Asymptotic Notation
Complexity Analysis

Week-02, Lesson 2

ALGORITHM DEFINITION

A finite set of statements that guarantees an optimal solution in finite interval of time

GOOD ALGORITHMS?

Run in less time

Consume less memory

But computational resources (time complexity) is usually more important

MEASURING EFFICIENCY

The efficiency of an algorithm is a measure of the amount of resources consumed in solving a problem of size *n*.

- The resource we are most interested in is time
- We can use the same techniques to analyze the consumption of other resources, such as memory space.

It would seem that the most obvious way to measure the efficiency of an algorithm is to run it and measure how much processor time is needed

Is it correct ?

FACTORS

Hardware

Operating System

Compiler

Size of input

Nature of Input

Algorithm

Which should be improved?

RUNNING TIME OF AN ALGORITHM

Depends upon

Input Size

Nature of Input

Generally time grows with size of input, so running time of an algorithm is usually measured as function of input size.

Running time is measured in terms of number of steps/primitive operations performed

Independent from machine, OS

FINDING RUNNING TIME OF AN ALGORITHM / ANALYZING AN ALGORITHM

Running time is measured by number of steps/primitive operations performed

Steps means elementary operation like

,+, *,<, =, A[i] etc

We will measure number of steps taken in term of size of input

SIMPLE EXAMPLE (1)

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

int Sum(int A[], int N)
{
   int s=0;
   for (int i=0; i< N; i++)
      s = s + A[i];
   return s;
}
```

How should we analyse this?

SIMPLE EXAMPLE (2)

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

int Sum(int A[], int N){
    int s=0;          1

    for (int i=0; i< N; i++)
     2             3       4
      s = s + A[i];
    5        6     7
    return s;
                 8
}
```

1,2,8: Once

3,4,5,6,7: Once per each iteration of for loop, N iteration

Total: 5N + 3

The *complexity function* of the algorithm is : $f(N) = 5N + 3$

SIMPLE EXAMPLE (3) GROWTH OF 5N+3

Estimated running time for different values of N:

N = 10	=> 53 steps
N = 100	=> 503 steps
N = 1,000	=> 5003 steps
N = 1,000,000	=> 5,000,003 steps

As N grows, the number of steps grow in *linear* proportion to N for this function *"Sum"*

WHAT DOMINATES IN PREVIOUS EXAMPLE?

What about the +3 and 5 in 5N+3?

- As N gets large, the +3 becomes insignificant
- 5 is inaccurate, as different operations require varying amounts of time and also does not have any significant importance

What is fundamental is that the time is *linear* in N.

Asymptotic Complexity: As N gets large, concentrate on the highest order term:

- Drop lower order terms/ constant such as +3
- Drop the constant coefficient of the highest order term i.e. N

ASYMPTOTIC COMPLEXITY

The 5N+3 time bound is said to "grow asymptotically" like N

This gives us an approximation of the complexity of the algorithm

Ignores lots of (machine dependent) details, concentrate on the bigger picture

COMPARING FUNCTIONS: ASYMPTOTIC NOTATION

Big Oh Notation: Upper bound

Omega Notation: Lower bound

Theta Notation: Tighter bound

BIG OH NOTATION [1]

If f(N) and g(N) are two complexity functions, we say

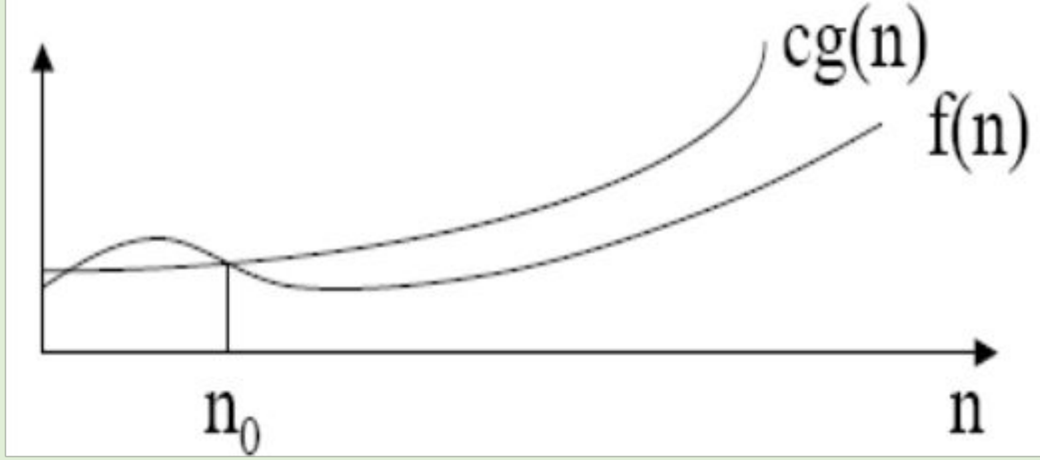
$$f(N) = O(g(N))$$

(read "f(N) is order g(N)", or "f(N) is big-O of g(N)")
if there are constants c and N such that for N > N ,

$$f(N) \leq c * g(N)$$

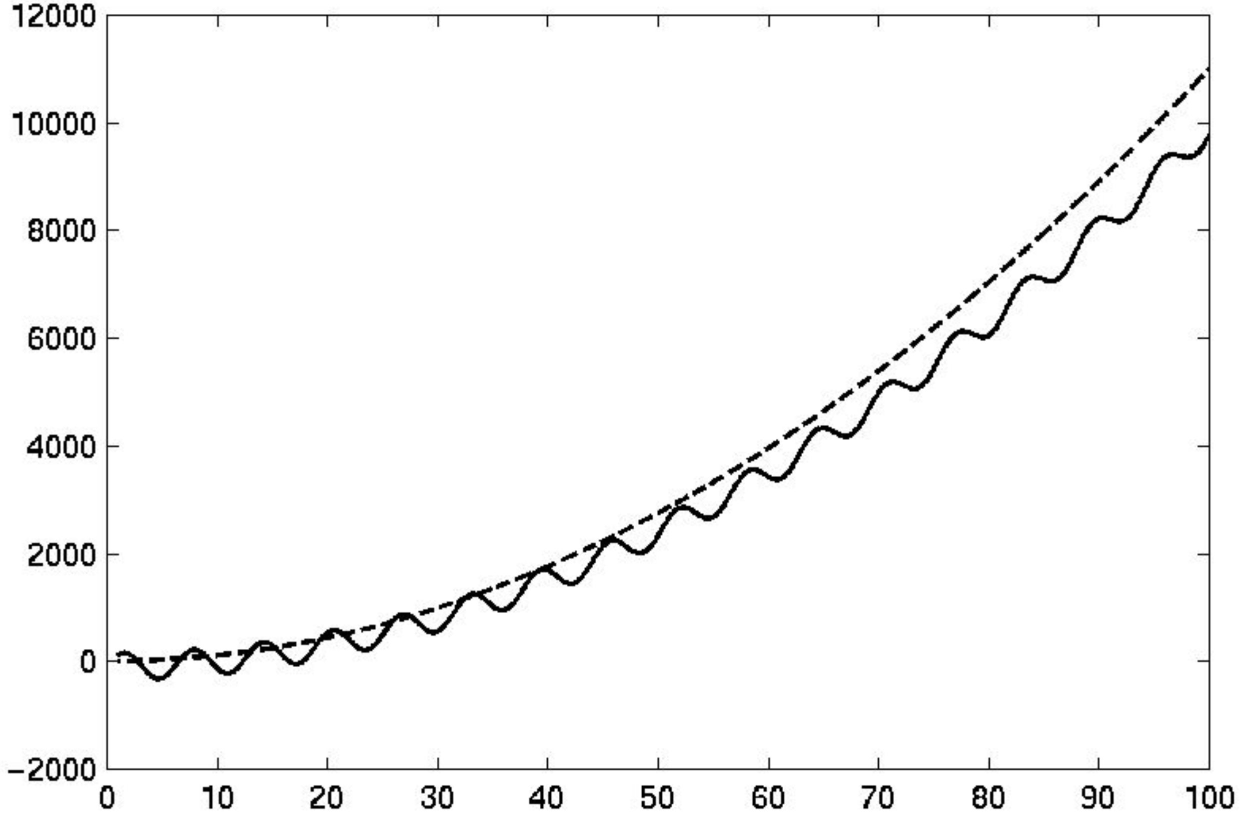
for all sufficiently large N.

BIG OH NOTATION [2]



- Function cg(n) always dominates f(n) to the right of n_0

O(F(N))



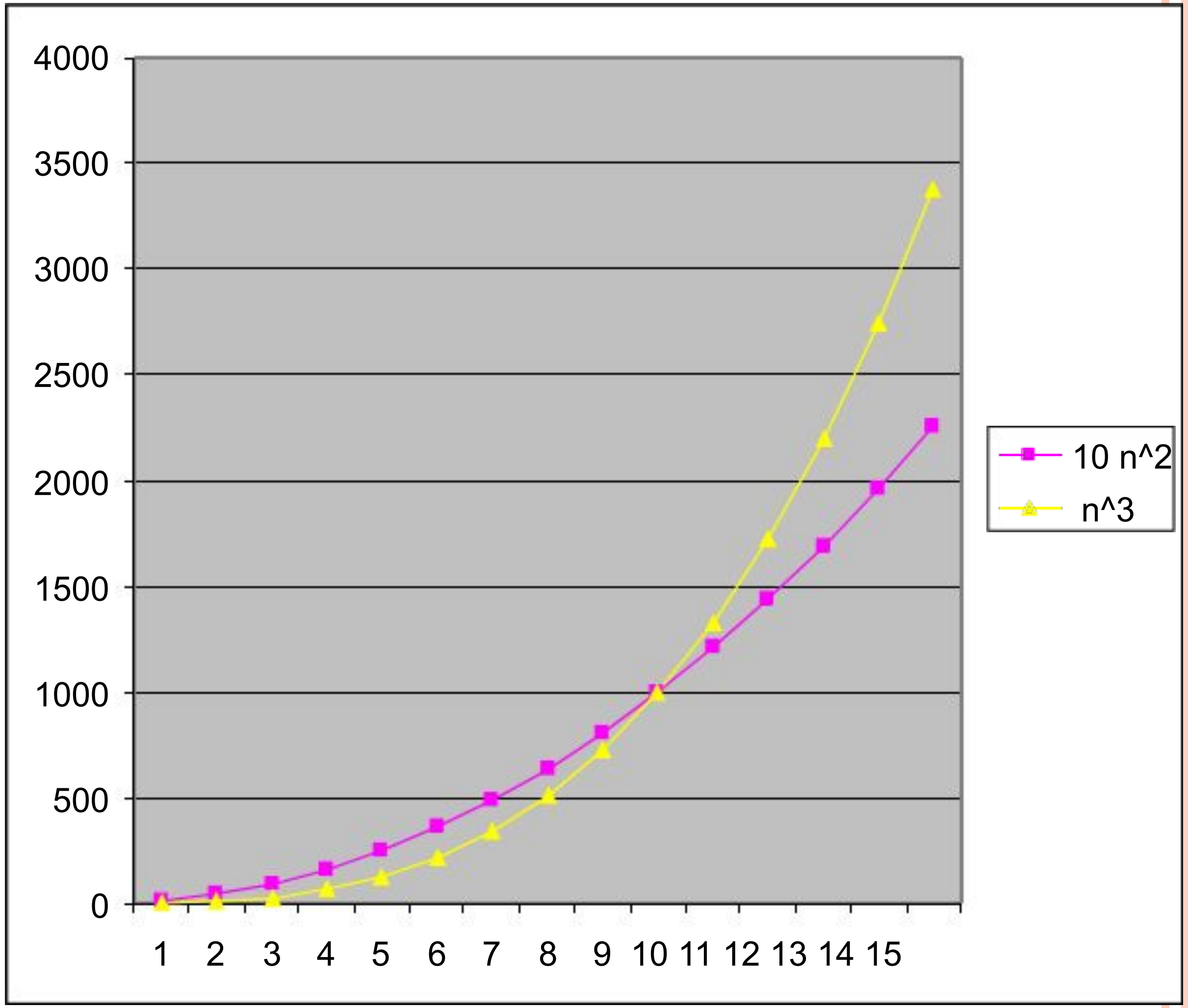
12000
10000
8000
6000
4000
2000
0
−2000
0
10
20
30
40
50
60
70
80
90
100

16

EXAMPLE (2): COMPARING FUNCTIONS

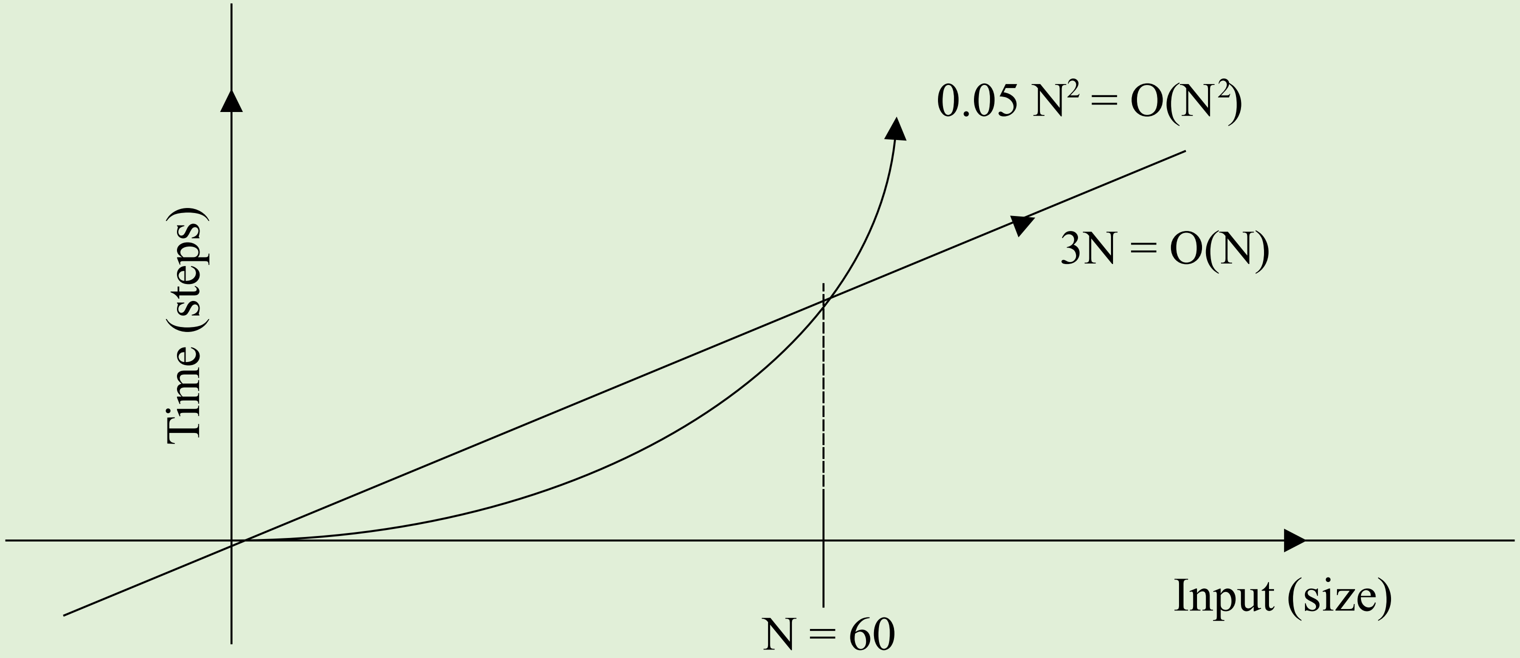
- Which function is better?

  $10\ n^2$ Vs n^3



COMPARING FUNCTIONS

As inputs get larger, any algorithm of a smaller order will be more efficient than an algorithm of a larger order



BIG-OH NOTATION

Even though it is correct to say "$7n - 3$ is $O(n^3)$", a better statement is "$7n - 3$ is $O(n)$", that is, one should make the approximation as tight as possible

Simple Rule:

Drop lower order terms and constant factors

$7n-3$ is $O(n)$

$8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

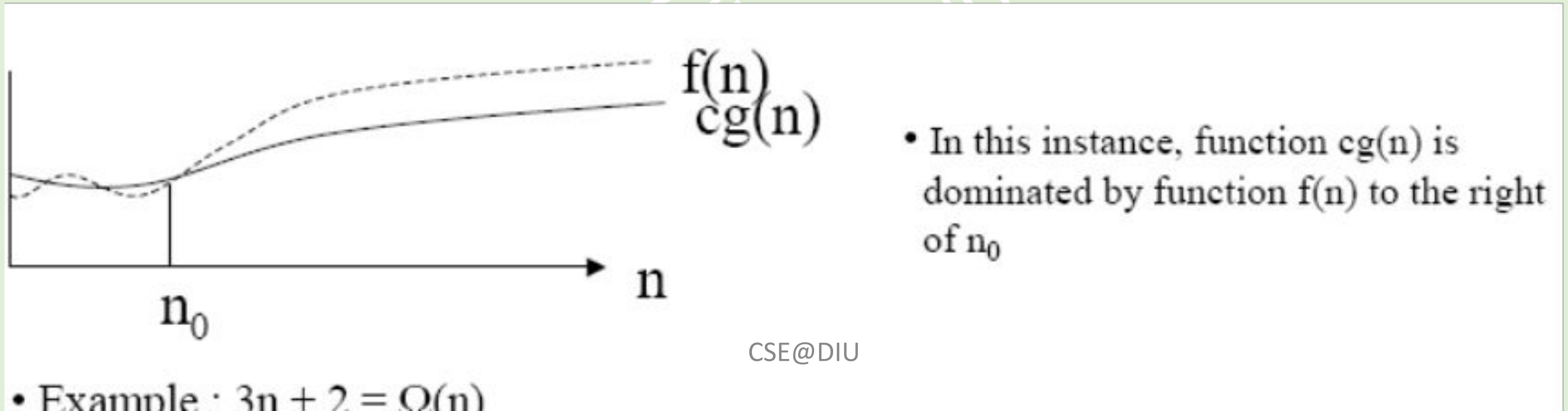
BIG OMEGA NOTATION

If we wanted to say "running time is at least…" we use Ω

Big Omega notation, Ω, is used to express the lower bounds on a function.

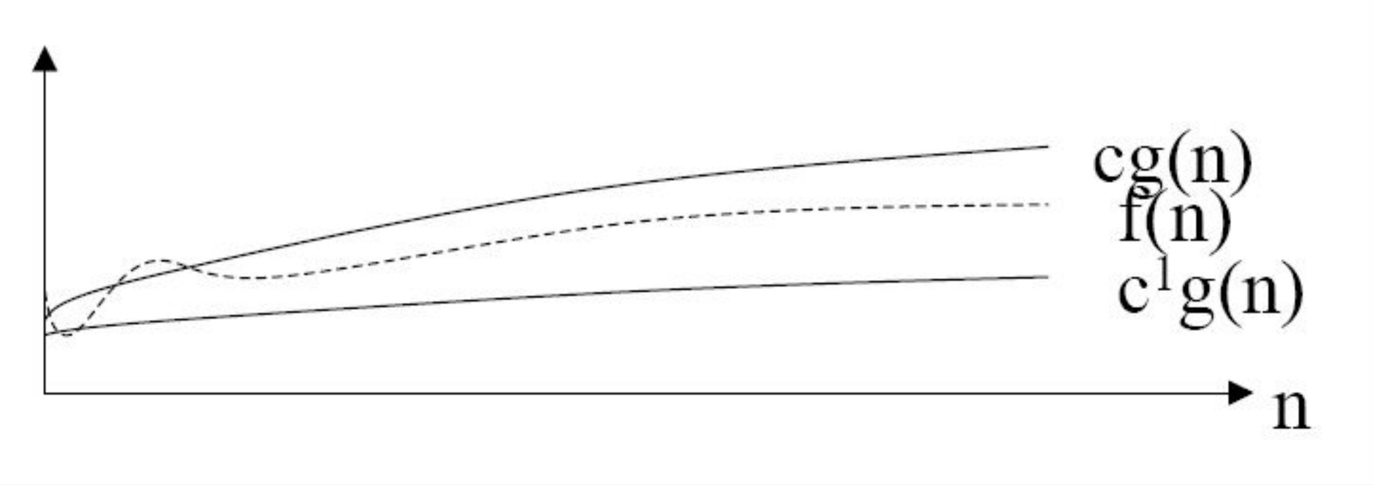
If f(n) and g(n) are two complexity functions then we can say:

f(n) is Ω(g(n)) if there exist positive numbers c and n_0 such that 0<=f(n)>=cΩ (n) for all n>=n_0



BIG THETA NOTATION

- If we wish to express tight bounds we use the theta notation, Θ
- $f(n) = \Theta(g(n))$ means that $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$



21

WHAT DOES THIS ALL MEAN?

If $f(n) = \Theta(g(n))$ we say that $f(n)$ and $g(n)$ grow at the same rate, asymptotically

If $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$, then we say that $f(n)$ is asymptotically slower growing than $g(n)$.

If $f(n) = \Omega(g(n))$ and $f(n) \neq O(g(n))$, then we say that $f(n)$ is asymptotically faster growing than $g(n)$.

WHICH NOTATION DO WE USE?

To express the efficiency of our algorithms which of the three notations should we use?

As computer scientist we generally like to express our algorithms as big O since we would like to know the upper bounds of our algorithms.

Why?

If we know the worse case then we can aim to improve it and/or avoid it.

PERFORMANCE CLASSIFICATION

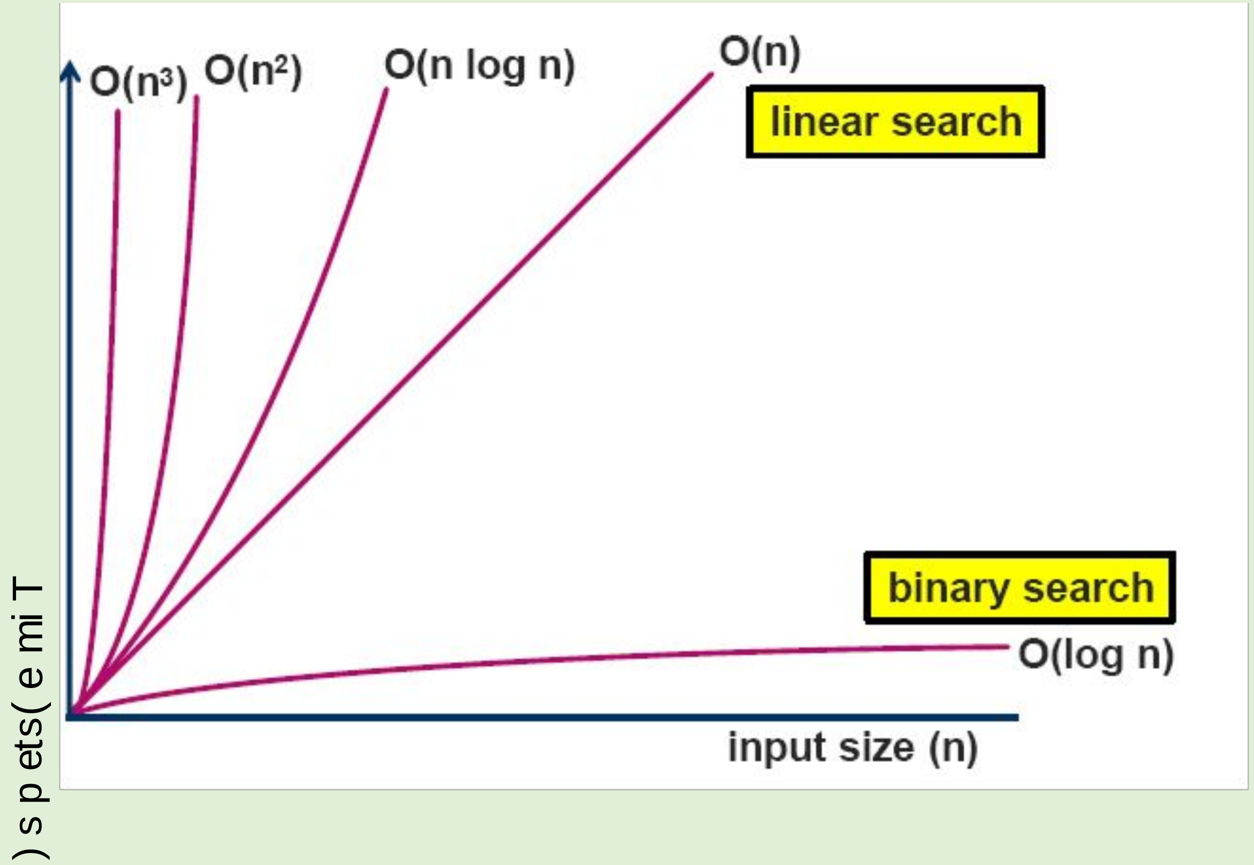
f(n)	Classification
1	*Constant*: run time is fixed, and does not depend upon n. Most instructions are executed once, or only a few times, regardless of the amount of information being processed
log n	*Logarithmic*: when *n* increases, so does run time, but much slower. Common in programs which solve large problems by transforming them into smaller problems.
n	*Linear*: run time varies directly with *n*. Typically, a small amount of processing is done on each element.
n log n	When *n* doubles, run time slightly more than doubles. Common in programs which break a problem down into smaller sub-problems, solves them independently, then combines solutions
n^2	*Quadratic*: when *n* doubles, runtime increases fourfold. Practical only for small problems; typically the program processes all pairs of input (e.g. in a double nested loop).
n^3	*Cubic*: when n doubles, runtime increases eightfold
2^n	*Exponential*: when n doubles, run time squares. This is often the result of a natural, "brute force" solution.

SIZE DOES MATTER[1]

What happens if we double the input size N?

N	$\log_2 N$	5N	$N \log_2 N$	N^2	2^N
8	3	40	24	64	256
16	4	80	64	256	65536
32	5	160	160	1024	$\sim 10^{9}$
64	6	320	384	4096	$\sim 10^{19}$
128	7	640	896	16384	$\sim 10^{38}$
256	8	1280	2048	65536	$\sim 10^{76}$

COMPLEXITY CLASSES



26

SIZE DOES MATTER[2]

Suppose a program has run time O(n!) and the run time for
n = 10 is 1 second

For n = 12, the run time is 2 minutes

For n = 14, the run time is 6 hours

For n = 16, the run time is 2 months

For n = 18, the run time is 50 years

For n = 20, the run time is 200 centuries

Textbooks & Web References

- Text Book (Chapter 3)
- Reference book ii (Chapter 2)

Thank you