ALGORITHM COMPLEXITY

INSTRUCTOR: KASHFIA SAILUNAZ

SOME PARTS ARE ADAPTED FROM THE TEXTBOOK & ENSF 593/594 LECTURE02 BY MOHAMMAD MOSHIRPOUR

OUTLINE

- Algorithmic Complexity
- Complexity Analysis
- Asymptotic Analysis
- Notations
- Complexity Measures
- Rules
- Examples

LEARNING OUTCOME

- At the end of this lecture, we will be able to-
  - Understand the necessity of complexity analysis,
  - Define and distinguish different types and notations of complexity analysis, and
  - Analyze complexities of algorithms.

ALGORITHMIC COMPLEXITY

- Algorithms must be –
  - correct,
  - efficient, and
  - easy to implement.
- Algorithmic Complexity : A measure of the performance of any algorithm or computation based on-
  - time required – Time Complexity
  - space required – Space Complexity
  - number of steps required – Computational Complexity
- Algorithmic complexity is measured with respect to the input size
  - Input size : n
  - Algorithmic Complexity : function of n

COMPLEXITY ANALYSIS

- Empirically – implementation based
- Logically – analyzing algorithms step by step

- // Compute the maximum element in the array a.

Algorithm max(a, n):

max ← a[0]	2 operations
i ← 1	1 operation
while i ≤ n-1 do	2 operations, n times
if max < a[i] then	2 operations, n-1 times
max ← a[i]	2 operations, n-1 times
i ← i + 1	2 operations, n-1 times
return max	1 operation
	Total = summation of all

COMPLEXITY ANALYSIS

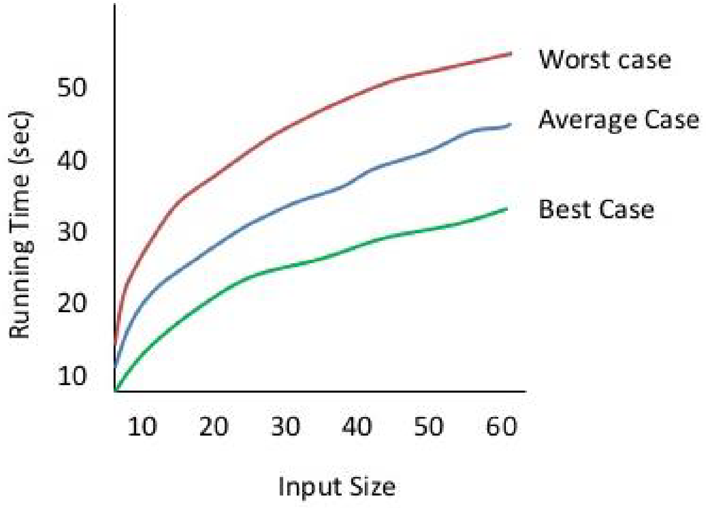
- Best case scenario : the first element a[0] is the maximum
- Worst case scenario : a is in ascending order and the last element is the maximum
- Average case scenario : others

- Best case : 2 + 1 + 2n + 4 (n-1) + 1 = 6n
- Worst case : 2 + 1 + 2n + 6 (n-1) + 1 = 8n - 2

```
Algorithm max(a, n):
        max ← a[0]                      2 operations
        i ← 1                           1 operation
        while i ≤ n-1 do                2 operations, n times
            if max < a[i] then          2 operations, n-1 times
                max ← a[i]              2 operations, n-1 times
            i ← i + 1                   2 operations, n-1 times
        return max                      1 operation
```

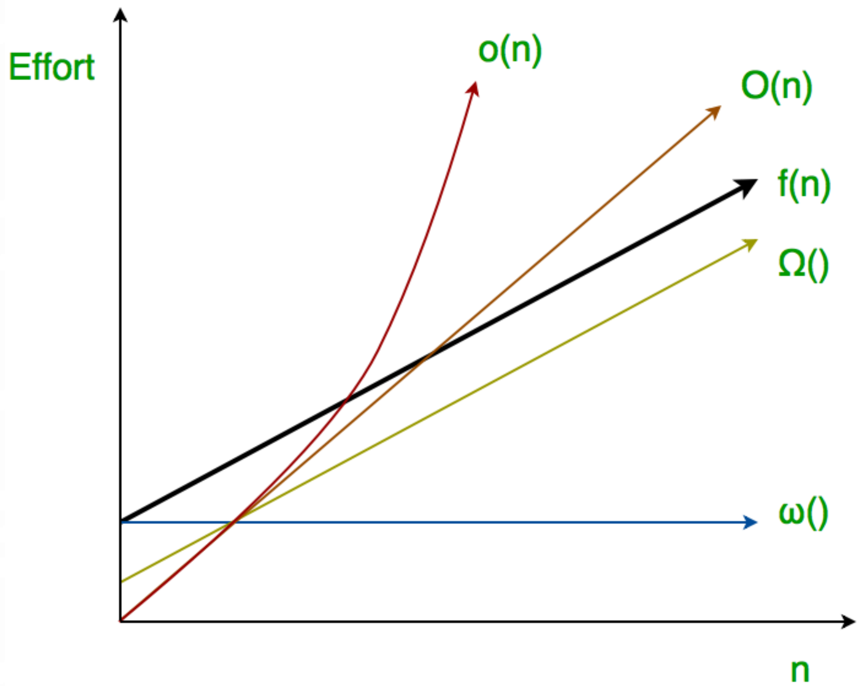
ASYMPTOTIC ANALYSIS

- Running time of operations/algorithms in mathematical units of computations
- Allows to simplify the complexity analysis
- Ignores constants, lower order terms etc.
  - For example, 8n – 2 will be O(n)
- Scenarios:
  - Best case – minimum time required for algorithm execution
  - Average case – average time required for algorithm execution
  - Worst case – maximum time required for algorithm execution



NOTATIONS

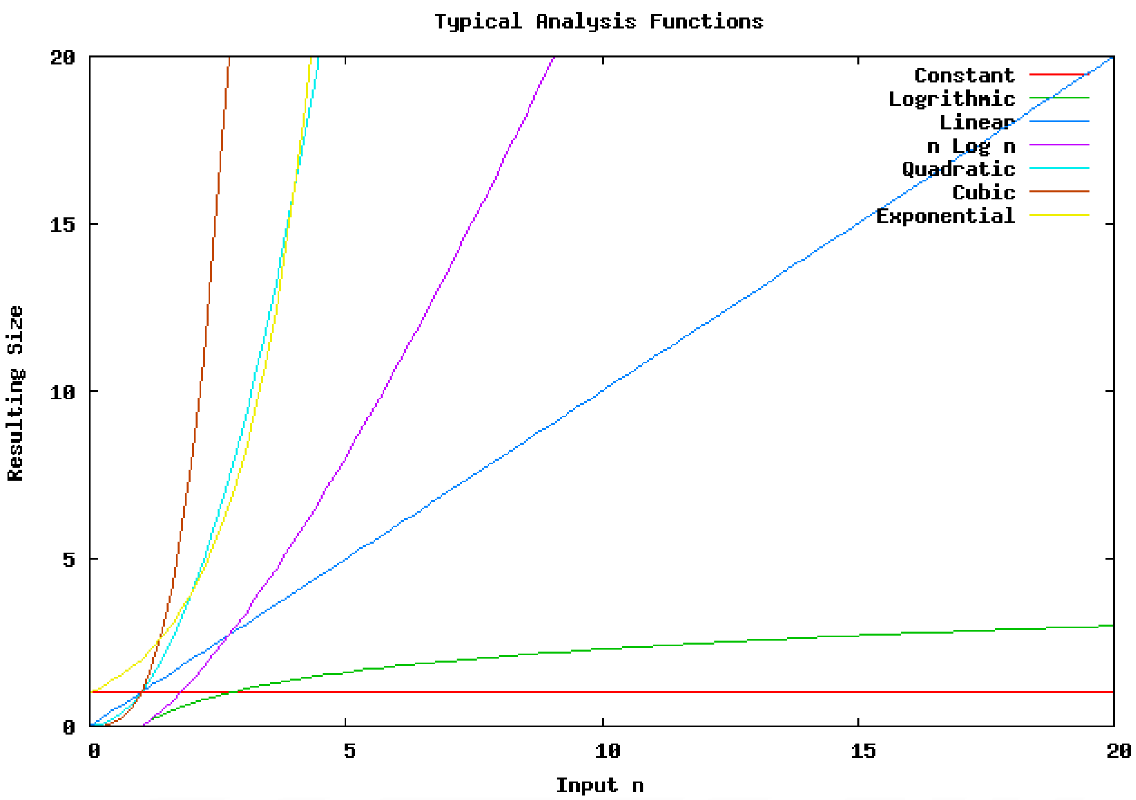
- **O – Big Oh Notation – upper bound/worst case time complexity**
- Ω – Big Omega Notation – lower bound/best case time complexity
- Θ – Theta Notation – upper and lower bound simultaneously
- o – Little Oh Notation – strict upper bound
- ω – Little Omega Notation – strict lower bound



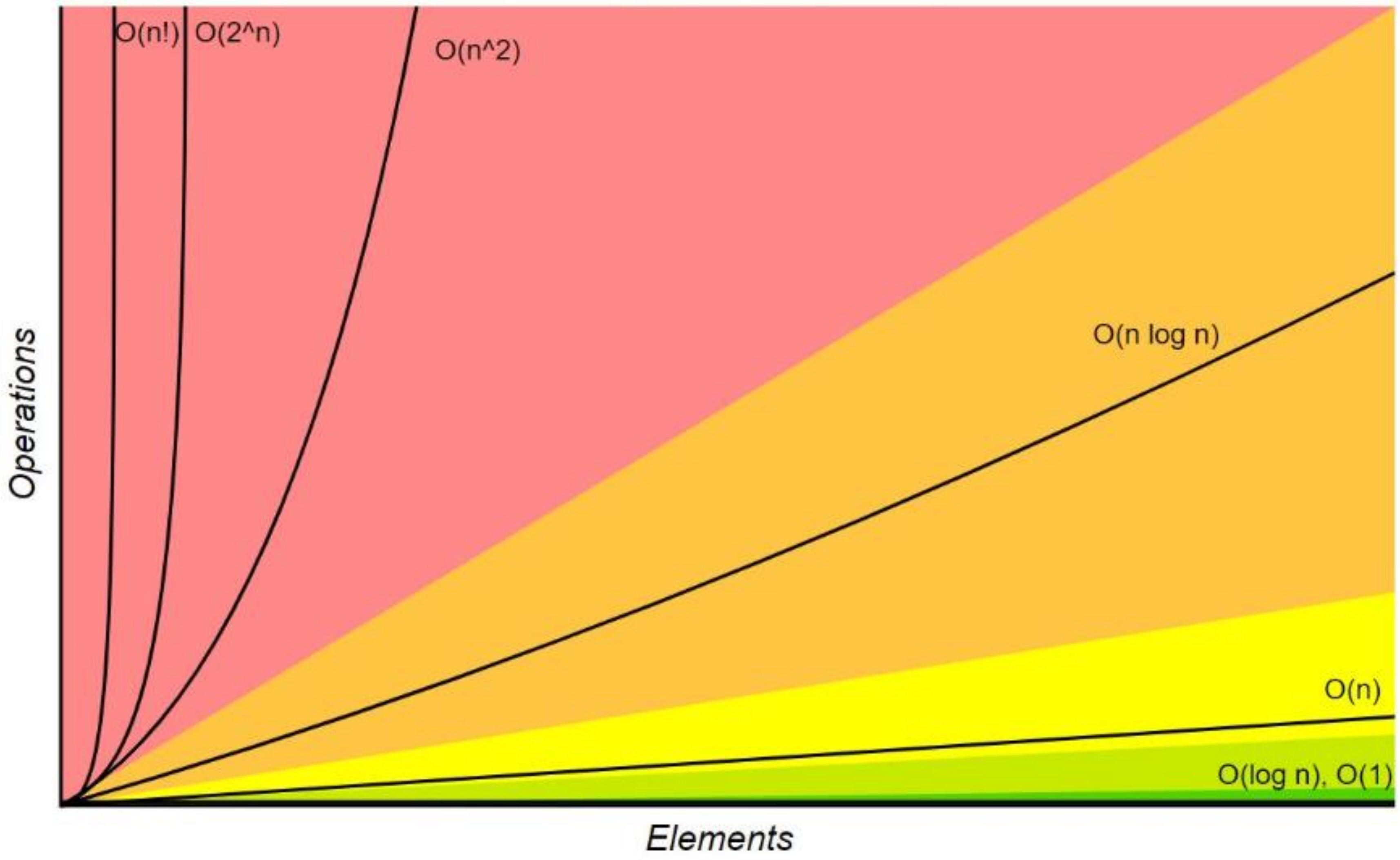
Source: https://www.geeksforgeeks.org/complete-guide-on-complexity-analysis/

COMPLEXITY MEASURES

- Classes of Algorithms:
  - O(1) constant
  - O(lg n) logarithmic
  - O(n) linear
  - O(n lg n) N Log N
  - O(n^2) quadratic
  - O(n^3) cubic
  - O(2^n) exponential



COMPLEXITY MEASURES



Source: https://devopedia.org/algorithmic-complexity

EXAMPLES

- // Compute the maximum element in the array a.

Algorithm max(a, n):

max ← a[0]	2 operations
i ← 1	1 operation
while i ≤ n-1 do	2 operations, n times
if max < a[i] then	2 operations, n-1 times
max ← a[i]	2 operations, n-1 times
i ← i + 1	2 operations, n-1 times
return max	1 operation
	Total = summation of all

Worst case : a is in ascending order and the last element is the maximum

$2 + 1 + 2n + 6\ (n-1) + 1 = 8n - 2$

EXAMPLES

- // Compute the maximum element in the array a.

```
Algorithm max(a, n):
        max ← a[0]                  O(1)        2 operations
        i ← 1                       O(1)        1 operation
        while i ≤ n-1 do            O(n)        2 operations, n times
            if max < a[i] then      O(n)        2 operations, n-1 times
                max ← a[i]          O(n)        2 operations, n-1 times
            i ← i + 1               O(n)        2 operations, n-1 times
        return max                  O(1)        1 operation
                                                ______________________
                                                Total = summation of all
```

Worst case : a is in ascending order and the last element is the maximum

$2 + 1 + 2n + 6\ (n-1) + 1 = 8n - 2 = O(n)$

RULES OF COMPLEXITY CALCULATIONS

- Big O – worst case scenario
- Simple statements that don't depend on inputs are O (1)
  - i.e. take constant time
- Ignore differences in execution times for simple statements
  - Multiplicative constants are discarded in big O analysis
- Use the worst case for conditional statements
  - i.e. Take the "longest path" through the algorithm
- If the number of steps is halved on each iteration of a loop, then the complexity is O(lg n)
  - Also true if multiplying by 1/3, ¼, etc.
- Sum rule: if the complexity of a sequence of statements is the sum of two or more terms, discard the lower order terms
  - i.e., $n^3 + n^2$ is O (n^3) + O (n^2) = O(n^3)
- Product rule: if a process is repeated for each n of another process, then O is the product of the O s of each process
  - i.e. Nested loop processing of a 2 D array is O (n). O (n) = O (n . n) = O(n^2)

EXAMPLES - 1

Algorithm squareSum (a, b, c, d) :

s_a ← a * a	O(1)
s_b ← b * b	O(1)
s_c ← c * c	O(1)
s_d ← d * d	O(1)
sum ← s_a + s_b + s_c + s_d	O(1)
return sum	O(1)
	O(1) + O(1) + O(1) + O(1) + O(1) + O(1)

COMPLEXITY = O(1)

EXAMPLES - 2

- Algorithm oddEvenCheck (a):

if a%2 = 0 then	O(1)
s ← "even"	O(1)
else s ← "odd"	O(1)
return s	O(1)

COMPLEXITY : O(1)

EXAMPLES - 3

- Algorithm arraySortingCheck (a, flag):

if (flag) then	O(1)
sort(a)	O(n lg n)
return true	O(1)
else return false	O(1)

COMPLEXITY : O(n lg n)

EXAMPLES - 4

- Algorithm simpleLoop :

x ← 0	O(1)
y ← 10	O(1)
for i ← 0 to 3 do	O(1)
x ← x + l	O(1)
y ← y – 1	O(1)
return x, y	O(1)

COMPLEXITY : O(1)

EXAMPLES - 5

- Algorithm simpleLoop :

s ← 0	O(1)	
for i ← 0 to n do	O(n)	$\left.\right\}$ O(n) * O(1) = O (n)
s ← s + I	O(1)	
return s	O(1)	

COMPLEXITY : O(n)

EXAMPLES - 6

- Algorithm doubleLoop:

s ← 0	O(1)	
for i ← 0 to n do	O(n)	
for k ← 0 to m do	O(m) or O(n)	O(n) * O(n) * O(1) = O (n^2)
s ← s + 1	O(1)	
return s	O(1)	

COMPLEXITY : O(n^2)

EXAMPLES - 7

- Algorithm tripleLoop:

x ← 0	O(1)
y ← 0	O(1)
for i ← 0 to n do	O(n)
for j ← 0 to n do	O(n)
for k ← 0 to n do	O(n)
x ← x + 1	O(1)
y ← y + 3	O(1)
return s	O(1)

O(n) * O(n) * O(n) = O (n^3)

COMPLEXITY : O(n^3)

EXAMPLES - 8

- Algorithm reverseLoop:

s ← 0	O(1)	
for i ← 0 to n do	O(n)	$O(n) * O(n) * O(1) = O\ (n^2)$
for k ← m to 1 do	O(m) or O(n)	
s ← s + 1	O(1)	
return s	O(1)	

COMPLEXITY : $O(n^2)$

EXAMPLES - 9

- Algorithm decreasedLoop:

s ← 0	O(1)
i ← 0	O(1)
while i < n do	O(lg n)
s ← s + 1	O(1)
i ← i * 2	O(1)
return s	O(1)

O(lg n) * O(1) * O(1) = O (lg n)

COMPLEXITY : O(lg n)

EXAMPLES - 10

- Algorithm decreasedLoop2:

s ← 0	O(1)
i ← n	O(1)
while i > 0 do	O(lg n)
s ← s + 1	O(1)
i ← i/2	O(1)
return s	O(1)

O(lg n) * O(1) * O(1) = O (lg n)

COMPLEXITY : O(lg n)

EXAMPLES - 11

- Algorithm dependentLoop:

s ← 0 O(1)

for i ← 0 to n do O(n)

for k ← i+1 to n do $O(n^2)$ = 1+2+…..+(n-2)+(n-1) = n(n-1)/2= $n^2/2$ – n/2

s ← s + 1 O(1)

return s O(1)

COMPLEXITY : $O(n^2)$

EXAMPLES - 12

- Algorithm recursiveFunc:
- Factorial of n = n! = n * (n-1) * (n-2) *…..* 2 * 1 = n * (n-1)!

- Iterative Algorithm
- Recursive Algorithm

EXAMPLES - 12

- Iterative Algorithm

Factorial(n):

f ← 1	O(1)
for i ← 2 to n do	O(n)
f ← f * i	O(1)
return f	O(1)

O(n) * O(1) = O (n)

COMPLEXITY: O(n)

- Recursive Algorithm

Factorial(n):

if (n==0) then	O(1)
return 1	
else	O(n)
return n * Factorial(n-1)	

COMPLEXITY: O(n)

EXAMPLES - 12

- Why is 'else' part O(n)?

3 operations – 1 comparison, 1 multiplication, 1 subtraction

$T(n) = T(n-1) + 3$

$= T(n-2) + 6$

$= T(n-3) + 9$

$= T(n-4) + 12$

……………………….

$= T(n-r) + 3r$

$T(0) = 1$, so we need to find r so that $(n-r) = 0$

If $n-r = 0$, then $r = n$

So, $T(n) = T(0) + 3n = 1+3n = O(n)$

- Recursive Algorithm

```
Factorial(n):
  if (n==0) then                O(1)
      return 1
  else                          O(n)
     return n * Factorial(n-1)
```

COMPLEXITY: O(n)

EXAMPLES - 13

- Algorithm for Fibonacci Series

0,1,1,2,3,5,8,13,21,34,55,89,144,…

OR

1,1,2,3,5,8,13,21,34,55,89,144,…

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

$F1 = 1$ (or 0), $F2 = 1$

EXAMPLES - 13

- Iterative Algorithm

Input: Some non-negative integer n
Output: The nth number in the Fibonacci Sequence
$A[0] \leftarrow 0;$
$A[1] \leftarrow 1;$
for $i \leftarrow 2$ ***to*** $n-1$ **do**
  $A[i] \leftarrow A[i-1] + A[i-2];$
return $A[n-1]$

- Recursive Algorithm

Input: Some non-negative integer n
Output: The nth number in the Fibonacci Sequence
if $n \leq 1$ **then**
  return n
else
  return $F(n-1) + F(n-2);$

Try it yourself

THANK YOU