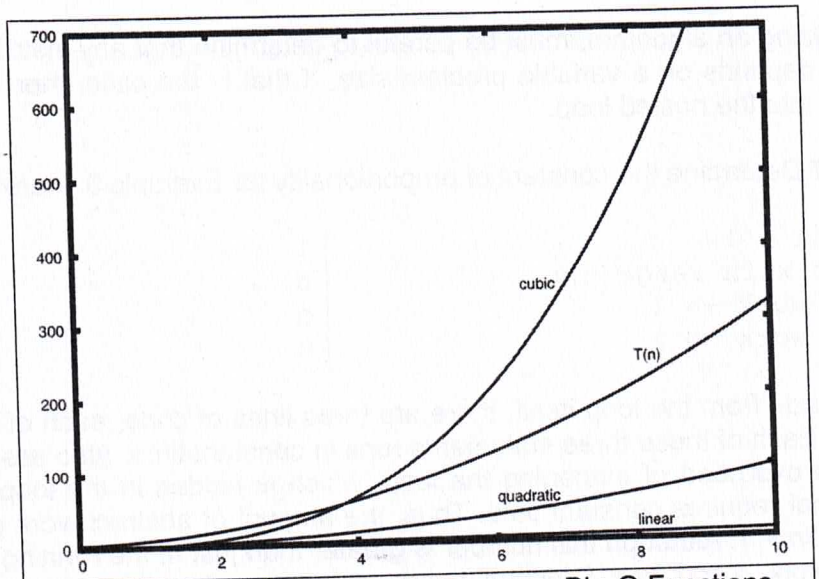


Figure 7: Comparing T(n) with Common Big-O Functions

Figure 7 shows a few of the common Big-O functions as they compare with the T(n) function discussed above. Note that T(n) is initially larger than the cubic function. However, as n grows, the cubic function quickly overtakes T(n). It is easy to see that T(n) then follows the quadratic function as n continues to grow.

3 Best-Case, Worst-Case, and Average-Case Performance

Refer to Example 11

Pseudocode:

```
FUNCTION ContainsValue(ARR, Value) RETURNS BOOLEAN
    DECLARE I ; INTEGER
        FOR I ← 1 TO LENGTH(ARR)
            IF (ARR[I] = Value)
                RETURN TRUE
                EXIT FOR
            ENDIF
        ENDFOR
    RETURN FALSE
ENDFUNCTION

DECLARE CODE : ARRAY[1:100] OF STRING
DECLARE ANS : BOOLEAN

found ← ContainsValue(CODE, "8910")
```

Python:

```
#AD13_TC_Exp11_ContainsValue
array = ['8566', '8512', '1123', '8910', '1212', '9695']
def ContainsValue (array, code):
    for i in range(len(array)-1):
        if (code == array[i]):
            return True
    return False
toFindCode = '8911'
found = ContainsValue (array, toFindCode)
print (toFindCode , 'found is ', found)
```

Best-Case: 1
Worst-Case: LENGTH(ARR)
Average-Case Performance: (1 + 2 + 3 + ... + LENGTH(ARR)) / LENGTH(ARR)

- **Best vs Worst Scenario**

Given this 6-element array: `['8566', '8512', '1123', '8910', '1212', '9695']` if we were searching for `toFindCode = '9695'` the algorithm would need 6 steps to find it, but if we were searching for `toFindCode = '8566'` it would only take 1 step. So best case scenario is when we look for a value that is in the *first cell* and worst case scenario is when the value is at the *last cell*, or not there at all.

The Big-O notation takes a **pessimistic** approach to performance and refers to the worst case scenario. It is really important when describing the complexities of any algorithm, and also when try to compute the complexity of the algorithms: *Always think of the worst case scenarios.*

3.1 How to calculate time complexity when there are many if, else statements inside loops?

Worst case time complexity is the most useful among best, average and worst. Therefore we need to consider worst case. We evaluate the situation when values in if-else conditions cause maximum number of statements to be executed.

For example consider the linear search function where we consider the case when element is present at the end or not present at all.

When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if else and other complex control statements.

3.2 How to calculate time complexity of recursive functions?

Time complexity of a recursive function can be written as a mathematical recurrence relation. To calculate time complexity, we must know how to solve recurrences.

4 Summary

- Algorithm speed is not measured in seconds but in terms of growth
- The Big-O Notation show how an algorithm scales against changes in the input dataset size
- O stands for *Order Of* — as such the Big-O Notation is approximate
- Algorithm running times grow at different rates:

$$O(1) < O(\log N) < O(N) < O(N \log N) < O(N^2) < O(2^N) < O(N!)$$

Tutorial 13[Algorithm Complexity] Q1, 2, 4

Tutorial 13 [Algorithm Complexity]

1. Assume that each of the following expressions indicates the number of operations performed by an algorithm for a problem size of n. Point out the dominant term of each algorithm and use big-O notation to classify it.

   a. $2^n - 4n^2 + 5n$
   b. $3n^2 + 6$
   c. $n^3 + n^2 - n$

2. For problem size n, algorithms A and B perform n^2 and $\frac{1}{2}n^2 + \frac{1}{2}n$ instructions, respectively. Which algorithm does more work? Are there particular problem sizes for which one algorithm performs significantly better than the other? Are there particular problem sizes for which both algorithms perform approximately the same amount of work?

3. At what point does an n^4 algorithm begin to perform better than a 2^n algorithm?

4. Given the following code fragments, what is its Big-O running time of each code fragment?

   A. O(n) B. O(n^2) C. O(log n) D. O(n^3)

   (a)
```
test = 0
for i in range(n):
        for j in range(n):
            test = test + i * j
```

   4

   (b)
```
test = 0
for i in range(n):
    test = test + 1

for j in range(n):
    test = test - 1
```

   (c)
```
i = n
while i > 0:
    k = 2 + 2
    i = i // 2
```

Annex:

Big-O Efficiency of Python List Operators

Operation	Big-O Efficiency
index []	O(1)
index assignment	O(1)
append	O(1)
pop()	O(1)
pop(i)	O(n)
insert(i,item)	O(n)
del operator	O(n)
iteration	O(n)
contains (in)	O(n)
get slice [x:y]	O(k)
del slice	O(n)
set slice	O(n+k)
reverse	O(n)
concatenate	O(k)
sort	O(n log n)
multiply	O(nk)