CM1602 : Data Structures and Algorithms for AI

2. Analysis of algorithms

Lecture 2 | R. Sivaraman

ROBERT GORDON UNIVERSITY ABERDEEN
TEF Gold
INFORMATICS INSTITUTE OF TECHNOLOGY

MODULE CONTENT

Lecture	Topic
Lecture 01	Introduction to Fundamentals of Algorithms
Lecture 02	Analysis of Algorithms
Lecture 03	Array and Linked Lists
Lecture 04	Stack
Lecture 05	Queue
Lecture 06	Searching algorithms and Sorting algorithms
Lecture 07	Trees
Lecture 08	Maps, Sets, and Lists
Lecture 09	Graph algorithms

Learning Outcomes

- Covers LO2 (LO2 : Evaluate algorithms and data structures using the theory of complexity analysis) for Module
- On completion of this lecture, students are expected to be able to:
  - Describe complexity analysis
  - Evaluate algorithms and data structures using complexity analysis for performance

Analysis of Algorithm

- **Analysis of Algorithms** is the determination of the amount of **time, storage** and/or other **resources** necessary to execute them.
- Analyzing algorithms is called **Asymptotic Analysis.**
- **Asymptotic Analysis** evaluate the performance of an algorithm.

Time Complexity

- **Time complexity** of an algorithm quantifies the amount of **time** taken by an algorithm.

- We can have three cases to analyze an algorithm:
  1. Worst Case
  2. Average Case
  3. Best Case

- Consider the algorithm given below:

```
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i+1
    return -1
```

- **Worst Case Analysis:** In the worst case analysis. we calculate upper bound on running time of an Algorithm

```
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i+1
    return -1
```

- **Worst Case Analysis** : The case that causes maximum number of operations to be executed.

- For Linear Search, the worst case happens when the element to be searched is not present in the array. (example : search for number 8).

- The worst case time complexity of linear search would be O(n).

```
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i+1
    return -1
```

- **Average Case Analysis:** We take all possible inputs and calculate computing time for all of the inputs.

```
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i+1
    return -1
```

- **Best Case Analysis:** Calculate lower bound on running time of an algorithm.

```
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i+1
    return -1
```

- The best case time complexity of linear search would be O(1).

```
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i+1
    return -1
```

- **Best Case Analysis** : The Case that causes minimum number of operations to be executed.

- For Linear Search, the best case happens when the element to be searched is present at the first location. (example : search for number 2).

- Most of the times, we do **worst case** analysis to analyze algorithms
- The **average case** analysis is not easy to do in most of the practical cases it is rarely done
- The **best case** analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information.

Asymptotic Notations

1. **Big O Notation**: is an Asymptotic notation for the **worst case**.

2. **Ω Notation** (omega notation): is an Asymptotic notation for the **best case**.

3. **Θ Notation** (theta notation): is an Asymptotic notation for the **worst case** and the **best case**.

Big O Notation

O(1)

- Time complexity of a function (or a 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()** function has **O(1)** time complexity.

```
def swap(s1, s2):
    return s2, s1
```

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

```
# c is constant
c=4
for i in range(1,c):
    #some O(1) expressions
    #print(i)
```

O(n)

- Time complexity of a loop is considered as O(n) if the loop variables is incremented/ decremented by a constant amount. For example:

```
# n is variable
# c is increment
for i in range(1,n,c):
    #some O(1) expressions
    print(i)
```

- Another example:

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
     // some O(1) expressions
}

for (int i = n; i > 0; i -= c) {
     // some O(1) expressions
}
```

$O(n^2)$

- Time complexity of nested loops is equal to the number of times the innermost statement is executed. $O(n^2)$ example:

```
# n is variable
# c is increment
for i in range(1,n,c):
    #some O(1) expressions
    for j in range(1,n,c):
        #some O(1) expressions
        print(i,j)
```

- Another example:

```
for (int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
        // some O(1) expressions
    }
}

for (int i = n; i > 0; i += c) {
    for (int j = i+1; j <=n; j += c) {
        // some O(1) expressions
}
```

O(log(n))

- Time complexity of a loop is considered as O(log(n)) if the loop variable is divided/ multiplied by constant amount.

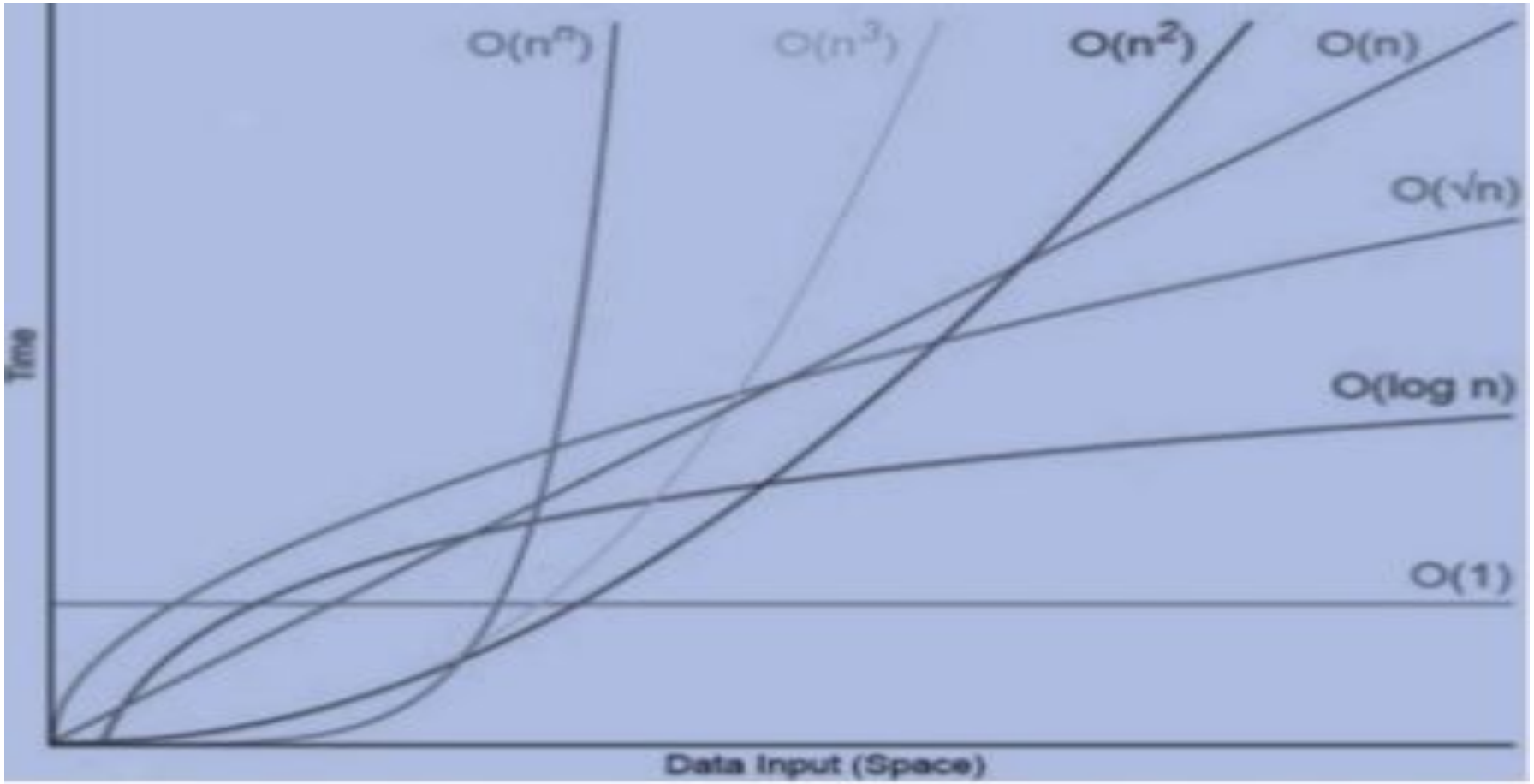
```
# n is variable
# c is constant
i=2
while i<=n:
    print(i)
    i=i*c
```

- Another example:

```
for (int i = 1; i <=n; i *= c) {
    // some O(1) expressions
}
for (int i = n; i > 0; i /= c) {
    // some O(1) expressions
}
```

Growth Orders

n	O(1)	O(log(n))	O(n)	O(nlog(n))	$O(N^2)$	$O(2^n)$	O(n!)
1	1	0	1	1	1	2	1
8	1	3	8	24	64	256	$40xx10^3$
30	1	5	30	150	900	$10x10^9$	$210x10^{32}$
500	1	9	500	4500	$25x10^4$	$3x10^{150}$	$1x10^{1134}$
1000	1	10	1000	$10x10^3$	$1x10^6$	$1x10^{301}$	$4x10^{2567}$
$16x10^3$	1	14	$16x10^3$	$224x10^3$	$256x10^6$	-	-
$1x10^5$	1	17	$1x10^5$	$17x10^5$	$10x10^9$	-	-


O(n^n)
O(n^3)
O(n^2)
O(n)
O(√n)
O(log n)
O(1)
Time
Data Input (Space)