CS211
Algorithms & Data Structures

Lecture 03

1444 - 2022

Dr. Sameer Mabrouk Alrehaili
College of Science and Computer Engineering, Yanbu

Chapter 2

Algorithm Analysis

Algorithms & Data Structures CS 211

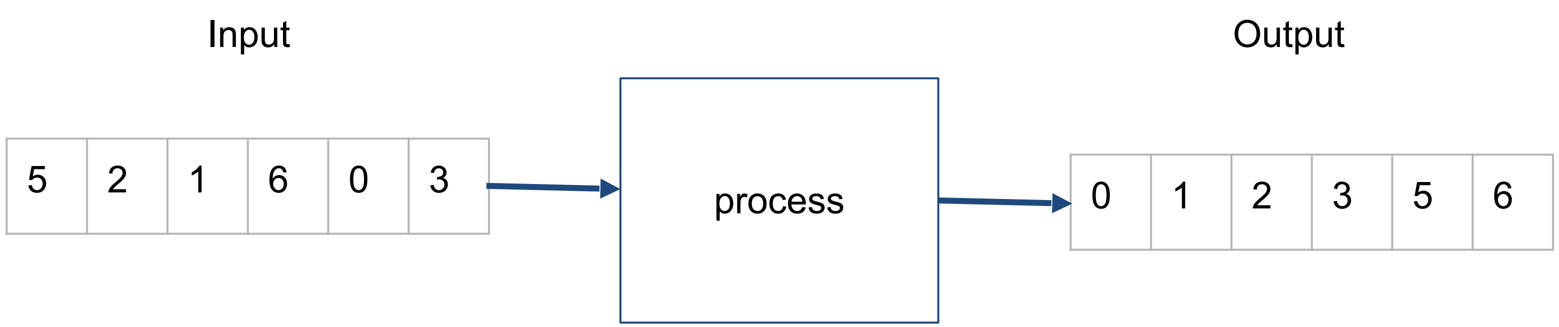
Objectives

- To measure the efficiency of an algorithm using experimental and theoretical approaches.
- Time and space complexity
- Worst case analysis
- Big-Oh notation
- Primitive operations

Algorithms & Data Structures CS 211

Efficiency

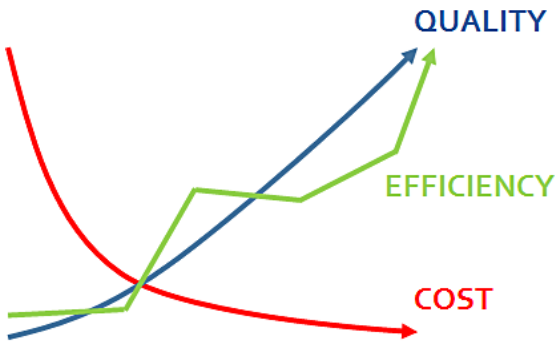
- After we covered how to write an algorithm in pseudocode, here we will cover how to analyse an algorithm.



Algorithms & Data Structures CS 211

Algorithm Analysis

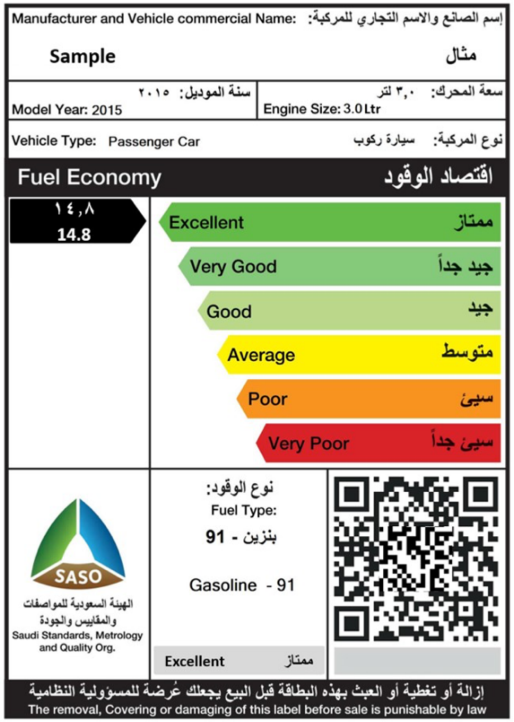
- Algorithm analysis is a methodology of measuring the amount of computational resources that an algorithm requires.
- Algorithm analysis is a methodology of measuring the efficiency of an algorithm.
- Efficiency = the amount of computational resources that an algorithm requires.

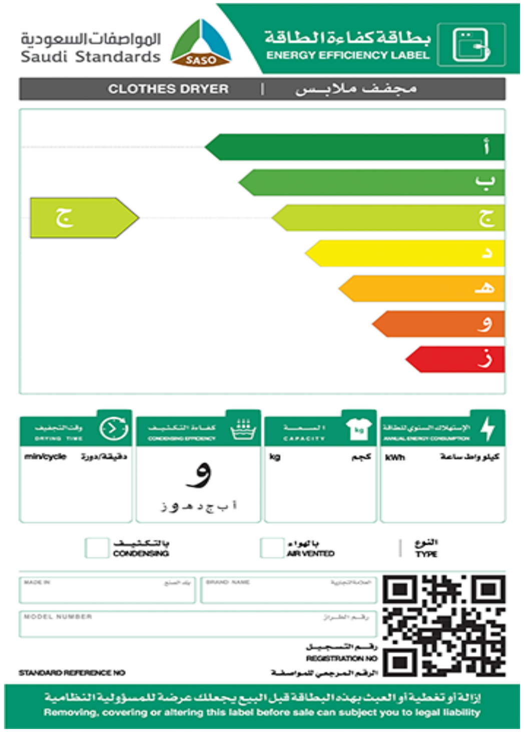


Algorithms & Data Structures CS 211

Efficiency

- Most fuel-efficient cars saves your money every time you fill up.
- Most efficient dryer machine will save money on your utility bills.





Algorithms & Data Structures CS 211

Algorithm Efficiency

- One important factor in developing an algorithm its efficiency.
- Efficiency (or complexity) is a measure of the amount of computational resources (time and space) that a particular algorithm consumes when it runs.
- Therefore an algorithm is considered efficient if its resource consumption (computational cost) is at below some acceptable level.
- Usually, the efficiency of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

Algorithms & Data Structures CS 211

Algorithm Efficiency

- There are different kinds of efficiency, such as financial cost, and use of resources. We will focus on time efficiency.
- Time efficiency: a measure of amount of time for an algorithm to execute.
- Space efficiency: a measure of the amount of memory needed for an algorithm to execute.

Algorithms & Data Structures CS 211

Efficiency vs Understandability

- It is important to write simple and understandable algorithm.
- While it is important to consider efficiency, it is not necessary to try and find the most efficient algorithm.
- Very efficient algorithm may be harder to understand.

Algorithms & Data Structures CS 211

Algorithm Analysis

- As mentioned before, there are a range of different approaches to solve a problem. But which one of them is the most efficient solution?

- How do we measure an algorithm efficiency?
- Algorithms can be analysed in two main ways:
  - Experimental analysis
  - Theoretical analysis (Asymptotic analysis)

Algorithms & Data Structures CS 211

Algorithm Analysis Approaches

Experimental Analysis

- Implementing an algorithm and run it with varying input size.
- Get the **actual running time**
  - Run the program using a method like System.currentTimeMillis() to get an accurate measure of the actual running.
- Implementation is difficult and time consuming
- To compare two algorithms, the same hardware and software environments must be used.

Theoretical Analysis

- Count the number of <u>primitive operations</u>
- Get **theoretical estimates** for the resources needed.
- Evaluates algorithms in a way that is independent from the hardware and software environments.
- This indicates how this number depends on the size of the input.
- <u>Primitive operations</u> are basic computations performed by an algorithm. For example, Addition, subtraction, multiplication, memory access, ….etc
- Non-basic operation
  - Sorting, searching, …. etc

Algorithms & Data Structures CS 211

Example of Experimental Analysis

```
public class test{
          public static void main(String[] args){

                    long start = System.currentTimeMillis();
                    //code should be here
                    long end = System.currentTimeMillis();
                    long elapsed = end-start;
                    System.out.println("Running time is "+ Elapsed + "ms");
          }
}
```

n	Time (ms) on Xeon(R) E3-1220 v6 3.5 GHz x4 (Quad-Core)	M1 8-core CPU 16-core Neural Engine, 14-core GPU 3.2 GHz x8 (Octa-Core)
10	0	0
100	2	1
1,000	7	3
10,000	23	12
100,000	121	69
1,000,000	1035	614
10,000,000	10,051	6044

Algorithms & Data Structures CS 211

Primitive operations

- Primitive operations is the low-level computations

operation	example	cost
Addition	a + b	1
Subtraction	a - b	1
Multiplication	a * b	1
Division	a / b	1
Comparing two numbers	a < b	1
Assigning a value	A←4, a←c	1
Indexing into an array	a[0]	1
Calling a method	max(A,10)	1
Returning from a method	return max	1
Evaluating an expression	a←a+1	2

Algorithms & Data Structures CS 211

Example of Theoretical Analysis

- By inspecting the following pseudocode, we can determine the maximum number of primitive operations executed, as a function of the input size. **f(n) or T(n)**

```
for i←0;i<n;i←i+1 do                    1+(n+1)+2n
            Print a[i]                  2n
end for
```

The running time is **T(n)** = 1+n+1+2n+2n

=5n+2 >=O(n)

Algorithms & Data Structures CS 211

Counting Primitive Operations

- Count the number of primitive operations executed by the following algorithm, as a function of the input size.

```
Function max(A, n)
        max=<--A[0]                                     2
        for i←1;i<n;i←i+1 do                            1+n+2(n-1)
                If a[i]>max then                        2(n-1)
                        Max←a[i]                        2(n-1)
                end if
        end for                                         1
        return max
End max
```

The running time **T(n)** = 2+1+n+2(n-1)+2(n-1)+2(n-1)+1

=3+n+2n-2+2n-2+2n-2+1=7n-2

=7n-2 >=O(n)

Algorithms & Data Structures CS 211

Counting Primitive Operations

- Count the number of primitive operations executed by the following algorithm, as a function of the input size.

```
Function multiply(A, n)
        P←1                                         1
                                                    1+
        for i←0;i<n;i←i+1 do                        (n+1)+2n
                                                    3n
                P←P*A[i]
        end for                                     1
        return P

End multiply
```

The running time **T(n)** = 1+1+n+1+2n+3n+1

=6n+4>=O(n)

Algorithms & Data Structures CS 211

Counting Primitive Operations

- Count the number of primitive operations executed by the following algorithm, as a function of the input size.

```
Function average(A, n)
      Avg←0                                  1
                                             1+
      for i←0;i<n;i←i+1 do                   (n+1)+2n
                                             3n
            Avg←Avg + A[i]
      end for                                2
      return Avg/n
End average
```

The running time $\mathbf{T(n)} = 1+1+n+1+2n+3n+2$

$=6n+5>=O(n)$

Algorithms & Data Structures CS 211

Analysis Types

- There are three cases to analyse the complexity of an algorithm:

  Let's assume you want to find the element that hold 1
- Best case (very rarely used)

1	0	2	3	4	5

- Average case (Rarely used)

4	2	1	5	0	3

- Worst case (Mostly used)

4	2	3	5	0	1

  - Average case time is often difficult to determine.
  - We focus on the worst case running time.

Algorithms & Data Structures CS 211

Running time

- we shouldn’t really care about the exact number of operations that are performed; instead, we should care about how the number of operations relates to the problem size.
- The fastest algorithm for 100 items may not be the fastest for 10,000 items.
- The running time of an algorithm typically grows with the input size.
- **Algorithm’s growth rate** is a measure of how quickly the time of an algorithm grows as a function of problem size.
- To express the time complexity of an algorithm, we use something called the “Big O notation”. **The Big O notation is a language we use to describe the time complexity of an algorithm**. It’s how we compare the efficiency of different approaches to a problem, and helps us to make decisions.

Algorithms & Data Structures CS 211

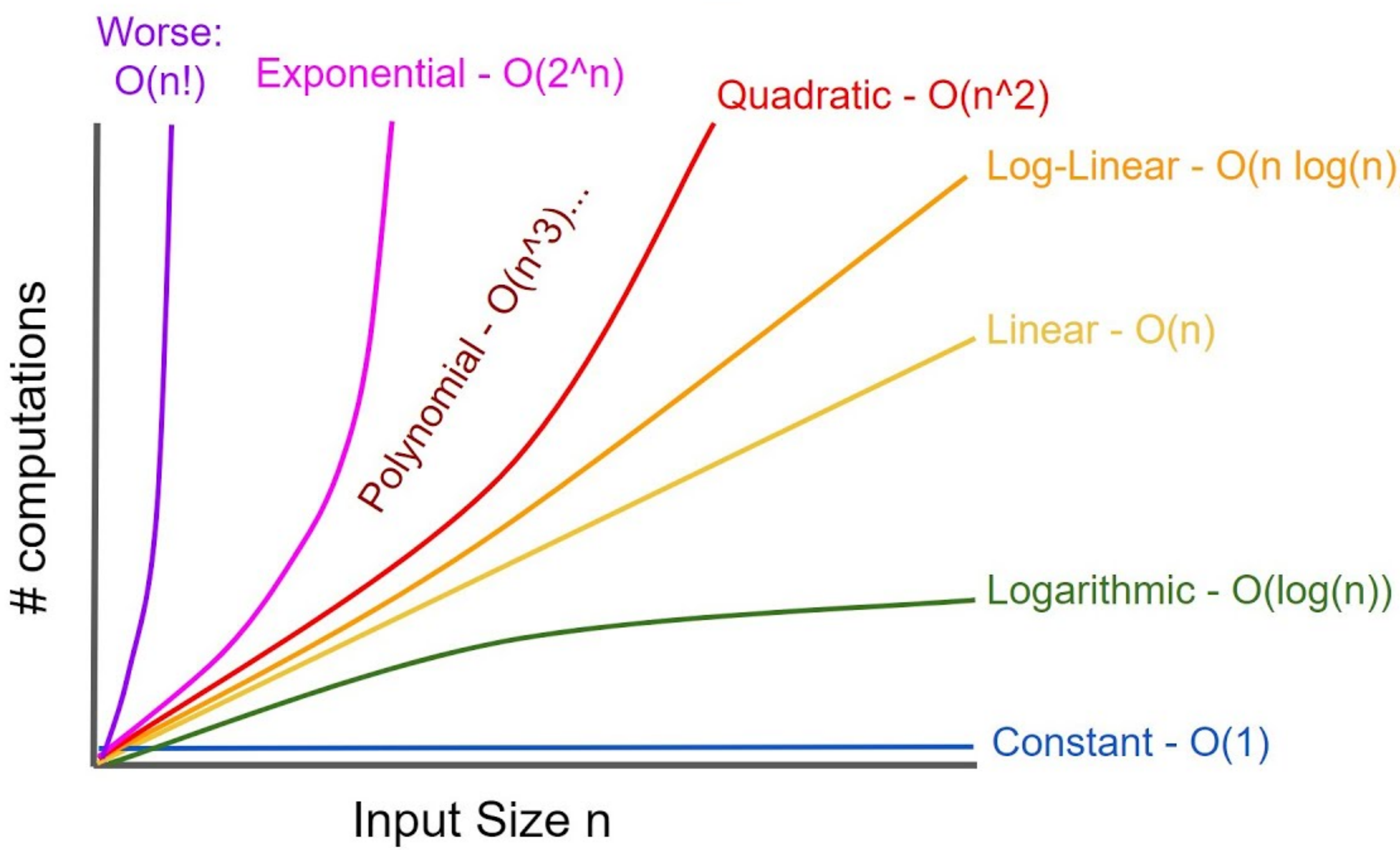
Big-O

- Big-O is the shorthand used to classify the time complexity of algorithms.
- It has a formal mathematical definition, but you just need to know how to classify algorithms into different **Big-O categories**.

O(1)	Constant time	
O(log n)	Logarithmic time	Runtime grows logarithmically in proportion to n.
O(n)	Linear time	It grows linearly as input size increases.
O(n log n)	Linearithmic time or log linear	
O(n^3)	Cubic time	
O(n^2)	Quadratic time	
Q(2^n)	Exponential time	
O(n!)	Factorial time	

Algorithms & Data Structures CS 211

Big-O categories



Algorithms & Data Structures CS 211

Primitive Operations

simple statement takes O(1) time.	int y= n + 25;	O(1)

Worst case O(n) if it in the loop, best case O(1)	if(n> 100)
	{
	…
	}else{
	..
	..
	}
		O(1)

For loop takes n time to complete	for(int i=0;i<n;i++)
	{
	..
	}
		O(n)

While loop takes n time	int i=0;
	while(i<n)
	{
	..
	i++;
	}
		O(n)

Algorithms & Data Structures CS 211

Primitive Operations

Loop takes n time and increases or decreases by a constant	`for(int i = 0; i < n; i+=5)`
	`    sum++;`
	
	`for(int i = n; i > 0; i-=5)`
	`    sum++;`
		O(n)

Loop takes n time and increases or decreases by a multiple	`for(int i = 1; i < =n; i*=2)`
	`    sum++;`
	
	`for(int i = n; i > 0; i/=2)`
	`    sum++;`
		O(log(n))

Nested loops contain size n and m	`for(int i=0;i<n;i++)`
	`{`
	`     for(int i=0;i<m;i++){`
	`     ..`
	`     ..`
	`     }`
	`}`
		O(nm)

Algorithms & Data Structures CS 211

Primitive Operations

First loop runs n times and the inner loop runs log(n) times or vice versa	`for(int i=0;i<n;i++)`
	`{`
	`    for(int j=1;i<=n;j*=4){`
	`    ..`
	`    ..`
	`    }`
	`}`
		O(n*log(n))

First loop runs n^2 times and the inner loop runs n times or vice versa	`for(int j=0;j<n*n;j++)`
	`{`
	`    for(int i=0;i<n;i++){`
	`    ..`
	`    ..`
	`    }`
	`}`
		O(n^3)

First loop runs n times and the inner loop runs n^2 times and the third loop runs n^2	`for(int i = 0; i < n; i++)`
	`    for( int j = 0; j < n * n; j++)`
	`        for(int k = 0; k < j; k++)`
	`            sum++;`
		O(n^5)