Algorithms Analysis and Design

Chapter 2

Analysis of Algorithms – Part1

Analysis of Algorithms

- After you have read and studied this chapter, you should be able to
  - Know what is analysis of algorithms (complexity analysis)
  - Know the type of analysis (best , worst, and average).
  - How to analyse the running time of iterative and recursive algorithms.
  - Know the definition and uses of Asymptotic notations (big-O notation, Omega ω, theta ϑ).
  - How to solve recurrence equations.

The Conversation

- *Boss: Your program is too slow! Rewrite it!*
- *You: But why? All we need to do is to buy faster computer!*

Is this really the solution?

Analysis of Algorithms

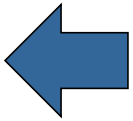
- **Issues**
  - Correctness
  - **Time efficiency**
  - Space efficiency
  - Optimality

Different criteria are used to analyze an algorithm. The main two criteria are:

- **Time** (How much time it is taking in the form of a function)
- **Space** (How much memory space it will consume)

- **Approaches**
  - Theoretical Analysis
  - Empirical Analysis

In this course, we are concerned primarily with the time analysis.



ما بحتاج أعبر السيدو كود لـ
كود برمجي في تاني الطريقة

بحتاج steps لـ time complicity وض خلال الـ steps بطلع الـ

Analysis of Algorithms

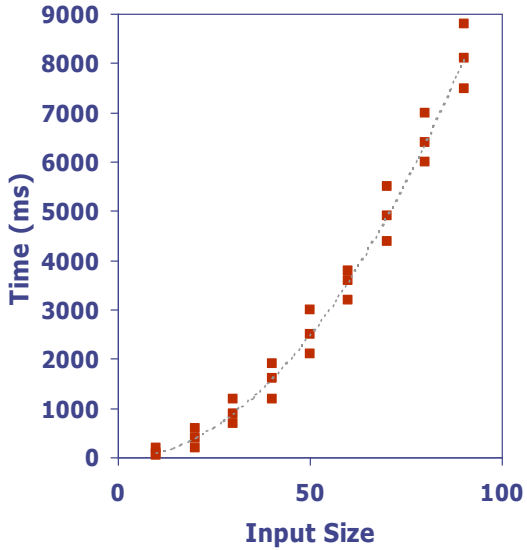
- An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.
- **What is the goal of analysis of algorithms?**
  - To compare algorithms mainly in terms of running time but also in terms of other factors (e.g., memory requirements, programmer's effort etc.)
- **What do we mean by running time analysis?**
  - **Determine how running time increases as the size of the problem increases.**

input size

function

Empirical analysis of time efficiency

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Use physical unit of time (e.g., milliseconds)
- Analyze and plot the results (the empirical data)



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult.
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used.

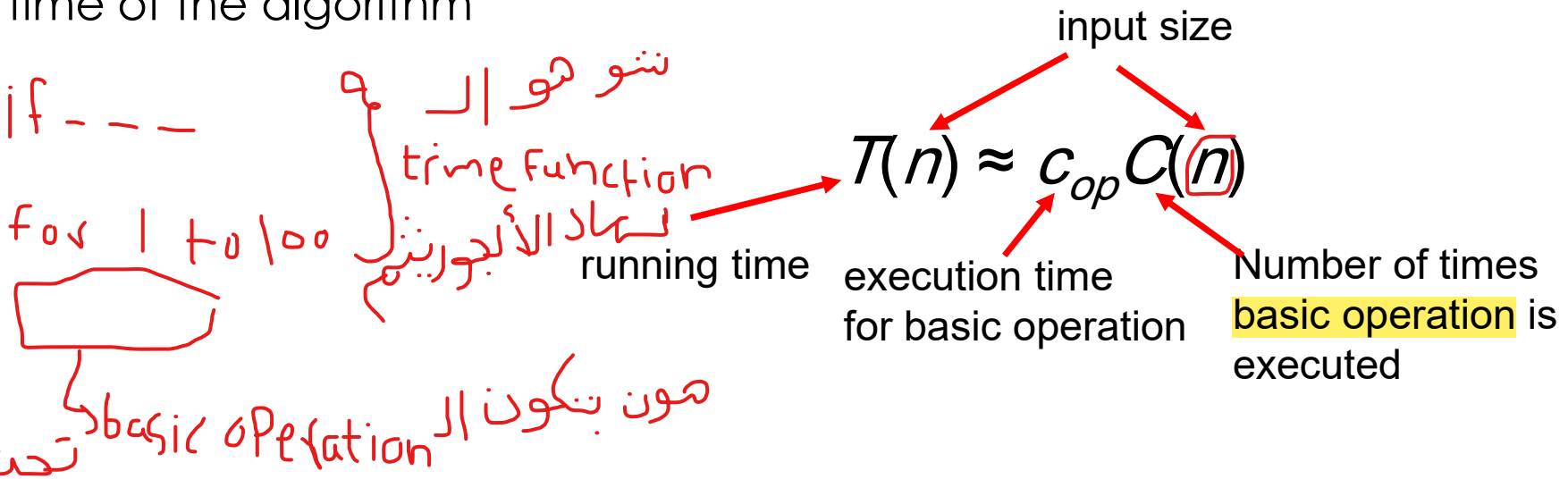
Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, *n*.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

كإني بعطي تحليل عام للالجوريثم بغض النظر بدك تشغلو على machine عادية أو على cloud أو على server أي عندما locally بدك تشتغل أي مشكلة هاد الالجوريثم وبنصا تودي الـ time أو الفنكشن قالكد

Theoretical analysis of time efficiency

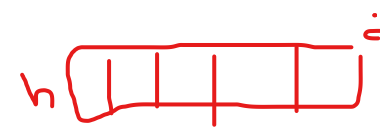
- Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*
- *Basic operation*: the operation that contributes most towards the running time of the algorithm



Assume problem size is *n, we want to express the running time as T(n)*

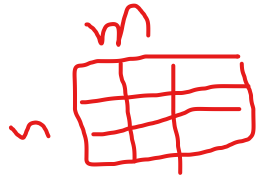
Problem size (Input size)

- Problem size is defined based on the problem at hand.
- Input size (number of elements in the input)
- **Examples**:
  - Number of elements in an array (for sorting problems).
  - # of elements in a matrix
  - Length of the strings in an anagram problem
  - vertices and edges in a graph
  - Number of cities in the Traveling Salesman Problem (TSP).



مصفوفة أحادية

ال Problem size هي n



مصفوفة ثنائية

ال Problem size هي n و m

لو كانت مصفوفة مربعة و n = m.

فال Problem size نعبر بس n

Input size and basic operation examples

Problem	Input size measure	Basic operation
Searching for key in a list of n items	Number of list's items, i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	n'size = number of digits	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

How do we compare algorithms?

- We need to define a number of objective measures.

  (1) Compare execution times?

  Not good: times are specific to a particular computer !!

  (2) Count the number of statements executed?

  Not good: number of statements vary with the programming language as well as the style of the individual programmer.

Ideal Solution

Express running time as **a function of the input size *n*** (i.e., *f(n)*).

- Compare different functions corresponding to running times.
- Such an analysis is independent of machine time, programming style, etc.

Time complexity/order of growth

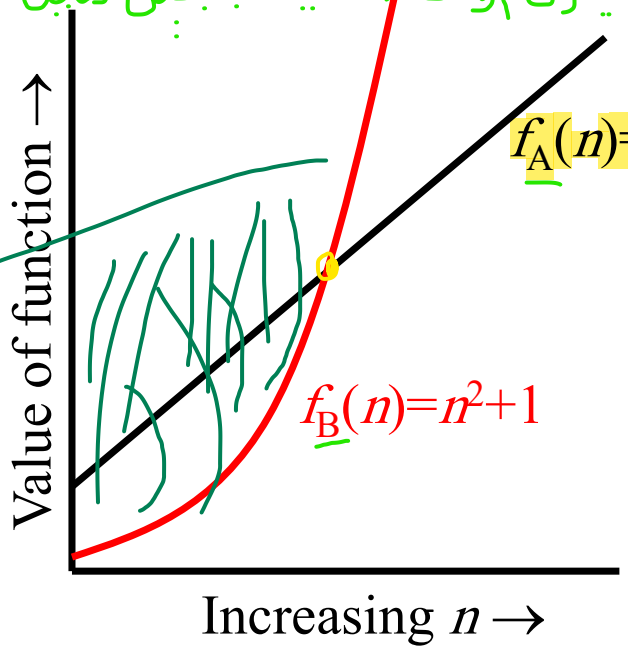
- Time Complexity/Order of Growth defines the amount of time taken by any algorithm with respect to the size of the input.
- Time Complexity specifies how the program would behave as the order of size of input is increased.
- Time Complexity is **commonly represented using Big O notation** i.e. O().
- **Inexact**, but provides a good basis for comparisons.

We will explain Big O notation and other notations later

- We want to have a good judgment on how an algorithm will perform if **the problem size gets very big ($n \to \infty$)**

Visualizing Orders of Growth

- On a graph, as you go to the right, a faster growing function eventually becomes larger...



Complexity Classes

- There are some common time complexity classes.
  - In analysis of algorithm, log refers to $\log_2$, or sometimes written simply as lg.

Notation	Name
O(1)	Constant.
O(log *n*)	Logarithmic.
O(*n*)	Linear.
O(*n* log *n*)	Log-linear or linearithmic
O(n^2)	Quadratic.
O(n^3)	Cubic.
O(n^c), $c > 1$	Polynomial, sometimes called algebraic. Examples: O(n^2), O(n^3), O(n^4).
O(c^n)	Exponential, sometimes called geometric. Examples: O(2^n), O(3^n).
O(*n*!)	Factorial, sometimes called combinatorial.

n is the problem size.

Here Big O notation is used

How to do time analysis?

Finding the time complexity of an algorithm

- **Time complexity** is the computational complexity that describes **the amount of computer time it takes to run an algorithm** as the **input size grows (towards infinity).**
- You want to **lower** the time complexity function to have **better performance**.
- Finding out the time complexity of your code can help you develop better algorithms that run faster.
- In general, you can determine the time complexity by **analyzing the program's statements (go line by line)**
- However, you have to be mindful how are the statements arranged. Suppose they are inside a **loop** or have **function calls** or even **recursion**.

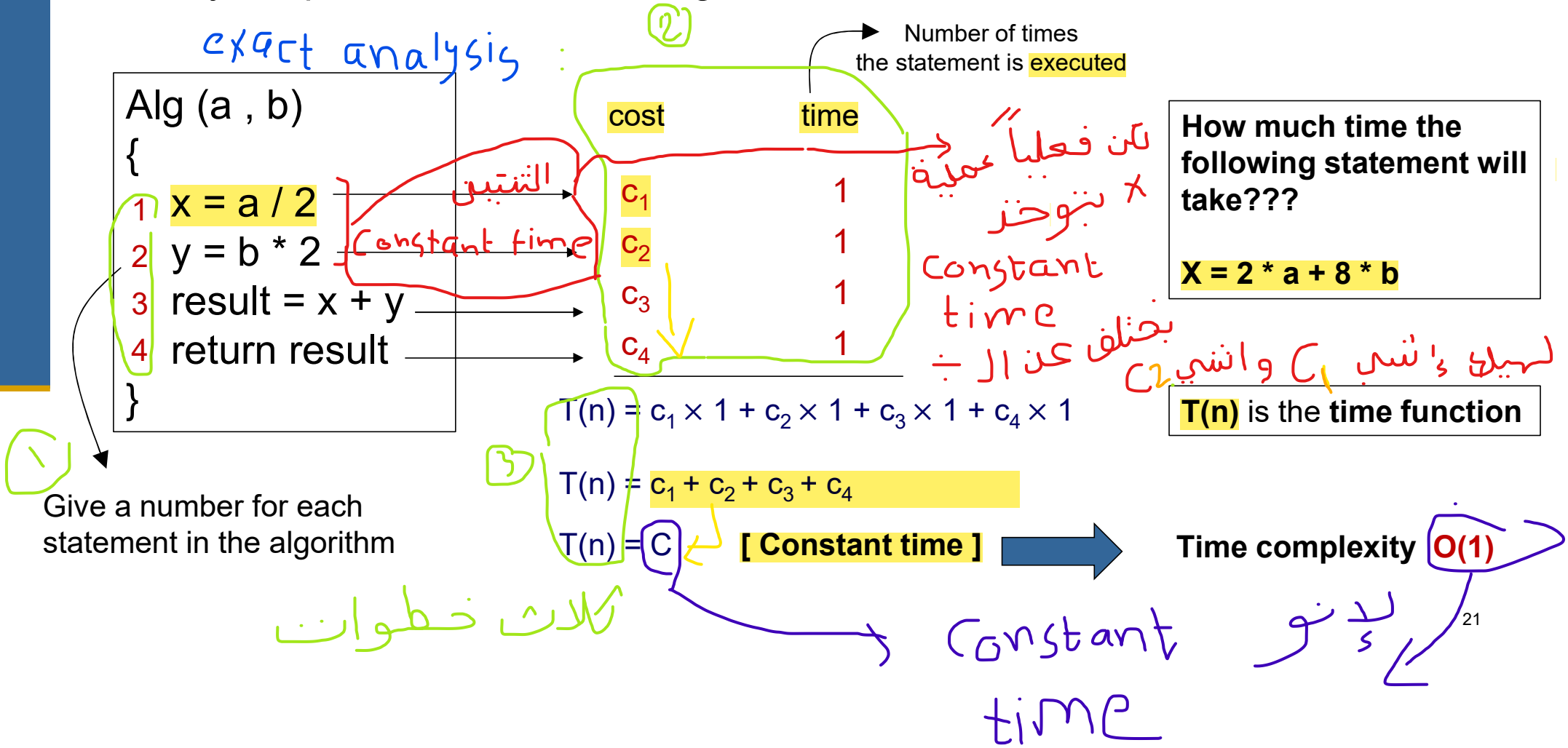
Finding the time complexity of an algorithm

- **In this tutorial you will learn how to find the time complexity of:**
  - Sequential statements.
  - Conditional statements
  - Loop statements(Linear time, Constant time, logarithmic time,,, etc)
  - Nested loop statements.
  - Function call statements.
  - Recursive functions.

Running time of iterative algorithms

Sequential statements

- Every simple statement in an algorithm takes one unit of time



Exercise

- Perform a line-by-line analysis of Algorithm ***squareSum*** and derive its time complexity. This algorithm compute the square sum of 3 numbers.

```
squareSum (x , y , z)
{
    sx = x * x
    sy = y * y
    sz = z * z
    sum = sx + sy + sz
    return sum
}
```

cost time

Note that each statement executes a **basic operation** (math and assignment). **Each line takes constant time**.

T(n) =

Time complexity _____

Example

- Associate a "cost" with each statement.
- Find the "total cost“ by finding the total number of times each statement is executed.

Algorithm 1

	Cost	#times
arr[1] = 0	c_1	1
arr[2] = 0	c_1	1
arr[3] = 0	c_1	1
……	……	
arr[n] = 0	c_1	1

$T(n) = c_1+c_1+...+c_1 = c_1 \times n$

Time complexity O(n)

Algorithm 2

Why n+1 ???

	Cost	#times
FOR i=1 to n	c_1	n+1
arr[i] = 0	c_2	n

$T(n) = c_1(n+1) + c_2(n) = c_1n+c_1\ c_2n$

$= (c_2 + c_1) \times n + c_2$

$= \mathbf{a\ n + b}$ where a, b are constants

Time complexity O(n)

Both algorithms are of the same order: ***O(n)***

Frequency count method for **Loops**

The following algorithm find the sum of an array elements.

Algorithm Sum(A , n)

```
{
1   s = 0
2   For i=1 to n
3       s = s + A [i]
4   return s
}
```

Cost	*time*
c_1	1
c_2	n+1
c_3	n
c_4	1

For check termination condition

If any statement is repeating for some number of times then the frequency of execution of that statement is used to find the time taken

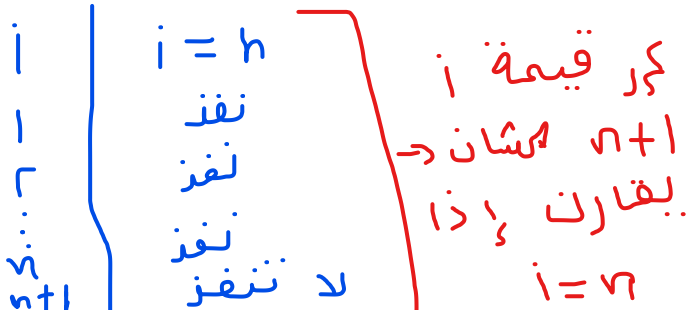
$T(n) = c_1 \times 1 + c_2 \times (n+1) + c_3 \times n + c_4 \times 1$

$T(n) = c_1 + c_2 n + c_2 + c_3 n + c_4$

$T(n) = (c_2 + c_3)\, n + (c_1 + c_2 + c_4)$

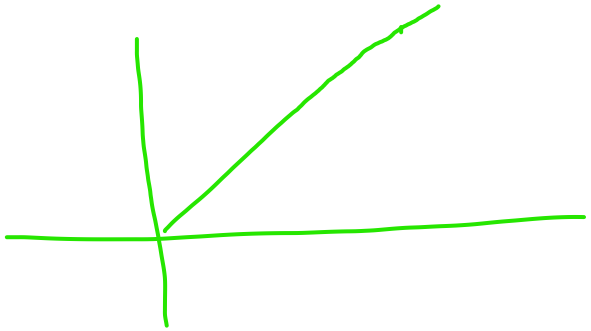
$T(n) = a\, n + b$ for some constants **a** , **b**

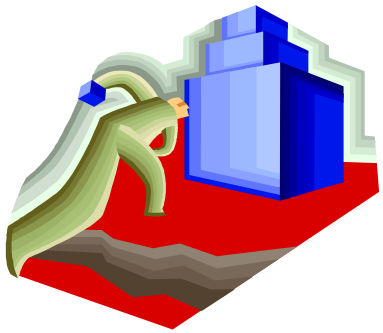
Time complexity O(n)



Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of previous *Sum* algorithm





Note

- Loop Index Starting Point

In this course, we'll start loop indices from 1 instead of 0.

- For instance

```
for i from 1 to n
```

Is functionality the same as

```
for i = 0 to n-1
```

Example (Constant-time Loops)

```
1  For i = 1 to 500
2      statement1
3      statement2
```

Cost	*time*
c_1	501
c_2	500
c_3	500

Constant numbers

$T(n) = c_1 \times 501 + c_2 \times 500 + c_3 \times 500$

$T(n) = C$ [constant time] ⇒ **Time complexity O(1)**

If a constant number bounds the loop, let's say 500 (or even 10000). Then, the runtime is constant

Nested Loops

Example: Find the sum of all elements of a square matrix (2D array)

```
Sum (A , n)
{
1  S = 0;
2  For i=1 to n
3    For j=1 to n
4      S += A [ i ][ j ];
}
```

	Cost	*time*
1	c_1	1
2	c_2	n+1
3	c_3	n×(n+1)
4	c_4	n×n

$T(n) = c_1 + c_2 \times (n+1) + c_3 \times n \times (n+1) + c_4 \times n^2$

$T(n) = c_1 + c_2 n + c_2 + c_3 n^2 + c_3 n + c_4 n^2$

$T(n) = (c_3 + c_4)\, n^2 + (c_2 + c_3)\, n + (c_1 + c_2)$

$T(n) = \mathbf{a}n^2 + \mathbf{b}n + \mathbf{c}$ where a, b, c are constants

Time complexity $O(n^2)$

Exercise: Addition of two matrices of size n x n

Perform a line-by-line analysis of Algorithm ***matrixAddition*** and derive a precise expression T(n) for its running time.

```
Algorithm matrixAddition ( A , B , n)
{                                          Cost        time
    For i=1 to n
       For j=1 to n
          C[i][j] = A[i][j] + B[i][j]
}
```

Nested Loops [cont]

Example: Matrix multiplication

```
Algorithm Multiply ( A , B , n)
{
   for i = 1 to n
      for j = 1 to n
         C[i][j] = 0
         for k =1 to n
            C[i][j] += A[i][k] * B[k][j]
}
```

Cost	*time*
c_1	n+1
c_2	$n \times (n+1)$
c_3	$n \times n$
c_4	$n \times n \times (n+1)$
c_5	$n \times n \times n$

$T(n) = c_1 \times (n+1) + c_2 \times n \times (n+1) + c_3\, n^2 + c_4 \times n^2 \times (n+1) + c_5\, n^3$

$$\mathbf{T(n) = a\, n^3 + b\, n^2 + c\, n + d}$$

Time complexity $\mathbf{O(n^3)}$

Example

Perform a line-by-line analysis of Algorithm ***xyz*** and derive T(n,m) for its running time.

```
A (n, m)
{
1   st1
2   For i=1 to n
3     For j=1 to m
4       st2
5       st3
}
```

Cost	*time*
c_1	1
c_2	n+1
c_3	n×(m+1)
c_4	n×m
c_5	n×m

$T(n,m) = c_1 + c_2 \times (n+1) + c_3 \times n \times (m+1) + c_4 \times n \times m + c_5 \times n \times m$

$T(n,m) = c_1 + c_2 n + c_2 + c_3 nm + c_3 n + c_4 nm + c_5 nm$

$T(n,m) = (c_3 + c_4 + c_5)\, nm + (c_2 + c_3)\, n + (c_1 + c_2)$

Time complexity **O(nm)**

$T(n,m) = \mathbf{a\,nm + bn + c}$ where a, b, c are constants

Example

Perform a line-by-line analysis of Algorithm ***B*** and derive T(n) for its running time.

```
B (n)
{
1  st1
2  For i=1 to n
3     For j=1 to 1000
4        st2
}
```

Cost	*time*
c_1	1
c_2	n+1
c_3	n×(1001)
c_4	n× 1000

Note that a constant number bounds the inner loop

$T(n) = c_1 + c_2 \times (n+1) + c_3 \times n \times 1001 + c_4 \times n \times 1000$

$T(n) = c_1 + c_2 n + c_2 + 1001\ c_3\ n + + 1000\ c_4\ n$

$T(n) = \mathbf{a\ n\ + b}$ where a, b are constants

Time complexity O(n)

Exercise

Perform a line-by-line analysis of Algorithm ***findRowSums*** and derive T(n) for its running time.

Input: A 2D array, arr, of size n x m
Output: An array, rowSums, containing the sum of the first 10 elements in each row
assume m>10

```
Algorithm findRowSums(arr, n, m):                    cost        time
    Initialize an empty array rowSums of size n

    FOR i from 1 to n
        rowSum = 0
        FOR j from 1 to
            rowSum = rowSum + arr[i][j]
        rowSums[i] = rowSum

    return rowSums
```

Example

How many times is the statement ***st1*** executed in the following cases?

For i = 1 to 5
    st1 — 5

For i = 1 to 5 step 2 (a b, i+=2) $\left\lfloor \frac{5-1}{2} \right\rfloor + 1 = 3$
    st1 — 3

For i=3 to 100 step 7
    st1 — 14 $\left(\left\lfloor \frac{100-3}{7} \right\rfloor + 1\right)$

floor: $\lfloor \; \rfloor$ يعني إذا في أعشار بنحذفن

General Rule:

For i = a to b step Inc — # of times the condition is checked = $\left(\left\lfloor \frac{b-a}{Inc} \right\rfloor + 1\right) + 1$

    st1 — # of times the statement is executed = $\left(\left\lfloor \frac{b-a}{Inc} \right\rfloor + 1\right)$

Exercise

1) Perform a line-by-line analysis of Algorithm ***xyz*** and find T(n).

```
xyz (n)
{
1   for i = 2 to n
2       statement1
3   for j=2 to n-1
4       statement2
}
```

منفصلة

(n-1)-2 +2

حسب القانون

(n-1)-2 +1

يلي داخل أقل بـ 1 loop

	Cost	time
1	c_1	n
2	c_2	$n-1$
3	c_3	$n-1$
4	c_4	$n-2$

$n-2+2$

بضرب وبجمع

$\rightarrow O(n)$

1) How many times will statement2 be executed when n = 10?

Exercise

1) Perform a line-by-line analysis for the following algorithm:

```
Alg (n)
{
1   i =1
2   while ( i <= n ) do
3       statement
4       i++
}
```

while حاجة
من ١ ← n

بعد التحليل..

$T(n) = O(n)$

Hint: It is possible to convert while loop into for loop

Exercise

1) Perform a line-by-line analysis for the following algorithm:

```
Alg (n)
{
    i =1
    while ( i <= n ) do
        statement
        i++
}
```

Hint: It is possible to convert while loop into for loop

Example

1) Perform a line-by-line analysis for the following algorithm:

واجب

```
MyAlg (n)
    count = 0
    FOR i from 1 to n step i * 2
        count +=1
    RETURN count
```

Exercise

1) Perform a line-by-line analysis for the following algorithm:

```
MyAlg (n)
    count = 0
    FOR i from 1 to n step i * 2
        FOR j from 1 to n
                count +=1
    RETURN count
```

Example

```
Alg ( n )
{
1  For i=1 to n
2     For j=i to n
3        st1
```

		Cost	time
		c_1	$n+1$
		c_2	$t_1+t_2+t_3+\ldots..t_n = t_i$, i = 1 …n
		c_3	$t_1-1+t_2-1 + \ldots t_n-1 = t_i-1$, i = 1 …n

Note that the number of times the inner loop is executed depends on the value of i in the outer loop

i	j	t_i	t_i-1
1	1	n+1	n
2	2	n	n-1
3	3	n-1	n-2
4	4	n-2	n-3
		-----	-----
n	n	2	1

$\sum$ $\sum$

$T(n) = c1 \times (n+1) + c2 \times \frac{n(n+1)}{2} + n + c2 \times \frac{n(n+1)}{2}$

$T(n) = an^2 + bn + c$

Time complexity $O(n^2)$

$= 2 + 3 + 4 + \ldots\ldots\ldots + n + n+1$ $= 1 + 2 + 3 + 4 + \ldots. + n + n = \frac{n(n+1)}{2} + n$

$= 1 + 2 + 3 + 4 + \ldots. + n = \frac{n(n+1)}{2}$

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

Example

```
Alg ( n )
{
1 For i=1 to n
2     For j=i to n
3         st1
}
```

i	j	t_i	t_i-1
1	1		
2	2		
3	3		
4	4		
n	n		

```
Alg ( n )
{
1 For i=1 to n
2     For j=1 to n
3         st1
}
```

i	j	t_i	t_i-1
1	1		
2	2		
3	3		
4	4		
n	n		

Approximate time complexity

```
Alg ( n )
{
1 For i=1 to n
2     For j=i to n
3         st1
}
```

```
Alg ( n )
{
1 For i=1 to n
2     For j=1 to n
3         st1
}
```

Exercise

Perform a line-by-line analysis of Algorithm ***Test*** and derive a precise expression T(n) for its running time.

```
Test (n)
{
    For i=2 to n
      For j=1 to i
        statement
}
```

What is the time complexity?

Analysis of conditional Statements

- Conditional statements, such as "if" and "else," play a crucial role in algorithm design and impact program flow.
- Execution time depends on the evaluation of the condition.

```
if x > 0:
    // Code block A
else:
    // Code block B
```

واحد منهن دا
بيتحقق

The time complexity depends on the actual value of ‘x’

Example

```
Algorithm Test (n)                cost        time
1    y = 0                        c1          1
2    if (n < 5)                   c2          1
3        y = y * 2                c3          t
     else
4        y = y * 4                c4          1-t
5    return y                     c5          1
```

t = 1 or 0

T(n) = c1 + c2 + c3xt + c4x(1-t) + c5

If t=1 (n<5) : T(n) c1 + c2 + c3 + c5 = A [A is constant]

If t=0 (n>=5): T(n) c1 + c2 + c4 + c5 = B [B is constant]

$$\mathbf{T}(\boldsymbol{n}) = \begin{cases} \boldsymbol{A} & ,\boldsymbol{n} < \mathbf{5} \\ \boldsymbol{B} & ,\boldsymbol{n} \geq \mathbf{5} \end{cases}$$

Time complexity O(n)

Selection statement (if/else)

```
Alg (n)
{
1       For k=1 to n
2           if (k+1) % 2 ==0
3               st1
            else
4               st2
}
```

Cost	*time*
c_1	n+1
c_2	n
c_3	n × t
c_4	n × (1-t)

t = 1 or 0

$T(n) = c_1 \times (n+1) + c_2 \times n + c_3 \times nt + c_4 \times n\,(1-t)$

$T(n) = an + bnt + c$

$$\mathbf{T}(n) = \begin{cases} dn + c, & t = 1 \\ an + c, & t = 0 \end{cases}$$

Time complexity **O(n)**

The worst-case we take whichever is larger

Example

Algorithm Test (n)

{	*Cost*	*time*
y = 0	c_1	1
If (n < 5)	c_2	1
y = y * 2	c_3	t
Else		
For i = 1 to n	c_4	(n+1) × (1 – t)
Y = y + i	c_5	n × (1-t)
return Y	c_6	1
}		

Since n has a higher order than 1, we can express the time complexity as O(n)

$$T(n) = \begin{cases} C, & n < 5 \\ an + b, & otherwise \end{cases}$$

$$T(n) = \begin{cases} O(1), & n < 5 \\ O(n), & otherwise \end{cases}$$

Best case

Worst case

If t = 1 : T(n) = c1 + c2 + c3 + c6 = **C** [constant time]

If t = 0 : T(n) = c1 + c2 + c4(n+1) + c5 n + c6

= an + b

Remember that we **care about the worst-case** so that we will **take the maximum possible runtime**.

Function call statements

When you calculate your programs' time complexity and **invoke a function**, you need to be aware of its runtime.

- If you wrote the function (user-defined), that might be a simple inspection of the implementation.
- If you are using a library function, you might need to check out the language/library documentation or source code.

Function call statements

To calculate time complexity of ***Alg ,*** you need to be aware of the called function ***G***.

Alg (n)

```
{                                   Cost      time
1     For i=1 to n                  c1        n+1
2          G(n)                     TG(n)     n
3     st                            c3        1
}
```

$T(n) = c_1 \times (n+1) + T_G(n) \times n + c_3$

$T(n) = c_1 \times (n+1) + (an + b) \times n + c_3$

$T(n) = an^2 + bn + c$

$O(n^2)$

G (n)

```
{                                   Cost      time
1     For i=2 to n                  c1        n
2          statement                c2        n-1
}
```

$T_G(n) = c_1 \times n + c_2 \times (n-1)$

$T_G(n) = an + b$

O(n)

Example

To calculate time complexity of ***Alg ,*** you need to be aware of the called function ***G***.

Alg (n)

```
{                                   Cost      time
1     For i=1 to n                  c1        n+1
2          G(n)                     TG(n)     n
3     st                            c3        1
}
```

$T(n) = c_1 \times (n+1) + T_G(n) \times n + c_3$

$T(n) = c_1 \times (n+1) + (an + b) \times n + c_3$

$T(n) = an^2 + bn + c$

$O(n^2)$

G (n)

```
{                                   Cost      time
1     For i=2 to n                  c1        n
2          statement                c2        n-1
}
```

$T_G(n) = c_1 \times n + c_2 \times (n-1)$

$T_G(n) = an + b$

O(n)

Example

Assuming that the time complexity of sort() function is O(n logn). What is the time complexity of the following Test:

```
Test (A , x)
{
      If ( x==1 )
           A.sort()          ——————→ The if block has a running time of O(n logn)
          return 1
      else
          return 0           ——————→ The else block has a running time of O(1)
}
```

$$\mathbf{T}(n) = \begin{cases} O(n\,logn) & , \quad x = 1 \\ o\,(1), & otherwise \end{cases}$$

Since n log n has a higher order than 1, we can express the time complexity as O(n log n)

Exercise

You have the following algorithm. fn1() , fn2(), and fn3() are three different functions

```
XYZ ( n )
{
    for i = 1 to n
        fn1()
        for j = 1 to n
            fn2()
            for k = 1 to n
                fn3()
}
```

Find the time complexity of XYZ in the following cases:

- All functions fn1() , fn2(), and fn3() have a runtime of O(1)
- if only fn1 and fn2 are constant and fn3 has a runtime of $O(n^2)$

Exercise

- Perform a line-by-line analysis of the algorithm ***arrayMax*** and derive a precise expression of T(n) of its running time. What is the order of growth?

Input array ***A*** of ***n*** integers
Output maximum element of ***A***

```
Algorithm arrayMax (A, n)
{
   currentMax ← A[1]
   for i ← 2 to n  do
      if A[i] > currentMax then
            currentMax ← A[i]
    return currentMax
}
```

Exercise

- 1) Perform a line-by-line analysis of the algorithm ***prefixAverages1*** and ***prefixAverages2*** and derive a precise expression of T(n) of their running time.

Input array X of n integers
Output array A of prefix averages of X

```
Algorithm prefixAverages1(X, n)
    A ← new array of n integers
    for i ← 1 to n do
        s ← X[1]
        for j ← 2 to i do
            s ← s + X[j]
        A[i] ← s / (i + 1)
    return A
```

```
Algorithm prefixAverages2(X, n)
    A ← new array of n integers
    s ← 0
    for i ← 1 to n do
        s ← s + X[i]
        A[i] ← s / (i + 1)
    return A
```

2) Compare the order of growth of ***prefixAverages1*** and ***prefixAverages2.*** which of them is more time efficient